Técnicas de los Sistemas Inteligentes

Universidad de Granada

Práctica 1

Desarrollo de un agente basado en búsqueda heurística
para el entorno GVG-AI

Ignacio Vellido Expósito
Grupo Lunes

Descripción general

Nota: Para el funcionamiento correcto del agente es necesario incluir los ficheros adjuntados, especialmente *Observation.java* y *ObservationType.java*. Estos archivos se han modificado para incluir el estado *ObservationType.DANGER* (para marcar una casilla como peligrosa) y para incluir el método *clone* en *Observation*.

Esta modificación se ha utilizado para evitar el uso de los métodos derivados de *StateObservation*, ya que entre su extrema lentitud en la ejecución (en especial la del método *advance*) y la poca cantidad de tiempo de cálculo complicaban enormemente la práctica.

Para evitar el uso de *advance* se ha añadido una variable *mapa* al agente que simula el estado del mundo.

Act

El método primeramente actualiza algunas variables del agente y comprueba si se encuentra en una situación de peligro. Si lo está entrará en la componente reactiva y si no en la deliberativa.

Tras haber decidido una acción usando alguna de las dos componentes, el agente comprueba si el movimiento hará caer una roca sobre él (estando la roca a distancia Manhattan de 2) para que en el siguiente turno se oriente en la dirección correcta y así evitar que le caiga encima.

Agente

El agente cuenta con variables para:

- Conocer el estado actual del juego: *estado_actual, el_mapa*
- Tratar con gemas: *numGemas, old_numGemas, gemas_accesibles, gema_objetivo*
- Recordar la salida, pues nunca cambia en la partida: *exit*
- Saber el efecto de sus acciones: *rocaCayendo*
- Evitar caer en bucles: *path_continuado*
- Calcular un path: *path, hay_solucion, reiniciarPath, abiertos, cerrados, mejorPadre, valorG, valor, origen*

Se incluye además la clase *Nodo* que almacena un estado del juego para su uso en *pathFinding*.

Comportamiento reactivo

Principalmente comprueba el número de enemigos en sus inmediaciones y en caso positivo realiza un movimiento para evitar el peligro. También puede entrar en este estado si hay una roca cayendo sobre él tanto por causa suya como ajena.

Funciones:

tierraCavadaEntre	Estando el enemigo a distancia Manhattan de 2, dice si la casilla entre él y el agente es accesible para el enemigo. Esto nos permite evitar entrar en comportamiento reactivo cuando no estamos afectados por el peligro.
getEnemigos	Devuelve en un mismo array las Observation de todos los enemigos.
enemigoCerca	Dada una casilla, dice si hay algún enemigo a una distancia menor o igual a la indicada.

evitarPeligro	Se llama cuando sabemos que el agente está en peligro. El método elige la mejor acción para el agente en base a la posición de los enemigos y su orientación.
comprobarPeligro	Comprueba si al agente le está cayendo una roca encima y si los enemigos están a una distancia Manhattan menor o igual que 2.
comprobarEnemigo	Comprueba si algún enemigo está a una distancia Manhattan menor o igual que 2.

Comportamiento deliberativo

Sabiendo que el agente no está en peligro, este calcula un camino hacia el mejor objetivo. En caso de que no termine de calcular el path por falta de tiempo, mandará *ACTION_*NIL y seguirá en el siguiente turno (en caso de que no haya aparecido peligro).

No se fomenta la velocidad de respuesta sigo la confianza en un camino sin peligro, se espera que A* consiga a través de la heurística hacer la búsqueda lo más rápida posible.

Funciones:

esTransitable	Solo devuelve *true* si la casilla es *EMPTY*, *GROUND* o *DANGER* y no se encuentra bajo una roca.
bajoRoca	Dice si la casilla está bajo una roca.
gemaBajoRoca	Dice si la gema se encuentra bajo una roca.
actualizarTrasUSE	Cambia un mapa dado suponiendo que se cava en la casilla indicada. Marca como *DANGER* todas las casillas bajo la roca que sean *EMPTY*, y como *EMPTY* el tantas de casillas que sean rocas como las marcadas como *DANGER*.
generarSucesores	Genera los hijos de un nodo suponiendo todas las acciones posibles excepto *USE* si no se cava bajo una roca.

Estrategia de búsqueda

Se sigue un algoritmo A* optimizado e interrumpible, es decir, puede realizar la búsqueda durante varios ticks. Su pseudocódigo se indica más adelante.

