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

DECSAI

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 2011-2012

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 las prácticas anteriores, se trabajará con una versión modificada del simulador software que implementa una aspiradora .Este simulador fue desarrollado inicialmente por el profesor Tsung-Che Chiang de la NTNU (Norwegian University of Science and Technology, Taiwan) .

La modificación del simulador para esta práctica consiste, a rasgos generales, en la extensión del 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. 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 de búsqueda MINIMAX, con profundidad limitada (con cota máxima 10), para dotar de comportamiento inteligente deliberativo a una aspiradora 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.

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 **Aspiradoras**, disponible en la web de la asignatura.

- Las bibliotecas adicionales **libopengl32.a**, **libglu32.a**, **libglut32.a**, disponibles en la web de la asignatura.
- Los mapas del mundo que representan tableros de juego de los agentes jugadores para validar su comportamiento, disponibles en la web de la asignatura.

La guía de instalación del software previamente mencionado puede consultarse en el guión de la práctica 1.

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 de las aspiradoras que se explica a continuación

El juego de las **ASPIRADORAS TRON** es un juego por turnos de dos jugadores (aspiradoras) capaces de moverse en un tablero de *nxn* casillas. El tablero puede tener **casillas con suciedad**, casillas que **representan un obstáculo** o casillas **vacías**. Cada jugador tiene conocimiento completo del estado del tablero, por tanto conoce en todo momento en qué posición se encuentra él mismo, dónde están los obstáculos en el tablero y qué casillas contienen suciedad. El objetivo del juego consiste en conseguir la mayor cantidad posible de unidades de basura aspirada, considerando que cada **casilla con suciedad puede contener varias unidades de basura**, y llevando a cabo movimientos que eviten los obstáculos

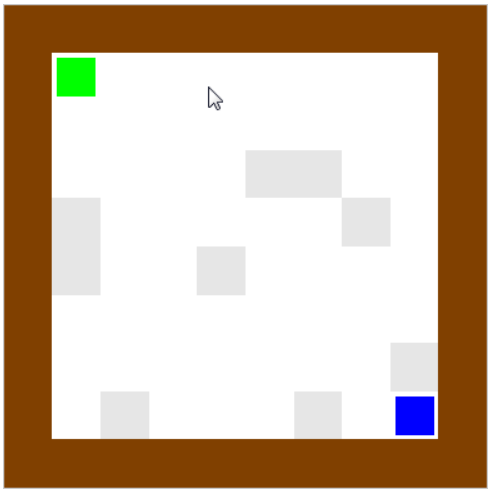


Figura 1. Tablero de 10x10 casillas en el que se muestran dos aspiradoras (jugadores) y 9 casillas con basura.

Al inicio del juego, el tablero contiene una cantidad inicial de casillas con basura, un número inicial de obstáculos (mayor o igual que 0) y cada jugador está situado en una casilla transitable del tablero.

Cada jugador puede llevar a cabo los siguientes movimientos:

- ***UP, DOWN, RIGHT, LEFT***: acciones para desplazarse desde la casilla actual (i,j) a la casilla superior (i-1,j), inferior (i+1,j), izquierda (i, j-1) y derecha (i, j+1), Es importante considerar que **los movimientos están sujetos a las siguientes restricciones:**
  - un movimiento siempre tiene que llevar a una casilla destino que no represente un obstáculo, que no no esté ocupada por el otro jugador y que pertenezca al tablero.
  - cada vez que un jugador abandona una casilla origen para llegar a una destino**, el entorno se modifica**, de manera que **aparece un nuevo obstáculo** en la casilla origen.
  - si la casilla destino contiene ***k*** unidades basura **el marcador del jugador** (es decir, la cantidad de basura total recogida por el jugador hasta ese momento) se incrementa automáticamente en ***k*** unidades. En definitiva, **la aspiración no se contempla como movimiento (acción) posible del jugador**, se produce de forma automática al pasar por una casilla con suciedad.

