Inteligencia Artificial

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

Tutorial práctica 1

Agentes reactivos (los extraños mundos de BelKan)

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

Curso 2023-2024

Tutorial 1: Práctica 1

Inteligencia Artificial

Contenido:

1. Introducción

El objetivo de la práctica es definir un comportamiento puramente reactivo para un agente que tiene 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 vivo y posteriormente ir mejorando su capacidad para descubrir zonas aún no exploradas.
2. Una vez se consigue la movilidad, trata de determinar las coordenadas exactas por las que me estoy moviendo en el mapa y utilizar la matriz **mapaResultado** para ir almacenando lo que voy viendo.
3. Gestionar los reinicios, es decir, determinar qué debe recordar y qué debe olvidar el agente entre vidas.
4. Definir qué comportamiento debe tener el agente frente a las otras entidades móviles del mundo.
5. 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 muy inicial del subobjetivo 1 antes mencionado, es decir, dar los primeros pasos (más o menos firmes) 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**) se pueden definir 2 fases:

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

La segunda fase es realmente el comportamiento, pero la primera es 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 sólo tienen vigencia en una iteración. Si queremos que nuestro agente tome una decisión en base a 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. Por consiguiente, 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:

1. En el fichero **jugador.hpp** debajo de la parte que pone **private** se declaran tantas variables como sean necesarias en la forma $< tipo > \ < identificador >$

   ```
   Action last_action;
   Orientacion brujula;
   ```

2. En ese mismo archivo, en el costructor de la clase se produce la inicialización:

```
// se da valor a las variables
last_action = actIDLE;
brujula = norte;
```

3. En el método **think**, en la primera parte que como hemos dicho antes se dedica a la actualización del mundo, se modifica su valor en función de la última acción realizada.

```
switch(last_action){
    case actTURN_SR:
        brujula = static_cast<Orientacion>((brujula+1)%8);
        break;
.....................................
}
```

4. La segunda parte del método **think** se usa para determinar la siguiente acción a realizar.

```
if (brujula == norte){
    accion = actTURN_L;
}
.....................................
last_action = accion;
```

Con esta primera información básica, ya podemos empezar a construir los primeros comportamientos básicos de nuestro agente reactivo.

2. Mis 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, aunque siempre debemos tener en mente el objetivo final de la práctica, reconocer el máximo posible del mapa.

Por esta razón, vamos a empezar definiendo 2 variables de estado que serán útiles durante la construcción de la práctica. Estas variables serán **current_state** y **last_action**. La primera nos permite recordar dónde estoy y hacía dónde voy a dar mi siguiente paso, mientras que la última es fundamental para realizar la actualización de la información y es la forma más simple de recordar cuál fue la última acción que realicé.

¿Por qué necesito una variable de estado como **current_state** si el agente tiene sensores (posF, posC y sentido) qué me permiten conocer su posición? La razón es que hay niveles en el juego donde esos sensores no funcionan correctamente y por tanto, será el propio agente el que tendrá que ir actualizando la información de su posición en base a las acciones de movimiento que realice durante la simulación.

```
#ifndef COMPORTAMIENTOJUGADOR_H
#define COMPORTAMIENTOJUGADOR_H

#include "comportamientos/comportamiento.hpp"
using namespace std;

struct state{
   int fil;
   int col;
   Orientacion brujula;
};

class ComportamientoJugador : public Comportamiento{
  public:
    ComportamientoJugador(unsigned int size) : Comportamiento(size){
      // Constructor de la clase
      current_state.fil = current_state.col = 99;
      current_state.brujula = norte;
      last_action = actIDLE;
    }

    ComportamientoJugador(const ComportamientoJugador & comport) :
    ↪  Comportamiento(comport){}
    ~ComportamientoJugador(){}

    Action think(Sensores sensores);
    int interact(Action accion, int valor);

  private:

  // Declarar aquí las variables de estado
  state current_state;
  Action last_action;
};

#endif
```

En el fichero **jugador.hpp**, haremos las siguientes modificaciones: primero incluiremos el tipo de dato **state** (de tipo registro con los campos **fil**, **col** y **brujula**) para poder definir variables que contengan la posición en fila y columna del agente junto con su orientación. Después, definimos las variables de estado que mencionábamos anteriormente debajo de **private**, la primera de tipo **state** (el nuevo tipo que hemos declarado) y la última de tipo **Action**.

