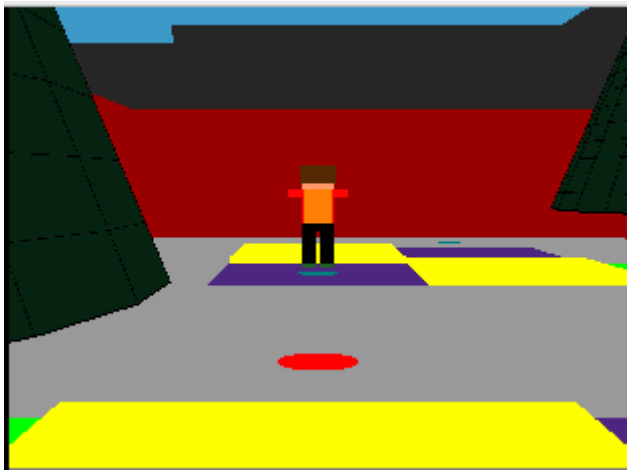
INTELIGENCIA ARTIFICIAL

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

Tutorial: Práctica 2 (Parte 1)

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 que tienen como misión reconocer y desplazarse por un mundo bidimensional cuadriculado. 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 se pueden considerar dos partes claramente diferenciadas. En los niveles 0 y 1 el estudiante tiene que trabajar con comportamientos reactivos, mientras que el resto de los niveles implica trabajar con procesos deliberativos. En este tutorial nos centraremos en los niveles reactivos.

El objetivo de la primera parte de la práctica es definir comportamientos reactivos para agentes que tienen capacidad de moverse sobre un terreno y cuya misión es descubrir los elementos que forman parte de ese terreno que le rodea. Aconsejamos que la construcción de dicho comportamiento se realice de forma incremental y mediante refinamiento paulatino. Si seguís este consejo, el proceso de construcción tendría la siguiente secuencia de subobjetivos:

1. Establecer una movilidad básica para el agente que le permita navegar por el mundo, inicialmente manteniéndose móvil y posteriormente ir mejorando su capacidad para descubrir zonas aún no exploradas.
2. Utilizar la matriz **mapaResultado** y **mapaCotas** para ir almacenando lo que voy viendo e ir cartografiando el terreno por el que el agente se va desplazando.
3. Definir qué comportamiento debe tener el agente frente a las otras entidades móviles del mundo.
4. Refinamiento del modelo general, para mejorar la capacidad de exploración del agente.

Este tutorial intenta ser una ayuda para poner en marcha la práctica y se centra en una aproximación inicial del subobjetivo 1 antes mencionado, es decir, dar los primeros pasos sobre el mundo.

Antes de nada, y para que el comportamiento que se defina esté más o menos estructurado y sea fácil de seguir, depurar y mejorar, es necesario entender que en el método que define el proceso (en nuestro caso, el método **think**[1]) se pueden considerar 2 fases:

(a) una primera fase de observación donde se actualizan los cambios que la última acción realizada provocó sobre el mundo
(b) una segunda fase para determinar la siguiente acción a realizar.

La segunda fase es realmente el comportamiento, pero la primera podría ser muy importante, ya que es necesaria para garantizar que la información usada para la toma de la decisión es correcta. La segunda fase está compuesta (en nuestro caso) por todas las reglas que guiarán el comportamiento del agente y que tendrán una estructura del tipo:

Regla 1:	**if (condicion^{1}){**
	}
Regla 2:	**else if (condicion^{2}){**
	}
.....	**.....**
.	
Regla n-1	**else if (condicion^{n-1}){**
	}
Regla por defecto	**else{**
	}

donde **condicion^{i}** es una condición lógica formada por los ***sensores*** o por las ***variables de estado*** definidas.

Sabemos que los sensores son los valores que recibimos directamente del mundo que nos rodea y cambian en cada iteración de la simulación y por consiguiente solo tienen vigencia en una iteración. Si queremos que nuestro agente tome una decisión con base en algo que no se está percibiendo en este instante, pero que sí percibió en algún instante anterior, tenemos que hacer uso de las ***variables de estado***. Las variables de estado son la memoria que tendrá el agente.

Sabemos que las variables de estado se usan formando parte de alguna o algunas condiciones de las reglas que compondrán el comportamiento del agente, ¿pero cómo se declaran?

En esta práctica en concreto la forma de trabajar con una variable de estado sería la siguiente:

- En el fichero "***.hpp***" del agente correspondiente, debajo de la parte que pone "***private:***" se declaran tantas variables como sean necesarias en la forma ***<tipo> <identificador>;***

```
Action last_action;
bool tiene_zapatillas;
int giro45Izq;
```

[1]Para aquellos que acepten la idea de construir cada nivel en un módulo independiente como se propone en el software que se proporciona, sería la estructura de cada uno de esos módulos.

