Memoria Práctica 1

Técnicas de los Sistemas Inteligentes. Grupo Lunes.

Pablo Baeyens Fernández Antonio Coín Castro

Curso 2018-2019

Índice

Descripción general de la solución

El objetivo de esta práctica es programar un controlador en el entorno GVGAI que consiga superar varios niveles del juego *Boulder Dash*. Debido a las particularidades de este juego, dicho controlador deberá integrar comportamiento reactivo y deliberativo, pues a pesar de haber elegido un plan de acción pueden presentarse imprevistos durante el camino que haya que solucionar.

La integración del comportamiento deliberativo y reactivo se lleva a cabo en la función `act`, que en cada turno devuelve la siguiente acción que debe realizar el avatar. A grandes rasgos, esta función se comporta del siguiente modo:

- En primer lugar, comprueba si hay un camino o plan en ejecución. En caso negativo, se encarga de llamar a la función adecuada para crear uno.
- Una vez que se tiene un plan calculado, se simula la siguiente acción. Si en la simulación no encontramos problemas, se procede a devolver la acción y finaliza la función. En otro caso, se pasa el control a una función `escape` para huir del posible peligro.
- Si en algún momento se detecta que el avatar ha quedado atrapado en un bucle, se ajustan distintos parámetros para intentar que salga de él. El último recurso es realizar una acción aleatoria mediante la función `randomEscape`. La detección de bucles es posible gracias al dato miembro `ultimaPos`, que mantiene siempre la última posición en la que se encontraba el avatar.

A la hora de buscar un plan de acción, siempre se intenta llegar al siguiente *objetivo*, que será

- una **gema** si no se ha alcanzado el número de gemas necesario para superar el nivel y hay alguna disponible,
- la **salida** si se ha alcanzado y es alcanzable o
- una casilla debajo de una **roca**. Este último objetivo se intenta cuando no se encuentra un camino viable al siguiente objetivo: en este caso, se intenta hacer caer una roca con la esperanza de abrir un nuevo camino. Esta circunstancia puede detectarse cuando el camino devuelto es `null`, y tiene la particularidad de que una vez alcanzado el objetivo debemos esperar 4 ticks a que caiga la roca antes de continuar.

Una versión bastante simplificada de la función principal del controlador puede verse en el **Algoritmo 1**.

Hay que tener en cuenta que el camino a seguir se calcula siempre a partir del estado actual del juego, y no contempla posibles modificaciones del mapa por movimientos de enemigos o de rocas. Es por esto que antes de ejecutar la acción propuesta comprobamos que no haya una roca en nuestro destino con la función `isSafe`, y también comprobamos con la función `shouldEscape` que no haya monstruos en la casilla a la

Algoritmo 1 Algoritmo principal del agente

```
function ACT(StateObservation so)
    avatar = getPlayer(so)
    if isInLoop() then
        return breakLoop(so)
    ultimaPos = avatar
    if path is empty then
        objective = computeNextObjective(so)
        path = getPath(ultimaPos, objective)
    siguienteAccion = path.next()
    if nothingToWorryAbout(siguienteAccion) then
        return siguienteAccion
    else
        path.clear()
        return escape(so)
```

que nos movemos ni en las adyacentes, simulando la acción con el método `advance` de `StateObservation`. Todas estas comprobaciones son las que se encapsulan en el pseudocódigo con `nothingToWorryAbout`.

El hecho de evitar posiciones con posibles monstruos en todas las casillas adyacentes nos puede conducir a una situación de bucle. Si detectamos dicha situación, cambiamos el modo de funcionamiento de la función `shouldEscape`: solo mira si hay un monstruo en la casilla a la que nos movemos, ignorando los alrededores.

Comportamiento deliberativo

La parte deliberativa del agente consiste en trazar un plan para llegar al siguiente objetivo, conociendo **únicamente** el estado actual del juego. Esto quiere decir que un plan que parece seguro en un momento dado puede resultar en la muerte del avatar en turnos posteriores (para evitar esto es necesario programar un comportamiento reactivo).

Para trazar un plan empleamos una versión ligeramente modificada del `PathFinder` que integra el propio entorno GVGAI, encapsulado en la clase `AEstrella`. En esencia, hacemos dos modificaciones relevantes

1. sustituimos la heurística por una que tenga en cuenta el estado del juego actual, y no únicamente el inicial y
2. adaptamos el algoritmo para que pueda buscar varias metas simultáneamente.

Sin entrar a describir el archiconocido algoritmo A* que se emplea para la búsqueda de caminos, cabe destacar que mantenemos siempre una lista de metas, ordenadas en

principio por cercanía al avatar, y que se actualizan en cada llamada al algoritmo mediante la función `updateGoals` (en función del objetivo fijado).

También disponemos de las funciones `isSafe` y `getNeighbours`, que nos permiten generar los vecinos de una casilla (arriba, abajo, izquierda y derecha) teniendo en cuenta solo las posiciones seguras o transitables. No se contemplan en esta etapa los enemigos como obstáculos, pues puede que se hayan movido en turnos posteriores. La implementación de ambas funciones es autoexplicativa.

Para utilizar este algoritmo llamamos a la función `getPath` pasándole el estado actual del juego, la posición inicial desde la que trazar el camino y el objetivo que queremos alcanzar (una gema, la salida, o excepcionalmente una casilla justo debajo de una roca). Esta función devuelve una lista de nodos (se traducen en acciones mediante la función `getAction`) que nos llevarán, en principio, al objetivo del tipo deseado con **menor coste heurístico**, que no necesariamente será el más cercano (la heurística no es admisible ya que está ajustada para evitar peligros).