Las inicializamos en el constructor de la clase, dónde por defecto tanto **current_state.fil** como **col** tendrán un valor por defecto (en este caso 99), **current_state.brujula** a **norte** y **last_action** a **actIDLE** ya que no se hizo nada en el instante anterior a empezar la simulación.

En este tutorial estoy asumiendo que no puedo acceder a la información que me proporcionan los sensores de posición y de orientación (nivel 1 o superior), por esa razón le doy dos valores por defecto. El valor en sí mismo ahora es irrelevante, pero sí que tiene que ser conocido para realizar correctamente la orientación del agente y que en un futuro (y si así lo deseamos) podamos aprovechar todo lo visto hasta el momento de poder almacenarlo en el **mapaResultado**.

Ahora en el fichero **jugador.cpp**, vamos a crear la primera parte de actualización de las variables de estado en el método **think**. Es fundamental en la práctica que **current_state** refleje fielmente la posición del agente. Un error en la actualización debida a no haber contemplado las colisiones por ejemplo, puede llevar a que la información que tengamos sobre el mundo no esté situada de forma correcta.

Aquí propongo una actualización de estas variables en el modelo más simple. En la versión final de la práctica deberán contemplarse todas las situaciones que podrían alterar este modelo básico de actualización de la posición del agente. En este caso, el modelo básico de actualización es bien simple: los valores de la fila y la columna sólo cambian si la última acción del agente (es decir, la anterior) fue avanzar (**actWALK**). La orientación cambia con las 2 acciones de giro. Así podemos plantear una estructura de decisión **switch-case** que contemple estas acciones del siguiente modo:

```
switch (last_action){
    case actWALK:
        // Actualización en caso de avanzar
        break;
    case actRUN:
        // Actualización en caso de correr
        break;
    case actTURN_SR:
        // Actualización en caso de girar 45º a la derecha
        break;
    case actTURN_L:
        // Actualización en caso de girar 90º a la izquierda
        break;
}
```

En el caso de los giros está bien claro, si giró a la derecha 45 grados, y la orientación del agente es a, la orientación final del agente será el siguiente valor contemplado en el sentido horario. Si codificamos los valores de la orientación empezando en el **norte** y siguiendo el sentido horario del 0 al 7 (como ocurre internamente en el tipo de dato enumerado de C++), podemos expresar el resultado de la acción anterior como **módulo (a+1)**. De igual forma, en el giro de 45 grados a la izquierda, los desplazamientos serían en el sentido antihorario, es decir, el resultado de esta acción está vinculado con la operación **módulo(a-1)**. Por consiguiente, el código podría quedar de la siguiente forma:

```
switch (last_action){
    case actWALK:
        // Actualización en caso de avanzar
        break;
    case actRUN:
        // Actualización en caso de correr
        break;
    case actTURN_SR:
       a = current_state.brujula;
        a = (a+1)%8;
```

```
        current_state.brujula = static_cast<Orientacion>(a);
        break;
    case actTURN_L:
        // Actualización de girar 90º a la izquierda
        break;
}
```

En el caso del giro a la izquierda el $(a+7)\,\%8$ es equivalente a $(a-1+8)\,\%8$ y equivalente a $(a-1)\,\%8$. La razón de usar la primera, es que el operador $\%$ en C++ no trabaja con los números negativos, por eso, hay que asegurarse que se aplica sobre números positivos.

Tarea 1

Se han dejado sin completar los casos para **actTURN_L**. Complétalo siguiendo lo que se indica en el párrafo anterior.

