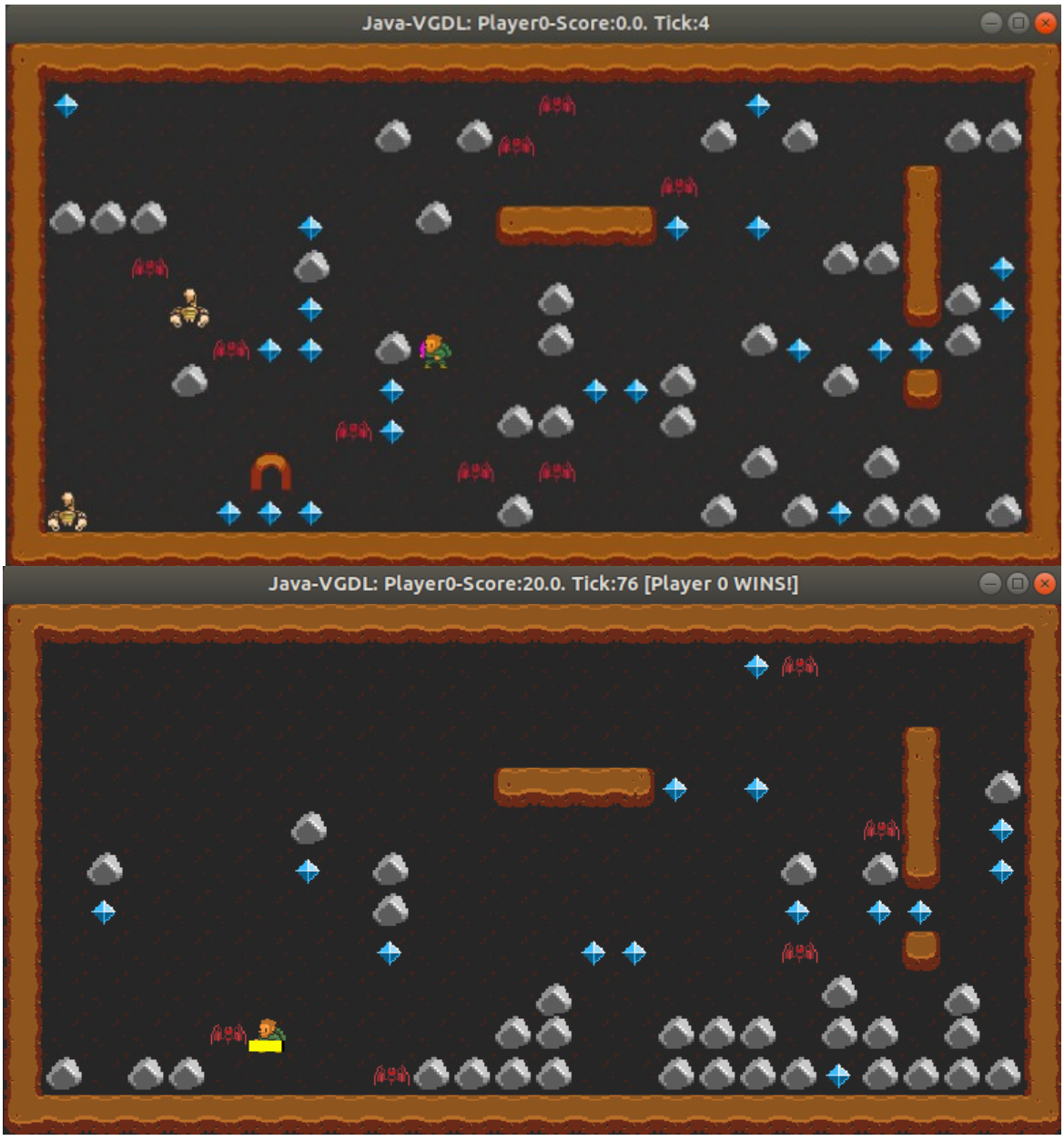
Técnicas de los Sistemas Inteligentes

Grupo 3 (Lunes)

Práctica 1 - Desarrollo de un agente basado en búsqueda heurística para el entorno GVG-AI



Alberto Estepa Fernández
Carlos Santiago Sánchez Muñoz

11 de abril de 2019

1. Descripción general de la solución

En este apartado vamos a hacer una introducción de cómo se ha desarrollado y la práctica, una explicación de los métodos principales y la integración del comportamiento reactivo/deliberativo. En primer lugar destacar que hemos usado el *wrapper* y nuestro *Agent* es una clase que hereda de *BaseAgent* explicada en el Tutorial 2.

