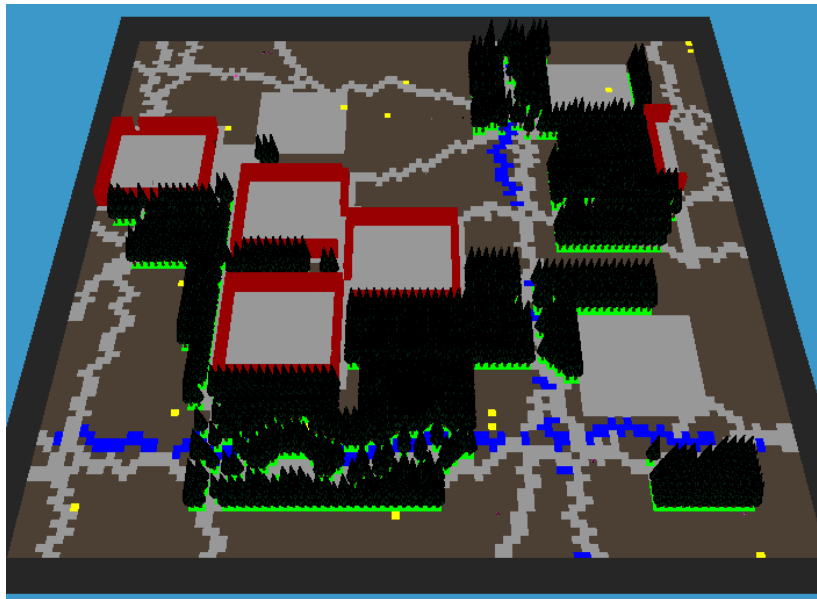


INTELIGENCIA ARTIFICIAL

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

Tutorial: Práctica 2

Agentes Reactivos/Deliberativos



DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN E INTELIGENCIA ARTIFICIAL

UNIVERSIDAD DE GRANADA

Curso 2022-2023

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 hacia una casillas destino. En este tutorial se pretende introducir al estudiante en la dinámica que debería seguir para la construcción de su práctica y para ellos se resolverá casi completamente el nivel 0 de dicha práctica.

En la práctica anterior se ha desarrollado un comportamiento reactivo sobre un mundo muy parecido al que se describe y con el que trabajaremos durante la práctica 2. La mayoría de los estudiantes estarán familiarizados con estos comportamiento y con los elementos que se vuelven a repetir en esta segunda práctica. Por esta razón, indicamos a aquellos estudiantes que no tengan claro este manejo, realicen el tutorial de la práctica 1 antes de leer este. Aunque en esta segunda, los agentes reactivos no están el centro del estudio, si que son necesarios para complementar los comportamientos deliberativos que aquí se verán.

Así que para este tutorial asumiremos que se ha realizado el tutorial de la práctica anterior y nos centraremos en los pasos que se requieren para implementar un algoritmo concreto de búsqueda sobre un problema concreto a partir de la definición del concepto de estado para ese problema por un lado para conseguir un plan de acciones y por otro lado hacer notar la necesidad de definir sistemas de monitorización del plan para garantizar su correcta ejecución.

2. Estructura general del agente jugador

En la descripción de esta práctica se indica que hay dos agentes involucrados en los procesos de desplazamiento por el mapa, el agente jugador y el agente sonámbulo aunque en realidad los movimientos del agente sonámbulo son ordenados por el agente jugador y por tanto, es el comportamiento de este último el único en el que nos debemos centrar.

La estructura del software que se proporciona es semejante al utilizado en la práctica anterior, y al igual que en aquella, la forma de proceder para la realización de la tarea consiste en construir la función `think` para resolver los distintos problemas que se plantean. La función

Action think(Sensores sensores)

toma como parámetro de entrada el valor de los sensores del agente en el instante actual, y devuelve un 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 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 a uno de los agente desde la casilla donde se encuentre a la casilla destino. Dicho plan se almacenará en una variable de estado. El método `think` 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 el método `think` que viene a ejecutar la siguiente acción del plan, si el plan existe, y si no existe, pues calcula el plan. Para ello vamos a definir dos variables de estado: `plan` que almacenará el plan a ejecutar y que la vamos a definir de tipo `list<Action>` en la parte privada de la clase `ComportamientoJugador` 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`. El constructor de clase la variable `hayPlan` se inicializará a `false`.

```

1  #ifndef COMPORTAMIENTOJUGADOR_H
2  #define COMPORTAMIENTOJUGADOR_H
3
4  #include "comportamientos/comportamiento.hpp"
5
6  #include <list>
7
8
9  class ComportamientoJugador : public Comportamiento {
10 public:
11     ComportamientoJugador(unsigned int size) : Comportamiento(size) {
12         // Inicializar Variables de Estado
13     }
14     ComportamientoJugador(std::vector< std::vector< unsigned char> > mapaR) : Comportamiento(mapaR) {
15         // Inicializar Variables de Estado
16         hayPlan = false;
17     }
18     ComportamientoJugador(const ComportamientoJugador & comport) : Comportamiento(comport){}
19     ~ComportamientoJugador(){}
20
21     Action think(Sensores sensores);
22     int interact(Action accion, int valor);
23
24 private:
25     // Declarar Variables de Estado
26     list<Action> plan; //Almacena el plan en ejecución
27     bool hayPlan;      //Si verdad indica que se está siguiendo un plan.
28 };
29
30
31 #endif

```

A partir de las variables anteriores, podemos plantear la siguiente estructura simple para el agente jugador.

```

Action ComportamientoJugador::think(Sensores sensores){
    Action accion = actIDLE;
    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 actIDLE es una constante del tipo Action.

Imaginemos por ejemplo que deseamos que el agente se mueva sobre el terreno como si fuera un caballo de ajedrez, es decir, que avance dos casillas en la dirección en la que se encuentra, gire a la derecha y haga un avance más. Una función que expresa esa secuencia de acciones podría ser como la que se describe a continuación.

