Técnicas de los Sistemas Inteligentes

Práctica 1: Técnicas de Búsqueda Heurística

Rafael Vázquez Conejo

Marzo 2020

Índice

1. Descripción General.

Comenzaré explicando por encima los métodos y variables que he implementado para obtener las diferentes soluciones en mi *Agent.java*.

Las variables que encontramos:

- boolean estoyEscapando → será true si el avatar tiene enemigos cerca y debe huir.
- boolean voy_a_gemas → será true si no estoy escapando de un enemigo y por lo tanto puedo ir a por gemas.
- boolean hay_escopriones → será true cuando al observar el estado del mapa de juego hay algún escorpión, lo que nos permite saber que tenemos que tenerlos en cuenta.

Respecto a las funciones que he definido las iré tratando conforme sean empleadas en cada comportamiento, pero comenzaremos con el método que decide la siguiente decisión del avatar, ya que este método será llamado en ambos comportamientos. Este método definido al final de *Agent.java* es "`act(StateObservation estado, ElapsedCpuTimer elapsedTimer)`" y las acciones más importantes que realiza por orden son:

1. Obtengo la posición del avatar en el mapa de juego.
2. Si hay un plan activado, elimino la última acción realizada por el avatar.
3. Calcular un camino si el avatar no tiene activado un comportamiento reactivo calculo un camino y no hay un plan definido.
4. Si hay un escorpión cerca activo el comportamiento reactivo y mis siguientes acciones serán para alejarme de este enemigo. En caso de que hubiera un camino calculado lo elimino y lo volveré a recalcular cuando no haya enemigos cerca.
5. Si no hay enemigos cerca y hay un plan de camino tomaré la acción que corresponde para seguir el camino, es decir si el avatar debe ir a la derecha tomaré la acción de ir a la derecha.
6. Si no hay enemigos cerca ni plan el agente no hace nada.

También he implementado mi propia clase Observation.java que me permite saber el objeto que se encuentra en cada casilla del mapa. Solamente he definido los elementos que me interesan para esta práctica, es decir las casillas tipo Tierra por ejemplo nos la he definido porque en nuestro caso no tenemos que interactuar con ellas.

2. Comportamiento Deliberativo.

El comportamiento deliberativo se utiliza para buscar las gemas y el portal.

Para calcular estos caminos desde una posición inicial a una posición final he implementado las clases *Node.java*, *PathFinder.java* y *A_estrella.java*. Para su creación me he inspirado en las clases definidas en “tools.pathfinder” existentes en el propio archivo GVGAI, las cuales son demasiado generales, por ello he creado mis propias clases directamente relacionadas con el problema que deseamos solucionar.

- Node.java → Esta clase contiene la representación de un nodo, el cual tendrá un coste total (*totalCost*), un coste estimado (*estimatedCost*), un nodo padre (*parent*) y una posición en el mapa (*position*). Esta clase nos permitirá comparar los nodos para ver cual posee menor coste respecto a otro. Además, gracias a la función “equals”, podremos saber si un nodo es el que deseamos, es decir, podremos comprobar si nuestro nodo actual es igual al nodo meta, por ejemplo.

- A_estrella.java → Esta clase contiene la implementación del algoritmo de búsqueda A*. La heurística utilizada es la distancia Manhattan. El algoritmo recibirá por parámetros el nodo incoo, el nodo al que queremos llegar (nodo meta), y el modo de calculo, este último parámetro que puede tomar valores 0 ó 1, haciendo referencia al modo en el que se generan los vecinos, modos que hemos implementado en nuestra clase PathFinder.java de los que hablaremos después.

  Tanto la lista de nodos sin explorar (Abiertos), cómo la de ya explorados (Cerrados) la hemos creado mediante colas de prioridad del tipo *Node* como estructura de datos.

- PathFinder.java → Esta clase contiene los diferentes modos de generar vecinos mencionados anteriormente. Encontramos 2 modos:

  - Modo 0, “getPortal”: este modo se utiliza cuando debemos calcular el camino hacia el portal, es utilizado tanto en el caso de no tener gemas que recoger e ir directos al portal, como cuando hemos recogido las 10 gemas y debemos buscar el portal. Este modo no genera cómo vecinos los nodos que sean gemas o muros.

  - Modo 1, “getGemas”: este modo es utilizado para calcular el camino de nuestro avatar hasta una gema. En este caso no se generan como vecinos los nodos muro.

