UNIVERSIDAD DE GRANADA

Técnicas y Sistemas Inteligentes

Práctica 1

Desarrollo de un agente basado en búsqueda heurística para el entorno GVGAI

Autor: Francisco Javier Bolívar Expósito Grupo: 3 Curso: 2019-2020

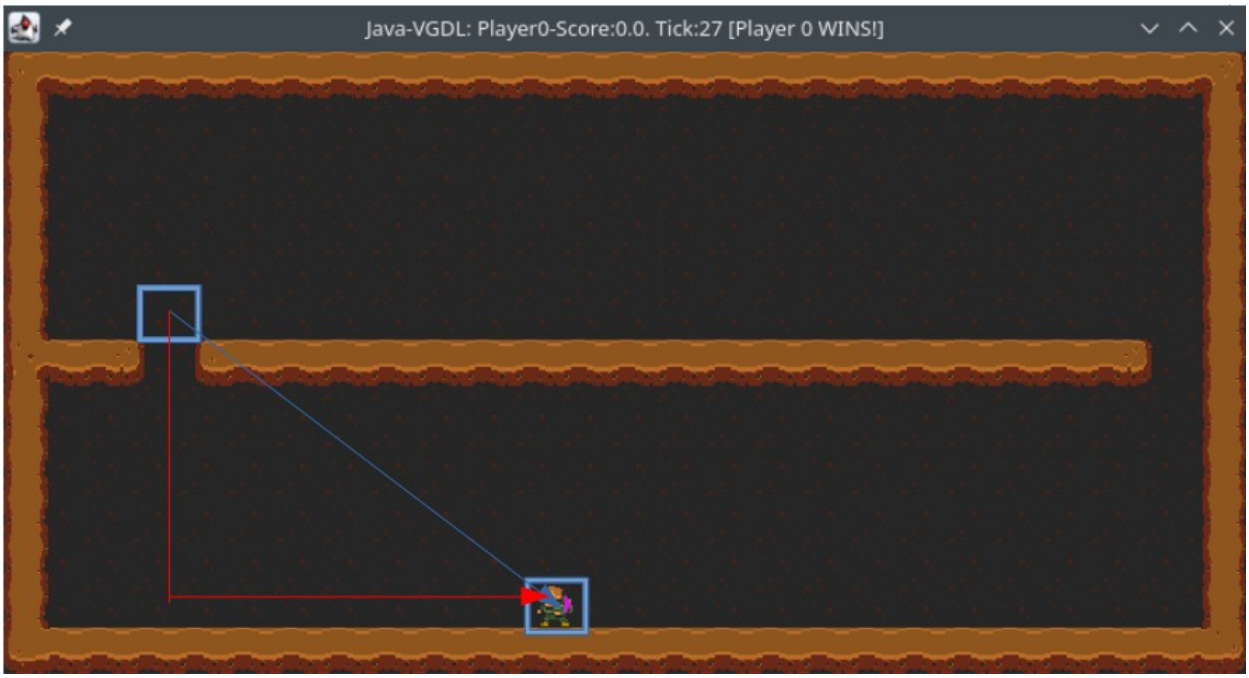
1. Comportamiento Deliberativo Simple

Para simplificar los métodos y el cálculo de distancias entre dos casillas todas las coordenadas devueltas en píxeles se escalan antes de pasarse a los métodos a las coordenadas del “grid”, es decir, a las coordenadas que indican una casilla del mapa, haciendo uso de una función ‘scale’.

Para implementar la búsqueda del camino óptimo entre una casilla inicio y una casilla destino se ha implementado el algoritmo A* siguiendo la siguiente estructura:

```
1. Introducimos el nodo inicial en una lista de abiertos que queda ordenada con
los nodos más prometedores de explorar.

2. Mientras la lista de abiertos no esté vacía

      2.1 Sacamos el primer nodo de abiertos y lo guardamos en el nodo actual.

      2.2 Si el nodo actual es solución devolvemos la lista de acciones que nos ha
      llevado hasta el.

2.3 Para cada acción disponible generamos un nodo hijo con el resultado de aplicar
esa acción, si este nodo no ha sido explorado ya lo añadimos a la lista de
abiertos.
```

La heurística considerada es la **distancia Manhattan**, en la que la distancia entre dos puntos es la suma de las diferencias (absolutas) de sus coordenadas. Se ha considerado esta una buena heurística, más adaptada al problema que por ejemplo la usual distancia Euclídea, ya que las entidades se desplazan por el mapa moviéndose a casillas adyacentes.



Comparación distancia entre dos casillas. Distancia Euclídea en Azul, Manhattan en Rojo

Se han implementado dos clases adicionales útiles para el A*.

‘State’ que representa un **estado del avatar** en el mundo. Dado por su Vector2d posición que indica sus coordenadas en el mapa y su orientación.

‘Node’ que representa un **nodo** en el espacio de búsqueda del A*. Compuesto de un ‘State’, el coste actual g, el coste estimado f, el nodo padre y la última acción realizada para llegar al nodo.

