Belkan

José María Ramírez González

Primer nivel

En este primer nivel se nos pedía buscar el camino mínimo en acciones implementando una búsqueda en anchura.

En mi caso, he optado por usar el algoritmo A*, modificado para que no incluya heurística de ningún tipo, lo que produce una **Búsqueda en Anchura**.

Esto se debe a que la fórmula de A* es la siguiente:

> $f(n) = g(n) + h'(n)$

Esta es la fórmula general, pero lo que hacemos es igualar $h'(n)$ a 0 y $g(n)$ a 1, esto se hace ignorando por completo la heurística y recorriendo los nodos en una cola (en contraposición con la pila que usamos en la búsqueda en profundidad).

Código

Hemos usado un **unordered_map**.

Al usar esta estructura de la STL, junto con una nueva estructura de nodos , evitamos la redundancia de tener una lista de acciones en cada nodo, algo que es **extremadamente ineficiente** y lo sustituimos por el unordered_map *Cerrados*, cuya clave es un estado (es decir, una posición) y almacena un nodo con el *estado padre* y la acción necesaria para llegar al *estado hijo* (el que usamos como clave).

Esto provoca una notable mejora en el uso de memoria y en la velocidad de ejecución.

```
struct nodoAnch {
    estado st;
    Action accion;
};
```

Para formar la lista de acciones, simplemente vamos recorriendo `Cerrados` en orden inverso.

Esto garantiza el menor número de giros posible (y por tanto, de acciones, al priorizar la línea recta frente a los giros en una posición) ya que el orden en el que introducimos los nodos en `Cerrados` define el orden en el que el método *find* de `Cerrados` encontrará los estados.

Lo primero que evaluamos es la posibilidad de avanzar, y esto es, por tanto, lo primero que se introduce en `Cerrados` (a no ser que no sea posible, debido a un obstáculo).

Ejemplo

Veamos una ejecución de esta implementación de A* en el *mapa50*:

Segundo nivel

En este segundo nivel se nos pedía buscar el camino mínimo en costo.

En mi caso, he optado por usar el algoritmo A*, comentado anteriormente, modificado para que incluya una ligera heurística con los costes asociados a los distintos tipos de casilla, lo que produce una **Búsqueda de Costo Mínimo**.

Código

Aquí los nodos abiertos los insertamos en una cola de prioridad, ordenada en función del coste de los nodos, así nos aseguramos que siempre escogemos el camino de menor coste.

Hemos creado un nuevo struct `NodoCost` con los siguientes elementos:

```
struct nodoCost {
    estado st;
    bool bikini;
    bool zapatillas;

    int costo;

    inline bool operator == (const nodoCost &otro) const {
        return st == otro.st && bikini == otro.bikini && zapatillas ==
otro.zapatillas;
    }

    inline bool operator != (const nodoCost &otro) const {
        return !(*this == otro);
    }

    inline bool operator<(const nodoCost &otro) const {
        return this->costo > otro.costo;
    }
};
```

También tenemos que tener en cuenta si tenemos el *bikini* o las *zapatillas* y puede que el jugador se desvíe a por el *bikini* o las *zapatillas* para reducir el coste final del recorrido.

Como vemos, evaluamos los costes de las casillas y forzamos colisiones en `Cerrados` para que en caso de que coincidan 2 estados, nos quedemos con el que lleva el menor coste asociado.

No tenemos en cuenta el elemento *costo* de los `nodoCost`, ya que precisamente si la metiéramos en la función de hashing, estaríamos evitando las colisiones que se producirían si 2 nodos tuvieran costo distinto, y con ello, la sustitución de caminos de mayor coste acumulado por los de menor coste.

Esta implementación es muy eficiente, tanto en memoria como en velocidad de ejecución (aunque se podría mejorar el uso de memoria).

Ejemplo

Veamos una ejecución de esta implementación de A* en el *mapa50*:

Tercer Nivel

En este tercer nivel se nos pedía buscar el camino mínimo en costo para 3 objetivos.

En mi caso, he optado por usar la implementación usada en el nivel anterior, pero con ligeras modificaciones que permiten su funcionamiento para 3 objetivos.

Código

Comprobamos que tenemos 3 destinos y creamos un `nodoCost` con los datos del origen.

Usamos una función para ordenar los destinos en función de la distancia (calculada en Manhattan), para asegurarnos de que vamos siempre al **destino más cercano**.

La función en cuestión es la siguiente:

Y una vez ordenado, lo pasamos a una versión del algoritmo implementado en el *Nivel 2* que altera el punto de origen que le pasamos (nuestro `nodoCost` *actual* en este caso), para hacer que cuando llegue al destino, nuestro `nodoCost` *actual* sea el estado del destino, junto con si llevamos o no zapatillas o bikini.

Usamos una lista de *Acciones* auxiliar (llamada *aux*) para que el plan se vaya cargando en la variable `plan` de forma correcta.

Ejemplo

Veamos una ejecución de esta implementación de A* en el *mapa50*:

Como vemos, una vez captura el primer objetivo, sabiendo que tiene el bikini, descarta la opción de darse la vuelta y continúa por el agua para hacer un camino que al final resulta de menor coste.