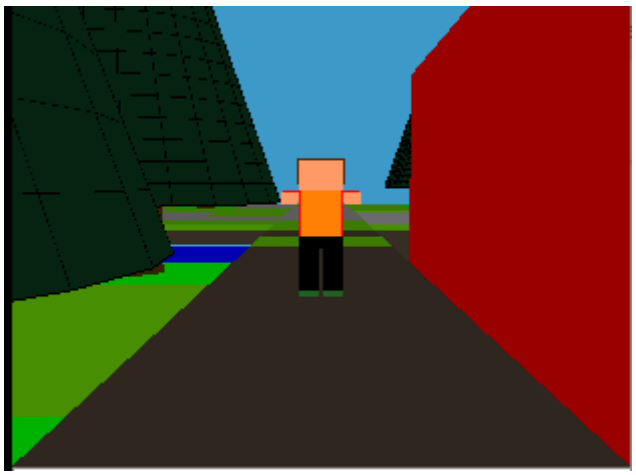
INTELIGENCIA ARTIFICIAL

E.T.S. de Ingenierías Informática y de Telecomunicación

Tutorial: Práctica 2 (Parte 2)

Agentes Reactivos/Deliberativos



DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN E INTELIGENCIA ARTIFICIAL

UNIVERSIDAD DE GRANADA

Curso 2024-2025

1. Introducción

El objetivo de la práctica es definir un comportamiento reactivo/deliberativo para un par de agentes cuya misión es desplazarse por un mapa. En este tutorial se pretende introducir al estudiante en la dinámica que debería seguir para la construcción de los niveles 2, 3 y 4 de la práctica.

En la primera parte de este tutorial se desarrollaron comportamientos reactivos. Esta segunda parte se sitúa como la continuación de la misma y tomando el software por donde se quedó. Entendemos que el estudiante habrá incluido más módulos para completar satisfactoriamente los niveles 0 y 1.

La mayoría de los estudiantes habrán realizado el tutorial 2 parte 1 y estarán familiarizados con las funciones que aquí se incluyen y que allí se definieron. Para aquellos estudiantes que no, recomendamos que vuelvan a mirar dicho tutorial.

Este tutorial 2 parte 2 se centra en realizar la implementación de un nivel ficticio de la práctica, construir el algoritmo de búsqueda en anchura para el agente `Auxiliar`, con la intención de servir de base para la construcción de los niveles 2 y 3 reales de la práctica y ayudar a definir otros algoritmos de búsqueda que el estudiante quiera emplear para el nivel 4.

2. Estructura general del agente

En la descripción de esta práctica se indica que hay dos agentes involucrados en los procesos de desplazamiento por el mapa, el agente `Rescatador` y el agente `Auxiliar`, aunque en este tutorial trabajaremos solo con el agente Auxiliar. Además, se continuará con el diseño que ya se señaló en la parte 1 de definir un módulo independiente para resolver cada nivel de la práctica. Al ser un nivel ficticio el que vamos a resolver aquí (la búsqueda en anchura para el agente `Auxiliar`), crearemos un nuevo método para la clase `ComportamientoAuxiliar` que llamaremos:

```
Action ComportamientoAuxiliarNivel_E(Sensores sensores)
```

que toma como parámetro de entrada el valor de los sensores del agente en el instante actual, y devuelve una acción. Esta forma de actuar se asemeja al comportamiento de un agente reactivo en el sentido que en cada iteración devuelve una única acción. Como en esta práctica parte se quieren evaluar comportamientos deliberativos necesitaremos adaptar el comportamiento de un agente deliberativo dentro de este esquema concreto. Lo que el agente hará será calcular un plan (en su versión más genérica de esta práctica), una secuencia de movimientos que permita trasladar al agente desde la casilla donde se encuentre a la casilla destino. Dicho plan se almacenará en una variable de estado. El método `ComportamientoAuxiliarNivel_E` en cada iteración de la simulación irá mandando ejecutar la siguiente acción de ese plan hasta que el agente llegue a la casilla destino.

Vamos a implementar ese comportamiento básico sobre este método que ejecutará la siguiente acción del plan, si el plan existe, y si no existe, se calculará el plan. Para ello vamos a definir dos variables de estado: `plan` que almacenará el plan a ejecutar y que será de tipo `list<Action>` en la parte privada de la clase `ComportamientoAuxiliar` y `hayPlan` que nos dirá si ya se ha calculado un plan para llegar a la casilla objetivo que definiremos en la misma parte privada, pero en este caso su tipo asociado será `bool`. En el constructor de clase (en el que se usa para los niveles 2 y 3) la variable `hayPlan` se inicializará a `false`.

En la Figura 1 se muestra como quedaría el archivo "**auxiliar.hpp**". Se puede observar que se mantienen las variables de estado que se definieron para resolver el nivel 0. Además, en la versión particular de este fichero de cada estudiante deberán aparecer todas aquellas variables, inicializaciones y métodos que usara para resolver los dos primeros niveles de la práctica. Las modificaciones con respecto a como quedó tras el tutorial 1 son las siguientes:

- En línea 7 aparece `#include <list>` para poder hacer uso del tipo de dato lista.
- En línea 40 se incluye como método público miembro de la clase al Nivel_E.
- En líneas de 49 a 51 las nuevas variables de estado `plan` y `hayPlan`.
- En línea 25 aparece la inicialización de `hayPlan` a `false` en el constructor de clase para los niveles 2 y 3.

```
#ifndef COMPORTAMIENTOAUXILIAR_H
#define COMPORTAMIENTOAUXILIAR_H

#include <chrono>
#include <time.h>
#include <thread>
#include <list>

#include "comportamientos/comportamiento.hpp"

class ComportamientoAuxiliar : public Comportamiento
{

public:
  ComportamientoAuxiliar(unsigned int size = 0) : Comportamiento(size)
  {
    // Inicializar Variables de Estado Niveles 0,1,4
    last_action = IDLE;
    tiene_zapatillas = false;
    giro45Izq = 0;
  }
  ComportamientoAuxiliar(std::vector<std::vector<unsigned char>> mapaR, std::vector<std::vector<unsigned char>> mapaC) : Comportamiento(mapaR,mapaC)
  {
    // Inicializar Variables de Estado Niveles 2,3
    hayPlan = false;
  }
  ComportamientoAuxiliar(const ComportamientoAuxiliar &comport) : Comportamiento(comport) {}
  ~ComportamientoAuxiliar() {}

  Action think(Sensores sensores);

  int interact(Action accion, int valor);

  Action ComportamientoAuxiliarNivel_0(Sensores sensores);
  Action ComportamientoAuxiliarNivel_1(Sensores sensores);
  Action ComportamientoAuxiliarNivel_2(Sensores sensores);
  Action ComportamientoAuxiliarNivel_3(Sensores sensores);
  Action ComportamientoAuxiliarNivel_4(Sensores sensores);

  Action ComportamientoAuxiliarNivel_E(Sensores sensores);


private:
  // Definir Variables de Estado
  Action last_action;
  bool tiene_zapatillas;
  int giro45Izq;

  // Variables de Estado para el nivel E
  list<Action> plan;
  bool hayPlan;
};

#endif
```

Figura 1: Inclusión de las variables de estado play y hayPlan.