- En ese mismo fichero, en el constructor de la clase se inicializan. Hay dos constructores. El constructor que tiene como parámetro ***unsigned int size*** es el que usará para los niveles 0, 1 y 4. El constructor que tiene dos parámetros de tipo ***vector<vector<unsigned char>>*** es el que se usa en los niveles 2 y 3. En este tutorial trabajaremos sobre el nivel 0, por eso, usaremos el constructor que le corresponde.

```
last_action = IDLE;
tiene_zapatillas = false;
giro45Izq = 0;
```

- Ya en el fichero ".cpp", en el módulo correspondiente a la definición del comportamiento, en la parte dedicada a la actualización del mundo, se modifican los valores de las variables de estado en función de la acción realizada.

```
if (sensores.superficie[0] == 'D') {
    tiene_zapatillas = true;
}
```

- En la segunda parte de ese módulo se usa para determinar la siguiente acción a realizar.

```
if (giro45Izq != 0){
    action = TURN_SR;
    giro45Izq --;
}
    ……….....
last_action = accion;
```

Con esta primera información básica, ya podemos empezar a construir los primeros comportamientos básicos de alguno de los agentes.

2. Los primeros pasos

Vamos a abordar el primero de los subobjetivos que hemos definido en la introducción "*movilidad básica"* y hacer que nuestro agente se mueva y lo vamos a hacer siguiendo las consideraciones que nos propone el nivel 0 de la práctica que consiste en moverse exclusivamente por casillas de tipo `camino 'C'`.

El comportamiento consistirá en mirar las casillas que quedan justo delante en el cono de visión del agente `Rescatador` y avanzar a la casilla de entre esas que sea de tipo camino. Cómo es posible que haya varias de tipo camino, establecemos como orden de prioridad que primero queremos avanzar a la que está justo delante (coordenada 2 del vector de visión), después la que está un poco a la izquierda (coordenada 1 del vector de visión) y después la que está un poco a la derecha (coordenada 3).
Una descripción en lenguaje natural de lo que nos gustaría ver en la evolución del agente sería que:

1. Si existe una casilla camino delante, ejecutará la acción `WALK` y pasará a ocupar dicha casilla.
2. Si es la casilla de un paso adelante, un paso a la derecha (coordenada 3 del vector de visión) la única que tuviera una casilla de tipo camino, esperaríamos que el agente hiciera en un primer instante la acción `TURN_SR`. De esta forma, en el instante siguiente estaríamos en la situación anterior.
3. Si es la de la izquierda (índice 1 del vector de visión) debería orientarse hacía dicha casilla, pero el agente no tiene ninguna acción que permita hacer un giro a la izquierda de 45 grados.
4. Si no se da ninguna de las anteriores, ejecutar la acción `TURN_L` esperando que cambiando de orientación se encuentren nuevas casillas de tipo camino.

De toda la descripción anterior, solo lo que se debe hacer en el punto 3 es lo que queda por especificar. Está claro que se puede definir la acción "`TURN_SL`", giro de 45 grados a la izquierda mediante la composición de dos acciones, en un primer instante `TURN_L` y después `TURN_SR`. Con esta intención usaremos una variable de estado que llamaremos `giro45Izq` que nos permitirá hacer esa composición de acciones.

Además, definiremos la variable de estado `tiene_zapatillas` para almacenar si pasó por una casilla de tipo zapatillas y también una variable de estado que almacenará la última acción que realizó el agente. La forma en la que se realiza este proceso, ya se describió en el apartado anterior y el fichero "**rescatador.hpp**" debería quedar como muestra la Figura 1.

```
1   #ifndef COMPORTAMIENTORESCATADOR_H
4   #include <chrono>
6   #include <thread>
7
8   #include "comportamientos/comportamiento.hpp"
9
10  class ComportamientoRescatador : public Comportamiento
11  {
12
13  public:
14    ComportamientoRescatador(unsigned int size = 0) : Comportamiento(size)
15    {
16      // Inicializar Variables de Estado Niveles 0,1,4
17      last_action = IDLE;
18      tiene_zapatillas = false;
19      giro45Izq = 0;
20    }
21    ComportamientoRescatador(std::vector<std::vector<unsigned char>> mapaR, std::vector<std::vector
22    {
23      // Inicializar Variables de Estado Niveles 2,3
24    }
25    ComportamientoRescatador(const ComportamientoRescatador &comport) : Comportamiento(comport) {}
26    ~ComportamientoRescatador() {}
27
28    Action think(Sensores sensores);
29
30    int interact(Action accion, int valor);
31
32    Action ComportamientoRescatadorNivel_0(Sensores sensores);
33    Action ComportamientoRescatadorNivel_1(Sensores sensores);
34    Action ComportamientoRescatadorNivel_2(Sensores sensores);
35    Action ComportamientoRescatadorNivel_3(Sensores sensores);
36    Action ComportamientoRescatadorNivel_4(Sensores sensores);
37
38  private:
39    // Variables de Estado
40    Action last_action;       //Almacena la última acción realizada por el agente
41    bool tiene_zapatillas;    //Indica si ya pasó por una casilla de tipo zapatillas
42    int giro45Izq;            //Indica si está haciendo un TURN_SL. (0 indica que no)
43
44  };
45
46  #endif
```

Figura 1: Declaración e inicialización de las variables de estado.