Variables de clase del *Agent*

Nuestro agente tiene diversos objetos de clase. Todos los objetos se actualizan cuando hay que calcular un plan, más adelante se explicarán las condiciones que hacen necesario calcular un plan. En todo momento se lleva un objeto de clase *grid* con los monstruos, rocas y gemas del juego.

Se guarda un *ArrayList<Node> path* con la ruta actual, que puede estar vacío o incluso ser *null* en función del *grid*. Tenemos la posición del player en *playerObs* que se actualiza en todos los t*icks* de la partida y también *ultimaPos* con la última posición en la que estuvo el jugador.

La forma en la que se han gestionado las gemas es fundamental en el éxito de esta práctica. Hemos decidido llevar una lista de gemas denominada *gemList* con todas las gemas presentes en ese t*ick* del juego. A su vez aquellas que sean alcanzables por el jugador en ese instante se guardan en *gemListReachable* y éstas las dividimos en dos en función de si tienen una roca encima obteniendo por tanto *gemListBoulderAbove* y *gemListNotBoulderAbove*.

Del mismo modo, se contabilizan las gemas ya obtenidas en *gotGems*, el próximo objetivo (portal, gema o incluso otro) se almacena en *nextGoal*, una lista con los obstáculos del juego llamada *obstacleTypes* en la cual se incluyen los monstruos porque si en una posición hay un monstruo te obstruye el camino ya que no sólo no lo podríamos atravesar sino que además nos mataría y por último el *ancho* y *alto* del grid así como una variable *quieto* que gestiona el número de *ticks* que lleva quieto el jugador para calcular un nuevo plan.

Alcanzando objetivos: programando A*

Calculando el camino: El A* está en *findPathMap()* con un *grid* actualizado:

```
findPathMap(start, goal)
   Inicializar openList y closedList como listas con prioridad
   start.totalCost ← 0
   start.estimatedCost ← hEstimatedCost(start, goal);
   Añadir start a openList
   Mientras openlist no vacía
      node ← Sacar de openlist
      Añadir node a closedList
      Si node == goal Devolver calculatePathMap(node)
      neighbours ← getNeighbours(node)
      Para cada neighbour en neighbours
         curDistance ← neighbour.totalCost;
         Si neighbour no está ni en openlist ni en closedlist
            Actualizar coste total y coste estimado de neighbour
            Añadir neighbour a openlist
         Fin Si
         En otro caso Si curDistance+node.costeTotal<neighbour.totalcost,
            Actualizar padre y costes
            Borrar de openlist y closedlist y añadir neighbour a openlist
         Fin En otro caso
      Fin Para
   Fin Mientras
   Si node ≠ goal Devolver null
Devolver calculatePathMap(node)
```

Expandiendo vecinos: El método *getNeighbours()* calcula todos los vecinos posibles sin obstáculos:

```
getNeighbours(Nodo)
   Inicializar Lista neighbours
   Para cada movimiento M
      Calcular NodoM ← Realizar M en Nodo
      Si !isObstacle(NodoM) entonces añadir nodoM a neighbours
   Fin Para
Devolver neighbours
```

Heurística: Encontramos nuestra heurística en *hEstimatedCost()*, un método con una cierta complejidad pues considera muchos casos:

- El coste inicial es la distancia Manhattan al objetivo. Si hay diferencia de abscisas y de ordenadas, es decir, hay un giro sumo 1 al coste debido al *tick* necesario en el giro.
- Si encima de la posición de comienzo hay una roca sumo 1000 al coste y si en la posición actual hay un monstruo también se suma 1000.
- Por cada posición con distancia Manhattan 1 al *start* en la que haya un monstruo se suma 100 y si coincide con la dirección hacia *goal* se suma el doble, 200.
- Por cada posición con distancia Manhattan 2 (una diagonal a distancia 1 coincide con una distancia Manhattan 2) al *start* en la que haya un monstruo se suma 10 y si coincide con la dirección hacia *goal* se suma el doble, 20.