A partir de las variables anteriores, podemos plantear la siguiente estructura simple para el agente Auxiliar en el archivo "**auxiliar.cpp**":

```
Action ComportamientoAuxiliar::ComportamientoAuxiliarNivel_E(Sensores sensores){
	Action accion = IDLE;
	if (!hayPlan){
		// Invocar al método de búsqueda
		hayPlan = true;
	}
	if (hayPlan and plan.size()>0){
		accion = plan.front();
		plan.pop_front();
	}
	if (plan.size()== 0){
		hayPlan = false;
	}
	return accion;
}
```

En el esquema anterior, bastaría con poner un procedimiento que calcule un plan para el agente, y el resto del esquema lo ejecutará hasta que el plan se termine. Indicar en el código anterior, que `Action` es el tipo de dato que codifica las acciones posibles del agente y que `IDLE` es una constante del tipo `Action`.

Imaginemos por ejemplo que deseamos que el agente se mueva sobre el terreno como si fuese un caballo de ajedrez, es decir, que avance dos casillas en la dirección en la que se encuentra, gire a la derecha (con dos acciones del tipo `TURN_SR` para completar un giro de 90 grados) y haga un avance más. Para realizar este tipo de movimiento, definimos la función `AvanzaASaltosDeCaballo` y la invocaremos en el método `ComportamientoAuxiliarNivel_E` en el lugar de construcción del plan. La Figura 2 muestra como quedarían ambas funciones.

```
120  list<Action> AvanzaASaltosDeCaballo(){
121      list<Action> secuencia;
122      secuencia.push_back(WALK);
123      secuencia.push_back(WALK);
124      secuencia.push_back(TURN_SR);
125      secuencia.push_back(TURN_SR);
126      secuencia.push_back(WALK);
127      return secuencia;
128  }
129
130  Action ComportamientoAuxiliar::ComportamientoAuxiliarNivel_E(Sensores sensores){
131      Action accion = IDLE;
132      if (!hayPlan){
133          // Invocar al método de búsqueda
134          plan = AvanzaASaltosDeCaballo();
135          hayPlan = true;
136      }
137      if (hayPlan and plan.size()>0){
138          accion = plan.front();
139          plan.pop_front();
140      }
141      if (plan.size()== 0){
142          hayPlan = false;
143      }
144      return accion;
145  }
```

Figura 2: Código de la función AvanzaASaltosDeCaballos e implementación de ejemplo de ComportamientoAuxiliarNivel_E.

Antes de compilar es necesario invocar a la función `ComportamientoAuxiliarNivel_E` desde el método `think`. Para ello utilizaremos el "case 3" del `switch` definido en ese método. Ahí, situaremos la llamada al método tal como se ve en la línea 22 de la Figura 3.

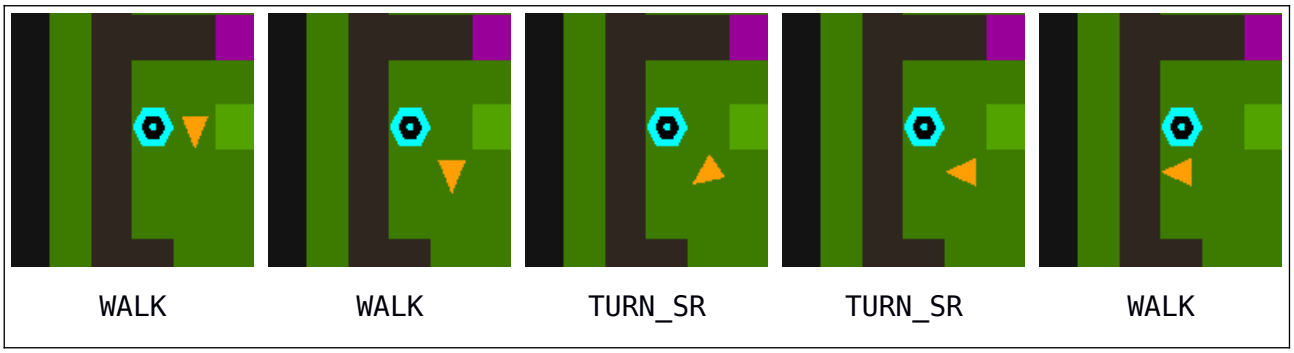
```
5   Action ComportamientoAuxiliar::think(Sensores sensores)
6   {
7       Action accion = IDLE;
8
9       switch (sensores.nivel)
10      {
11      case 0:
12          accion = ComportamientoAuxiliarNivel_0 (sensores);
13          break;
14      case 1:
15          // accion = ComportamientoAuxiliarNivel_1 (sensores);
16          break;
17      case 2:
18          // accion = ComportamientoAuxiliarNivel_2 (sensores);
19          break;
20      case 3:
21          // accion = ComportamientoAuxiliarNivel_3 (sensores);
22          accion = ComportamientoAuxiliarNivel_E (sensores);
23          break;
24      case 4:
25          // accion = ComportamientoAuxiliarNivel_4 (sensores);
26          break;
27      }
28
29      return accion;
30  }
```

Figure 3: Incluir en el caso 3 del `switch` del método `think` la llamada a `ComportamientoAuxiliarNivel_E`.

Una vez hecho la anterior, compilamos y ponemos en un terminal

```
./practica2 mapas/mapa30.map 1 3 7 7 2 11 6 4 12 5 0
```

y vamos pulsando al botón de *"paso"*, podremos observar como en un primer momento no hay plan, se incluye en la variable `plan` el movimiento del caballo y se aplica la primera acción de ese plan. En los siguientes 4 instantes se aplica la siguiente acción hasta que el plan se queda sin acciones. En el siguiente instante, se calcula de nuevo el plan y se ejecuta la primera acción. Este proceso se repite hasta que la simulación termine.



3. Construyendo el nivel ficticio E

El esquema anterior en el método `ComportamientoAuxiliarNivel_E` es suficiente para llevar la ejecución de un plan cuando las condiciones del mundo son estáticas y se tiene información completa del mundo. Por tanto, el esquema anterior nos valdrá para los niveles 2 y 3 de la práctica y nos podremos centrar exclusivamente en implementar el algoritmo de búsqueda que nos piden e insertarlo en el lugar del `AvanzaASaltosDeCaballo`.

En este nivel E (***especial***) que usamos para ilustrar como se deberían realizar los niveles 2 y 3 de la práctica vamos a implementar el algoritmo de búsqueda en anchura para llevar al agente `Auxiliar` a la casilla objetivo. Una descripción en pseudocódigo de este algoritmo podría ser algo como lo siguiente:

```
function BREADTH-1ST-SEARCH(problem)
begin
     current_node ← problem.INITIAL-STATE()
     frontier ← a FIFO queue
     explored ← an empty set
     frontier.insert(current_node);

     while (!frontier.empty() and !problem.Is_Solution(current_node)) do
     begin
           frontier.delete(current_node)
           explored.insert(current_node)
           for each action applicable to current_node do
           begin
                 child ← problem.apply(action, current_node)
                 if (problem.Is_Solution(child) then current_node = child
                 else if child.state() is not in explored or frontier then
                       frontier.insert(child)
           end
           if (!problem.Is_Solution(current_node) then
                 current_node ← frontier.next()
     end
     if (problem.Is_Solution(current_node)) then return SOLUTION(current_node)
     else return failure
end
```

Una descripción en lenguaje natural de ese proceso diría que se usan dos listas para recorrer el espacio de estados que se genera en la búsqueda, una lista con *alma* de COLA llamada `frontier` (que también se suele denominar `frontera` o `abiertos` en otras descripciones) que almacenará los estados pendientes de explorar y una lista con *alma* de CONJUNTO/SET llamada `explored` (o `cerrados` en otras descripciones) que almacena los estados que ya han sido explorados. La misión de la primera es mantener los estados aún no explorados en el mismo orden que se generaron en la exploración, mientras que la misión de la segunda es asegurar que un estado ya explorado no se vuelve a explorar.

