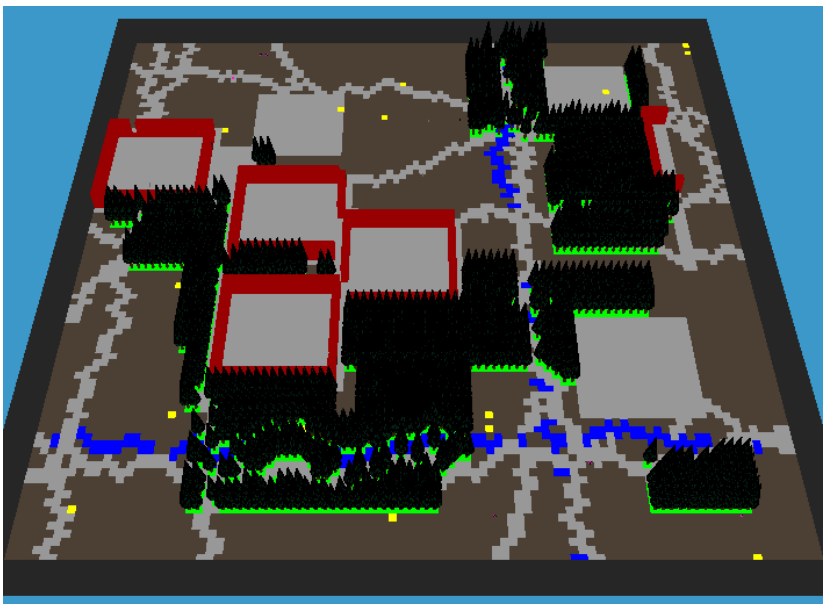
INTELIGENCIA ARTIFICIAL

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 2020-2021

1.Introducción

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

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**' (precipicio) y '**M**' (muro), y que 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 Ilustración 1 se muestra este comportamiento.

```
// 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;

  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 los niveles 0,1,2 y 3 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 los primeros cuatro niveles y la rellena con el carácter **'?'** en el nivel 4.

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.

```
// 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"

mapaResultado se usa en todos los niveles para construir el plan, ya que en esa variable se encuentra la información sobre el contenido del mapa. La única diferencia es que en los primeros niveles (0, 1, 2 y 3), **mapaResultado** contiene el mapa sin incertidumbre sobre el terrero, mientras

que en el nivel 4, el terreno no se conoce. Por tanto, el agente debe ir explorando, moviéndose por el mundo, y escribiendo el valor de los sensores en **mapaResultado** para ir recomponiendo el mapa. En ese caso, cada vez que se escribe un valor en **mapaResultado**, se refresca automáticamente el entorno gráfico y el nuevo valor se pinta en el mapa.

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.

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 de los niveles 0, 1, 2 y 3 veremos que se comporta como si usara el sistema sensorial. Sin embargo, si lo usamos sobre el nivel 4, 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 '**?**'.

```
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 4.

Para este segundo caso, sería necesario actualizar la variable **mapaResultado** conforme avanzamos sobre el mapa en el nivel 4. Para ello, pasamos la información sensorial de "superficie" a dicha variable. En Ilustración 2, 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.

Si sobre el simulador se selecciona el mapa30.map y el nivel 4, tras incluir esta versión del método **think()**, se puede observar cómo 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 4.

2.3. Controlando la ejecución de un plan

Si nos situamos en el Nivel 0 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 en la ilustración 4.