```
list<Action> AvanzaASaltosDeCaballo(){
    list<Action> secuencia;
    secuencia.push_back(actFORWARD);
    secuencia.push_back(actFORWARD);
    secuencia.push_back(actTURN_R);
    secuencia.push_back(actFORWARD);
    return secuencia;
}
```

Si en el método `think()` incluimos la llamada a `AvanzaASaltosDeCaballo()` en el lugar donde debe estar la invocación para construir el plan obtendremos algo parecido a esto:

```
1  #include "../Comportamientos_Jugador/jugador.hpp"
2  #include "motorlib/util.h"
3
4  #include <iostream>
5  #include <cmath>
6  #include <set>
7  #include <stack>
8
9  list<Action> AvanzaASaltosDeCaballo()
10 {
11     list<Action> secuencia;
12     secuencia.push_back(actFORWARD);
13     secuencia.push_back(actFORWARD);
14     secuencia.push_back(actTURN_R);
15     secuencia.push_back(actFORWARD);
16     return secuencia;
17 }
18
19 // Este es el método principal que se piden en la practica.
20 // Tiene como entrada la información de los sensores y devuelve la acción a realizar.
21 // Para ver los distintos sensores mirar fichero "comportamiento.hpp"
22 Action ComportamientoJugador::think(Sensores sensores)
23 {
24     Action accion = actIDLE;
25     if (!hayPlan)
26     {
27         cout << "Calculando un nuevo plan\n";
28         plan = AvanzaASaltosDeCaballo();
29         hayPlan = true;
30     }
31     if (hayPlan and plan.size() > 0)
32     {
33         cout << "Ejecutando la siguiente acción del plan\n";
34         accion = plan.front();
35         plan.pop_front();
36     }
37     if (plan.size() == 0)
38     {
39         cout << "Se completó el plan\n";
40         hayPlan = false;
41     }
42     return accion;
43 }
44
```

Si una vez que compilamos, ponemos en un terminal

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

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 3 instantes se aplica la siguiente acción hasta que la 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 0

El esquema anterior en el método `think` 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 4 primeros niveles de la práctica (del nivel 0 al 3) y nos podremos centrar exclusivamente en implementar el algoritmo de búsqueda que nos piden e insertarlo en el lugar del `AvanzaASaltosDeCaballo`.

En el nivel 0 debemos implementar el algoritmo de búsqueda en anchura para el problema de llevar al agente jugador 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

Ahora toca adaptar esta estrategia de búsqueda para nuestro problema y para ello debemos definir el concepto de estado para este nivel 0 de la práctica. La definición que se haga establecerá la complejidad/tamaño del espacio de búsqueda y por tanto, que soluciones son alcanzables y que tiempo se necesitará para alcanzarlas.

El nivel 0 indica que debemos llevar al agente jugador desde su posición inicial hasta la casilla objetivo y además el plan que se devuelva como solución no puede incluir ninguna orden de movimiento para el agente sonámbulo. Desde ese punto de vista parecería que el estado para este problema solo debería contener la posición del agente jugador (es decir su fila, su columna y su orientación) ya que el agente sonámbulo permanecerá inmóvil (sin cambiar su posición ni su orientación) en el desplazamiento del agente jugador.

El hecho de que el jugador sonámbulo no se mueva, no significa que no influya en el plan que debe construir el agente jugador ya que el primero podría estar ocupando una casilla que el segundo está considerando como transitable y por tanto que podría formar parte del camino para obtener la solución. Por esta razón, la posición del agente sonámbulo debe formar parte del estado del problema.

```
11 struct ubicacion{
12     int f;
13     int c;
14     Orientation brujula;
15
16     bool operator==(const ubicacion &ub)const {
17         return (f == ub.f and c == ub.c and brujula == ub.brujula);
18     }
19 };
```

Así, definiremos estado en el problema como la ubicación del agente jugador y la ubicación del agente sonámbulo. En el fichero `comportamiento.hpp` se define el tipo de dato ubicación del que haremos uso para definir el concepto de estado en el problema de nivel 0.

Así, en el fichero `jugador.hpp` pasará a definir el estado del nivel 0 como:


```

1  #ifndef COMPORTAMIENTOJUGADOR_H
2  #define COMPORTAMIENTOJUGADOR_H
3
4  #include "comportamientos/comportamiento.hpp"
5
6  #include <list>
7
8  struct stateN0{
9      ubicacion jugador;
10     ubicacion sonambulo;
11
12     bool operator== (const stateN0 &x) const{
13         if (jugador == x.jugador and
14             sonambulo.f == x.sonambulo.f and sonambulo.c == x.sonambulo.c){
15             return true;
16         }
17         else {
18             return false;
19         }
20     }
21 };
22
23 class ComportamientoJugador : public Comportamiento {
24 public:
25     ComportamientoJugador(unsigned int size) : Comportamiento(size) {
26         // Inicializar Variables de Estado

```

Sobre el tipo de dato `stateN0` defino el operador de igualdad: dos estados son iguales si la ubicación del jugador en los dos estados es la misma y la posición (sin tener en cuenta la orientación) del agente sonámbulo es la misma.

3.2. Primera aproximación a la búsqueda en anchura

Fijado el concepto de estado, pasamos a hacer una primera implementación del algoritmo de búsqueda en anchura intentando transcribir el pseudocódigo que se mostró anteriormente. El resultado sería el siguiente:

```

1  /** primera aproximación a la implementación de la búsqueda en anchura */
2  bool AnchuraSoloJugador(const stateN0 &inicio, const ubicacion &final,
3                          const vector<vector<unsigned char>> &mapa){
4      stateN0 current_state = inicio;
5      list<stateN0> frontier;
6      list<stateN0> explored;
7      frontier.push_back(current_state);
8      bool SolutionFound = (current_state.jugador.f == final.f and
9                          current_state.jugador.c == final.c);
10
11     while (!frontier.empty() and !SolutionFound) {
12         frontier.pop_front();
13         explored.push_back(current_state);
14
15         // Generar hijo actFORWARD
16         stateN0 child_forward = apply(actFORWARD, current_state, mapa);
17         if (child_forward.jugador.f == final.f and child_forward.jugador.c == final.c){
18             current_state = child_forward;
19             SolutionFound = true;
20         }
21         else if (!Find(child_forward, frontier) and !Find(child_forward, explored)){
22             frontier.push_back(child_forward);
23         }
24
25         if (!SolutionFound) {
26             // Generar hijo actTURN_L
27             stateN0 child_turnl = apply(actTURN_L, current_state, mapa);
28             if (!Find(child_turnl, frontier) and !Find(child_turnl, explored)){
29                 frontier.push_back(child_turnl);
30             }
31             // Generar hijo actTURN_R
32             stateN0 child_turnr = apply(actTURN_R, current_state, mapa);
33             if (!Find(child_turnr, frontier) and !Find(child_turnr, explored)){
34                 frontier.push_back(child_turnl);
35             }
36         }
37
38         if (!SolutionFound and !frontier.empty())
39             current_state = frontier.front();
40     }
41
42     return SolutionFound;
43 }

```

Describiremos poco a poco esta implementación empezando por los argumentos que se han considerado. Se puede observar que tiene 3 argumentos de entrada, el estado inicial llamado `inicio` que es de tipo `stateN0`, la casilla destino a donde debe llegar el agente jugador llamada `final` y que es de tipo `ubicacion` y por último el mapa sobre el que se realizará la búsqueda que se llama `mapa` y cuyo tipo es una matriz de `unsigned char`. La salida de la función es un valor lógico indicando si ha sido posible encontrar el plan.

Vamos a descomponer en 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. Empezamos viendo la estructura general

