Desarrollo de juegos con Inteligencia Artificial

Práctica 1 - **GRUPO L**

Sara Mesa

Claudia Morago

Iris Muñoz

Índice

Estructura del proyecto

Los scripts de los distintos apartados de la práctica se presentan en forma de varias clases dentro de la carpeta grupoL. **Es necesario intercambiar dichas clases en el objeto Agent para ver todos los resultados:**

- Para observar solamente el camino más corto hasta la salida seleccionar:
  - Navigation Agent: GrupoL.Agent
  - Navigation Class: GrupoL.AStar
- Para añadirle la recogida de los cofres seleccionar:
  - Navigation Agent: GrupoL.Agent
  - Navigation Class: GrupoL.AStarCofres
- Para añadirle la persecución de los enemigos seleccionar:
  - Navigation Agent: GrupoL.AgentZombies
  - Navigation Class: GrupoL.AStarZombies

En el código de la clase Agent y AgentZombies existe una opción de debug comentada para simular el supuesto de que no exista un camino posible (mundo bloqueado).

Requisitos funcionales mínimos (RF)

• RF1. El agente alcanza la salida desde su posición inicial sin atravesar muros ni límites. ✅

• RF2. El cálculo de ruta se solicita bajo demanda y el agente consume los CellInfo del camino en orden. ✅

• RF3. El algoritmo implementa GetPath(...) devolviendo una secuencia no vacía de celdas transitables cuando existe solución. ✅

• RF4. Ante la ausencia de ruta (mundo bloqueado), el agente reacciona (mensaje, reintento, o termina de forma controlada). ✅

Decisiones de diseño adoptadas en la implementación

- Algoritmo A*

Se ha implementado el algoritmo A* siguiendo las siguientes decisiones de diseño: El uso de una estructura auxiliar Node en lugar de trabajar directamente con CellInfo. Se ha decidido no modificar CellInfo ni almacenar información en las celdas propias del mapa proporcionadas por WorldInfo, evitando así problemas de consistencia e integridad. Node envuelve CellInfo y añade el cálculo de costes e implementa el nodo padre. Por otro lado, se ha creado una cuadrícula de nodos únicos para evitar crear

objetos nuevos cada vez que se llama a GetPath(), reduciendo carga de memoria dinámica y simplificando la gestión de nodos. Antes de ejecutar A* todos los nodos reinician sus costes y sus referencias, garantizando independencia. Además, se han implementado dos listas de nodos separando así los nodos pendientes de evaluar de los ya evaluados.

Por último, se ha escogido la heurística de Manhattan para calcular movimientos en 4 direcciones (no se han tomado en cuenta diagonales). Se ha reconstruido el camino usando los nodos padres y se ha mantenido la separación lógica del mundo ya dada de la lógica de navegación implementada.

- ## Recogida de cofres

Para la base de la clase que implementa la recogida de cofres, AStarCofres; se ha utilizado el código existente en la clase AStar. La diferencia clave con la clase original se encuentra en el método GetPath(), en el que ahora se implementa una lista con todos los cofres de la escena. Al tratarse de objetos estáticos se pueden guardar sus posiciones y ordenarlos del que está más cerca del agente al que está más lejos (no van a variar), de forma que el agente seguirá el mejor camino siempre.

Se implementó un bucle while que controle que mientras queden cofres sin recoger el agente siga buscándolos en el orden de la lista y que cuando se recojan se borren de la misma. Esta implementación es eficiente porque desde un inicio el agente irá por el camino menos costoso gracias a la lista y, además, gracias a la variable RecorridoCompleto, en el que se meten los cofres en orden y luego la casilla de la salida; no se producen más llamadas a métodos de las necesarias y el agente siempre concluirá su movimiento yendo a la casilla de salida. Es por añadido un método fiable puesto que si no se encuentra un camino accesible o válido se termina el recorrido y se envía un mensaje a la consola.

- ## Captura de enemigos

Para la captura de los enemigos se han implementado una clase AStarZombies y una nueva clase AgentZombies.