Por último devolver el coste estimado.

Método principal: act()

```
act(stateObs,elapsedTimer){
   Actualizamos player y grid
   Si player ≠ ultimaPos
      quieto = 0;
      Si path no está vacío eliminar la posición 0
   Fin Si
   En otro caso ++quieto
   Si getNumGems(stateObs) > gotGems
      quieto ← 0
      gotGems ← getNumGems(stateObs);
      Vaciar path
   Fin Si
   Si path vacío || quieto>=4 || monsterClosed() || boulderAbovePlayer()
      Si gotGems >= NUM_GEMS_FOR_EXIT
         portal ← getExit(stateObs);
         path ← findPathMap(player, portal);
      Fin Si
      En otro caso
         nextGoal ← bestGem(stateObs)
         Si nextGoal ≠ null entonces path ← findPathMap(player, nextGoal)
         En otro caso path ← null;
      Fin En otro caso
   Fin Si
   Si path==null
      Mientras path==null
         pos = calculatePos()
         path = findPathMap(player, pos)
      Fin Mientras
   Fin Si
   Si path no es vacío
      siguientePos ← Posición 0 de path
      Calcular siguienteAccion
      ultimaPos ← player
      Devolver siguienteAccion
   Fin Si
Devolver ACTION_NIL
```

2. Comportamiento reactivo

Nos referimos a comportamiento reactivo del agente a la continua interacción con el entorno. Es un proceso continuo de recibir estímulos por medio de los sensores y generar una respuesta adaptada en cada caso.

En nuestro programa, el agente tendrá información actualizada en cada momento del entorno, mediante los sensores proporcionados por *StateObservation* que nos proporcionará el mapa actual en cada *tick* (una matriz de *Observation*, objetos que disponen de dos coordenadas y el tipo del elemento situado en dicha coordenada). Destacamos la posición del *Player* que, obviamente, es información relevante, así como la posición de los diversos peligros que tendrá que solventar el agente (rocas, muros y enemigos), además de la posición de los objetivos correspondientes (gemas y portales).

Además en cada *tick* o instante del juego podremos sabemos si el jugador ha variado su posición con respecto al *tick* anterior.

El comportamiento reactivo se inicia una vez obtenida la decisión deliberada de qué objetivo próximo alcanzar por el agente.

Así, entrando de lleno en el programa, hemos programado varios métodos para la ejecución de este comportamiento (pondremos el pseudocódigo de métodos relevantes y de mayor complejidad):

- **monsterClosed():** Comprueba si hay un monstruo cerca de la posición dada. Tiene en cuenta las cuatro casillas en horizontal más cercanas al muñeco las cuatro verticales más cercanas y las de las esquinas. Representamos un dibujo explicativo con las posiciones a comprobar (o) desde la posición dada (P) :

o
ooo
ooPoo
ooo
o

```
monsterClosed(coordenada x del mapa, coordenada y del mapa)
   Si isMonster(linea horizontal) Devolver true, Fin Si
   Si isMonster(linea vertical) Devolver true, Fin Si
   Si isMonster(diagonales) Devolver true, Fin Si
Devolver false
```

- **boulderAbove():** Comprueba si hay una roca cayendo justo encima del jugador (es un método para una posición genérica, pero solo lo utilizaremos para el jugador, por eso el nombre especificado).

```
boulderAbovePlayer(coordenada x del mapa, coordenada y del mapa)
   Si isType(x,y-1, casillaVacía) Y  y-2>0)
      Si isType(x,y-2, Roca) Devolver true
      Fin Si
   Fin Si
Devolver false
```

- **isObstacle():** Comprueba si la posición dada es un obstáculo (se considera obstáculo tanto cualquier elemento de obstacleTypes (*BAT*, *SCORPION*, *WALL*, *BOULDER*) como si la posición de encima es una roca).

```
isObstacle(coordenada x del mapa, coordenada y del mapa)
   Si (x<0 Ó x>ancho del mapa) Devolver true, Fin Si
   Si (y<0 Ó y>alto del mapa) Devolver true, Fin Si
   Para Cada observation De mapa[x][y] Hacer:
      Si obstacleTypes Contiene A tipo de observation Devolver true
   Fin Para
   Si isType(x,y-1, roca) Y  isType(x,y, casillaVacía)
      Devolver true
   Fin Si
Devolver False
```

