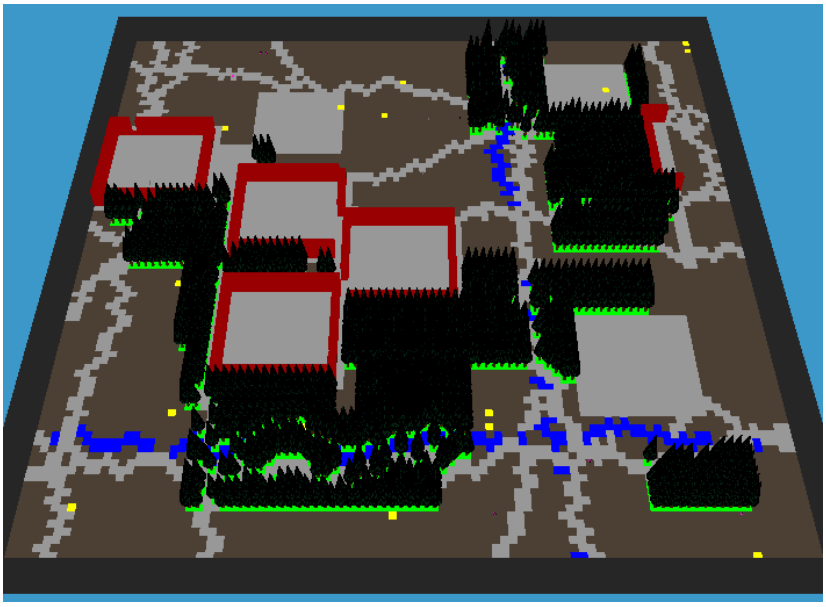
INTELIGENCIA ARTIFICIAL

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. Si seguís este consejo, el proceso de construcción tendría la siguiente secuencia de subjetivos en cada uno de los niveles que se piden:

NIVEL 1:

1. Definir en la función '**PathFinding**()' un algoritmo de búsqueda en espacio de estados de los vistos en clase de teoría. Recomendamos empezar por la búsqueda en anchura, para una vez más avanzada la práctica acometer un algoritmo algo más complejo, pero mejor en eficiencia, si fuese necesario.
2. Definir un comportamiento básico en la función '**think**()', que básicamente llame a la función '**PathFinding**()' para encontrar un plan, y que permita en la siguientes iteraciones ir aplicando las distintas acciones del plan hasta llegar a la casilla objetivo.

NIVEL 2:

1. Modificar el comportamiento anterior para detectar la posibilidad de choque con un aldeano. Si esto ocurre, una posibilidad es volver a construir un nuevo camino llamando a la función '**PathFinding**()' indicándole que la posición donde se encuentra el aldeano que obstruye el paso no es transitable. Por si ocurriera la situación anterior es necesario que el propio sistema reactivo recuerde sus coordenadas en filas, columnas y orientación para poder volver a invocar a '**PathFinding**()'.
2. En este nivel sería un buen momento para proponerse estudiar si es necesario cambiar el algoritmo de búsqueda por otro que pudiera ser más eficiente para este problema y estudiar un poco las ventajas que pudieran ofrecer un algoritmo sobre otro.

NIVEL 3:

Si has llegado aquí, es que ya tienes un algoritmo que te permite planificar de rutas y un comportamiento que te permite ir ejecutando el plan y es capaz de superar los contratiempos que provocan los aldeanos. Todo ese trabajo es directamente aprovechable en este nivel.

A diferencia de los niveles anteriores, no conoces tu posición en el mapa, y sin esa posición es imposible planificar una ruta. Por consiguiente, lo que tienes que hacer es construir un agente reactivo que te permita alcanzar un punto amarillo, los llamados '**PK**' que te dan en los sensores '**mensajeF**' y '**mensajeC**', la fila y la columna en la que te encuentras. Por tanto, un posible esquema a seguir sería:

1. Desarrollar un agente reactivo que basándose exclusivamente en su sistema sensorial te permita alcanzar un punto amarillo y obtener tu posición en el mapa.

