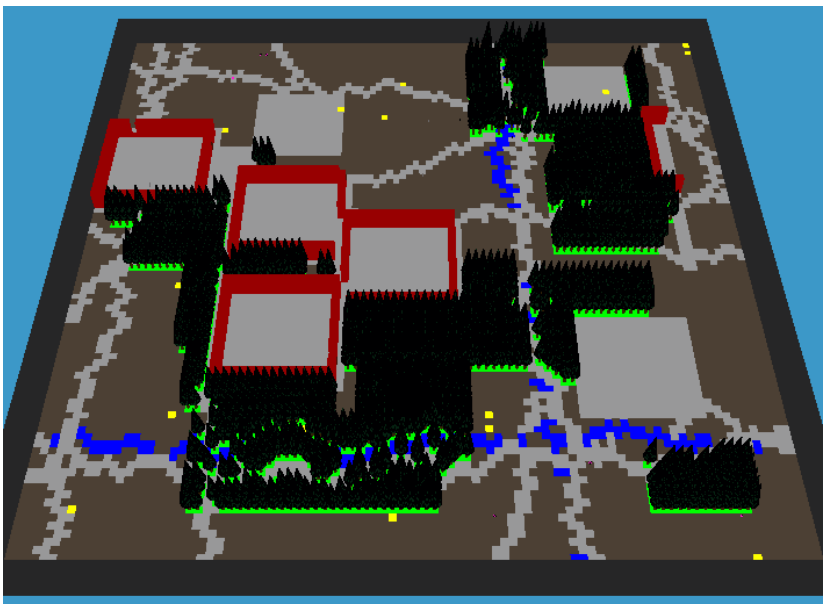
INTELIGENCIA ARTIFICIAL

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

Tutorial: Práctica 2

Agentes Reactivos/Deliberativos



DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN E INTELIGENCIA ARTIFICIAL

UNIVERSIDAD DE GRANADA

Curso 2021-2022

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

El objetivo de la práctica 1 ha sido el desarrollo de un comportamiento reactivo sobre un mundo muy parecido al que se describe y con el que trabajaremos durante la práctica 2. La mayoría de los estudiantes estarán muy familiarizados con estos comportamiento y con los elementos que se vuelven a repetir en esta segunda práctica. Por esta razón, indicamos a aquellos estudiantes que no tengan claro este manejo, realicen el tutorial de la práctica 1 antes de leer este. Aunque en esta segunda, los agentes reactivos no están el centro del estudio, si que son necesarios para complementar los comportamientos deliberativos que aquí se verán.

Así, que para este tutorial asumiremos que se ha realizado el tutorial de la práctica anterior y nos centraremos en la forma en la que combinar los comportamientos deliberativos y reactivos.

La estructura del software que se proporciona es semejante al utilizado en la práctica anterior, y al igual que en aquella, la forma de proceder para la realización de la tarea consiste en construir la función ***think()*** para resolver los distintos problemas que se plantean. La función **Action *think*(Sensores sensores)** toma como parámetro de entrada el valor de los sensores del agente en el instante actual, y devuelve un acción. Esta forma de actuar se asemeja al comportamiento de un agente reactivo. Como en esta práctica se quieren evaluar comportamientos deliberativos necesitaremos adaptar el comportamiento de un agente deliberativo dentro de este esquema concreto. Lo que el agente hará será calcular un plan (en su versión más genérica de esta práctica, una secuencia de movimientos que permita llevar al agente desde la casilla donde se encuentra a una casilla destino) que se almacenará en una variable de estado. El método ***think()*** en cada iteración de la simulación irá aportando la siguiente acción de ese plan, hasta que el plan se termine. Esto sería una descripción a grandes rasgos de lo que se pretende. En la siguiente sección “Controlando la ejecución de un plan” veremos como esto se implementa en la práctica aplicándolo para completar el nivel 0 de la misma. Lo que se describe en dicha sección será válido también para los niveles 1 y 2. En esos niveles lo que faltará será implementar los algoritmos de búsqueda. Por últiimo, en los niveles 3 y 4, será necesario involucrar algunos comportamiento reactivos y combinarlos con los deliberativos para mejorar el funcionamiento del agente para reconocer mayor porcentaje del mapa o para aumentar la capacidad de conseguir objetivos.

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, una versión del algoritmo de búsqueda en profundidad.

Echémosle un vistazo a la función PathFinding en la ilustración 1.

```
40  // Llama al algoritmo de busqueda que se usara en cada comportamiento del agente
41  // Level representa el comportamiento en el que fue iniciado el agente.
42  bool ComportamientoJugador::pathFinding(int level, const estado &origen, const list<estado> &destino, list<Action> &plan)
43  {
44      switch (level)
45      {
46      case 0:
47          cout << "Demo\n";
48          estado un_objetivo;
49          un_objetivo = objetivos.front();
50          cout << "fila: " << un_objetivo.fila << " col:" << un_objetivo.columna << endl;
51          return pathFinding_Profundidad(origen, un_objetivo, plan);
52          break;
53
54      case 1:
55          cout << "Optimo numero de acciones\n";
56          // Incluir aqui la llamada al busqueda en anchura
57          cout << "No implementado aun\n";
58          break;
59      case 2:
60          cout << "Optimo en coste\n";
61          // Incluir aqui la llamada al busqueda de costo uniforme/A*
62          cout << "No implementado aun\n";
63          break;
64      case 3:
65          cout << "Reto 1: Descubrir el mapa\n";
66          // Incluir aqui la llamada al algoritmo de busqueda para el Reto 1
67          cout << "No implementado aun\n";
68          break;
69      case 4:
70          cout << "Reto 2: Maximizar objetivos\n";
71          // Incluir aqui la llamada al algoritmo de busqueda para el Reto 2
72          cout << "No implementado aun\n";
73          break;
74      }
75      return false;
76  }
```

Ilustración 1: 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 el nivel 2 el algoritmo de costo uniforme o el algoritmo A*, y en el los niveles 3 y 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. Cada vez que se aborde la realización de un nivel a partir del nivel 1, aquí se espera que el estudiante coloque las llamadas a los métodos que haya definido que reproduzcan a los algoritmos de búsqueda pedidos.

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 2: 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 3.

```
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 3: Método think. Siguiendo un plan.

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.
- 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 y 2 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 estos niveles hay que ir construyendo el mapa para poder hacer planes o caminos, y con consiguiente es imprescindible escribir en ella. Para poder escribir sobre **mapaResultado** es necesario estar seguro de estar correctamente 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 suspender esta práctica al menos, pudiéndose poner en conocimiento estos hechos a instancias superiores para la toma de medidas disciplinarias más severas.