```

1  /** primera aproximación a la implementación de la búsqueda en anchura */
2  bool AnchuraSoloJugador(const stateN0 &inicio, const ubicacion &final,
3                          const vector<vector<unsigned char>> &mapa){
4      stateN0 current_state = inicio;
5      list<stateN0> frontier;
6      list<stateN0> explored;
7      frontier.push_back(current_state);
8      bool SolutionFound = (current_state.jugador.f == final.f and
9                             current_state.jugador.c == final.c);
10
11 > while (!frontier.empty() and !SolutionFound) {--
40 }
41
42     return SolutionFound;
43 }

```

Vemos que los datos que se usan son los siguientes:

`current_state`: almacena el estado actual y que inicialmente toma el valor del parámetro `inicio`.

`frontier`: una lista que mantiene los estados pendientes de explorar, inicialmente vacía.

`explored`: una lista que mantiene los estados ya explorados, inicialmente vacía.

`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_state` 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 11 a la línea 40) lo único que se hace es meter el estado 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 12 y 13], (2) generar todos los descendientes del estado actual [desde la línea 15 hasta la línea 36] y (3) tomar el siguiente valor de estado actual de la lista de estados pendientes de explorar [líneas 38 y 39].

```

11 while (!frontier.empty() and !SolutionFound){
12     frontier.pop_front();
13     explored.push_back(current_state);
14
15     // Generar hijo actFORWARD
16     stateN0 child_forward = apply(actFORWARD, current_state, mapa);
17     if (child_forward.jugador.f == final.f and child_forward.jugador.c == final.c){
18         current_state = child_forward;
19         SolutionFound = true;
20     }
21     else if (!Find(child_forward, frontier) and !Find(child_forward, explored)){
22         frontier.push_back(child_forward);
23     }
24
25     if (!SolutionFound){
26         // Generar hijo actTURN_L
27         stateN0 child_turnl = apply(actTURN_L, current_state, mapa);
28         if (!Find(child_turnl, frontier) and !Find(child_turnl, explored)){
29             frontier.push_back(child_turnl);
30         }
31         // Generar hijo actTURN_R
32         stateN0 child_turnr = apply(actTURN_R, current_state, mapa);
33         if (!Find(child_turnr, frontier) and !Find(child_turnr, explored)){
34             frontier.push_back(child_turnr);
35         }
36     }
37
38     if (!SolutionFound and !frontier.empty())
39         current_state = frontier.front();
40 }

```

- (1) 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 jugador, solo 3 acciones (las que implican que dicho agente se pueda desplazar) deben ser consideradas `actFORWARD`, `actTURN_L` y `actTURN_R`. 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 `actTURN_L` [\[líneas desde la 26 a la 30\]](#).

Primero se crea una nueva variable de tipo `stateN0` que almacenará el resultado de aplicar la acción correspondiente al estado actual y que en este caso se llama `child_turnl`. El cálculo de nuevo estado se hace a través de una función llamada `apply` que describiremos más tarde, pero que aquí asumiremos que nos devuelve el estado resultante de aplicar la acción.

Después 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 estado en la lista de estados pendientes de explorar. 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 estado 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 `actFORWARD` tiene algo más. Por la naturaleza de este problema, la acción `actFORWARD` 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 generará 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 [líneas de la 17 a la 20].

Por último, indicar que en la línea 25 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 se toma el nuevo valor para `current_state`. 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 `apply`. Existirían muchas posibilidades de implementación de esta función. En la siguiente subsección la dedicamos a describir una posible implementación de dicha función.

3.2.1. Generando los siguientes estados (`apply`)

En esta subsección se describe la implementación usada para construir la función `apply` que genera a partir del estado actual. La implementación hace uso de dos funciones auxiliares. La primera de ellas es la función `CasillaTransitable`.

```
1  /** Devuelve si una ubicación en el mapa es transitable para el agente */
2  bool CasillaTransitable(const ubicacion &x, const vector<vector<unsigned char>> &mapa)
3  {
4      return (mapa[x.f][x.c] != 'P' and mapa[x.f][x.c] != 'M');
5  }
```

Esta función toma dos argumentos de entrada, una ubicación `x` y una matriz bidimensional `mapa` y devuelve si la casilla determinada por la ubicación `x` es transitable en el mapa. La casilla se considera que es transitable si no es un precipicio ('P') ni un muro ('M').

La otra función de la que hace uso es `apply` es `NextCasilla`. Esta función toma como dato de entrada una ubicación y devuelve la ubicación de la casilla que estaría delante del agente en el sentido de su avance.

```

1  /** Devuelve la ubicación siguiente según el avance del agente*/
2  ubicacion NextCasilla(const ubicacion &pos)
3  {
4      ubicacion next = pos;
5      switch (pos.brujula)
6      {
7          case norte:
8              next.f = pos.f - 1;
9              break;
10         case noreste:
11             next.f = pos.f - 1;
12             next.c = pos.c + 1;
13             break;
14         case este:
15             next.c = pos.c + 1;
16             break;
17         case sureste:
18             next.f = pos.f + 1;
19             next.c = pos.c + 1;
20             break;
21         case sur:
22             next.f = pos.f + 1;
23             break;
24         case suroeste:
25             next.f = pos.f + 1;
26             next.c = pos.c - 1;
27             break;
28         case oeste:
29             next.c = pos.c - 1;
30             break;
31         case noroeste:
32             next.f = pos.f - 1;
33             next.c = pos.c - 1;
34             break;
35         default:
36             break;
37     }
38
39     return next;
40 }
41

```

A partir de las dos funciones anteriores, planteamos `apply` como una función con tres argumentos de entrada: la acción que se quiere realizar, el estado actual del agente y el mapa del juego y devuelve como cambia el estado actual tras realizar esa acción sobre el mapa elegido.

