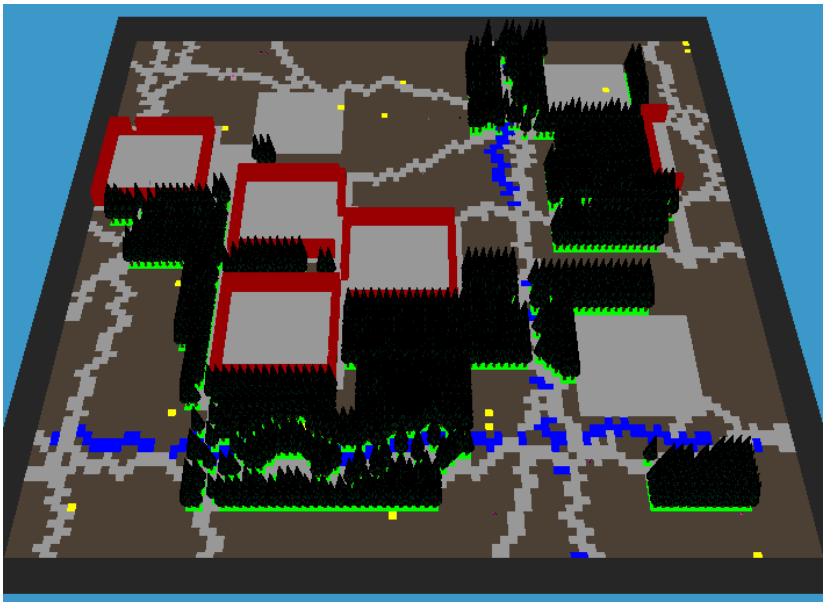
INTELIGENCIA ARTIFICIAL

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 2016-2017

1. Introducción

El objetivo de la práctica es definir un comportamiento puramente reactivo para un agente cuya misión es construir un mapa del entorno que le rodea. Os 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 subojetivos:

1. Establecer una movilidad básica para el agente que le permita navegar por el mundo manteniéndose vivo.
2. Una vez que puedo moverme con cierta facilidad, encontrar 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 que debe recordar y que debe olvidar el agente entre vidas.
4. Gestión de objetos. Esto implica no sólo cogerlos, sino también establecer una determinada política de gestión de la mochila, así como lo que estos objetos puedan influir sobre los tres puntos anteriores.
5. Comportamiento frente a las otras entidades móviles del mundo.
6. Refinamiento del modelo general, para mejorar la capacidad de exploración del agente.

En este tutorial intenta ser una ayuda para poner en marcha la práctica y se centra en el primer subobjetivo antes mencionado, es decir, a 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") hay 2 fases: (a) en una primera fase, se observa cómo ha cambiado el mundo en función de la última acción que se realizó y se actualiza la información que tenemos del mundo y (b) una segunda fase, se determina cual es la siguiente acción a realizar. La segunda fase es el comportamiento, pero la primera es muy importante, ya que intenta asegurar que la decisión que se tome en la segunda fase sea 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 un 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 rodean 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 la forma en la que se usan las variables de estado (forman 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 declarar una variable de estado es la siguiente:

(1) En el fichero "***jugador.hpp***" debajo de la parte que pone "***private:***" se declaran las variables (tantas como se estimen necesarias)

(2) En ese mismo fichero, en el constructor de la clase se inicializa.

(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.

(4) En la segunda parte del método "think" se usa en alguna regla.

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 4 variables de estado que serán útiles durante la construcción de la práctica. Estas variables serán "fil", "col","brujula" y "ultimaAccion". Las tres primeras son las básicas para recordar donde estoy y hacía donde 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 cual fue la última acción que realizó. En el fichero "jugador.hpp", definimos estas variables debajo de "private:", las tres primeras de tipo "int" y la última de tipo "Action". Las inicializamos en el constructor de la clase, dónde por defecto tanto "fil" como "col" a un valor de defecto (en este caso 99), "brujula" a 0, tomando el siguiente criterio de codificación (0 indica orientación Norte, 1 orientación Este, 2 orientación Sur y 3 orientación Oeste) y "ultimaAccion" con no hice nada en la iteración anterior.