La acciones de avance dependen de la orientación. En la convención que vamos a usar, supondremos que el **norte** está en los valores bajos de las filas y el **sur** en los valores altos. De igual modo, vamos a asociar el **este** a valores altos de las columnas y el **oeste** a los bajos. Por consiguiente, y siguiendo este convenio, avanzar estando orientado al **norte** indica decrementar en una unidad el valor de la fila y mantener constante el valor de la columna, mientras que avanzar estando hacia el **sur** es igualmente no cambiar la columna, pero en este caso incrementar el valor de la fila. Así, el proceso de actualización quedaría como sigue:

```
switch (current_state.brujula){
   case norte:   current_state.fil--; break;
   case noreste: current_state.fil--; current_state.col++;  break;
   case este:    current_state.col++; break;
   case sureste: /*Actualizacion*/ break;
   case sur:     /*Actualizacion*/ break;
   case suroeste:/*Actualizacion*/ break;
   case oeste:   /*Actualizacion*/ break;
   case noroeste:/*Actualizacion*/ break;
  }
```

Tarea 2

Se han dejado sin completar los casos desde sureste a noroeste. Complétalos poniendo cómo cambian las coordenadas de fila y columna del agente tras hacer un movimiento de una casilla en el sentido indicado en el *case*. Recuerda que la posición de partida del agente viene determinada por los campos **fil** y **col** de la variable de estado **current_state**.

Poniéndolo todo junto, tendríamos el código que se muestra a continuación:

```
#include "../Comportamientos_Jugador/jugador.hpp"
#include <iostream>
using namespace std;

Action ComportamientoJugador::think(Sensores sensores) {

    Action accion = actIDLE;
    int a;

    // Actualizacion de las variables de estado
    switch(last_action) {
       case actWALK:
          switch (current_state.brujula) {
             case norte: current_state.fil--; break;
             case noreste: current_state.fil--;
                           current_state.col++;
                           break;
             case este: current_state.col++; break;
             case sureste:  /* Actualizacion */ break;
             case sur:      /* Actualizacion */ break;
             case suroeste: /* Actualizacion */ break;
             case oeste:    /* Actualizacion */ break;
             case noroeste: /* Actualizacion */ break;
           }
           break;

       case actRUN:
          // Actualizacion en caso de correr
          break;

       case actTURN_SR:
          a = current_state.brujula;
          a = (a + 1 ) % 8;
          current_state.brujula = static_cast<Orientacion>(a);
          break;

       case actTURN_L:
          // Actualizacion tras girar 90º a la izquierda
          break;
    }
    // devuelve el valor de la accion
    return accion;
}
```

Con esto (a falta de sustituir los comentarios de actualización por las soluciones a las tareas 1 y 2), en principio, conseguimos mantener bien la ubicación del agente. Ahora haremos que se mueva.

En un principio vamos a considerar que sólo podremos avanzar si vamos a una casilla que es de terreno arenoso (**T**) o pedregoso (**S**) y no hay ningún otro agente ocupando esa casilla. En caso de que no podamos avanzar vamos a girar 45 grados siempre a la izquierda. Para construir esta primera regla, sólo necesitamos hacer uso de los sensores, y en concreto, sólo de los sensores de visión (**sensores.terreno** y **sensores.agentes**). La regla es la siguiente:

```
if ((sensores.terreno[2]=='T' or  sensores.terreno[2]=='S') and
        sensores.agentes[2]=='_'){
      accion = actWALK;
  } else{
      accion = actTURN_L;
  }
```

Ya tendríamos casi listo este primer comportamiento. Sólo nos faltaría incluir una cosa. Antes de **return accion**; debemos asignar a **last_action** la acción que se va a ejecutar para poder hacer correctamente la actualización de la ubicación en la siguiente iteración. El método **think** quedaría como se indica seguidamente:

```
#include "../Comportamientos_Jugador/jugador.hpp"
#include <iostream>
using namespace std;

Action ComportamientoJugador::think(Sensores sensores) {

    Action accion = actIDLE;
        int a;

    // Actualizacion de las variables de estado
    switch(last_action) {
       case actWALK:
          switch (current_state.brujula) {
             case norte: current_state.fil--; break;
             case noroeste: current_state.fil--;
                            current_state.col++;
                            break;
             case este: current_state.col++; break;
             case sureste:  /* Actualizacion */ break;
             case sur:      /* Actualizacion */ break;
             case suroeste: /* Actualizacion */ break;
             case oeste:    /* Actualizacion */ break;
             case noroeste: /* Actualizacion */ break;
          }
          break;
.....................................................................
          case actTURN_SR:
             a = current_state.brujula;
             a = (a + 1 ) % 8;
             current_state.brujula = static_cast<Orientacion>(a);
             break;

          case actTURN_L:
             // Actualizacion tras girar 90 a la izquierda
             break;
    }

    if (sensores.terreno[2] == 'T' or sensores.terreno[2] == 'S') and
        sensores.agentes[2] == '_') {
        accion = actWALK;
    }
    else{
       accion = actTURN_L;
    }