Al tratarse de una búsqueda online, la clave de las decisiones recae sobre el agente, que constantemente tiene que determinar el mejor camino según cambia el entorno. La clase AStarZombies es tan solo una versión simplificada de la versión original, AStar. En esencia, se deja el método GetPath() únicamente con la llamada al método AStar() para que calcule el camino que haga falta, siendo la decisión del camino a tomar exclusivamente del criterio del agente.

Por su parte, la clase AgentZombies se encarga de determinar cuál es el mejor curso de acción según la situación del entorno en cada momento. Para ello se sirve, al igual que el Agent original; del método GetNextDestination, si bien este ha sido modificado para que el agente pueda priorizar qué objetivos están más cerca en cada instante. Previa a la explicación de GetNextDestination se van a comentar los métodos necesarios para su correcto funcionamiento.

En primer lugar, se implementó un método para determinar si hay zombies cerca del agente (mayor prioridad respecto a los cofres ya que se mueven en tiempo real). Dicho método sería ZombieMasCercano(), que le indica si quedan zombies a perseguir y le devuelve la posición del zombie más cercano. El método funciona con unos bucles que recorren el grid y determinan si hay casillas con zombies en ellas (casillas con la propiedad Enemy), se calcula un camino desde la posición del agente para cada zombie y se comparan, devolviéndose aquel que tenga menor coste.

De la misma forma se implementó un método CofreMasCercano(), que está implementado con la misma lógica que ZombieMasCercano() (teniendo en cuenta las casillas del grid con la propiedad Treasure) y devuelve el camino hacia el cofre que queda más cerca del agente. Este método también es necesario porque, aunque los cofres sigan siendo estáticos, el camino del agente va a ser diferente según persiga a los zombies, así que es más eficiente que los recorra también en función del que esté más cerca en tiempo real y no como anteriormente, que solo se tenía en cuenta su posición al inicio.

Adicionalmente se crearon los métodos DistNumAlZombie() y DistNumAlCofre(). Estos métodos convierten el camino más corto hasta un zombie y un cofre respectivamente, en un valor numérico. La razón de este método es la necesidad de comparar la longitud de los caminos en cada instante para que el agente fuera capaz de decidir si seguir persiguiendo a un zombie o cambiar e ir a por un cofre según la situación.

En el código de GetNextDestination se crea un objeto “zombie” que guarda la posición devuelta por ZombieMasCercano(), y sólo si no es null (es decir hay al menos un zombie) y si la distancia a dicho zombie es menor que la distancia al cofre más cercano se establece el zombie como CurrentObjective; así, si hay un cofre más cerca que el zombie el agente cambiará de objetivo. En caso de no cumplirse estas condiciones, si se cumple que el objeto “cofre” que guarda la posición devuelta por CofreMasCercano() y de nuevo sólo si no es null (quedan cofres por recoger) se establece el cofre más cercano como objetivo. Finalmente, si no quedan ni zombies ni cofres por recoger, CurrentObjective pasa a ser la salida, que sería el objetivo final del ejercicio.

El código creado hace que el agente gestione objetivos en tiempo real, permitiendo búsquedas online complejas. En el enunciado se pide que el agente atrape a los zombies y acto seguido recoja los cofres y se dirija a la salida; esto se podría conseguir eliminando las condiciones del método ZombieMasCercano() que comparan la distancia con los cofres, de forma que el agente se centraría en los zombies y después en el resto de objetivos. Sin embargo y como ya mencionado, se ha querido proporcionar de juicio extra al agente para conseguir una ruta menos costosa en conjunto.

Eficiencia, completitud y optimalidad del algoritmo empleado

El algoritmo A* es completo, es decir, encuentra una solución siempre que exista un camino entre el punto de origen y el destino. Se explora el espacio de búsqueda hasta alcanzar la meta con el menor costo posible. La implementación de la heurística distancia Manhattan asegura que el coste estimado no supere el coste real y el primer camino encontrado al llegar al objetivo sea siempre el óptimo, por tanto, el A* implementado también es óptimo.

En cuanto a la comparación con otras alternativas, se ha concluido que el algoritmo A* con distancia de Manhattan es el más adecuado para este tipo de escenarios. El algoritmo Greedy, por ejemplo, es más rápido, pero no garantiza optimalidad. Por otro lado, algoritmos como BFS también son óptimos, pero son mucho más lentos ya que exploran muchos más nodos a medida que el mundo aumenta.

A* es eficiente al dirigir la búsqueda con una heurística. El acceso a nodos es rápido gracias al grid fijo de nodos, el cálculo del nodo con menor coste se realiza con las listas implementadas, aunque existen estructuras más eficientes que se pueden utilizar, para el tamaño del mapa dado la solución es suficientemente rápida y facilita la implementación. En general, el rendimiento es adecuado para mapas pequeños y medianos, se sacrifica algo de eficiencia por claridad y facilidad de depuración.

Heurísticas utilizadas

Se ha mencionado previamente que la heurística utilizada es la distancia Manhattan que se define como: $h(n) = |x_1 - x_2| + |y_1 - y_2|$ siendo (x_1, y_1) la celda inicial y (x_2, y_2) la celda con el objetivo final. El movimiento del agente se ve limitado a cuatro direcciones, no hay diagonales y todas las celdas tiene un coste uniforme al moverse. Está heurística es apropiada ya que nuestro grid es ortogonal y por tanto la geometría coincide bien con los movimientos permitidos. Por otro lado, la distancia Manhattan es computacionalmente muy eficiente ya que solo requiere de sumas facilitando la ejecución en tiempo real.

Comportamiento del agente

La implementación del agente en la clase Agent tiene como objetivo principal ordenar las celdas de la ruta para que el agente las recorra de manera ordenada. Para ello, usa el método GetPath() para obtener la ruta óptima calculada con el algoritmo A* (clase AStar). Esta ruta se almacena como una cola y, posteriormente, se van extrayendo sus elementos, de manera que el orden que queda siempre es del nodo más cercano al más lejano (salida).

Si la cola está vacía significa que no existe una ruta posible, y se devuelve un mensaje indicándolo.

La lógica del movimiento del agente ya estaba implementada en la clase NavigationMovement. Por eso es fundamental usar la interfaz INavegationAgent que define los métodos que Agent debe usar para conectarse correctamente con NavigationMovement.

Agent se encarga de pasarle a NavigationMovement las coordenadas correspondientes a la siguiente celda, a la que el agente debe moverse. Cada vez que el agente alcanza una celda, Agent devuelve la siguiente celda de la ruta, y así sucesivamente, permitiendo que NavigationMovement gestione el desplazamiento continuo hasta alcanzar el objetivo final.

Como resultado de la implementación de estas clases, el personaje se mueve desde la posición inicial hasta la salida por el camino óptimo, sin atravesar paredes. Se implementaron también otras dos clases: una que recoge los cofres antes de ir a la salida, y otra que atrapa a los enemigos. La clase Agent trabaja igual con estas dos clases, ya que sólo necesita usar la lista correspondiente; no calcula el algoritmo, sólo usa la cola que devuelve.

Con potencial de mejora, para mundos con paredes dinámicas que bloqueen el objetivo, se podría implementar la reacción ante los nuevos obstáculos y que, por consecuencia, se vuelva a recalcular la ruta, en lugar de devolver un mensaje como lo hace hasta ahora. No obstante, para aplicar estas mejoras sería mucho más eficiente usar el algoritmo de búsqueda en línea, de manera similar a como se hace en la captura de los enemigos, en la que está explicado el funcionamiento de AgentZombies.

Conclusiones

La realización de la práctica ha reforzado nuestros conocimientos sobre modularidad: separar el agente del algoritmo permitió una implementación clara y facilitó la división de tareas al trabajar en equipo. Además permitirá el uso de otros algoritmos y otros agentes en el futuro.

Trabajar con interfaces obliga a pensar antes de programar: qué datos entran, qué datos salen, y quién debe encargarse de cada cosa. Esto demuestra la importancia de establecer estos contratos, resultando verdaderamente útiles en la organización y conexión de las clases.

Finalmente, se ha podido observar en tiempo real cómo funcionan los algoritmos vistos en la teoría, en un mundo con su grid y sus celdas. Se compararon las fortalezas y limitaciones de A*, concluyendo que ofrece completitud, optimalidad y la mejor eficiencia para este tipo de escenarios. Sin duda, estos conocimientos podrán servir para implementaciones futuras en otros videojuegos.