Se usa una variable `current_node` para almacenar el estado que se está evaluando en cada momento. En el instante inicial esa variable toma el valor del estado inicial del problema.

El resto del proceso es simple de describir; mientras la lista de nodos pendientes de explorar (`frontier`) no esté vacía y el estado actual no sea una solución, entonces se hace lo siguiente: se elimina el estado actual de la `frontier` y se inserta en la lista de estados explorados (`explored`). Para cada acción aplicable sobre el estado actual, se aplica y se genera un estado descendiente (`child`). Si ese estado es ya solución, la búsqueda termina. Si no lo es y ese estado no está en ninguna de las dos listas, se inserta en `frontier`.

El procedimiento solo se saldrá del ciclo o bien porque `current_node` es solución o porque `frontier` se vació. En el primer caso, se devolverá la solución asociada a ese estado. El segundo caso indica que después de explorar todos los estados que se podían generar desde el estado inicial del problema, ninguno de ellos satisfacía ser solución al problema.

3.1. Definición del concepto de estado y nodo

Ahora toca adaptar esta estrategia de búsqueda para nuestro problema y para ello debemos definir el concepto de *estado.* Lo asociaremos al nivel 3 en lugar del 2, ya que es en este nivel el que espera que sea el Auxiliar el que llegue al destino. La definición que se haga establecerá la complejidad/tamaño del espacio de búsqueda y, por tanto, que soluciones son alcanzables y qué tiempo se necesitará para alcanzarlas.

El estado para este problema contiene la posición del agente Auxiliar (es decir su fila, su columna y su orientación) y también si está o no en posesión del objeto Zapatillas. La razón de incluir este último elemento se debe a que ese objeto hace transitables las casillas de tipo bosque (`'B'`) y afecta a la solución que se pueda encontrar.

Así, en el fichero `auxiliar.hpp` pasaré a definir el estado para este problema como:

```
struct EstadoA {
  int f;
  int c;
  int brujula;
  bool zapatillas;

  bool operator==(const EstadoA &st) const{
    return f == st.f && c == st.c && brujula == st.brujula and zapatillas ==
st.zapatillas;
  }
};
```

Se incluye en la definición el operador "==" para poder realizar comparaciones entre datos de tipo `EstadoA`.

También es necesario definir el "nodo" que se utilizará en el algoritmo. Un nodo contiene la información de un estado junto con información adicional que se requiera para el uso del algoritmo concreto de búsqueda. En este caso, en el nodo incluiremos un campo llamado `secuencia` en el

que almacenaremos las acciones realizadas desde el estado inicial hasta el estado que se contempla en este nodo, y obviamente el `estado` asociado.

```
struct NodoA{
  EstadoA estado;
  list<Action> secuencia;

  bool operator==(const NodoA &node) const{
    return estado == node.estado;
  }
};
```

Se incluye en la definición el operador de igual para poder comparar datos de tipo `NodoA`. Como se puede observar, se considera que dos datos de este tipo son iguales, si sus estados son iguales independientemente del contenido del campo `secuencia`.

3.2. Primera aproximación a la búsqueda en anchura

Fijado el concepto de estado y nodo, pasamos a hacer una primera implementación del algoritmo de búsqueda en anchura intentando transcribir el pseudocódigo que se mostró anteriormente. En la Figura 4 se muestra la estructura general[1].

```
381  /**
382   * @brief Primera versión del algoritmo de Búsqueda en Anchura para el agente Auxiliar.
383   *
384   * @param inicio Estado incial de la búsqueda
385   * @param final Estado final de la búsqueda. Solo es relevante la fila y la columna
386   * @param terreno Matriz que contiene información sobre el tipo de terreno
387   * @param altura Matriz que contiene información sobre la altura de las casillas
388   * @return La secuencia de acciones para llegar al estado final
389   * @note Devuelve un plan vacío si no es posible encontrar un plan válido
390   */
391  list<Action> ComportamientoAuxiliar::AnchuraAuxiliar(const EstadoA &inicio, const EstadoA &final,
392                                                       const vector<vector<unsigned char>> &terreno, const vector<vector<unsigned char>> &altura){
393      NodoA current_node;
394      list<NodoA> frontier;
395      list<NodoA> explored;
396      list<Action> path;
397
398
399      current_node.estado = inicio; // Asigno el estado inicial al nodo actual.
400      frontier.push_back(current_node);   EstadoA NodoA::estado
401      bool SolutionFound = (current_node.estado.f == final.f and current_node.estado.c == final.c);
402  >   while (!SolutionFound and !frontier.empty()){…
442
443      // Devuelvo el camino si se ha encontrado solución
444      if (SolutionFound) path = current_node.secuencia;
445
446
447      return path;
448  }
```

Figura 4: Implementación de la primera versión de la búsqueda en Anchura para el agente `Auxiliar`. Estructura General.

1Recordar incluir el método en la clase ComportamientoAuxiliar en el archivo "**auxiliar.hpp**"

Describiremos poco a poco esta implementación empezando por los argumentos que se han considerado. Se puede observar que tiene 4 argumentos de entrada, el estado inicial llamado `inicio` que es de tipo `EstadoA`, la casilla destino a donde debe llegar el agente jugador llamada `final` y que es de tipo `EstadoA` y por último los mapas sobre el que se realizará la búsqueda que se llama `terreno` el que contiene la información sobre los accidentes geográficos y `altura` con la información sobre la altura de las casillas. La salida de la función es la secuencia de acciones que llevan desde el estado inicial al final si es posible encontrar una solución o un plan vacío en otro caso.

Vamos a realizar la descripción del algoritmo en dos partes, por un lado, veremos la declaración de los datos que se usan junto con la estructura general del proceso de búsqueda, mientras que por el otro lado, veremos el cuerpo del ciclo que se dedica a la generación de los nuevos estados.

Vemos que los datos que se usan son los siguientes:

- `current_node`: almacena el estado actual y que inicialmente toma el valor del parámetro `inicio`.
- `frontier`: una lista que mantiene los nodos pendientes de explorar, inicialmente vacía.
- `explored`: una lista que mantiene los nodos ya explorados, inicialmente vacía.
- `path`: variable que devolverá la secuencia de acciones encontrada como solución.
- `SolutionFound`: una variable lógica que determina si ya se ha encontrado un estado que satisface la condición de ser solución. Esta variable se inicializa mirando si `current_note` satisface la condición de ser solución, es decir, que el agente jugador esté en la fila y en la columna de la casilla destino.

Además de la declaración y antes de entrar en el ciclo principal del proceso de búsqueda (que va desde la línea 402 a la línea 442) lo único que se hace es meter el nodo actual en la lista de nodos pendientes de explorar que es equivalente a meter en dicha lista el estado inicial.

Del ciclo que se plantea en este proceso se sale por dos razones: `SolutionFound` es cierto, lo que indica que se ha encontrado solución o la lista de estados pendientes de explorar está vacía, en cuyo caso, se ha explorado todo el espacio alcanzable desde el estado inicial y no ha sido posible encontrar solución. Por tanto, el valor de `SolutionFound` determina si fue posible encontrar una solución y es el valor que se devuelve.