Pasamos a definir el comportamiento que indicamos antes en el fichero "**rescatador.cpp**", y lo haremos en el método `ComportamientoRescatadorNivel_0`. No olvidar, en el método `think` descomentar la llamada a este método en el caso 0 de la sentencia `switch`.

```
1   #include "../Comportamientos_Jugador/rescatador.hpp"
2   #include "motorlib/util.h"
3
4   Action ComportamientoRescatador::think(Sensores sensores)
5   {
6       Action accion = IDLE;
7
8       switch (sensores.nivel)
9       {
10      case 0:
11          accion = ComportamientoRescatadorNivel_0 (sensores);
12          break;
13      case 1:
14          // accion = ComportamientoRescatadorNivel_1 (sensores);
15          break;
16      case 2:
17          // accion = ComportamientoRescatadorNivel_2 (sensores);
18          break;
19      case 3:
20          // accion = ComportamientoRescatadorNivel_3 (sensores);
21          break;
22      case 4:
23          // accion = ComportamientoRescatadorNivel_4 (sensores);
24          break;
25      }
26
27      return accion;
28  }
```

Figura 2: Descomentar el caso 0 de la instrucción `switch`

La implementación de la primera versión del comportamiento que sigue caminos se muestra en la figura 3 y en ella podemos distinguir las dos partes que ya se indicaban en la introducción de este tutorial: se empieza con la parte de actualización de información en función del resultado de la última acción realizada (en este caso, solo ver si se cayó en una casilla de tipo zapatillas), y después aparece la parte que desarrolla el comportamiento en sí (que va desde la línea 43 a la línea 63).

```
35  Action ComportamientoRescatador::ComportamientoRescatadorNivel_0(Sensores sensores)
36  {
37      Action accion;
38      // El comportamiento de seguir un camino hasta encontrar un puesto base.
39      // Actualización de variables de estado
40      if (sensores.superficie[0] == 'D') tiene_zapatillas = true;
41
42
43      // Definición del comportamiento
44      if (sensores.superficie[0] == 'X'){     // Llegué al Objetivo
45          accion = IDLE;
46      }
47      else if (giro45Izq != 0){               // Estoy haciendo un TURN_SL
48          accion = TURN_SR;
49          giro45Izq--;
50      }
51      else if (sensores.superficie[2] == 'C'){  //
52          accion = WALK;
53      }
54      else if (sensores.superficie[1] == 'C'){
55          giro45Izq = 1;
56          accion = TURN_L;
57      }
58      else if (sensores.superficie[3] == 'C'){
59          accion = TURN_SR;
60      }
61      else {
62          accion = TURN_L;
63      }
64
65
66      // Devolver la siguiente acción a hacer
67      last_action = accion;
68      return accion;
69  }
```

Figura 3: Primera versión del comportamiento de seguir caminos para el `Rescatador`.

La línea 40 indica que si el agente, en el instante actual se encuentra encima de una casilla de tipo zapatilla (recordar que la posición 0 del vector de visión es la información sobre la casilla en la que se encuentra el agente y que el vector `superficie` es el que indica los accidentes geográficos), se adquiere el objeto `zapatillas`.

Se puede observar que el comportamiento mantiene una estructura del tipo "`if-then-else`" tal y como ya se indicaba en la introducción. Esta estructura puede ser cómoda para definir, pero sobre todo para depurar el comportamiento.

Una descripción rápida en lenguaje natural de las reglas que describen el comportamiento sería:

- Línea 44, si el agente está en una casilla de tipo `'X'` (puesto base) se ha cumplido el objetivo, por lo tanto, "no hagas nada".

- Línea 47, es la regla asociada a la composición de acciones que se deben hacer para generar la acción "`TURN_SL`". Cuando la variable de estado `giro45Izq` no vale 0, indica que aún queda para terminar la acción hacer un `TURN_SR`. Se decrementa `giro45Izq` para indicar que ya se terminó la acción compuesta "`TURN_SL`".

- Línea 51, es la regla de avanzar si delante hay una casilla de tipo camino.

- Línea 54, es la regla para hacer la acción de "`TURN_SL`" a partir de la composición de acciones primitivas. Aquí inicia la primera de las dos acciones necesarias, `TURN_L`, y pone la variable de estado `giro45Izq` a 1, para encadenarla después con la regla de la línea 47.

- Línea 58, si hay una casilla de tipo camino en la parte derecha, se orienta para intentar avanzar por esa casilla.

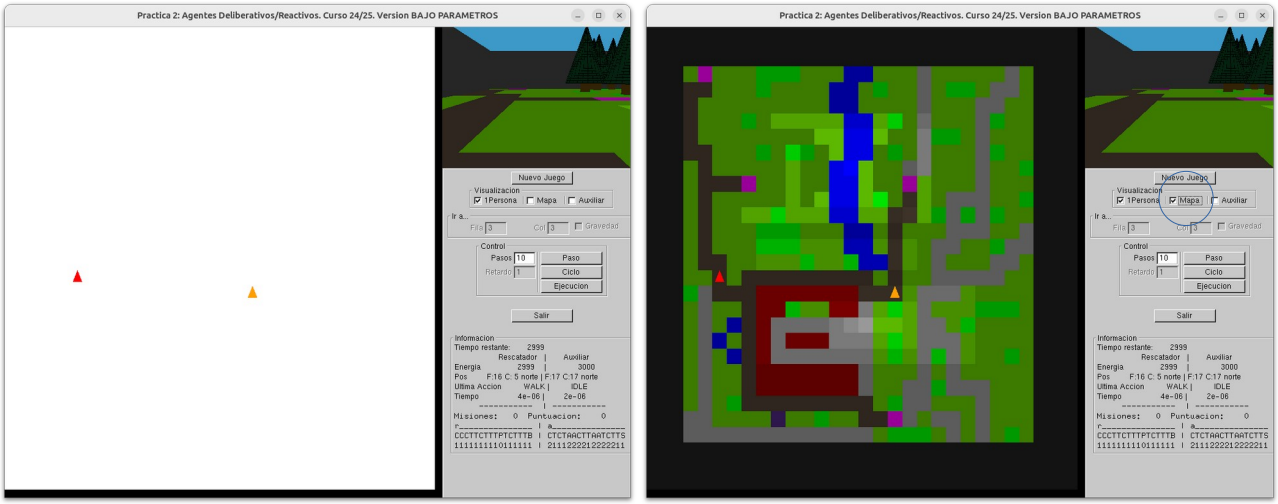
- Por último, en la línea 61 se establece la regla por defecto que indica que si ninguna de las anteriores es cierta es porque no hay camino delante en cuyo caso gira hacía un lado para buscar casillas de tipo camino.

En la última parte se devuelve la acción seleccionada, pero antes se almacena en la variable de estado `last_action`, la acción que se va a enviar para su ejecución.

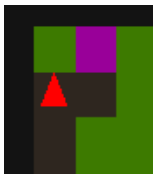
Ahora pasaremos a probar el funcionamiento de este comportamiento, para ello ejecutar desde un línea de comandos

```
./practica2 ./mapas/mapa30.map 0 0 17 5 0 17 17 0 3 3 0
```

Una vez levantado el entorno, activar el check "mapa" para visualizar el mapa y pulsar el botón de "Paso" para ver como el agente `Rescatador` va avanzando por el camino.



Se puede observar que llega a la situación donde puede ver en su línea de visión en la posición 3 una casilla `puesto base`, pero ignora orientarse hacia ella. De hecho, si se continúa con la simulación, se verá que nunca accede a una casilla que sea distinta de una casilla de tipo `camino`. La razón es simple, en el comportamiento no se considera acceder a casillas `puesto base`. Parece razonable que las casillas `puesto base` tengan prioridad sobre las casillas de tipo `camino`.



Con esa intención, pasamos a modificar el código creando la función `VeoCasillaInteresanteR`:

```
35  int VeoCasillaInteresanteR (char i, char c, char d){
36      if (c == 'X') return 2;
37      else if (i == 'X') return 1;
38      else if (d == 'X') return 3;
39      else if (c == 'C') return 2;
40      else if (i == 'C') return 1;
41      else if (d == 'C') return 3;
42      else return 0;
43  }
```

```
46  Action ComportamientoRescatador::ComportamientoRescatadorNivel_0(Sensores sensores)
47  {
48      Action accion;
49      // El comportamiento de seguir un camino hasta encontrar un puesto base.
50      // Actualización de variables de estado
51      if (sensores.superficie[0] == 'D') tiene_zapatillas = true;
52
53
54      // Definición del comportamiento
55      if (sensores.superficie[0] == 'X'){     // Llegué al Objetivo
56          accion = IDLE;
57      }
58      else if (giro45Izq != 0){               // Estoy haciendo un TURN_SL
59          accion = TURN_SR;
60          giro45Izq--;
61      }
62      else {
63          int pos = VeoCasillaInteresanteR(sensores.superficie[1], sensores.superficie[2], sensores.superficie[3]);
64          switch (pos)
65          {
66          case 2:
67              accion = WALK;
68              break;
69          case 1:
70              giro45Izq = 1;
71              accion = TURN_L;
72              break;
73          case 3:
74              accion = TURN_SR;
75              break;
76          case 0:
77              accion = TURN_L;
78              break;
79          }
80      }
81
82      // Devolver la siguiente acción a hacer
83      last_action = accion;
84      return accion;
85  }
```

Figura 4: Segunda versión del comportamiento de seguir caminos para el `Rescatador`.

Devuelve 2, si la opción más interesante sería estar en la casilla de justo delante, 1 si sería la de 45 grados a la izquierda y 3 si fuera la de 45 grados a la derecha.

