



ugr

Universidad  
de Granada

# SEMINARIO 3

## Presentación Práctica 2

### Métodos de Búsqueda

**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 Simulador
4. Pasos del desarrollo de la práctica
5. Método de evaluación de la práctica

# Índice

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

# 1. Introducción

- El objetivo de esta práctica consiste en la implementación de:
  - *Un algoritmo de búsqueda en profundidad*
  - *Una técnica de escalada*
- Trabajaremos con un simulador software modificado de una 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.
  - Conocer las técnicas más representativas de búsqueda no informada en un espacio de estados (en profundidad, en anchura y sus variantes), y saber analizar su eficiencia en tiempo y espacio.
  - 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.

# 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 Simulador
4. Pasos del desarrollo de la práctica
5. Evaluación de la práctica



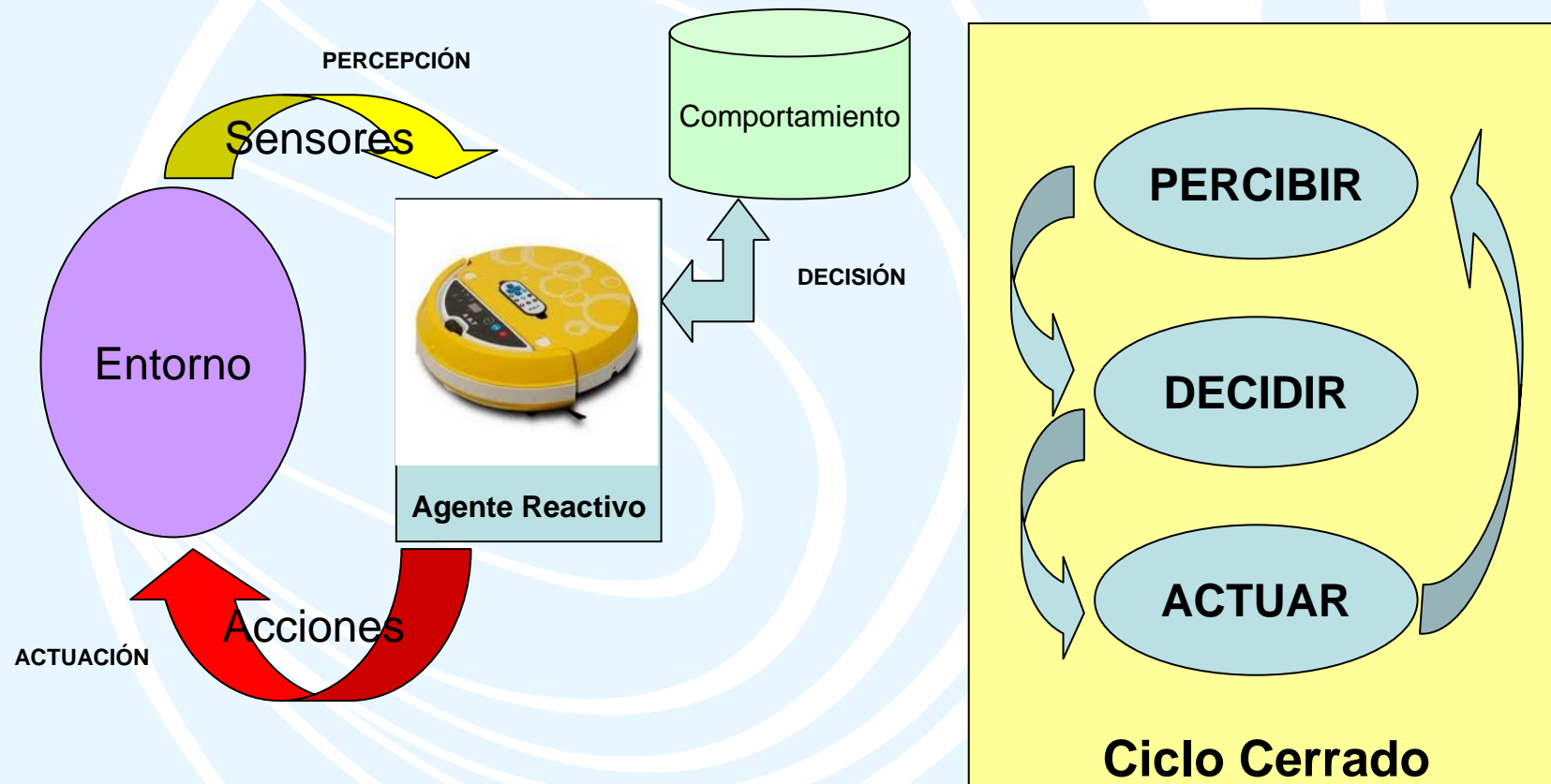
## 2. Presentación del Problema

- Aspiradora Inteligente

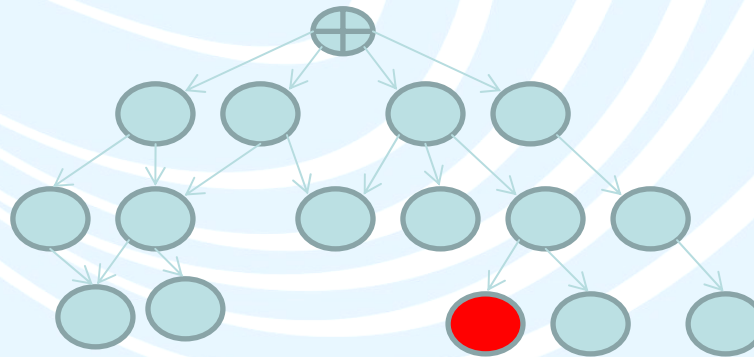




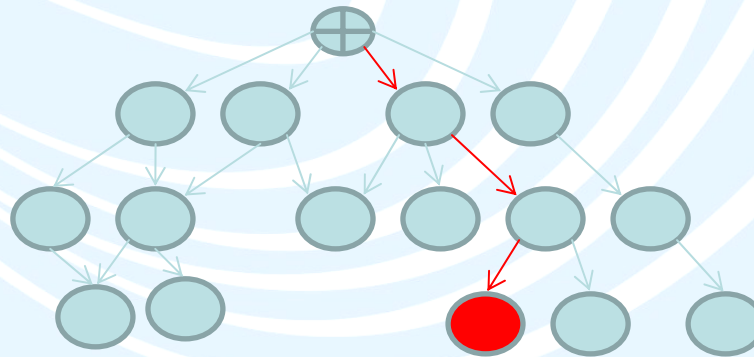
## 2. Presentación del Problema



## 2. Presentación del Problema



## 2. Presentación del Problema



## 2. Presentación del Problema

Ahora el problema se plantea como:

- Dado un estado inicial
- Encontrar una secuencia de acciones que limpie completamente una habitación consumiendo la menor cantidad de energía.

Sabiendo que

1. No va a aparecer nueva suciedad en la habitación
2. Un movimiento consume 1 unidad de energía y aspirar una unidad de suciedad consume 2 unidades.

# Índice

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

### 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\practica2**”
2. Descargar **AgentMod.zip** desde la [web](#) de la asignatura y cópielo en la carpeta anterior.



- (a) <http://decsai.ugr.es>
- (b) Entrar en acceso identificado
- (c) Elegir la asignatura “Inteligencia Artificial”
- (d) Seleccionar “Material de la Asignatura”
- (e) Seleccionar “Práctica 2”
- (f) Seleccionar “Material para la Práctica 2”



# 3. Presentación del Simulador

## 3.1. Compilación del Simulador

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

– AgentMod

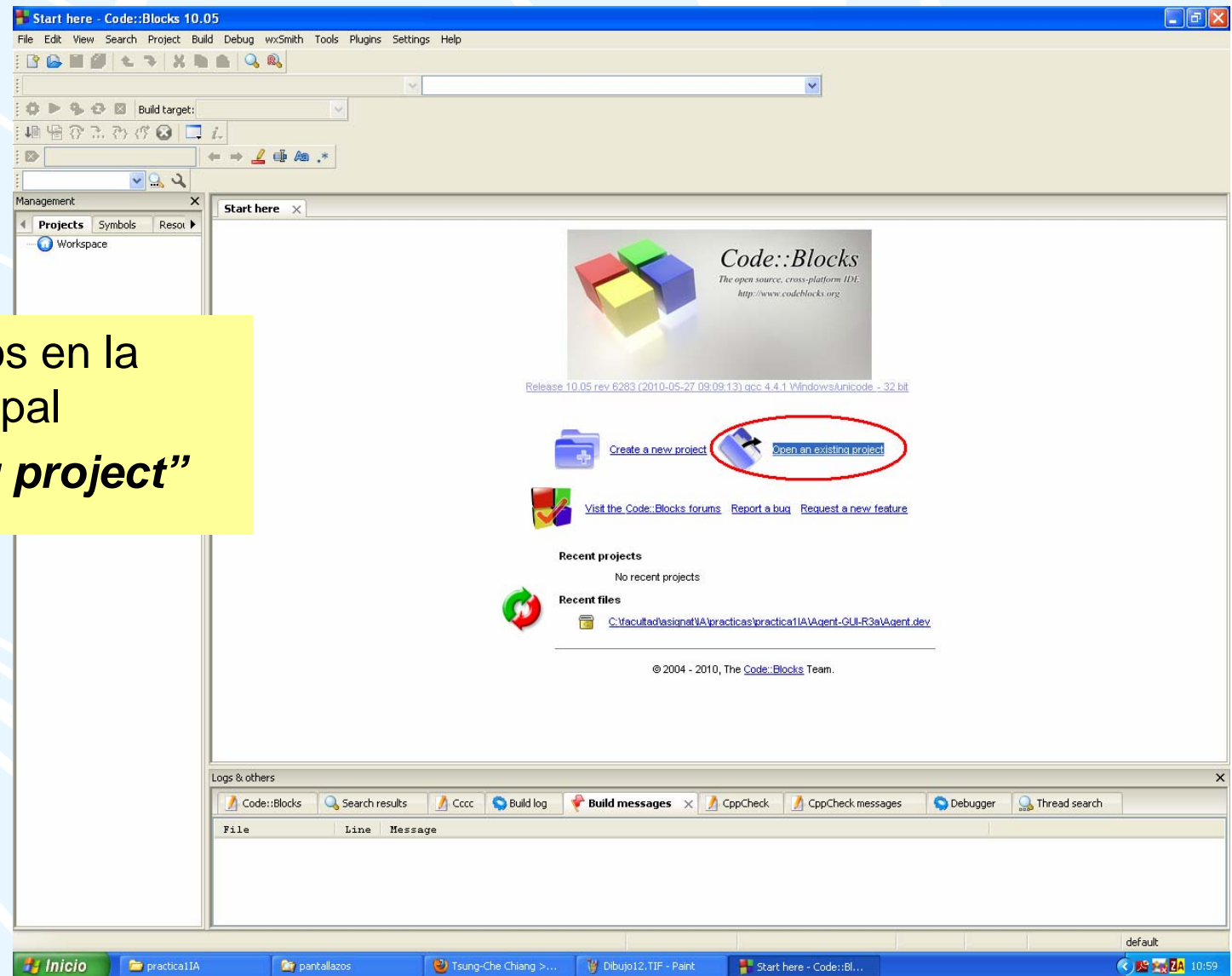
## 4. Abrimos “CodeBlocks”

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

# 3. Presentación del Simulador

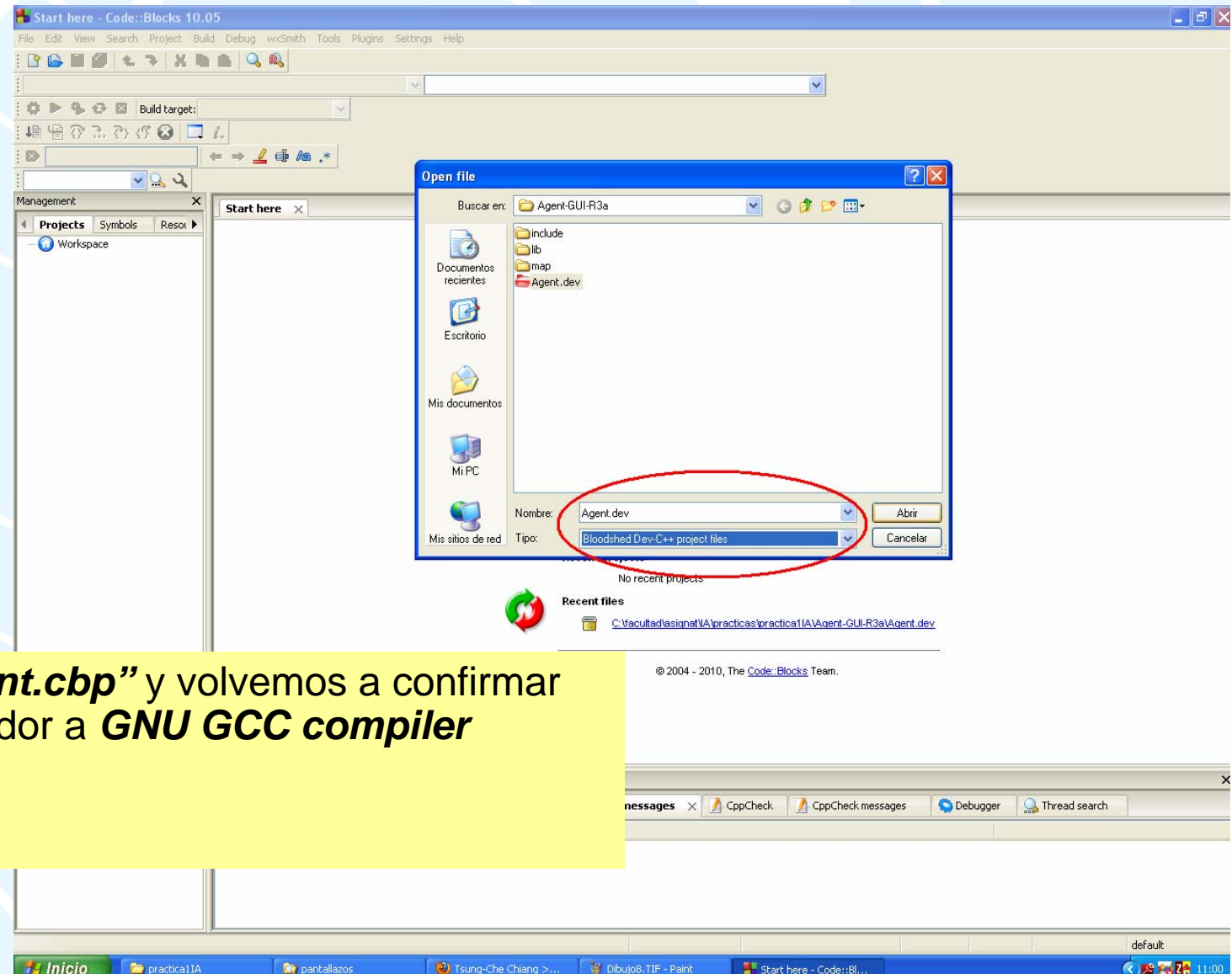
## 3.1. Compilación del Simulador

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



# 3. Presentación del Simulador

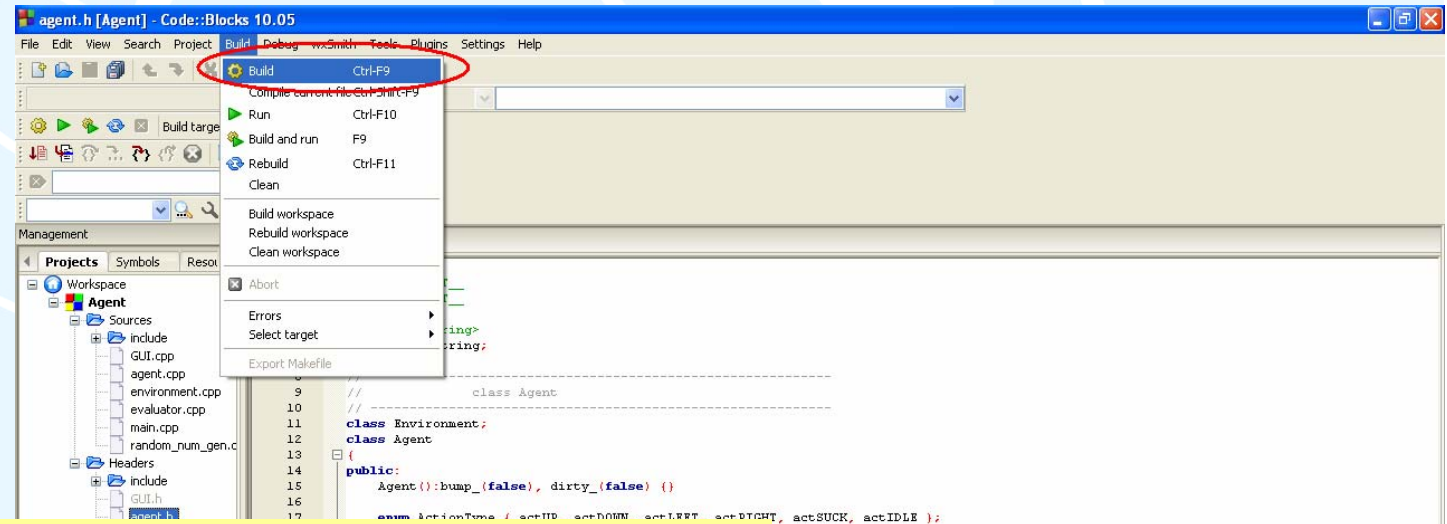
## 3.1. Compilación del Simulador



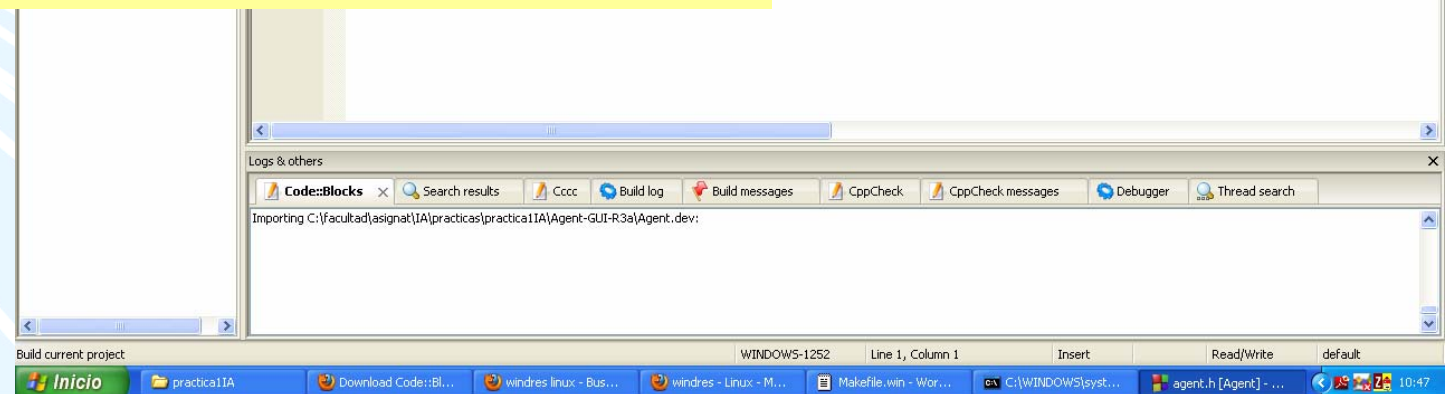
6. Abrimos **“Agent.cbp”** y volvemos a confirmar como Compilador a **GNU GCC compiler**

# 3. Presentación del Simulador

## 3.1. Compilación del Simulador

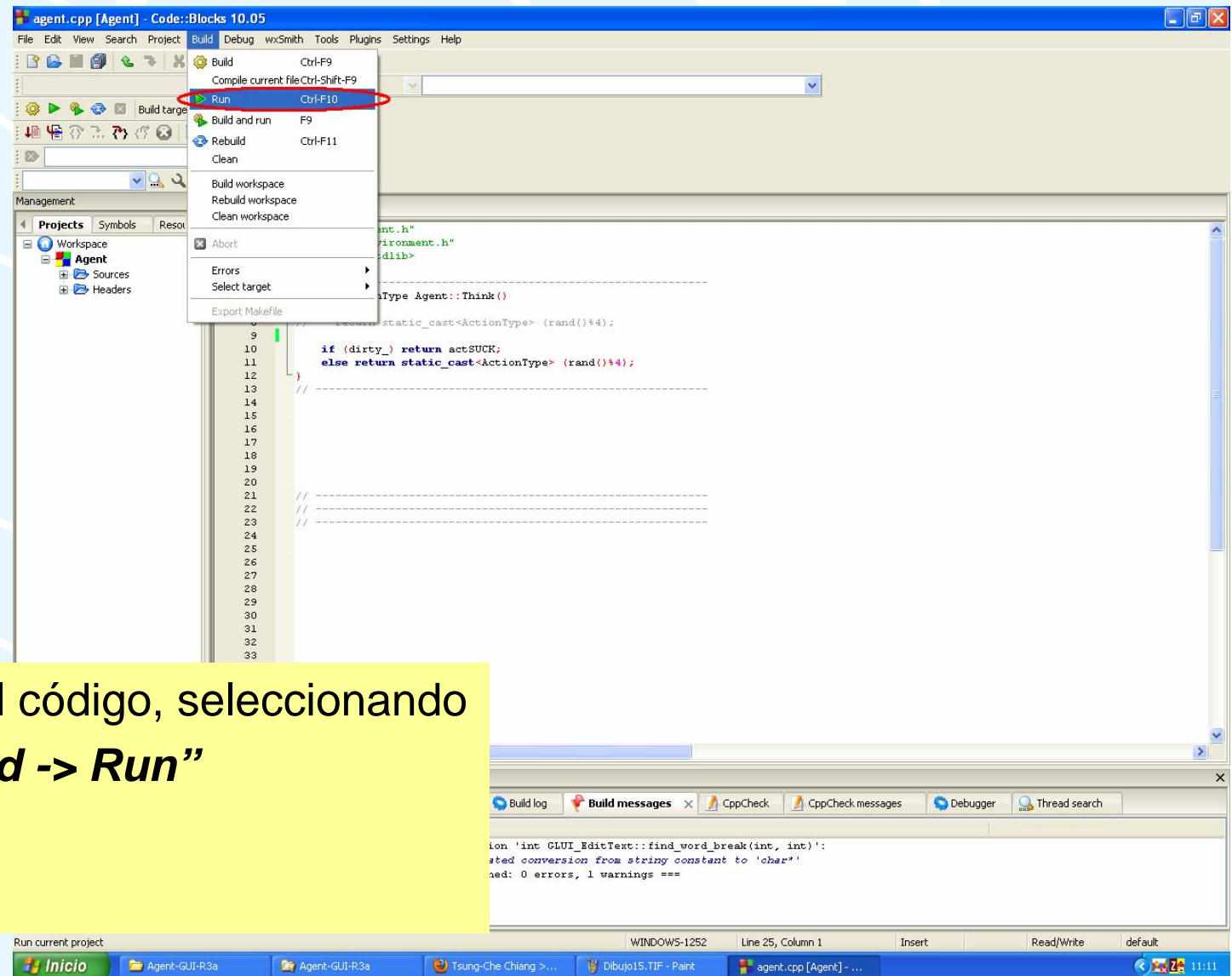


7. Compilamos el proyecto seleccionando  
***“Build → Build”***



# 3. Presentación del Simulador

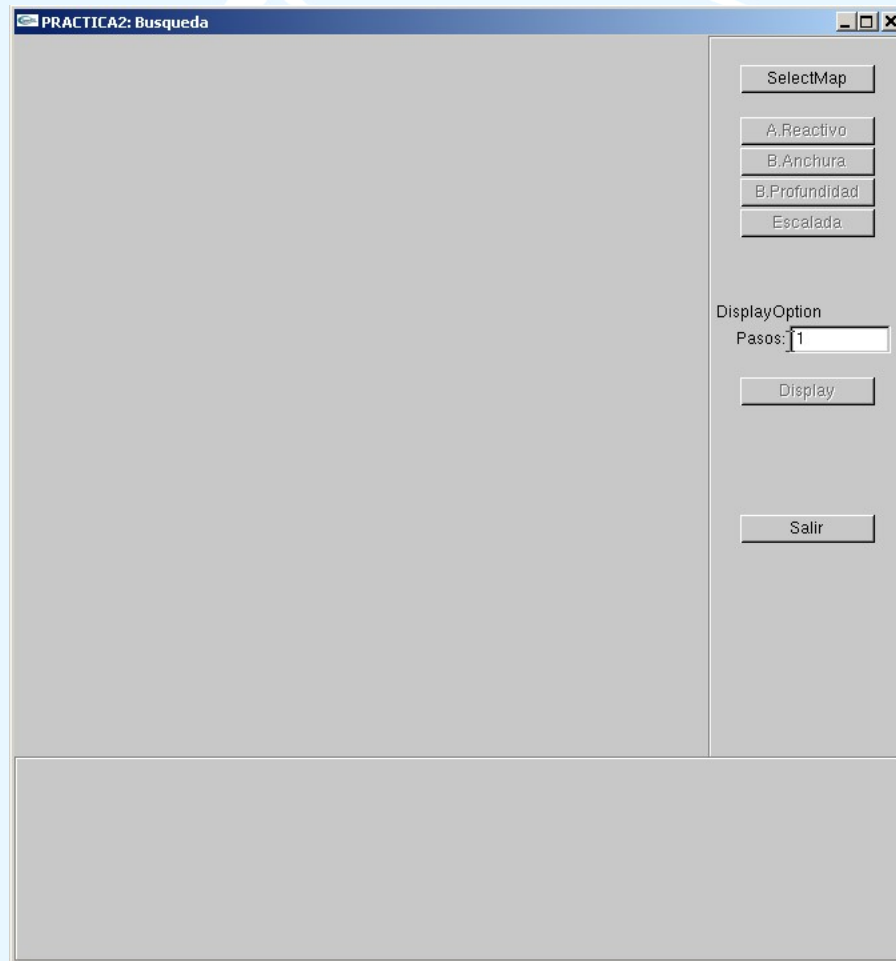
## 3.1. Compilación del Simulador



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

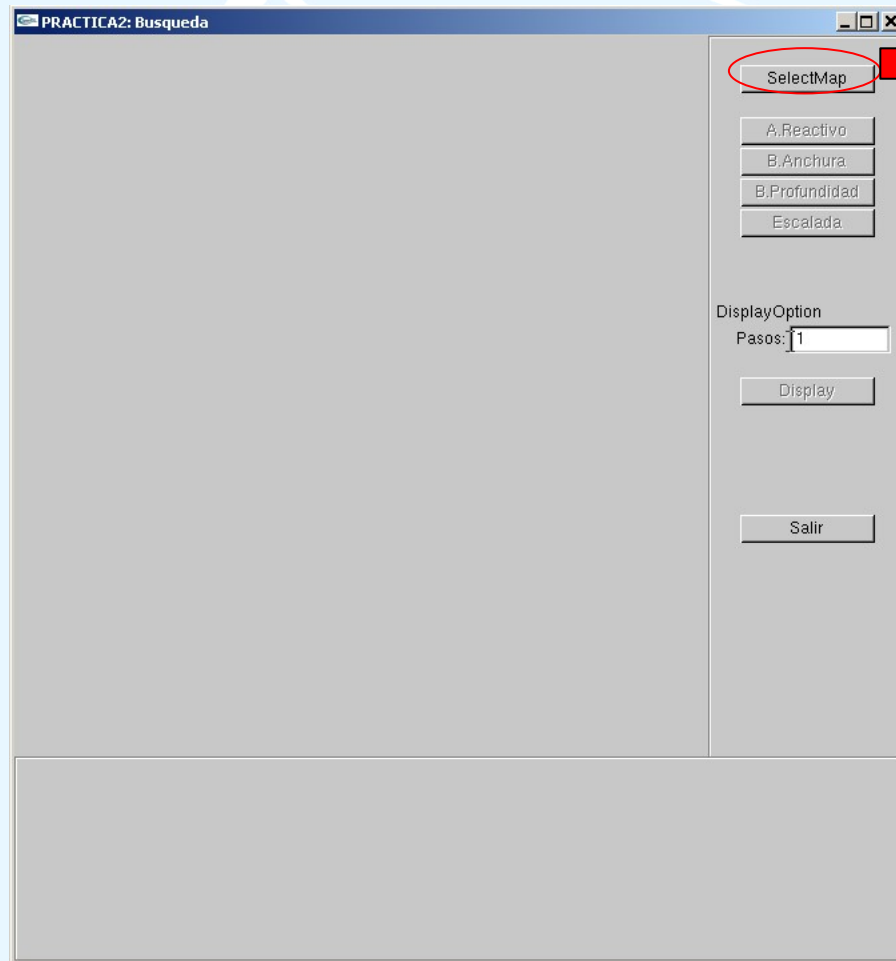
# 3. Presentación del Simulador

## 3.2. Ejecución del Simulador



# 3. Presentación del Simulador

## 3.2. Ejecución del Simulador



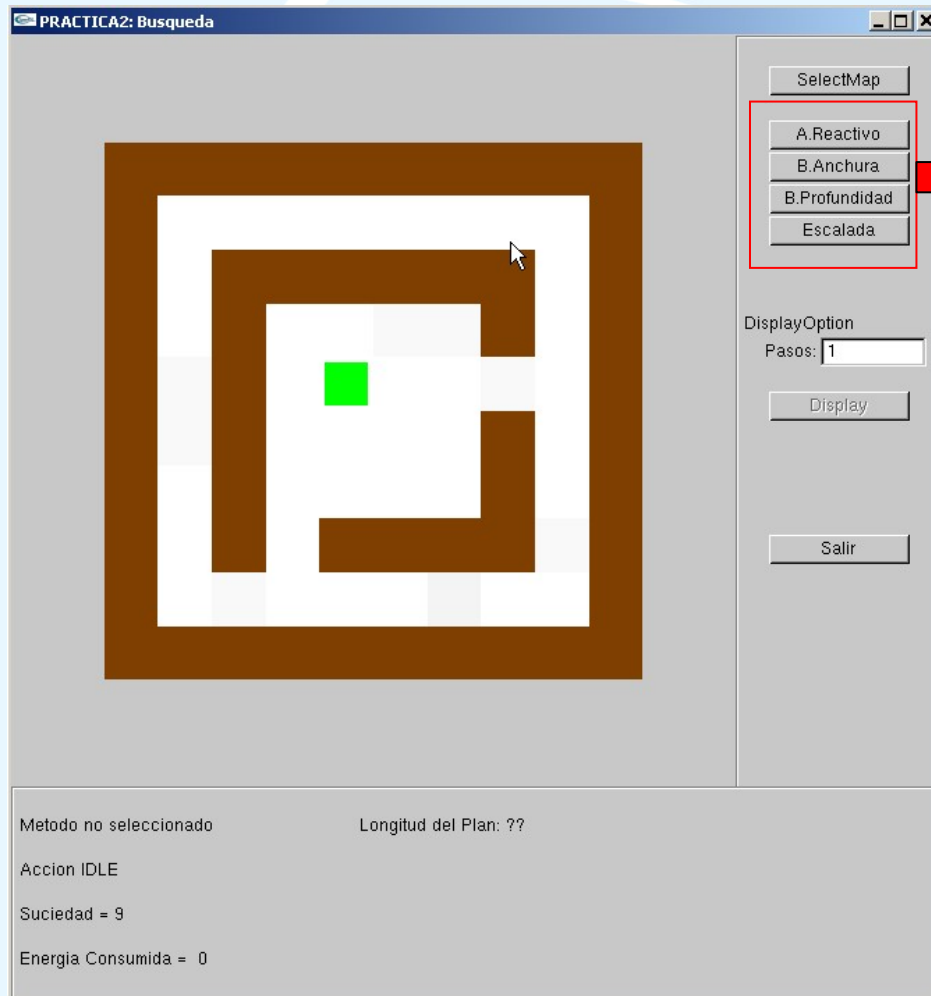
***“SelectMap”*** Selecciona el fichero de problema sobre el que Realizar la simulación.

