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 1

Agentes Reactivos

DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN E INTELIGENCIA ARTIFICIAL
UNIVERSIDAD DE GRANADA
Curso 2011-2012

1. Introducción

1.1. Motivación

La primera práctica de la asignatura ***Inteligencia Artificial*** consiste en el diseño e implementación de agentes reactivos, capaces de percibir el ambiente y actuar de acuerdo a unas reglas simples predefinidas. Se trabajará con un simulador software. Para ello, se proporciona al alumno un entorno de programación, junto con el software necesario para simular el entorno. Es esta práctica, se diseñará e implementará una aspiradora inteligente basada en los ejemplos del libro *Stuart Russell, Peter Norvig, "Inteligencia Artificial: Un enfoque Moderno", Prentice Hall, Segunda Edición, 2004*. El simulador que utilizaremos fue desarrollado por el profesor Tsung-Che Chiang de la NTNU (Norwegian University of Science and Technology, Taiwan).

Las aspiradoras inteligentes son robots de uso doméstico de un coste aproximado entre 100 y 300 euros, que disponen de sensores de suciedad, un aspirador y motores para moverse por el espacio (ver Figura 1). Cuando unas aspiradora inteligente se encuentra en funcionamiento, esta recorre toda la dependencia o habitación donde se encuentra, detectando y succionando suciedad hasta que, o bien ha terminado de recorrer la dependencia, o bien se le aplica algún otro criterio de parada (batería baja, tiempo límite, etc.). Algunos enlaces que muestran el uso de este tipo de robots son los siguientes:

- http://www.youtube.com/watch?v=C1mVaje_BUM
- http://www.youtube.com/watch?v=dJSc_EKfTsw

Figura 1: Aspiradora inteligente

Este tipo de robots es un ejemplo comercial más de máquinas que implementan técnicas de Inteligencia Artificial y, más concretamente, mediante la Teoría de Agentes. En su versión más simple

(y también más barata), una aspiradora inteligente presenta un comportamiento reactivo puro: Busca suciedad, la limpia, se mueve, detecta suciedad, la limpia, se mueve, y continúa con este ciclo hasta que se da una cierta condición de parada. Otras versiones más sofisticadas permiten al robot *recordar* la dependencia a limpiar mediante el uso de representaciones icónicas como mapas, lo cual hace que el aparato ahorre energía y sea más eficiente en su trabajo. Finalmente, las aspiradoras más elaboradas (y más caras) pueden, además de todo lo anterior, realizar un plan de trabajo de modo que se pueda limpiar la suciedad en el menor tiempo posible y de la forma más eficiente. Son capaces de detectar su nivel de batería y volver automáticamente al cargador cuando esta se encuentra a un nivel bajo. Estas últimas pueden ser catalogadas como *agentes deliberativos* (se estudiarán en la segunda parte del tema 2 (búsqueda) y en el tema 3 de la asignatura, *Búsqueda heurística: métodos locales y de búsqueda por el mejor nodo*).

En esta primera práctica, centraremos nuestros esfuerzos en implementar el comportamiento de este tipo de artefactos asumiendo un comportamiento reactivo. Utilizaremos las técnicas estudiadas en el tema 2 de la asignatura para el diseño de agentes reactivos.

1.2. Cómo funciona una aspiradora inteligente

Explicaremos, de una forma muy simplificada, cuál es el funcionamiento interno de un robot aspiradora. La aspiradora (agente) vive en un mundo modelado mediante cuadrículas. Entre sus acciones, tiene la posibilidad de aspirar la suciedad, moverse hacia la izquierda, derecha, arriba o abajo. Adicionalmente, dispone de un sensor de choque que se activa cuando la aspiradora encuentra un obstáculo durante su movimiento hacia la casilla donde pretendía moverse. En cada momento, la aspiradora selecciona una acción entre las disponibles, atendiendo a la función interna de selección de acciones que se ha implementado en el agente. Las Figuras 2, 3 y 4 muestran un ejemplo de las acciones de la aspiradora a diseñar según el simulador que utilizaremos en la práctica. En la Figura 2 se observa, con tonos de gris, el grado de suciedad de cada casilla del mundo. La aspiradora ha realizado un movimiento hacia abajo desde la casilla inmediatamente superior (flecha verde apuntando hacia abajo). Por otra parte, en la figura 3 se observa que, desde la posición inicial de la aspiradora, ésta seleccionó la acción de moverse hacia arriba, hecho que activó el sensor de choque al encontrar un obstáculo (flecha de color rojo). Por tanto, la aspiradora no pudo hacer tal movimiento. Por último, la Figura 4 muestra que la aspiradora detectó suciedad en la casilla donde se encuentra y seleccionó la acción de limpiar (recuadro amarillo).

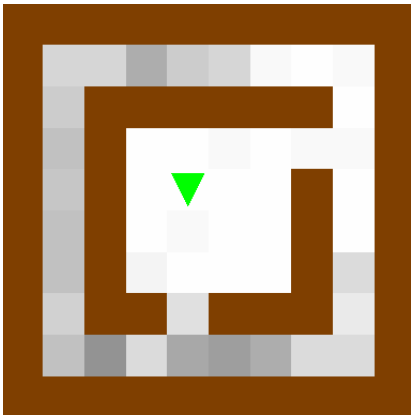

Figura 2: Movimiento hacia debajo de la aspiradora desde la casilla superior

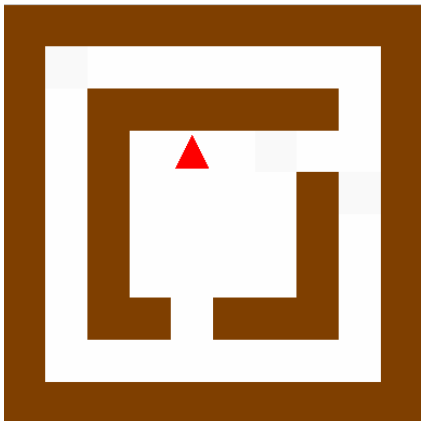

Figura 3: Movimiento hacia arriba de la aspiradora. Choque con obstáculo

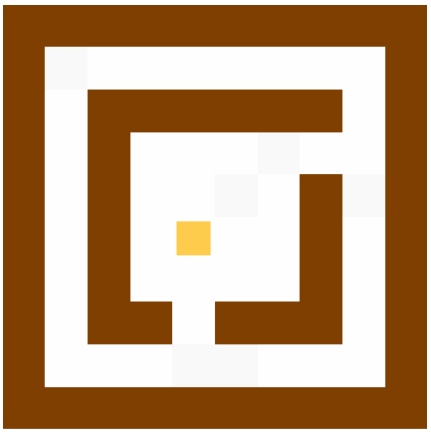

Figura 4: Movimiento de succión de suciedad en una casilla

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 **Agent-GUI-R3a**, 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 del agente para validar su comportamiento, disponibles en la web de la asignatura.

Se proporciona una guía de instalación del software previamente mencionado en la sección 4 de este guión de prácticas.

3. Objetivo de la práctica

La práctica tiene como objetivo diseñar e implementar un agente reactivo que resuelva el problema del robot aspirador de la mejor forma posible. Asumimos, como complejidad adicional, que cada movimiento del agente cuesta una cantidad de energía determinada y que el entorno es no determinista (una casilla que se haya limpiado puede volver a ensuciarse). Para ello, haremos las siguientes suposiciones:

- El entorno puede representarse en una matriz de tamaño máximo 10x10, donde los bordes de la matriz sólo pueden ser paredes de la habitación (es decir, asumimos que la habitación está cerrada y no tiene salida).
- Cada casilla de la matriz está o bien vacía, o bien conteniendo un obstáculo. Si está vacía, la casilla puede contener suciedad.

