# IA DEL VIDEOJUEGO

En este videojuego, toda la IA se centra en el comportamiento de los enemigos. Esta se basa en la búsqueda de caminos para dirigir a los enemigos hacia distintos objetivos y para ello se ha implementado una adaptación del algoritmo A\* en la que se han añadido restricciones de espacio y de tiempo de proceso. A continuación se detalla el funcionamiento del núcleo principal de la IA.

**ELEMENTOS QUE INTERVIENEN**

* Estructura de nodos:

typedef struct Nodo {

TPosition position; // Posicion en la matriz del mapa (i,j)

u8 f,g,h;

struct Nodo \*padre;

TEntitySide side; // Lado hacia el que miraría la entidad en este nodo debido al movimiento anterior

}TNodo;

Al utilizar un tile map, podemos hacer una equivalencia entre un nodo y un tile, y cada uno de estos representa una coordenada del mapa.

* Almacén de nodos: colaFrontera[30]
* Funciones de apoyo al algoritmo principal:
  + void quicksort(TNodo \*nodos, i8 first, i8 last): Ordenamiento del array de nodos según su f.
  + u8 manhattanDistance(TPosition\* init, TPosition\* finish).
  + void getChildrens(TNodo\* hijos, i8 \*n\_nodos, TPosition \*finish): Obtención de los nodos hijos de un determinado nodo.
  + void A(TEnemy \*enemy, TPosition \*finish): Función del algoritmo principal.

**PSEUDOCÓDIGO A\* (adaptado)**

A\*

{

n apunta a colaFrontera[0]

n = posicionNodoOrigen

n-> calcular g,h,f

nNodosAlmacenados = 1

nNodosFrontera = 1

Mientras colaFrontera != vacia && nodosExplorados < 10

Si n es meta

salir del bucle

Si no

expandir n (getChildrens)

recalcular nodosAlmacenados

Si nNodosFrontera > 1

n++ // n apunta al siguiente nodo en la colaFrontera

nNodosFrontera = nNodosFrontera-1

ordenar colaFrontera (quicksort) // De esta forma queda el nodo más prioritario en la posicion a la que apunta n

Si nNodosAlmacenados >= 28

salir del bucle

FinSi

FinMientras

calcular movimiento a partir del nodo al que apunta N

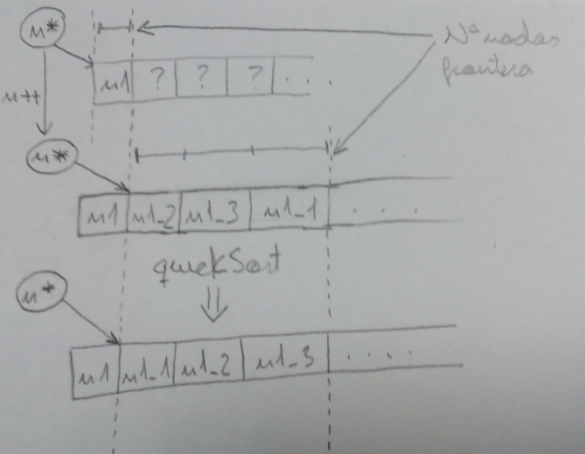
almacenar 4 movimientos adicionales del camino calculado

}

**FUNCIONAMIENTO**

El algoritmo trabaja sobre un array de nodos “constante” de 30 posiciones el cual es reordenado en cada vuelta del bucle de búsqueda en función de lo prometedor que sea.

El funcionamiento básico de la exploración de los diferentes nodos mediante el array de nodos es la siguiente:



1. Se crea un puntero N a la primera posición de colaFrontera[ ]
2. Se expande el nodo al que apunta N
3. Una vez expandido el nodo y almacenado sus hijos, se apunta a la siguiente posición del array. De esta forma junto con otra pequeña restricción que comentaremos más adelante, se descartan los nodos explorados para no repetirlos
4. Se reordenan las posiciones del array desde N hasta el último nodo introducido.
5. Tenemos el nodo más prometedor en N y repetimos el proceso

Cuándo un nodo correspondiente a una posición del mapa se introduce en el array, se marca internamente para que no se vuelva a introducir otro nodo con la misma posición. Esta opción es posible ya que el cálculo del camino está restringido a la expansión de 10 nodos, por lo que los caminos calculados van a ser en distancias cortas y es difícil que haya diferentes caminos que pasen por un mismo nodo, a parte que no se requiere una solución óptima, sino que se pone más intención en optimización de memoria y ciclos de reloj para que el juego sea fluido y de esta forma se ahorra tener que crear una “lista interior” y las comparaciones que el uso de esta conllevaría.

Restricciones y diferencias principales añadidas sobre A\* tradicional:

* Máximo 10 nodos expandidos.
* Cada posición del mapa solo podrá almacenarse una vez (un nodo) en el array de nodos.
* Supresión de la “lista interior” y su función.
* Siempre devuelve una solución.