Seleccionamos el fichero  
***“mapa10a.map”***



# 3. Presentación del Simulador

## 3.2. Ejecución del Simulador



**“A.Reactivo”** Invoca al agente reactivo para resolver el problema.

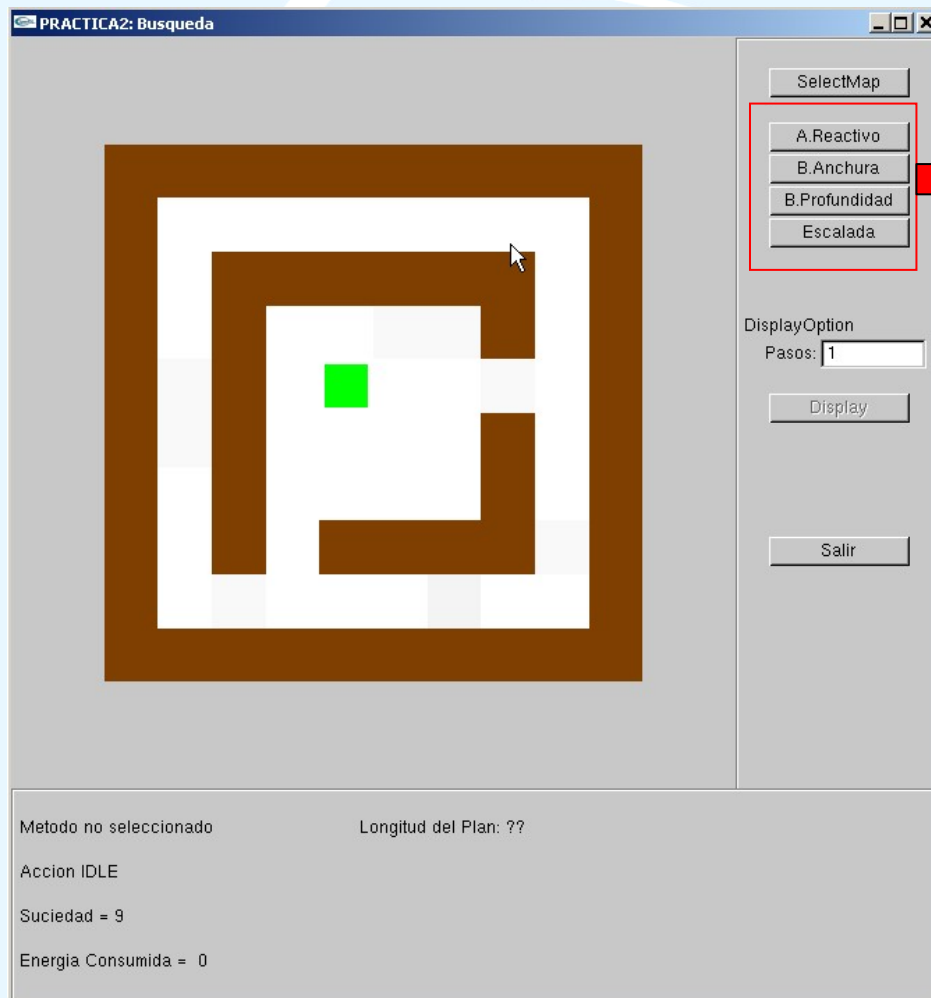
**“B.Anchura”** invoca una búsqueda en anchura como solución.

**“B.Profundidad”** Invoca una búsqueda en profundidad.

**“Escalada”** Una técnica de escalada se usa para encontrar la solución.

# 3. Presentación del Simulador

## 3.2. Ejecución del Simulador



**“A.Reactivo”** Invoca al agente reactivo para resolver el problema.

**“B.Anchura”** invoca una búsqueda en anchura como solución.

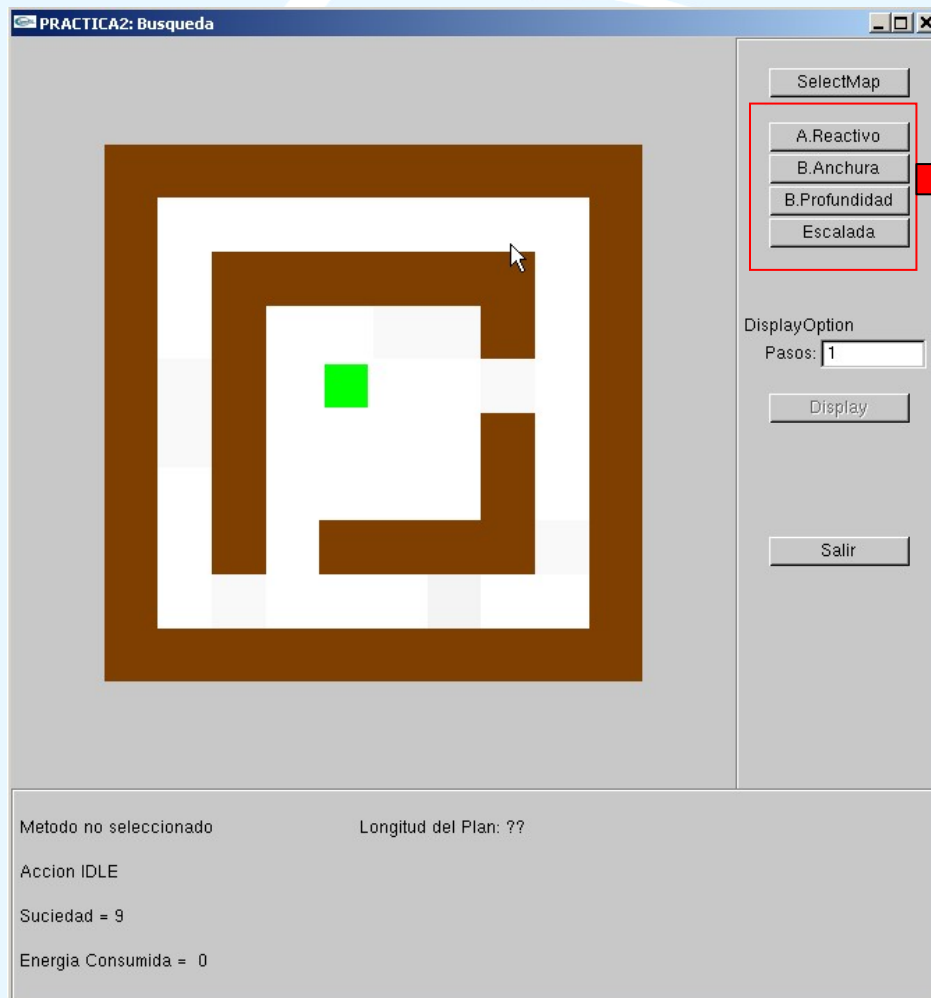
**“B.Profundidad”** Invoca una búsqueda en profundidad.

**“Escalada”** Una búsqueda de escalada se usa para encontrar una solución.

Ya  
implementado

# 3. Presentación del Simulador

## 3.2. Ejecución del Simulador



**“A.Reactivo”** Invoca al agente reactivo para resolver el problema.

**“B.Anchura”** invoca una búsqueda en anchura como solución.

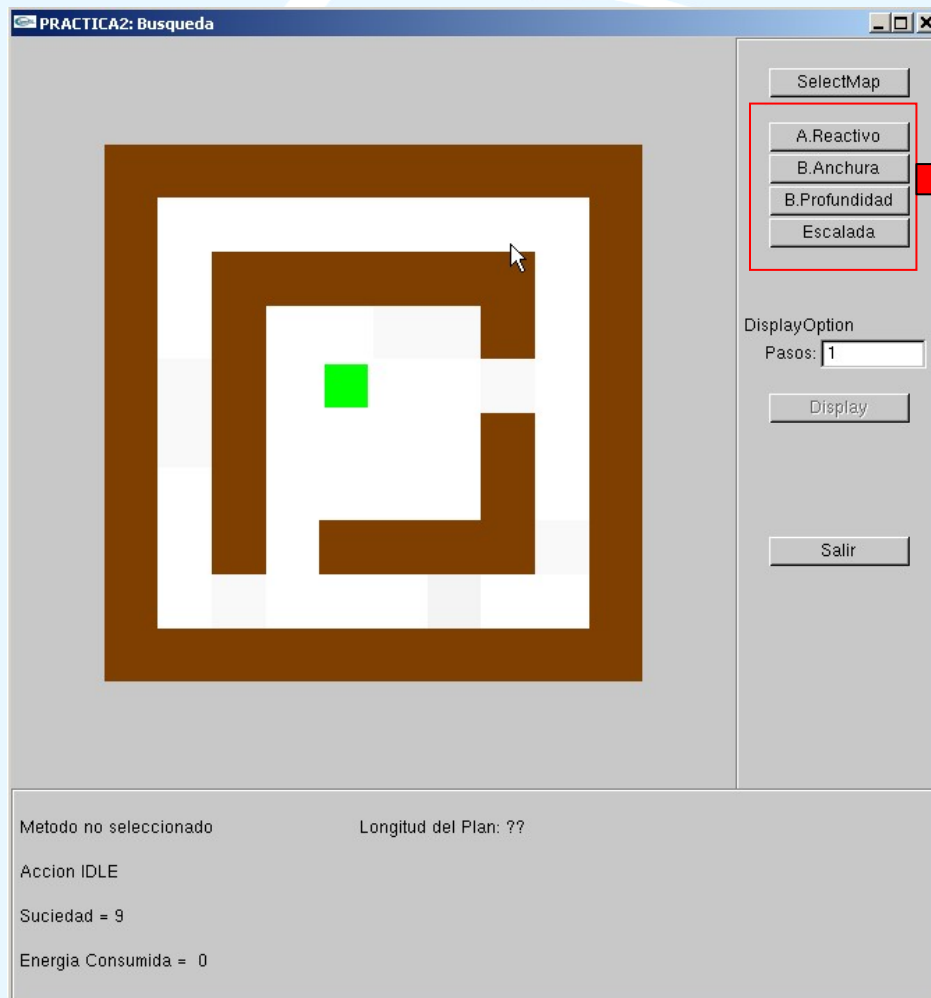
**“B.Profundidad”** invoca una búsqueda en profundidad.