- La geografía de la habitación es desconocida a priori por el agente.
- La suciedad de cada casilla se mide en números enteros no negativos (0, 1, 2, 3, …).
- En cada instante de tiempo, cada casilla tiene la probabilidad de ensuciarse 1 unidad de suciedad más de lo que esté con probabilidad ***P***. En esta práctica, se impondrá una probabilidad ***P=0.01***.
- El agente no puede atravesar los obstáculos.
- La energía consumida por el agente se mide en números enteros no negativos (0, 1, 2, 3, …).
- El tiempo es discreto (no continuo) para el agente. Así, hablaremos de **instantes de tiempo** t=1, 2, 3…, etc., simplificando el problema de esta forma.
- En cada instante de tiempo, el agente sólo puede realizar una única acción.
- En caso de realizar un movimiento para cambiar de posición en el entorno, el agente sólo puede moverse una casilla (arriba, abajo, izquierda o derecha) en un instante de tiempo. Cada uno de estos movimientos consume 1 punto de energía. También puede decidir no realizar ningún movimiento, con un consumo de energía de 0 puntos.
- El agente puede limpiar la casilla en la que se encuentra, reduciendo en 1 la suciedad de dicha casilla con un coste de 2 puntos de energía. Si la suciedad de la casilla antes de limpiarse fuese 0, este valor no se vería reducido tras la acción de limpieza del agente.

El comportamiento del robot deberá intentar optimizar las siguientes medidas:

- **Máxima limpieza**. El comportamiento óptimo del agente permitirá minimizar la suma de las suciedades al cuadrado de todas las casillas sin obstáculos. Si asumimos que la suciedad se representa con la matriz $\boldsymbol{M_{10x10xT}}$, donde ***M(i,j,t)*** es la cantidad de suciedad de la casilla ***(i,j)*** en el instante de tiempo ***t***, y asumiendo que las casillas con obstáculo no tienen suciedad, entonces esta medida se calcula como:

$$S = \sum_{t=0}^{T} \sum_{i=1}^{10} \sum_{j=1}^{10} (M(i, j, t))^2$$

- **Mínimo consumo.** El agente deberá realizar las acciones de modo que tenga el mínimo consumo de energía eléctrica posible.

La suma de las dos medidas anteriores dará como resultado la eficacia total del agente.

3.1. ¿Qué métodos de diseño se pueden utilizar?

La elección del tipo de agente y el diseño para realizar la selección de acciones sólo está restringida a que sea uno de los estudiados en clase de teoría, en los temas 1 y 2 del programa de la asignatura, o una variación del mismo diseñada o consultada por el alumno al profesor. En este contexto, el alumno es libre de seleccionar cualquier técnica o técnicas a implementar para resolver el problema (funciones lógicas, redes lógicas, sistemas de producción, arquitectura de pizarra, etc.). En concreto, las tareas a realizar deberán incluir:

- **Análisis del problema** (análisis del entorno, dificultades, sensores del agente, actuadores del agente, restricciones, requisitos, etc.)
- **Estructura interna del agente:** Estructura detallada de los módulos que componen el agente y el funcionamiento interno del mismo. Justificación del diseño del agente seleccionado.
- **Diseño de la función de selección de acciones:** Tipo de función de selección de acciones elaborada (funciones lógicas, redes lógicas, sistemas de producción, arquitectura de pizarra, etc.), justificando su elección en función de sus ventajas e inconvenientes. Explicación (con un ejemplo simple) de su funcionamiento.
- **Implementación** del agente en el simulador.

Para mayor detalle, consulte la sección 6 de este documento: *Evaluación y entrega de prácticas*.

4. Software

4.1. Instalación de CodeBlocks

CodeBlocks v10.05 es el entorno de programación en C++ que utilizaremos para realizar la práctica. Se puede descargar directamente de forma gratuita desde la web oficial **http://www.codeblocks.org/** o desde la web de la asignatura. Aquellos alumnos que deseen un entorno alternativo, también pueden utilizar el entorno **Dev-C++**. Aunque este documento no da soporte para su uso, se ha comprobado que el simulador compila y funciona correctamente utilizando también este software.

Para la instalación de **CodeBlocks v10.05 para Windows**, siga los siguientes pasos:

1. Descargue el fichero **codeblocks-10.05mingw-setup.exe** desde alguna de las webs proporcionadas anteriormente.
2. Haga doble clic con el ratón sobre el fichero que acaba de descargar. Si Windows le indica que está intentando ejecutar un programa, acepte su ejecución.
3. La primera ventana que aparece en la bienvenida al programa de instalación (Figura 5). Haga clic en el botón **next** para continuar.

Figura 5: Benvenida al instalador de CodeBlocks

4. En la siguiente ventana, se debe aceptar el acuerdo de licencia haciendo clic sobre el botón **I Agree** (Figura 6).

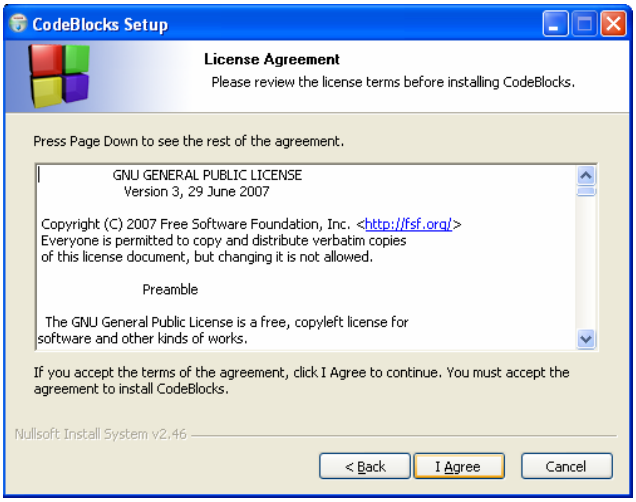

Figura 6: Aceptación del acuerdo de licencia GNU

5. La siguiente ventana permite seleccionar el tipo de instalación. Seleccione **la instalación completa: "Full: All plugins, all tools, just everithing"** (Figura 7).

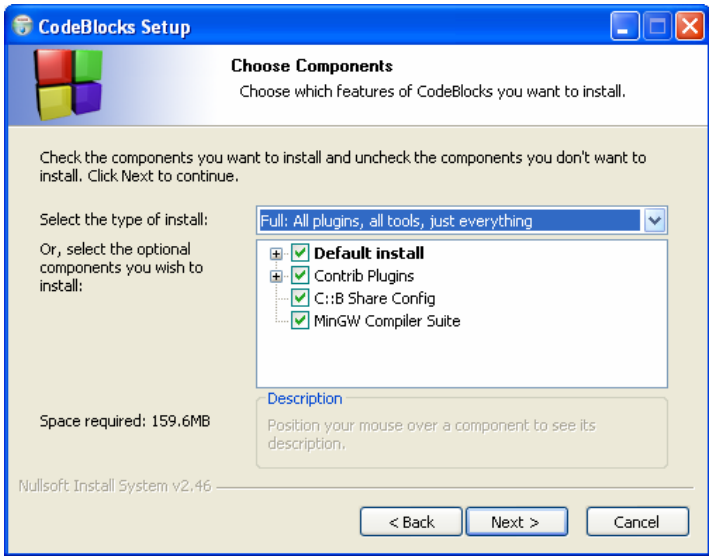

Figura 7: Selección de instalación completa

6. A continuación, deberemos seleccionar la carpeta donde deseamos instalar **CodeBlocks** pulsando sobre el botón **"Browse"** (en la Figura 8, por ejemplo, se seleccionó "**C:\facultad\CodeBlocks**"). Una vez hecho esto, pulsaremos sobre el botón **Install** para iniciar la instalación.

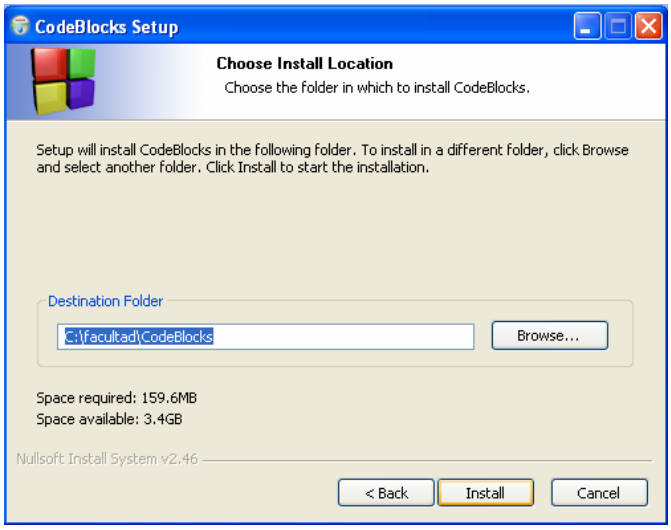

Figura 8: Selección de carpeta destino e inicio de instalación

7. Finalmente, el programa se encuentra instalado en la carpeta seleccionada. Antes de utilizarlo por primera vez, instalaremos el software de simulación de la aspiradora inteligente.

4.2. Instalación del simulador Agent-GUI-R3a

El simulador **Agent-GUI-R3a** nos permitirá implementar el comportamiento del agente y visualizar las acciones en una interfaz de usuario mediante ventanas. Para instalarlo, siga los siguientes pasos:

1. Descargue el fichero **Agent-GUI-R3a.rar** 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 "**C:\facultad\IA\practica1**".
2. Desempaquete el fichero en una subcarpeta de nombre **Agent-GUI-R3a**. Esta subcarpeta, tras desempaquetar el fichero anterior, contendrá un conjunto de archivos y las subcarpetas "**include**", "**lib**" y "**map**".
3. Descargue las bibliotecas auxiliares de la práctica 1 desde la web de la asignatura (fichero "**BibliotecasP1.rar**"). Extraiga los ficheros en la subcarpeta "**lib**" donde instaló el simulador (Por ejemplo, para la carpeta personal descrita en el paso 1: Extraer en "**C:\facultad\IA\practica1\Agent-GUI-R3a\lib"**).
4. Ya está instalado el simulador. A continuación, el siguiente paso es hacer la puesta a punto del entorno **CodeBlocks** para poder compilar el proyecto.

4.3. Compilación del simulador en CodeBlocks

Antes de poder compilar el proyecto del simulador, es necesario realizar ciertos ajustes para incluir las bibliotecas adicionales de la práctica. Para ello, siga los siguientes pasos:

1. Inicie **CodeBlocks** por primera vez. En esta primera ejecución, **CodeBlocks** nos pide que seleccionemos un compilador de C/C++ por defecto (Figura 9). Sólo lo pedirá esta vez, ya que esta información quedará registrada en su configuración interna. Seleccionaremos el compilador **GNU GCC Compiler** (la primera opción) y pulsaremos el botón **ok**.

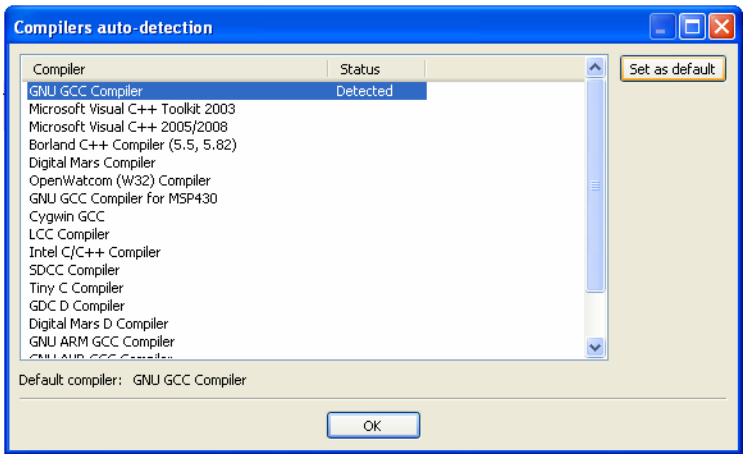

Figura 9: Selección del compilador por defecto de CodeBlocks

2. En esta primera ejecución de **CodeBlocks**, también se nos pregunta si deseamos que se asocien las extensiones de ficheros de programación en C/C++ a este entorno de programación. Si es lo que deseamos, pulsaremos sobre la opción "**Yes, associate Code::Blocks with every supported type (including project files from other IDEs)**", y pulsaremos sobre el botón **ok** (Figura 10). **CodeBlocks** tampoco volverá a hacer esta consulta en futuras ejecuciones.

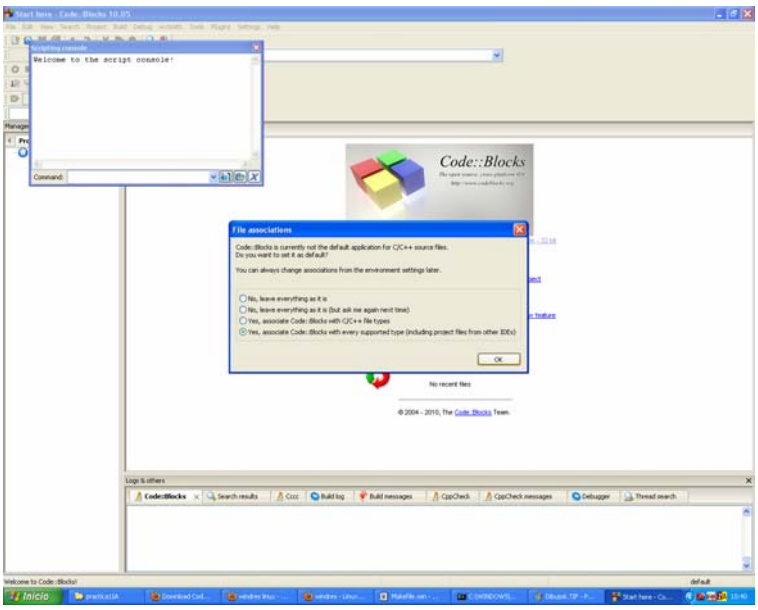

Figura 10: Asociación de extensiones de ficheros de programación en C/C++

3. Para abrir el **proyecto del simulador Agent-GUI-R3a**, pulsaremos sobre la opción "**Open an existing project"** de la ventana principal del entorno **CodeBlocks** (Figura 11). Aparecerá una nueva ventana para que viajemos a la carpeta donde hemos instalado el simulador y lo seleccionemos. Para ello, deberemos escoger el tipo de fichero "**Bloodshed Dev-C++ project**

files”, y seleccionar el nombre de fichero “**Agent.dev”**, tal y como muestra la Figura 12, y pulsaremos en el botón **Abrir**.

