
# Memoria  Práctica 1:

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

## Técnicas de los Sistemas Inteligentes

Christian Vigil Zamora
Sergio Aguilera Ramírez

3º - Grupo 2

# 1. DESCRIPCIÓN GENERAL DE LA SOLUCIÓN

Para obtener nuestra solución hemos recurrido a la integración de comportamientos reactivos y deliberativos. Los comportamientos reactivos han sido los que nos han resultado más fáciles de determinar pues hemos tratado de dar con ellos intentando resolver los mapas nosotros mismos de forma manual, para así ser dueños de cada acción que tomamos y determinar en qué situaciones sería necesario un comportamiento reactivo. Una vez hecho ésto, empezamos a plantearnos cuáles serían los comportamientos deliberativos más aceptados para nuestro agente.

El algoritmo principal en el que basamos nuestra búsqueda es el A*, hemos mantenido el proporcionado por GVG-AI puesto que no nos ha sido necesario modificarlo, ya que hemos añadido métodos auxiliares que forman parte del comportamiento deliberativo que complementan las carencias del A* proporcionado.

Para explicar la integración de comportamiento reactivo/deliberativo y la estrategia de búsqueda y heurística empleada, vamos a describir el proceso de razonamiento que lleva a cabo nuestro agente.

Primeramente, como parte del comportamiento deliberativo, hemos modificado el método **isObstacle** de la clase **PathFinder** ya que el proporcionado sólo consideraba como obstáculo las casillas del mapa que tuvieran una piedra o un muro, pero como a nosotros nos conviene no considerar también los nodos que puedan tener NPCS, lo que hemos hecho es que para cada una de esas comprobaciones que hacía el método sobre las casillas del mapa, tenga en cuenta que si esa casilla o alguna de las 8 que la rodean coinciden con la posición de un NPC, esas casillas se consideran también como obstáculo, descartando así nodos peligrosos. Una vez tenemos descartados dichos nodos, hacemos uso del método **bloqueaNPC**, el cual es llamado en el método **act** del Agente y cuya finalidad es aislar las casillas que rodean a un NPC, en éste caso, las 8 que lo rodean, 3 superiores, 3 inferiores y las 2 de los laterales, de esa forma nos aseguramos que esas casillas que son “peligrosas” podríamos decir, no van a ser consideradas a la hora de calcular un camino. Teniendo ya las casillas peligrosas aisladas entra en juego la heurística que hemos considerado, que consiste en tratar de realizar caminos lo más cortos posibles, y eso sólo es posible si centramos como objetivo la gema más cercana con respecto de la posición del agente. En el método **mejorCamino** es dónde plasmamos la heurística comentada, pues dicho método se encarga de

calcular un camino hacia la gema más cercana comprobando que esa gema no de un camino nulo, por lo que no siempre la gema inmediatamente más cercana es la que va a ser elegida para calcular el camino. Calcular el camino con respecto de la gema elegida es posible gracias al método que hemos implementado, **getPathAux** al que le pasamos un nodo inicio, que siempre es la posición del agente en ese momento y un nodo objetivo, que en función de cuando sea llamada, ese nodo objetivo será una gema si estamos tratando de calcular un plan hacia una gema, o un portal en caso de haber conseguido las gemas suficientes. Su función es devolver un camino mediante la función **_findPath** proporcionada por GVG-AI, que básicamente efectúa el algoritmo A* sobre los nodos proporcionados como argumentos.

Una vez calculado el plan, podemos hablar de los comportamientos reactivos del agente. El agente va a ir tomando acciones del plan ya elaborado, además de considerar la posibilidad de topar con piedras en el camino, lo cual será explicado en su apartado correspondiente.

Para que el PathFinder pueda realizar correctamente un camino, es necesario que su mapa y su sensor de Observación estén actualizados, por lo que en cada ejecución del método **act**, los actualizamos.

El aspecto más relevante que podríamos destacar de nuestro trabajo es la consideración constante de los NPCS. Si bien en un principio pensábamos que lo más relevante serían las piedras, al final nos dimos cuenta de que no, que la clave para generar buenas soluciones es tener controlados a los NPCS, dentro de lo posible, dada su aleatoriedad. De esa forma, evitando que el agente tome acciones cercanas a NPCS, rebaja la posibilidad de que puedan interferir en su camino, pues tiene a no desbloquear casillas cercanas a ellos.

**ACLARACIÓN:** En el apartado 3, en el pseudocódigo de los métodos bloqueaNPC e isObstacle, cuando ponemos “X.position” siendo X una observación o una casilla, “position” se refiere a la posición de las 8 casillas que se comprueban, no sólo a la posición exacta. Se ha puesto de esa forma para reducir el tamaño del pseudocódigo.

A continuación, mostramos la localización de los métodos comentados:

- **Agent.java :** bloqueaNPC, mejorCamino.
- **newPathFinder.java :** isObstacle, getPathaux.
- **newAStar.java :** _findPath hecho público.