```

1  /** Devuelve el estado que se genera si el agente puede avanzar.
2  * Si no puede avanzar, se devuelve como salida el mismo estado de entrada.
3  */
4  stateN0 apply(const Action &a, const stateN0 &st, const vector<vector<unsigned char>> &mapa){
5      stateN0 st_result = st;
6      ubicacion sig_ubicacion;
7      switch (a){
8          case actFORWARD: // si proxima casilla es transitable y no está ocupada por el sonámbulo
9              sig_ubicacion = NextCasilla(st.jugador);
10             if (CasillaTransitable(sig_ubicacion, mapa) and
11                 !(sig_ubicacion.f == st.sonambulo.c and sig_ubicacion.c == st.sonambulo.c)){
12                 st_result.jugador = sig_ubicacion;
13             }
14             break;
15          case actTURN_L:
16              st_result.jugador.brujula = static_cast<Orientacion>((st_result.jugador.brujula + 6) % 8);
17              break;
18          case actTURN_R:
19              st_result.jugador.brujula = static_cast<Orientacion>((st_result.jugador.brujula + 2) % 8);
20              break;
21          }
22      return st_result;
23  }
24  }

```

La descripción de esta función es muy simple. En función de cada posible acción de las que se puede realizar (recordamos que para el nivel 0 solo 3 acciones permiten el movimiento del agente jugador) se calcula el estado resultado de aplicar dicha acción sobre el estado `st` almacenándose el resultado final en la variable `st_result` que se inicializa con el valor de `st` al principio de la función.

En el caso de los giros, el cálculo del estado generado es simple ya que la acción no tiene precondiciones y siempre se obtiene un estado “válido”. Podemos ver [\[en la línea 16\]](#) que consiste en aplicar una fórmula para cambiar la orientación del estado de partida sabiendo que implica dos pasos de 45 grados a la izquierda. De forma semejante se haría con el giro de 90 grados a la derecha.

El caso de la acción de avanzar si que puede implicar la generación de estados “inválidos”. Un estado será inválido si se intenta avanzar a una casilla no transitable o la casilla que en este momento está ocupada por el agente sonámbulo. Para evitar la generación de estos estados, se fija la condición de [la línea 11](#). Esto implica que cuando avanzar genera un estado inválido, se devuelve como siguiente estado el mismo estado de partida.

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 y una lista de estados y devuelve si está o no en la lista.

```

1  /** Encuentra si el elemento item está en lista */
2  bool Find(const stateN0 &item, const list<stateN0> &lista)
3  {
4      auto it = lista.begin();
5      while (it != lista.end() and !(*it) == item)
6          it++;
7
8      return !(it == lista.end());
9  }

```

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. 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 necesitamos invocar al algoritmo en el método `think`.

```

1  Action ComportamientoJugador::think(Sensores sensores)
2  {
3      Action accion = actIDLE;
4      if (!hayPlan)
5      {
6          cout << "Calculando un nuevo plan\n";
7          c_state.jugador.f = sensores.posF;
8          c_state.jugador.c = sensores.posC;
9          c_state.jugador.brujula = sensores.sentido;
10         c_state.sonambulo.f = sensores.SONposF;
11         c_state.sonambulo.c = sensores.SONposC;
12         c_state.sonambulo.brujula = sensores.SONsentido;
13         goal.f = sensores.destinoF;
14         goal.c = sensores.destinoC;
15
16         hayPlan = AnchuraSoloJugador(c_state, goal, mapaResultado);
17         if (hayPlan) cout << "Se encontró un plan\n";
18     }
19     if (hayPlan and plan.size() > 0)
20     {
21         cout << "Ejecutando la siguiente acción del plan\n";
22         accion = plan.front();
23         plan.pop_front();
24     }
25     if (plan.size() == 0)
26     {
27         cout << "Se completó el plan\n";
28         hayPlan = false;
29     }
30     return accion;
31 }

```

Esta sería la nueva configuración del método `think`, donde se modifica todo el bloque relativo al cálculo del nuevo plan [desde las líneas 6 a la 18]. Previamente a invocar al algoritmo de búsqueda [líneas de la 7 a la 14] se rellenan las variables `c_state` de tipo `stateN0` y `goal` de tipo `ubicacion` que almacenan el estado actual (ubicación del agente jugador y del agente sonámbulo) y la casilla destino respectivamente.

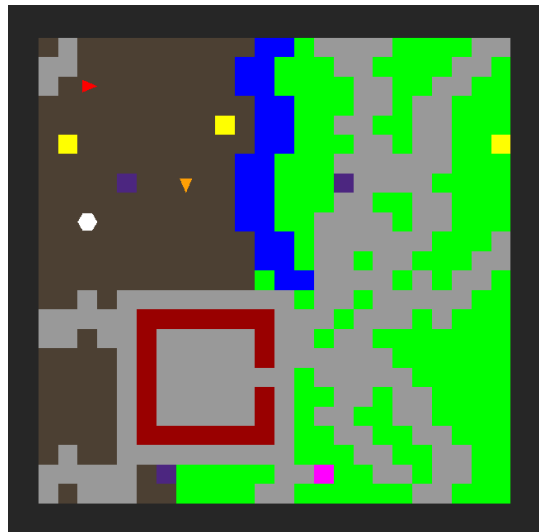
Las variables `c_state` y `goal` son variables de estado que se declaran en la parte privada de la clase `ComportamientoJugador` en el fichero `jugador.hpp`.

Por último, en línea 16 si invoca al algoritmo de búsqueda pasando como argumentos de entrada `c_state`, `goal` y `mapaResultado` (la matriz bidimensional donde se encuentra codificado el mapa) y se devuelve un valor lógico. En línea 17 se saca el mensaje “Se encontró un plan” si el resultado de invocar al método de búsqueda devuelve `true`.

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

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

Observamos que no aparece que nada haya cambiado incluso si continuamos pulsando el botón de “paso” nada parece cambiar. Solo podemos ver en la consola del terminar mensajes indicando que se calcula un nuevo plan y que se completó dicho plan.



La razón es que el algoritmo solo verifica que la casilla objetivo es alcanzable desde la casilla del agente jugador, pero no construye y devuelve la secuencia de acciones que llevan a que se produzca ese desplazamiento. Por tanto, necesitamos cambiar el algoritmo de búsqueda para que devuelva esa secuencia de acciones.

3.3. Segunda aproximación a la búsqueda en anchura

La primera implementación de la búsqueda en anchura ha dado como resultado un proceso que determina si es posible o no encontrar una secuencia de acciones para llevar al agente jugador desde su casilla a la casilla destino pero no devuelve el plan concreto que consigue eso.

En esta segunda aproximación incluiremos los cambios necesarios en la primera versión del algoritmo de búsqueda para devolver el plan de acciones cuando sea posible encontrar ese plan.

Antes de vamos a definir una función que nos permitirá ver gráficamente el plan que se ha obtenido sobre el simulador. Posteriormente incluiremos las modificaciones necesarias en el algoritmo de búsqueda para obtener el plan.

3.3.1. Una función para representar visualmente el plan en el simulador

Vamos a definir aquí una función que nos permitirán ver en la parte gráfica del simulador los planes que se vayan construyendo por parte del agente jugador. Para su implementación hemos hecho uso de una función auxiliar llamada AnularMatriz.