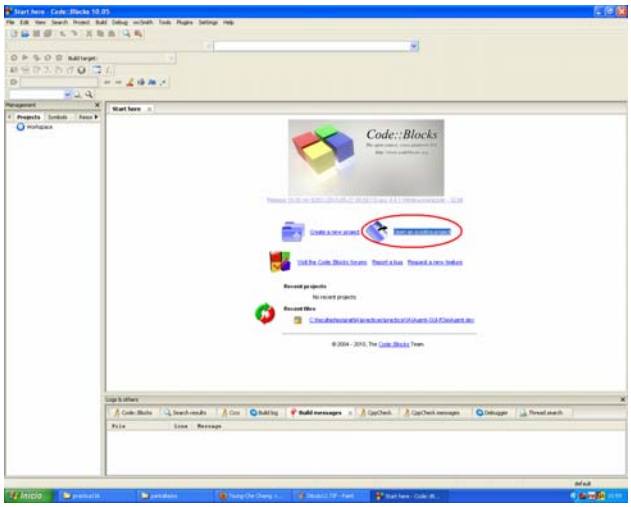

Figura 11: Abrir un fichero existente en CodeBlocks

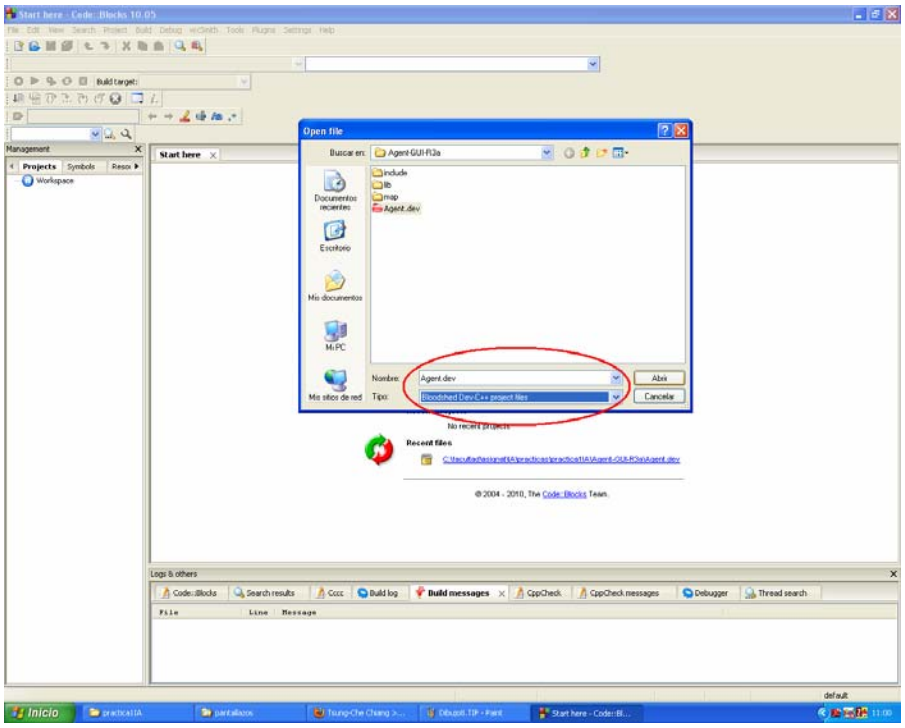

Figura 12: Selección del tipo de proyecto a abrir y el fichero principal del mismo

4. De nuevo, **CodeBlocks** nos pide que confirmemos el compilador por defecto a usar para compilar el simulador. Por tanto, volvemos a seleccionar el compilador **GNU GCC Compiler** y pulsamos el botón **ok** (Figura 13).

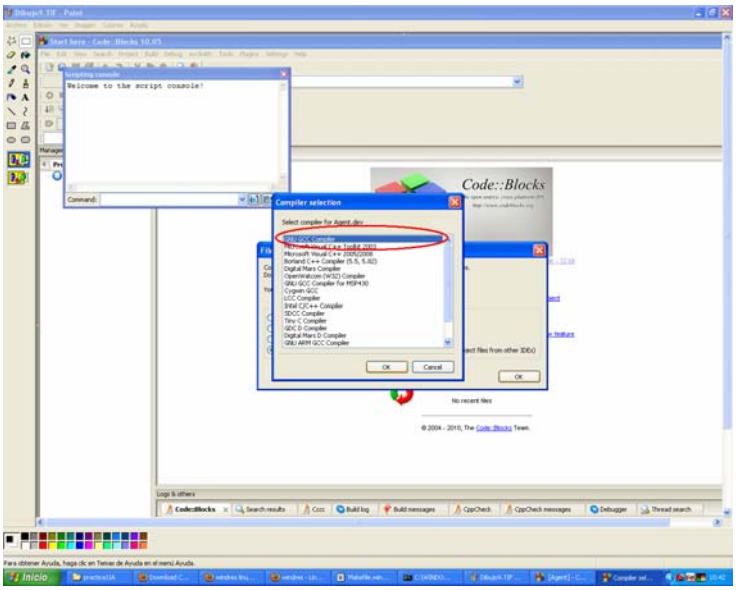

Figura 13: Selección del compilador para el proyecto *Agent.dev*

5. Tras realizar los pasos anteriores, hemos conseguido abrir el proyecto. En la parte izquierda de la pantalla podremos ver el proyecto **Agent**, junto con sus ficheros de código fuente **.cpp** (subcarpeta **Sources**) y los correspondientes ficheros de cabecera **.h** (subcarpeta **Headers**), tal y como muestra la Figura 14. **Los ficheros que mayor interés proporciona para la práctica son *agent.h* y *agent.cpp*,** los cuales tendremos que modificar para implementar el comportamiento deseado del agente.

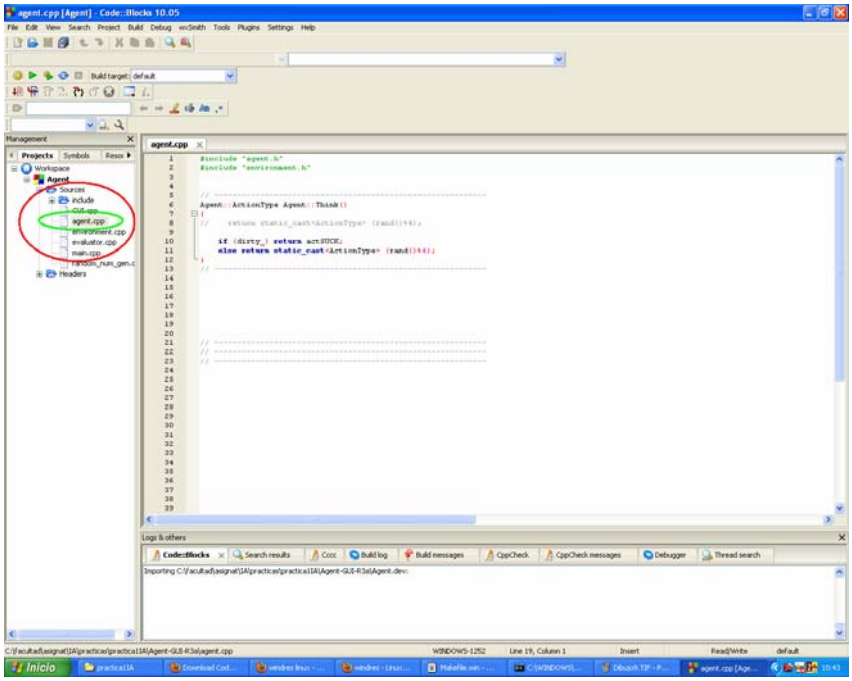

Figura 14: Ficheros del proyecto *Agent*

6. **Inclusión de bibliotecas adicionales**. Para incluir las bibliotecas adicionales que necesitamos para compilar el proyecto, debemos abrir la opción "**Build options**" del menú "**Project**" (Figura 15). Allí, deberemos seleccionar la opción "**Linker settings**" y eliminar las bibliotecas que pueda haber en el recuadro "**Link libraries**", tal y como se muestra en la Figura 16. Por último, pulsaremos en el botón **add** para insertar las nuestras (ficheros **libglu32.a**, **libglut32.a, libopengl32.a**), las cuales extrajimos anteriormente en la carpeta "**lib**" del proyecto (Figura 17). Al incluir las 3 bibliotecas, pulsaremos el botón **ok** para volver a la ventana principal.