Heurística empleada

Para calcular el coste heurístico de un nodo aprovechamos el algoritmo de cálculo de caminos ya existente (`PathFinder`), que nos proporciona una estimación más o menos buena del coste de ir de una posición a otra, aunque solo tiene información del estado inicial del juego. Notamos que este algoritmo utiliza internamente como coste heurístico la distancia Manhattan, y evita obstáculos de tipo muro (id. 0) y de tipo roca (id. 7).

La idea es, para cada nodo, considerar como valor heurístico el mínimo de los costes que proporciona el `PathFinder` a todos los posibles objetivos del tipo deseado, partiendo desde dicho nodo. Si en alguno de los casos no hay camino posible, se considera como valor la distancia Manhattan entre las casillas.

Además, en cada uno de los caminos a cada objetivo se ajusta el coste heurístico (antes de calcular el mínimo) por elementos peligrosos y/o deseables. En particular, se incrementa considerablemente el coste por cada monstruo *cercano* en el camino (en el propio camino o en alguna casilla cercana a una casilla del camino), y se decrementa ligeramente el coste si conseguimos pasar por debajo de una roca (es decir, hacerla caer).

Tras hacer varias pruebas, hemos considerado que un monstruo está *cerca* si está a 3 o menos casillas de distancia en cualquier dirección del nodo a considerar en cada caso. El coste por cada monstruo cercano se incrementa de forma proporcional a su distancia al nodo, obteniendo el mayor incremento en el caso de distancia 0.

Una versión en pseudocódigo de la heurística empleada por nuestro agente puede verse en el **Algoritmo 2**.

Algoritmo 2 Función heurística

```
function heuristicEstimatedCost(Node n, StateObservation so)
    cost = +∞
    for goal in goals do
        path = staticPf.getPath(n, goal)
        if path is null then
            pathCost = manhattanDistance(n, goal)
        else
            pathCost = path.size()
            for node in path do
                for each nearby node m do                ▷ También el propio nodo n
                    if so.isMonsterIn(m.x, m.y) then
                        pathCost += 11 / dist(node, m)   ▷ El caso m=node es especial
                if so.isRockIn(node.x, node.y - 1) then  ▷ Hay una roca encima
                    pathcost -= 1
        cost = min(cost, pathCost)
    return cost
```

Comportamiento reactivo

La parte reactiva del agente se concentra en la función `escape`. Como ya se ha explicado en la primera sección de esta memoria, esta función toma el control cuando se determina que es necesario realizar una acción no prevista inicialmente para poder avanzar en el juego sin quedar eliminado.

La implementación es muy sencilla. En primer lugar se generan los vecinos *seguros* de la posición actual del avatar, es decir, aquellas casillas de las 8 adyacentes que no contienen un obstáculo. Esto se realiza mediante las funciones `isSafeforEscape` y `getNeighboursForEscaping`[1]. A continuación, para cada vecino se simula la acción que nos permite ir de la posición actual a dicho vecino, y se llama a la función `shouldEscape` para comprobar si el movimiento es peligroso. En caso de no serlo se añade el vecino en cuestion a un vector de vecinos candidatos para el escape.

En principio, la función `shouldEscape` comprueba si hay monstruos en cualquiera de las casillas adyacentes, y esto puede provocar que ninguno de los vecinos considerados esté libre de peligro. En este caso se repiten todas las comprobaciones eliminando las casillas diagonales, y si aun así no hay ningún vecino candidato, se realiza una última ronda de comprobaciones considerando únicamente la casilla a la que nos movemos (y ninguna de alrededor).

[1] Una explicación de por qué se utilizan funciones distinas para generar los vecinos en la parte deliberativa se encuentra en la última sección de esta memoria.

Una vez que tenemos un vector con los candidatos, los ordenamos en función de su distancia media a los monstruos cercanos (en este caso, cercano significa a menos de 6 casillas de distancia).

Ahora hacemos una distinción de casos:

- Si el (único) vecino que provoca un desplazamiento real de casilla (y no solo un giro) está en la lista de candidatos, lo elegimos directamente.
- En otro caso, elegimos aleatoriamente de entre los dos mejores candidatos. Esto es así para evitar en la medida de lo posible caer en bucles a la hora de escapar.

Una vez hecha la elección, limpiamos el plan anterior si lo hubiera, y devolvemos la acción que nos lleva al vecino elegido.

Comentarios al código proporcionado

Hemos observado que el método `getObservationGrid` de la clase `StateObservation` tiene un comportamiento inesperado. Cuando una roca está cayendo, mantiene y actualiza correctamente las dos posiciones que ocupa mientras cae, pero cuando deja de caer, sigue mostrando que la roca está en dos posiciones a la vez (en la que está realmente y la de justo encima). Esto provoca que, en ciertas situaciones muy concretas, el agente no consiga encontrar un camino correctamente (por ejemplo, si el único camino posible pasa por encima de una roca que ha caído).

No hemos conseguido arreglar este problema empleando el método `getMovablePositions`. Como este método no considera que una roca pueda estar en dos casillas a la vez, no tenemos forma inmediata de saber si una roca está cayendo y supone un peligro a largo plazo.

Lo único que hemos podido hacer para paliar el efecto adverso de este comportamiento es utilizar la función `getMovablePositions` a la hora de escapar (es decir, en el comportamiento reactivo), pues en ese caso solo nos interesa el estado inmediatamente siguiente del juego, y no tenemos que planificar un camino en función de si una roca está cayendo.