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 2

Métodos de Búsqueda

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

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

DECSAI

1. Introducción

1.1. Motivación

La segunda práctica de la asignatura ***Inteligencia Artificial*** consiste en el diseño e implementación de algunas técnicas de búsqueda (sin y con información) y al igual que en la práctica anterior, se trabajará con un simulador software que implementará una aspiradora inteligente basada en los ejemplos del libro *Stuart Russell, Peter Norvig, "Inteligencia Artificial: Un enfoque Moderno", Prentice Hall, Segunda Edición, 2004*. Este simulador que utilizaremos fue desarrollado por el profesor Tsung-Che Chiang de la NTNU (Norwegian University of Science and Technology, Taiwan) y en este caso, trabajaremos con una versión modificada para definir agentes deliberativos. Utilizaremos las técnicas de búsqueda estudiadas en los temas 2 y 3 de la asignatura y estudiaremos su comportamiento en relación a los agentes puramente reactivos.

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 **AgentMod**, 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.

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

3. Objetivo de la práctica

La práctica tiene como objetivo diseñar e implementar agentes deliberativos que resuelvan 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 **NO** es dinámico (una casilla que se haya limpiado **NO** puede volver a ensuciarse). Para ello, haremos las siguientes suposiciones:

- El entorno puede representarse en una matriz de tamaño máximo TxT, 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 conocida a priori por el agente.
- La suciedad de cada casilla se mide en números enteros no negativos (0, 1, 2, 3, …).
- 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 1 punto.
- 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 el siguiente objetivo: **limpiar todas las casillas de la habitación con el menor coste de energía.**

Las técnicas a usar son las siguientes:

- Un agente reactivo.
- Una técnica de búsqueda en anchura.
- Una técnica de búsqueda en profundidad.
- Una técnica basada en un método de escalada (búsqueda con información).

El agente reactivo a incluir en esta práctica es el desarrollado por el alumno en la práctica anterior. Así, que cada alumno debe adaptarlo a la nueva implementación. La única diferencia fundamental en este caso, es que debe considerar que los mapas no son de tamaño **10x10**, si no que pueden tener un tamaño superior, siendo el máximo de **80x80**.

La técnica de búsqueda en anchura viene ya implementada en el software que se adjunta con la práctica. Se incluye su implementación para que sirva de guía al alumno para realizar los otros dos métodos de búsqueda.

La técnica de búsqueda en profundidad tiene que ser implementada y puede utilizar cualquiera de las variantes descritas en el temario de teoría.

El método de escalada también debe ser implementado, y al igual que en el caso anterior, cualquiera de los métodos dados en clase de teoría será válido.

4. Instalación y descripción del simulador

4.1. Instalación del simulador

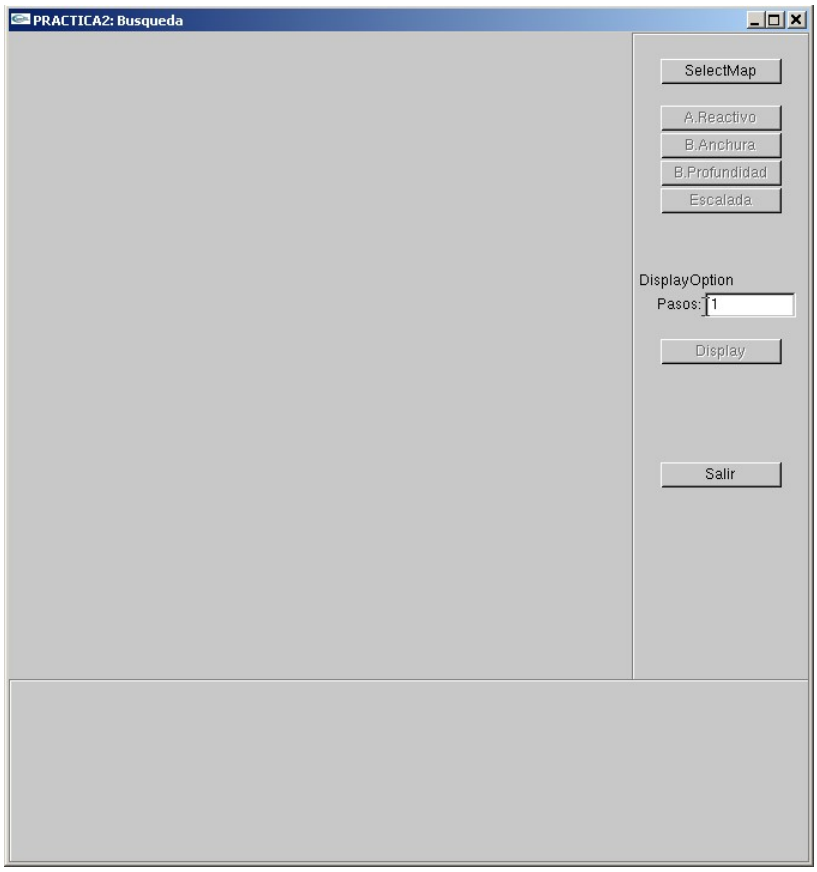
El simulador **AgentMod** nos permitirá implementar el comportamiento del agente y visualizar las acciones en una interfaz de usuario mediante ventanas. Para instalarlo, siga estos pasos:

1. Descargue el fichero **AgentMod.rar** (o **AgentMod.zip**) desde la web de la asignatura, y cópielo su carpeta personal dedicada a las prácticas de la asignatura de ***Inteligencia Artificial***. Supongamos, para los siguientes pasos, que esta carpeta se denomina "**U:\IA\practica2**".
2. Desempaquete el fichero en la raíz de esta carpeta.
3. Ya está instalado el simulador. A continuación, el siguiente paso es compilar el proyecto "**agent.cbp"** en el entorno **CodeBlocks**.

4.2. Ejecución del simulador

Tomamos la opción de abrir un proyecto existente y elegimos el proyecto ***"agent.cbp"*** y lo compilamos.

Una vez compilado el proyecto del simulador, para ejecutarlo pulsaremos sobre la opción "**Run**" del menú **"Build**" (alternativamente, también podemos hacer doble clic sobre el programa **Agent.exe** generado en la carpeta del proyecto tras su compilación). Aparecerá una ventana como la que se muestra en la figura siguiente.

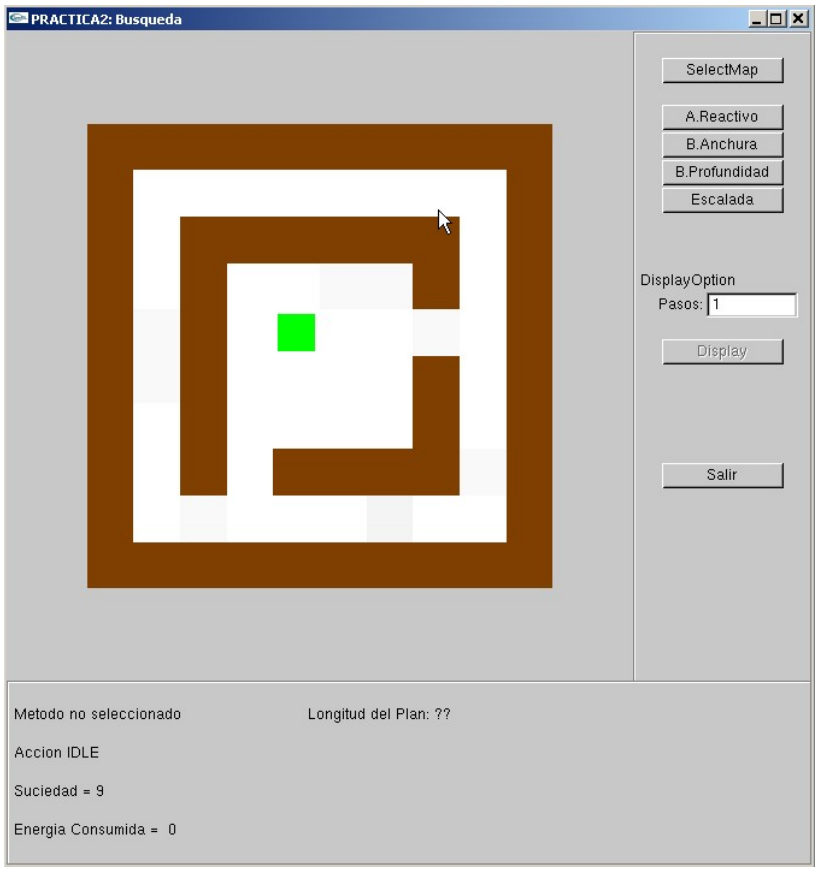


En esta ventana, la opción que nos interesa se encuentra en el botón "**Select Map**", que nos permite seleccionar un mapa del entorno. Por defecto, el simulador trae un único mapa "**mapa10a.map**".

