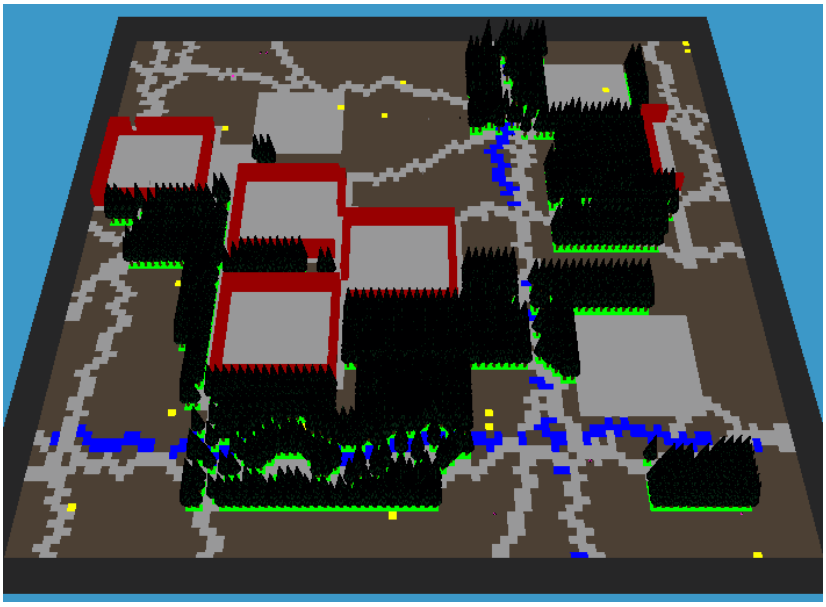
INTELIGENCIA ARTIFICIAL

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

Tutorial: Práctica 2

Agentes Reactivos/Deliberativos

(Los extraños mundos de BelKan)



DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN E INTELIGENCIA ARTIFICIAL

UNIVERSIDAD DE GRANADA

Curso 2019-2020

1. Introducción

El objetivo de la práctica es definir un comportamiento reactivo/deliberativo para un agente cuya misión es desplazarse por un mapa hacia un punto destino. Os aconsejamos que la construcción de la práctica se realice de forma incremental.

En este tutorial intenta ser una ayuda para poner en marcha la práctica y que el estudiante se familiarice con el software.

2. Mis primeros pasos

Vamos a empezar con un comportamiento reactivo simple semejante al que se ha visto en el ejercicio 1 de la relación de problemas 1. Para ello, eliminaremos completamente todas las sentencias que aparecen en el método ***think***.

2.1. Emulando a la hormiga

En dicho ejercicio se pedía que la hormiga siguiera un rastro de feromonas. En este caso, vamos a pedir a nuestro agente que mientras no tenga ningún impedimento avance en el mapa y si se encuentra un obstáculo gire a la derecha. ¿Cuáles son los obstáculos para el agente? Son las casillas etiquetadas con '**P**' y '**M**' y qué no estén ocupadas por un aldeano. Por consiguiente, vamos a definir un comportamiento donde el agente gira cuando tengo los valores anteriores y en otro caso, avanza.

En la siguiente imagen se muestra este comportamiento.

```
// Este es el método principal que debe contener los 4 Comportamientos_Jugad
// que se piden en la práctica. Tiene como entrada la información de los
// sensores y devuelve la acción a realizar.
Action ComportamientoJugador::think(Sensores sensores) {
  Action accion = actIDLE;

  if (sensores.terreno[2]=='P' or sensores.terreno[2]=='M' or
      sensores.superficie[2]=='a'){
        accion = actTURN_R;
      }
  else {
        accion = actFORWARD;
  }

  return accion;
}
```

Ilustración 1: Método think que simula el comportamiento visto de una hormiga.

Como se puede ver en la figura anterior, la implementación se hace en la función '**think**()' y hace uso de dos sensores. El sensor **terreno** que indica como es el terreno que tiene delante el agente con la estructura que se ilustra en el guion, y del sensor **superficie** que indica si la casilla está ocupada. En concreto, nos fijamos en la posición 2, que es justo la casilla que tiene enfrente el agente. En el caso de que dicha casilla no sea transitable (en eso se reduce a ser '**P**' o 'M') o que esté ocupada por un aldeano (el carácter '**a**') que miramos en el sensor **superficie** el agente gira a la derecha. En otro caso, el agente avanza.