Cuando aparecemos por primera vez en el mapa, no conocemos las coordenadas del agente, por esa razón le doy dos valores por defecto. El valor en si mismo es ahora mismo irrelevante, pero sí que tiene que ser conocido para la fase orientación y podamos aprovechar todo lo visto hasta el momento de poder ponerlo en el "mapaResultado".

jugador.hpp jugador.cpp

```
#ifndef COMPORTAMIENTOJUGADOR_H
#define COMPORTAMIENTOJUGADOR_H

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

class ComportamientoJugador : public Comportamiento{

  public:
    ComportamientoJugador(unsigned int size) : Comportamiento(size){
      // Inicializar Variables de Estado
      fil = col = 99;
      brujula = 0; // 0: Norte, 1:Este, 2:Sur, 3:Oeste
      ultimaAccion = actIDLE;
    }

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

    Action think(Sensores sensores);

    int interact(Action accion, int valor);


    ComportamientoJugador * clone(){return new ComportamientoJugador(*this);}

  private:
    // Declarar Variables de Estado
    int fil, col, brujula;
    Action ultimaAccion;



};


#endif
```

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 “fil”, “col” y “brujula” reflejen fielmente la posición del agente. Un error en la actualización, debida a 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 “fil” y “col” sólo cambian si la anterior acción del agente fue avanzar (“actFORWARD”), y “brujula” sólo cambia cuando se realiza un giro. Así podemos plantear una estructura de decisión “switch-case” que contemple estas acciones del siguiente modo:

```
switch (ultimaAccion){
        case actFORWARD:
                // Que hacer en caso de que fuera avanzar
                break;
        case actTURN_L:
                // Que hacer en caso de que giró a la izquierda
                break;
        case actTURN_R:
                // Que hacer en caso de que giró a la derecha
                break;
}
```

En el caso de los giros está bien claro, si giró a la derecha y "brujula" vale 'a', el resultado debe ser que "brujula" valga "a+1". Como los valores de "brujula" válidos son del 0 al 3, hacemos que el resultado sea el módulo(a+1). De igual forma, en el giró a la izquierda, "brujula" decrementa en 1 su valor, es decir, módulo(a-1). Por consiguiente, el código anterior quedaría como

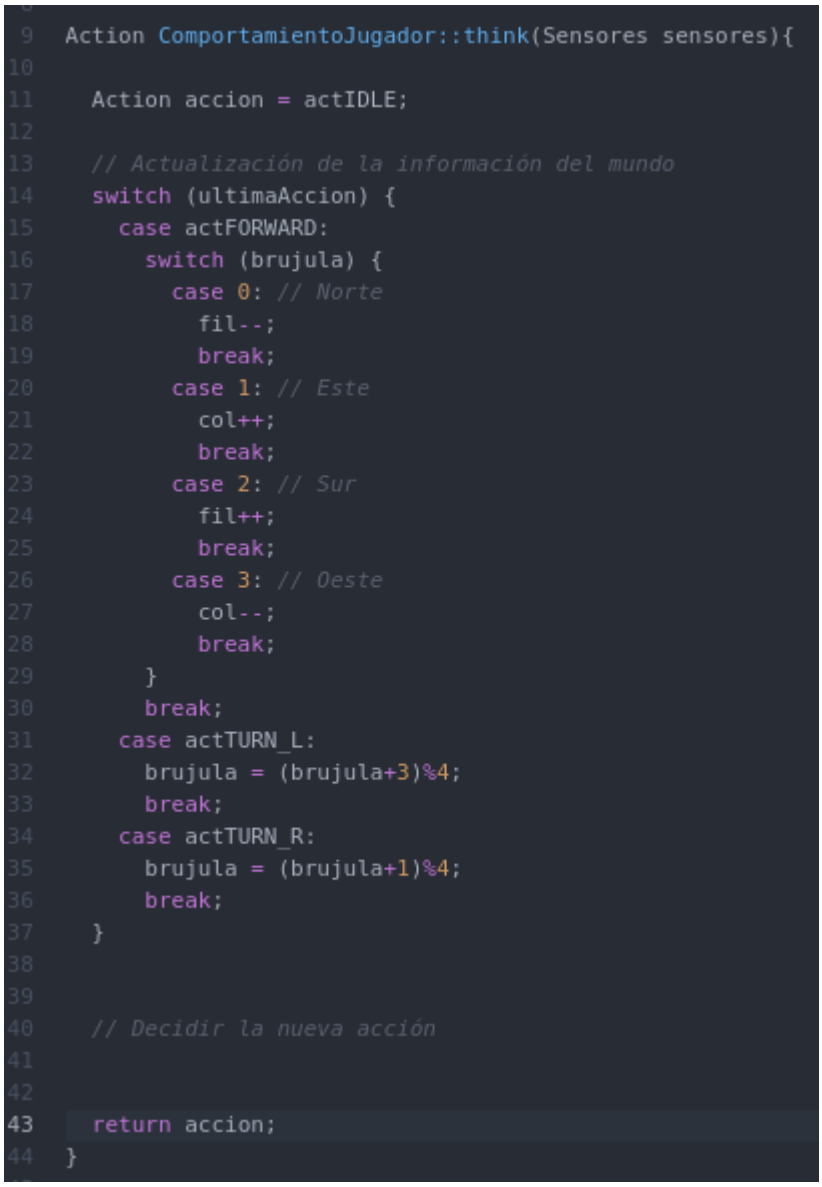
```
switch (ultimaAccion){
        case actFORWARD:
                // Que hacer en caso de que fuera avanzar
                break;
        case actTURN_L:
                brujula = (brujula+3)%4;
                break;
        case actTURN_R:
                brujula = (brujula+1)%4;
                break;
}
```