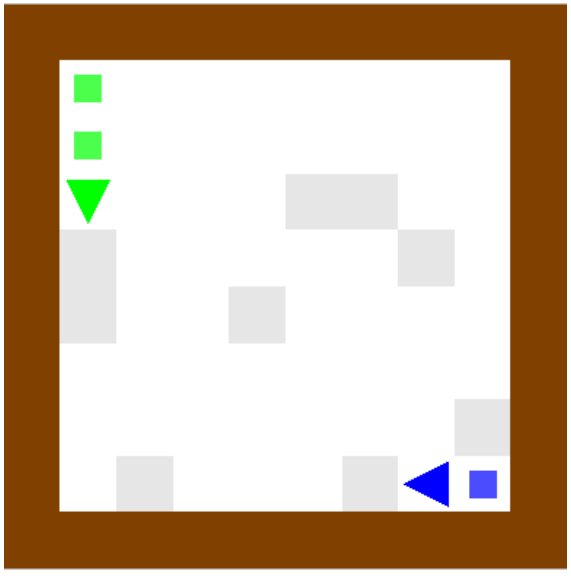


Figura 2(a). Un estado del tablero en el que el Jugador 1 (verde) ha avanzado 2 posiciones hacia abajo dejando un rastro de dos obstáculos (uno por cada casilla visitada). El Jugador 2 (azul) ha avanzado una posición hacia la izquierda. En ese estado el turno corresponde a Jugador 2.

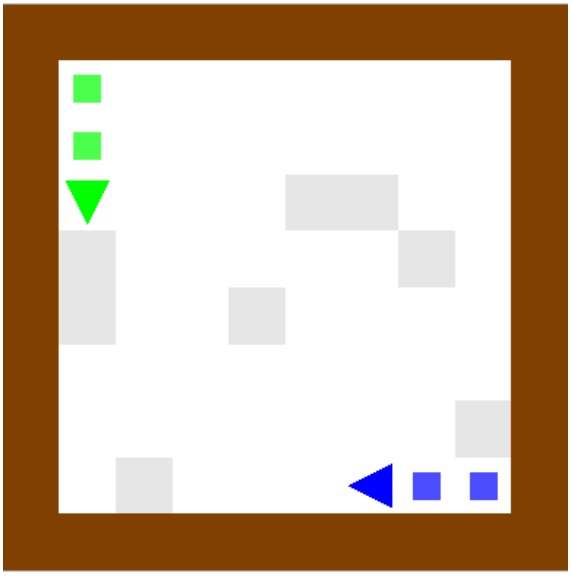


Figura 2(b). Un estado del tablero en el que el Jugador 2 ha avanzado una posición a la izquierda (desde el estado de la Figura 2(a). La basura de la casilla en la posición de destino desaparece.

El juego termina cuando se de alguna de las siguientes condiciones:

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.

En todos los casos, el ganador del juego es el jugador con el mayor número de unidades de basura aspirada. Se producirá un empate si la cantidad de basura de cada jugador es la misma al finalizar el juego.

OBJETIVO DE LA PRÁCTICA:

A partir de estas consideraciones iniciales, el objetivo de la práctica es 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.

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 **Aspiradoras** 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.

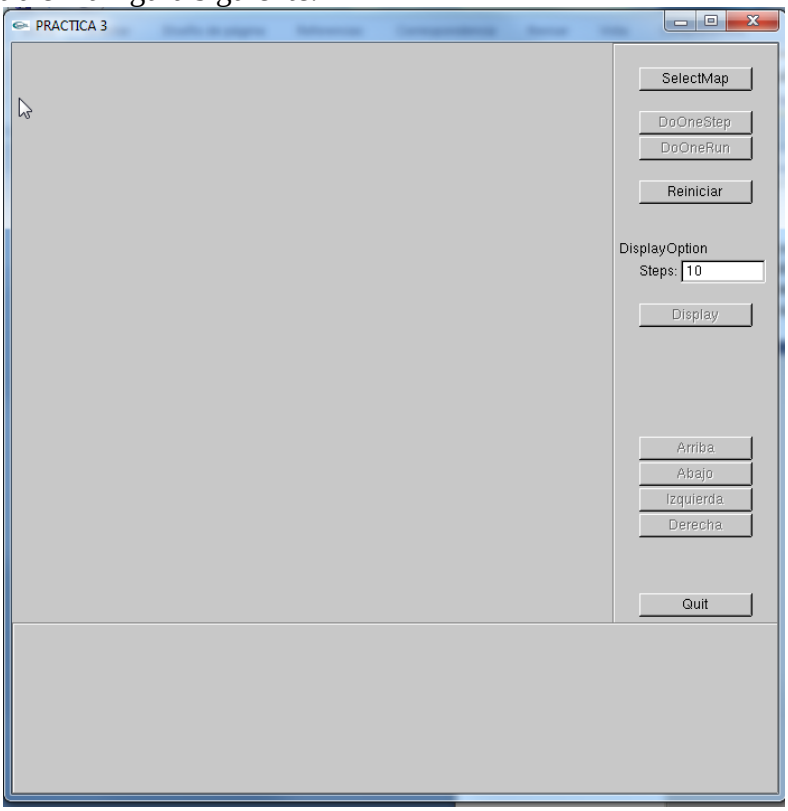
Para instalarlo, seguir estos pasos:

1. Descargue el fichero **Aspiradoras.rar** (o **Aspiradoras.zip**) desde la web de la asignatura, y cópielo 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 "**Aspiradoras.cbp"** en el entorno **CodeBlocks**.

4.2. Ejecución del simulador

Tomamos la opción de abrir un proyecto existente y elegimos el proyecto ***"aspiradoras.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 **Aspiradoras.exe** generado en la carpeta del proyecto tras su compilación). Aparecerá una ventana como la que se muestra en la figura siguiente.

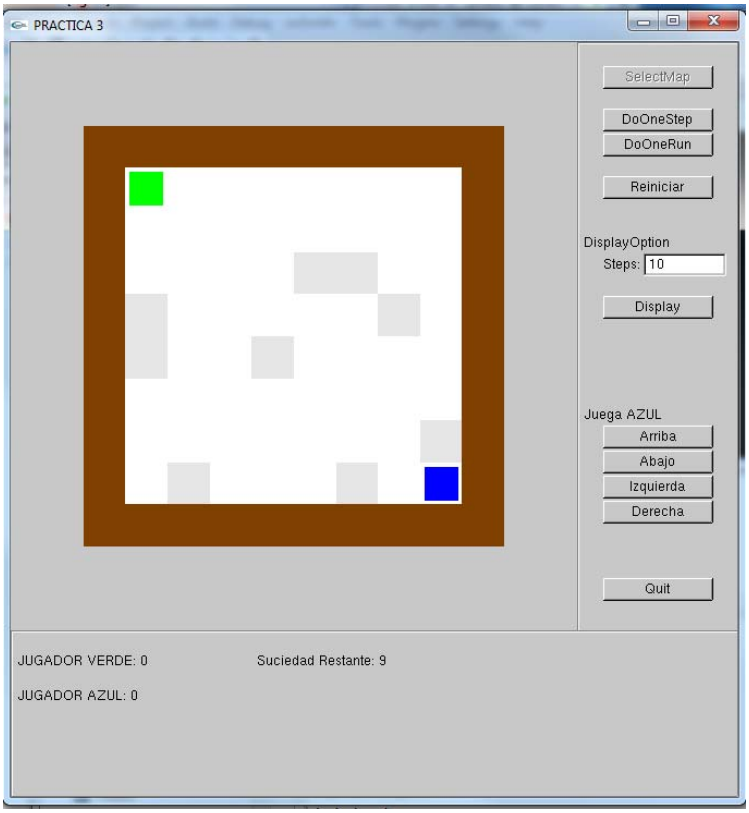


En esta ventana, la opción que nos interesa se encuentra en el botón "**Select Map**", que nos permite seleccionar un mapa de entorno que en este caso representa un tablero de juego.

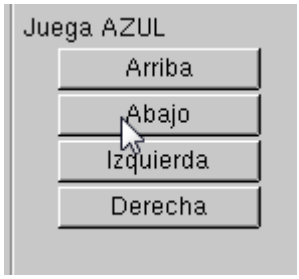
Una vez pulsado "**Select Map"** aparece una nueva ventana en la que además de seleccionar un tablero de juego, podemos especificar, para cada uno de los dos jugadores, si deseamos que sea Humano o Automático. El comportamiento seleccionado para cada jugador determinará el funcionamiento del juego. Por defecto, el simulador trae un único mapa "**mapa10a.map**".