Modificamos ahora el comportamiento usando la función anterior, y quedando el método `ComportamientoRescatadorNivel_0` como muestra la figura 4.

Podemos observar que en la línea 63 se invoca a la función `VeoCasillaInteresanteR` y en la variable `pos` se almacenará el valor 0, si no hay nada interesante, 1 si conviene avanzar por la de 45 grados a la izquierda, 2 si seguir recto o 3 si es mejor el avance por la de 45 grados a la derecha. En lo que en la versión inicial era una estructura condicional `if-then-else` sobre las tres opciones, ahora se ha transformado en un `switch` pero con la misma intención.

Si compilamos la nueva versión y volvemos a ejecutar

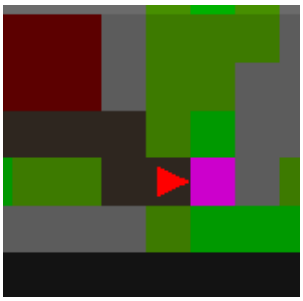
```
./practica2 ./mapas/mapa30.map 0 0 17 5 0 17 17 0 3 3 0
```

Veremos que ahora sí que el agente termina en la casilla de tipo `puesto base` que la vez anterior rechazó.

3. Considerando la altura

Probemos a ver que ocurre si ejecutamos la siguiente llamada:

```
./practica2 ./mapas/mapa30.map 0 0 24 7 2 17 17 0 3 3 0
```



Podemos ver que empieza a avanzar hacía la casilla de puesto base situada en la parte inferior del mapa, pero que justo cuando va a acceder y tras realizar una acción de avance, el simulador manda un mensaje de texto (qué se puede ver en el terminal) diciendo *"El rescatador ha chocado. Casilla destino demasiado alta"*.

Si en el simulador miramos lo que dice el sensor de cota del agente Rescatador vemos que nos dice que la casilla que tiene justo delante está a altura 3 mientras que la casilla en la que se encuentra está a altura 1, es decir, hay un desnivel de 2 entre ambas casillas que solo sería superable si tuviera las `zapatillas`.

Debemos corregir el código en dos sentidos, por un lado, tener en cuenta la altura a la hora de tomar la decisión sobre cuál es la siguiente casilla interesante a visitar y, por otro lado, dotar al agente de la posibilidad de acceder a una casilla tipo zapatillas para conseguir ese objeto.

Para lo segundo, vamos a modificar la función `VeoCasillaInteresanteR` que es la que establece por donde queda la siguiente casilla más interesante a visitar. En este caso, parecería razonable que siga siendo la casilla `puesto base` la más interesante, pero si no tengo las `zapatillas` aún, poner por delante visitar una casilla de tipo `zapatillas` antes que una casilla de tipo `camino`. Cambio la parametrización de dicha función para incluir un parámetro de tipo `bool` que indicará si ya tengo las `zapatillas`. En el interior, altero la estructura de estructuras condicionales para incluir la casilla de tipo zapatilla. El resultado final de todas esas modificaciones daría el siguiente código:

```
35  int VeoCasillaInteresanteR (char i, char c, char d, bool zap){
36      if (c == 'X') return 2;
37      else if (i == 'X') return 1;
38      else if (d == 'X') return 3;
39      else if (!zap) {
40          if (c == 'D') return 2;
41          else if (i == 'D') return 1;
42          else if (d == 'D') return 3;
43      }
44      if (c == 'C') return 2;
45      else if (i == 'C') return 1;
46      else if (d == 'C') return 3;
47      else return 0;
48  }
```

Si en `CompotarmientoRescatadorNivel_0` cambiamos la invocación a la función de `VeoCasillaInteresanteR` por:

```
int pos = VeoCasillaInteresanteR(sensores.superficie[1], sensores.superficie[2],
                                 sensores.superficie[3], tiene_zapatillas);
```

y volvemos a compilar, y ejecutamos

./practica2 ./mapas/mapa30.map 0 0 24 7 2 17 17 0 3 3 0

Vemos que ahora sí que el agente `Rescatador` llega a la casilla puesto base tras coger las zapatillas.

Sin embargo, es posible que nos encontremos en una situación donde no tengamos las zapatillas y haya una diferencia de altura que nos impida el paso, por tanto, es fundamental el considerar la diferencia de altura en los movimientos de los agentes. Probar que el agente se queda atascado colisionando constantemente si ejecutamos:

./practica2 ./mapas/mapa30.map 0 0 24 10 2 17 17 0 3 3 0

Para solucionarlo, modificamos el comportamiento del agente en el siguiente sentido: el agente se orientará para avanzar hacía la casilla más interesante que él pueda alcanzar haciendo una acción `WALK`, es decir, aquella que es adyacente a donde se encuentra y es accesible por altura al agente.