```

```
46      // devuelve el valor de la accion
47      last_action = accion;
48      return accion;
49  }
```

Ahora, compila el software (recuerda incluir en el lugar indicado del código la resolución de las tareas 1 y 2). En un terminal escribe la siguiente sentencia:

```
./practica1 ./mapas/mapa30.map 1 0 4 5 1
```

La sentencia manda la ejecución de una simulación que se desarrolla en el **mapa30.map**, usando 1 como semilla del generador de números aleatorios, toma el nivel 0 (los sensores de posición y orientación funcionan) siendo la posición inicial del agente la casilla de fila 4 y columna 5 y está con orientación noreste.

Una vez iniciada la simulación, activa la opción **Mostrar Mapa** para ver la posición real del agente sobre el mundo y ve pulsando el botón **Paso**. Verás que cuando lo que tiene delante en el sentido de su marcha no es ni gris ni marrón, gira 90 grados a la izquierda. Si después de varias pulsaciones a **Paso** dais al botón **Ejecución**, veréis que llega un punto en el que la ruta que sigue el agente entra en un ciclo.

3. Intentando moverme mejor

En el comportamiento anterior, siempre hago los giros hacia la izquierda. A priori, uno podría pensar que esa es la razón por la que su comportamiento entra en un ciclo. Parece también lógico que, si en lugar de girar a la izquierda, girara a la derecha, el resultado sería el mismo, ¿pero qué ocurre si el giro depende de una variable aleatoria? Bueno, pues vamos a testear el resultado. Para eso, hacemos lo siguiente:

1. En el fichero **jugador.hpp** creamos una variable lógica llamada **girar_derecha**, que definimos debajo de **private** junto a las otras variables de estado e inicializamos a **false** en el constructor de la clase.

2. En el fichero **jugador.cpp**, en la parte de actualización, en los **case** relativos a los giros, cada vez que se haya aplicado un giro en la acción anterior, volvemos a darle un nuevo valor aleatorio a la variable **giro_derecha**.

3. En el fichero **jugador.cpp**, en la parte del comportamiento, modificamos el comportamiento anterior por este:

```
if ((sensores.terreno[2]=='T' or sensores.terreno[2]=='S')
      and sensores.agentes[2]=='_'){
         accion = actWALK;
 } else if (!girar_derecha){
        accion = actTURN_L;
        girar_derecha = (rand()%2 ==0);
 } else{
        accion = actTURN_SR;
        girar_derecha = (rand()%2 ==0);
 }
```

De esta forma, el método **think** quedaría así:

```
1  ..........................................................................
2      // Decision de la nueva accion
3      if (sensores.terreno[2] == 'T' or sensores.terreno[2] == 'S')
4           andsensores.agentes[2] == '_') {
5          accion = actWALK;
6      }
7      else if (!girar_derecha) {
8         accion = actTURN_L;
9         girar_derecha = (rand() % 2 == 0);
10     }
11     else {
12        accion = actTURN_SR;
13        girar_derecha = (rand() % 2 == 0);
14     }
15
16     // devuelve el valor de la accion
17     last_action = accion;
18     return accion;
19 }
```

Ahora, volvemos a compilar y repetimos el comando anterior en ejecución:

```
./practica1 ./mapas/mapa30.map 1 0 4 5 1
```

Tiene un comportamiento mejor que el anterior ya que parece que no se queda atrapado en un ciclo, pero en realidad siempre está recorriendo las mismas casillas. Una idea de mejora para intentar no pasar por las mismas casillas, es anotar de alguna manera sobre cada casilla cuando fue la última vez que se pasó por ella. Con esa información sobre las casillas, podría establecer un proceso de decisión en el que en cada momento el agente decida ir por la casilla que hace más tiempo que no ha visitado. Buscar un proceso que haga que el agente se mueva a casillas no visitadas es un objetivo que el estudiante debe perseguir para mejorar el movimiento del agente.