Los modos 0 y 1 son muy similares, la principal razón por la que los he creado por separado es debido a la situación en la que se emplean. Si se calcula el camino hacia el portal significa que el avatar ya tiene las gemas o que en ese mapa no hay, caso del deliberativo simple, y si es el primer caso de tener todas las gemas no es recomendable que el avatar coja una gema camino al portal, puesto que esto ya no sería necesario ni recomendable.

He usado el algoritmo A* utilizando como heurística la distancia Manhattan que nos proporciona la distancia mínima entre dos coordenadas. Posee la siguiente expresión:

$$distancia = \sum |x - y|$$

a) Comportamiento Deliberativo Simple.

Caso en el que no hay enemigos ni gemas en el mapa, el agente debe encontrar el camino hasta el portal. Para este caso he implementado el método *calcularNuevoCamino*, el cual recibe como parámetros la posición del agente en el mapa del juego, el número de gemas que posee el agente y el estado del mapa.

Este método nos permite calcular un camino desde el avatar hasta una gema o un portal, en este caso será hasta un portal. Para ello:

1. Obtenemos la posición de las gemas en el mapa
2. Si ese valor de posición es igual a 10 (caso del reactivo-deliberativo) o es nulo,porque no hay gemas (nuestro caso actual) obtendremos la posición del portal y aplicaremos el algoritmo A* para encontrar el camino que nos llegará desde nuestra posición actual a la posición del portal.
3. Devolvemos ese camino, tras ello las acciones a realizar para llegar al portal se irán decidiendo en la función "*act*" definida en el primer apartado.

Definiré a continuación el funcionamiento de mi algoritmo A*
Este algoritmo recibe como parámetros el nodo inicio, el nodo meta al que deseamos llegar y el modo de calculo explicado anteriormente.

```
FUNCIÓN A* (inicio, meta, modoCalculo):
        nodo ←   Ø
        Inicializo las PriorityQueue de ambas listas y el nodo inicial.
        lista_abiertos.add(inicio);
        MIENTRAS tamaño de la lista_abiertos != 0
                nodo ← lista_abiertos.poll()            //obtengo el primer nodo
                lista_cerrados.add(nodo)

                //compruebo si he llegado a la meta
                SI nodo.position.equals(meta.position)
                        Devuelvo calcularCamino(nodo)
                FIN-SI

                //genero los vecinos de mi nodo actual según el modo
                Inicializo vecinos              //Arraylist del tipo Node
                SWITCH(modoCalculo)
                        CASE 0:
                                vecinos ← getPortal(nodo)
                        CASE 1:
                                vecinos ← getGemas(nodo)
                FIN-SWITCH

                PARA cada vecino de nodo
                        nodo_vecino = vecinos.get(vecino)
                        distancia_actual = nodo_vecino.totalCost
                        SI lista_abiertos no contiene nodo_vecino &&
                                                        lista_cerrados no contiene nodo_vecino
                                //f(n) = g(n) + h(n)
                                nodo_vecino.totalCost =   distancia_actual + nodo.totalCost
                                nodo_vecino.parent = nodo
                                lista_abiertos.add(nodo_vecino)
                        FIN-SI
```

```
                    SINO-SI distancia_actual + nodo.totalCost < nodo_vecino.totalCost
                        //f(n) = g(n) + h(n)
                        nodo_vecino.totalCost =   distancia_actual + nodo.totalCost
                        nodo_vecino.parent = nodo
                        SI lista_abiertos contiene nodo_vecino
                            eliminar nodo_vecino de lista_abiertos
                        FIN-SI
                        SI lista_cerrados contiene nodo_vecino
                            eliminar nodo_vecino de lista_cerrados
                        FIN-SI
                        lista_abiertos.add(nodo_vecino)
                    FIN-SINO-SI
                FIN-PARA
            FIN-MIENTRAS
            SI nodo.position.equals(meta.position)
                Devolver calcularCamino(nodo)
            FIN-SI
            SINO
                Devolver null;
            FIN-SINO
FIN
```

La función utilizada para devolver la solución, "*calcularCamino*" nos devuelve un vector del tipo Node con nuestro camino, obteniendo los nodos padre de nuestros nodos escogidos.

Observamos que en mi algoritmo para cada nodo que exploramos generamos sus vecinos, esto lo realizo gracias a dos método implementas en mi clase PathFinder, "getPortal" y "getGemas".