Tras cargar el mapa "**mapa10a.map**", la vista del simulador cambia mostrando el mundo del agente, la posición donde comienza e información relativa al mapa. Cabe destacar sobre esta información, el item "Suciedad", que aparece en la parte inferior de la ventana, indicando el número de unidades de suciedad que hay en ese mapa.

La carga del mapa además activa las siguientes 4 opciones:

- **A.Reactivo:** Elegir esta opción implica que vamos a resolver el problema de limpiar la habitación utilizando un *agente reactivo*, en concreto, el realizado por el alumno en la práctica anterior. En el software que se proporciona, el comportamiento por defecto del agente es el mismo que se incluyó en la práctica 1. Más adelante se describirá como incluir vuestra versión en el simulador.

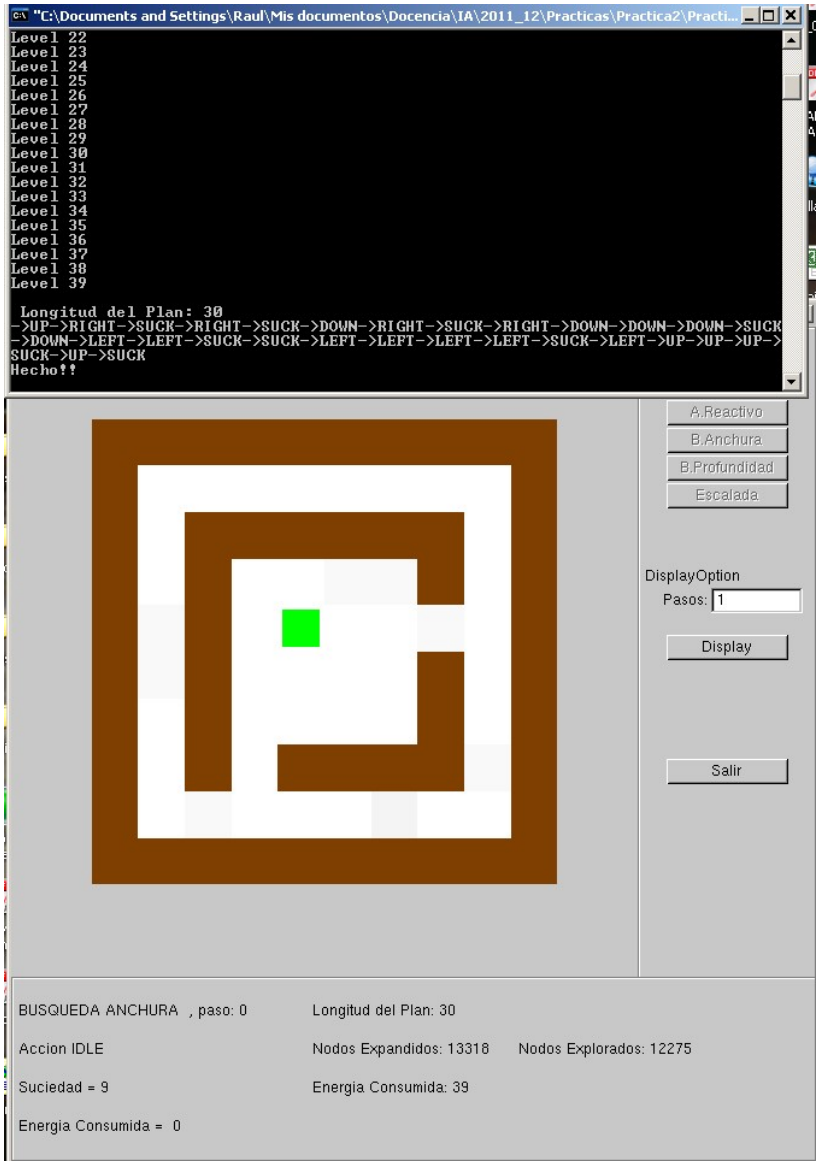


- **B.Anchura:** Pulsar esta opción, implica que se va a utilizar una búsqueda en anchura para encontrar la forma más eficiente para resolver el problema. En el software se encuentra ya implementada una variación de este tipo de búsqueda conocida como ***Búsqueda con costo***.

- **B.Profundidad:** Se selecciona una técnica de búsqueda en profundidad como solución a limpiar la suciedad de la habitación.

- **Escalada:** En este caso, se opta por una solución heurística basada en una técnica de escalada como herramienta para la solución de limpieza del mapa.