Funciones:

cloneList	Devuelve el contenido de un ArrayList en otro.
recuperarPath	Guarda en la variable *path* la sucesión de acciones hasta el objetivo
pathFinding	Implementación del algoritmo A*

Heurística

Se parte de la distancia Manhattan hacia el objetivo añadiendo una penalización por peligro.

Funciones:

calculaH	Distancia Manhattan con penalización si la casilla tiene un enemigo cerca.
calculaG	Coste 1 a no ser que la casilla esté marcada como peligrosa, que en cuyo caso sería 10.

Componente reactiva

Entramos en esta componente cuando detectamos que la seguridad del agente se compromete, es decir, cuando se encuentra en una situación en la que el siguiente movimiento puede determinar su muerte.

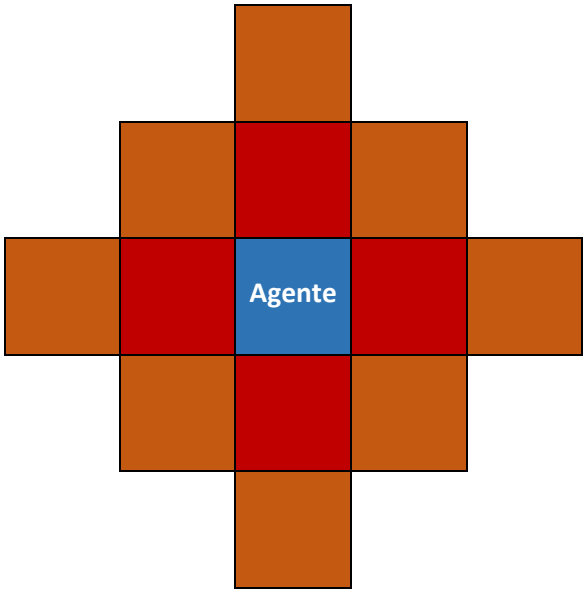
```
componenteReactiva() {  // Se activa si hay enemigo cerca
                        // o roca cayendo
  action <- evitarPeligro()

  Recordamos que hay que modificar el path
}
```

Este estado de peligro se activa de la siguiente manera:

Por una parte, si lo que nos afecta es una roca cayendo (se encuentra en la casilla superior), nos movemos en cualquier dirección posible.

Por otro lado, en el caso de que sea un enemigo, este estará situado a una distancia Manhattan de 2 o 1, dejándonos con las posibilidades de la figura. Según la posición exacta de este estaremos interesados en una acción u otra.



Las casillas en rojo son en las que si no hacemos ningún movimiento nuestra supervivencia depende de la acción del enemigo. Las marcadas en naranja indican que el agente puede moverse por error a una casilla también accesible por un enemigo.

Ambos tipos de casillas nos ponen en peligro, con las rojas vemos que es importante moverse en cualquier dirección con tal de aumentar la distancia con el enemigo, y por tanto el agente intentará tener que cambiar de dirección.

Si el enemigo se encuentra en una naranja es posible que el agente no entre en estado de peligro si la casilla que los separa no es accesible por el enemigo (por ejemplo, *WALL* o GROUND).

En caso de que no tengamos ningún movimiento posible, no hacemos nada con la esperanza de que sobrevivamos este turno.

La detección de peligro interrumpirá el path y la búsqueda, obligando a volver a llamar a *pathFinding* en el siguiente tick en el que no se detecte peligro. Los motivos son claros, es bastante probable que el peligro nos haya forzado a movernos a una casilla que no se encontraba en la ruta, por lo que debemos volver a calcularla.

Componente deliberativa

Siempre que la componente reactiva no se activa se utilizará la deliberativa. Esta se basa en lo siguiente:

- Si tenemos un path calculado y no acabado, continuamos con él.
- En caso contrario, podemos:
    - Calcular el path hacia la salida.
    - Elegir una gema y calcular el path hacia ella.

Si el agente no tiene tiempo suficiente para encontrar el camino solución hacia el objetivo, parará la ejecución y continuará en el siguiente turno (suponiendo que no se activa la componente reactiva).

La selección de una gema se basa en lo siguiente:

Se parte de un vector de posibles gemas, inicialmente incluyendo el conjunto completo. Antes de llamar a *pathFinding*, el agente calcula cuál de ellas es la más cercana (con la condición de que no esté en una casilla peligrosa y no esté bajo una roca, en cuyo caso fomenta coger otra) y la elige como casilla objetivo.

En el caso de que esa gema no se pueda alcanzar en este momento, la gema se elimina del vector y se elige una nueva en la siguiente iteración. Este vector también se actualiza al coger una nueva gema, eliminando la gema encontrada.

Para evitar la pérdida de tiempo, si el agente pasa mucho tiempo intentando coger la misma gema (10 turnos), se le indica que busque otra.

```
componenteDeliverativa() {
  if (path vacío) {
    if (se ha continuado el path más de 10 veces) {
      Quitamos gema actual de las accesibles
      reiniciarPath = true
    }

    if (reiniciarPath == true) {
      if (hemos cogido todas las gemas)
        Llamamos pathFinding() hacia la salida
      else {
        Reiniciamos la lista de gemas accesibles si hemos alcanzado
        alguna o la lista está vacía

        Calculamos la gema objetivo entre las accesibles

        Llamamos pathFinding() hacia la gema
      }
    }
    else {  // Debemos continuar calculando el path
      Llamamos pathFinding() hacia la salida o a la gema objetivo
      según el número de gemas que tengamos
    }

    if (no se ha encontrado solución en pathFinding) {
      action <- NIL

      if (no debemos continuar el path en el siguiente turno)
        Quitamos gema actual de las accesibles
    }
    else
      action <- Siguiente acción del path
  }
  else
    action <- Siguiente acción del path

  if (se ha cavado la cailla superior)
    Recordar que hay una roca cayendo

  return action
}
```

Estrategia de búsqueda

Se ha utilizado un algoritmo A* optimizado en la que para *Abiertos* se utiliza una estructura PriorityQueue y para *Cerrados* un HashSet. Estas estructuras nos dan más velocidad a la hora de recuperar los datos que nos interesan de *Abiertos* y *Cerrados*.

También se utilizan HasMap para almacenar el mejor padre de un nodo, su *g* y su *f*.

El algoritmo sigue el pseudocódigo de la derecha:

```
pathFinding() {
  if (reiniciarPath == true) {
    Inicializamos Abiertos, Cerrados, mejorPadre, valorF y valorG
    Abiertos <- nodo con casilla actual, g = 0 y mejorPadre nulo
  }

  while (Abiertos no esté vacío y tengamos más de 10 milisegundos) {
    Nodo actual = valor de Abiertos con menor F

    if (actual está en la misma casilla que el objetivo) {
      reiniciarPath = True
      recuperarPath(mejorPadre, actual)
      return
    }

    Cerrados <- actual
    hijos = generarSucesores(actual)

    for (Nodo sucesor en hijos) {
      if (sucesor no está en Cerrados) {
        posible_g = valor g de sucesor

        if (sucesor no está en Abiertos)
          Abiertos <- sucesor
        else {
          if (valorG no contiene la g del sucesor)
            valorG[sucesor] <- infinito

          if (posible_g > valorG[sucesor])
            continuamos al siguiente hijo
        }

        mejorPadre[sucesor] <- actual
        valorG[sucesor] <- posible_g
        valorF[sucesor] <- f del sucesor
      }
    }
  }

  if (no se ha encontrado solución) {
    reiniciarPath = True
    return
  }

  // Si llega aquí nos hemos quedado sin tiempo pero abiertos
  // no está vacío
  reiniciarPath = false
  return
}
```

Heurística

La heurística comprueba si existe peligro en esa casilla (usando el mismo método que el de la componente reactiva) y calcula la distancia Manhattan al objetivo. En caso de peligro se penaliza la heurística sumándole 10. Se ha elegido esta por su simplicidad, ya que la distancia Manhattan nunca sobreestima el camino y el peligro nos anima a considerar otros caminos.

```
Integer calculaH (actual, objetivo) {
  if (peligro)
    return distancia Manhattan + 100
  else
    return distancia Manhattan
}
```

La función de coste será 1 excepto en el caso de que la casilla esté marcada como *DANGER*, en cuyo caso será 10. Esto hace que evitemos atravesar zonas con rocas cayendo, pero nos permite tomar el riesgo en el caso de que sea la única opción.

El marcar como *DANGER* no hace que no encontremos la solución cerrando todos los caminos posibles, puesto que el algoritmo parará cuando se quede sin tiempo y volverá a recuperar un mapa limpio.