```
47  // Llama al algoritmo de busqueda que se usara en cada comportamiento del agente
48  // Level representa el comportamiento en el que fue iniciado el agente.
49  bool ComportamientoJugador::pathFinding (int level, const estado &origen,
50    const list<estado> &destino, list<Action> &plan){
51    switch (level){
52      case 0: cout << "Demo\n";
53              estado un_objetivo;
54              un_objetivo = destino.front();
55              return pathFinding_Profundidad(origen,un_objetivo,plan);
56              break;
57
58      case 1: cout << "Optimo numero de acciones\n";
59              // Incluir aqui la llamada a la busqueda nivel 1
60              cout << "No implementado aun\n";
61              break;
62      case 2: cout << "Optimo en coste 1 Objetivo\n";
63              // Incluir aqui la llamada a la busqueda nivel 2
64              cout << "No implementado aun\n";
65              break;
66      case 3: cout << "Optimo en coste 3 Objetivos\n";
67              // Incluir aqui la llamada a la busqueda nivel 3
68              cout << "No implementado aun\n";
69              break;
70      case 4: cout << "Algoritmo de busqueda usado en el reto\n";
71              // Incluir aqui la llamada a la busqueda nivel 4
72              cout << "No implementado aun\n";
73              break;
74    }
75    return false;
76  }
```

Ilustración 4: Métodos pathfinding.

Podemos ver que tiene 4 parámetros: ***level*** que indica el algoritmo de búsqueda a utilizar, ***origen*** establece la casilla donde empieza el camino y ***destino*** que especifica las casillas destino (por eso es una lista de estados) y ***plan*** que devuelve la lista de acciones a realizar para ir desde origen hasta la lista de destinos.

Vemos que en función del valor de ***level*** se dispara un caso distinto de ***switch***. Por tanto, ***level*** es un valor entre 0 y 4 y está asociado al nivel seleccionado por el usuario a la hora de invocar al software. Aquí, en realidad, se decide qué algoritmo de búsqueda se va a utilizar en cada nivel. En el nivel 0 ya se encuentra implementado el algoritmo de búsqueda en profundidad. En el nivel 1 se

espera que se implemente el algoritmo de búsqueda en anchura. En los niveles 2 y 3 el algoritmo de costo uniforme o el algoritmo A*, y en el nivel 4, el estudiante puede usar uno de los ya implementados en los niveles anteriores o bien implementar uno nuevo (entre los vistos en clase) que considere más apropiado para resolver ese nivel.

Podemos ver que para el comportamiento 0 (***level*** = 0), 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. Debido a que en el ***nivel 0*** sólo se espera una casilla objetivo (de tipo **estado**), y el parámetro de entrada **destino** es una lista de **estado**, antes de invocar a la *búsqueda en profundidad* se extrae la primera casilla de esa lista de estados (**un_objetivo** = **destino**.**front**()) que será usado como parámetro destino en la llamada al proceso de búsqueda. Esta forma de actuar será semejante cuando se especifiquen los niveles 1 y 2, ya que en ellos también se espera sólo un objetivo. No será así en el nivel 3, ya que se requieren los 3 objetivos para su funcionamiento. En el nivel 4, será el estudiante el que decida si pasa sólo uno de los objetivos o pasa los tres al algoritmo de búsqueda.

Así, la función **think** se encargará 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 cuatro variables de estado, **actual**, **destino, plan** y **hayplan** en el fichero "**jugador**.**hpp**".

```
private:
  // Declarar Variables de Estado
  estado actual;
  list<estado> objetivos;
  list<Action> plan;
  bool hayPlan;
```

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

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

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.

Podemos distinguir 4 bloques:

- líneas de la 15 a la 18, se actualiza la variable **actual** que mantiene las coordenadas y orientación del agente dentro del mapa. Dicha actualización se realiza accediendo directamente a la información sensorial del agente.
- líneas de la 20 a la 27, se actualiza la lista de casillas meta y se almacena en la variable de estado **objetivos**. Al igual que el caso anterior, se hace accediendo al sistema sensorial.

```
13  Action ComportamientoJugador::think(Sensores sensores) {
14
15    // actualizo la variable actual
16    actual.fila        = sensores.posF;
17    actual.columna     = sensores.posC;
18    actual.orientacion = sensores.sentido;
19
20    // Capturo los destinos
21    objetivos.clear();
22    for (int i=0; i<sensores.num_destinos; i++){
23      estado aux;
24      aux.fila = sensores.destino[2*i];
25      aux.columna = sensores.destino[2*i+1];
26      objetivos.push_back(aux);
27    }
28
29    // Si no hay plan, construyo uno
30    if (!hayPlan){
31      hayPlan = pathFinding (sensores.nivel, actual, objetivos, plan);
32    }
33
34    Action sigAccion;
35    if (hayPlan and plan.size()>0){ //hay un plan no vacio
36      sigAccion = plan.front(); //tomo la siguiente accion del Plan
37      plan.erase(plan.begin()); //eliminamos la accion del plan.
38    }
39    else {
40      cout << "no se pudo encontrar un plan\n";
41    }
42
43    return sigAccion;
44  }
```

Ilustración 6: Método think. Siguiendo un plan.

- líneas de la 29 a la 32, se invoca a la construcción de un camino, si no hay ya un plan activo. La función PathFinding invocará en función del nivel seleccionado por el usuario (sensores.nivel) al algoritmo de búsqueda.
- líneas de la 34 a la 41, se produce el control del plan. Mientras exista el plan, se toma la primera acción del plan y se almacena en **sigAccion**, se saca de la lista de acciones y se prepara para ser ejecutada por el agente.

Por último, se devuelve la acción seleccionada.

3. Algunas preguntas frecuentes

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

mapaResultado es una variable que podéis considerar como una variable global. En los niveles 0, 1, 2 y 3 contiene el mapa completo y se usa en el algoritmo de búsqueda para encontrar el camino. En el nivel 4 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. Obviamente, teniendo en cuenta la orientación actual del agente, se puede poner sobre **mapaResultado** toda la información que da el sensor del terreno.

(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 parejas de ficheros “jugador.cpp”, “jugador.hpp” 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.

(d) 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.