Ahora pasamos a describir el cuerpo del ciclo del proceso de búsqueda y en él podemos distinguir 3 partes: (1) eliminar el estado actual de la lista de pendientes de explorar y meterlo en la lista de explorados [líneas 403 y 404], (2) generar todos los descendientes del estado actual [desde la línea 411 hasta la línea 434] y (3) tomar el siguiente valor de estado actual de la lista de estados pendientes de explorar [líneas 437 y 440].

```
402     while (!SolutionFound and !frontier.empty()){
403         frontier.pop_front();
404         explored.push_back(current_node);
405 
406         // Compruebo si estoy en una casilla que de las zapatillas
407         if (terreno[current_node.estado.f][current_node.estado.c] == 'D'){
408             current_node.estado.zapatillas = true;
409         }
410 
411         // Genero el hijo resultante de aplicar la acción WALK
412         NodoA child_WALK = current_node;
413         child_WALK.estado = applyA(WALK, current_node.estado, terreno, altura);
414         if (child_WALK.estado.f == final.f and child_WALK.estado.c == final.c){
415             // El hijo generado es solucion
416             child_WALK.secuencia.push_back(WALK);
417             current_node = child_WALK;
418             SolutionFound = true;
419         }
420         else if (!Find(child_WALK, frontier) and !Find(child_WALK, explored)){
421             // Se mete en la lista frontier despues de añadir a secuencia la acción
422             child_WALK.secuencia.push_back(WALK);
423             frontier.push_back(child_WALK);
424         }
425 
426         // Genero el hijo resultante de aplicar la acción TURN_SR
427         if (!SolutionFound){
428             NodoA child_TURN_SR = current_node;
429             child_TURN_SR.estado = applyA(TURN_SR, current_node.estado, terreno, altura);
430             if (!Find(child_TURN_SR, frontier) and !Find(child_TURN_SR, explored)){
431                 child_TURN_SR.secuencia.push_back(TURN_SR);
432                 frontier.push_back(child_TURN_SR);
433             }
434         }
435 
436         // Paso a evaluar el siguiente nodo en la lista "frontier"
437         if (!SolutionFound and !frontier.empty()){
438             current_node = frontier.front();
439             SolutionFound = (current_node.estado.f == final.f and current_node.estado.c == final.c);
440         }
441     }
442 
```

Figure 5: Implementación de la primera versión de la búsqueda en Anchura para el agente `Auxiliar`. Ciclo Principal

(1) En la parte de actualización de las listas se usan las operaciones primitivas para eliminar el primer elemento de una lista en el caso de `frontier` y para añadir al final de una lista en el caso de `explored`.

(2) Se puede observar que la parte destinada a la generación de los estados descendientes del estado actual es lo que ocupa la mayor parte del código. Debido a que en este nivel solo se puede mover el agente `Auxiliar`, solo 2 acciones (las que implican que dicho agente se pueda desplazar) deben ser consideradas (`WALK` y `TURN_SR`). El proceso que se hace para cada una de ellas es similar y para unificar la descripción nos centraremos en como se genera el descendiente de `TURN_SR` [líneas desde la 426 a la 434].

Primero, en línea 428 se crea una nueva variable de tipo `NodoA` que almacenará el resultado de aplicar la acción correspondiente al nodo actual y que en este caso se llama `child_TURN_SR`. El cálculo del nuevo estado se hace a través de una función llamada `applyA` que describiremos más adelante, pero que aquí asumiremos que nos devuelve el nodo resultante de aplicar la acción.

Después (línea 430) se comprueba si el nuevo estado generado ya estaba en alguna de las dos listas (`frontier` o `explored`). Si no está en ninguna, entonces se añade como un

nuevo nodo en la lista de estados pendientes de explorar después de añadir en la lista de acciones del nodo la acción realizada (línea 431). En esta implementación incluimos una función auxiliar llamada `Find` destinada a comprobar si un elemento está o no en la lista. Más adelante mostraremos la implementación de esta función.

El generar el siguiente nodo para la acción y comprobar si ya se había generado antes es común a todas las acciones que se pueden aplicar al estado actual aunque podemos observar que la acción `WALK` tiene algo más. Por la naturaleza de este problema, las acciones `WALK` es la única que permite al agente cambiar de casilla, y solo cambiando de casilla es posible llegar al destino. Por esta razón, **y porque en la búsqueda en anchura, cuando se genera el primer estado que satisface las condiciones de ser solución es la solución al problema y el proceso de búsqueda debe terminar**, se incluye dicha verificación en la acción WALK [líneas de la 414 a la 419].

Por último, indicar que en la línea 427 se incluye una condición para que no se generen los estados descendientes de los giros si ya se generó un estado que es solución.

(3) En esta última parte del cuerpo del ciclo [línea 438] se toma el nuevo valor para `current_node`. Solo se toma un nuevo valor, si en la generación de los descendientes no se ha encontrado un estado que satisface las condiciones de ser solución y obviamente, si la lista de nodos pendientes de explorar (`frontier`) no está vacía.

Los procesos que permiten obtener el siguiente estado a partir del estado actual se encuentra encapsulado dentro de la función `applyA`. Existirían muchas posibilidades de implementación de esta función.

3.2.1. Generando los siguientes estados (`applyA`)

En esta subsección se describe una posible implementación de la función `applyA` que genera a partir del estado actual el estado resultante de aplicar una acción. La implementación hace uso de dos funciones auxiliares. La primera de ellas es la función `CasillaAccesibleAuxiliar`.

```
    bool CasillaAccesibleAuxiliar(const EstadoA &st, const vector<vector<unsigned char>> &terreno,
const vector<vector<unsigned char>> &altura){
       EstadoA next = NextCasillaAuxiliar(st);
       bool check1 = false, check2 = false, check3 = false;
       check1 = terreno[next.f][next.c] != 'P' and terreno[next.f][next.c] != 'M';
       check2  =  terreno[next.f][next.c]  !=  'B'  or  (terreno[next.f][next.c]  ==  'B'  and
st.zapatillas);
       check3 = abs(altura[next.f][next.c] - altura[st.f][st.c]) <= 1;
       return check1 and check2 and check3;
    }
```

Esta función toma tres argumentos de entrada, un `EstadoA st` y dos matrices bidimensionales `terreno` que indica los accidentes geográficos y `altura` que indica la altura de las casillas y devuelve si sería posible por parte del agente Auxiliar hacer una acción `WALK` desde esa posición.