Supongamos que decidimos resolver el problema utilizando un técnica de búsqueda en anchura, y por consiguiente pulsamos el botón ***"B.Anchura"***, en este caso nos aparecerá algo parecido a lo que muestra la siguiente figura:



En la parte superior, se muestra la ventana del terminal que nos muestra cierta información sobre como se está realizando el proceso de búsqueda, en este caso, va apareciendo el nivel del grafo por donde va la búsqueda. El proceso termina devolviendo la longitud y la descripción de dicho plan.

En la parte inferior/derecha se muestran las características de la solución encontrada. En el ejemplo anterior, se ha encontrado una solución que consume **39 unidades de energía** mediante un plan de longitud **30 pasos**. Además, aparece información sobre los nodos expandidos y explorados durante el proceso de búsqueda.

Una vez encontrada la solución por el método seleccionado, el simulador se pone en modo ***"Display"***, es decir, mostrar sobre el entorno la ejecución del plan encontrado. Al igual que en la práctica anterior, usaremos ***"Pasos:"*** para dar valor al número de acciones consecutivas que deseamos ver y pulsaremos el botón ***"Display"***, para ver esa secuencia.

Para ver el comportamiento de los diferentes métodos sobre los diferentes mapas, no tenemos más que repetir el proceso que aquí se describe, cambiando o bien el mapa o bien el método solución.

5. Pasos para construir la práctica

5.1. Incluir el agente reactivo

El ***agent.h*** que se proporciona es el siguiente:

```
#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_,
         dirty_;

};

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

Dentro de la definición de la clase, debéis incluir vuestra definición de "agent.h" en el archivo anterior, teniendo en cuenta las siguientes consideraciones:

- El método que en la práctica anterior se llamaba ***ActionType Think()***, que contiene el comportamiento del agente, ahora debe ser renombrado como ***Environment::ActionType AgenteReactivo()***.

- El método ***void Perceive(const Environment &env)*** es igual que el de la práctica 1.

- Se redefine el método ***Think()*** con una nueva parametrización y será el encargado de invocar a los distintos métodos de búsqueda a partir del valor de ***option***

- Deben aparecer los nuevos métodos que implementan las búsquedas:

    - ***Plan Búsqueda_Anchura (state start)***

    - ***Plan Búsqueda_Profundidad (state start)***

    - ***Plan Escalada (state start)***

- Manteniendo lo anterior, incorporar todos los datos miembros de la clase, así como los constructores y destructores que hubieseis definido y los métodos adicionales necesarios para el funcionamiento del agente.

Como se puede observar, aparecen dos nuevas clases en esta implementación, la clase ***Plan*** y la clase ***state***. La clase ***Plan*** se encarga de almacenar la salida obtenida por los métodos de búsqueda y los objetos de esta clase serán los encargados de interactuar con el entorno gráfico. **ESTA CLASE NO PODRÁ SER MODIFICADA**.

Por otro lado, la clase ***state*** será el elemento utilizado para representa una instancia en la búsqueda en el espacio de estados. La definición realizada aquí es muy general, y para algunas implementaciones habrá datos miembro que no serán de utilidad, pero están incluidos para contemplar el mayor número posible de implementaciones diferentes.

```
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.
```

A diferencia de la anterior, el alumno es libre de alterar la definición de esta clase a su gusto y hacer una representación diferente del concepto de estado en este problema. **IMPORTANTE: si optas por**

cambiar esta clase, debes adaptar los métodos que están declarados en la clase en base a tu nueva definición.

5.2. El fichero agent.cpp

El método de búsqueda en anchura se encuentra ya implementado en la versión que se os proporciona y debe ser utilizado como referencia para construir los otros dos métodos, objeto de la práctica.

Antes de describir la implementación de este algoritmo, veremos la estructura del fichero agent.cpp y algunas funciones útiles para la realización de la práctica.

