INTELIGENCIA ARTIFICIAL

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

Tutorial: Práctica 2

Incorporando Procesos Deliberativos

(Los extraños mundos de BelKan II)

DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN E INTELIGENCIA ARTIFICIAL

UNIVERSIDAD DE GRANADA

Curso 2016-2017

1. Introducción

La única pretensión de este tutorial es dar una idea de la forma en la que se puede insertar un comportamiento deliberativo dentro de una arquitectura básicamente reactiva y por consiguiente ser un simple comienzo para que el estudiante desarrolle su práctica, modificando aquellas cosas que crea que son mejorables. Por tanto, es sólo un punto de partida y como tal debe ser tomado.

Vamos a partir como comportamiento base para este tutorial de los ficheros ***jugador.cpp*** y ***jugador.hpp*** resultantes del tutorial de la práctica 1. La descripción de ese comportamiento era bastante simple y se puede resumir de forma muy breve diciendo: el agente sólo se movía sobre casillas de tipo **'S'**, **'T'** o **'K'** que estuvieran desocupadas (es decir, que no tuvieran ningún objeto o personaje en esa posición). El movimiento consistía en avanzan siempre que fuera posible. Si no era posible, entonces una variable de estado (que se generaba de forma aleatoria) determinaba si el giro era a la derecha o a la izquierda. Tenía definida una variable de estado para saber si había pasado por una casilla **'K'**, en cuyo caso, escribía en mapaResultado, la casilla en la que se encuentra el agente en ese momento.

Los pasos que vamos a seguir en este tutorial son los siguientes:

- Describir el fichero **jugador.hpp** para describir las variables de estado y estruturas de datos que se utilizarán en la definición del comportamiento.
- Definición del algoritmo de ***"path finding"***, tanto el prototipo como sus posibles implementaciones.
- Detallar la estructura global que debe tener el método **think()**, no sólo con la separación ya vista en el tutorial anterior entre *"Actualización de la información"* y *"Comportamiento"*, sino también, la separación entre *"Acciones de Movimiento"* y *"Resto de Acciones"* dentro del propio *"Comportamiento"*.
- Implementación de un comportamiento muy básico reactivo/deliberativo.

2. El fichero jugador.hpp

En la siguiente imagen se muestra el fichero "jugador.hpp" que contiene las variables de estado que usaremos para definir el comportamiento del agente.

jugador.hpp

```hpp
#ifndef COMPORTAMIENTOJUGADOR_H
#define COMPORTAMIENTOJUGADOR_H
#include <list>
#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;
      girar_derecha = false;
      bien_situado = false;
      tengo_regalo = false;

      estoy_ejecutando_plan =false;
      error_plan =false;
    }

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

    Action think(Sensores sensores);

    int interact(Action accion, int valor);


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



  private:

    struct estado{
      int fila;
      int columna;
      int orientacion;
    };


    // Declarar Variables de Estado
    int fil, col, brujula;
    Action ultimaAccion;
    bool girar_derecha;
    bool bien_situado;
    bool tengo_regalo;

      // Para el plan
    bool estoy_ejecutando_plan;
    bool error_plan;
    list <Action> plan;


    void Reiniciar();
    bool pathFinding(const estado &origen, const estado &destino, list<Action> &plan);


};
```

Las variables de estado definidas son:

- **fil**, **col**, **brujula**, **ultimaAccion**, **girar_derecha** y **bien_situado** heredadas del comportamiento definido en el tutorial 1.
- **tengo_regalo**, una nueva variable de estado que tomará el valor verdadero cuando esté en posesión de un objeto de tipo regalo, y falso en otro caso.
- **estoy_ejecutando_plan**, indica si he elaborado un camino a seguir o no.
- **error_plan**, es la variable usada en la parte de monitorización del plan y me dice si la siguiente acción del plan se puede o no realizar.

Además, de las anteriores variables de estado, tenemos definida una variable llamada **plan** que almacenará el camino encontrado para trasladar al agente entre dos posiciones del mundo.

También se ha definido el tipo de dato **estado** que es un struct con 3 campos: **fila**, **columna** y **orientacion**. Este tipo de dato se utiliza para determinar el punto de origen y de destino para encontrar el camino en el algoritmo de "path finding".

Se incorpora también los prototipos de las funciones **Reiniciar()** y **pathFinding()**, la primera que reinicia las variables después de recibir una señal de reset y la segunda que implementa el algoritmo de búsqueda.