Después de compilar y ejecutar el software, cargamos un mapa (mapa30.map por ejemplo) y pulsamos el botón de 'paso'. Veremos que el agente avanza hasta que se encuentra de frente con una casilla que no sea ninguna de las anteriores en cuyo caso gira.

En cualquier caso este ha sido tan solo un paso que nos permite ver en movimiento al agente, pero que no responde a lo que es necesario realizar en la práctica.

2.2. Usando la información del mapa

En el nivel 1 tenemos la información completa del mapa sobre el que el agente tiene que construir los caminos. ¿Dónde está esa información? En una variable llamada **mapaResultado** que el sistema se encarga de rellenar con el mapa original antes de que empiece la simulación en el nivel 1 y la rellena del carácter **'?'** en el nivel 2.

El agente tiene el control total sobre esta variable durante el resto de la simulación, de manera, que puede tanto leer como escribir sobre ella. El sistema sólo la usa para reproducir su contenido en el entorno gráfico. Esto último sólo lo hace cuando se está en el nivel 2.

Como ejemplo del uso de esta variable, vamos a intentar reproducir el comportamiento anterior *"avanzar cuando no tenga un obstáculo delante y girar a la derecha en otro caso"*, pero en lugar de usar los sensores terreno y superficie, teniendo en cuenta los valores de **mapaResultado**. Saber cuál es la casilla que tengo delante depende de mi orientación actual. Así, una posible forma de proceder se muestra en la ilustración 2.

```
// Este es el método principal que debe contener los 4 Comportamientos_Jugador
// que se piden en la práctica. Tiene como entrada la información de los
// sensores y devuelve la acción a realizar.
Action ComportamientoJugador::think(Sensores sensores) {
  Action accion = actIDLE;

  unsigned char contenidoCasilla;

  switch (sensores.sentido) {
    case norte: contenidoCasilla = mapaResultado[sensores.posF - 1][sensores.posC];
                break;
    case este: contenidoCasilla = mapaResultado[sensores.posF][sensores.posC + 1];
                break;
    case sur: contenidoCasilla = mapaResultado[sensores.posF + 1][sensores.posC];
                break;
    case oeste: contenidoCasilla = mapaResultado[sensores.posF][sensores.posC - 1];
                break;
  }


  if (contenidoCasilla=='P' or contenidoCasilla=='M' or
      sensores.superficie[2]=='a'){
        accion = actTURN_R;
      }
  else {
        accion = actFORWARD;
  }

  return accion;
}
```

Ilustración 2: Método think para usar la variable "mapaResultado" en lugar del sensor "superficie"

De esta manera, en la variable ***contenidoCasilla*** tengo el contenido de la casilla que tengo delante y ya podría preguntar si es una casilla de las transitables para el agente.

Se puede observar que sobre la versión anterior, se ha cambiado la invocación al sensor de terreno **sensores.terreno[2]**, por la nueva variable **contenidoCasilla**, y que se mantiene **sensores.superfice[2]** ya que es la única forma que tenemos de saber si hay un aldeano en esa casilla.

Probando el método anterior sobre el simulador para alguno de los comportamientos del nivel 1, veremos que se comporta como si usara el sistema sensorial. Sin embargo, si lo usamos sobre el nivel 2, veremos que no es capaz de detectar los obstáculos. La razón es que en ese nivel no se conoce el mapa y por tanto la variable mapaResultado contiene '?'.

Para este segundo caso, sería necesario actualizar la variable mapaResultado conforme avanzamos sobre el mapa en el nivel 2. Para ello, pasamos la información sensorial de "superficie" a dicha variable. En Ilustración 3, se muestra un ejemplo simple que pasa la información de lo que ve el agente justo delante por su sensor de visión a esta variable.

```
Action ComportamientoJugador::think(Sensores sensores) {
  Action accion = actIDLE;

  unsigned char contenidoCasilla;

  mapaResultado[sensores.posF][sensores.posC] = sensores.terreno[0];
  switch (sensores.sentido) {
    case norte: mapaResultado[sensores.posF - 1][sensores.posC] = sensores.terreno[2];
                break;
    case este: mapaResultado[sensores.posF][sensores.posC + 1] = sensores.terreno[2];
                break;
    case sur: mapaResultado[sensores.posF + 1][sensores.posC] = sensores.terreno[2];
                break;
    case oeste: mapaResultado[sensores.posF][sensores.posC - 1]= sensores.terreno[2];
                break;
  }


  if (sensores.terreno[2]=='P' or sensores.terreno[2]=='M' or
      sensores.superficie[2]=='a'){
        accion = actTURN_R;
      }
  else {
        accion = actFORWARD;
  }

  return accion;
}
```

Ilustración 3: Método think. Ejemplo para rellenar la variable mapaResultado en el nivel 2.

