UNIVERSIDAD
DE GRANADA

TÉCNICAS DE LOS SISTEMAS INTELIGENTES

Práctica 1, Grupo 3.

Agente Basado en Heurísticas en el entorno GVGAI

Lukas Häring García

Marzo 31, 2020

Comportamientos.

Cada nivel presenta dificultades añadidas de forma incremental, por lo que cada nivel requiere de técnicas complementarias para ser resueltos. Por ejemplo, vamos a unificar dentro de la heurística, la gestión de enemigos, así, el camino óptimo también tendrá en cuenta la resolución de enemigos.

Hay que destacar que la información en el mapa se encuentra codificada en píxeles, por lo que habrá que realizar la conversión para posicionar cada elemento en su rejilla, es por ello que cuando nos refiramos al *factor*, nos referimos a $\frac{1}{tam\ bloque}$, este número será multiplicado por la posición de cada elemento, gema, posición del personaje, enemigos, etc. De aquí en adelante, supondremos que hemos multiplicado por dicho factor cuando nos refiramos a posiciones.

1. Deliberativo simple.

Aunque en este momento no hay gema, se tendrá en cuenta para el futuro de la práctica el mirar si hay para gestionar el momento en el que tomar la salida.

Se ha creado una clase *A_Node* que mantiene información sobre el nodo del que viene (null si no viene de ninguno), la posición, el vector de dirección al que está mirando tras realizar el movimiento, información sobre la función heurística, “g”, “h” y “fcal”. Dentro de esta clase, que está en el ámbito de la clase del Jugador, tiene los los métodos para el cálculo de cada uno respectivamente.

Para el cálculo de “g”, se tendrá en cuenta el valor “g” del nodo del que viene “ g’ ”, incrementado en 1, ya que es el valor de desplazar de una celda a otra sumado a esto el coste de 1 por cambio de dirección del que venía.

$$g = g' + 1 + \{dir \neq dir' \quad 1 \quad otherwise\ 0\}$$

El cálculo de “h” irá variando ya que añadiremos nuestra heurística para la gestión de enemigos, explicada más abajo. Por ahora, solo mantendrá información sobre la distancia, en la métrica del espacio, de la posición del nodo objetivo (en este caso la salida, pero podría ser una gema).

$$h = || \widehat{actual} - \widehat{objetivo} ||_{taxicab} / || \widehat{v} ||_{taxicab} = |x| + |y|$$

Claro está que $f = h + g$, por lo que definimos que un nodo *A_Node con valor "f"* está más cerca que otro con valor "f'", sí solo sí, $f < f'$. Así, podremos ordenar dichos nodos para poder seleccionar aquel más relevante de los no explorados.
Una vez definido dicha estructura, se presentará el algoritmo A*. Este recibe el estado del juego (StateObservation), posición de Inicio, posición del nodo objetivo y el tiempo utilizado.

Como anteriormente hemos definido un tipo de orden, vamos a utilizar una estructura eficiente, conocida como **Cola con Prioridad**, ésta estructura realizará el trabajo "feo" de ordenar los elementos a medida que los vamos introduciendo, esta estructura mantendrá información de "la lista de abiertos", nodos que aún no hemos explorado y pueden ser prometedores.

Además, vamos a mantener una matriz de la dimensión del mapa para mantener información sobre los nodos explorados y aquél mejor explorado.

Para comenzar, vamos a crear un nodo inicial, que será introducido en dicha lista y en la matriz, mantendrá información de la posición de inicio; la dirección a la que se encuentra el jugador mirando inicialmente y con valor "h", "g", y "fcal" de 0.0. Mientras haya algún elemento en la "lista de abiertos", vamos a coger el primero de dicha cola y vamos a comprobar si este es igual al nodo objetivo. **Nota**: Comprobamos que dos vectores son iguales realizando una resta y viendo si la longitud de dicho vector es menor a un ε.