En el caso del giro a la izquierda el "***(brujula+3)%4***" es equivalente a "***(brujula-1+4)%4***" y equivalente a "***(brujula-1)%4)***". La razón de usar la primera, es que el operador "%" en C++, no va bien con los números negativos, por eso, hay que asegurarse que se aplica sobre números positivos.

La acción de avanzar depende de la orientación. En la convención que vamos a voy a usar, supondremos que el Norte está en los valores bajos de "fil" y el Sur en los valores altos de "fil". De igual modo, voy asociar el Este a valores altos de "col" y el Oeste a valores pequeños de "col". Por consiguiente, y siguiendo este convenio, avanzar estando orientado al Norte indica decrementar en una unidad el valor de la "fil", mientras que avanzar estando hacia el Sur es incrementar "fil". Así, el proceso de actualización quedaría como sigue:

```
switch (brujula){
        case 0: // Norte
                fil--;
                break;
        case 1: // Este
                col++;
                break;
        case 2: // Sur
                fil++;
                break;
        case 3: // Oeste
                col--;
                break;
}
```

Poniéndolo todo junto, tendríamos el código que muestra la figura.



Con esto, 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 objeto o personaje en esa casilla. En caso de que no podamos avanzar vamos a girar 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.superficie”). La regla es la siguiente:

```
if ((sensores.terreno[2]=='T' or
   sensores.terreno[2]=='S') and
 sensores.superfice[2]=='_'){
        accion = actFORWARD:

}
else{
        accion = actTURN_L;
}
```

Ya tendríamos casi listo este primer comportamiento. Sólo nos faltaría incluir una instrucción más. Antes de “return accion;”, debemos asignar a “ultimaAccion” 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 muestra la figura.

```
Action ComportamientoJugador::think(Sensores sensores){

  Action accion = actIDLE;

  // Actualización de la información del mundo
  switch (ultimaAccion) {
    case actFORWARD:
      switch (brujula) {
        case 0: // Norte
          fil--;
          break;
        case 1: // Este
          col++;
          break;
        case 2: // Sur
          fil++;
          break;
        case 3: // Oeste
          col--;
          break;
      }
      break;
    case actTURN_L:
      brujula = (brujula+3)%4;
      break;
    case actTURN_R:
      brujula = (brujula+1)%4;
      break;
  }

  // Decidir la nueva acción
  if ( (sensores.terreno[2]=='T' or sensores.terreno[2]=='S') and
        sensores.superficie[2]=='_'){
    accion = actFORWARD;
        }
  else {
    accion = actTURN_R;
  }

  // Recordar la ultima accion
  ultimaAccion = accion;
  return accion;
```

Ahora, compilad el software, ejecutadlo y seleccionar el mapa “mapa30.map”. Veréis que antes o después termina ciclando y no se mueve sobre el mapa.