**“Escalada”** Una técnica de escalada se usa para encontrar una solución.

Hay que adaptarlo de la práctica 1

# 3. Presentación del Simulador

## 3.2. Ejecución del Simulador



**“A.Reactivo”** Invoca al agente reactivo para resolver el problema.

**“B.Anchura”** invoca una búsqueda en anchura como solución.

**“B.Profundidad”** Invoca una búsqueda en profundidad.

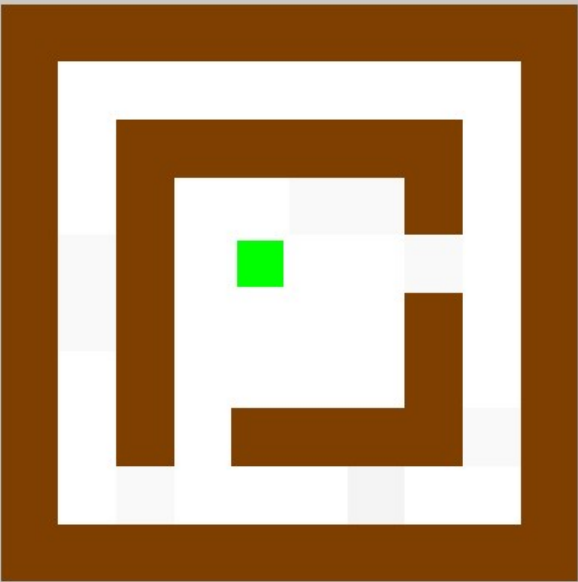
**“Escalada”** Una técnica de escalada se usa para encontrar la solución.

Deben ser implementados

C:\Documents and Settings\Raul\Mis documentos\Docencia\IA\2011\_12\Practicas\Practica2\Practi...

```
Level 22
Level 23
Level 24
Level 25
Level 26
Level 27
Level 28
Level 29
Level 30
Level 31
Level 32
Level 33
Level 34
Level 35
Level 36
Level 37
Level 38
Level 39

Longitud del Plan: 30
->UP->RIGHT->SUCK->RIGHT->SUCK->DOWN->RIGHT->SUCK->RIGHT->DOWN->DOWN->DOWN->SUCK
->DOWN->LEFT->LEFT->SUCK->SUCK->LEFT->LEFT->LEFT->LEFT->SUCK->LEFT->UP->UP->UP->
SUCK->UP->SUCK
Hecho!!
```



A.Reactivo  
B.Anchura  
B.Profundidad  
Escalada

DisplayOption  
Pasos: 1  
Display  
Salir

BUSQUEDA ANCHURA , paso: 0      Longitud del Plan: 30  
Accion IDLE      Nodos Expandidos: 13318      Nodos Explorados: 12275  
Suciedad = 9      Energia Consumida: 39  
Energia Consumida = 0

Pulsamos la opción “B.Anchura”

Veremos que en el terminal van apareciendo los niveles por donde va explorando en el grafo

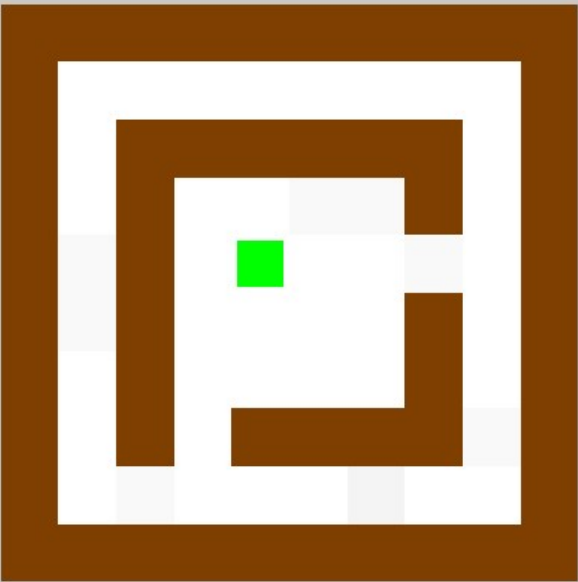
La búsqueda termina mostrando la secuencia de acciones así como la longitud del plan.

Los valores concretos del proceso de búsqueda se muestran en la parte inferior de la ventana.

C:\Documents and Settings\Raul\Mis documentos\Docencia\IA\2011\_12\Practicas\Practica2\Practi...

```
Level 22
Level 23
Level 24
Level 25
Level 26
Level 27
Level 28
Level 29
Level 30
Level 31
Level 32
Level 33
Level 34
Level 35
Level 36
Level 37
Level 38
Level 39

Longitud del Plan: 30
->UP->RIGHT->SUCK->RIGHT->SUCK->DOWN->RIGHT->SUCK->RIGHT->DOWN->DOWN->DOWN->SUCK
->DOWN->LEFT->LEFT->SUCK->SUCK->LEFT->LEFT->LEFT->LEFT->SUCK->LEFT->UP->UP->UP->
SUCK->UP->SUCK
Hecho!!
```



A.Reactivo  
B.Anchura  
B.Profundidad  
Escalada

DisplayOption  
Pasos: 1  
Display

Salir

BUSQUEDA ANCHURA , paso: 0      Longitud del Plan: 30  
Accion IDLE      Nodos Expandidos: 13318      Nodos Explorados: 12275  
Suciedad = 9      Energia Consumida: 39  
Energia Consumida = 0

Con **DisplayOption** podemos ver la secuencia de acciones obtenidas.

**Pasos** establece el número de acciones consecutivas que muestra

# Índice

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



## 4. Pasos del desarrollo de la práctica

1. Descripción de la clase ***agent***
2. Inclusión del agente reactivo
3. Definición de estado (state)
4. El fichero agent.cpp
5. Búsqueda en anchura
6. Búsqueda en profundidad
7. Técnica de escalada

# 4. Pasos del desarrollo de la práctica

## 4.1. Descripción de la clase *agent*

```
#include <string>
#include "environment.h"
#include "state.h"
#include "plan.h"

using namespace std;

// -----
//          class Agent
// -----

class Agent
{
public:
    Agent(){
        bump_ = false;
        dirty_ = false;
    }

    ~Agent(){
    }

    void Perceive(const Environment &env);
    Environment::ActionType AgenteReactivo();
    Plan Busqueda_Anchura(state start);
    Plan Busqueda_Profundidad(state start);
    Plan Escalada(state start);
    Plan Think(const Environment &env, int option);

private:
    bool bump_;
    bool dirty_;
};

string ActionStr(Environment::ActionType);
```

La función se mantiene igual que en la práctica anterior.

# 4. Pasos del desarrollo de la práctica

## 4.1. Descripción de la clase *agent*

```
#include <string>
#include "environment.h"
#include "state.h"
#include "plan.h"

using namespace std;

// -----
//          class Agent
// -----

class Agent
{
public:
    Agent(){
        bump_ = false;
        dirty_ = false;
    }

    ~Agent(){
    }

    void Perceive(const Environment &env);
    Environment::ActionType AgenteReactivo();
    Plan Busqueda_Anchura(state start);
    Plan Busqueda_Profundidad(state start);
    Plan Escalada(state start);
    Plan Think(const Environment &env, int option);

private:
    bool bump_;
    bool dirty_;
};

string ActionStr(Environment::ActionType);
```

Esta es ahora la función que contendrá el comportamiento del agente reactivo

# 4. Pasos del desarrollo de la práctica

## 4.1. Descripción de la clase *agent*

```
#include <string>
#include "environment.h"
#include "state.h"
#include "plan.h"

using namespace std;

// -----
//          class Agent
// -----

class Agent
{
public:

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

    ~Agent(){
    }

    void Perceive(const Environment &env);
    Environment::ActionType AgenteReactivo();
    Plan Busqueda_Anchura(state start);
    Plan Busqueda_Profundidad(state start);
    Plan Escalada(state start);
    Plan Think(const Environment &env, int option);

private:
    bool bump_;
    bool dirty_;
};

string ActionStr(Environment::ActionType);
```

Estos tres métodos implementan los tres procesos de búsqueda de la práctica:

Anchura, Profundidad y Escalada

# 4. Pasos del desarrollo de la práctica

## 4.1. Descripción de la clase *agent*

```
#include <string>
#include "environment.h"
#include "state.h"
#include "plan.h"

using namespace std;

// -----
//          class Agent
// -----

class Agent
{
public:
    Agent(){
        bump_ = false;
        dirty_ = false;
    }

    ~Agent(){
    }

    void Perceive(const Environment &env);
    Environment::ActionType AgenteReactivo();
    Plan Busqueda_Anchura(state start);
    Plan Busqueda_Profundidad(state start);
    Plan Escalada(state start);
    Plan Think(const Environment &env, int option);

private:
    bool bump_;
    bool dirty_;
};

string ActionStr(Environment::ActionType);
```

Ahora el método **Think()** tiene una parametrización distinta y juega el papel de interfaz con el simulador.

Toma una variable de entorno y el método de búsqueda elegido y devuelve una secuencia de acciones

# 4. Pasos del desarrollo de la práctica

## 4.1. Descripción de la clase *agent*

```
#include <string>
#include "environment.h"
#include "state.h"
#include "plan.h"

using namespace std;

// -----
//          class Agent
// -----

class Agent
{
public:

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

    ~Agent(){
    }

    void Perceive(const Environment &env);
    Environment::ActionType AgenteReactivo();
    Plan Busqueda_Anchura(state start);
    Plan Busqueda_Profundidad(state start);
    Plan Escalada(state start);
    Plan Think(const Environment &env, int option);

private:
    bool bump_;
    bool dirty_;
};

string ActionStr(Environment::ActionType);
```

Aparecen dos nuevos tipos de datos:

- **state** que define una posible configuración del estado del mundo
- **Plan** que almacena un conjunto de acciones

# 4. Pasos del desarrollo de la práctica

## 4.2. Inclusión del agente reactivo

```
#include <string>
#include "environment.h"
#include "state.h"
#include "plan.h"

using namespace std;

// -----
//          class Agent
// -----

class Agent
{
public:

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

    ~Agent(){
    }

    void Perceive(const Environment &env);
    Environment::ActionType AgenteReactivo();
    Plan Busqueda_Anchura(state start);
    Plan Busqueda_Profundidad(state start);
    Plan Escalada(state start);
    Plan Think(const Environment &env, int option);

private:
    bool bump_;
    bool dirty_;

};

string ActionStr(Environment::ActionType);
```

Para incluir vuestra versión del agente, hacer:

(1) Incluir vuestros datos miembros del agente

**Importante, si usasteis una representación de mapa circular, adaptarlo para mapas de tamaño 80x80. Si optasteis por un mapa de doble ancho, poned dimensión 200x200 y fijad la posición de inicio en la (80,80).**



# 4. Pasos del desarrollo de la práctica

## 4.2. Inclusión del agente reactivo

```
#include <string>
#include "environment.h"
#include "state.h"
#include "plan.h"

using namespace std;

// -----
//          class Agent
// -----

class Agent
{
public:
    Agent(){
        bump_ = false;
        dirty_ = false;
    }

    ~Agent(){
    }

    void Perceive(const Environment &env);
    Environment::ActionType AgenteReactivo();
    Plan Busqueda_Ambigua(state start);
    Plan Busqueda_Profundidad(state start);
    Plan Escalada(state start);
    Plan Think(const Environment &env, int option);

private:
    bool bump_;
    bool dirty_;
};

string ActionStr(Environment::ActionType);
```

Para incluir vuestra versión del agente, hacer:

(2) Renombrar el antiguo método

***ActionType Think()***

como

***Environment::ActionType AgenteReactivo()***

Incluir su implementación en ***agent.cpp***

# 4. Pasos del desarrollo de la práctica

## 4.2. Inclusión del agente reactivo

```
#include <string>
#include "environment.h"
#include "state.h"
#include "plan.h"

using namespace std;

// -----
//          class Agent
// -----

class Agent
{
public:

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

    ~Agent(){
    }

    void Perceive(const Environment &env);
    Environment::ActionType AgenteReactivo();
    Plan Busqueda_Anchura(state start);
    Plan Busqueda_Profundidad(state start);
    Plan Escalada(state start);
    Plan Think(const Environment &env, int option);

private:
    bool bump_;
    bool dirty_;
};

string ActionStr(Environment::ActionType);
```

Para incluir vuestra versión del agente, hacer:

(3) Incluir en **agent.h** la definición del resto de métodos usados en la implementación y en agent.cpp la implementación de dichos métodos más las funciones auxiliares.