2. A partir de tener tu posición, ya si podrías invocar al constructor de rutas pero teniendo en cuenta que hay partes del mapa que desconoces. Por consiguiente, hay dos cosas que hacer: (a) determinar que se hace con las casillas que desconozco del mapa en la función '**PathFinding**()', y (b) ampliar el supuesto de situaciones en las cuales un camino no te lleve a tu destino (por ejemplo, en el camino cuando se planificó se pasaba por una casilla desconocida y cuando te acercas resulta que es un muro)
3. Si resuelves el problema anterior, podrías plantearte un refinamiento del modelo general, para mejorar la capacidad de alcanzar objetivos.

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.

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 todas aquellas casillas que están etiquetadas con valores distintos a la '**S**', '**T**' y '**K**' y que no estén ocupadas por un aldeano. Por consiguiente, vamos a definir un comportamiento donde el agente avanza cuando tengo los valores anteriores y en otro caso, se gira.

```
31  Action ComportamientoJugador::think(Sensores sensores) {
32    if ( (sensores.terreno[2]=='S' or sensores.terreno[2]=='T' or
33          sensores.terreno[2]=='K') and sensores.superficie[2]!='a'){
34      return actFORWARD;
35    }
36    else {
37      return actTURN_R;
38    }
39  }
```

En la siguiente imagen se muestra este comportamiento.

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 guión, 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 sea suelo (en eso se reduce ser '**S**', '**T**' o '**K**') y que no esté ocupada por un aldeano que miramos en el sensor **superficie** el agente avanza. En otro caso, gira a la derecha.

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 un obstáculo 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 los niveles 2 y 3 es que el agente sepa en todo momento su posición correcta dentro del mapa, ya que cuando está en esta situación pueda ejecutarse invocar al 'PathFinding'. En nuestro problema, estar correctamente ubicado consiste en conocer en cada iteración 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.

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 de los niveles 1 y 2 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 siguiente imagen.

```
32  Action ComportamientoJugador::think(Sensores sensores) {
33    // Capturar los valores de fil y col
34    if (sensores.mensajeF != -1){
35      fil = sensores.mensajeF;
36      col = sensores.mensajeC;
37    }
38
39    // Actualizar el efecto de la ultima accion
40    switch (ultimaAccion){
41      case actTURN_R: brujula = (brujula+1)%4; break;
42      case actTURN_L: brujula = (brujula+3)%4; break;
43      case actFORWARD:
44        switch (brujula){
45          case 0: fil--; break;
46          case 1: col++; break;
47          case 2: fil++; break;
48          case 3: col--; break;
49        }
50        cout << "fil: " << fil << "  col: " << col << " Or: " << brujula << endl;
51    }
52
53    // Sistema de movimiento
54    Action sigAccion;
55    if ( (sensores.terreno[2]=='S' or sensores.terreno[2]=='T' or
56          sensores.terreno[2]=='K') and sensores.superficie[2]!='a'){
57        sigAccion = actFORWARD;
58    }
59    else {
60      sigAccion = actTURN_R;
61    }
62
63    ultimaAccion = sigAccion;
64    return sigAccion;
65  }
```

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 33 a la línea 37 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 40 a la línea 51 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 54 a la 61, 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 definái 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 los niveles 1 y 2 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 niveles 1 y 2 y la rellena del carácter **'?'** en el nivel 3.

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 3.

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:

```
Action ComportamientoJugador::think(Sensores sensores) {
  // Capturar los valores de fil y col
  if (sensores.mensajeF != -1){
    fil = sensores.mensajeF;
    col = sensores.mensajeC;
  }

  // Actualizar el efecto de la ultima accion
  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;
      }
      cout << "fil: " << fil << "  col: " << col << " Or: " << brujula << endl;
  }

  // Sistema de movimiento
  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;
  }

  Action sigAccion;
  if ( (contenidoCasilla=='S' or contenidoCasilla=='T' or
        contenidoCasilla=='K') and sensores.superficie[2]!='a'){
      sigAccion = actFORWARD;
  }
  else {
    sigAccion = actTURN_R;
  }

  ultimaAccion = sigAccion;
  return sigAccion;
}
```

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 y asumimos que ya se ha implementado un algoritmo de búsqueda en la función **PathFinding**.

Podéis bajar de la zona de descargas de material de la asignatura en "**decsai**" dentro de la carpeta "**Para la presentación**" el archivo "**Tutorial paso 2**". Si se descomprime el archivo, aparecen los ficheros "**jugador.cpp**" y "**jugador.hpp**". Debeis sustituir estos ficheros por los que tienen el mismo nombre en el software de la práctica 2 colgando de la carpeta "**Comportamientos_Jugador**".

En este caso, la función **think** se encarga 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.

En el fichero "jugador.cpp" en la función think podemos distinguir 6 bloques:

1. Capturar la fila y la columna en la que se encuentra el agente.
2. Actualización de **fil** y **col** en función de la última acción aplicada.
3. Detección del cambio de la casilla objetivo.
4. Generación de un camino de la posición actual del agente a la casilla objetivo.
5. Control del plan.
6. Actualizar y enviar la acción elegida.

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.

```
136   // Determinar si ha cambiado el destino desde la ultima planificacion
137   if (hayPlan and (sensores.destinoF != destino.fila or sensores.destinoC != destino.columna)){
138     cout << "El destino ha cambiado\n";
139     hayPlan = false;
140   }
```

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.

```
142   // Determinar si tengo que construir un plan
143   if (!hayPlan){
144     estado origen;
145     origen.fila = fil;
146     origen.columna = col;
147     origen.orientacion = brujula;
148
149     destino.fila = sensores.destinoF;
150     destino.columna = sensores.destinoC;
151
152     hayPlan = pathFinding(origen,destino,plan);
153   }
```

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 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.

```
156   // Ejecutar el plan
157   Action sigAccion;
158   if (hayPlan and plan.size()>0){
159     sigAccion = plan.front();
160     plan.erase(plan.begin());
161   }
162   else {
163     sigAccion = actIDLE;
164   }
```

Un último comentario aunque muy importante. La que se ha implementado en la función **PathFinding** para este tutorial es una función de búsqueda que **NO SE PUEDE USAR EN LA**

VERSIÓN QUE SE ENTREGUE. Se recuerda que tiene que ser uno de los algoritmos de búsqueda en espacio de estados que se enseña en la parte teórica de la asignatura.

Si os fijáis en la última parte de la función PathFinding podéis ver la siguiente invocación:

```
108     // Descomentar para ver el plan en el mapa
109     VisualizaPlan(origen, plan);
110
111     return true;
112   }
```

La función VisualizaPlan tiene dos argumentos, origen que indica la posición dada al algoritmo de búsqueda de donde debe empezar el camino y plan, la secuencia de acciones obtenida. Esta función hace que se pinte sobre el mapa del entorno gráfico el plan. Obviamente, esta función no es obligatorio usarla, pero puede ayudaros para ver los planes que construye vuestra implementación del algoritmo. Si tenéis errores en tiempo de ejecución relacionadas con la confección del camino, podéis comentar esta función para facilitar la depuración de dichos errores.

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 1 y 2 contiene el mapa completo y se usa en el algoritmo de búsqueda para encontrar el camino. En el nivel 3 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 3 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.

El comportamiento deliberativo (el algoritmo de búsqueda) que se define en este tutorial no se considera como un comportamiento entregable como solución a la práctica. Así que aquellos que tengan la intención de entregar esto como su práctica o un comportamiento con ligeras variaciones de este, les recomendaría que no lo hagan, por qué será considerada como una práctica copiada e implicará suspender la asignatura.

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.