La lista de abiertos **se mantiene ordenada por el coste total f**, es decir el coste para llegar hasta el nodo + el coste heurístico, se ha implementado la interfaz compareTo en la clase Nodo para que se ordenen automáticamente de menor a mayor coste.

La lista de cerrados se ha implementado como un HashSet, implementando en Node la función HashCode. De esta forma podemos comprobar si un nodo ya se ha explorado y está en la lista de cerrados en un tiempo constante usando tablas hash.

Para devolver la lista de acciones que ha llevado hasta la solución se ha implementado una función en Node que reconstruye esta lista. Para esto hasta llegar a un nodo que no tenga padre se añade la última acción realiza a la lista y se cambia al nodo padre.

Finalmente comentar como se ha decidido implementar la generación de los hijos. Se ha implementado en State una función para **simular la consecuencia de una acción**, teniendo en cuenta que es necesario **girar** para estar orientado correctamente y poder moverse en una dirección, parte necesaria para encontrar el camino óptimo.

Además se usa un HashSet con las coordenadas de los obstáculos para devolver un estado nulo si la acción llevaría a chocarse con un obstáculo, **evitando** así **generar hijos que intenten atravesar paredes**. El HashSet de posiciones de paredes se inicializa en el constructor pudiendo conocer así rápidamente si en una posición hay un obstáculo sin malgastar tiempo de computación.

Con esta implementación del A* el comportamiento deliberativo simple que se ejecutará en cada tick es el siguiente:

- Si el plan (lista de acciones a realizar) está vacío lo creamos con el resultado de una búsqueda A* desde el avatar hasta el portal.
- Si quedan acciones en el plan sacamos la primera y la realizamos.

De esta forma llegamos al portal en el nivel proporcionado 5 en **27 ticks**, obteniendo el camino óptimo.

2. Comportamiento Deliberativo Complejo

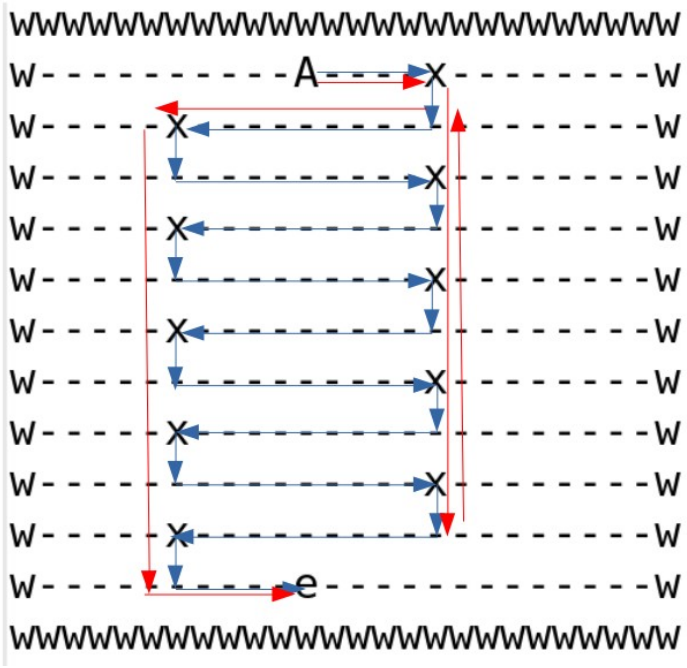
El principal problema de este comportamiento es escoger 10 gemas a recoger y en que orden hacerlo, una vez hecho esto podemos simplemente aplicar el A* para recoger cada gema.

Primero implementé una heurística que simplemente recogía la **gema más cercana** al avatar, los resultados eran muy pobres, recogiendo gemas por un camino no óptimo y/o alejándose del portal que es el destino final. Aun así esta primera aproximación era una forma gráfica para entender mejor que factores influían en escoger un buen camino y así desarrollar una buena heurística.

Partiendo de solucionar el problema de alejarse del portal se consideró coger la gema con menor **distancia al avatar + distancia al portal**. De esta forma las gemas seleccionadas se encontraban en un camino hacia el portal. Sin embargo, el orden de recogida de gemas era mucho peor, empezando por ejemplo por una gema muy próxima al portal obviando todas las que se podían recoger en el camino y después volviendo a la otra punta del mapa a por otra gema.

Viendo que esta heurística proporcionaba una buena selección de gemas pero no un buen orden se decidió separar el problema en 2 partes, **seleccionar las gemas y decidir el orden en el que recogerlas.**

Es interesante recoger las gemas próximas al portal las últimas, para acabar así en una posición cercana a este, por lo tanto se probó como **heurística para el orden** maximizar la **distancia al portal**. Los resultados eran mejores, pero no en todos los casos merece la pena tener de criterio solo la distancia al portal (ejemplo en azul a continuación).



Orden de recogida.
Azul: max. DistPortal Rojo: min. distAvatar/distPortal

Finalmente las heurísticas escogidas son: **seleccionar las 10 gemas con menor (distancia al avatar + distancia al portal)**, y recogerlas en el orden dado de menor a mayor por **(distancia al avatar / distancia al portal)**. Repitiendo la decisión tras coger cada gema, esta heurística proporciona resultados significativamente mejores sin requerir de un gran tiempo de ejecución.