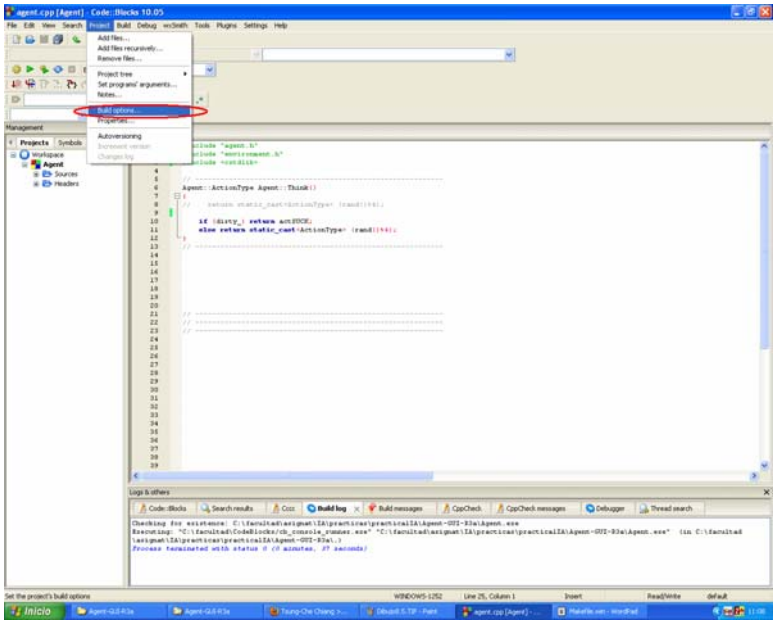

Figura 15: Acceso a opciones de compilación del proyecto

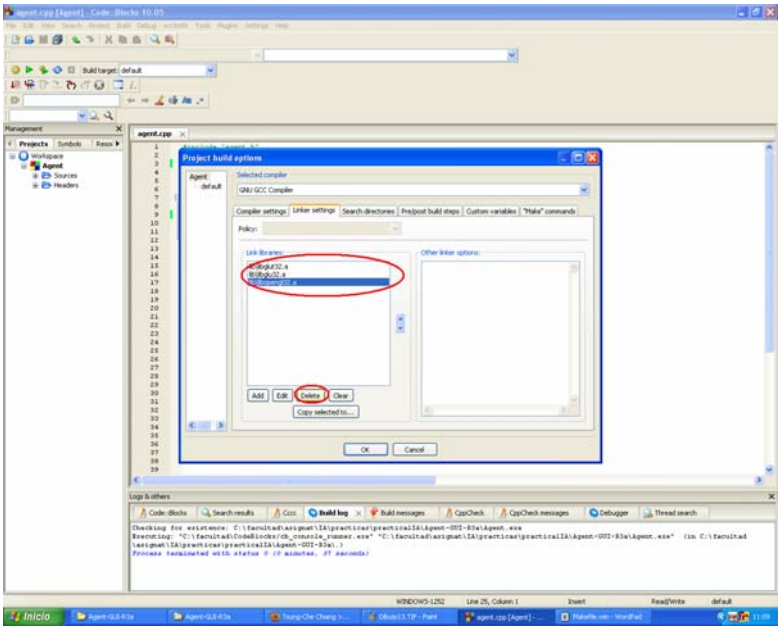

Figura 16: Eliminación de bibliotecas asociadas existentes

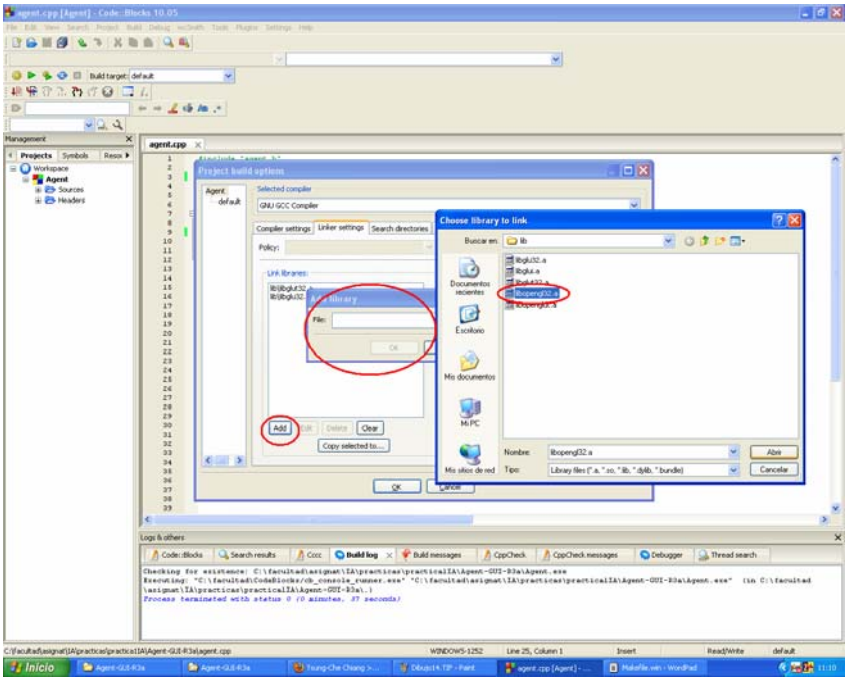

Figura 17: Inclusión de bibliotecas propias

7. **Compilación y depuración de errores**. El proyecto se compila desde la opción "**Build**" del menú "**Build**" (Figura 18). En el caso de existencia de errores, el programa no generará ningún fichero ejecutable y mostrará los errores encontrados durante el proceso de compilación. En nuestro caso, hay un error premeditado, con el fin de ilustrar esta característica del entorno de programación, que consiste en que el compilador no encuentra la función **rand** en el fichero **Agent.cpp** (Figura 19). Para solucionar este error, añadiremos a la línea 3 de este fichero el código **#include<cstdlib>**. Hecho esto, podremos compilar el proyecto sin problemas.