Tras cargar el mapa "**mapa10a.map**", la vista del simulador cambia mostrando el tablero de juego con los dos jugadores visualizados como un recuadro Verde y otro Azul (en adelante Jugador Verde y Jugador

Azul). En este tablero el Jugador Verde está inicialmente situado en la esquina superior izquierda del tablero y el Jugador Azul en la esquina inferior derecha.. El tablero contiene también un conjunto de casillas con suciedad.
En la parte inferior de la ventana aparece información sobre la "Suciedad restante", (9 en este mapa), Marcador de suciedad aspirada por el Jugador Verde (inicialmente 0) y por el Jugador Azul (inicialmente 0). Recordar que una casilla con suciedad puede incluir varias unidades de basura.



La carga del mapa además activa los siguientes botones:



En la parte superior de la botonera se informa del jugador que posee el turno (la elección del que juega primero se realiza de forma aleatoria), esta información irá cambiando conforme vayan cambiando los turnos de los jugadores. Por cada jugador, existe la posibilidad de controlar sus movimientos mediante

los botones "**Arriba**", **Abajo**", "**Izquierda**" y "**Derecha**". Estos botones están siempre activos, con independencia de que el Jugador con el turno sea Humano o Máquina. No obstante, los botones sólo tienen efecto real cuando el jugador es Humano. En definitiva, el funcionamiento de estos botones es el siguiente:

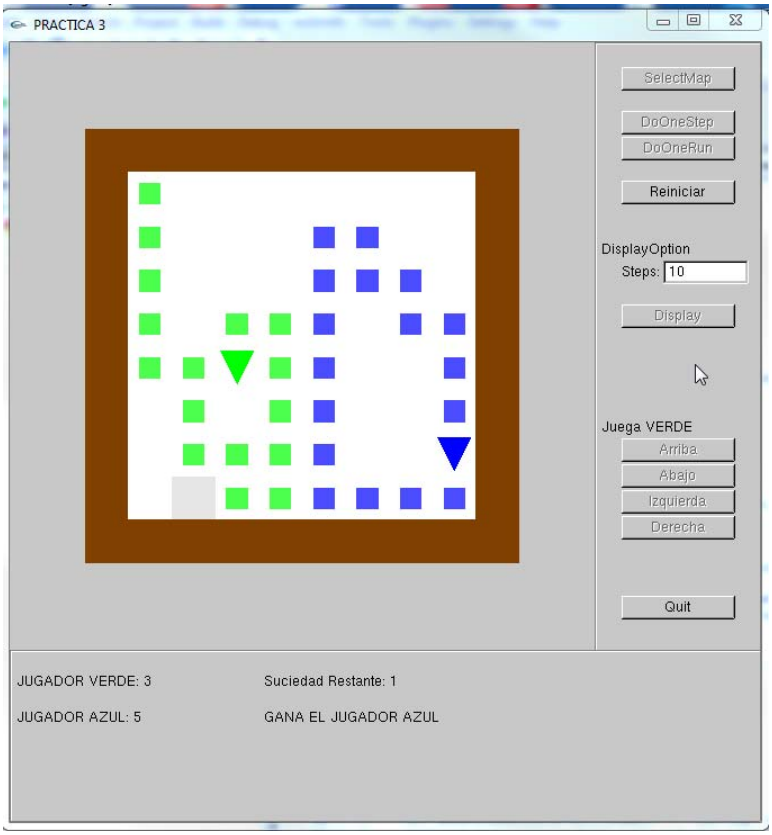
- Si el jugador con turno es Humano, cada botón permite mover el jugador en su correspondiente dirección.
- Si el jugador con turno es Máquina, la pulsación de un botón hará que el jugador lleve a cabo el movimiento que ha decidido internamente (con independencia del botón pulsado).

Una vez pulsado alguno de estos botones, el turno pasa al siguiente jugador.

El resto de botones tiene el siguiente comportamiento asociado:

- **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

Recordar que cada vez que un jugador se mueve a una casilla destino, la casilla origen es reemplazada con un obstáculo. Además, como se ha explicado antes, el juego termina bien cuando la cantidad de basura recogida por un jugador supera la suma de la recogida por su oponente más la restante en el tablero, o bien cuando un jugador queda bloqueado porque no puede llevar a cabo más movimientos.