En este caso de comportamiento deliberativo simple llamados a la función getPortal, esta función explora todos los vecinos de mi nodo actual dándole un valor a la componente "x" e "y" a cada nodo dependiendo del tipo de casilla que sea y añadiendo dichos valores a mi vector de vecinos. En este caso le daremos valor a aquellas casillas vacías, a la casilla Portal y a los escorpiones, las demás casillas nos son indiferentes a la hora de obtener nuestro camino.

He decidido esta implementación porque de esta forma directamente no tenemos en cuenta las casillas de muros evitándolas en todo momento, lo cual nos reduce el número de cálculos.

b) Comportamiento Deliberativo Compuesto.

Caso en el que no hay enemigos pero si gemas en el mapa, necesitando 10 de estas gemas antes de salir por el portal. El agente debe encontrar el camino hasta las diferentes gemas, según su cercanía a estas, y tras ello deberá encontrar el camino hasta el portal.

Este comportamiento se implementa en el mismo método anterior, "*nuevoCamino*" pero en esta ocasión si hemos obtenido posiciones de gemas en el mapa por lo tanto nos saltamos la primera condición hasta que hayamos obtenido 10 gemas y por lo tanto podamos ir al portal de la misma forma explicada anteriormente.

1. Busco las posiciones de las gemas gracias a la función getResourcesPositions().
2. Comienzo por la gema más cercana y recorro todas ellas aplicando mi algoritmo A* desde la posición de mi agente hasta la gema que estoy evaluando.
3. De esta forma obtengo un camino hasta la gema, y si no lo encuentro exploro otra gema.

Ejecutamos el mismo algoritmo A*, pero en este caso estamos en el modo 1, "*getGemas"*.

Este modo funciona similar al anterior, vuelvo a generar los vecinos, dándole valor a aquellos vecinos que sean del tipo de casillas que me interesen. En este caso me interesan las casillas vacías, las gemas y los escorpiones. No genero como vecino los muros, así evito ir a por una gema que esté rodeada de muro y sea inaccesible.

Tras conseguir todas mis gemas, las 10, se realizará lo explicado en el apartado anterior del comportamiento deliberativo simple, es decir se volverá a ejecutar el algoritmo A* pero se generaran los vecino con "*getPortal*" esta vez. Por este motivo he decidido implementar estas dos soluciones en el mismo método ya que en este segundo caso una vez termina la búsqueda de gemas realiza la búsqueda del portal, base del comportamiento deliberativo simple.

Mencionar que las gemas que el agente va obteniendo las voy almacenando en una variable en el método "*act*" denominada "*n_gemas*" la cual actualizo con el método *getAvatarResources()*.

2. Comportamiento Reactivo.

El comportamiento reactivo se utiliza cuando nuestro agente tiene enemigos cerca y debe huir de ellos. En nuestro ejercicio esos enemigos serán los escorpiones.

Este comportamiento se maneja mediante la pila de acciones, en la cual dependiendo de lo que quiera hacer el agente se almacenarán unas acciones u otras que el agente irá realizando hasta que la pila quede vacía. Definida en el siguiente atributo:

Stack <Types.ACTIONS> acciones

a) Comportamiento Reactivo Simple.

Las decisiones que tomaré para saber si tengo un enemigo cerca y las acciones que tendré que realizar para huir de él están definidas en "Agent.java". El método clave que llamará a todas las demás funciones que establecerán las acciones de huida es "*boolean escorpionCerca*", este método recibe como parámetros el estado del mapa de juego y la posición de mi agente en este, y devuelve true si hay un enemigo en las casillas que evalúa.

Este método comprueba si hay enemigos cercanos al agente en un rango de casillas que mostraré a continuación. Dependiendo de dónde estén los enemigos el agente huirá a un determinado lado usando una de las funciones de escape definidas, las cuales trataremos más adelante.