Para ello, se modificará el código para comprobar las diferencias de altura entre la casilla actual del agente y las posiciones 1, 2 y 3 de su vector cota antes de invocar a la función `VeoCasillaInteresanteR`, de manera que los argumentos que tengan una diferencia de altura que no permita el movimiento se pasarán como si fueran precipicios (se usará 'P' pero valdría cualquier tipo de terreno que no sea ni `'C'`, ni `'D'`, ni `'X'`). Para esta tarea, haré uso de la función auxiliar `ViablePorAlturaR`:

```
51  char ViablePorAlturaR (char casilla, int dif, bool zap){
52      if (abs(dif)<=1 or (zap and abs(dif)<=2))
53          return casilla;
54      else
55          return 'P';
56  }
```

Ahora la función `ComportamientoRescatadorNivel_0` quedaría como muestra la Figura 5.

```
59  Action ComportamientoRescatador::ComportamientoRescatadorNivel_0(Sensores sensores)
60  {
61      Action accion;
62      // El comportamiento de seguir un camino hasta encontrar un puesto base.
63      // Actualización de variables de estado
64      if (sensores.superficie[0] == 'D') tiene_zapatillas = true;
65
66
67      // Definición del comportamiento
68      if (sensores.superficie[0] == 'X'){     // Llegué al Objetivo
69          accion = IDLE;
70      }
71      else if (giro45Izq != 0){               // Estoy haciendo un TURN_SL
72          accion = TURN_SR;
73          giro45Izq--;
74      }
75      else {
76          char i = ViablePorAlturaR(sensores.superficie[1], sensores.cota[1]-sensores.cota[0], tiene_zapatillas);
77          char c = ViablePorAlturaR(sensores.superficie[2], sensores.cota[2]-sensores.cota[0], tiene_zapatillas);
78          char d = ViablePorAlturaR(sensores.superficie[3], sensores.cota[3]-sensores.cota[0], tiene_zapatillas);
79
80          int pos = VeoCasillaInteresanteR(i, c, d, tiene_zapatillas);
81          switch (pos)
82          {
83          case 2:
84              accion = WALK;
85              break;
86          case 1:
87              giro45Izq = 1;
88              accion = TURN_L;
89              break;
90          case 3:
91              accion = TURN_SR;
92              break;
93          case 0:
94              accion = TURN_L;
95              break;
96          }
97      }
98
99      // Devolver la siguiente acción a hacer
100     last_action = accion;
101     return accion;
102 }
```

Figura 5: Tercera versión del comportamiento de seguir caminos para el `Rescatador`.

En las líneas de la 76 a la 78 se invoca a la función `ViablePorAlturaR()` para asignar a las variables `i` (izquierda), `c` (centro), `d` (derecha) el carácter asociado a la casilla que se ve en el sensor de superficie siempre que dicha casilla sea accesible por altura. A partir de esas 3 variables locales, `i`, `c` y `d`, se determina la más interesante invocando ahora a `VeoCasillaInteresanteR`.

Si compilamos y volvemos a ejecutar

./practica2 ./mapas/mapa30.map 0 0 24 10 2 17 17 0 3 3 0

Observamos que ahora ya no se queda bloqueado en la colisión con la casilla, sino que se sigue moviendo buscando una nueva forma para llegar a una casilla `puesto base`.

4. Replicando el comportamiento para el agente Auxiliar

Esta sección se plantea como una tarea a realizar por el estudiante para replicar lo que se ha hecho en este tutorial con el agente Rescatador. Recordaremos los pasos a seguir:

1. Definir las variables de estado necesarias. Dicha definición se realizará en la parte privada de la clase `ComportamientoAuxiliar` en el fichero "**auxiliar.hpp**". Para este agente será necesario definir las mismas 3 variables de estado: `last_action`, `giro45Izq` y `tiene_zapatillas`, y además se inicializarán de la misma forma que se hizo con el otro agente.

2. Se han de definir las funciones:
   (a) `VeoCasillaInteresanteA`
   (b) `ViablePorAlturaA`

   IMPORTANTE:
   I. En las funciones (que no sean métodos de la clase) añadir al final del identificador de la función un `R` para las que sean para el agente `Rescatador` y una `A` para las del agente `Auxiliar`, para evitar que el compilador nos diga que estamos redefiniendo la función.
   II. Si el estudiante lo desea, estas dos funciones podrían ser implementadas, no como funciones auxiliares, sino como funciones miembro de la clase. En este segundo caso, ya no sería necesario añadir al final del identificador las letras R o A.
   III. Para este nivel, en el agente `Auxiliar` no es relevante tener zapatillas, ya que no le permite moverse mejor por las casillas `camino`. Este hecho implicaría no considerar el parámetro `bool zap` en las dos funciones anteriores.

3. Se tiene que repetir el esquema del comportamiento definido en `ComportamientoAuxiliarNivel_0`. Recordando que el agente auxiliar solo tiene las acciones `WALK` y `TURN_SR`, y que la implementación de la acción virtual "`TURN_SL`" se tiene que hacer

componiendo acciones `TURN_SR`. Por otro lado, también hay que tener en cuenta que el límite de diferencia de altura entre casillas que puede superar el `Auxiliar` es de 1 en valor absoluto en todos los casos.