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 cicla. Parece también lógico, que si en lugar de girar a la izquierda lo hiciera hacía la derecha, el resultado sería el mismo, pero qué ocurre si el giro depende de una variable aleatoria? Bueno, pues vamos a comprobar cual sería el resultado. Para eso, hacemos lo siguiente:

- 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 clase, también junto a las otras.
- En el fichero "jugador.cpp", en la parte de actualización, en los "case" relativos al giro, cada vez que se haya aplicado un giro en la acción anterior, volvemos a darle un nuevo valor a la variable "giro_derecha".
- En el fichero "jugador", en la parte del comportamiento, modificamos el comportamiento anterior por este:

```
if ((sensores.terreno[2]=='T' or
   sensores.terreno[2]=='S') and
 sensores.superfice[2]=='_'){
        accion = actFORWARD:
}
else if (!girar_derecha){
        accion = actTURN_L;
        if (rand()%2==0) girar_derecha = true;
        else girar_derecha = false;
}
else{
        accion = actTURN_R;
        if (rand()%2==0) girar_derecha = true;
        else girar_derecha = false;
}
```

Para usar la función "rand()" es necesario añadir la biblioteca "stdlib.h" en el fichero "jugador.cpp".

El método "think" quedaría de la siguiente forma:

```
Action ComportamientoJugador::think(Sensores sensores){

  Action accion = actIDLE;

  // Actualización de la información del mundo
  switch (ultimaAccion) {
    case actFORWARD:
      switch (brujula) {
        case 0: // Norte
          fil--;
          break;
        case 1: // Este
          col++;
          break;
        case 2: // Sur
          fil++;
          break;
        case 3: // Oeste
          col--;
          break;
      }
      break;
    case actTURN_L:
      brujula = (brujula+3)%4;
      if (rand()%2==0) girar_derecha=true;
      else  girar_derecha=false;
      break;
    case actTURN_R:
      brujula = (brujula+1)%4;
      if (rand()%2==0) girar_derecha=true;
      else  girar_derecha=false;
      break;
  }

  // Decidir la nueva acción
  return accion;
}
```

Tiene un comportamiento mejor que los anteriores, aunque está muy claro, que es necesario mejorarlo. Esto es sólo un ejemplo, en esta práctica estamos buscando una forma mejor de moverse. Pensad en alternativas, implementadlas y evaluad su comportamiento.

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”.

En esta matriz se van añadiendo los distintos elementos que forman parte del terreno del mapa cuando estamos seguros de su posición exacta. Para eso, necesitamos saber dónde nos encontramos, ya que inicialmente estamos en unas coordenadas desconocidas.

La forma de obtener dichas coordenadas es mediante las casillas 'K' representadas de color amarillo en el terreno. Cuando estamos encima de ellas los campos "mensajeF" y "mensajeC" de la variable de tipo registro "sensores" nos devuelven la fila y la columna de donde nos encontramos respectivamente. Para eso, antes tenemos que modificar el comportamiento anterior para permitir que el agente pase por dichas casillas. Para eso, sólo hace falta incluir la siguiente modificación en la primera regla:

```
if ((sensores.terreno[2]=='T' or
   sensores.terreno[2]=='S'  or
  sensores.terreno[2]=='K') and
 sensores.superfice[2]=='_'){
        accion = actFORWARD:
}
```

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

```
if ((sensores.terreno[0]=='K' and !bien_situado){
        fil = sensores.mensajeF;
        co l= sensores.mensajeC;
        bien_situado = true;
}
```

Se ha incluido una nueva variable de estado "bien_situado". Esta variable nos dice si el valor de las variables "fil" y "col" coincide con las del mapa original. 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").

Nuestro agente ya puede saber cuándo está bien situado, ahora sólo quedaría que toda la información que aparece en cada iteración en "sensores.terreno" se pase 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[fil][col]=sensores.terreno[0];
}
```

Colocado todo junto, el método "think" quedaría como muestra la figura siguiente.

```
Action ComportamientoJugador::think(Sensores sensores){

  Action accion = actIDLE;

  // Actualización de la información del mundo
  switch (ultimaAccion) {
    case actFORWARD:
      switch (brujula) {
        case 0: // Norte
          fil--;
          break;
        case 1: // Este
          col++;
          break;
        case 2: // Sur
          fil++;
          break;
        case 3: // Oeste
          col--;
          break;
      }
      break;
    case actTURN_L:
      brujula = (brujula+3)%4;
      if (rand()%2==0) girar_derecha=true;
      else  girar_derecha=false;
      break;
    case actTURN_R:
      brujula = (brujula+1)%4;
      if (rand()%2==0) girar_derecha=true;
      else  girar_derecha=false;
      break;
  }

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

  if (sensores.terreno[0]=='K' and !bien_situado){
    fil = sensores.mensajeF;
    col = sensores.mensajeC;
    bien_situado = true;
  }