El rango de casillas que compruebo es el siguiente:

(x-2, y-2)	(x-1, y-2)	(x, y-2)	(x+1, y-2)	(x+2, y-2)
(x-2, y-1)	(x-1, y-1)	(x, y-1)	(x+1, y-1)	(x+2, y-1)
(x-2, y)	(x-1, y)	A	(x+1, y)	(x+2, y)
(x-2, y+1)	(x-1, y+1)	(x, y+1)	(x+1, y+1)	(x+2, y+1)
(x-2, y+2)	(x-1, y+2)	(x, y+2)	(x+1, y+2)	(x+2, y+2)

Siendo A la posición actual de mi agente exploro las 24 posiciones mostradas. Además en cada posición dependiendo de la posición del agente en el mapa ejecutaré un método de huida u otro. Esto lo hago principalmente para evitar ir directamente a las esquinas. Supongamos que el agente se encuentra en una posición de y <= 6 teniendo "y" un valor máximo de 12, si el enemigo se acerca por la derecha y no podemos huir a la izquierda, porque hay un muro por ejemplo, la siguiente opción de huida será ir por la casilla de abajo y no la de arriba para evitar acercarnos a una esquina del mapa. De la misma forma si la "y" es un valor superior a 6 y tenemos que huir hacia la izquierda y no podemos, nuestra siguiente opción será huir por la casilla de arriba si está libre, de nuevo para evitar ir a una esquina del mapa. De la misma forma ocurrirá si el enemigo viene por la izquierda.

De la misma forma ocurre con la componente "x". Si el agente está en una posición x <= 15 siendo la "x" máxima 25 y un enemigo viene por abajo, nuestra primera opción será huir hacia arriba pero si esto no es posible la siguiente opción será huir a la derecha para evitar ir al borde. De la misma forma si nuestra "x" tiene un valor superior a 15 y no podemos huir hacia arriba de un enemigo que nos viene desde abajo, nuestra siguiente opción será huir por la izquierda. Lo mismo ocurrirá en el caso de que un enemigo nos venga por arriba.

Las diferentes funciones definidas para la huida dependiendo de por dónde se aproxime el enemigo reciben como parámetros el estado del mapa y la posición del jugador en éste y almacenarán en la pila de acciones las acciones más convenientes.

PeligroDerechaArriba()

Se utiliza cuando “y” es mayor que 6 y el enemigo se encuentra en una de las siguientes casillas:

(x+1, y), (x+2, y), (x+1, y+1), (x+2, y+2), (x+2, y+1)

Se sigue el siguiente razonamiento:

- Añadir acciones para ir hacia la izquierda si la casilla de la izquierda está libre.
- En caso contrario, compruebo en la casilla de arriba.
- En caso contrario, compruebo en la casilla de abajo.
- En caso contrario, espero y deshago el plan.

PeligroDerechaAbajo()

Se utiliza cuando “y” es <= 6 y el enemigo se encuentra en una de las siguientes casillas:

(x+1, y), (x+2, y), (x+1, y-1), (x+2, y-2), (x+2, y-1)

Se sigue el siguiente razonamiento:

- Añadir acciones para ir hacia la izquierda si la casilla de la izquierda está libre.
- En caso contrario, compruebo en la casilla de abajo.
- En caso contrario, compruebo en la casilla de arriba.
- En caso contrario, espero y deshago el plan.

PeligroIzquierdaArriba()

Se utiliza cuando “y” es mayor que 6 y el enemigo se encuentra en una de las siguientes casillas:

(x-1, y), (x-2, y), (x-1, y+1), (x-2, y+2), (x-2, y+1)

Se sigue el siguiente razonamiento:

- Añadir acciones para ir hacia la derecha si la casilla de la derecha está libre.
- En caso contrario, compruebo en la casilla de arriba.
- En caso contrario, compruebo en la casilla de abajo.
- En caso contrario, espero y deshago el plan.

PeligroIzquierdaAbajo()

Se utiliza cuando “y” es <= 6 y el enemigo se encuentra en una de las siguientes casillas:

(x-1, y), (x-2, y), (x-1, y-1), (x-2, y-2), (x-2, y-1)

Se sigue el siguiente razonamiento:

- Añadir acciones para ir hacia la derecha si la casilla de la derecha está libre.
- En caso contrario, compruebo en la casilla de abajo.
- En caso contrario, compruebo en la casilla de arriba.
- En caso contrario, espero y deshago el plan.

PeligroAbajoIzquierda()

Se utiliza cuando “x” es mayor que 15 y el enemigo se encuentra en una de las siguientes casillas: (x, y+1), (x, y+2), (x+1, y+2)

Se sigue el siguiente razonamiento:

- Añadir acciones para ir hacia arriba si la casilla de arriba está libre.
- En caso contrario, compruebo en la casilla de la izquierda.
- En caso contrario, compruebo en la casilla de la derecha.
- En caso contrario, espero y deshago el plan.

