Memoria Práctica 1

Álvaro Beltrán

April 4, 2020

Grupo 3, Jose Ángel Segura Muros

1 Introducción

En esta práctica 1 se nos pide, realizar los comportamientos deliberativo, reactivo y reactivo-deliberativo de un agente para jugar al juego Boulder Dash de GVGAI con algunas facilidades escritas en la descripción de la práctica.

2 Comportamiento Deliberativo

Para el comportamiento deliberativo voy a dividir la explicación en tres secciones, como llegar al portal de forma óptima usando un algoritmo A^* , como generar el orden de gemas a recorrer y como integrarlo en el método act de Agent.

2.1 Algoritmo A^*

Para llegar al portal he implementado un algoritmo A^* común con una nueva clase nodo que ahora pasaré a explicar. Para la heurística de cada nodo sumo los pasos que llevamos recorridos más la distancia manhattan hasta el portal. Para facilitar la implementación he creado una nueva clase denominada PathFinding donde se encuentra este algoritmo.

La clase **Nodo** se compone de :

- Un vector de acciones llamado recorrido, donde almaceno todas las acciones que he hecho hasta llegar a ese estado.
- Un objeto estado donde almaceno la posición y la orientación actual del personaje. Uso este objeto para actualizar el estado del nuevo nodo en el constructor del mismo.
- Dos enteros h y f donde almaceno la distancia manhattan al portal y el número de acciones hechas hasta el momento.
- Un real llamado id, Que es el identificador del nodo el cual genero de esta forma:

$$posicion.x * 1000000 + posicion.y * 1000 + orientacion.x * 10 + orientacion.y$$

La única diferencia a la hora de la implementación del algoritmo A^* es que cuando el nodo hijo generado es un muro añadimos este a nodos cerrados. Para nodos abiertos he usado una cola con prioridad definiendo un comparador sobre los enteros f y h del nodo. Y para nodos cerrados he creado un HashMap cuya clave es el identificador ya explicado anteriormente y cuyo valor es el propio nodo.

2.2 Orden de gemas a recorrer

Para este problema he creado una clase en el archivo Agent llamada OrdenGemas que será la encargada de generar la solución al problema.

He implementado un algoritmo parecido al Simulate Annealing (que pasaré a explicar más adelante) usando en algunas partes el algoritmo A^* creado en la sección anterior. He decidido usar este algoritmo por que tiene cierto parecido con los algoritmos genéticos que me parecen de gran interés y por que se ha demostrado su eficiencia con problemas como el viajante de comercio el cual se parece mucho a nuestro problema.

He enfocado el problema no para que devuelva el path que el personaje a de seguir, si no para devolver el orden de las gemas que el personaje debe conseguir.

Algoritmo

En este algoritmo necesitamos una solución, a priori aleatoria, para comenzar a mejorarla durante un número de iteraciones determinado. Para ello primero voy a describir la clase OrdenGemas creada.

La clase **OrdenGemas** se compone de:

- Una lista de observaciones llamada gemas donde almaceno las gemas por orden de cercanía al personaje y cuyo índice dentro de la lista será su identificador de la gema (el cual usaremos para dar el orden de gemas a recorrer)
- Dos listas de enteros llamadas solucion y solucionPrev donde almacenaremos la solución actual y la solución anterior que teníamos.
- Dos listas de enteros llamadas *gemas_descartadas* y *gemas_descartadas_prev* donde almacenaremos los enteros de las gemas que no hemos cogido en al solución (ya que la solución se compone de 10 gemas.)
- Dos enteros distancia y distanciaPrev donde almacenamos la distancia al recorrer la solución y la solución previa respectivamente.
- Un HashMap llamado cache, donde almacenaremos las distancias entre las posiciones (gema-gema,gema-portal,personaje-gema). La clave de este cache es:

$$id = pos.x * 100000000 + pos.y * 1000000 + ((ori.x + 3)\%3) * 100000 \\ +((ori.y + 3)\%3) * 10000 + posHacia.x * 100 + posHacia.y$$

  pos es la posición donde estamos y ori es la orienteción donde estamos, mientras que posHacia es la posición a donde queremos ir. Con esta clave conseguimos determinar de forma unívoca la distancias entre dos casillas. Lo que guardamos en la caché es la distancia entre esas dos casillas y la orientación con la que llegamos a esa casilla para así poder calcular la distancia a otra casilla posteriormente.

Una vez definidos los elementos de la clase podemos meternos de lleno en el algoritmo, para **inicializar** el algoritmo necesitamos una solución previa la cual se me ocurrió generar la solución greedy pero generar esta solución invertía demasiado tiempo para lo

mediocre que era. Por eso decidí comenzar con una solución aleatoria.

Esta inicialización del algoritmo la realizamos en el constructor de la clase, para ello, primero creo un vector auxiliar con todos los enteros respectivos a las gemas (ej: si hay doce gemas, un vector [0‖1‖...‖11]), miro si todas las gemas son accesibles desde el avatar y si alguna no lo es, esta gema se se elimina del vector creado. Una vez sabemos que en nuestro vector todas las gemas son accesibles, hacemos un shuffle y quitamos las gemas sobrantes (solo puede haber 10) añadiéndolas en *gemas_descartadas*. Tras esto generamos la solución añadiendo un -1 (avatar), luego el vector auxiliar y luego añado -2 (portal).

