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

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*. Este algoritmo está basado en el algoritmo base A* (es una extensión de este), sin embargo, he decidido implementar este ya que obtiene los mismos resultados, pero no almacena todos los posibles nodos que puede visitar y reduce considerablemente el consumo de memoria.

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 sus implementaciones.

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;
```

He añadido un constructor para poder crear un nuevo nodo a partir de una posición (x,y):

```
1     /**
2    * Constructor a partir de una posicion (x,y) del mapa.
3    *
4    * @param position
5    */
6   Node(Vector2d position){
7     this.position = position;
8   }
```

Para acceder y modificar estas variables tendremos sus correspondientes métodos *getX* y *setX*. Para compara un nodo con otro, es decir, comparar los valores de *f()*, he añadido la siguiente función:

```
1     /**
2      * Compara el valor de f() del nodo actual con otro.
3      *
4      * @param n Nodo a comparar.
5      * @return -1 si el actual es menor que n,
6      *          0 si actual es igual que n,
7      *          1 si actual es mayor que n.
8      */
9     @Override
10    public int compareTo(Node n) {
11        if(this.fScore < n.fScore)
12            return -1;
13        if(this.fScore > n.fScore)
14            return 1;
15        return 0;
16    }
```

También dispondremos de una función que determina si un nodo es igual que otro, pero no en cuanto a sus valores de *f()*, si no en cuanto a sus estados:

```
1     /**
2      * Determina si otro nodo es igual al nodo actual. Dos nodos son
3      * iguales si sus estados son iguales (no f ()).
4      *
```

```
     * @param otro nodo Nodo a comparar.
     * @return true si el otro nodo es igual, false en caso contrario.
     */
    @Override
    public boolean equals(Object o)
    {
        return this.position.equals(((Node)o).position);
    }
```

Para obtener los nodos sucesores he utilizado la siguiente función:

```
    /**
     * Genera una lista de los nodos sucesores al nodo actual.
     * Devolvemos cada una de las casillas que rodean a la actual,
     * comprobando antes que esta casilla sea transitable, es
     * decir, no sea un obstaculo.
     *
     * @param tiposObs son las casillas no transitables del mapa.
     * @return lista de nodos sucesores.
     */
    public List<Node> getSuccessors(ArrayList<Vector2d> tiposObs){

        // Declaramos una lista para los nodos
        List<Node> successors = new ArrayList<Node>();

        // Insertamos en ella las casillas que rodean a la actual
        Node top = new Node(new Vector2d(this.position.x,
                                         this.position.y-1));
        top.setParent(this);
        Node bottom = new Node(new Vector2d(position.x, position.y+1));
        bottom.setParent(this);
        Node left = new Node(new Vector2d(position.x-1, position.y));
        left.setParent(this);
        Node right = new Node(new Vector2d(position.x+1, position.y));
        right.setParent(this);

        // Comprobamos que no son casillas no transitables
        if (!tiposObs.contains( top.position )) {
            successors.add(top);
        }
        if (!tiposObs.contains( bottom.position )) {
            successors.add(bottom);
        }
        if (!tiposObs.contains( left.position )) {
            successors.add(left);
        }
        if (!tiposObs.contains( right.position )) {
            successors.add(right);
        }

        return successors;
    }
```

Por último, tenemos la función que determina la *h()*. Como estamos en un mapa cuadriculado, la distancia al padre siempre va a ser 1, por tanto:

```
1     /**
2      * Devuelve la distancia entre el nodo actual y el padre.
3      *
4      * @return 1, porque estamos en un mapa cuadriculado.
5      */
6     public double distFromParent() {
7       return 1;
8     }
```

Una vez hemos terminado con la clase *Node*, vamos a pasar a la clase ***IDAStar***

2.1. Deliberativo simple

2.2. Deliberativo compuesto

3. Reactivo

3.1. Reactivo simple

3.2. Reactivo compuesto

4. Deliberativo-Reactivo