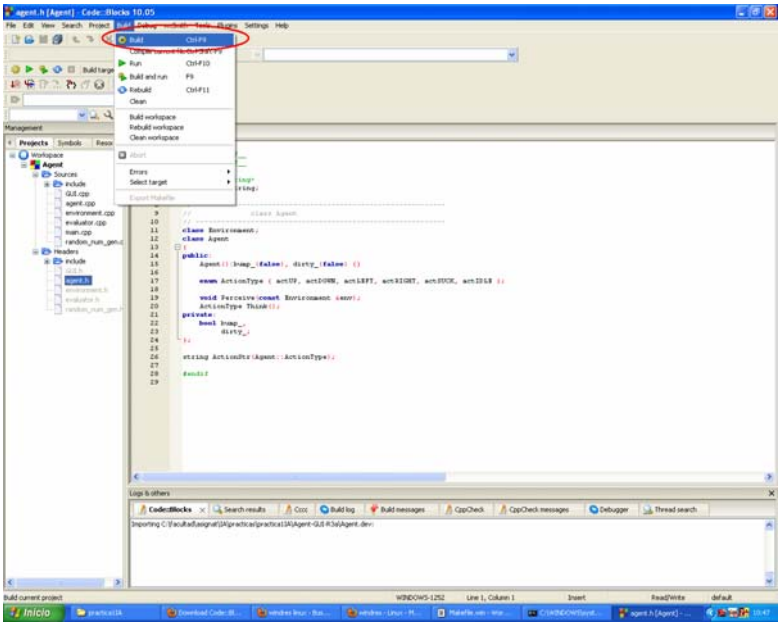

Figura 18: Compilación del proyecto

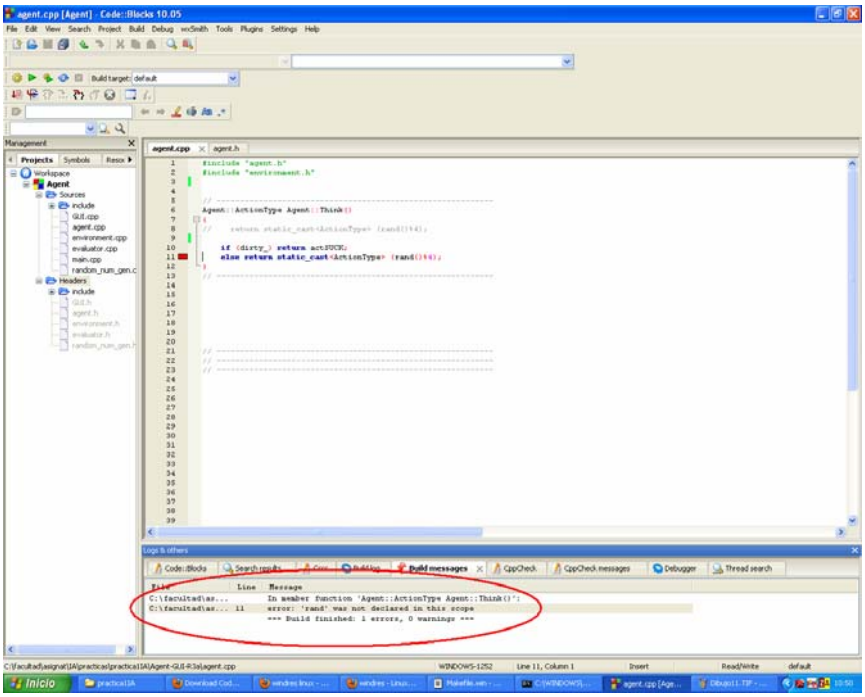

Figura 19: Error. Función *rand* no declarada. Fichero Agent.cpp, línea 11

4.4. Ejecución del simulador

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 **Agent.exe** generado en la carpeta del proyecto tras su compilación). Aparecerá una ventana como la que se muestra en la Figura 20.

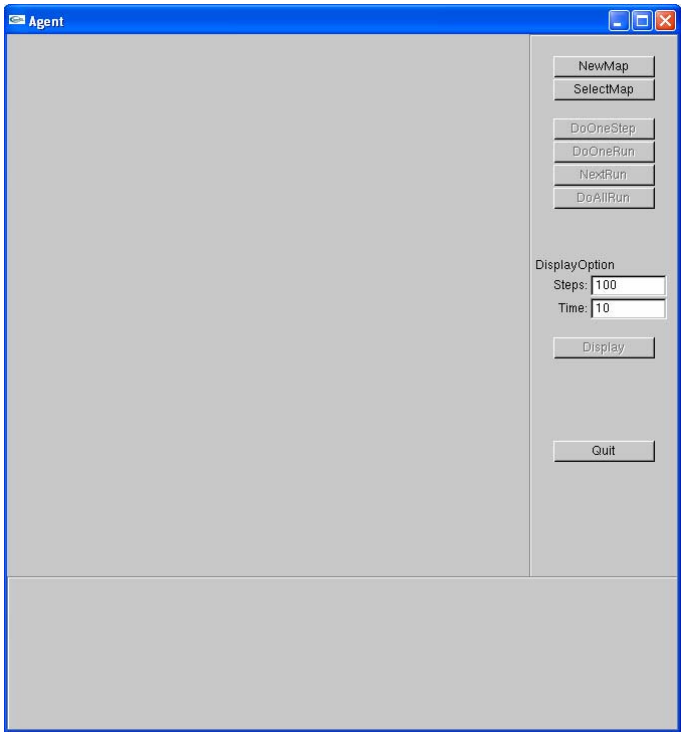

Figura 20: Vista del simulador

En esta ventana, la opción que nos interesa se encuentra en el botón "**Select Map**", que nos permite seleccionar un mapa del entorno. Por defecto, el simulador trae un único mapa "**Agent.map**", aunque en la sección 5 describimos cómo crearnos nuestros propios mapas para realizar la práctica.

Tras cargar el mapa "**Agent.map**", la vista del simulador cambia mostrando el mundo del agente, la posición donde comienza, y se activan las opciones de ejecución de simulación (Figura 21): "**DoOneStep**", "**DoOneRun**", etc.. Las que nos interesan son:

- **DoOneStep:** Permite avanzar un instante de tiempo, suficiente para que el agente realice una de las acciones disponibles (moverse arriba, abajo, izquierda o derecha, limpiar 1 punto de suciedad, o quedarse quieto).
- **DoOneRun:** Permite ejecutar un número de pasos establecido a priori **en el campo de texto "Steps"**, en la sección "**DisplayOption**".

- **Display:** Es equivalente a "**DoOneRun**" pero, además, muestra el recorrido del agente en el simulador.

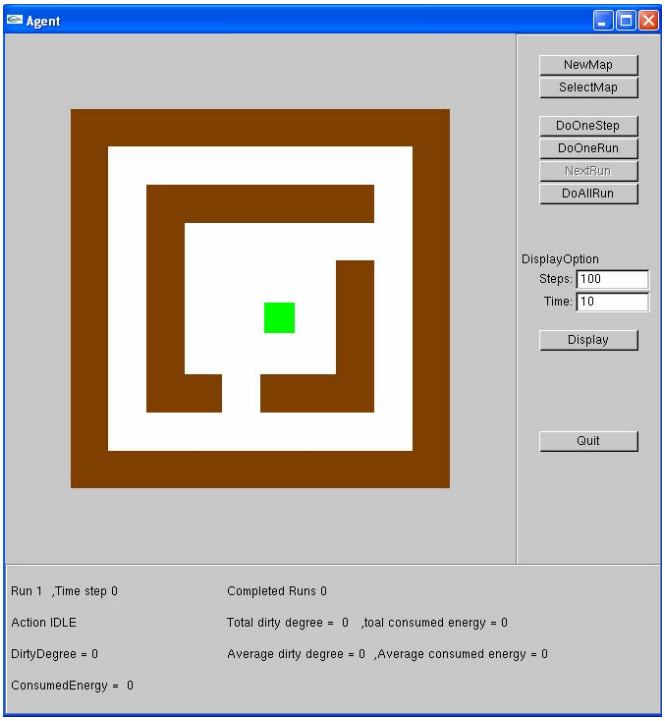

Figura 21: Vista del simulador en mapa

Adicionalmente, también resultan de interés para la práctica las variables de estado que se muestran en la parte inferior:

- **Run**: Ejecución actual.
- **TimeStep**: Instante de tiempo actual dentro de la ejecución en curso.
- **TotalDirtyDegree**: Suma de la suciedad de cada casilla al cuadrado, a lo largo de los instantes de tiempo transcurridos en la ejecución actual.
- **ConsumedEnergy**: Energía consumida por el agente hasta el instante de tiempo actual.
- **Etc**.