4. Escribiendo en mapaResultado

A partir de ahora, vamos a asumir que el movimiento básico lo hacemos con el modelo que acabamos de implementar en la sección anterior. Ahora que nos movemos, y empezamos a descubrir los distintos terrenos del mapa, deberíamos poder empezar a escribir el resultado en la matriz **mapaResultado**. Recordemos que esta matriz es la que almacenará la información sobre lo que conocemos sobre el mapa.

En esta matriz se van añadiendo los distintos elementos que forman parte del terreno del mapa cuando el agente está seguro de su posición exacta. Para eso, necesitamos saber dónde nos encontramos con exactitud, ya que en algunos de los niveles del juego no se conoce al principio la posición.

La forma de obtener dichas coordenadas es mediante las casillas **G** (representadas de color celeste en el terreno). Cuando estamos encima de ellas los campos **posF** y **posC** de la variable de tipo registro sensores nos devuelve la fila y la columna de donde nos

encontramos respectivamente y en sentido la orientación actual del agente. Para eso, sobre el comportamiento anterior incluiremos que las casillas **G** transitables para nuestro agente y lo haremos de la siguiente forma:

```
if ((sensores.terreno[2] == 'T' or sensores.terreno[2] == 'S'  or
   sensores.terreno[2] == 'G') and (sensores.agentes[2] == '_'){
   accion = actWALK;
}
```

Incluido el cambio anterior, el agente puede situarse sobre una casilla celeste de **Posicionamiento**. Ahora sólo hace falta incluir una regla en la parte de actualización del conocimiento que capture los valores de fila, columna y orientación en estas casillas. La regla es bien simple:

```
if (sensores.terreno[0]=='G' and !bien_situado){
   current_state.fil = sensores.posF;
   current_state.col= sensores.posC;
   current_state.brujula = sensores.sentido;
   bien_situado = true;
}
```

Se ha incluido una nueva variable de estado **bien_situado**. Esta variable nos dice si el valor de la variable **current_state** refleja la posición correcta en el mapa del agente. Como siempre, para incluir una variable de estado el proceso es igual: ir a **jugador.hpp** y debajo de **private** declarar la variable y en el constructor inicializarla (en este caso a **false**).

Ahora el agente ya puede saber cuándo está bien situado. Estando en esta situación, toda la información que aparece en cada iteración en **sensores.terreno** se debería pasar a **mapaResultado**. En este tutorial, sólo pondremos en dicha matriz el valor de la casilla en la que se encuentra en este momento. También en la parte de actualización de información, pondríamos la siguiente regla:

```
if (bien_situado){
      mapaResultado[current_state.fil][current_state.col] =
       ↪  sensores.terreno[0];
}
```

ya que la coordenada 0 del sensor de terreno corresponde con el tipo de casilla en la que actualmente se encuentra el agente. Colocado todo junto, la parte final del método **think** quedaría como muestra la figura siguiente:

```
1  .....
2      if (sensores.terreno[0] == 'G' and !bien_situado) {
3         current_state.fil = sensores.posF;
4         current_state.col = sensores.posC;
5         current_state.brujula = sensores.sentido;
6         bien_situado = true;
7      }
8
9      if (bien_situado) {
```

```
10          mapaResultado[current_state.fil][current_state.col] =
            ↪  sensores.terreno[0];
11      }
12
13      // Decision de la nueva accion
14      if (sensores.terreno[2] == 'T' or sensores.terreno[2] == 'S') or
15          sensores.terreno[2] == 'G' and sensores.agentes[2] == '_') {
16          accion = actWALK;
17      }
18      else if (!girar_derecha) {
19          accion = actTURN_L;
20          girar_derecha = (rand() % 2 == 0);
21      }
22      else {
23          accion = actTURN_SR;
24          girar_derecha = (rand() % 2 == 0);
25      }
26
27      // devuelve el valor de la accion
28      last_action = accion;
29      return accion;
30  }
```