PeligroAbajoDerecha()

Se utiliza cuando “x” es <= que 15 y el enemigo se encuentra en una de las siguientes casillas: (x, y+1), (x, y+2), (x-1, y+2)

Se sigue el siguiente razonamiento:

- Añadir acciones para ir hacia arriba si la casilla de arriba está libre.
- En caso contrario, compruebo en la casilla de la derecha.
- En caso contrario, compruebo en la casilla de la izquierda.
- En caso contrario, espero y deshago el plan.

PeligroArribaIzquierda()

Se utiliza cuando “x” es mayor que 15 y el enemigo se encuentra en una de las siguientes casillas: (x, y-1), (x, y-2), (x+1, y-2)

Se sigue el siguiente razonamiento:

- Añadir acciones para ir hacia abajo si la casilla de abajo está libre.
- En caso contrario, compruebo en la casilla de la izquierda.
- En caso contrario, compruebo en la casilla de la derecha.
- En caso contrario, espero y deshago el plan.

PeligroArribaDerecha()

Se utiliza cuando “x” es <= que 15 y el enemigo se encuentra en una de las siguientes casillas: (x, y-1), (x, y-2), (x-1, y-2)

Se sigue el siguiente razonamiento:

- Añadir acciones para ir hacia abajo si la casilla de abajo está libre.
- En caso contrario, compruebo en la casilla de la derecha.
- En caso contrario, compruebo en la casilla de la izquierda.
- En caso contrario, espero y deshago el plan.

b) Comportamiento Reactivo Compuesto

La resolución de este caso en el que en lugar de un escorpión hay dos es la misma que en el comportamiento reactivo simple esto es posible debido a que los métodos anteriores devuelven las acciones de la casilla más cercana al personaje, es decir si hubiera dos enemigos dentro del rango de casillas que evalúo tendría prioridad las acciones para huir de aquel enemigo que se encuentre más cerca de mí, siempre teniendo en cuenta la existencia de otro enemigo.

Estos comportamientos reactivos serán llamados en la función "*act*" en la cual se ejecutará al método "*escorpionesCerca()*" para comprobar si se debe ejecutar el comportamiento reactivo y no se desarrollará ningún comportamiento deliberativo ya que para ello debe haber un plan, es decir un camino establecido, cosa que no tenemos en este caso ya que el portal es inaccesible y no hay gemas.

He ejecutado este comportamiento 10 veces en el nivel 8, nivel en el que hay 2 enemigos, y de esas 10 ejecuciones el agente ha aguantado todo el tiempo en 8 de ellas y ha muerto en 2. Las causas de su muerte en ambos casos ha sido porque ambos escorpiones se han acercado a la vez al agente y al huir del más cercano ha acabado acercándose al otro. He probado a intentar mejorar heurística alejándome de los muros, de esa forma tendría más espacio de acción, pero el resultado me ha sido peor en algunos casos, por lo que he decidido implementar esta forma en su lugar.

4. Comportamiento Reactivo-Deliberativo.

En este caso nos encontramos en la situación de recoger las 10 gemas y tras ello irnos por el portal, pero esta vez hay un enemigo que debemos evitar. Esta combinación entre mis comportamientos anteriormente definidos la he implementado en el método "*act*".

Este método ya definido al principio, ya que es en él donde establezco todas las acciones que realiza el agente, comenzará en este caso con un comportamiento deliberativo, obteniendo el camino hasta la gema más cercana. Tras ello llamaré a la función "*escorpionCerca()*", la cual si nos devuelve true será porque debemos huir del escorpión por lo que debemos cancelar nuestro camino hasta la gema y centrarnos en huir, para ello le asigno a mi variable "*voy_a_gemas"* el valor false. Nos encontramos entonces ante el caso de tener que huir del enemigo, para ello extraigo de la pila la acción a realizar, asigno a mi variable "*estoyEscapando*" un valor true y en caso de haber un camino establecido lo elimino.

Una vez hemos escapado del enemigo la variable "*estoyEscapando"* volverá a ser false, lo que nos permitirá volver a diseñar un plan hacia las gemas o el portal, si ya tenemos las 10. De nuevo comprobaremos si hay un enemigo cerca, si ahora estamos ante el caso de que no lo hay "*voy_a_gemas"* será igual a true y por lo tanto hay un camino, es decir un plan y podemos seguirlo, ejecutando gracias a nuestro algoritmo A* las acciones para seguir el camino hacia nuestra meta actual. De nuevo este camino podrá ser cancelado si hay un enemigo cerca, y tras huir de este se generará uno nuevo.