Se invita al alumno a experimentar con las opciones anteriores. Aclaramos que, en este ejemplo, el comportamiento implementado en el agente consiste en limpiar la casilla donde se encuentra si esta está sucia, o moverse en una dirección seleccionada al azar en caso contrario.

4.5. Mapas adicionales

El profesorado de la asignatura pone a disposición del alumno mapas adicionales para que éste evalúe los comportamientos implementados. Estos mapas se pueden descargar desde la web de la asignatura, seleccionando el fichero **MapasPractica1.rar**. Para instalar los mapas en el simulador, basta con extraer los ficheros del archivo anterior y copiarlos a la carpeta **"map"** del simulador.

5. Implementación del agente

5.1. Descripción de los ficheros del simulador

El simulador se encuentra instalado en la carpeta seleccionada en el apartado 4.2 de esta memoria (por defecto, **Agent-GUI-R3a**). Esta carpeta contiene los ficheros de código fuente y las bibliotecas necesarias para su correcta compilación. En concreto, estos son:

- Carpeta **include**: Contiene ficheros de código fuente y objeto (**.o**) necesarios para compilar la interfaz del simulador. No son relevantes para la elaboración de la práctica, aunque sí para que esta pueda compilar y ejecutarse correctamente.
- Carpeta **lib**: Contiene las bibliotecas (**.a**) básicas para poder acceder a la interfaz gráfica de OpenGL y la creación de ventanas en Windows.
- Carpeta **map**: Contiene los mapas disponibles en el simulador para modelar el mundo del agente. Discutiremos este aspecto en el apartado 5.3 de este guión.
- Fichero **Agent.exe**: Es el programa resultante, compilado y ejecutable, tras la compilación del proyecto.
- Ficheros **Agent.dev** y **Makefile.win**: Son los ficheros principales del proyecto. Contienen toda la información necesaria para poder compilar el simulador.
- Ficheros **Agent_private.*** y **agent.ico**: Ficheros de recursos de Windows para la compilación (iconos, información de registro, etc.).
- Fichero **main.cpp**: Código fuente de la función principal del programa simulador.
- Ficheros **random_num_gen.***: Ficheros de código fuente que implementan una clase para generar números aleatorios.
- Ficheros **GUI.***: Código fuente para implementar la interfaz del simulador.
- Ficheros **evaluator.***: Código fuente que implementa las funciones de evaluación del agente (energía consumida, suciedad acumulada, etc.), los cuales se muestran en la parte inferior de la ventana principal del simulador.
- Ficheros **environment.***: Código fuente que implementa el mundo del agente (mapa del entorno, suciedad en cada casilla, posición del agente, etc.).
- Ficheros **agent.***: Código fuente que implementa al agente.

5.2. Implementación del agente

Entre todos los ficheros descritos en el apartado 5.1 de este guión, únicamente será necesario modificar los archivos **agent.cpp** y **agent.h** para poder llevar a cabo las tareas de esta práctica. Concretamente, el agente se implementa en la clase **Agent**, definida en **Agent.h**. El agente tiene dos variables miembro, de tipo **bool**:

- Variable **bump_**: Esta variable la obtiene automáticamente el agente desde el entorno. Está asociada al sensor de choque, y tiene el valor **true** si el agente ha chocado contra un obstáculo intentando hacer el movimiento anterior, y el valor **false** en caso contrario.
- Variable **dirty_**: Esta variables la obtiene automáticamente el agente desde el entorno. Está asociada al sensor de suciedad, y tiene el valor **true** si el agente ha detectado suciedad en la casilla donde se encuentra, y el valor **false** en caso contrario.

Además, el agente dispone de los siguientes métodos:

- Método **void Perceive(const Environment &env)**: Este método lo utiliza el simulador para que el agente pueda percibir el entorno. Como entrada, tiene una variable de tipo **Environment** con información sobre el mundo del agente. El cometido de **Perceive** en esta práctica consiste en asignar valores a las variables **bump_** y **dirty_** según la información leída por los sensores de choque y suciedad. De este modo, la implementación de este método debe ser la siguiente (**no debe ser modificada por el alumno para la elaboración de la práctica**):

```
void Agent::Perceive(const Environment &env) {
        bump_ = env.isJustBump();
        dirty_ = env.isCurrentPosDirty();
}
```

- Método **ActionType Think()**: Este método contiene la función de selección de acciones del agente. **Este método deberá ser modificado por el alumno para implementar el comportamiento deseado del agente.** Como mínimo, debe tener en cuenta los valores de las variables internas **bump_** y **dirty_**. Como salida, este método devuelve la acción a realizar seleccionada. El tipo de salida, **ActionType**, está definido como tipo enumerado **enum** dentro de la clase **Agent**:

  ```
  enum ActionType { actUP, actDOWN, actLEFT, actRIGHT, actSUCK, actIDLE };
  ```

  Por tanto, los valores que puede tener una variable de tipo **ActionType** podrán ser alguno de los 6 siguientes:
  - Valor **actUP**: Acción de moverse a la casilla inmediatamente superior a la actual. Al ser tipo enumerado, esta acción tiene el valor entero asociado 0.
  - Valor **actDOWN**: Acción de moverse a la casilla inmediatamente inferior a la actual. Al ser tipo enumerado, esta acción tiene el valor entero asociado 1.

- Valor **actLEFT**: Acción de moverse a la casilla inmediatamente a la izquierda a la actual. Al ser tipo enumerado, esta acción tiene el valor entero asociado 2.
- Valor **actRIGHT**: Acción de moverse a la casilla inmediatamente a la derecha a la actual. Al ser tipo enumerado, esta acción tiene el valor entero asociado 3.
- Valor **actSUCK**: Acción de limpiar 1 unidad de suciedad en la casilla actual donde se encuentra el agente. Al ser tipo enumerado, esta acción tiene el valor entero asociado 4.
- Valor **actIDLE**: No produce ninguna acción. El agente permanece inmóvil. Al ser tipo enumerado, esta acción tiene el valor entero asociado 5.

Como ejemplo de implementación de un comportamiento del agente, el siguiente código (a implementar en el fichero **agent.cpp**) realiza una succión de suciedad en caso de que la casilla en la que se encuentra el agente esté sucia, o un movimiento a una casilla adyacente seleccionada de forma aleatoria en otro caso:

```
Agent::ActionType Agent::Think() {
	int i;

	if (dirty_) return actSUCK;
	else i= rand()%4;

	switch(i) {
		case 0: return actUP; break;
		case 1: return actDOWN; break;
		case 2: return actLEFT; break;
		case 3: return actDOWN; break;
		default: return actIDLE;
	}
}
```

Se recomienda al alumno modificar el código del método **Think** en el fichero **agent.cpp** para comprobar diferentes comportamientos simples (sólo moverse en una dirección, alternar entre moverse y succionar, realizar algún movimiento dependiendo del sensor de choque, etc.), con el fin de familiarizarse con el proceso de implementación y prueba en el simulador.