Inicialmente se calcularon las distancias con la distancia Manhattan, pero se decidió calcularlas como la cantidad de acciones que devuelve un A* entre los dos puntos. De esta manera obtenemos una distancia más precisa **al tener en cuenta los giros necesarios y si hay o no obstáculos en medio**.

Para mejorar el tiempo de ejecución las distancias de todas las gemas al portal se

precalculan en el constructor. Al cambiar el estado del avatar en cada turno no es tan fácil precalcular las distancias desde este a todas las gemas, no se ha considerado necesario ya que con esta implementación los tiempos son muy bajos.

Si nos quedan gemas por recoger utilizamos la heurística para seleccionar la siguiente gema y utilizamos A* para ir hasta ella, en caso contrario vamos al portal. De esta forma se incluye este comportamiento a la práctica.

El problema de buscar el camino más corto pasando por todos los nodos, con un inicio y final definido, es un problema NP-Completo. Teniendo en cuenta que en el nivel final es necesario replanificar se ha elegido esta estrategia que da una buena solución aunque no sea la óptima. Podría haber sido interesante analizar si con una cantidad reducida de gemas la solución óptima era obtenible en los 40ms límite y combinar ambas estrategias según el número de gemas.

3. Comportamiento Reactivo Simple / Complejo

El comportamiento reactivo implementado es válido para superar ambos niveles. Aunque algunas mejoras solo fueron necesarias e introducidas para poder superar el comportamiento complejo.

Se ha utilizado la técnica de los **mapas de calor** sugerida en la presentación de la práctica. Estar **pegado** a **las paredes limita** el número de movimientos disponibles para **escapar**, por lo tanto se ha decidido que las paredes también generen “calor” o “peligro” alrededor suya.

Los valores del mapa de calor que generan las paredes se han calculado y almacenado en una matriz en el constructor del programa, para los monstruos en vez de actualizar todo el mapa cada vez que se muevan consultamos su distancia a la casilla sobre la que queremos saber el peligro y con este valor la calculamos.

La **distancia de los escorpiones** se calculó inicialmente con la distancia Manhattan, pero igual que en el comportamiento deliberativo complejo, esto no tiene en cuenta los giros y los obstáculos. Como el A* imitaba el comportamiento de un avatar que tenía que girar antes de moverse no simulaba bien la distancia que haría un monstruo. Para conseguir una distancia más precisa se ha añadido una opción al A* para **simular el comportamiento de un escorpión**, añadiendo esta opción también a la función de simular el resultado de una acción de la clase State.

Podemos obtener el peligro de una casilla con la función getHotness, que devuelve el máximo entre el peligro de una casilla por estar cerca de una pared y el que genera cada escorpión en esa casilla usando el A* adaptado.

Finalmente la decisión de que acción tomar se hace de la siguiente manera:

Para las 4 casillas al lado del avatar, y su posición actual, seleccionamos las que tienen **menos peligro** en el mapa de calor.

De estas casillas seleccionadas priorizamos a las que podemos llegar instantáneamente, por la necesidad de inmediatez que tiene huir de los monstruos. La máxima prioridad está en **movernos hacia la casilla hacia la que se está orientado**, seguido por quedarnos en la casilla actual. Si no tenemos ninguna de estas opciones se seleccionará una opción aleatoria de las casillas seleccionadas restantes.

Se han realizado 1500 ejecuciones de los dos niveles que consisten es esquivar escorpiones, desactivando el entorno gráfico y cambiando la semilla sumándole el número de iteración. Después se ha guardado la salida en un fichero y se han contado el número de muertes con ‘grep -c “:-1” FILE’.

En el nivel de 1 escorpión se ha obtenido un 97,8% de victorias.

En el nivel de 2 escorpiones se ha obtenido un 88,46% de victorias.

4. Comportamiento reactivo-deliberativo

Este último comportamiento integra el comportamiento reactivo y deliberativo implementados en los niveles anteriores, con algunos ligeros cambios al actuar para combinarlos y así poder obtener los objetivos a la vez que evitamos a los escorpiones.

Se ha definido un estado de estar o no en peligro. Por cada turno en el que el agente se encuentra en **peligro ejecutamos el comportamiento reactivo** para alejarnos del escorpión rápidamente, si había un **plan de acciones calculado lo eliminamos**, ya que ya no es válido. Mientras **no esté en peligro** recogerá las gemas normalmente con el **comportamiento deliberativo**.

El agente entra en el estado de peligro si algún escorpión se encuentra a una distancia (calculada con el A* que explora los movimientos de un escorpión) del avatar menor o igual a la definida por el umbral ‘enter_danger_threshold’ (inicializado a 3).

Para definir cuando el agente sale del estado de peligro se ha definido otro umbral mayor, para favorecer ligeramente que se elija una gema objetivo que no este bloqueada por el escorpión al estar más alejados de este, escapar de forma más segura y no entrar/salir constantemente del estado. Este umbral está definido en ‘exit_danger_threshold’ (inicializado a 4).

Tras 3000 ejecuciones en el lvl9 (monstruo y gemas) se ha obtenido un 99,17% de victorias.