Una vez completado y adaptado el comportamiento a las características especiales del agente `Auxiliar` probar que funciona correctamente con la siguiente llamada

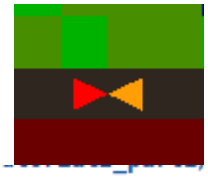
./practica2 ./mapas/mapa30.map 0 0 24 10 2 17 17 0 3 3 0

5. Resolviendo las interacciones entre los agentes

Terminada la tarea propuesta en la sección anterior, el estudiante tiene definido un comportamiento para cada uno de los agentes que aparentemente resuelve lo que se pide en el nivel. Pero veamos que ocurre en una ejecución como:

./practica2 ./mapas/mapa30.map 0 0 16 9 2 16 14 6 3 3 0

Se puede observar que cada agente bloquea a otro impidiendo que desarrollen su comportamiento de forma normal. ¿Cómo podría resolverse interacciones de este tipo?



```
El Rescatador ha chocado con un auxiliar
El auxiliar ha chocado con el rescatador
```

Es evidente que en la definición de los comportamientos de los agentes no se ha tenido en cuenta la posibilidad que los agentes se pudieran encontrar y resulta muy evidente que es necesario que esto se contemple.

En este punto quiero recordar que el orden usado por el simulador para aplicar las acciones de los agentes es: primero al agente `Rescatador` y después la acción del agente `Auxiliar`. Tener en cuenta este orden puede ayudar a resolver situaciones de conflicto, ya que el agente `Rescatador` siempre puede evitar la colisión, pero el `Auxiliar` no.

También se propone en este tutorial como tarea que el alumno sea capaz de evitar este tipo de interacciones entre los agentes en el nivel 0. Uno de los test de la autoevaluación irá expresamente dirigido a comprobar que el estudiante ha diseñado un comportamiento para resolver este problema.

6. Usando mapaResultado y mapaCotas

`mapaResultado` y `mapaCotas` son dos variables globales de tipo matriz que almacenan respectivamente la superficie del mapa y la altura de las casillas. En los niveles donde es conocido el mapa (niveles 2 y 3), es donde el entorno coloca la información para que la conozcan los agentes. En los niveles 0, 1 y 4, estas variables se deben utilizar para ir completando la información del mapa y, por tanto, la misión sería "escribir" sobre ellos la información que se va recibiendo por el sistema sensorial de cada agente.

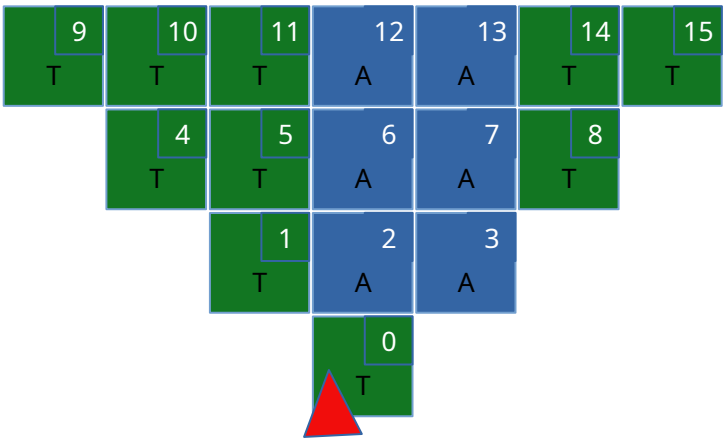
Una de las primeras cosas que hay que decir es que aunque aparentemente `mapaResultado` es una variable que comparten el agente `Rescatador` y el agente `Auxiliar`, **en realidad no es así**. Cada agente es independiente y su información es única. Eso quiere decir que aunque la variable se llame igual y se use igual en los dos agentes, son variables distintas. Hay un `mapaResultado` para `Rescatador` y otro para `Auxiliar`. De igual forma pasa con `mapaCotas`.

En los niveles 0, 1 y 4, el simulador toma lo que cada agente escribe en la variable `mapaResultado` y lo "pinta" en el simulador. Haciendo esto, se puede ver por donde se han movido los agentes y que partes del mapa ya han visitado. Lo que se pinta es la unión de ambas matrices. Si sobre una casilla hay divergencia, es decir, no han escrito la misma información los dos agentes, la que prevalece es lo que escribió el agente `Rescatador`.

¿Cómo escribir en esas variables? Es bastante simple; con los sensores `posF` y `posC` conozco cuál es la posición en la que se encuentra el agente. Con el sensor `rumbo` conozco cuál es su orientación real sobre el mapa. Pues con esos tres valores puedo proyectar sobre el `mapaResultado` lo que veo a través de mis sensores de `superficie` y `cota`.

Pongamos como ejemplo el caso en el que el agente `Rescatador` se encuentre con `rumbo norte`, siendo su posición `posF` y `posC`.