```
1  /** pone a cero todos los elementos de una matriz */
2  void AnularMatriz(vector<vector<unsigned char>> &matriz)
3  {
4      for (int i = 0; i < matriz.size(); i++)
5          for (int j = 0; j < matriz[0].size(); j++)
6              matriz[i][j] = 0;
7  }
```

La función AnularMatriz toma como entrada una matriz bidimensional de unsigned char que modifica para poner todas sus componentes a cero.

```
1  /** Permite pintar sobre el mapa del simulador el plan partiendo desde el estado st */
2  void ComportamientoJugador::VisualizaPlan(const stateN0 &st, const list<Action> &plan)
3  {
4      AnularMatriz(mapaConPlan);
5      stateN0 cst = st;
6
7      auto it = plan.begin();
8      while (it != plan.end()){
9          switch (*it){
10             case actFORWARD:
11                 cst.jugador = NextCasilla(cst.jugador);
12                 mapaConPlan[cst.jugador.f][cst.jugador.c] = 1;
13                 break;
14             case actTURN_R:
15                 cst.jugador.brujula = (Orientacion)((cst.jugador.brujula + 2) % 8);
16                 break;
17             case actTURN_L:
18                 cst.jugador.brujula = (Orientacion)((cst.jugador.brujula + 6) % 8);
19                 break;
20             case actSON_FORWARD:
21                 cst.sonambulo = NextCasilla(cst.sonambulo);
22                 mapaConPlan[cst.sonambulo.f][cst.sonambulo.c] = 2;
23                 break;
24             case actSON_TURN_SR:
25                 cst.sonambulo.brujula = (Orientacion)((cst.sonambulo.brujula + 1) % 8);
26                 break;
27             case actSON_TURN_SL:
28                 cst.sonambulo.brujula = (Orientacion)((cst.sonambulo.brujula + 7) % 8);
29                 break;
30             }
31             it++;
32         }
33 }
```

La función de VisualizaPlan tiene dos argumentos, un dato de tipo stateN0 que nos indica la ubicación de los dos agentes (jugador y sonámbulo) en el momento que fue elaborado el plan y un plan a realizar. El resultado es que la función modifica una matriz interna del simulador llamada mapaConPlan, del tamaño del mapa del juego donde el valor 0 indica que no será una casilla

transitada por el plan, el valor 1 que será una casilla transitada por el agente jugador y un 2 que será transitada por el agente sonámbulo.

La función recorre el plan sobre `mapaConPlan` para determinar las casillas que serán usadas por los dos agentes y las marca en esa matriz.

3.3.2. De estado a nodo

Normalmente, en la implementación de los algoritmos de búsqueda es necesario incluir más información que la del estado para poder realizar las tareas que indica dicho algoritmo. A esa implementación de un estado con más información se le conoce con el nombre de nodo y suele ser muy habitual, cuando se describe un algoritmo de búsqueda, usar el concepto de nodo.

En nuestro caso, usar solo la información del estado para el problema de búsqueda es lo que no nos permite almacenar la secuencia de acciones. Por esa razón, vamos a definir un nodo, que obviamente incluirá al estado, pero además información para construir el plan. Existen versiones que la información que incluyen para reconstruir la secuencia de acciones es un apuntador al padre que lo generó, de manera que cuando se encuentra el nodo con el estado solución, se construye el camino enlazando los padres desde dicha solución hasta el estado inicial. En lugar de esa opción, nosotros escogeremos una versión más simple que consiste en almacenar en cada nodo la secuencia de acción que se han aplicado para llegar a ese nodo. De esta manera, una vez que se encuentra el nodo solución, el plan es devolver justo esa secuencia de acciones.

```
1  /** Definición del tipo nodo para el nivel 0 */
2  struct nodeN0{
3      stateN0 st;
4      list<Action> secuencia;
5
6      bool operator==(const nodeN0 &n) const{
7          return (st == n.st);
8      }
9
10     bool operator<(const nodeN0 &n) const{
11         if (st.jugador.f < n.st.jugador.f)
12             return true;
13         else if (st.jugador.f == n.st.jugador.f and st.jugador.c < n.st.jugador.c)
14             return true;
15         else if (st.jugador.f == n.st.jugador.f and st.jugador.c == n.st.jugador.c and st.jugador.brujula < n.st.jugador.brujula)
16             return true;
17         else
18             return false;
19     }
20 };
21 |
```

Podemos observar que hemos definido `nodeN0` como un `struct` con 2 campos, `st` que codifica el estado y `secuencia` que almacenará la secuencia de acción que han sido necesarias desde el estado inicial para llegar al estado codificado en `st`.

Además definimos el operador de igualdad, indicando que dos nodos son iguales si los estados que codifican son iguales. También se ha definido el operador de menor estricto. La razón se verá más adelante. Aquí de momento solo indicar que se considera que dados 2 nodos, se considera menor el que tiene menos valor de fila en el agente jugador, a igualdad en el valor de fila el que tenga menor

valor de columna en el agente jugador y a igualdad en filas y columnas en la ubicación del agente jugador, el que la codificación ordinal de su orientación sea menor también en el agente jugador. Como se puede observar, la ubicación del agente sonámbulo se considera irrelevante para determinar la diferencia entre dos estados. La razón es que en el nivel 0, la ubicación del agente sonámbulo solo se incluye para evitar que el agente jugador acceda a la casilla que ocupa el sonámbulo, pero el agente sonámbulo en sí no cambiará su posición o no al menos por una acción realizada por el agente jugador.

Dada la definición anterior de nodo, la segunda versión que se propone del algoritmo de búsqueda consiste en modificar la primera versión para incluir en el lugar de `stateN0`, `nodeN0` y con ello poder tener el plan. Obviamente, otros cambios son necesarios. Las listas que inicialmente eran de `stateN0` ahora pasan a ser de `nodeN0`, y por tanto, para buscar estados dentro de las listas de nodos tendremos que sobrecargar la función `Find` que se encarga de encontrar si estado está o no en la lista de nodos pendientes de explorar o nodos explorados.

```
1  /** Sobrecarga de la función Find para buscar en lista de nodeN0*/
2  bool Find(const stateN0 &item, const list<nodeN0> &lista)
3  {
4      auto it = lista.begin();
5      while (it != lista.end() and !(it->st == item))
6          it++;
7      return !(it == lista.end());
8  }
```

Se puede observar que la función es una reformulación de la inicial sabiendo ahora que la lista es de nodos y que el estado se codifica en el campo `st` de dicho nodo. Al igual que la inicial, el orden de complejidad es de $O(n)$ siendo n el tamaño de la lista.

3.3.3. Segunda versión del algoritmo de búsqueda en anchura

Ya tenemos todos los elementos para plantear la segunda versión del algoritmo de búsqueda en anchura y en este caso si que podremos devolver la secuencia de acciones si el plan se encuentra.