Si este es final, el nodo que hemos retirado de la lista. En caso contrario, vamos a expandir a sus **vecinos directos**, si es posible. Verificamos si podemos expandir un nodo, si no existe un muro en la matriz de observación "getObservationGrid()". Si no existe un muro, entonces creamos el respectivo *A_Node*, además de calcular el valor "f". Si previamente no había un nodo, lo marcamos como visitado y lo añadimos en la lista de abiertos o Si previamente estaba explorado y el coste de "g" es **menor o igual** al previo, reemplazamos y lo añadimos a la lista.

Como vemos, el valor de la heurística, es constante para todo nodo en el cómputo del A* (pero no en el tiempo), además, existen dos nodos cuya heurística, en este caso, la distancia en dicha métrica, sea idéntica y sea la el valor "g", trabajará como el discriminador ***(Véase Imagen 1)***. Esto provocará que el valor en la matriz, se actualice al nodo creado, ya que lo mantendrá como el mejor.

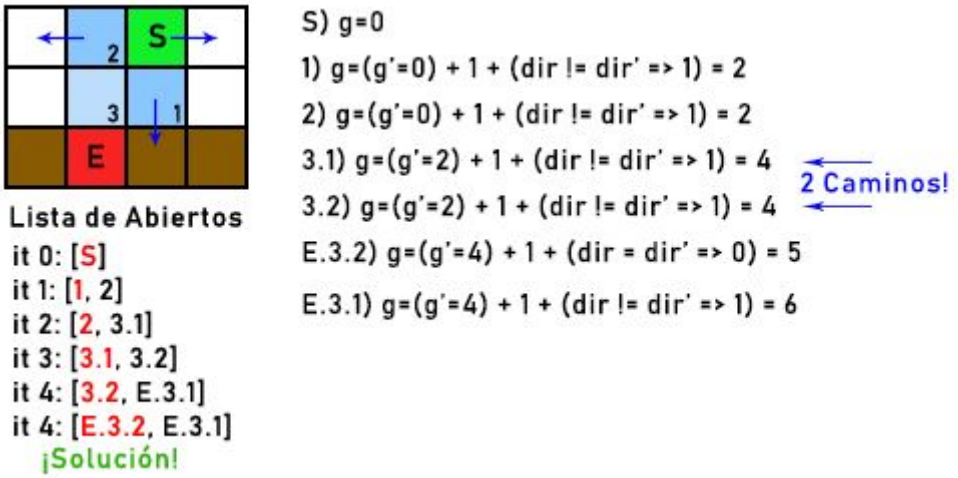


Imagen 1. Bifurcación con solución al final.

Una vez generados los posibles vecinos, se volverá a repetir este proceso hasta que se haya alcanzado el nodo final o quede vacío, si es así, se devolverá null. Si se encontró solución, este devolverá un *A_Node* en la posición del nodo objetivo, como este tiene un puntero al anterior y así sucesivamente hasta que lleguemos al inicio, podemos realizar la técnica de backtracking y decidir a partir de los dos primeros nodos, qué camino tomar.

Si el nodo devuelto tuviera como puntero al anterior, quiere decir que este es inicial y final, por lo que el recorrido se habría realizado y se comprobará que elemento hay debajo, si hay una entrada, se tomará *ACTION_SCAPE*. Si se puede realizar *backtracking*, entonces se mirará dos elementos por delante y se irá "escalando" hasta que éste sea nulo, así obtendremos el primer nodo y el siguiente.

Con estos dos elementos, podemos obtener la dirección a tomar, para ello, vamos a tener una lista con un orden específico, ABAJO, IZQUIERDA, ARRIBA y DERECHA, cuyos valores numéricos, representan 0, 1, 2, 3 respectivamente (idx). Utilizando la dirección a la que está mirando en ambos nodos,

$$\widehat{v} = normalize(\widehat{current} - \widehat{next}) \Rightarrow idx = 1 + floor(\widehat{v}_\alpha \cdot \frac{2}{\pi})$$

Donde $\widehat{v}_\alpha \in [-\pi, \pi]$ es el ángulo sobre el eje x, lo único que hacemos es transformar el dominio a $[0, 3] \in N$, (**Véase Imagen 2**).