El cono de visión quedaría, por estar en orientación norte, como muestra la siguiente imagen:



En este caso, sabemos que:

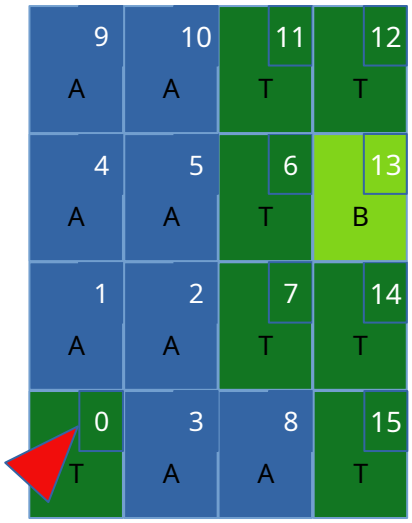
```
mapaResultado[sensores.posF][sensores.posC] = sensores.superficie[0]
```

es decir, la posición 0 de `superfice` indica el tipo de casilla en la que actualmente está.

Por tener orientación norte también sabemos que:

```
mapaResultado[sensores.posF-1][sensores.posC-1] = sensores.superficie[1];
mapaResultado[sensores.posF-1][sensores.posC] = sensores.superficie[2]
mapaResultado[sensores.posF-1][sensores.posC+1] = sensores.superficie[3]
```

Recordar que para orientaciones `noreste`, `sureste`, `noroeste` y `suroeste`, el cono de visión se representa:



Siguiendo este esquema podremos situar los 16 valores. De igual forma se puede proceder con `mapaCotas` y el sensor `cota`.

En la Figura 6 se muestra una versión incompleta de la función `SituarSensorEnMapaR`, donde se sitúan los 4 primeros valores del sensor superficie para las orientaciones `norte`, `noreste` y `este` en el parámetro `<vector<vector<unsigned char> > m` (en la llamada a esta función aquí se situaría `mapaResultado`).

```
58  void SituarSensorEnMapaR(vector<vector<unsigned char>> &m, vector<vector<unsigned char>> &a, Sensores sensores)
59  {
60      // cout << "estoy en situarsensor en matriz de mapa\n";
61      m[sensores.posF][sensores.posC] = sensores.superficie[0];
62
63      int pos = 1;
64      switch (sensores.rumbo)
65      {
66      case norte:
67          m[sensores.posF-1][sensores.posC-1] = sensores.superficie[1];
68          m[sensores.posF-1][sensores.posC] = sensores.superficie[2];
69          m[sensores.posF-1][sensores.posC+1] = sensores.superficie[3];
70
71          break;
72      case noreste:
73          m[sensores.posF - 1][sensores.posC] = sensores.superficie[1];
74          m[sensores.posF - 1][sensores.posC + 1] = sensores.superficie[2];
75          m[sensores.posF][sensores.posC + 1] = sensores.superficie[3];
76
77          break;
78      case este:
79          m[sensores.posF-1][sensores.posC+1] = sensores.superficie[1];
80          m[sensores.posF][sensores.posC+1] = sensores.superficie[2];
81          m[sensores.posF+1][sensores.posC+1] = sensores.superficie[3];
82          break;
83
84      case sureste:
85          break;
86
87      case sur:
88
89          break;
90      case suroeste:
91
92          break;
93
94      case oeste:
95
96          break;
97      case noroeste:
98
99          break;
100     }
```

Figura 6: Versión para completar por el estudiante de la función `SituarSensorEnMapaR`

El estudiante debe terminar esta función incluyendo en `mapaResultado` (que estará asociado al parámetro `m`) y en `mapaCota` (que estará asociado con el parámetro `a`) los valores que el agente `Rescatador` va observando por sus sensores de visión `superficie` y `cota` respectivamente.

Esta función se invoca desde la parte de actualización de la información del mundo en el módulo `ComportamientoRescatadorNivel_0`. En la Figura 7 se muestra como sería la invocación al principio de dicho módulo.

```
104  Action ComportamientoRescatador::ComportamientoRescatadorNivel_0(Sensores sensores
105  {
106      Action accion;
107      // El comportamiento de seguir un camino hasta encontrar un puesto base.
108      // Actualización de variables de estado
109      SituarSensorEnMapaR(mapaResultado,mapaCotas,sensores);
110      if (sensores.superficie[0] == 'D') tiene_zapatillas = true;
111
112
113      // Definición del comportamiento
114      if (sensores.superficie[0] == 'X'){     // Lllegué al Objetivo
115          accion = IDLE;
116      }
```

Figura 7: Inclusión en la línea 109 de la llamada a la función `SituarSensorEnMapaR`

Esta función no es relevante para el nivel 0, pero sí que lo será para el nivel 1, ya que el contenido de `mapaResultado` será lo que se comparará con el mapa real para comprobar cuantas casillas de tipo `sendero` y `camino` han sido descubiertas.

7. 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 solo 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.

Los dos primeros niveles de la práctica trata sobre agentes reactivos, por consiguiente, los comportamientos deliberativos no están permitidos.

Muchos elementos que forman parte de la práctica no se han tratado en este tutorial. Esos elementos son relevantes para mejorar la capacidad de exploración 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.