La otra función de la que hace uso `applyA` es `NextCasillaAuxiliar`. Esta función toma como dato de entrada un estado y devuelve estado en el que quedaría el agente Auxiliar después de hacer una acción `WALK`.

```
    EstadoA NextCasillaAuxiliar(const EstadoA &st){
       EstadoA siguiente = st;
       switch (st.brujula)
       {
       case norte:
              siguiente.f = st.f - 1;
              break;
       case noreste:
              siguiente.f = st.f - 1;
              siguiente.c = st.c + 1;
              break;
       case este:
              siguiente.c = st.c + 1;
              break;
       case sureste:
              siguiente.f = st.f + 1;
              siguiente.c = st.c + 1;
              break;
       case sur:
              siguiente.f = st.f + 1;
              break;
       case suroeste:
              siguiente.f = st.f + 1;
              siguiente.c = st.c - 1;
              break;
       case oeste:
              siguiente.c = st.c - 1;
              break;
       case noroeste:
              siguiente.f = st.f - 1;
              siguiente.c = st.c - 1;
       }
       return siguiente;
    }
```

A partir de las dos funciones anteriores, planteamos `applyA` como una función con cuatro argumentos de entrada: la acción que se quiere realizar, el estado actual del agente y terreno y altura, mapas donde se representan los accidentes geográficos y la altura de las casillas. Devuelve el estado resultante tras aplicar la acción. Si no es aplicable la acción, devuelve como estado el mismo que recibió como argumento.

```
    EstadoA applyA(Action accion, const EstadoA & st, const vector<vector<unsigned char>> &terreno,
const vector<vector<unsigned char>> &altura){
       EstadoA next = st;
       switch(accion){
              case WALK:
                     if (CasillaAccesibleAuxiliar(st,terreno,altura)){
                            next = NextCasillaAuxiliar(st);
                     }
              break;
              case TURN_SR:
                     next.brujula = (next.brujula+1)%8;
                     break;
       }
       return next;
    }
```

La descripción de esta función es muy simple. En función de cada posible acción de las que se puede realizar (recordamos que el agente Auxiliar solo puede realizar tres acciones) se cálcula el estado resultado de aplicar dicha acción sobre el estado `st` almacenándose el resultado final en la variable `next` que se inicializa con el valor de `st` al principio de la función.

3.2.2. Función para buscar en una lista (`Find`)

Para que quede definitivamente terminada la implementación de la primera implementación de la búsqueda en anchura queda por mostrar la codificación de la función `Find`. Esta función tiene como argumentos de entrada un estado concreto `st` y una lista de estados y devuelve si está o no en `lista`.

```
bool Find (const NodoA & st, const list<NodoA> &lista){
   auto it = lista.begin();
   while (it != lista.end() and !((*it) == st)){
          it++;
   }
   return (it != lista.end());
}
```

Se puede observar que es una implementación de búsqueda secuencial sobre una lista y que tiene un orden de complejidad `O(n)`, siendo `n` el tamaño de la lista.

3.2.3. Función para visualizar el recorrido en el mapa (`VisualizaPlan`)

Incluimos aquí la implementación de una función que se usa para proyectar como será el recorrido que ha planificado el agente. Toma como argumentos el estado inicial del plan y la lista de acciones a realizar. El recorrido se refleja en el mapa del simulador marcando con un punto naranja las casillas que serán transitadas por el agente `Auxiliar`.

```
void ComportamientoAuxiliar::VisualizaPlan(const EstadoA &st, const list<Action> &plan)
{
   AnularMatrizA(mapaConPlan);
   EstadoA cst = st;

   auto it = plan.begin();
   while (it != plan.end())
   {
          switch (*it)
          {
          case WALK:
                 switch (cst.brujula)
                 {
                 case 0:
                        cst.f--;
                        break;
                 case 1:
                        cst.f--;
                        cst.c++;
                        break;
                 case 2:
                        cst.c++;
                        break;
                 case 3:
                        cst.f++;
                        cst.c++;
                        break;
                 case 4:
                        cst.f++;
                        break;
                 case 5:
                        cst.f++;
                        cst.c--;
                        break;
                 case 6:
                        cst.c--;
                        break;
```

```
                    case 7:
                           cst.f--;
                           cst.c--;
                           break;
                    }
                    mapaConPlan[cst.f][cst.c] = 2;
                    break;
             case TURN_SR:
                    cst.brujula = (cst.brujula + 1) % 8;
                    break;
             }
             it++;
      }
   }
```

Este módulo está definido como un método público de la clase `ComportamientoAuxiliar`, por esa razón se debe incluir en el archivo "**auxiliar.hpp**", dentro de la clase, el prototipo de este módulo.

3.2.4. Probando la primera implementación de la búsqueda en anchura

Una vez implementado el algoritmo de búsqueda podemos pasar a probar su funcionamiento ante algún problema de búsqueda. Para eso redefinimos el contenido del método `ComportamientoAuxiliarNivel_E`.

```
   Action ComportamientoAuxiliar::ComportamientoAuxiliarNivel_E(Sensores sensores){
      Action accion = IDLE;
      if (!hayPlan){
             // Invocar al método de búsqueda
             EstadoA inicio, fin;
             inicio.f = sensores.posF;
             inicio.c = sensores.posC;
             inicio.brujula = sensores.rumbo;
             inicio.zapatillas = tiene_zapatillas;
             fin.f = sensores.destinoF;
             fin.c = sensores.destinoC;
             plan = AnchuraAuxiliar(inicio, fin, mapaResultado, mapaCotas);
             VisualizaPlan(inicio,plan);
             hayPlan = plan.size() != 0 ;
      }
      if (hayPlan and plan.size()>0){
             accion = plan.front();
             plan.pop_front();
      }
      if (plan.size()== 0){
             hayPlan = false;
      }
      return accion;
   }
```

Se puede observar que lo único que cambia con respecto a lo que aparece en la Figura 2 es la parte relacionada con la invocación al algoritmo de búsqueda (lo que hay en el bloque con el comentario **`// Invocar al método de búsqueda`**). En realidad, lo que se hace en ese trozo de código es definir e inicializar el estado inicial y final que se pedirán en el algoritmo de búsqueda antes de llamar al propio algoritmo. El estado inicial se inicializa con los valores de fila, columna, orientación y si tiene zapatillas, mientras que en el estado final son solo necesarios los valores de fila y columna de la casilla objetivo.

Después de asignar los valores a inicio y fin, se hace la invocación al módulo `AnchuraAuxiliar` pasandoles como argumentos el estado `inicial` y `fin` y los mapas `mapaResultado` y `mapaCotas` que contienen los accidentes geográficos y la altura de las casillas del mapa. El resultado se almacena en la variable de estado `plan`.

La siguiente sentencia hace que se vea el recorrido encontrado en el mapa del simulador y se asigna la variable de estado `hayPlan`. Esta variable de estado tendrá el valor `true` solo si se ha encontrado un plan.

Para probar el funcionamiento, compilamos y ejecutamos con el siguiente comando

```
./practica2 mapas/mapa30.map 1 3 5 5 2 10 10 4 12 5 0
```