Por último, en el constructor de la clase, se inicializan todas la variables de estado, tal y como ya se vió en la práctica anterior.

3. El algoritmo de "path finding"

El algoritmo de "path finding" es el encargado de devolver un camino (usando exclusivamente acciones de movimiento) entre dos puntos del mapa. Una posible cabecera para esta función sería la siguiente:

```
bool pathFinding(const estado &origen, const estado &destino, list<Action> &plan);
```

dónde **origen** representa el punto inicial de la búsqueda y **destino** el punto final, y **plan** es el camino resultante. El tipo de dato asociado a los puntos de origen y destino es **estado**. En este tipo de dato, no sólo se refleja la **fila** y la **columna** del punto, sino también la orientación del agente en el momento de pedir el camino, ya que es relevante para construir el plan. En el caso del punto destino, es posible que la orientación no sea relevante, y se puede ignorar.

En el archivo **jugador.cpp** aparece una implementación de un algoritmo de búsqueda de caminos muy simple, y que **no podrá ser usado por los estudiantes como algoritmo de búsqueda final** (aunque sí para hacer las pruebas que crean oportunas). En su lugar, el estudiante deberá implementar uno de los algoritmos de búsqueda vistos en clase en el tema 3 de teoría.

El algoritmo de búsqueda proporcionado devuelve el camino entre dos puntos del mapa resultante de reducir primero las columnas y luego las filas. Este algoritmo no tiene en cuenta si hay obstáculos por el camino, por consiguiente, el camino que devuelve puede ser inviable.

En la implementación que use el estudiante, tendrá que tener en cuenta la viabilidad del camino resultante. El método devuelve un dato ***"bool"*** que debe interpretarse como que existe camino posible entre los dos puntos o no.

4. Estructura global del método think()

En la práctica anterior hablábamos de una separación del método **think()** en dos partes: por una parte, ***"Actualizar Información"*** que permite cambiar los valores de las variables de estado definidas para el problema en función de la última acción que se realizó, y en una segunda parte, al final del método **think()** se describen el conjunto de reglas que definen el comportamiento del agente.

Además, de esa separación, también vamos a considerar una separación en la parte del comportamiento de manera que todos los comportamientos que no están relacionados con acciones de movimiento irán primero, mientras que los relacionados con movimientos irán al final.

En la parte de actualización de la información se debe incluir la información relativa a la monitorización del plan. Esta monitorización implica saber si el plan se va realizando bien y en el caso, de que no sea así, intentar arreglarlo o establecer que es necesario replanificar de nuevo.

5. Un ejemplo de comportamiento

El comportamiento que queremos definir es una extensión del que se propuso en el tutorial 1 siguiendo la siguiente idea:

(a) el agente sólo tomará un único objeto de tipo regalo y cuando lo tenga siempre lo llevará en la mano

(b) no cogerá ningún otro objeto, ni usará la mochila

(c) todos los personajes y objetos distintos del Rey y de los regalos serán considerados como obstáculos, al igual que el agua, la puerta y los árboles.

(d) Si tiene un Rey enfrente entregará el regalo si lo tiene en la mano.

(e) Sólo planificará el camino para llegar al primer regalo de los que aparecen en el sensor con la lista de los regalos.

(f) El movimiento por defecto será el definido en el tutorial 1.

jugador.cpp

```
#include "../Comportamientos_Jugador/jugador.hpp"
#include <iostream>
#include <vector>
#include <queue>
using namespace std;


bool ComportamientoJugador::pathFinding(const estado &origen, const estado &destino, list<Action> &plan){=

void ComportamientoJugador::Reiniciar(){
  fil = col = 99;
  ultimaAccion = actIDLE;
  bien_situado = false;
  brujula = 0;

  tengo_regalo = false;
  estoy_ejecutando_plan =false;
  error_plan =false;

}

bool hayObstaculoDelante(const vector<unsigned char> & terreno, const vector<unsigned char> & superficie){=

void PintaPlan(list<Action> plan){=


Action ComportamientoJugador::think(Sensores sensores){=

int ComportamientoJugador::interact(Action accion, int valor){
  return false;
}
```

La imagen anterior, muestra las funciones que componen el fichero “jugador.cpp”. Podemos ver las siguientes funciones:

- **pathFinding**(): la función para el cálculo del camino, descrita previamente.
- **Reiniciar**(): la función que se invoca cuando muere el personaje. Es semejante a la que se vió en el tutorial 1. La única diferencia es que se inicializan las tres nuevas variables de estado: **tengo_regalo**, **estoy_ejecutando_plan** y **error_plan**.
- **HayObstaculoDelante**(): esta función devuelve si en la casilla de enfrente hay un obstáculo para el agente.
- **PintaPlan**(): es un método para pintar por el terminal el plan.
- **Interact**(): método que debe quedarse como está.
- **think**(): El método que incluye el comportamiento y que pasamos a describir ahora más detenidamente.

jugador.cpp

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

  Action accion = actIDLE;

  if (sensores.reset){
    Reiniciar();
  }

  // Actualización de la información del mundo
  switch (ultimaAccion) {
    case actFORWARD:
      if (!sensores.colision){
        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;
    case actPICKUP:
      if (sensores.objetoActivo == '4'){
        tengo_regalo = true;
        estoy_ejecutando_plan = false;
      }
      break;
    case actGIVE:
      if (sensores.objetoActivo == '_'){
        tengo_regalo = 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;
  }
```

Al principio del **think()** se sitúa la parte de actualización de la información. Como se puede ver en la imagen, se replica la información que ya se proporcionó en la práctica anterior con dos modificaciones.

En el nuevo comportamiento se van a incluir dos nuevas reglas, la primera de ellas hará uso de la acción ***actPICKUP*** y la segunda de la acción ***actGIVE***. Por esa razón, en la parte de actualización, hay que incluir los casos de estas dos nuevas acciones para, por un lado, verificar que la acción se realizó correctamente, y por otro lado, para poder dar el valor correcto a las variables de estado definidas.

En el comportamiento que se va a definir, como se dijo anteriormente, el agente sólo cogerá un objeto de tipo **regalo** e ignorará el resto de objetos. Por eso, las actualizaciones que se proponen, están orientadas a este objeto.

Además, sólo se hará búsqueda de camino para ir hasta un objeto de tipo regalo. Por esa razón, al tomar un objeto de este tipo, se inicializa la variable de estado **estoy_ejecutando_plan** a falso.

Las dos últimas estructuras condicionales hacen referencia a la ubicación del agente al pasar por una casilla amarilla y a escribir en el mapaResultado. Estructuras que vienen heredadas de la versión inicial del comportamiento.

```
197   // Control del plan
198   if (error_plan){
199     estoy_ejecutando_plan = false;
200   }
201
202   // Monitorizacion del plan
203   if (estoy_ejecutando_plan and plan.front() == actFORWARD and hayObstaculoDelante(sensores.terreno, sensores.superfic
204     if (sensores.superficie[2] == 'a' or sensores.superficie[2] == 'l'){
205       plan.push_front(actIDLE);
206       error_plan = false;
207     }
208     else {
209       error_plan=true;
210     }
211   }
212   else {
213     error_plan = false;
214   }
215
216   // Planificacion
217   if (bien_situado and !tengo_regalo and !estoy_ejecutando_plan and sensores.regalos.size()>0){
218     estado origen;
219     origen.fila = fil;
220     origen.columna = col;
221     origen.orientacion = brujula;
222
223     estado destino;
224     destino.fila = sensores.regalos[0].first;
225     destino.columna = sensores.regalos[0].second;
226
227     estoy_ejecutando_plan = pathFinding(origen, destino, plan);
228   }
229
```

En la parte final de la parte de actualización de la información, vienen los elementos relativos al control de la parte deliberativa. La variable de estado **error_plan** es la encargada de indicar si la siguiente acción del plan se puede llevar a cabo. Si tomó en la iteración anterior el valor falso, eso indica que se debe replanificar o cambiar el objetivo. Por esa razón, en la línea 198-199, se anula la ejecución del plan actual.

De las líneas 202-214 se dedica a detectar si el plan se puede seguir ejecutando. Como en este caso, el plan únicamente incluye acciones de movimiento, y la única acción de movimiento que puede producir un error es ***actFORWARD***, la regla de actualización mira si esa acción es viable en este instante. La acción de avanzar puede no ser viable por varias razones: **(a)** en este instante hay un personaje de juego que pasa por esa casilla o **(b)** la casilla contiene un tipo de terreno que en este instante es un obstáculo para el agente. En el caso **(a)**, el plan puede repararse, incluyendo al principio una acción de espera (***actIDLE***) a que se mueva el personaje de esa casilla. Sin embargo, el segundo caso, en principio no tiene un arreglo tan simple, por eso, en este caso, se considera que el plan ha fallado.

De las líneas 216-228 se describe el proceso de planificación del camino. En este caso, la condición para construir un camino es que el agente conozca las coordenadas del mapa original, no esté en posesión de un regalo, no este ejecutando otro plan en este mismo momento y la lista de regalos del sensor no este vacía.

El procedimiento para encontrar el camino es simple, se usan dos variables de tipo **estado**, una para definir el origen (dónde está el robot) y otra para definir el destino (en este caso, donde está el primer regalo de la lista), y se invoca al método pathFinding() que devuelve el plan, y en la variable **estoy_ejecutando_plan** verdadero si el plan es viable.