# 4. Pasos del desarrollo de la práctica

## 4.3. Definición de estado (state)

```
private:
    int **mundo; // mapa de la habitación
    int posX, posY, Consumed_Energy, Pending_Dirty, Tam_X, Tam_Y;
    int last_action; // última acción realizada
    list<state>::iterator pos_padre; // Un iterador al estado padre
    double g, h, f; // evaluación del nodo g: costo actual, h: costo hasta un estado objetivo, f=g+h
    list<int> road; // lista que almacena la secuencia de acciones hasta el momento.
```

- Un mapa del estado de la habitación
- Tamaño de la habitación
- Posiciones X,Y de la aspiradora sobre la habitación
- Energía consumida hasta el momento y unidades de suciedad
- La última acción que llevo a este estado
- Un enlace al estado del que proviene.
- Valores de costo actual (g), costo hasta el objetivo (h) y la combinación (f)
- La lista de acciones realizadas hasta el momento

# 4. Pasos del desarrollo de la práctica

## 4.4. El fichero *agent.cpp*

```
Plan Agent::Think(const Environment &env, int option){
    state start(env);
    Plan plan;

    switch (option){
        case 0: //Agente Reactivo
            break;

        case 1: //Busqueda Anchura
            plan = Busqueda_Anchura(start);
            cout << "\n Longitud del Plan: " << plan.Get_Longitud_Plan()
            plan.Pinta_Plan();
            break;

        case 2: //Busqueda Profundidad
            plan = Busqueda_Profundidad(start);
            cout << "\n Longitud del Plan: " << plan.Get_Longitud_Plan()
            plan.Pinta_Plan();
            break;

        case 3: //Busqueda Profundidad
            plan = Escalada(start);
            cout << "\n Longitud del Plan: " << plan.Get_Longitud_Plan() << endl;
            plan.Pinta_Plan();
            break;

    }

    return plan;
}
```

El método **Think()** toma dos entradas:

- Un dato **env** que representa el estado actual del mundo
- Un dato **int** que indica el método de búsqueda seleccionado.

# 4. Pasos del desarrollo de la práctica

## 4.4. El fichero *agent.cpp*

```
Plan Agent::Think(const Environment &env, int option){  
    state start(env);  
    Plan plan;  
  
    switch (option){  
        case 0: //Agente Reactivo  
            break;  
        case 1: //Busqueda Anchura  
            plan = Busqueda_Anchura(start);  
            cout << "\n Longitud del Plan: " << plan.Get_Longitud_Plan() << endl;  
            plan.Pinta_Plan();  
            break;  
        case 2: //Busqueda Profundidad  
            plan = Busqueda_Profundidad(start);  
            cout << "\n Longitud del Plan: " << plan.Get_Longitud_Plan() << endl;  
            plan.Pinta_Plan();  
            break;  
        case 3: //Busqueda Profundidad  
            plan = Escalada(start);  
            cout << "\n Longitud del Plan: " << plan.Get_Longitud_Plan() << endl;  
            plan.Pinta_Plan();  
            break;  
    }  
    return plan;  
}
```

A partir de **env** se genera el estado inicial **start** que será el argumento de los distintos métodos de búsqueda.

# 4. Pasos del desarrollo de la práctica

## 4.4. El fichero *agent.cpp*

```
Plan Agent::Think(const Environment &env, int option){
    state start(env);
    Plan plan;

    switch (option){
        case 0: //Agente Reactivo
            break;
        case 1: //Busqueda Anchura
            plan = Busqueda_Anchura(start);
            cout << "\n Longitud del Plan: " << plan.Get_Longitud_Plan() << endl;
            plan.Pinta_Plan();
            break;
        case 2: //Busqueda Profundidad
            plan = Busqueda_Profundidad(start);
            cout << "\n Longitud del Plan: " << plan.Get_Longitud_Plan() << endl;
            plan.Pinta_Plan();
            break;
        case 3: //Busqueda Profundidad
            plan = Escalada(start);
            cout << "\n Longitud del Plan: " << plan.Get_Longitud_Plan() << endl;
            plan.Pinta_Plan();
            break;
    }

    return plan;
}
```

***option=0*** indica que la elección fue el agente reactivo.

El agente se invoca directamente desde el entorno, por eso aquí no aparece.

# 4. Pasos del desarrollo de la práctica

## 4.4. El fichero *agent.cpp*

### ***bool InsertarLista (lista, estado, it)***

- Inserta ***estado*** en la ***lista*** de estados y devuelve un enlace ***it*** a la posición donde se ha introducido. En este caso, retorna un valor de verdad.
- Si ya existe ***estado*** en la ***lista***, ***it*** contiene un enlace al que hay en lista. En este caso, retorna un falso.

```
bool InsertarLista(list<state> &lista, const state &st, list<state>::iterator &it){
    char ch;
    it= lista.begin();
    bool salida=false;
    while (it!=lista.end() and !(*it==st) )
        it++;
    if (it==lista.end()){
        lista.push_back(st);
        it = lista.end();
        it--;
        salida=true;
    }
    return salida;
}
```

class state {...}



# 4. Pasos del desarrollo de la práctica

## 4.5. Búsqueda en anchura

```
//
struct Comparar{
bool operator() (const pair<double,list<state>::iterator > &a, const pair<double,list<state>::iterator > &b){
    return (a.first > b.first );
}
};

// -----
// Búsqueda en Anchura
// -----
Plan Agent::Busqueda_Anchura(state start){
    Plan plan;
    typedef pair<double,list<state>::iterator > elementoCola;

    int last_level=0; // Indica el nivel del grafo por donde va la búsqueda
    int estados_evaluados = 0; // Indica el número de nodos evaluados
    state aux = start; // start es el estado inicial
    state sigActions[6], mejor_solucion; // para almacenar las siguientes acciones y la mejor solución
    int n_act;

    list<state> lista; // Lista que almacenara todos los estados
    list<state>::iterator p, padre; // Declara dos iteradores a la lista
    priority_queue<elementoCola,vector<elementoCola>, Comparar > cola; //Declaración de la cola con prioridad
    elementoCola la_siguiente; // Declara una variable del tipo almacenado en la cola con prioridad
    Insertar_en cola(lista,aux,padre); // Inserta el estado inicial en la lista y (padre) es un iterador a su posición.
```

Los elementos importantes son:

- Una lista con todos los estados generados

# 4. Pasos del desarrollo de la práctica

## 4.5. Búsqueda en anchura

```
//
struct Comparar{
bool operator() (const pair<double,list<state>::iterator > &a, const pair<double,list<state>::iterator > &b){
    return (a.first > b.first );
}
};

// -----
// Búsqueda en Anchura
// -----
Plan Agent::Busqueda_Anchura(state start){
    Plan plan;
    typedef pair<double,list<state>::iterator > elementoCola;

    int last_level=0; // Indica el nivel del grafo por donde va la búsqueda
    int estados_evaluados = 0; // Indica el número de nodos evaluados
    state aux = start; // start es el estado inicial
    state sigActions[6], mejor_solucion; // para almacenar las siguientes acciones y la mejor solución
    int n_act;

    list<state> lista; // Lista que almacenara todos los estados
    list<state>::iterator n_padre; // Declara dos iteradores a la lista
    priority_queue <elementoCola, vector<elementoCola>, Comparar > cola; //Declaración de la cola con prioridad
    elementoCola sigEstado; // Declara una variable del tipo almacenado en la cola con prioridad
    InsertarLista(lista,aux,padre); // Inserta el estado inicial en la lista y (padre) es un iterador a su posición.
```

Los elementos importantes son:

- Una lista con todos los estados generados
- Una cola con prioridad para guardar los estados por expandir

# 4. Pasos del desarrollo de la práctica

## 4.5. Búsqueda en anchura

```
//
struct Comparar{
bool operator()( const pair<double,list<state>::iterator > &a, const pair<double,list<state>::iterator > &b){
    return (a.first < b.first );
}
};

// -----
// Búsqueda en Anchura
// -----
Plan Agent::Busqueda_Anchura(state start){
    Plan plan;
    typedef pair<double,list<state>::iterator > elementoCola;

    int last_level=0; // Indica el nivel del grafo por donde va la búsqueda
    int estados_evaluados = 0; // Indica el número de nodos evaluados
    state aux = start; // start es el estado inicial
    state sigActions[6], mejor_solucion; // para almacenar las siguientes acciones y la mejor solución
    int n_act;

    list<state> lista; // Lista que almacenará todos los estados
    list<state>::iterator n_padre; // Declara dos iteradores a la lista
    priority_queue <elementoCola, vector<elementoCola>, Comparar > cola; //Declaración de la cola con prioridad
    elementoCola siguiente; // Declara una variable del tipo almacenado en la cola con prioridad

    InsertarLista(lista,aux,padre); // Inserta el estado inicial en la lista y (padre) es un iterador a su posición.
```

Los elementos importantes son:

- Una lista con todos los estados generados
- Una cola con prioridad para guardar los estados por expandir
  - El método necesario para ordenar la cola

# 4. Pasos del desarrollo de la práctica

## 4.5. Búsqueda en anchura

```
//
struct Comparar{
bool operator() (const pair<double,list<state>::iterator > &a, const pair<double,list<state>::iterator > &b){
    return (a.first > b.first );
}
};

// -----
// Búsqueda en Anchura
// -----
Plan Agent::Busqueda_Anchura(state start){
    Plan plan;
    typedef pair<double,list<state>::iterator > elementoCola;

    int last_level=0; // Indica el nivel del grafo por donde va la búsqueda
    int estados_evaluados = 0; // Indica el número de nodos evaluados
    state aux = start; // start es el estado inicial
    state soluciones[6], mejor_solucion; // para almacenar las siguientes acciones y la mejor solución
    int n;

    list<state> lista; // Lista que almacenara todos los estados
    list<state>::iterator p, padre; // Declara dos iteradores a la lista
    priority_queue<elementoCola,vector<elementoCola>, Comparar > cola; //Declaración de la cola con prioridad
    elementoCola siguiente; // Declara una variable del tipo almacenado en la cola con prioridad

    Inserta(lista,aux,padre); // Inserta el estado inicial en la lista y (padre) es un iterador a su posición.
```