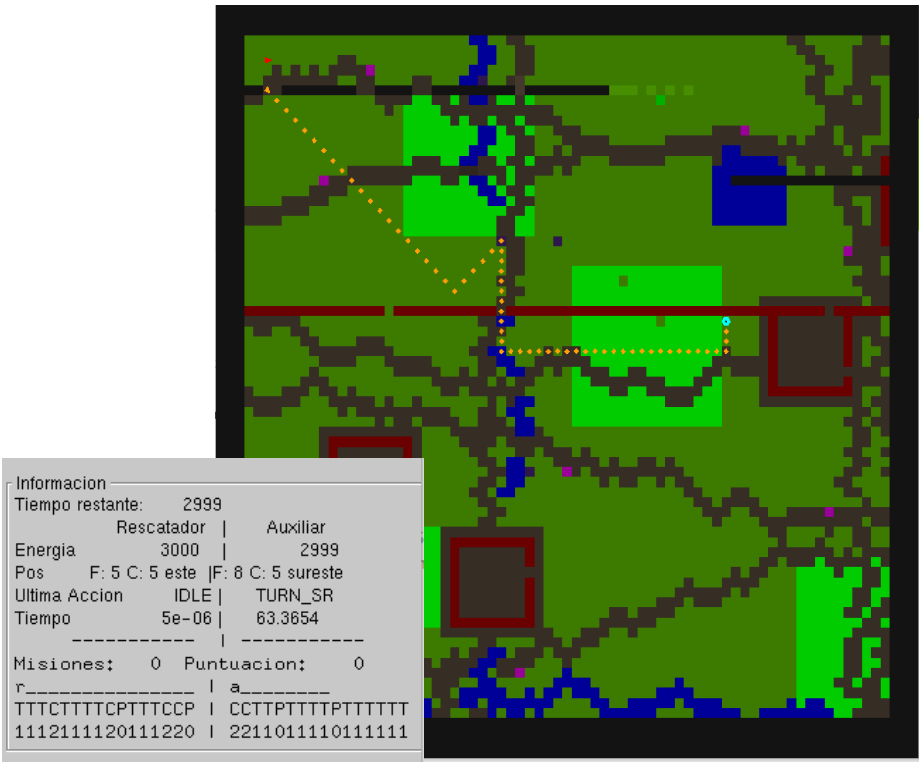
Nos devuelve un plan que tiene 33 acciones. El recorrido se muestra con puntos naranja sobre las casillas que deberá recorrer para llegar a la casilla objetivo.

Si se va pulsando el botón "Paso", se ve como el agente va recorriendo las casillas marcadas hasta llegar al objetivo. Una vez en el objetivo, la simulación termina.

Repetimos la ejecución pero ahora con la siguiente llamada[2]

```
./practica2 mapas/mapa75.map 1 3 5 5 2 8 5 2 31 54 0
```

2Ten paciencia. La ejecución tarda, no se "colgó". Si tarda más de una hora, mmmm, lo mismo sí se colgó :)



En este caso, **tarda unos 63 segundos** en conseguir el plan (este valor de tiempo depende del ordenador donde se ha ejecutado y por tanto puede darte un valor distinto). El plan está compuesto por 85 acciones.

3.3. Segunda versión de la búsqueda en anchura: acelerando la búsqueda

En esta segunda versión, modificaremos ligeramente el método de búsqueda para mejorar el tiempo de respuesta del algoritmo. Las modificaciones afectarán fundamentalmente a las estructuras de datos que utilizamos para almacenar las dos listas. La lista de nodos pendientes de explorar (`frontier`) está implementada siguiendo el TDA lista. Las operaciones que requiere el algoritmo de búsqueda sobre esta lista son 4: insertar, leer el primero, sacar el primero y buscar. Las 3 primeras operaciones son muy eficientes y se podrían haber hecho usando un TDA cola (`queue`), pero en la implementación de `queue` en la `STL` de `C++` no tiene un operador para buscar y por eso razón se usó la TDA de lista usándola como si fuera una cola (el primero en entrar es el primero en salir).

La búsqueda de elementos en una lista tiene orden de complejidad O(n) como ya hemos referido varias veces a lo largo de este tutorial y no resulta un mecanismo muy eficiente si se requiere hacerlo muchas veces y si las listas son muy grandes como es el caso de este algoritmo de búsqueda. Se requiere buscar en `frontier` cada vez que se genera un nuevo estado.

En esta versión del algoritmo de búsqueda en anchura que aquí proponemos, vamos a eliminar la búsqueda de nodos en `frontier`, así que mantenemos el tipo `list` asociado a esta lista y todas las búsquedas de nodos se harán sobre la lista `explored`.

Si todas las búsquedas las haremos sobre `explored`, se deberá usar una estructura de datos eficiente para realizar esta operación en dicha lista. Vamos a optar por usar el TDA `set` para su implementación. Sobre esta estructura de datos, las operaciones de búsqueda e inserción son de $O(\log_2(n))$ que son justo las operaciones que el algoritmo de búsqueda requiere.

El tipo de dato `set` de la `STL` de `C++` requiere definir un operador de orden entre los elementos que forman parte de la lista, ya que en la base de su implementación hay un árbol binario de búsqueda. Por esa razón, cuando definimos el tipo `nodoA`, definimos el operador *menor estricto*. Es este operador el que usará el tipo `set` para ordenar los elementos dentro de la estructura. Modificaremos la defínción del `struct NodoA` en el archivo "**auxiliar.hpp**" para que quede como se indica a continuación:

```
    struct NodoA{
      EstadoA estado;
      list<Action> secuencia;

      bool operator==(const NodoA &node) const{
        return estado == node.estado;
      }

      bool operator<(const NodoA &node) const{
        if (estado.f < node.estado.f) return true;
        else if (estado.f == node.estado.f and estado.c < node.estado.c) return true;
        else if (estado.f == node.estado.f and estado.c == node.estado.c and estado.brujula <
node.estado.brujula) return true;
        else if (estado.f == node.estado.f and estado.c == node.estado.c and estado.brujula ==
node.estado.brujula and estado.zapatillas < node.estado.zapatillas) return true;
        else return false;
      }
    };
```

Aunque debe tener estructura de relación de orden por los requerimientos del `set`, desde el punto de vista del papel que juega dentro del algoritmo de búsqueda en realidad se puede ver como un operador de igualdad, y con esa visión se podría leer como: *dos nodos son distintos si la fila del agente auxiliar de un nodo es menor que la fila del agente auxiliar del otro nodo. También son distintos si siendo lo anterior igual, la columna del agente de un nodo es menor que la columna del agente del otro nodo. Siendo las filas y las columnas iguales, será distinto si las orientaciones no coinciden. Por último, son distintos si siendo las filas, las columnas y las orientaciones iguales para los dos nodos, uno tiene las zapatillas y el otro no. Si no pasa lo que se ha descrito antes, los dos nodos son iguales*.

Bueno, ya hemos tomado la decisión sobre los tipos de datos de las listas, `frontier` seguirá siendo una lista de `NodoA` sobre el que no se hará ninguna búsqueda y `explored` será un `set` de `nodoA` y solo sobre él se realizarán todas las búsquedas para encontrar los nodos repetidos.

```
424 list<Action> ComportamientoAuxiliar::AnchuraAuxiliar_V2(const EstadoA &inicio, const EstadoA &final,
425                                                         const vector<vector<unsigned char>> &terreno, const vector<vector<unsigned char>> &altura){
426     NodoA current_node;
427     list<NodoA> frontier;
428     set<NodoA> explored;
429     list<Action> path;
430
431
432     current_node.estado = inicio; // Asigno el estado inicial al nodo actual.
433     frontier.push_back(current_node);
434     bool SolutionFound = (current_node.estado.f == final.f and current_node.estado.c == final.c);
435     while (!SolutionFound and !frontier.empty()){…
481
482     // Devuelvo el camino si se ha encontrado solución
483     if (SolutionFound) path = current_node.secuencia;
484
485
486     return path;
487 }
```