```

1  /** Version segunda de la búsqueda en anchura: ahora sí devuelve un plan*/
2  list<Action> AnchuraSoloJugador_V2(const stateN0 &inicio, const ubicacion &final,
3                                     const vector<vector<unsigned char>> &mapa){
4      nodeN0 current_node;
5      list<nodeN0> frontier;
6      list<nodeN0> explored;
7      list<Action> plan;
8      current_node.st = inicio;
9      bool SolutionFound = (current_node.st.jugador.f == final.f and
10                             current_node.st.jugador.c == final.c);
11      frontier.push_back(current_node);
12
13 > while (!frontier.empty() and !SolutionFound){ ...
50
51
52     if (SolutionFound)
53         plan = current_node.secuencia;
54
55     return plan;
56 }

```

La primera novedad que se puede ver es que ahora la función devuelve una lista de acciones en lugar de un valor lógico, es decir, que ahora el resultado de ejecutar la función será la secuencia de acciones si es posible encontrar un plan.

En relación a la declaración de los datos que se usarán en el algoritmo, vemos que lo que era el estado actual (`current_state`) es sustituido por el nodo actual (`current_node`) que inicializa su campo `st` con el estado inicial (inicio, [línea 8](#)). Las listas de estados se han convertido en listas de nodos tanto `frontier`, para los estados pendientes de ser explorados, como `explored`, con los estados ya explorados). Como en la versión primera, el nodo inicial (antes estado inicial) se inserta en `frontier` y se inicializaba la variable `SolutionFound` mirando si ya el estado inicial es solución.

Se puede observar que el ciclo principal se sigue manteniendo con las mismas condiciones y más adelante describiremos los cambios que básicamente están relacionados con la adaptación del paso de estado a nodo.

En la parte final del algoritmo ahora sí que podemos ver un cambio. Cuando se encuentra solución se devuelve el plan contenido en el campo `secuencia` del nodo encontrado como solución.

```

13 while (!frontier.empty() and !SolutionFound){
14     frontier.pop_front();
15     explored.push_back(current_node);
16
17     // Generar hijo actFORWARD
18     nodeN0 child_forward = current_node;
19     child_forward.st = apply(actFORWARD, current_node.st, mapa);
20     if (child_forward.st.jugador.f == final.f and child_forward.st.jugador.c == final.c)
21     {
22         child_forward.secuencia.push_back(actFORWARD);
23         current_node = child_forward;
24         SolutionFound = true;
25     }
26     else if (!Find(child_forward.st, frontier) and !Find(child_forward.st, explored)){
27         child_forward.secuencia.push_back(actFORWARD);
28         frontier.push_back(child_forward);
29     }
30
31     if (!SolutionFound){
32         // Generar hijo actTURN_L
33         nodeN0 child_turnl = current_node;
34         child_turnl.st = apply(actTURN_L, current_node.st, mapa);
35         if (!Find(child_turnl.st, frontier) and !Find(child_turnl.st, explored)){
36             child_turnl.secuencia.push_back(actTURN_L);
37             frontier.push_back(child_turnl);
38         }
39         // Generar hijo actTURN_R
40         nodeN0 child_turnr = current_node;
41         child_turnr.st = apply(actTURN_R, current_node.st, mapa);
42         if (!Find(child_turnr.st, frontier) and !Find(child_turnr.st, explored)){
43             child_turnr.secuencia.push_back(actTURN_R);
44             frontier.push_back(child_turnr);
45         }
46     }
47
48     if (!SolutionFound and !frontier.empty())
49         current_node = frontier.front();
50 }

```

Mirando ahora el contenido del cuerpo del ciclo principal del proceso de búsqueda podemos observar que la estructura se mantiene con las 3 partes: (1) la actualización de las listas de frontier y explored que se mantienen igual [líneas 14 y 15], (2) el cálculo de aplicar todos las acciones al estado actual para generar todos los siguientes estados que básicamente permanece igual [líneas de la 17 a la 46] y (3) y sacar el siguiente estado actual de la lista frontier si no se ha encontrado solución y la lista no está vacía [líneas 48 y 49].

En realidad solo se ha modificado de la parte del cálculo de la siguiente acción la adaptación a que ahora current_node es de tipo nodeN0 y no de tipo stateN0. Por eso en línea 33 child_turnl es de tipo nodeN0 y se le asigna el actual nodo, en línea 34 la generación del nuevo estado sigue haciendo uso de la función apply, pero se le pasa como argumento solo el estado (el campo st de nodo) y se asigna también al campo st y en línea 35 se hace uso de la función Find que sobrecargamos para buscar sobre listas de nodos.

En línea 36 si aparece una verdadera novedad que es la actualización del campo secuencia del nodo. En ella se añade a la secuencia que venía del padre esta nueva acción que acabamos de aplicar. Esto se hace cuando ese estado no había sido generado previamente.

Hemos explicado las modificaciones en el caso de aplicar la acción actTURN_L, para actFORWARD y actTURN_R las modificaciones son semejantes.

3.3.4. Probando la segunda implementación de la búsqueda en anchura

Ya tenemos codificado completamente el algoritmo de búsqueda en anchura y en este caso nos devolverá la secuencia de acciones si encuentra solución. Ahora pasaremos a probarlo: en el método think sustituimos la anterior llamada de

```
hayPlan = AnchuraSoloJugador(c_state, goal, mapaResultado);  
if (hayPlan) cout << "Se encontró un plan\n";
```

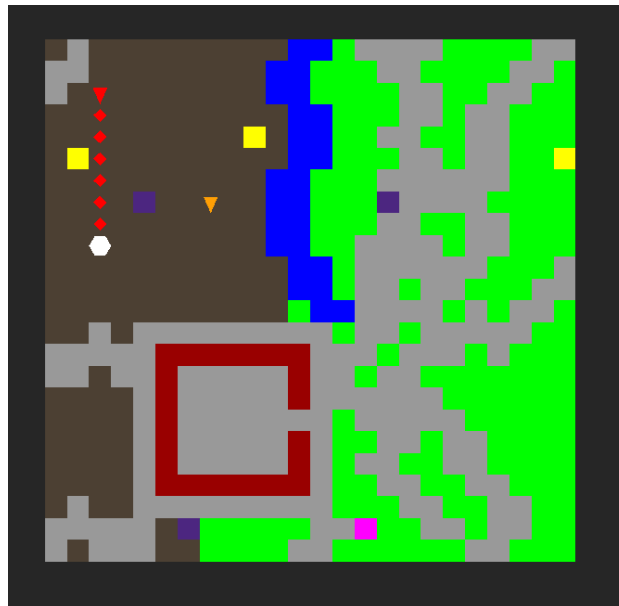
por