La línea

```
mapaResultado[current_state.fil][current_state.col] =
↪  sensores.terreno[0];
```

podría sustituirse por la invocación a una función que se encargará de colocar toda la información contenida en el sensor terreno en **mapaResultado**. De momento, solo colocaremos la información contenida en la componente cero, pero más adelante se puede exterder para colocar la información sea cual sea la orientación del agente siempre que esté bien situado. La llamada sería de esta forma:

```
PonerTerrenoEnMatriz(sensores.terreno, current_state,
↪  mapaResultado);
```

Siendo la definición de esta función tal y como se indica a continuación:

```
1  void PonerTerrenoEnMatriz(cons vector<unsigned char> & terreno,
2          const state &st, vector<vector<unsigned char> &matriz) {
3     // extiende esta version inicial donde solo se pone la componente 0
4     // en la matriz; la nueva version debe poner todas las componentes
5     // de terreno en funcion de la orientacion del agente
6     matriz[st.fil][st.col] = terreno[0];
7  }
```

Volvemos ahora a compilar y ejecutamos el siguiente comando:

```
./practica1 ./mapas/mapa30.map 1 0 6 6 1
```

Tras dejar aglgún tiempo de ejecución terminamos con un resultado parecido al mostrado en la Fig. 1.

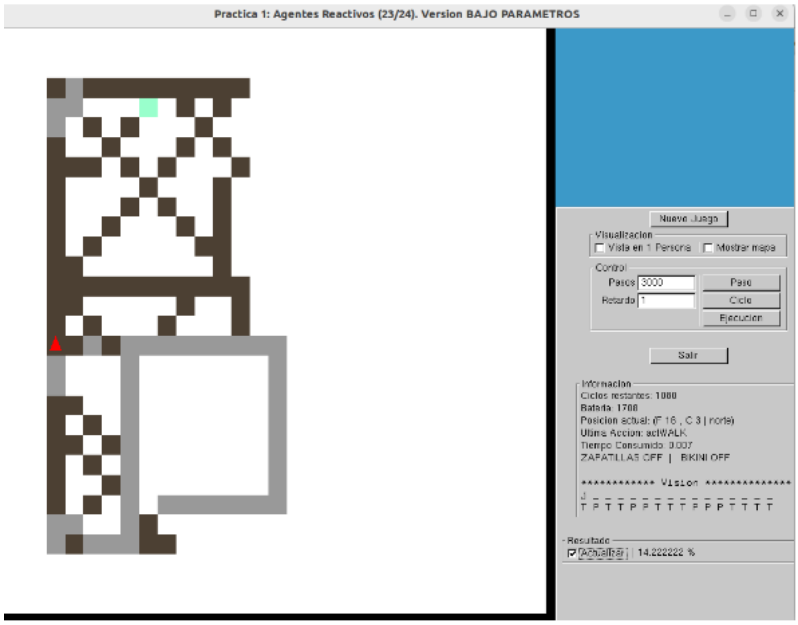

Figura 1: Comportamiento del agente

Esto pone de manifiesto que se debe mejorar el movimiento del agente si se desea que realmente explore bien todo el mapa.

5. Métrica para medir la calidad de la solución

Llegados a este momento, podemos considerar que ya hemos definido una primera aproximación a un comportamiento muy simple para que el agente reconozca algo del mapa, aunque conforme avancéis en la práctica os daréis cuenta que es demasiado básico y no considera muchos elementos que pueden ocurrir en el mundo y que son fundamentales para hacer un buen reconocimiento del terreno. No sólo eso, al no considerar las situaciones de choque o reinicio, es posible que nos podamos encontrar con errores de **Violación de segmento ('core' generado)** por intentar escribir en posiciones invalidas de **mapaResultado**.

