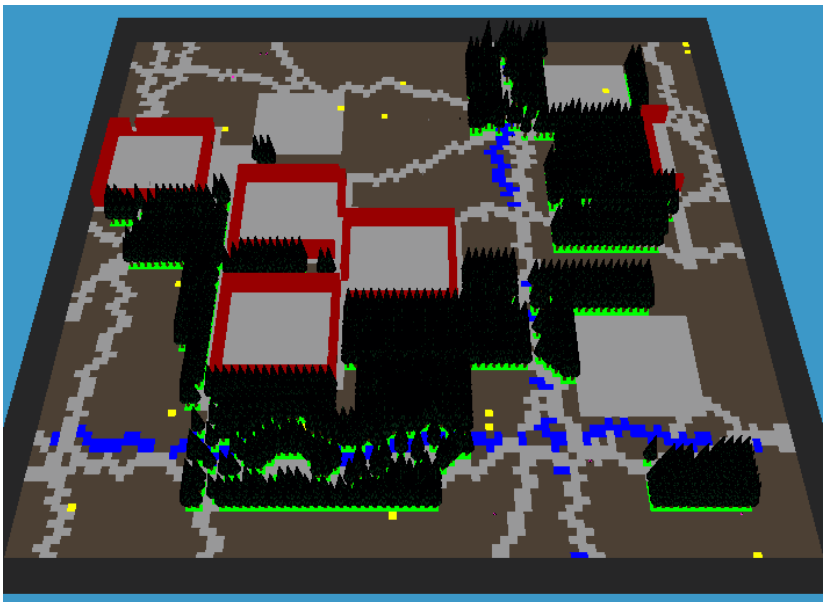
INTELIGENCIA ARTIFICIAL

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 2017-2018

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 familiarize 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**', '**M**' y '**D**' 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
// 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) {
  if (sensores.terreno[2]=='P' or sensores.terreno[2]=='M' or
      sensores.terreno[2]=='D' or sensores.superficie[2]=='a'){
    return actTURN_R;
  }
  else{
    return actFORWARD;
  }
}
```

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**', '**M**' o '**D**') o que esté ocupada por un aldeano 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. **Incluyendo memoria al agente**

Una cuestión muy importante para el nivel 2 es que el agente sepa en todo momento su posición correcta dentro del mapa, ya que cuando está en esta situación puede invocar al '***PathFinding***' (el proceso que le permite obtener un camino entre su posición actual y la casilla marcada como objetivo. En nuestro problema, estar correctamente ubicado consiste en conocer la fila y la columna donde me encuentro, así como la orientación actual. Para almacenar dicha información vamos a usar 4 variables de estado: **fil** y **col** que indican fila y columna actual respectivamente, **brujula** que indica la orientación y **ultimaAccion** que será la variable que nos va a ayudar a actualizar las variables de estado anteriores. Las tres primeras se declaran como enteras (ya vienen declaradas en el fichero 'jugador.hpp' en la versión original del software) y **ultimaAccion** se declara de tipo **Action** y tenemos que incluirla en el fichero 'jugador.hpp' junto con las anteriores quedando como muestra la siguiente imagen.

```
private:
  // Declarar Variables de Estado
  int fil, col, brujula;
  estado destino;
  list<Action> plan;

  // Nuevas Variables de Estado
  Action ultimaAccion;
```

Además, es importante inicializar las variables **brujula** y **ultimaAccion** en los constructores de la clase quedando como se muestra a continuación.

```
class ComportamientoJugador : public Comportamiento {
  public:
    ComportamientoJugador(unsigned int size) : Comportam
      // Inicializar Variables de Estado
      fil = col = 99;
      brujula = 0; // 0: Norte, 1:Este, 2:Sur, 3:Oeste
      destino.fila = -1;
      destino.columna = -1;
      destino.orientacion = -1;
      ultimaAccion = actIDLE;
    }
    ComportamientoJugador(std::vector< std::vector< unsi
      // Inicializar Variables de Estado
      fil = col = 99;
      brujula = 0; // 0: Norte, 1:Este, 2:Sur, 3:Oeste
      destino.fila = -1;
      destino.columna = -1;
      destino.orientacion = -1;
      ultimaAccion = actIDLE;
    }
    ComportamientoJugador(const ComportamientoJugador &
    ~ComportamientoJugador(){}
```

Se puede observar, que se ha inicializado con el valor **actIDLE**, indicando que antes de empezar la simulación el agente no hizo ninguna acción. Por otro lado, también se puede ver que ya viene inicializada la variable **brujula**. Hay 4 valores posibles para esta variable: norte, este, sur y oeste. En este caso, se ha tomado el criterio de considerar que el valor 0 indica orientación norte, 1 orientación este, 2 orientación sur y 3 orientación oeste. Como en el enunciado nos dicen que el agente siempre aparece orientado al norte, entonces la inicialización de **brujula** debe ser a 0.