```
plan = AnchuraSoloJugador_V2(c_state, goal, mapaResultado);  
VisualizaPlan(c_state, plan);  
hayPlan = true;
```

compilamos y ejecutamos

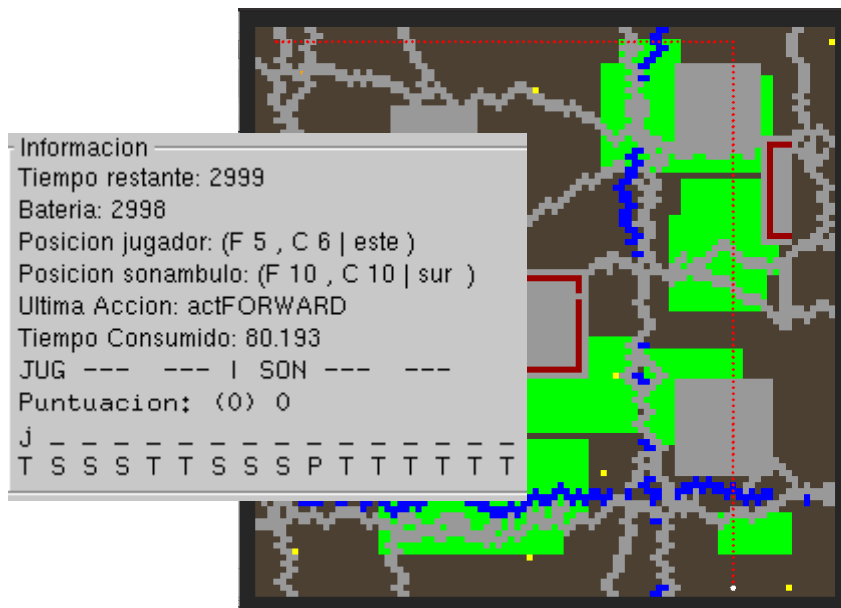
```
./practica2 mapas/mapa30.map 1 0 5 5 2 10 10 4 12 5
```

y obtenemos el siguiente resultado, un plan de longitud 10.



Repetimos la ejecución pero ahora con la siguiente llamada

```
./practica2 mapas/mapa100.map 1 0 5 5 2 10 10 4 95 80
```



En este caso, se tardan unos 80 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 168 acciones.

3.4. Tercera versión de la búsqueda en anchura: acelerando la búsqueda

En esta tercera y última 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 documento 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 en anchura requiere.

El tipo de dato set de la STL de C++ requiere definir una operación 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 `nodeN0`, definimos el operador menor estricto. Es este operador el que usará el tipo set para ordenar los elementos dentro de la estructura. Recordemos como estaba definido:

```
bool operator<(const nodeN0 &n) const{
    if (st.jugador.f < n.st.jugador.f)
        return true;
    else if (st.jugador.f == n.st.jugador.f and st.jugador.c < n.st.jugador.c)
        return true;
    else if (st.jugador.f == n.st.jugador.f and st.jugador.c == n.st.jugador.c and st.jugador.brujula < n.st.jugador.brujula)
        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 jugador de un nodo es menor que la fila del agente jugador del otro nodo. También son distintos si siendo lo anterior igual, la columna del agente jugador de un nodo es menor que la columna del agente jugador del otro nodo. Por último, son distintos si siendo las filas y las columnas iguales para los dos nodos, la orientación del agente jugador de un nodo es menor que la del agente jugador del otro nodo. 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 `nodeN0` sobre el que no se hará ninguna búsqueda y `explored` será un set de `nodeN0` y solo sobre él se realizarán todas las búsquedas para encontrar los nodos repetidos.

```
1  /** Version tercera de la búsqueda en anchura: ahora sí devuelve un plan*/
2  list<Action> AnchuraSoloJugador_V3(const stateN0 &inicio, const ubicacion &final,
3                                     const vector<vector<unsigned char>> &mapa){
4      nodeN0 current_node;
5      list<nodeN0> frontier;
6      set<nodeN0> explored;
7      list<Action> plan;
8      current_node.st = inicio;
9      bool SolutionFound = (current_node.st.jugador.f == final.f and
10                           current_node.st.jugador.c == final.c);
11      frontier.push_back(current_node);
12
13 > while (!frontier.empty() and !SolutionFound){
14     ...
15 }
16
17 if (SolutionFound)
18     plan = current_node.secuencia;
19
20 return plan;
21 }
```

Mirando la estructura más externa podemos observar que solo hay un cambio con respecto a la versión 2 y es en la línea 6 donde ahora `explored`, la lista de nodos ya explorados, se define de tipo set.

```

13 while (!frontier.empty() and !SolutionFound){
14     frontier.pop_front();
15     explored.insert(current_node);
16
17     // Generar hijo actFORWARD
18     nodeN0 child_forward = current_node;
19     child_forward.st = apply(actFORWARD, current_node.st, mapa);
20     if (child_forward.st.jugador.f == final.f and child_forward.st.jugador.c == final.c){
21         child_forward.secuencia.push_back(actFORWARD);
22         current_node = child_forward;
23         SolutionFound = true;
24     }
25     else if (explored.find(child_forward) == explored.end()){
26         child_forward.secuencia.push_back(actFORWARD);
27         frontier.push_back(child_forward);
28     }
29
30     if (!SolutionFound){
31         // Generar hijo actTURN_L
32         nodeN0 child_turnl = current_node;
33         child_turnl.st = apply(actTURN_L, current_node.st, mapa);
34         if (explored.find(child_turnl) == explored.end()){
35             child_turnl.secuencia.push_back(actTURN_L);
36             frontier.push_back(child_turnl);
37         }
38         // Generar hijo actTURN_R
39         nodeN0 child_turnr = current_node;
40         child_turnr.st = apply(actTURN_R, current_node.st, mapa);
41         if (explored.find(child_turnr) == explored.end()){
42             child_turnr.secuencia.push_back(actTURN_R);
43             frontier.push_back(child_turnr);
44         }
45     }
46
47     if (!SolutionFound and !frontier.empty()){
48         current_node = frontier.front();
49         while (!frontier.empty() and explored.find(current_node) != explored.end()){
50             frontier.pop_front();
51             if (!frontier.empty())
52                 current_node = frontier.front();
53         }
54     }
55 }

```

Las cosas que cambian dentro del ciclo principal del algoritmo de búsqueda en anchura [desde la línea 13 a la 55] 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 15.
- no se busca en las dos listas sino que solo se busca en la lista de explored y eso se expresa en las líneas 25, 34 y 41.
- el proceso para elegir el siguiente nodo actual [líneas de la 47 a 54]. 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 dicha lista. 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

```
hayPlan = AnchuraSoloJugador_V2(c_state, goal, mapaResultado);
```