Si sobre el simulador se selecciona el mapa30.map y el nivel 2, tras incluir esta versión del método think(), se puede observar como a medida que el agente se mueve, va pintando sobre el mapa lo que percibe delante de él. Una versión extendida de esta idea, donde toda la información visual se lleve a mapaResultado tendrá que realizarse en la práctica para afrontar el nivel 2.

2.3. Controlando la ejecución de un plan

Nos situamos en el Nivel 1 sabemos que hay implementado un algoritmo de búsqueda en la función **PathFinding**, en concreto, el algoritmo de búsqueda en profundidad.

Echémosle un vistazo a la función PathFinding:

```
59  // Llama al algoritmo de busqueda que se usará en cada comportamiento del age
60  // Level representa el comportamiento en el que fue iniciado el agente.
61  bool ComportamientoJugador::pathFinding (int level, const estado &origen,
62    const estado &destino, list<Action> &plan){
63    switch (level){
64      case 1: cout << "Busqueda en profundad\n";
65              return pathFinding_Profundidad(origen,destino,plan);
66              break;
67      case 2: cout << "Busqueda en Anchura\n";
68              // Incluir aqui la llamada al busqueda en anchura
69              break;
70      case 3: cout << "Busqueda Costo Uniforme\n";
71              // Incluir aqui la llamada al busqueda de costo uniforme
72              break;
73      case 4: cout << "Busqueda para el reto\n";
74              // Incluir aqui la llamada al algoritmo de búsqueda usado en el n
75              break;
76    }
77    cout << "Comportamiento sin implementar\n";
78    return false;
79  }
```

Ilustración 4: Métodos pathfinding.

Podemos ver que tiene 4 parámetros: ***level*** que indica el algoritmo de búsqueda a utilizar, ***origen*** y ***destino*** que establecen los puntos de inicio y fin del camino a trazar sobre el mapa y ***plan*** que devuelve la lista de acciones a realizar para ir desde origen hasta destino.

Vemos que en función del valor de ***level*** se dispara un caso distinto de ***switch***. Por tanto, ***level*** es un valor entre 1 y 4 y está asociado a lo que se ha dado en llamar en la práctica el comportamiento. Así, ***level*** con valor entre 1 y 3 son los tres comportamientos pedidos en el *nivel 1*, mientras que ***level*** con valor 4 es el comportamiento 4 o *nivel 2*. Aquí en realidad se decide que algoritmo de búsqueda se va a utilizar en cada nivel. En el nivel 1 ya se encuentran fijados (profundidad, anchura y coste uniforme), mientras que para el comportamiento 4 (o nivel 2) el estudiante puede decidir que algoritmo de búsqueda usar, o bien uno de los anteriores, o bien definir de entre los vistos en clase.

Podemos ver que para el comportamiento 1 (***level*** = 1), ya se encuentra la llamada al algoritmo de búsqueda en profundidad (que se encuentra implementado en el software de partida), que nos devuelve en ***plan*** la secuencia de acciones a realizar para alcanzar la casilla objetivo.

Así, la función **think** se encargara de invocar a **PathFinding** para construir un camino que lleve al agente a la casilla objetivo y por otro lado, controlar la ejecución del plan.

Para llevar a cabo esta tarea de control, se definen tres variables de estado, **hayplan**, **destino** y **plan** en el fichero “jugador.hpp”.

```
private:
  // Declarar Variables de Estado
  estado actual, destino;
  list<Action> plan;

  // Nuevas variables de Estado
  Action ultimaAccion;
  bool hayPlan;
```

Ilustración 5: Inclusión de nuevas variables de estado para el control del plan

La primera de ellas (hayplan) es una variable lógica que toma el valor verdadero cuando ya se ha construido un plan viable. La segunda variable es de tipo **estado** y se usará para almacenar las coordenadas de la casilla destino. Por último, **plan** que es de tipo lista de acciones almacenará la secuencia de acciones que permite al agente trasladarse a la casilla objetivo.