- Métodos básicos usados para los anteriores métodos principales:

  - **isType():** Comprueba si la posición del mapa dada es del tipo correspondiente

```
isType(coordenada x del mapa, coordenada y del mapa, tipo)
   Para Cada observation De mapa[x][y] Hacer:
      Si tipo de la observation Es Igual A tipo Devolver true
      Fin Si
   Fin Bucle
Devolver false
```

  - **isMonster():** Comprueba si la posición del mapa es del tipo monstruo. Solo debe llamar a isType(coordenada x del mapa, coordenada y del mapa, monstruo), por eso obviamos el pseudocódigo.

Comportamiento reactivo en el ACT

El metodo *act* siempre tratará de especificar el objetivo y el camino más favorable para el jugador a ese objetivo. La reactividad permitirá recalcular el camino si hay un monstruo cerca de la posición del jugador (*monsterClosed(posición del jugador)*), si hay una roca cayendo hacia el jugador en ese momento (*boulderAbovePlayer(posición del jugador)*), si está atascado o parado en una posición mas 3 *ticks*, o, claro está, si no tenemos un camino definido.

También puede darse que el comportamiento deliberativo no proporcione un camino hacia un objetivo (por ejemplo, que estemos atrapados en una sección de mapa y solo haya un camino de salida, pero haya algún peligro que impida pasar por ese camino). En esos casos disponemos de un método que intente poner a salvo al agente y hacer algunos movimientos para intentar escapar.

```
calculatePos()
   Inicializar x, y a las posición del jugador
   Si (!isObstacle(x-1,y) Y !isType(x-1,y-1,roca) n ← new nodo(x-1,y)
   En otro caso,
      Si (!isObstacle(x+1,y) Y !isType(x+1,y-1,roca) n ← new nodo(x+1,y)
   En otro caso,
      Si (!isObstacle(x,y+1) Y !isType(x,y,roca) n ← new nodo(x,y+1)
   En otro caso, Si (y>2)
      Si (!isObstacle(x,y-1) Y !isType(x+1,y-2,roca) n ← new nodo(x,y-1)
   En otro caso, n ← new nodo(x,y)
   Fin En otro caso
Devolver n
```

3. Comportamiento deliberativo

Hemos dotado al agente de una arquitectura deliberativa en la cual el agente parte de un estado inicial y es capaz de generar planes para alcanzar sus objetivos. Ya ha sido explicada la parte deliberativa del método principal *act* y el algoritmo A* implementado en el método *findPathMap()*. Es por eso que en esta sección se va a detallar el proceso para escoger la gema más prometedora y algunos casos extremos a tener en cuenta.

Calculando la gema más prometedora

El primer método a destacar es el que gestiona absolutamente todo el proceso y es *bestGem()* para el cual se ha procurado dar la máxima eficiencia posible pues los límites de tiempo son exigentes. Pero antes hay que introducir otros métodos necesarios para su ejecución.

- **calculateGems():** Actualiza las 4 listas de gemas explicadas en la sección 1.

```
calculateGems(StateObservation stateObs)
   start ← playerObs
   Vaciar las 4 listas de gemas
   gemsList ← getGemsList(stateObs)
   Para cada gema en gemsList
      goal ← gema
      way ← findPathMap(start, goal);
      Si way ≠ null && way no vacío
            gemsListReachable.add(gem);
            Si la casilla superior es roca añadir a gemsListBoulderAbove
            En otro caso añadir a gemsListNotBoulderAbove
      Fin Si
   Fin Para
```

- **positionSurroundedByMonster():** este método nos devuelve si una localización (x,y) está rodeada (distancia Manhattan 1) por un monstruo. Puede que una gema sea muy prometedora y cercana pero puede tener un monstruo custodiándola.

```
positionSurroundedByMonster(x, y)
   Si isMonster(x-1,y) Devolver true
   Si isMonster(x+1,y) Devolver true
   Si isMonster(x,y-1) Devolver true
   Si isMonster(x,y+1) Devolver true
Devolver false;
```