  // Decidir la nueva acción
  if ( (sensores.terreno[2]=='T' or sensores.terreno[2]=='S') and
       sensores.superficie[2]=='_'){
    accion = actFORWARD;
       }
  else if (!girar_derecha) {
    accion = actTURN_L;
  }
  else {
    accion = actTURN_R;
  }

  // Recordar la ultima accion
  ultimaAccion = accion;
  return accion;
}
```

5. Las operación de reinicio

Algo importante en esta práctica es que hacer cuando el agente muere. El resultado de la muerte, ya sea por el consumo de todas las unidades de vida o por alguna situación durante la simulación que la haya producido, es siempre igual, se reaparece en otra casilla del mapa de coordenadas desconocidas y con orientación norte.

Por tanto, hay que estar atento al valor del sensor que indica este hecho. En nuestro caso, vamos a construir una función llamada "Reinicio" que se encarga de poner los valores apropiados a las variables de estado definidas para que el sistema siga funcionando correctamente.

En el método "think" sólo habría que añadir una estructura condicional que en función de dicho valor (sensores.reset), llame a la función "Reinicio" o siga con el proceso que ya se definió anteriormente.

6. Cómo determinar como de bueno es el comportamiento definido

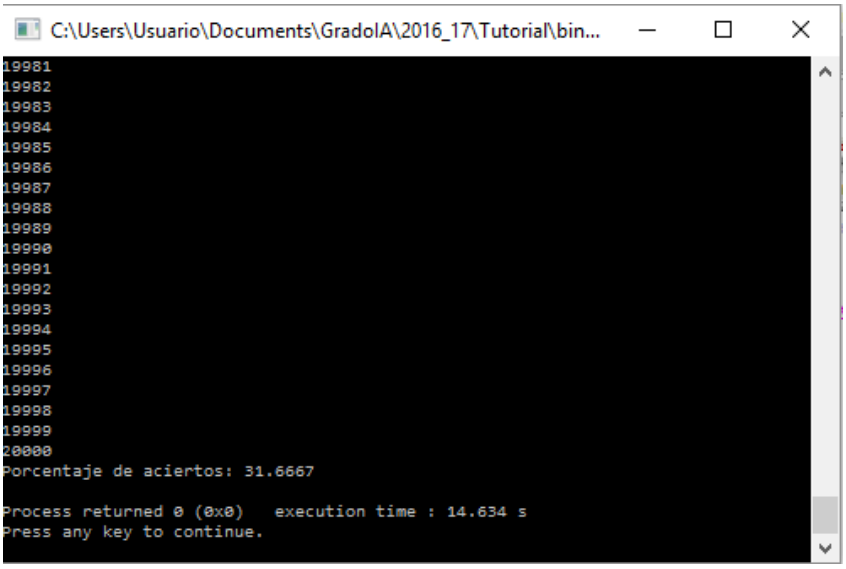
Llegados a este momento, podemos considerar que ya hemos definido una primera aproximación a un comportamiento muy simple para que el agente reconozca el mapa, aunque conforme avancéis en la práctica os daréis cuenta que demasiado básico y no considera muchos elementos que pueden ocurrir en el mundo y que son fundamentales para hacer un buen reconocimiento del terreno.

Ahora queremos considerar como de bueno es este comportamiento. Usando el software con entorno gráfico, el proceso de ejecutar las 20,000 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:

./BelkanSG ./mapas/mapa30.map

En el caso del software de Windows, debemos cerrar el proyecto "belkan1.cbp" y abrir el proyecto "belkan1SG.cbp", recompilar el software, desplegar el menú "Project->set program's arguments…" y en la ventana que aparece, en el campo "program arguments:", poner el nombre del mapa sobre el que se quiere ejecutar. En este caso pondremos "./mapas/mapa30.map".



El resultado dice que se ha reconocido correctamente el 31,67 % del mapa original. La intención de la práctica es definir un

comportamiento reactivo inteligente, por consiguiente, debe ser capaz de tener un buen comportamiento en mapas con topologías diversas. En este sentido, una valoración más adecuada de la aproximación debería ser la media sobre al menos los 5 mapas que se entregan con el software.

Los resultados sobre los 5 mapas se muestran en la siguiente tabla:

Mapa30	31,67 %
Mapa50	24,68 %
Mapa100	15,21 %
Mapa100n	2,61 %
Mapa100n2	2,87 %
Media	**15,41 %**

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

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 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, por qué será considerada como una práctica copiada e implicará suspender la asignatura.

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.