En los constructores de clase es necesario inicializar la variable **hayPlan** a falso. Las otras variables no es necesario inicializarlas. El método ***think*** quedaría como muestra la ilustración 6.

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

  // Calculamos el camino hasta el destino si no tenemos aun un plan
  if (!hayPlan){
    actual.fila = sensores.posF;
    actual.columna = sensores.posC;
    actual.orientacion = sensores.sentido;
    destino.fila = sensores.destinoF;
    destino.columna = sensores.destinoC;
    hayPlan = pathFinding(sensores.nivel, actual, destino, plan);
  }

  Action sigAccion;
  if (hayPlan and plan.size()>0){ // Hay un plan no vacio
    sigAccion = plan.front(); //tomamos la siguiente accion del hayPlan
    plan.erase(plan.begin()); //eliminamos la acción de la lista de acciones
  }
  else {
    // Aqui solo entra cuando no es posible encontrar un comportamiento
    // o está mal implementado el método de busqueda
  }


  return sigAccion;
}
```

Ilustración 6: Método think. Siguiendo un plan.

La invocación a **PathFinding** es muy simple. Se crea una variable de tipo **estado** para almacenar la posición actual del agente. Se asigna en la variable de estado **destino** las coordenadas de la casilla destino provenientes de los sensores **destinoF** y **destinoC**. Tras invocar a la función **PathFinding**, en plan tendremos la secuencia de acciones y **hayPlan** toma el valor **true**. El primer argumento de PathFinding hace uso de sensor ***nivel***, que determina que algoritmo se seleccionará para construir el camino.

La parte de seguir el plan es muy simple. Mientras exista plan y la lista de acciones sea mayor que cero, se toma la siguiente acción y se elimina de la lista.

En el nivel 1, donde el mapa es conocido y no hay otros agentes móviles en el mapa, el camino proyectado debe funcionar. Sin embargo, en el nivel 2, donde puede haber partes del mapa desconocidos y lo que pueden entorpecer los aldeanos los caminos planificados, obligan a definir un comportamiento que permita corregir las contingencias y en su caso replanificar a la luz del nuevo conocimiento.

3. Algunas preguntas frecuentes

(a) ¿Se puede escribir sobre la variable mapaResultado?

mapaResultado es una variable que podéis considerar como una variable global. En el nivel 1 contiene el mapa completo y se usa en el algoritmo de búsqueda para encontrar el camino. En el nivel 2 contiene en todas las casillas **'?'**, es decir, que indica que no se sabe el contenido de ninguna casilla. Por tanto, en este nivel hay que ir construyendo el mapa para poder hacer un camino, y con consiguiente es imprescindible escribir en ella. Para poder escribir sobre **mapaResultado** es necesario estar seguro de estar posicionado en el mapa. Así, si suponemos que **fil** contiene la fila exacta donde se encuentra el agente y **col** es la columna exacta, entonces:

mapaResultado[fil][col] = sensores.terreno[0];

coloca en el mapa el tipo de terreno en el que está en ese instante el agente.

(b) ¿Se pueden declarar funciones adicionales en el fichero "jugador.cpp"?

Por supuesto, se pueden definir tantas funciones como necesitéis. De hecho es recomendable para que el método **Think()** sea entendible y sea más fácil incorporar nuevos comportamientos.

(c) ¿Puedo entregar 2 parejas de ficheros "jugador.cpp", "jugador.hpp" uno para cada uno de los niveles?

No. Sólo se puede entregar un par jugador.cpp jugador.hpp que sea aplicable a todos los niveles que se hayan resuelto.

(e) Mi programa da un “core” ¿Cómo lo puedo arreglar?

La mayoría de los “segmentation fault” que se provocan en esta práctica se deben a direccionar posiciones de matrices o vectores fuera de su rango. Como recomendación os proponemos que en el código verifiquéis antes de invocar a una matriz o a un vector el valor de las coordenadas y no permitir valores negativos o mayores o iguales a las dimensiones de la matriz o el vector.

4. 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.