Mirando la estructura general podemos observar que solo hay un cambio con respecto a la versión anterior y es en la línea 428 donde ahora `explored`, la lista de nodos ya explorados, se define de tipo `set`.

```
435     while (!SolutionFound and !frontier.empty()){
436         frontier.pop_front();
437         explored.insert(current_node);
438         // Compruebo si estoy en una casilla que de las zapatillas
439         if (terreno[current_node.estado.f][current_node.estado.c] == 'D'){
440             current_node.estado.zapatillas = true;
441         }
442
443         // Genero el hijo resultante de aplicar la acción WALK
444         NodoA child_WALK = current_node;
445         child_WALK.estado = applyA(WALK, current_node.estado, terreno, altura);
446         if (child_WALK.estado.f == final.f and child_WALK.estado.c == final.c){
447             // El hijo generado es solucion
448             child_WALK.secuencia.push_back(WALK);
449             current_node = child_WALK;
450             SolutionFound = true;
451         }
452         else if (explored.find(child_WALK) == explored.end())
453         {
454             // Se mete en la lista frontier despues de añadir a secuencia la acción
455             child_WALK.secuencia.push_back(WALK);
456             frontier.push_back(child_WALK);
457         }
458
459         // Genero el hijo resultante de aplicar la acción TURN_SR
460         if (!SolutionFound){
461             NodoA child_TURN_SR = current_node;
462             child_TURN_SR.estado = applyA(TURN_SR, current_node.estado, terreno, altura);
463             if (explored.find(child_TURN_SR) == explored.end()){
464                 child_TURN_SR.secuencia.push_back(TURN_SR);
465                 frontier.push_back(child_TURN_SR);
466             }
467         }
468
469         // Paso a evaluar el siguiente nodo en la lista "frontier"
470         if (!SolutionFound and !frontier.empty()){
471             current_node = frontier.front();
472             while (explored.find(current_node) != explored.end() and !frontier.empty()){
473                 frontier.pop_front();
474                 current_node = frontier.front();
475             }
476         }
477     }
478
479     // Devuelvo el camino si se ha encontrado solución
480     if (SolutionFound) path = current_node.secuencia;
481
482
483     return path;
484 }
```

Las cosas que cambian dentro del ciclo principal del algoritmo de búsqueda en anchura [desde la línea 435 a la 481] son:

- la operación de añadir un nuevo nodo a la `explored` (antes al ser una lista se usaba el método `push_back` y ahora al ser un `set` se usa el método `insert`) en la línea 437.
- no se busca en las dos listas sino que solo se busca en la lista de `explored` y eso se expresa en las líneas 452 y 463.
- el proceso para elegir el siguiente nodo actual [líneas de la 470 a 476]. Para entender esto necesitamos conocer que implica no buscar nodos en `frontier`.
- Antes de esta modificación no había nodos repetidos ni en `frontier` ni en `explored`. Ahora puede suceder que varias instancias de un mismo nodo aparezcan múltiples veces en `frontier`. En principio, esto no implicaría un comportamiento distinto al algoritmo de búsqueda en anchura siempre y cuando antes de tomar un nodo como nodo actual, se verifique que no es un nodo que ya se haya explorado. Pues justo eso es lo que hace el trozo de código de las líneas anteriormente referidas, van tomando nodos de `frontier` en el mismo orden en el que fueron añadidos a la lista hasta encontrar el primero de ellos que verifica no estar ya en `explored`. Con esta modificación, lo que hacemos es permitir que la lista `frontier` ocupe potencialmente mucho más espacio a cambio de hacer menos procesos de búsqueda con la intención de reducir el tiempo necesario para encontrar una solución.

Vamos a probar el funcionamiento de esta tercera versión, y para ello sustituimos en la función que invoca al algoritmo de búsqueda en el método `ComportamientoAuxiliarNivel_E` la línea

```
hayPlan = AnchuraAuxiliar(inicio, fin, mapaResultado, mapaCotas);
```

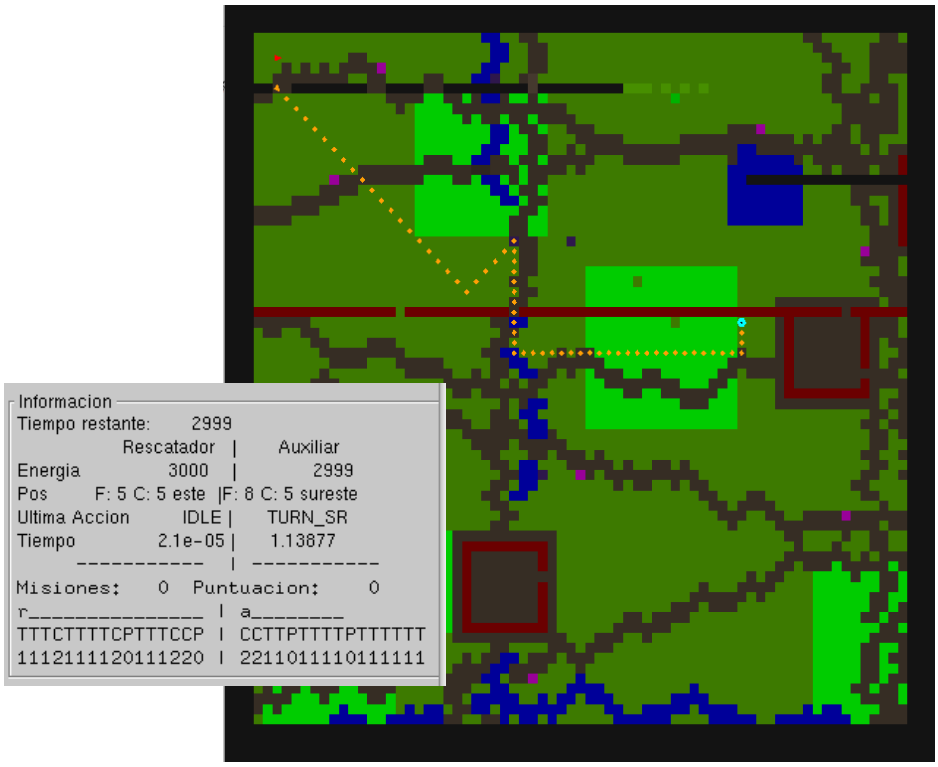
por

```
hayPlan = AnchuraAuxiliar_V2(inicio, fin, mapaResultado, mapaCotas);
```

compilamos y ejecutamos

```
./practica2 mapas/mapa75.map 1 3 5 5 2 8 5 2 31 54 0
```

obtenemos



Se puede observar que el tiempo que se ha consumido para el agente `Auxiliar` es de unos 1.1 segundos frente a los 63 que necesitó la versión anterior. El plan resultante es el mismo y contiene 85 acciones.

4. Comentarios Finales

En este tutorial se ha planteado la solución a un nivel ficticio de una dificultad semejante a los niveles 2 y 3 de la práctica 2. Muchas de las funciones aquí definidas será de utilidad directa para resolver esos niveles y otras deberan cambiarse para adaptarse a las peculiaridades de cada uno de los algoritmos que se pide y también a las peculiaridades de los agentes. Indicar que la estructura general entre los algoritmos que se piden en la práctica y el que se ha desarrollado aquí son muy parecidos. Aunque son muy parecidos, el estudiante tiene que tener en cuenta que hay dos grandes diferencias con los algoritmos que debe implementar en la práctica:

1. La gestión de la lista frontier. Aquí, en la búsqueda en anchura, se comporta como una cola. En los otros algoritmos no es así y deberá seleccionarse una estructura de datos apropiada para hacer eficiente la gestión de frontier.
2. La condición de finalización de la búsqueda. En anchura, la búsqueda termina cuando se genera un hijo que satisface las condiciones de alguno de los estados finales. Por esa razón, en la implementación, después de hacer `WALK` se miraba si se había llegado a la casilla objetivo. En los algoritmos pedidos, **la primera solución puede no ser la mejor solución** y

por tanto la condición de salida es distinta ya que se piden soluciones óptimas en consumo de energía.

Con el fin de ayudar al estudiante en el desarrollo de este tutorial, la siguiente imagen indica el orden en el que se han incluyendo los distintos módulos en el archivo "**auxiliar.cpp**".

```
120   // ======================================== Nivel Especial del Tutorial 2p2 ============================
121 > /**…
126 > void AnularMatrizA(vector<vector<unsigned char>> &m)…
136
137 > /**…
143 > void ComportamientoAuxiliar::VisualizaPlan(const EstadoA &st, const list<Action> &plan)…
194
195 > /**…
201 > EstadoA NextCasillaAuxiliar(const EstadoA &st){…
235
236 > /**…
245 > bool CasillaAccesibleAuxiliar(const EstadoA &st, const vector<vector<unsigned char>> &terreno, const vector<v
253
254 > /**…
264 > EstadoA applyA(Action accion, const EstadoA & st, const vector<vector<unsigned char>> &terreno, const vector<
278
279 > /**…
287 > bool Find (const NodoA & st, const list<NodoA> &lista){…
294
295 > /**…
301 > void ComportamientoAuxiliar::PintaPlan(const list<Action> &plan, bool zap)…
344
345 > /**…
355   list<Action> ComportamientoAuxiliar::AnchuraAuxiliar(const EstadoA &inicio, const EstadoA &final,
356 >                                                     const vector<vector<unsigned char>> &terreno, const vector<vector<unsign
413
414 > /**…
424   list<Action> ComportamientoAuxiliar::AnchuraAuxiliar_V2(const EstadoA &inicio, const EstadoA &final,
425 >                                                     const vector<vector<unsigned char>> &terreno, const vector<vector<unsign
485
486 > /**…
491 > list<Action> AvanzaASaltosDeCaballo(){…
500
501 > /**…
507 > Action ComportamientoAuxiliar::ComportamientoAuxiliarNivel_E(Sensores sensores){…
```

Una estructura semejante, en cuanto a las funciones definidas aquí, pero adaptadas para al otro de los agentes, debe realizar el estudiante. Entre esas adaptaciones está la definición de las estructuras `EstadoR` y `NodoR` que definirán el concepto de estado y de nodo del agente `Rescatador`.

Para la realización del nivel 3 real de la práctica que puede interferir lo realizado en este `nivel E` descrito en este tutorial. El estudiante puede decidir poner un nombre distinto al nuevo concepto de estado o nodo del nuevo algoritmo o bien, redifinir `EstadoA` y `NodoA` para aplicarlo a su algoritmo y eliminar los métodos `AnchuraAuxiliar` y `AnchuraAuxiliar_V2` de los archivos. Estos métodos son un apoyo para el estudiante, pero no tienen porqué aparecer en la versión final de la práctica que se entregará.

Por último una pequeña mención al último de los niveles de la práctica en cuanto a su esquema general. Si el estudiante sigue la modularización propuesta en el software, se realizará en el módulo que se llama `ComportamientoAuxiliarNivel_4`. Dicho módulo tendrá una estructura semejante al que se propone en el `Nivel_E` que debe ser igual al que hay en los niveles 2 y 3. Sin embargo, en ese nivel pueden aparecer circunstancias que hagan que el plan desarrollado no se pueda llevar a término y por tanto, hay que monitorizar cada acción antes de realizarla para evaluar si se puede

hacer, y si no se puede hacer, habria que adaptar o replanificar el recorrido hacía el objetivo. Por tanto, en una versión avanzada del nivel 4, el módulo `ComportamientoAuxiliarNivel_4` tendrá una estructura más compleja incluyendo estructuras condicionales definiendo comportarmientos reactivos en algunos casos para mejorar la eficiencia del agente.

5. Algunos códigos para la versión del agente Rescatador

Se incluyen los códigos de algunas funciones que se incluiran en el fichero "**rescatador.cpp**" y que el estudiante puede utilizar para la definición de los niveles la práctica que afectan al agente `Rescatador`.

5.1. AnularMatrizR

```
void AnularMatrizR(vector<vector<unsigned char>> &m)
{
   for (int i = 0; i < m[0].size(); i++)
   {
          for (int j = 0; j < m.size(); j++)
          {
                 m[i][j] = 0;
          }
   }
}
```

5.2. ComportamientoRescatador::VisualizaPlan

En esta implementación se asume que el estudiante ha definido un `struct` llamado `EstadoR` que define el concepto de estado para el agente `Rescatador` y que tiene al menos como campos `f`, `c` y `brujula` indicando respectivamente la fila, la columna y la orientación del agente.

```
void ComportamientoRescatador::VisualizaPlan(const EstadoR &st, const list<Action> &plan)
{
   AnularMatrizR(mapaConPlan);
   EstadoR cst = st;

   auto it = plan.begin();
   while (it != plan.end())
   {
          switch (*it)
          {
          case RUN:
                 switch (cst.brujula)
                 {
                 case 0:
                        cst.f--;
                        break;
                 case 1:
                        cst.f--;
                        cst.c++;
                        break;
                 case 2:
                        cst.c++;
                        break;
                 case 3:
                        cst.f++;
                        cst.c++;
```

```
                        break;
                case 4:
                        cst.f++;
                        break;
                case 5:
                        cst.f++;
                        cst.c--;
                        break;
                case 6:
                        cst.c--;
                        break;
                case 7:
                        cst.f--;
                        cst.c--;
                        break;
                }
                mapaConPlan[cst.f][cst.c] = 3;
        case WALK:
                switch (cst.brujula)
                {
                case 0:
                        cst.f--;
                        break;
                case 1:
                        cst.f--;
                        cst.c++;
                        break;
                case 2:
                        cst.c++;
                        break;
                case 3:
                        cst.f++;
                        cst.c++;
                        break;
                case 4:
                        cst.f++;
                        break;
                case 5:
                        cst.f++;
                        cst.c--;
                        break;
                case 6:
                        cst.c--;
                        break;
                case 7:
                        cst.f--;
                        cst.c--;
                        break;
                }
                mapaConPlan[cst.f][cst.c] = 1;
                break;
        case TURN_SR:
                cst.brujula = (cst.brujula + 1) % 8;
                break;
        case TURN_L:
                cst.brujula = (cst.brujula + 6) % 8;
                break;
        }
        it++;
    }
}
```

5.2. ComportamientoRescatador::PintaPlan

```
void ComportamientoRescatador::PintaPlan(const list<Action> &plan, bool zap)
{
   auto it = plan.begin();
   while (it != plan.end())
   {
          if (*it == WALK)
          {
                 cout << "W ";
          }
          else if (*it == RUN)
          {
                 cout << "R ";
          }
          else if (*it == TURN_SR)
          {
                 cout << "r ";
          }
          else if (*it == TURN_L)
          {
                 cout << "L ";
          }
          else if (*it == CALL_ON)
          {
                 cout << "C ";
          }
          else if (*it == CALL_OFF)
          {
                 cout << "c ";
          }
          else if (*it == IDLE)
          {
                 cout << "I ";
          }
          else
          {
                 cout << "-_ ";
          }
          it++;
   }
   cout << "( longitud " << plan.size();
   if (zap) cout << "[Z]";
   cout << ")\n";
}
```