Por ejemplo, esta figura corresponde a un estado final de juego en el que queda una casilla con suciedad y se da la condición:

Marcador(Jugador Azul) > Marcador (Jugador Verde) + Suciedad Restante.

Observar que cada jugador ha dejado una estela de obstáculos, indicando el camino que ha recorrido.

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.

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.

```
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
  ()][Position1_Y()] > 0:maze_[Position2_X()][Position2_Y()] > 0); }
33        bool isJustBump(int jugador) const { return (jugador==1?bump1_:bump2_); }
34        int  DirtAmount(int x, int y) const;
35        int  DirtAmount() const;
36        //int  RandomSeed() const { return randomSeed_; }
37        int  SizeMaze() const {return MAZE_SIZE;}
38        int **SeeMap() const;
39        int Position1_X() const {return agent1PosX_;}
40        int Position1_Y() const {return agent1PosY_;}
41        int Position2_X() const {return agent2PosX_;}
42        int Position2_Y() const {return agent2PosY_;}
43        int Marcador(int jug) const;
44        int Total_Suciedad() const;
45        void SaveMap() const;
46        int GenerateNextMove(Environment *V, int jug) const;
47        void possible_actions(bool *act, int jug) const;
48        int Last_Action(int jug) const;
49        bool operator==(const Environment & env) const;
50        bool a_movement_is_possible(int jug) const;
51
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    };
```

La clase Environment de esta práctica es una ampliación de la misma clase en las prácticas anteriores, e incorpora información esencial para representar el entorno del juego.. Es fundamental entender que la clase ***Environment*** **parte siempre de la asunción de que hay dos agentes, o jugadores**, en el entorno. **Los dos jugadores del entorno se declaran en el fichero GUI.cpp como dos variables, globales player1 y player2,** de la clase Player. En adelante nos referiremos a ellos por tanto como player1 y player2.
A continuación se describen las variables y métodos esenciales de esta clase, para la comprensión y la elaboración de la práctica:

Variables de la Clase Environment

static const int OBSTACLE = -1, OBSTACLE_VERDE=-2, OBSTACLE_AZUL=-3;

Constantes que representan obstáculos en el tablero (mapa del entorno), respectivamente, obstáculo inicial, obstáculo introducido por el jugador 1, obstáculo introducido por el jugador 2.

int MAZE_SIZE;

Tamaño del tablero

bool bump1_, bump2_;

Variables que representan si un jugador ha detectado un obstáculo (player1 y player2 respectivamente).

int agent1PosX_, agent1PosY_, agent2PosX_, agent2PosY_;

Coordenadas de las posiciones de los dos jugadores (la coordenada X representa desplazamientos arriba/abajo en el tablero, mientras que la coordenada Y representa desplazamiento izquierda/derecha.

int marcador1_, marcador2_, unidades_suciedad_;

Marcador de unidades de suciedad recogidas por cada jugador y cantidad de unidades de suciedad restantes.

int **maze_;

Mapa del tablero

ActionType preAction1_, preAction2_;

Acciones realizadas en el estado anterior del juego (respectivamente para player1 y player2).

Métodos Destacables de la Clase Environment

void PintaAspiradora(int jugador, int w, int h, int x, int y) const;

Dibuja la aspiradora *jugador* en la posición *x,y*

void Show(int,int) const;

Muestra en la ventana el tablero.

void AcceptAction(int jugador, ActionType accion);

Actualiza la información del entorno dependiendo del jugador con el turno y de la acción ejecutada por el jugador. La información del entorno actualizada consiste en

- La nueva posición del jugador, actualizada según la acción ejecutada

- Los nuevos obstáculos en el tablero: si el jugador es el Jugador1 aparece un obstáculo verde en la casilla origen del movimiento, si es el Jugador2 aparece un obstáculo azul.
- Incremento del marcador del jugador, en el caso de que la casilla destino de movimiento contenga suciedad, con el número de unidades de basura que contenga la casilla.

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.

int Position1_X()

Devuelve la posición X (fila del tablero) del jugador 1.

