UNIVERSIDAD DE GRANADA

Técnicas de los Sistemas Inteligentes
grado en ingeniería informática

Práctica 1

Desarrollo de un agente basado en búsqueda

Autor
José María Sánchez Guerrero

Rama
Computación y Sistemas Inteligentes || Grupo 2 - Pablo Mesejo

Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación

Curso 2019-2020

Índice

1. Introducción **2**

2. Deliberativo **2**
2.1. Deliberativo simple 4
2.2. Deliberativo compuesto 5

3. Reactivo **6**
3.1. Reactivo simple 6
3.2. Reactivo compuesto 6

4. Deliberativo-Reactivo **7**

1. Introducción

La práctica consiste en implementar varios agentes que se puedan desenvolver adecuadamenete en los distintos escenarios propuestos. Dispondremos de 5 mapas distintos, cada uno de ellos con un escenario distinto.

- En el primero simplemente tendremos que encontrar la salida
- En el segundo hay que recoger 10 gemas y luego salir
- En el tercero aguantar 2000 ticks del juego sin morir con un enemigo
- El cuarto aguantar 2000 ticks con varios enemigos
- El último será una fusión entre todos los otros, ya que hay que recoger 10 gemas y salir sin ser atrapado por ningun enemigo.

En mi caso he utilizado un algoritmo IDA* para encontrar los caminos más óptimos, y una técnica reactiva que consiste en mantener al avatar lo más alejado posible de los enemigos en cada tick.

2. Deliberativo

Para nuestros agentes deliberativos vamos a utilizar, como hemos mencionado antes, el **algoritmo IDA***. He elegido este algoritmo porque está basado en el algoritmo base A* (es una extensión de este), sin embargo, he decidido implementar IDA ya que obtiene los mismos resultados, pero no necesita almacenar todos y cada uno de los posibles nodos candidatos. La gran ventaja que nos ofrece esto, es que reduce considerablemente el consumo de memoria.

No obstante, pese a tener un consumo bajo en memoria, se podría haber implementado un algoritmo más eficiente en cuanto a tiempo, sobre todo si tenemos en cuenta que disponemos 40 milisegundos por cada *tick*. En caso de que queramos calcular una ruta en este periodo de tiempo, puede ser que nos de error. En mi caso, lo he dejado así porque los mapas son bastante pequeños y no muy complejos, por lo que no debería de tener ningún problema; sin embargo, para mapas más difíciles sí que habría que mejorar su eficiencia.

A la hora de implementarlo, he necesitado dos clases. La primera es la clase "***IDAStar.java***", en la cual está implementado todo el algoritmo; y otra llamada "***Node.java***", la cual utilizamos para representar de una forma más sencilla cada

una de las casillas del tablero adaptadas a nuestro algoritmo. A continuación vamos a detallar cómo lo hemos hecho.

Empecemos con la clase ***Node*** ya que la usaremos dentro de la otra. Lo primero es la declaración de variables a utilizar:

```
1     // Valor heuristico del nodo actual
2     private double hScore = Double.MIN_VALUE;
3     // Coste del camino recorrido
4     private double gScore = 0.0;
5     // La suma de los valores anteriores
6     private double fScore = 0.0;
7
8     // Padre del nodo actual
9     private Node parent;
10    // Posicion en coordenadas (x,y) del nodo actual
11    private Vector2d position;
```

Como es un algoritmo basado en el A*, tenemos que poner las 3 variables características de éste: $f() = g() + h()$. Junto a ellas, también hay que poner la posición en coordenadas (x, y) del nodo actual, y otra variable para el padre, es decir, la posición de la cual se ha generado.

He añadido un constructor para poder crear un nuevo nodo a partir de una variable *Vector2d*, que representa una posición concreta en el mapa. Para acceder y modificar todas estas variables tendremos sus correspondientes métodos *getX()* y *setX(value)*.

Implementaremos las siguientes funciones que nos podrán ser útiles en determinadas situaciones. Para comparar un nodo con otro, es decir, comparar los valores de *f()*, he añadido una función que devuelve -1, 1 o 0 dependiendo de si es menor, mayor o igual al nodo a comparar, respectivamente. Tenemos otra función que nos dirá si un nodo es igual a otro (no en cuanto al valor de *f()*, sino al estado de los nodos). Para obtener los nodos sucesores del nodo actual, la cual inserta en una lista los nodos correspondientes a las casillas que rodean a la actual, comprobando antes que esta casilla sea transitable, es decir, no sea un obstaculo. Por último, tenemos la función que determina la *h()*. Como estamos en un mapa cuadriculado, la distancia al padre siempre va a ser 1.

Una vez hemos terminado con la clase *Node*, vamos a pasar a la clase ***IDAStar***. En ella tenemos las siguientes variables:

```
1     // Creamos las variables para los nodos
2
3     // Nodo inicial desde donde saldra nuestro avatar
4     private Node initialState;
5     // Nodo objetivo para el camino
6     private Node goalState;
```

```
7     // Posiciones de los obstaculos del mapa
8     private ArrayList<Vector2d> tiposObs;
```

Aqui también tenemos un constructor para que nos genere un camino a partir del nodo inicial, otro final y una lista con las posiciones de los objetos por las que no puede pasar nuestro avatar.

La primera función más relevante de esta clase es ***search()***, en la cual comienza la búsqueda. En ella realizaremos una búsqueda recursiva (que detallaremos posteriormente), hasta que la cota hallada sea 0, o lo que es lo mismo, hasta que hayamos llegado al objetivo. Devuelve un *Node* que es el último nodo de la ruta óptima. Devolverá nulo si no se puede encontrar el nodo objetivo.

Ahora vamos a pasar a la función ***recursive_search()***, la cual es la que maneja todo el algoritmo y encuentra el camino óptimo desde dentro de la función *search()*. Ésta busca recursivamente los hijos de los nodos, evitando buscar el camino hacia abajo con una f más alta que el límite f actual. Si se encuentran caminos con un limite más alto, devolverá la f más pequeña sobre el límite encontrado. Este límite sobre el f mas pequeño es un nuevo límite f potencial durante la próxima iteración. La función devolverá 0 si se encuentra el nodo objetivo e Integer.MAX_VALUE si no se puede encontrar el nodo objetivo.

Por último, tenemos una función ***getPath()*** que devuelve una lista de nodos que representa el camino óptimo. Como hemos mencionado, la función *search()* sólo devuleve el nodo final, asi que esta función se encargará de pasar por todos los padres del último nodo e insertalos en la lista, hasta llegar al nulo.

Con esto ya hemos terminado de explicar las clases y funciones auxiliares que utilizaremos para los modelos deliberativos, asi que pasemos a ver el funcionamiento de ambos.

2.1. Deliberativo simple

Antes de ejecutar el *act* (en este y en todos los agentes), se inicializan las variables correspondientes al factor de escala, posición del avatar, portal, objetos, muros, etc...; y en el caso del deliberativo, también inicializamos y calculamos el path. Esto lo hacemos aquí, porque tenemos más margen de tiempo que en el *act*, donde sólo tenemos 40 milisegundos para tomar una decisión.

La ventaja que tenemos en este agente deliberativo simple, es que no es necesario recalcular ningún path, por lo que el obtenido en un principio es que nos servirá hasta el final. Finalmente, lo único que tendremos que hacer en el *act* es obtener el primer elemento del camino calculado (siguiente posición del avatar), calcular el

siguiente movimiento a partir de él y devolver su acción correspondiente.

2.2. Deliberativo compuesto

3. Reactivo

3.1. Reactivo simple

3.2. Reactivo compuesto

4. Deliberativo-Reactivo