```
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;
}
```

El método ***Think()*** es el que juega el papel de interfaz con el simulador que se comunica con él a través de dos parámetros ***env*** que informa del estado actual del mundo y ***option*** que determina que método de búsqueda se va a realizar. Lo que devuelve al simulador este método es un objeto de la clase ***Plan***, es decir, la secuencia de acciones para resolver el problema.

Lo primero que hace este método es transformar ***env*** en un objeto de la clase ***state*** que se considerará como el estado inicial y será el parámetro ***start*** requerido por los métodos de búsqueda.

Como se puede observar, option=0 designa el uso del agente reactivo para resolver el problema. En este caso, el simulador no invoca a esta función, si no que directamente invoca al método

Environment::ActionType AgenteReactivo() que devuelve la siguiente acción a realizar (no un plan completo).

Además, del método ***Think()***, el fichero ***agent.cpp*** incluye la función ***InsertarLista()***:

```
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;
}
```

Esta función es útil para gestionar la lista de estados que se está generando. Toma como entrada una lista de estados (***lista***) y un nuevo estado (***st***), y devuelve un iterador (***it***) a la posición donde se ha insertado el nuevo estado en la lista de estados. La función devuelve también un valor lógico con la siguiente interpretación:

> si ***st*** no está en la lista, lo inserta en la misma, en ***it*** está la posición de insercción y devuelve ***true***,
> en otro caso, ***st*** no se incluye en la lista por estar ya, ***it*** contiene el iterador al estado equivalente a ***st*** que esta en ***lista*** y se devuelve ***false***.

Por último, y una vez que adaptéis el agente reactivo, aparece también vuestra implementación en este fichero. A modo de ejemplo, en la versión actual aparece la versión por defecto ya conocida de este método que se proporcionaba en la práctica anterior.

```
// -----------------------------------------------------------
Environment::ActionType Agent::AgenteReactivo()
{
    if (dirty_) return Environment::actSUCK;
    else return static_cast<Environment::ActionType> (rand()%4);

}
```

5.3. Búsqueda en Anchura (*Búsqueda con costo*)

A continuación explicaremos la implementación incluida para resolver el problema usando 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 );
}
};

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

  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 <elemento_cola, vector<elemento_cola>, Comparar > cola; //Declaración de la cola con prioridad

  elemento_cola 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.
```

En la figura de arriba se muestra los datos utilizados para la búsqueda. De entre ellos destacamos:

- ***lista*** que es la lista de estados que almacenará todos los estados que se van generando durante el proceso de búsqueda.
- ***cola*** que es la lista de estados que están pendientes de expandirse. Para un problema donde todas las acciones tengan el mismo costo asociado sería suficiente con una cola normal. En este caso, como hay acciones que consumen más energía que otras, definimos una cola con prioridad, para sacar en cada iteración la de menor costo. De esta manera, estamos seguros que la primera solución que encontremos es una solución óptima. La cola con prioridad definida es de pares (***double***, ***iterador***). En la parte de ***double***, se almacena el costo que se ha necesitado para llegar a dicho estado, mientras que en la parte ***iterador*** guardamos un enlace con la posición que ocupa dicho estado en la lista de estados anterior. La razón de este ***iterador*** es no almacenar dos veces el mismo estado.
- ***aux*** es un objeto de tipo estado que almacena el estado actual por donde se encuentra la búsqueda. Inicialmente toma el valor de ***start*** (el estado inicial), y más adelante irá tomando el valor del primero de la cola.
- ***sigActions*** es un vector de estados que almacena los descendientes del estado actual.

El proceso comienza insertando en la lista el nodo inicial, y mientras no se encuentre un estado solución, es decir, un estado con **suciedad=0,** se hace lo siguiente:

- Se exploran los descendientes del estado actual.
- Para cada descendiente se mira si no está en la lista de estados previamente generados y se añade a la lista. Si se añade a la lista, también se inserta en la cola.
- Se extrae el primer el elemento de la cola y se repite el proceso.

```
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
}
```

Justo ese proceso, es el que se reproduce en este código, donde

- ***n_act = aux.Generate_New_States(sigActions),*** obtiene los descendientes del estado actual ***aux*** y los almacena en el vector de estados ***sigActions***, indicando ***n_act*** el número de descendientes válidos del estado.

- Para cada uno de los descendientes,
  - Se enlaza con su estado padre, que es el estado actual.
  - Se inserta en la lista de estados si no está, se toma su valor gasto de energía con ***Get_g()*** y se añade a la cola su costo y un iterador a su posición en la lista.

- Se saca el siguiente nodo de la cola y se considera como el nodo actual.

El proceso termina cuando se encuentra un estado solución, y el camino se añade a un objeto de la variable Plan que almacena la secuencia de acciones.

5.4. Búsqueda en Profundidad

Mientras que la búsqueda en anchura, implementada anteriormente, asegura encontrar una solución óptima, los métodos en profundidad, en principio, no pueden asegurar el obtener ese óptimo. Compensan esa desventaja con la rapidez en encontrar una solución.