Análogo para las funciones:

int Position1_Y() const {return agent1PosY_;}

int Position2_X() const {return agent2PosX_;}

int Position2_Y() const {return agent2PosY_;}

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.

*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 a[i], 0<= i <= 3, representa si jug puede llevar a cabo, respectivamente, la acción UP, DOWN, LEFT o RIGHT.

*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.

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.

5.2. Representación de los jugadores (Clase Player)

La clase Player se utiliza para representar un jugador (es la clase equivalente a Agent en las anteriores prácticas).

```
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.

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.

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. La fecha límite de la entrega de la memoria será comunicada con suficiente antelación por el profesor de prácticas en clase, y publicada en la página web de la asignatura.
- Se realizará una defensa de la práctica. La fecha de dicha defensa se publicará con suficiente antelación en la web de la asignatura para cada grupo/alumno, y será comunicada también en clase de prácticas por el profesor. 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 3.**
    - 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 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.

- La **fecha de entrega de la memoria práctica** será comunicada con la antelación suficiente mediante comunicado del profesor en el laboratorio de prácticas y en la web de la asignatura.

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 en los términos en que se ha explicado previamente. Estos ficheros deberán entregarse mediante la plataforma web de la asignatura, en un fichero ZIP que NO contenga carpetas separadas, es decir, todos los ficheros aparecerán en la carpeta donde se descomprima el fichero ZIP. **No se evaluarán aquellas prácticas que contengan ficheros ejecutables o virus.**

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

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

IMPORTANTE:

Ya que parte de la evaluación de la práctica consiste en un torneo entre los alumnos, es necesario distinguir de forma unívoca la clase Player que desarrolla cada uno. Así, que **antes de subir la práctica a la plataforma para su evaluación,** es necesario incluir versiones personalizadas de vuestros ficheros "player.cpp" y "player.h" de la siguiente forma:

Supongamos que el documento de identificación del alumno (DNI, NIF, pasaporte,…) es 29874712-V, entonces hay que hacer lo siguiente:

1. Hacer una copia de los ficheros "player.cpp" y "player.h" y **renombrar** las copias como "player29874712.cpp" y "player29874712.h" respectivamente, en el mismo directorio que el resto de ficheros fuente..
2. **Abrir el nuevo "player29874712.cpp**" y en este fichero:
    a. **Sustituir en la línea 3** #include "player.h" por #include "player29874712.h"
    b. En el menú de CodeBlocks, tomar la opción "Search", seleccionar "replace" y **reemplazar en todo el fichero la palabra "Player" por la palabra "Player29874712"** (ambas palabras sin meterlas entre dobles comillas y ambas empezando la primera letra en mayúscula).
    c. **Salvar** el fichero "player29874712.cpp"
3. **Abrir el nuevo fichero "player29874712.h**"y en este fichero,
    a. **Aplicar el mismo reemplazo** que en el fichero "player29874712.cpp" y salvar el fichero.
4. Abrir el fichero "GUI.cpp" y en este fichero hacer lo siguiente:
    a. **Sustituir** en línea 11 #include "player.h" por #include "player29874712.h"
    b. **Buscar** las dos siguientes sentencias:

Player *player1=NULL;

Player *player2=NULL;

y **sustituirlas** por

Player29874712 *player1=NULL;

Player29874712 *player2=NULL;

c. **Salvar** el fichero “GUI.cpp” con el nuevo nombre “GUI29874712.cpp”

5. **Crear un nuevo proyecto “aspiradoras2.cbp**”, para ello hacer los siguiente:
   a. 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”.
   b. Cliquear sobre el menú superior la opción “Project”, seleccionar “Add files…” y marcar los archivos “player29874712.cpp”, “player29874712.h” y “GUI29874712.cpp” y pulsar el botón “Abrir”.
   c. Por último, pulsar sobre el menú superior la opción “File”, seleccionar “Save Project as…” y nombrar al nuevo proyecto como “aspiradoras2.cbp”.

Así, en el fichero comprimido que hay que entregar deben aparecer los siguientes ficheros fuente:

- “**player.cpp**” y “**player.h**”
- “**player29874712,cpp**”, “**player29874712.h**” y “**GUI29874712.cpp**”
- El fichero de proyecto CodeBlocks **aspiradoras2.cbp**