Ahora queremos considerar cómo de bueno es este comportamiento. Usando el software con entorno gráfico, el proceso de ejecutar las 3,000 iteraciones consume mucho tiempo. Por esa razón, se incluye en el software una versión sin entorno gráfico que permite realizar la simulación mucho más rápido. En nuestro caso, una vez compilado, ejecutaremos el siguiente comando:

```
./practica1SG ./mapas/mapa30.map 1 0 6 6 1
```

En el ejemplo anterior, el resultado obtenido sería un reconocimiento del 6.11 % del mapa original. Extenderé para estudiar cómo funciona este mismo comportamiento sobre el mismo mapa y condiciones iniciales, pero sobre el resto de niveles de dificultad, así que procedería a hacer la siguiente secuencia de simulaciones:

simulación	porcentaje
./practica1SG ./mapas/mapa30.map 1 1 6 6 1	15.88 %
./practica1SG ./mapas/mapa30.map 1 2 6 6 1	**core**
./practica1SG ./mapas/mapa30.map 1 3 6 6 1	**core**

Ahora repetiré el proceso anterior de revisar el comportamiento en todos los niveles pero para nuevos mapas. En este caso elegiré **mapa50** y **mapa75**. Para el caso del mapa de 50 consideraré la siguiente configuración inicial:

```
./practica1SG ./mapas/mapa50.map 1 x 3 3 0
```

donde **x** tomará los valores del 0 al 3 para considerar todos los niveles. De igual forma para el mapa de 75 consideraré la configuración:

```
./practica1SG ./mapas/mapa75.map 1 x 60 11 0
```

Los resultados obtenidos por mapas y niveles se muestran en la siguiente tabla:

nivel	mapa30	mapa50	mapa75
0	15.88	22.04	11.37
1	15.88	22.04	11.37
2	**core**	17.52	14.15
3	**core**	17.52	14.15

La media global es de un $13{,}49\,\%$ de reconocimiento del mapa. Se puede observar que para el nivel 3 de todos los mapas, la simulación termina dando un **core**. La razón es que después de ser atacado por un lobo, el agente es reiniciado. En el comportamiento que se está usando, no se tiene en cuenta que se pasa a no estar bien situado (**bien_situado** debería ser falso). Este hecho hace que se siga escribiendo en **mapaResultado** sobre posiciones que pueden ser incorrectas. Es muy importante que se revisen estas situaciones antes de entregar el software.

Siguiendo con la idea que aquí se describe, un nuevo comportamiento tendrá un mejor funcionamiento si consigue mejorar en media sobre los mapas que se consideren. En cualquier caso, el estudiante también puede usar la fórmula que se indica en el guion para dar valoración numérica a la práctica.

Tarea 3

Evalúa si afecta de forma positiva el sustituir la condición

```
if (sensores.terreno[0] == 'G' and !bien_situado)
```

por la condición

```
if (sensores.posF != -1 and !bien_situado)
```

para actualizar los valores de la variable de estado **current_state**.

Es importante indicar que lo que aquí se describe es sólo un ejemplo. El estudiante deberá seleccionar los mapas y las situaciones de inicio que considere oportunos para

evaluar el comportamiento que esté diseñando, ya que no se conocen los mapas sobre los que se realizará la evaluación final de la práctica, aunque sí se sabe que serán mapas muy parecidos a los que se han proporcionado.

Hay que entender que el comportamiento que se ha descrito aquí es muy simple, por consiguiente, como referencia para los estudiantes, la soluciones que propongan deberá incrementar substancialmente estos resultados.

6. 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.

El comportamiento que se define en este tutorial no se considera como un comportamiento **entregable** como solución a la práctica. Así que aquellos que tengan la intención de entregar esto como su práctica o un comportamiento con ligeras variaciones de este, les recomendaría que no lo hagan, porque será considerada como una práctica copiada.

La práctica es sobre agentes reactivos, por consiguiente, los comportamientos deliberativos no están permitidos excepto, que dicho comportamiento se restrinja exclusivamente a la información sensorial del agente.

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.