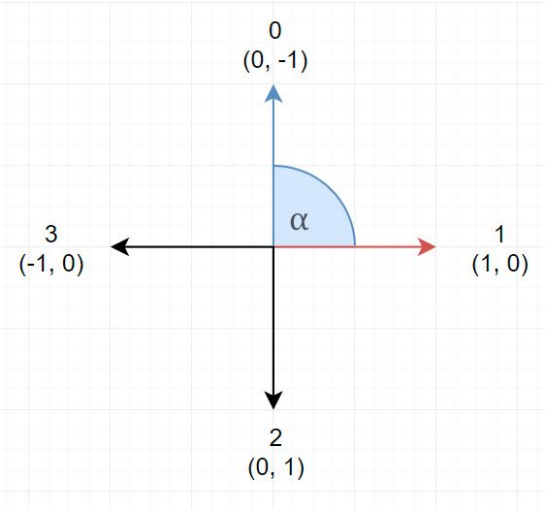


Imagen 2. Transformación de la dirección a un índice.

Esto se realiza para evitar utilizar condicionales (branching) y si la normalización se hace utilizando la inversa de raíz cuadrada, se vería una mayor ganancia.

Finalmente, se devolverá dicha decisión tomada. Este algoritmo requiere aproximadamente 1ms - 2ms según el análisis realizado en mi computadora, por lo que podremos utilizarlo en el futuro sin ningún problema, por ejemplo en la gestión de gemas.

2. Deliberativo Compuesto.

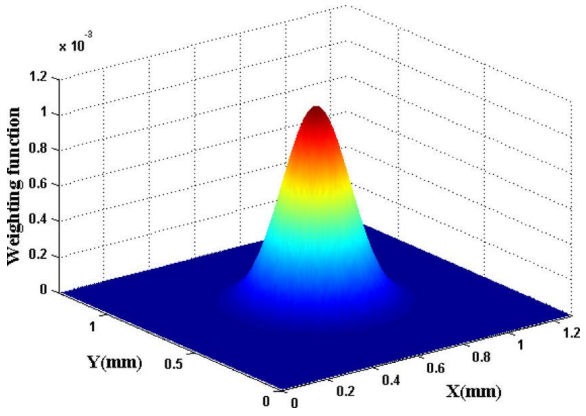
Para planificar un recorrido que seleccione las 10 gemas cuyo recorrido sea más corto, vamos a utilizar la técnica de Branch and Bound. Para ello, los nodos van a contener la información del índice a la siguiente gema a recoger y sin repetir. Además, del valor acumulado hasta llegar a ella a través de las demás gemas planificadas (o del inicio).

Vamos a utilizar otra lista ordenada para explorar los nodos en profundidad, dándo prioridad a los más profundos. Una vez alcanzada una profundidad de 11 (10 Gemas + Entrada) y obtenido el valor acumulado, se cambiará el umbral de poda si este es menor al anterior. El umbral de poda es la suma de los recorridos para coger las 10 gemas, si el recorrido del nodo fuera superior a este, no se generarían sus hijos.

Se romperá el algoritmo si este supera *900ms*. El resultado será una lista posiciones a la inversa, que obtuvo menor umbral. Si por un camino tomado, se cogido una otra gema planificada, esta se eliminará de la lista de posiciones.

3. Reactivo Simple y Compuesto.

La técnica utilizada va a permitir generalizar para un número “k” de enemigos, la técnica es el llamado “mapa de calor”, que será útil ya que podremos incluirla en nuestra heurística. Un mapa de calor está basado en funciones exponenciales, por lo que vamos a centrarnos en la función gaussiana normalizada bidimensional, ya que esta tiene la peculiaridad de que a medida que nos alejamos, se aproxima a cero. Al encontrarnos en un mapa discreto, vamos a utilizar una discretización de esta función bidimensional, más en concreto de $[-3\sigma, +3\sigma]$ ya que la representa al 99,8%.



$$\frac{1}{273}\begin{bmatrix} 1 & 4 & 7 & 4 & 1 \\ 4 & 16 & 26 & 16 & 4 \\ 7 & 26 & 41 & 26 & 7 \\ 4 & 16 & 26 & 16 & 4 \\ 1 & 4 & 7 & 4 & 1 \end{bmatrix}$$

Imagen 3.Función Gausiana a la Izq. A la derecha su discretización 5x5.