Los elementos importantes son:

- Una lista con todos los estados generados
- Una cola con prioridad para guardar los estados por expandir
  - El método necesario para ordenar la cola
- El estado actual inicializado con el estado inicial

# 4. Pasos del desarrollo de la práctica

## 4.5. Búsqueda en anchura

```
//
struct Comparar{
bool operator() (const pair<double,list<state>::iterator > &a, const pair<double,list<state>::iterator > &b){
    return (a.first > b.first );
}
};

// -----
// Búsqueda en Anchura
// -----
Plan Agent::Busqueda_Anchura(state start){
    Plan plan;
    typedef pair<double,list<state>::iterator > elementoCola;

    int last_level=0; // Indica el nivel del grafo por donde va la búsqueda
    int estados_evaluados = 0; // Indica el número de nodos evaluados
    state aux = start; // start es el estado inicial
    state sigActions[6], mejor_solucion; // para almacenar las siguientes acciones y la mejor solución
    int i=0;

    list<state> lista; // Lista que almacenara todos los estados
    list<state>::iterator p, padre; // Declara dos iteradores a la lista
    priority_queue <elementoCola, vector<elementoCola>, Comparar > cola; //Declaración de la cola con prioridad
    elementoCola siguiente; // Declara una variable del tipo almacenado en la cola con prioridad

    Inserta(lista,aux,padre); // Inserta el estado inicial en la lista y (padre) es un iterador a su posición.
```

Los elementos importantes son:

- Una lista con todos los estados generados
- Una cola con prioridad para guardar los estados por expandir
  - El método necesario para ordenar la cola
- El estado actual inicializado con el estado inicial
- Un vector de estados para almacenar los descendientes de actual

# 4. Pasos del desarrollo de la práctica

## 4.5. Búsqueda en anchura

Se inserta el estado actual en la lista de estados.

```
InsertarLista(lista,aux,padre);  
  
while (!aux.Is_Solution()){  
    // Indica si ha incrementado el nivel del grafo por donde está buscando  
    if (aux.Get_g()!=last_level){  
        cout << "Level " << aux.Get_g() << endl;  
        last_level = aux.Get_g();  
    }  
  
    estados_evaluados++; // Incremento del número de estados evaluados  
  
    n_act=aux.Generate_New_States(sigActions); // Genera los nuevos estados a partir del estado (aux)  
  
    // Para cada estado generado, pone un enlace al estado que lo genero,  
    // lo inserta en la lista, y si no estaba ya en dicha lista, lo incluye en la cola con prioridad.  
    // El valor de prioridad en la lista lo da el método "Get_g()" que indica la energía consumida en dicho estado.  
    for (int i=0; i<n_act; i++){  
        sigActions[i].Put_Padre(padre);  
        if (InsertarLista(lista, sigActions[i], p) ){  
            double value = sigActions[i].Get_g();  
            cola.push( pair<double,list<state>::iterator > (value,p) );  
        }  
    }  
  
    // Saca el siguiente estado de la cola con prioridad.  
    padre = cola.top().second;  
    aux = *padre;  
    cola.pop();  
}  
  
// Llegados aquí ha encontrado un estado solución, e  
// incluye la solución en una variable de tipo plan.  
plan.AnadirPlan(aux.Copy_Road(), lista.size(), estados_evaluados );  
  
return plan; // Devuelve el plan  
}
```



# 4. Pasos del desarrollo de la práctica

## 4.5. Búsqueda en anchura

```
InsertarLista(lista,aux,padre); // Inserta el estado inicial en la lista y (padre) es un iterador

while (!aux.Is_Solution()){
    // Indica si ha incrementado el nivel del grafo por donde está buscando
    if (aux.Get_g() != last_level){
        cout << "Level " << aux.Get_g() << endl;
        last_level = aux.Get_g();
    }

    estados_evaluados++; // Incremento del número de estados evaluados

    n_act=aux.Generate_New_States(sigActions); // Genera los nuevos estados a partir del estado (aux)

    // Para cada estado generado, pone un enlace al estado que lo genero,
    // lo inserta en la lista, y si no estaba ya en dicha lista, lo incluye en la cola con prioridad.
    // El valor de prioridad en la lista lo da el método "Get_g()" que indica la energía consumida en dicho estado.
    for (int i=0; i<n_act; i++){
        sigActions[i].Put_Padre(padre);
        if (InsertarLista(lista, sigActions[i], p) ){
            double value = sigActions[i].Get_g();
            cola.push( pair<double,list<state>::iterator > (value,p) );
        }
    }

    // Saca el siguiente estado de la cola con prioridad.
    padre = cola.top().second;
    aux = *padre;
    cola.pop();
}

// Llegados aquí ha encontrado un estado solución, e
// incluye la solución en una variable de tipo plan.
plan.AnadirPlan(aux.Copy_Road(), lista.size(), estados_evaluados );

return plan; // Devuelve el plan
}
```

Mientras el estado actual no sea un estado solución.



# 4. Pasos del desarrollo de la práctica

## 4.5. Búsqueda en anchura

```
InsertarLista(lista,aux,padre); // Inserta el estado inicial en la lista y (padre) es un iterador a su posición.

while (!aux.Is_Solution()){
    // Indica si ha incrementado el nivel del grafo por donde está buscando
    if (aux.Get_g()!=last_level){
        cout << "Level " << aux.Get_g() << endl;
        last_level = aux.Get_g();
    }

    estados_evaluados++; // Incremento del número de estados evaluados

    n_act=aux.Generate_New_States(sigActions);

    // Para cada estado generado, pone un enlace al estado que lo genero,
    // lo inserta en la lista, y si no estaba ya en dicha lista, lo incluye en la cola con prioridad
    // El valor de prioridad en la lista lo da el método "Get_g()" que indica la energía consumida en dicho estado.
    for (int i=0; i<n_act; i++){
        sigActions[i].Put_Padre(padre);
        if (InsertarLista(lista, sigActions[i], p) ){
            double value = sigActions[i].Get_g();
            cola.push( pair<double,list<state>::iterator > (value,p) );
        }
    }

    // Saca el siguiente estado de la cola con prioridad.
    padre = cola.top().second;
    aux = *padre;
    cola.pop();
}

// Llegados aquí ha encontrado un estado solución, e
// incluye la solución en una variable de tipo plan.
plan.AnadirPlan(aux.Copy_Road(), lista.size(), estados_evaluados );

return plan; // Devuelve el plan
}
```

Se generan los descendientes del estado actual

# 4. Pasos del desarrollo de la práctica

## 4.5. Búsqueda en anchura

```
InsertarLista(lista,aux,padre); // Inserta el estado inicial en la lista y (padre) es un iterador a su posición.

while (!aux.Is_Solution()){
    // Indica si ha incrementado el nivel del grafo por donde está buscando
    if (aux.Get_g()!=last_level){
        cout << "Level " << aux.Get_g() << endl;
        last_level = aux.Get_g();
    }

    estados_evaluados++; // Incremento del número de estados evaluados

    n_act=aux.Generate_New_States(sigActions); // Genera los nuevos estados a partir del estado (aux)

    // Para cada estado generado, pone un enlace al estado que lo genero,
    // lo inserta en la lista, y si no estaba ya en dicha lista, lo incluye en la cola con prioridad
    // El valor de prioridad en la lista lo da el método "Get_g()" que indica la energía consumida
    for (int i=0; i<n_act; i++){
        sigActions[i].Put_Padre(padre);
        if (InsertarLista(lista, sigActions[i], p) ){
            double value = sigActions[i].Get_g();
            cola.push( pair<double,list<state>::iterator > (value,p) );
        }
    }

    // Sacar el siguiente estado de la cola con prioridad.
    padre = cola.top().second;
    aux = *padre;
    cola.pop();
}

// Llegados aquí ha encontrado un estado solución, e
// incluye la solución en una variable de tipo plan.
plan.AnadirPlan(aux.Copy_Road(), lista.size(), estados_evaluados );

return plan; // Devuelve el plan
}
```

Para cada estado  
descendiente

# 4. Pasos del desarrollo de la práctica

## 4.5. Búsqueda en anchura

```
InsertarLista(lista,aux,padre); // Inserta el estado inicial en la lista y (padre) es un iterador a su posición.

while (!aux.Is_Solution()){
    // Indica si ha incrementado el nivel del grafo por donde está buscando
    if (aux.Get_g()!=last_level){
        cout << "Level " << aux.Get_g() << endl;
        last_level = aux.Get_g();
    }

    estados_evaluados++; // Incremento del número de estados evaluados

    n_act=aux.Generate_New_States(sigActions); // Genera los nuevos estados a partir del estado (aux)

    // Para cada estado generado, pone un enlace al estado que lo genero,
    // lo inserta en la lista, y si no estaba ya en dicha lista, lo incluye en la cola con prioridad
    // El valor de prioridad en la lista lo da el método "Get_g()" que indica la energía consumida
    for (int i=0; i<n_act; i++){
        sigActions[i].Put_Padre(padre);
        if (InsertarLista(lista, sigActions[i], p) ){
            double value = sigActions[i].Get_g();
            cola.push( pair<double,list<state>::iterator > (value,p) );
        }
    }

    // Saca el siguiente estado de la cola con prioridad.
    padre = cola.top().second;
    aux = *padre;
    cola.pop();
}

// Llegados aquí ha encontrado un estado solución, e
// incluye la solución en una variable de tipo plan.
plan.AnadirPlan(aux.Copy_Road(), lista.size(), estados_evaluados );

return plan; // Devuelve el plan
}
```

Se enlaza con el estado actual

# 4. Pasos del desarrollo de la práctica

## 4.5. Búsqueda en anchura

```
InsertarLista(lista,aux,padre); // Inserta el estado inicial en la lista y (padre) es un iterador a su posición.

while (!aux.Is_Solution()){
    // Indica si ha incrementado el nivel del grafo por donde está buscando
    if (aux.Get_g() != last_level){
        cout << "Level " << aux.Get_g() << endl;
        last_level = aux.Get_g();
    }

    estados_evaluados++; // Incremento del número de estados evaluados

    n_act=aux.Generate_New_States(sigActions); // Genera los nuevos estados a partir del estado (aux)

    // Para cada estado generado, pone un enlace al estado que lo genero,
    // lo inserta en la lista, y si no estaba ya en dicha lista, lo incluye en la cola con prioridad
    // El valor de prioridad en la lista lo da el método "Get_g()" que indica la energía consumida
    for (int i=0; i<n_act; i++){
        sigActions[i].Put_Padre(padre);
        if (InsertarLista(lista, sigActions[i], p) ){
            double value = sigActions[i].Get_g();
            cola.push( pair<double,list<state>::iterator > (value,p) );
        }
    }

    // Sacar el siguiente estado de la cola con prioridad.
    padre = cola.top().second;
    aux = *padre;
    cola.pop();
}

// Llegados aquí ha encontrado un estado solución, e
// incluye la solución en una variable de tipo plan.
plan.AnadirPlan(aux.Copy_Road(), lista.size(), estados_evaluados );

return plan; // Devuelve el plan
}
```

Se inserta en la lista de estados si no está

# 4. Pasos del desarrollo de la práctica

## 4.5. Búsqueda en anchura

```
InsertarLista(lista,aux,padre); // Inserta el estado inicial en la lista y (padre) es un iterador a su posición.

while (!aux.Is_Solution()){
    // Indica si ha incrementado el nivel del grafo por donde está buscando
    if (aux.Get_g()!=last_level){
        cout << "Level " << aux.Get_g() << endl;
        last_level = aux.Get_g();
    }

    estados_evaluados++; // Incremento del número de estados evaluados

    n_act=aux.Generate_New_States(sigActions); // Genera los nuevos estados a partir del estado

    // Para cada estado generado, pone un enlace al estado que lo genero,
    // lo inserta en la lista, y si no estaba ya en dicha lista, lo incluye en la cola con prioridad
    // El valor de prioridad en la lista lo da el método "Get_g()" que indica la energía consumida
    for (int i=0; i<n_act; i++){
        sigActions[i].Put_Padre(padre);
        if (InsertarLista(lista, sigActions[i], p) ){
            double value = sigActions[i].Get_g();
            cola.push( pair<double,list<state>::iterator > (value,p) );
        }
    }

    // Saca el siguiente estado de la cola con prioridad.
    padre = cola.top().second;
    aux = *padre;
    cola.pop();
}

// Llegados aquí ha encontrado un estado solución, e
// incluye la solución en una variable de tipo plan.
plan.AnadirPlan(aux.Copy_Road(), lista.size(), estados_evaluados );

return plan; // Devuelve el plan
}
```

Se inserta en la cola de estados aún por expandir.

Se inserta el par costo en energía consumido para llegar a este estado junto con un enlace a este estado en la lista de estados

# 4. Pasos del desarrollo de la práctica

## 4.5. Búsqueda en anchura

```
InsertarLista(lista,aux,padre); // Inserta el estado inicial en la lista y (padre) es un iterador a su posición.

while (!aux.Is_Solution()){
    // Indica si ha incrementado el nivel del grafo por donde está buscando
    if (aux.Get_g()!=last_level){
        cout << "Level " << aux.Get_g() << endl;
        last_level = aux.Get_g();
    }

    estados_evaluados++; // Incremento del número de estados evaluados

    n_act=aux.Generate_New_States(sigActions); // Genera los nuevos estados a partir del estado (aux)

    // Para cada estado generado, pone un enlace al estado que lo genero,
    // lo inserta en la lista, y si no estaba ya en dicha lista, lo incluye en la cola con prioridad.
    // El valor de prioridad en la lista lo da el método "Get_g()" que indica la energía consumida en dicho estado.
    for (int i=0; i<n_act; i++){
        sigActions[i].Put_Padre(padre);
        if (InsertarLista(lista, sigActions[i], p) ){
            double value = sigActions[i].Get_g();
            cola.push( pair<double,list<state>::iterator > (value,p) );
        }
    }

    // Saca el siguiente estado de la cola con prioridad.
    padre = cola.top().second;
    aux = *padre;
    cola.pop();
}

// Llegados aquí ha encontrado un estado solución, e
// incluye la solución en una variable de tipo plan.
plan.AnadirPlan(aux.Copy_Road(), lista.size(), estados_evaluados );

return plan; // Devuelve el plan
}
```

Se saca el siguiente estado de la cola y se asigna al estado actual



# 4. Pasos del desarrollo de la práctica

## 4.5. Búsqueda en anchura

```
InsertarLista(lista,aux,padre); // Inserta el estado inicial en la lista y (padre) es un iterador a su posición.

while (!aux.Is_Solution()){
    // Indica si ha incrementado el nivel del grafo por donde está buscando
    if (aux.Get_g()!=last_level){
        cout << "Level " << aux.Get_g() << endl;
        last_level = aux.Get_g();
    }

    estados_evaluados++; // Incremento del número de estados evaluados

    n_act=aux.Generate_New_States(sigActions); // Genera los nuevos estados a partir del estado (aux)

    // Para cada estado generado, pone un enlace al estado que lo genero,
    // lo inserta en la lista, y si no estaba ya en dicha lista, lo incluye en la cola con prioridad.
    // El valor de prioridad en la lista lo da el método "Get_g()" que indica la energía consumida en dicho estado.
    for (int i=0; i<n_act; i++){
        sigActions[i].Put_Padre(padre);
        if (InsertarLista(lista, sigActions[i], p) ){
            double value = sigActions[i].Get_g();
            cola.push( pair<double,list<state>::iterator > (value,p) );
        }
    }

    // Sacar el siguiente estado de la cola con prioridad.
    padre = cola.top().second;
    aux = *padre;
    cola.pop();
}

// Llegados aquí ha encontrado un estado solución, e
// incluye la solución en una variable de tipo plan.
plan.AnadirPlan(aux.Copy_Road(), lista.size(), estados_evaluados );

return plan; // Devuelve el plan
}
```

Cuando se encuentra un estado solución, se añade a un objeto de tipo **Plan**



# 4. Pasos del desarrollo de la práctica

## 4.5. Búsqueda en anchura

```
InsertarLista(lista,aux,padre); // Inserta el estado inicial en la lista y (padre) es un iterador a su posición.

while (!aux.Is_Solution()){
    // Indica si ha incrementado el nivel del grafo por donde está buscando
    if (aux.Get_g()!=last_level){
        cout << "Level " << aux.Get_g() << endl;
        last_level = aux.Get_g();
    }

    estados_evaluados++; // Incremento del número de estados evaluados

    n_act=aux.Generate_New_States(sigActions); // Genera los nuevos estados a partir del estado (aux)

    // Para cada estado generado, pone un enlace al estado que lo genero,
    // lo inserta en la lista, y si no estaba ya en dicha lista, lo incluye en la cola con prioridad.
    // El valor de prioridad en la lista lo da el método "Get_g()" que indica la energía consumida en dicho estado.
    for (int i=0; i<n_act; i++){
        sigActions[i].Put_Padre(padre);
        if (InsertarLista(lista, sigActions[i], p) ){
            double value = sigActions[i].Get_g();
            cola.push( pair<double,list<state>::iterator > (value,p) );
        }
    }

    // Sacar el siguiente estado de la cola con prioridad.
    padre = cola.top().second;
    aux = *padre;
    cola.pop();
}

// Llegados aquí ha encontrado un estado solución, e
// incluye la solución en una variable de tipo plan.
plan.AnadirPlan(aux.Copy_Road(), lista.size(), estados_evaluados );

return plan; }
```

Se devuelve la  
secuencia de acciones

# 4. Pasos del desarrollo de la práctica

## 4.6. Búsqueda en profundidad

```
// -----  
Plan Agent::Busqueda_Profundidad(state start){  
    Plan plan;  
    state aux = start;  
    int estados_evaluados = 0;  
  
    // IMPLEMENTA AQUÍ LA BUSQUEDA EN PROFUNDIDAD  
  
    //plan.AnadirPlan(aux.Copy_Road(), lista.size(), estados_evaluados );  
  
    return plan;  
}
```

Este es uno de los dos métodos que tenéis que implementar.

Podéis utilizar cualquier variante de la búsqueda en profundidad vista en clase.

Se pueden utilizar los recursos definidos para la implementación de la búsqueda en anchura.

# 4. Pasos del desarrollo de la práctica

## 4.7. Método de Escalada

```
double Heuristica(const state &estado){  
    // Suciedad que queda pendiente en la habitacion  
    return estado.Get_Pending_Dirty();  
}  
  
// -----  
  
Plan Agent::Escalada(state start){  
    Plan plan;  
    state aux = start;  
    int estados_expandidos=0, estados_evaluados=0;  
  
    // IMPLEMENTA AQUÍ EL MÉTODO DE ESCALADA  
  
    plan.AnadirPlan(aux.Copy_Road(), estados_expandidos, estados_evaluados );  
  
    return plan;  
}
```

Este es el otro método que tenéis que implementar.

Podéis utilizar cualquier variante de las técnicas de escalada vista en clase.

# 4. Pasos del desarrollo de la práctica

## 4.7. Método de Escalada

```
double Heuristica(const state &estado){  
    // Suciedad que queda pendiente en la habitacion  
    return estado.Get_Pending_Dirty();  
}  
  
// -----  
  
Plan Agent::Escalada(state start){  
    Plan plan;  
    state aux = start;  
    int estados_expandidos=0, estados_evaluados=0;  
  
    // IMPLEMENTA AQUÍ EL MÉTODO DE ESCALADA  
  
    plan.AnadirPlan(aux.Copy_Road(), estados_expandidos, estados_evaluados );  
  
    return plan;  
}
```

Es un método de búsqueda heurística.

Hay que definir una función bien informada adecuada al método que se implemente.

Aquí se muestra una posible función heurística.

Este es el otro método que tenéis que implementar.

Podéis utilizar cualquier variante de las técnicas de escalada vista en clase.

# Índice

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

Un único archivo comprimido (zip o rar) que llamado “***practica2***” contenga dos carpetas:

- Una de las carpetas con la memoria de la práctica (en formato pdf)
- La otra carpeta con los archivos “***agent.cpp***” y “***agent.h***” con las implementaciones requeridas.

***No ficheros ejecutables***



## 5. Evaluación de la práctica

¿Qué debe contener la memoria de la práctica?

1. Análisis del problema
2. Descripción de la solución propuesta
3. Tabla con los resultados obtenidos sobre los distintos mapas.

***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 3 puntos, NO APTO implica tener un **0** en esta práctica.
3. Evaluación de cada método de búsqueda
  - se evalúa **de 0 a 4**.
  - el valor concreto es el resultado de interpolar entre la mejor y la peor solución encontrada en cada uno de los algoritmos.

# 5. Evaluación de la práctica

## ¿Cómo se evalúa?

Sobre las implementaciones desarrolladas (*hasta 4 puntos*):

- Será evaluada sobre un mapa distinto a los aportados junto con el material de la práctica
- En el caso de la búsqueda en profundidad, obtendrá un máximo de 2 puntos quién obtenga la solución más cercana al óptimo haciendo menos de 2000 evaluaciones. Obtendrá 1 punto el que obtenga la solución más lejana. 0 puntos quién no obtenga solución alguna, y un valor entre 1 y 2 proporcional al mejor y peor valor el resto.
- En el caso del método de escalada se aplicará el mismo criterio anterior.

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