- **nearestGem():** este método recibe como parámetro una lista de gemas y calcula la más cercana.

```
nearestGem(list)
   Si list no está vacía
      xDiff ← |playerObs.getX() - list.get(0).getX()|
      yDiff ← |playerObs.getY() - list.get(0).getY()|
      minimo ← xDiff + yDiff,   pos ← 0;
      Para i = 1 hasta tamaño de list
         xDiff ← |playerObs.getX() - list.get(i).getX()|
            yDiff ← |playerObs.getY() - list.get(i).getY()|
            Si minimo > xDiff + yDiff
                minimo ← xDiff + yDiff,   pos = i
            Fin Si
      Fin Para
      Devolver list(pos)
   Fin Si
En otro caso Devolver null
```

¿Y si ninguna gema es alcanzable?

Existe la posibilidad que ninguna de las gemas sean alcanzables, ya que están rodeadas por rocas. Para ello tenemos un método que intenta liberarlas tirando rocas por abajo, o por los lados.

```
liberateGems(StateObservation stateObs)
   goal ← liberateGemsFromBelow()
   Si goal es nulo, goal ← liberateGemsOnTheRigth()
   Si goal es nulo, goal ← liberateGemsOnTheLeft()
Devolver goal
```

Vamos a resumir el comportamiento de los 3 métodos anteriores usados en *liberateGems()* ya que si profundizamos en el pseudocódigo nos extenderíamos demasiado en la redacción de la Memoria. Por un lado *liberateGemsFromBelow()* recorrerá la lista de gemas, que serán todas inalcanzables (si no, no se hubiese llamado al método) y para cada una, recorremos el vector de posiciones del mapa por debajo de ella hasta encontrar una roca y si alguna posición de debajo es alcanzable por el jugador (existe un camino entre el jugador y esas coordenadas), elige esa posición como objetivo, o si no es posible, pasamos a la siguiente gema hasta encontrar un objetivo.

Hay que tener en cuenta que puede haber muros debajo de la gema y tendremos que abrir por un lado el camino, así *liberateGemsOnTheRight()* recorrerá la lista de gemas y para cada una, buscará a la derecha de ella una roca y si existiese, buscaremos en posiciones inferiores alguna alcanzable (como en el caso anterior). Si existiese, devolvemos esa posición como objetivo, si no, seguiríamos con la siguiente gema. Por último, *liberateGemsOnTheLeft()* es análogo al anterior pero buscando en este caso rocas a la izquierda de la gema.

¿Y si no hay gemas?

Por último, puede ser que no existan mas gemas en el mapa, en tal caso existe la posibilidad de que dos monstruos se choquen y se conviertan en gema. Para tal caso extremo hemos programado un método que libera monstruos cavando alrededor suyo (también sirve para liberar gemas atrapadas por monstruos). Si no hay monstruos atrapados *calculatePos()* hará movimientos.

```
liberateMonsters(StateObservation stateObs)
   start ← coordenadas del player
   enemiesList ← getEnemiesList()
   Para cada monstruo de enemiesList Hacer:
      Para cada posición posible de movimiento del monstruo Hacer:
         Si es alcanzable
            Añadir Ground a obstacleTypes
            Si ahora ya no es alcanzable
               Eliminar Ground de obstacleTypes
               Devolver la posición
            En Otro caso solo eleminar Ground de obstacleTypes
         Fin Si
      Fin para cada
   Fin para cada
Devolver null
```

Metodo principal: bestGems()

Éste es el método principal para calcular la gema más prometedora y gestiona según las condiciones del *grid* a cual de todos los métodos explicados en la sección es necesario llamar.

```
bestGem(stateObs)
   calculateGems(stateObs);
   Si gemsListReachable no vacía obs ← nearestGem(gemsListNotBoulderAbove)
      Si obs == null, obs ← nearestGem(gemsListBoulderAbove)
      Si obs ≠ null
         Mientras(positionSurroundedByMonster(obs) Y tamaño de las listas>0)
            Obs ← nearestGem(la lista de la que parte - el anterior obs)
         Fin mientras
      Fin Si
   En otro caso, Si gemsList no está vacía obs ← liberateGems()
   En otro caso, Si numero de enemigos > 2 entonces obs ← liberateMonsters()
Devolver obs
```