Ahora en cada iteración se va a crear un mapa de calor con valores 0.0, del tamaño del mapa. Para cada enemigo, vamos a “posicionar” dicha matriz sobre la posición de estos, sobre el centro de la matriz. Pero nos surge un gran problema, el mapa de calor es capaz de atravesar muros, por lo que tenemos que comprobar si la casilla no está bloqueada, para ello, vamos a utilizar el algoritmo de Bresenham (Véase **Imagen 4**). Se comprobará que la línea que calcula dicho algoritmo no ha sido bloqueada (se puede atravesar por el jugador para cada celda que calcula el algoritmo). Si se puede atravesar, incrementamos el valor de la matriz en el mapa, en caso contrario, mantenemos el valor que ya estaba.

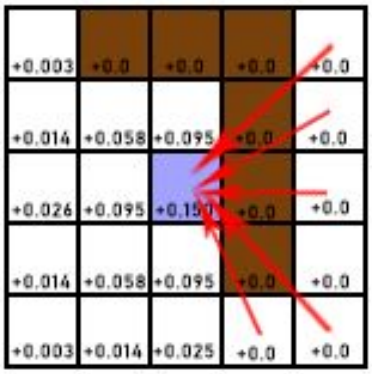


Imagen 4. Mapa de calor de un enemigo rodeado por muros..

Para introducir el mapa de calor en la heurística, vamos a definir la integral discreta, como la suma de valores, también atravesables, de una región sobre el mapa de calor, por lo que añadido a lo que teníamos antes,

$$h = ||v_i - v_s||_{taxicab} + w \sum_{i=-c}^{c} \sum_{j=-c}^{c} hm(x+i,\ y+j) \leq ||v_i - v_s||_{taxicab} + w \int_{c} hm(x,y)dc$$

Siendo w un factor sobre el mapa de calor y $hm(x,y)$ la función bidimensional de nuestro mapa de calor. Se ha tomado la decisión de realizar una integral ya que permite especificar el ratio de proximidad al mapa de calor de los enemigos, arriesgando menos cuanto mayor sea el área de integración "c^2". Además, el factor w permite manipular la probabilidad de arriesgarse, es decir, cuanto mayor sea, más prudente serán los movimientos y cuanto menor, menos.

Un comentario extra, todo estos cálculos podrían haber sido optimizados utilizando multiplicación de matrices utilizando la GPU y la separabilidad de un "kernel gaussiano", aplicándose por filas y luego por columnas o viceversa.

Para mantenerse alejado constantemente de los enemigos, en cada tick, he decidido coger una región radio r, recomendable mayor al kernel, sobre el personaje, para cada celda, se ha ejecutado A* desde la posición del personaje y se ha tomado aquel camino que minimice f ya que se ve afectado por el mapa de calor que a su vez, esquivando a los enemigos.

Esta estrategia hace que el personaje tienda a ir a las paredes, pero como este necesita girar, hace que esté siempre en desventaja con respecto a los enemigos ya que estos no lo necesitan.

3. Reactivo Deliberativo.

Este trata de unificar todo lo anteriormente expuesto.En primer lugar, vamos a utilizar la planificación de las gemas, ignorando a los enemigos, su mapa de calor.

Una vez planificado, vamos a utilizar dicha lista de posiciones para generar otra ruta en tiempo real que sí depende del mapa de calor. Recogiendo así las gemas en el orden preestablecido, además de ser capaz de esquivar. Pero surge un problema, si el enemigo está protegiendo una gema, es decir, su mapa de calor está sobre la gema, puede ser que necesitemos planificar de nuevo. Esto requeriría de un cómputo enorme, por lo que, en vez de replanificar, vamos a generar los dos caminos posibles para la dos gemas, la actual y la siguiente, teniendo en cuenta el mapa de calor, si el resultado de tomar la siguiente, resulta de tomar un camino más corto, entonces cambiamos las entre sí, las posiciones planificadas.

Tras realizar varios test, he observado que tras utilizar esta técnica, ha pasado de un 86% a un 97% de los mapas ofrecidos (utilizando 1000 mapas), considerándose una notoria mejoría.