## 2. COMPORTAMIENTO REACTIVO

En el comportamiento reactivo llevamos a cabo los movimientos del agente con respecto al plan obtenido mediante el método **getPathAux.** El hecho de decidir cuál va a ser la siguiente acción a tomar por el agente se lleva a cabo comparando las coordenadas 'x' e 'y' actuales del agente con respecto a las coordenadas de la siguiente posición, es decir, si siguientepos.x != avatar.x sabemos que el agente se va a desplazar a otra columna del mapa por lo que en función de si la siguiente posición es mayor o menor que la posición del avatar, deberá girar a la derecha o a la izquierda respectivamente, es decir avanzar una columna o retrocederla. En cambio si siguientepos.x == avatar.x && siguientepos.y != avatar.y, nos indica que el agente se mantiene en la misma columna pero va a cambiar de fila, y nuevamente al igual que con las columnas, en función de la siguiente posición, el agente deberá moverse hacia arriba o hacia abajo, guardando como siguiente acción el movimiento respectivo a la comparación de dichas coordenadas.

Consideramos que no merece la pena hacer mayor énfasis en la toma de decisiones por el agente pues no hemos agregado nada nuevo. Dónde sí hemos añadido un comportamiento nuevo ha sido a la hora de dar el siguiente paso, pues consideramos la opción de que pueda haber una piedra que nos obstaculice el camino o caiga encima del agente.

Para llevar a cabo ese comportamiento, antes de de dar el siguiente paso, comprobamos mediante una observación al mapa que la siguiente casilla a la que vamos a acceder no esté vacía y comprobamos si en dicha casilla hay una piedra. El hecho de comprobar si hay una piedra es para garantizar que no nos quedamos bloqueados o que no llevamos a cabo una acción que pueda hacer caer una piedra sobre el agente.

Una situación que ejemplifica lo comentado sería:

ya que el agente pretende alcanzar la gema pero una gema que ha caído obstaculiza el camino. Para solucionar dicho problema, hemos creado una variable de control llamada **nuevoPath** cuyo valor inicial es false y que es gestionada gracias a la comprobación comentada antes, puesto que si

observamos que la siguiente posición contiene una piedra que nos obstaculiza u es peligrosa, esa variable toma valor true y provoca que el agente rechace el plan actual y calcule uno nuevo.

## 3. COMPORTAMIENTO DELIBERATIVO

```
Función mejorCamino(stateObs, Avatar)

        Posiciones = getResourcesPositions(Avatar.position)
        gema = primera gema de Posiciones

        Para cada gema en Posiciones
                camino = getPathaux(Avatar,gema)

                si camino == null
                        gema = siguiente gema de Posiciones
            Fin
                si no
                        nuevoPath = false
                        Devuelve camino
            Fin
        Fin

        Devuelve camino
Fin
```

En éste método es dónde se encuentra la heurística que hemos considerado. Vamos iterando sobre las gemas más cercanas en función de la posición del avatar y con cada una de ellas se genera un camino. Si el camino que proporciona esa gema es nulo, pasamos a la siguiente gema, así hasta dar con la más cercana posible que nos proporcione un camino no nulo. También en éste método, una vez calculamos el camino, ponemos la variable de control **nuevoPath**, a false, indicando que no es necesario calcular un nuevo camino. El motivo por el que elegimos la gema más cercana en cada momento, es para reducir el número de ticks necesarios en la resolución del mapa.

```
Función isObstacle(fila, columna)

        Npcs = getNPCPositions(Avatar.position)

        Para cada observacion en el mapa[fila][columna]

                si observacion contiene un muro o piedra
                        Devuelve true
                Fin


                Para cada npc en Npcs
                        si observacion.position == npc.position
                                Devuelve true
                        Fin
                Fin
        Fin

        Devuelve false
Fin
```

El método isObstacle es de tipo Booleano. Su funcionamiento es el siguiente: realiza una comprobación sobre los nodos del mapa que se le indican. En esa comprobación observa si la casilla es considerada un obstáculo por ser piedra o muro y además comprueba si en algunas de las 8 posiciones que rodean a la casilla contienen un NPC. De ésta forma evitamos tener en cuenta nodos que son obstáculos o que son peligrosos por contener un NPC. Un ejemplo de ésta situación sería:

```
Función bloqueaNPC(stateObs)

        npcs = getNPCPositions(Avatar.position)

        Para cada npc en npcs
                si mapa[npc.position] != empty()
                        mapa[npc.position].tipo = piedra
                Fin
        Fin

Fin
```

Éste método se encarga de bloquear a los NPCS. Obtiene una lista con la posición de todos ellos y simula que en las 8 posiciones que lo rodean, si no están vacías, hay una piedra. De esa forma, esas posiciones no serán tomadas en cuenta a la hora de realizar un camino y por lo tanto el agente no pasará por ninguna de esas casillas, que si lo hiciera, daría lugar a la liberación de un NPC o de su muerte. Éste método es llamado constantemente en el método **act** del agente, puesto que la posición de los NPCS no es estática, sino que varía. Ejemplo práctico de éste método: