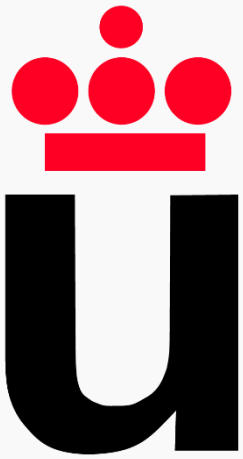

GRADO EN DISEÑO Y DESARROLLO DE VIDEOJUEGOS

DESARROLLO DE JUEGOS CON INTELIGENCIA ARTIFICIAL

Práctica grupal 2: Árboles de búsqueda

Profesores
Alfredo Cuesta
Dan Casas

Autores: Grupo X
Pedro Casas Martínez
Adrián Vaquero Portillo

Fecha de entrega
29 de diciembre de 2019

Índice

1. Descripción del algoritmo empleado para solucionar el problema.

Para esta práctica se ha empleado el algoritmo de búsqueda por árboles, específicamente el algoritmo A*. En este algoritmo, se van recorriendo las futuras acciones posibles del agente, representadas por los nodos de un árbol, priorizando las que tengan mejor valor f*, que suma la heurística (la predicción de cuánto queda para llegar a la meta) y el coste que supondría llegar hasta dicha acción.

La heurística escogida consiste en la distancia Manhattan entre la posición actual del agente y la meta. Esta heurística se trata simplemente de la suma de la distancia horizontal y vertical hasta la meta.

A la hora de expandir los nodos, se han evitado ciclos generales manteniendo un registro de todos los nodos que ya se han visitado mientras se explora el árbol.

Cabe destacar que se ha empleado el algoritmo A* tanto para la búsqueda de la meta como para la de los enemigos, si bien la primera solo se realiza una vez, mientras que la segunda se debe hacer después de cada acción, debido a que los enemigos van cambiando de posición.

2. Características de diseño e implementación

Descripción general

En nuestra clase *AStarMind*, que se añade como componente al personaje, hay un plan general que es una lista pública de nodos, que contiene de forma ordenada cada una de las posiciones que ha de recorrer el personaje para llegar a la meta actual, que podría ser un enemigo o la casilla Goal.

Cuando se accede a la función que define el siguiente movimiento del personaje, se comprueba si dicha lista está vacía. De ser así, se calcula un nuevo plan. En caso de haber enemigos, el plan consistirá en llevar al personaje al enemigo más cercano. Si ya no hay enemigos, consistirá en alcanzar la meta.

Después se comprueba si se ha creado un plan, se quita el nodo de la siguiente acción del plan, y si se está buscando a un enemigo, se borra el plan para ser recalculado la próxima vez que se llame a la función, puesto que el enemigo habrá cambiado de posición. Por último, se realiza el movimiento que indicará el nodo de la siguiente acción.

La función *setPlan*

Se trata de una función que devuelve una lista de nodos que consta de las posiciones por las que tendrá que pasar el personaje para llegar a la meta de forma ordenada.

Para esto, se van expandiendo los nodos desde un nodo ubicado en la posición actual del personaje mediante la función *expandNode* hasta llegar a la meta, si es posible llegar a ella. Tras esto, se crea el plan recorriendo los padres del nodo final hasta llegar al nodo inicial y añadiéndolos a la lista que se devolverá.

La función *expandNode*

Esta función calcula los cuatro hijos posibles y utiliza la función checkNode para saber si debe expandirlos o no.

Lo primero que hace es ordenar la lista abierta, tras lo cual saca de la lista abierta el nodo con la menor f*, que se corresponderá con el primero de la lista. De este se calculan las posiciones de las cuatro casillas que le rodean, que son sus posibles hijos, y se guardan en un array. Por cada elemento del array se ejecuta la función *checkNode*, que recibe la posición del hijo.

La función *checkNode*

Esta función decide si debe añadir un nodo a la lista abierta. Para ello, recibe una posición del escenario. Para ser añadido, debe cumplir unas condiciones:

- Debe estar dentro de los límites del escenario
- No debe ser un obstáculo, es decir, la casilla del escenario debe ser *"Walkable"*
- No debe ser un nodo que ya ha sido explorado, para evitar ciclos generales
- Como se deben eliminar todos los enemigos antes de llegar a la meta, si se está buscando un enemigo, no debe ser la meta.

Si ha cumplido todas las condiciones, se añade a la lista abierta y a la lista de nodos visitados. Si no, se sale de la función con un return.

La clase *Node*

Se ha desarrollado una clase *Node* que se usa en la creación de nodos para las listas y los planes. Un objeto de tipo *Node* contiene un coste, un valor f*, un padre y un *CellInfo* que representa su propia celda en el escenario.

En el constructor de *Node* se calcula el coste del nodo como el coste del padre + 1. Además, recibe como parámetro la meta para calcular directamente el valor f*. Esto se hace sumando el coste + la distancia Manhattan a la meta, que consiste simplemente en la suma del valor absoluto de la diferencia entre la posición del nodo y la meta en horizontal y vertical.

La clase *Node* define "getters" para la f*, el padre y el *CellInfo* del nodo. También tiene una función *equals* que devuelve true si las posiciones del nodo actual y del que se compara son iguales. Por último, se ha definido un comparador de nodos que devuelve aquel cuya f* sea menor, que se usa para ordenar la lista abierta en la implementación del algoritmo A*.

3. Dificultades y obstáculos en el desarrollo del algoritmo

Stack overflow por ciclos

Durante la implementación de la función que expande los nodos, surgieron problemas de stack overflow que causaban que Unity se quedara congelado. Se descubrió que esto estaba sucediendo porque no se estaban evitando ciclos generales de forma correcta, resultando en bucles infinitos.

Para solucionar el problema, se creó una lista global para almacenar todos los nodos visitados. De esta forma, al expandir nodos, en la función *checkNode*, se comprueba si el nodo que se está comprobando actualmente ya ha sido visitado, y si es así se descarta, para que no sea incluido en la lista abierta.

Mejoras en la clase *RandomMind*

El hecho de que los enemigos pudieran atravesar paredes o salirse del escenario suponía un problema para que el personaje los derrotara antes de poder ir a la meta.

Para solucionarlo, se hicieron algunos cambios en la clase *RandomMind*: cuando se genera el valor aleatorio del 0 al 3, se entra en un switch, y por cada valor, se examina si yendo por la dirección correspondiente el enemigo se saldría del escenario o si entraría en una celda que no es *"Walkable"*. Si ocurre uno de estos dos casos, no se devuelve ningún movimiento.

Problemas al generar múltiples planes

Cuando se implementó la búsqueda de enemigos, el algoritmo no estaba preparado para, una vez creado un plan, crear otro después. Debido a esto, siempre se creaba el mismo plan, y el personaje siempre trataba de hacer el primer movimiento de ese plan.

Para solucionarlo, al final de la función *setPlan*, se añadieron unas líneas de código que limpian la lista abierta y la lista de nodos visitados, y establecen el nodo meta a "null", siendo esto necesario para que se pueda generar una lista desde cero.

4. Discusión sobre los resultados obtenidos.

En conclusión, estamos satisfechos con el funcionamiento del algoritmo que hemos implementado. Nos ha resultado más simple de implementar que el algoritmo de Q-Learning, y hemos podido observar que los resultados han sido muy similares, por lo tanto, consideramos que es un algoritmo más práctico para este problema en particular debido a su simplicidad. Hemos podido observar la efectividad de la heurística de distancia Manhattan en problemas con un escenario en dos dimensiones como el que se presentaba en esta práctica. También nos alegra haber podido crear una clase flexible, que sirve tanto para buscar solo la meta en la escena de *PathFinding*, como para buscar tanto los enemigos como la meta en la escena de *Enemies*.