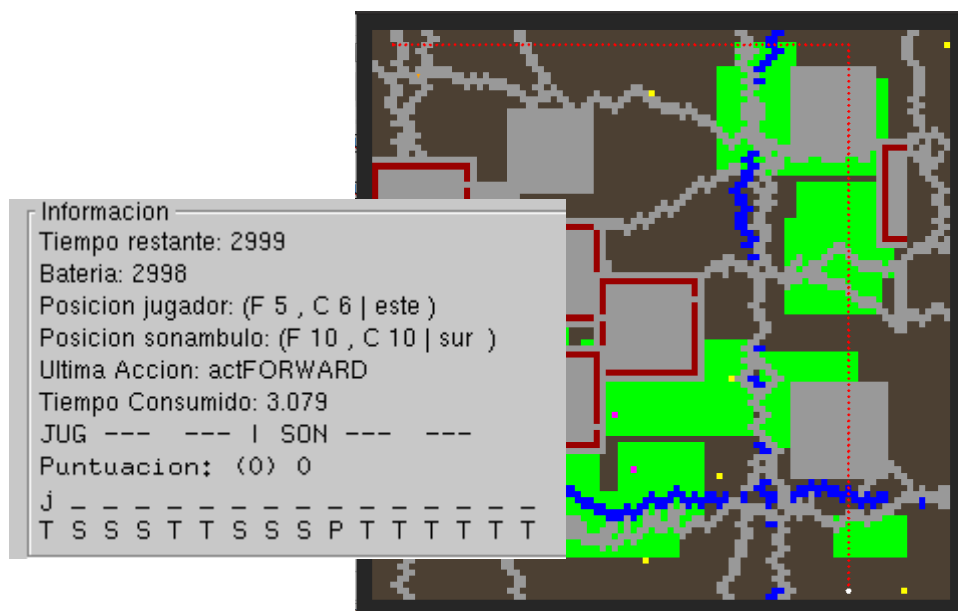
por

```
hayPlan = AnchuraSoloJugador_V3(c_state, goal, mapaResultado);
```

compilamos y ejecutamos

```
./practica2 mapas/mapa100.map 1 0 5 5 2 10 10 4 95 80
```

obtenemos



Se puede observar que el tiempo que se ha obtenido es de unos 3 segundos frente a los 80 que se tardó en la versión anterior. En cuanto al número de acción que implicaba esta secuencia de acciones es de 168.

4. Estructurando el método think

La sección anterior la hemos dedicado a describir como desarrollar el nivel 0 de la práctica 2 partiendo del software que se puede descargar desde <https://github.com/ugr-ccia-IA/practica1>. Pero el nivel 0 es sólo uno de los niveles que el estudiante tiene que resolver de la práctica 2 y por esa razón vamos a proponer una estructura del método think que permita añadir con facilidad la

```
1 Action ComportamientoJugador::think(Sensores sensores)
2 {
3     Action accion = actIDLE;
4     if (sensores.nivel != 4)
5     {
6         if (!hayPlan)
7         {
8             cout << "Calculando un nuevo plan\n";
9             c_state.jugador.f = sensores.posF;
10            c_state.jugador.c = sensores.posC;
11            c_state.jugador.brujula = sensores.sentido;
12            c_state.sonambulo.f = sensores.SONposF;
13            c_state.sonambulo.c = sensores.SONposC;
14            c_state.sonambulo.brujula = sensores.SONsentido;
15            goal.f = sensores.destinoF;
16            goal.c = sensores.destinoC;
17            switch (sensores.nivel)
18            {
19                case 0:
20                    plan = AnchuraSoloJugador_V3(c_state, goal, mapaResultado);
21                    break;
22                case 1: // Incluir aqui la llamada al algoritmo de busqueda para el nivel 1
23                    cout << "Pendiente de implementar el nivel 1\n";
24                    break;
25                case 2: // Incluir aqui la llamada al algoritmo de busqueda para el nivel 2
26                    cout << "Pendiente de implementar el nivel 2\n";
27                    break;
28                case 3: // Incluir aqui la llamada al algoritmo de busqueda para el nivel 3
29                    cout << "Pendiente de implementar el nivel 3\n";
30                    break;
31            }
32            if (plan.size() > 0)
33            {
34                VisualizaPlan(c_state, plan);
35                hayPlan = true;
36            }
37        }
38        if (hayPlan and plan.size() > 0)
39        {
40            cout << "Ejecutando la siguiente acción del plan\n";
41            accion = plan.front();
42            plan.pop_front();
43        }
44        if (plan.size() == 0)
45        {
46            cout << "Se completó el plan\n";
47            hayPlan = false;
48        }
49    }
50    else
51    {
52        // Incluir aqui la solución al nivel 4
53    }
54    return accion;
55 }
```

implementaciones del resto de niveles que quedan por completar.

Los niveles 1, 2 y 3 tienen un enunciado muy parecido al del nivel 0 y por tanto tienen una forma de resolución muy parecida, al punto, que resolverlos consiste en proponer un nuevo módulo (una función) con una estructura muy parecida a `AnchuraSoloJugador_V3`.

Sin embargo el nivel 4 si que puede ser muy diferente a los demás, ya que será necesario combinar comportamientos reactivos con deliberativos teniendo en cuenta que ahora el entorno es dinámico y que hace que pueda que los planes no se puedan terminar con éxito.

Por tanto, para enganchar las soluciones de los niveles del 1 al 3 con el programa principal, bastaría asignar [en la líneas 23, 26 y 29](#) la llamada al módulo que resuelve el nivel 1, 2 y 3 respectivamente. La llamada sería semejante a la del nivel 0, es decir, asignar a la variable de estado plan la secuencia de acciones que se obtienen para llevar al agente correspondiente dependiendo del nivel a la ubicación destino.

La resolución del nivel 4 se hara en el bloque del else de [la línea 50](#).

Decir que usar esta organización del método think no es obligatoria, solo se da como orientación. Cualquier otra organización sensata, que respete los criterios de las buenas prácticas de programación y que lleve a conseguir las soluciones que se plantean en los distintos niveles obviamente también será válida.

5. Comentarios Finales

Este tutorial tiene como objetivo dar un pequeño empujón en el inicio del desarrollo de la práctica y lo que se propone es sólo una forma de dar respuesta (la más básica en todos los casos) a los problemas con los que os tenéis que enfrentar. Por tanto, todo lo que se propone aquí es mejorable y lo debéis mejorar.

Muchos elementos que forman parte de la práctica no se han tratado en este tutorial. Esos elementos son relevantes para mejorar la capacidad del agente e instamos a que se les preste atención.

Por último, resaltar que la práctica es individual y que la detección de copias (trozos de código iguales o muy parecidos entre estudiantes) implicará el suspenso en la asignatura.