Una vez generada la solución tenemos que calcular cuanta distancia consume recorrer esta solución. Para ello vamos simulando con un método de la clase el recorrido de la solución. Para el cálculo de la distancia entre las gemas usamos el algoritmo A^* , de forma que si la distancias entre las gemas ya la hemos generado al calcular la distancia de otra solución, esta estará en la caché y si no se añadirá.

En este momento ya tenemos inicializado el objeto para comenzar con el algoritmo que he llamado en el objeto SimulateAnealing, el cual pasaré a explicar poco a poco:

```
1  public void simulateAnnealing(ElapsedCpuTimer elapsedTimer){
2          while(elapsedTimer.remainingTimeMillis()>20){
3              swapGemas();
4              if(distancia<=distanciaPrev)
5                  continue;
6              else
7                  revertSwap();
8          }
9
10 }
```

Este algoritmo es la primera versión que creé el cual voy a explicar y posteriormente diré que falla, para explicar la versión final que añade un pequeño cambio.

Vemos dos funciones importantes, swapGemas() y revertSwap(). swapGemas() lo que hace es escoger dos posiciones aleatorias del vector de solución distintas de la posición del personaje y del portal. Intercambiarlas por la gema de la otra posición o por una gema de las gemas descartadas (elección aleatoria). También calcula la nueva distancia y guarda la solución y distancia anteriores.

revertSwap() lo que hace simplemente es revertir el cambio colocando como solución y distancias las anteriores.

De esta forma consigo que se realice esto mientras haya tiempo en el constructor.

El problema que me encontré en este algoritmo es el estancamiento de las soluciones, aunque consigue soluciones generalmente buenas, otras veces no consigue salir de una solución mala, ya que cambiar solo parejas cambia poco la solución. Pero solucionar este problema es fácil así:

```
public void simulateAnnealing(ElapsedCpuTimer elapsedTimer){
        while(elapsedTimer.remainingTimeMillis()>20){
            swapGemas();
            if(distancia<=distanciaPrev)
                continue;
            else
                this.revertSwap();
                shuffle();
                if(distancia<=distanciaPrev)
                    continue;
                else
                    this.revertSwap();
        }
}
```

Con la función shuffle() hacemos por decirlo así un swap aleatorio en toda la solución (hacemos shuffle sobre la lista de solución).

De esta forma hacemos que las soluciones buenas se generen con mayor probabilidad guardándolas y haciendo cambios sobre ellas. La eficacia de este algoritmo radica en que al ejecutarlo en el constructor de agent llegamos a hacer hasta 3,000,000 de iteraciones ayudándonos para ello de la caché de distancias.

2.3 Método Act de Agent

He realizado el algoritmo para las gemas en el constructor de agente para disponer de más tiempo. Luego en el Método de Act con la variable booleana creada para saber si estoy en el nivel 1 y 2 hago un conjunto de secuencias para recorrer el orden de las gemas en caso de haberlo o si no ir al portal.

3 Reactivo

Para el comportamiento reactivo del agente he creado un método de la clase Agent llamado evitar enemigo que llamo siempre que no tengo decisión, el método modifica el plan a seguir por el personaje.
En el método compruebo si hay enemigos a una cierta distancia, si no los hay, me muevo de forma aleatoria intentando no quedarme en las esquinas. Si hay enemigos, busco en las casillas de mi alrededor cual está más lejos de todos los enemigos. Si no hay casillas posibles elimino el enemigo más lejano y vuelvo a buscar, así hasta que no hay enemigos. Si ya no hay enemigos y no he encontrado casilla posible me quedo quieto. En el caso

de que halla casillas posibles doy prioridad a si estoy mirando hacia la posición que dice, si no escojo una aleatoria.
Con esto encuentro un problema cuando me encierran en una esquina y aleatoriamente van directos a por el personaje.

4 Reactivo-Deliberativo

En el comportamiento reactivo-deliberativo he implementado un método para saber si tengo enemigos cerca. Si tengo enemigos cerca aplico el método que creé para el comportamiento reactivo. En caso de no tener enemigos cerca voy en dirección de la gema que me dice el orden de gemas que ya he definido con el constructor de Agente, cuando cojo una gema por que haya pasado por allí, también la contabilizo. Haciendo esto repetidamente, cuando consigo 10 gemas voy para el portal.
En este comportamiento no he encontrado problemas, pues después de depurar tanto ambos por separado ya sabía que debía funcionar.

5 Conclusión

Me ha parecido una práctica muy interesante en la que puedes intentar superarte a ti mismo haciendo mejores algoritmos. Lo que más me ha gustado ha sido el comportamiento deliberativo donde he podido dar rienda suelta a mi imaginación para hacer un algoritmo competente. Pero, a mi parecer, está práctica necesita de mucho tiempo para hacerla con cierto rigor y ganas.