**MOVIMIENTO DE LOS ENEMIGOS**

El algoritmo explicado junto con ciertos patrones de toma de decisiones basados en un estado/tipo de enemigo define el comportamiento de estos.

Estructura de un enemigo:

typedef struct Enemy {

TEntity entity;

u8 range\_sides, range\_front; // Tamaño del rango (ancho, largo)

TSensor range; // Rango de visión calculado->{i\_min, i\_max, j\_min, j\_max}

TEntitySide nextMoves[4]; // Movimientos almacenados

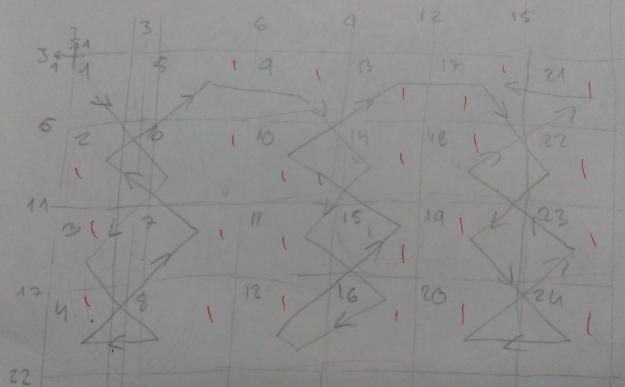
TAnimation\* animation;

TEnemyType type; // Estado/tipo de enemigo

}TEnemy;

Como vemos, los enemigos disponen de un sensor de visión y dependiendo de este tenemos tres opciones a la hora de hacer un movimiento:

* **Personaje principal está dentro del rango:** Ejecutamos el A\* con origen en la posición del enemigo y con destino en la posición del personaje principal.
* **Humano está dentro del rango:** Ejecutamos el A\* con origen en la posición del enemigo y con destino en la posición del humano.
* **No hay nada en el rango (movimiento básico):** En este caso, se calcula una posición del mapa “aleatoriamente” en base a la posición actual del enemigo. Para ello, se ha subdividido el mapa en diferentes zonas representadas con la siguiente imagen:



Dependiendo de la zona en la que se encuentre el enemigo, se calcula una posición aleatoria dentro de la “zona destino” que se muestra en el dibujo (siguiendo la dirección de las flechas). Una vez obtenida esta posición, se hace el cálculo del camino haciendo uso de la función A\*.

Cada vez que se ejecuta una búsqueda de camino, a parte del siguiente movimiento a realizar, se almacenan los 4 próximos movimientos del camino. De esta forma hemos conseguido:

* Dotar de memoria al enemigo, por si al cambiar su orientación en el movimiento pierde de vista el objetivo visualizado ya que es un sensor de vista frontal, y permite una mejora en la persecución.
* Poder realizar movimientos memorizados y de esta forma ahorrar cálculos para ganar fluidez en la jugabilidad.

**TOMA DE DECISIONES**

En base a los dos estados posibles de los enemigos, diferenciamos los siguientes tipos de comportamiento para la decisión del siguiente movimiento:

1. **Agresivo (objetivo prioritario personaje principal):**
   1. Siempre que el personaje principal se encuentre dentro del rango de visión, llama al algoritmo A\* para calcular el camino hacia él, sin hacer uso de la memorización mientras siga el personaje principal dentro del rango.
   2. Si no ve al personaje principal, consulta si tiene movimientos pendientes en la persecución de algún objetivo calculado anteriormente, y si lo tiene lo lleva a cabo.
   3. Si no se cumple ninguno de los predicados anteriores, consulta el rango de visión por si encuentra algún humano en él, y de encontrarlo, realiza la búsqueda para comérselo.
   4. De lo contrario, realiza el movimiento básico.
2. **Competidor (objetivo prioritario humanos):**
   1. Si hay un humano en el rango de visión, realiza la búsqueda para comérselo.
   2. Si no ve humanos, consulta los movimientos pendientes, y de tener, ejecuta uno.
   3. Si no tiene movimientos pendientes, consulta el rango para ver si encuentra el personaje principal, y si lo encuentra, hace una búsqueda hacia su posición y ejecuta un movimiento.
   4. De no cumplirse nada de lo anterior, realiza el movimiento básico.

**RESULTADOS Y CONCLUSIONES**

Tras muchas pruebas y modificaciones con la intención de optimizar el algoritmo en cuanto a memoria y tiempo de proceso, junto con una buena distribución a la hora de actualizar el movimiento de cada enemigo del mapa (calibrado del bucle principal), se ha conseguido una muy buena fluidez del juego con el máximo número de enemigos del juego (12), incluso se podrían introducir más enemigos con un buen rendimiento, por lo que el resultado ha sido muy bueno.