```
232    // Decidir acciones de no movimiento
233    if (sensores.superficie[2] == '4' and sensores.objetoActivo == '_'){
234      accion = actPICKUP;
235    }
236    else if (sensores.superficie[2] == 'r' and sensores.objetoActivo == '4'){
237      accion = actGIVE;
238    }
```

Entramos en la parte de definición del comportamiento del agente, y como comentamos en la sección anterior, los comportamientos los separaremos en dos partes: primero todos aquellos que no impliquen movimiento (coger objetos, guardar objetos, sacar objetos, dar objetos, ...) y después todos aquellos que impliquen el desplazamiento del agente.

En la imagen anterior, tenemos las dos primeras reglas del comportamiento que son bastante simples de entender. La primera dice que si tengo delante un **regalo** y la mano vacía, entonces que coja el **regalo**. La segundo dice que si tengo un **rey** delante y un **regalo** en la mano, que le de el **regalo** al **rey**. Estas son las dos únicas reglas que voy a considerar que no implican movimiento.

A continuación aparecen las reglas de movimiento. Hay que tener en cuenta que hay dos modalidades de movimiento: la primera marcada por un plan, si este se está ejecutando y la segunda por el que llamaremos movimiento por defecto.

```
239    // Decidir la nueva acción de movimiento
240    else if (estoy_ejecutando_plan and !error_plan){
241      accion = plan.front();
242      plan.erase(plan.begin());
243      PintaPlan(plan);
244      if (plan.size()==0){
245        estoy_ejecutando_plan = false;
246      }
247    }
248    else if ( !hayObstaculoDelante(sensores.terreno, sensores.superficie)){
249      accion = actFORWARD;
250    }
251    else if (!girar_derecha) {
252      accion = actTURN_L;
253    }
254    else {
255      accion = actTURN_R;
256    }
257
```

En este caso, el movimiento por defecto es el que ya se definió en el tutorial 1 y que se muestra desde las líneas 248-256 y es el que menos precedencia tiene. El movimiento motivado por el plan se describe en las líneas 240-247.

El proceso es bien simple, se toma como acción la primera acción del plan, y se elimina dicha acción de la secuencia. Si el tamaño del plan es 0, indica que el plan ya ha terminado.

6. Algunas reflexiones finales

En este tutorial se ha definido un comportamiento muy simple que tiene los elementos básicos para casi por azar entregar un regalo a un rey. La idea es que la versión que realice el estudiante sea capaz de definir un comportamiento que haciendo uso de un algoritmo de “path finding” eficiente actúe de una forma más inteligente.

El estudiante en esta práctica, independientemente de tener que saber implementar un algoritmo de búsqueda para este problema concreto, se enfrenta a algunas decisiones de diseño interesantes como:

(a) ¿Cuándo debo empezar a construir los caminos? Parece claro, que hasta no estar bien orientado, no debería planificar caminos, pero justo después de eso, ¿sería aconsejable empezar a planificar? En nuestra opinión, durante el desarrollo deberíais experimentar con distintas situaciones, ya que posiblemente, sin un conocimiento suficiente del mapa, muchos de los planes pueden terminar siendo no viables.

(b) ¿Uso la búsqueda para todo? Recuerda que tienes un tiempo máximo que no puedes superar. Valora bien el algoritmo que vas a implementar (o incluso implementa varios) para estudiar cual se adecúa mejor al problema en tiempo.

(c) ¿Podría representar el plan usando algo distinto de una lista? Si, sólo tienes que gestionarlo bien para que se codifique correctamente el plan.

(d) ¿Me centro en conseguir misiones o en descubrir el mundo? Deberías balancear correctamente la consecución de ambos objetivos, aunque recuerda que descubrir el mundo tiene una valoración superior.