Este es el primer método que se pide realizar en la práctica. Consiste en implementar un método de búsqueda en profundidad de entre las distintas variantes que se han visto en clase. El objetivo es encontrar una variante que obtengan un buen valor de energía.

En el fichero agent.cpp se encuentra el código que se muestra debajo.

```
// -----------------------------------------------------------
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;
}
```

Se pide completar el método haciendo uso, bien de las herramientas ya definidas para la búsqueda en anchura en la clase ***state***, bien de las posibilidades que ofrece la ***stl*** de ***C++*** o bien de las que el propio alumno quiera desarrollar.

5.5. Método de Escalada

Hay problemas en los que los métodos de búsqueda sin información no son aplicables para encontrar una solución por lo extraordinariamente extenso del espacio de búsqueda. En estos casos, se aplican técnicas que usan información sobre el problema (heurística) para encontrar soluciones óptimas o semióptimas.

Una de las técnicas más conocidas en este ámbito, son las llamadas técnicas de escalada. En esta práctica se pide implementar alguna de las vistas en clase que sea adecuada para resolver el problema de limpiar la habitación.

```
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;
}
```

En la figura anterior, se muestra la parte del fichero agent.cpp destinada a la implementación del método de escalada. Como hemos dicho anteriormente, estas técnicas usan información sobre el problema. Dicha información se expresa mendiante una función heurística. En este caso, se ha definido una heurística muy simple, que es el número de unidades de suciedad que aún queda en la habitación. Esta heurística es válida para una primera propuesta de método de escalada, sin embargo, es una heurística poco informada (lo descubriréis por su facilidad para quedarse en óptimos locales).

Se pide por tanto en este apartado, implementar un método de escalada y definir una función heurística que permita encontrar soluciones semióptimas mejores, es decir, soluciones no óptimas pero que estén cerca de las soluciones óptimas.

6. Evaluación y entrega de prácticas

La **calificación final** de la práctica se calculará de la siguiente forma:

- Se entregará una memoria de prácticas (ver apartado 6.1 de este guión) al finalizar las tareas a realizar. La fecha límite de la entrega de la memoria será comunicada con suficiente antelación por el profesor de prácticas en clase, y publicada en la página web de la asignatura.
- Se realizará una defensa de la práctica. La fecha de dicha defensa se publicará con suficiente antelación en la web de la asignatura para cada grupo/alumno, y será comunicada también en clase de prácticas por el profesor. El objetivo de esta defensa es verificar que la memoria

entregada ha sido realizada por el alumno. Por tanto, esta defensa requerirá de la ejecución del simulador con los comportamientos realizados por los alumnos, en clase de prácticas, y de la respuesta a cuestiones del trabajo realizado. La calificación de la defensa será **APTO** o **NO APTO**. Una calificación **NO APTO** en la defensa implica el suspenso con calificación **0** en la práctica. Una calificación **APTO** permite al alumno obtener la calificación según los criterios explicados en el punto siguiente.

- La práctica se califica numéricamente de **0 a 10**. Se evaluará como la suma de los siguientes criterios:
  - La memoria de prácticas se evalúa de **0 a 3.**
  - Las cuestiones realizadas por el profesor durante la defensa de prácticas y correctamente respondidas por el alumno se evalúan de **0 a 3**.
  - La eficacia de la búsqueda en profundidad se evalúa de **0 a 2** puntos. El alumno que obtenga la mejor solución (el óptimo en consumo de energía o el semióptimo más cercano en un mapa específico para la evaluación con menos de 2000 estados evaluados) obtendrá una calificación máxima en este apartado de 2 puntos. El alumno que obtenga la peor solución obtendrá una calificación mínima de 1. 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. Si no encontraron solución la calificación es 0.
  - La eficacia del método de escalada se evalúa de **0 a 2** puntos. El alumno que obtenga la mejor solución (el óptimo en consumo de energía o el semióptimo más cercano en un mapa específico para la evaluación) obtendrá una calificación máxima en este apartado de 2 puntos. El alumno que obtenga la peor solución obtendrá una calificación mínima de 1. 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. Si no encontraron solución la calificación es 0.

- 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**, **agent.h y si el alumno lo ve necesario state.cpp y state.h**) con los métodos de búsqueda en profundidad y una técnica de escalada. 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
2. Descripción de la solución planteada (para la búsqueda en profundidad y el método de escalada, junto con la función heurística diseñada para este último método)
3. Resultados obtenidos por la búsqueda en profundidad y la técnica de escalada en los distintos mapas.