Adicionalmente, si el alumno estima necesario generar nuevas variables, vectores o matrices miembro de la clase **Agent**, así como crear nuevos métodos auxiliares que sean necesarios para desarrollar las tareas (según el diseño que haya realizado para el comportamiento del agente en la práctica), estas deben declararse e inicializarse de la siguiente forma:

- **Declaración de nuevas variables miembro**: En el fichero **agent.h**, la sección privada de la clase (líneas 21-23) contiene declaradas las variables miembro **bump_** y **dirty_**. Las variables adicionales que desee crear el alumno deberán declararse en esta sección. Por ejemplo, si el alumno estimase necesaria una variable **aux**, el código privado de la clase quedaría de la siguiente forma:

```
…
private:
        bool bump_,
                dirty_;
        int aux;
…
```

- **Inicialización de variables miembro**: Las variables deben ser inicializadas en el **constructor por defecto** de la clase **Agent**. En el código inicial que se proporciona en el simulador, el constructor por defecto inicializa las variables **bump_** y **dirty_** al valor **false** en la línea 15 del fichero **agent.h** de la siguiente forma:

```
Agent():bump_(false), dirty_(false) {}
```

El código anterior equivale al siguiente:

```
Agent() {
        bump_= false;
        dirty_= false;
}
```

Para inicializar la nueva variable **aux** en el constructor al valor 0, bastaría con modificar el código de la siguiente forma:

```
Agent() {
        bump_= false;
        dirty_= false;
        aux= 0;
}
```

Una vez implementada la inicialización, la variable estaría lista para ser usada y modificada en el método **Think** del agente.

- **Declaración de nuevos métodos:** Deben declararse en el fichero **agent.h, dentro de la clase Agent**. Por cuestiones formales, deberían declararse en la sección privada de la clase. Por ejemplo, si deseamos crear un nuevo método **void MiMetodo(),** que se encargue de realizar algún tipo de procesamiento auxiliar, este se podría declarar de la siguiente forma:

```
…
private:
        bool bump_,
                dirty_;
        int mimundo[10][10],

        void MiMetodo();
…
```

- **Implementación de los nuevos métodos:** Mientras que la declaración de los métodos se realiza durante la definición de la clase (fichero **agent.h**), su implementación debe incluirse en el fichero **agent.cpp**. Así, para implementar el método previamente declarado **MiMetodo**, se debería incluir en el fichero **agent.cpp** el siguiente código (NOTA: No olvidar incluir el prefijo **Agent::** previo al nombre del método para indicar que es método de la clase **Agent** y no una función independiente):

```
…
void Agent::MiMetodo() {
        // Código fuente del método
}
…
```

5.3. Generación de mapas

Además de los mapas proporcionados por defecto y por el profesorado en la web de la asignatura, se da la posibilidad al alumno de generar sus propios mapas para realizar pruebas adicionales. **Los mapas realizados deben colocarse en la carpeta "map" del simulador.**

Un mapa es un fichero de texto cuyo nombre acaba obligatoriamente en la extensión "**.map**". Su contenido se ilustra en la Figura 22, la cual muestra el contenido del mapa por defecto proporcionado por el simulador:

```
// Initial position, dirty probability, random seed, map
5 5 0.01 1
0 0 0 0 0 0 0 0 0 0
0 - - - - - - - - 0
0 - 0 0 0 0 0 0 - 0
0 - 0 - - - - - - 0
0 - 0 - - - - 0 - 0
0 - 0 - - - - 0 - 0
0 - 0 - - - - 0 - 0
0 - 0 0 - 0 0 0 - 0
0 - - - - - - - - 0
0 0 0 0 0 0 0 0 0 0
```

Figura 22: Esquema de fichero de definición de mapas

- La primera línea del fichero contiene un comentario que indica el significado de los valores en la siguiente línea.
- La segunda línea del fichero contiene la posición (X,Y) donde el agente comienza la exploración (comenzando a contar desde la coordenada (0,0)), la probabilidad de que la suciedad de cada casilla aumente en cada instante de tiempo (**debe ser fija al valor P=0.01**, como se indicó en el apartado 3 de la memoria). El último valor es el número de mapa (valor fijo igual a 1, irrelevante para el desarrollo de esta práctica). Todos estos valores están separados por un único espacio en blanco.

- Las siguientes líneas contienen las filas y columnas del mapa que define el mundo del agente. Un valor **O** (**o** mayúscula) indica que en la casilla hay un obstáculo, mientras que un valor **–** (signo negativo) sirve para definir una casilla vacía. Todos los valores de una misma fila se deben separar por un único espacio en blanco.
- El mapa debe tener un tamaño de 10x10 fijo.
- Todas las casillas del borde del mapa deben tener valor **O**, con el fin de satisfacer la restricción comentada en el apartado 3 acerca de que la habitación esté cerrada.

Con estas indicaciones, y guardando el mapa como fichero de texto con extensión "**.map**" en la carpeta "**map**" del simulador (editándolo, por ejemplo, con el programa **notepad** de Windows), el alumno dispone de nuevos mecanismos para evaluar su implementación sobre el comportamiento del agente en entornos adicionales a los proporcionados.

6. Evaluación y entrega de prácticas

La calificación final de la práctica será la media ponderada de los aspectos susceptibles de evaluación en la práctica. Para cada comportamiento se realizarán 10 ejecuciones del algoritmo en un mapa proporcionado por el profesor durante la defensa de la práctica.

La calificación de la eficacia del agente diseñado se obtendrá en función de la suciedad acumulada al cuadrado a lo largo de las 10 ejecuciones, más la energía consumida media por el agente. A menor valor de esta medida, mayor será la calificación. En cada ejecución, se realizarán 2000 acciones por el agente para comprobar su comportamiento.

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 4. Para obtener la máxima calificación, es de especial relevancia que el alumno explique en la memoria claramente el diseño

propuesto para el comportamiento del agente (diseño de la estructura del agente, diseño de la función de selección de acciones, etc.), así como un buen análisis de la solución aportada. También se valorará positivamente que el alumno explique con claridad la relación entre el diseño realizado y la implementación.

- Las cuestiones realizadas por el profesor durante la defensa de prácticas y correctamente respondidas por el alumno se evalúan de 0 a 4. El valor de cada pregunta es el mismo.
- La eficacia de la solución se evalúa de 0 a 2 puntos. El alumno que obtenga la mejor solución ***(menor valor en la cantidad final de suciedad en la habitación)*** obtendrá una calificación máxima en este apartado de 2 puntos. El alumno que obtenga la peor solución obtendrá una calificación mínima de 0. El resto de alumnos obtendrá una calificación proporcional a la de los compañeros que hayan obtenido la mínima y máxima calificación, en función de la bondad de la solución.

- 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 (**agent.cpp** y **agent.h**) con el comportamiento requerido para el agente. Estos ficheros deberán entregarse mediante la plataforma web de la asignatura, en un fichero ZIP que contenga carpetas separadas. La descripción de los ficheros y las carpetas contenidas en este fichero ZIP deberán estar correctamente indicadas en la memoria de prácticas. Una carpeta deberá contener **sólo el código fuente de estos dos ficheros** con la solución del alumno, y la otra carpeta deberá contener la versión electrónica de la memoria. **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 (entorno, características del agente, etc.)
2. Descripción de la solución planteada
3. Resultados obtenidos por la solución aportada en los distintos mapas.
4. Código fuente de los ficheros **agent.cpp** y **agent.h**.