```
13  Action ComportamientoJugador::think(Sensores sensores) {
14    // Capturar los valores de filas y columnas
15    if (sensores.mensajeF != -1){
16      fil = sensores.mensajeF;
17      col = sensores.mensajeC;
18    }
19
20    // Actualizar el efecto de la última acción
21    switch (ultimaAccion) {
22      case actTURN_R: brujula = (brujula+1)%4; break;
23      case actTURN_L: brujula = (brujula+3)%4; break;
24      case actFORWARD:
25          switch (brujula) {
26            case 0: fil--; break;
27            case 1: col++; break;
28            case 2: fil++; break;
29            case 3: col--; break;
30          }
31        break;
32    }
33    cout << "Fila: " << fil << "  Col: " << col << "  Or: " << brujula << endl;
34
35    // Sistema de Movimiento
36    Action sigAccion;
37    if (sensores.terreno[2]=='P' or sensores.terreno[2]=='M' or
38        sensores.terreno[2]=='D' or sensores.superficie[2]=='a'){
39      sigAccion = actTURN_R;
40    }
41    else{
42      sigAccion = actFORWARD;
43    }
44
45    // Recordar la última accion
46    ultimaAccion = sigAccion;
47    return sigAccion;
48  }
```

Una vez que tenemos declaradas las variables de estado anteriores, en el método '***think***' de '***jugador.cpp***' debemos incluir el proceso que nos permita ir actualizando los valores de estas variables. Antes de hacer esto, debemos recordar que en la primera iteración del nivel 1 el entorno nos devuelve nuestra posición en los sensores ***mensajeF*** y ***mensajeC***. Estos sensores devuelven el valor -1 cuando no tienen información. Aprovecharemos esta circunstancia para capturar los valores iniciales de **fil** y **col**. La implementación de todo este proceso se muestra en la anterior imagen.

Mirando el método de arriba a abajo podemos considerar cuatro bloques que nos ayudan a describir que hace.

- El primero de ellos que va de la línea 13 a la línea 18 monitoriza en cada iteración el valor del sensor **mensajeF**. Si dicho sensor se activa (es decir, manda un valor distinto de -1) entonces actualizamos nuestras variables de estado **fil** y **col** con los valores que aportan los sensores.
- El segundo bloque que va de la línea 20 a la línea 32 es el encargado de mantener correctamente las variables de estado **fil**, **col** y **brujula** en base a la última acción que se realizó. En el caso de los giros a la derecha y a la izquierda, sólo tengo que sumar 1 o -1 al valor anterior de **brujula**. La expresión se ve algo más compleja, ya que esta variable debe tomar valores entre 0 y 3 (por eso lo del módulo). La operación módulo (%) trabaja con números enteros positivos, y como (a-1)%N es equivalente a (a+(N-1))%N, aplicamos esta última para evitar que **brujula** tome valores negativos.

  En el caso de ser la última acción avanzar, modifico las variables **fil** y **col** dependiendo de la orientación. En el caso de *norte* y **sur** (valores 0 y 2 de **brujula**) se modifica **fil**. En los otros dos casos (*este* y *oeste*) se modifica **col**.

  En el final de este bloque se escribe por el dispositivo de salida los valores de esas variables de estado para que podamos comprobar que lo hace correctamente. Obviamente, esta sentencia se podría omitir.
- El tercer bloque que va de la línea 35 a la 43, aparece el comportamiento que vimos en la sección anterior. La única diferencia es que en lugar de usar directamente 'return', el resultado se guarda en la variable **sigAccion**.
- El último bloque que incluye las dos últimas sentencias del método, tiene por un lado el almacenar en la variable de estado **ultimaAccion** la acción que se va a devolver, para poder así usarla en la siguiente iteración en el bloque segundo. Por último, se devuelve la acción seleccionada en el tercer bloque.

¿Qué ocurre si el agente pasa por una casilla amarilla (las etiquetadas como 'K') con el comportamiento anterior? Os propongo que para la resolución de ese problema definàis una variable de estado adicional de tipo lógico (de nombre **estoy_bien_situado**) y luego inicializarla en los constructores con el valor false. Por último, Incluid esta variable de estado en la condición del primer bloque.

2.3. 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 de 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**. Para ello, vamos a suponer que estamos bien situados en el mapa y por consiguiente las variables de estado **fil**, **col** y **brujula** reflejan mi posición correcta en el mapa.

Saber cúal es la casilla que tengo delante depende de mi orientación actual. Así, una posible forma de proceder sería la siguiente:

```
unsigned char contenidoCasilla;
switch(brujula){
  case 0: contenidoCasilla = mapaResultado[fil-1][col]; break;
  case 1: contenidoCasilla = mapaResultado[fil][col+1]; break;
  case 2: contenidoCasilla = mapaResultado[fil+1][col]; break;
  case 3: contenidoCasilla = mapaResultado[fil][col-1]; break;
}
```

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, quedando el método como se muestra en la siguiente imagen:

```
33  cout << "Fila: " << fil << "  Col: " << col << "  Or: " << brujula << endl;
34
35  // Sistema de Movimiento
36  unsigned char contenidoCasilla;
37    switch(brujula){
38      case 0: contenidoCasilla = mapaResultado[fil-1][col]; break;
39      case 1: contenidoCasilla = mapaResultado[fil][col+1]; break;
40      case 2: contenidoCasilla = mapaResultado[fil+1][col]; break;
41      case 3: contenidoCasilla = mapaResultado[fil][col-1]; break;
42    }
43
44  Action sigAccion;
45  if (contenidoCasilla=='P' or contenidoCasilla=='M' or
46      contenidoCasilla=='D' or sensores.superficie[2]=='a'){
47    sigAccion = actTURN_R;
48  }
49  else{
50    sigAccion = actFORWARD;
51  }
52
53  // Recordar la última accion
54  ultimaAccion = sigAccion;
55  return sigAccion;
56 }
```

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.

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.

Echemosle un vistazo a la función PathFinding:

```
59  // Llama al algoritmo de busqueda que se usará en cada comportamiento del agente
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 nivel 2
75              break;
76    }
77    cout << "Comportamiento sin implementar\n";
78    return false;
79  }
```

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”.

```
45    private:
46      // Declarar Variables de Estado
47      int fil, col, brujula;
48      estado destino;
49      list<Action> plan;
50
51      // Nuevas Variables de Estado
52      Action ultimaAccion;
53      bool hayPlan;
```

La primera de ellas 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:

```
// Este es el método principal que debe contener los 4 Comportamientos
// 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) {
  // Capturar los valores de filas y columnas
  if (sensores.mensajeF != -1){
    fil = sensores.mensajeF;
    col = sensores.mensajeC;
    ultimaAccion = actIDLE;
  }

  // Actualizar el efecto de la última acción
  switch (ultimaAccion) {
    case actTURN_R: brujula = (brujula+1)%4; break;
    case actTURN_L: brujula = (brujula+3)%4; break;
    case actFORWARD:
        switch (brujula) {
          case 0: fil--; break;
          case 1: col++; break;
          case 2: fil++; break;
          case 3: col--; break;
        }
      break;
  }

  // Mirar si ha cambiado el destino
  if (sensores.destinoF != destino.fila or sensores.destinoC != destino.columna){
    destino.fila = sensores.destinoF;
    destino.columna = sensores.destinoC;
    hayPlan = false;
  }

  // Calcular un camino hasta el destino
  if (!hayPlan){
    actual.fila = fil;
    actual.columna = col;
    actual.orientacion = brujula;
    hayPlan = pathFinding(sensores.nivel, actual, destino, plan);
  }


  Action sigAccion;
  if (hayPlan and plan.size()>0){  // Hay un plan y hay que seguirlo
    sigAccion = plan.front();
    plan.erase(plan.begin());
  }
  else {  // No hay plan y se activa un comportamiento reactivo
    if (sensores.terreno[2]=='P' or sensores.terreno[2]=='M' or
        sensores.terreno[2]=='D' or sensores.superficie[2]=='a'){
      sigAccion = actTURN_R;
    }
    else{
      sigAccion = actFORWARD;
    }
  }

  // Recordar la última accion
  ultimaAccion = sigAccion;
  return sigAccion;
}
```

En la función ***think*** podemos distinguir 6 bloques:

1. Capturar la fila y la columna en la que se encuentra el agente. (líneas 14 a 19).
2. Actualización de **fil** y **col** en función de la última acción aplicada. (líneas 21 a 33).
3. Detección del cambio de la casilla objetivo. (líneas 35 a 40).
4. Generación de un camino de la posición actual del agente a la casilla objetivo. (líneas 42 a 48).
5. Control del plan. (líneas 52 a 64).
6. Actualizar y enviar la acción elegida. (líneas 66 a 68).

Los bloques 1, 2 y 6 coinciden con los vistos en los ejemplos anteriores. Nos centramos ahora en describir los otros tres bloques.

El bloque 3 de detección del cambio de la casilla destino no es realmente necesario para la ejecución de la práctica, pero si que puede ser útil para que el programador pueda experimentar con distintas casillas destinos y probar el comportamiento del algoritmo de búsqueda.

Es muy simple de describir y lo que hace es que si se está ejecutando un plan (**hayPlan** es **true**) y el destino actual detectado en los sensores **destinoF** y **destinoC** no coindicen con el destino para el que se construyó el plan (variable de estado **destino**), entonces poner la variable **hayPlan** a falso.

El bloque 4 es el encargado de invocar a PathFinding para construir un camino. La única condición ncesaria para llamar a este método es que no haya plan.

La invocación 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 contruir el camino.

El bloque 5 se encarga de controlar la ejecución del plan construido. Los pasos son: si hay plan entonces se toma la primera acción de la lista de acciones y se elimina dicha acción de la lista. Si no hay plan, no se hace nada.

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.