PRACTICA 2: AGENTES REACITVOS/DELIBERATIVOS (LOS EXTRAÑOS MUNDOS DE BELKAN)

DESCRIPCION DEL PROBLEMA

La práctica tiene como objetivos poder definir un comportamiento reactivo /deliberativo para un agente que pretende moverse sobre un mapa para alcanzar un objetivo. Existen dos niveles, uno en el que se pide alcanzar un objetivo usando 3 algoritmos de búsqueda diferente y un segundo nivel en el que es necesario alcanzar un conjunto de objetivos aleatorios de forma consecutiva hasta darse alguna de las siguientes condiciones de parada: consumo absoluto de la batería recargable (3000 de batería total), consumo de tiempo de ejecución (3000 en total) o agotar el tiempo consumido, que es el tiempo total que está el programa calculando caminos (cuyo límite está en que en total haga 300 unidades como mucho)

SOLUCION PLANTEADA PARA EL NIVEL 1

En cuanto al nivel 1, los algoritmos de búsqueda planteados son realmente similares al de búsqueda en profundidad proporcionado como herramienta en la práctica, pero con pequeñas variaciones.

En el caso de la búsqueda por anchura, me baso en el pseudocódigo de la página 34/128 del pdf del Tema 3 de la asignatura. Este algoritmo es eficiente si las metas están cerca, pero consume memoria exponencial. Su desarrollo se basa en una planificación FIFO (por lo que se usa una cola para la lista de Abiertos), en lugar de la LIFO usada en el de Profundidad (que hacía uso de una pila para los Abiertos).

En la búsqueda con costo uniforme, el pseudocódigo de referencia se encuentra en la página 38/128 del mismo pdf del Tema 3. En este caso, se tiene en cuenta el costo de cada casilla pues, esta, dependiendo del tipo que sea (determinado por su letra) tendrá asociado un valor que signifique la batería que pierde al pasar por esta. Este algoritmo requiere de la creación de un atributo nuevo en el struct *nodo* llamado costo que contendrá, tras ser calculado con la función *ValorCasilla(const estado &stado),* su coste individual. Al existir cierta prioridad en la elección del siguiente nodo, la estructura de datos usada para la lista de abiertos en este algoritmo es una cola con prioridad a la que se le modifica el *operator* < poniendo el requisito de que el coste menor tiene más prioridad. Ese valor no es tenido en cuenta más que en los nodos en los que se implica avanzar, pues el giro cuesta 1 unidad de batería nos encontremos donde nos encontremos.

Tras estas implementaciones, el nivel 1 queda funcionando, llegando a ser eficiente incluso en los mapas más grandes y complejos (como vértigo).

Javier Ramirez Pulido

SOLUCION PLANTEADA PARA EL NIVEL 2

En este nivel, el número de detalles a tener en cuenta es mucho mayor que en el anterior, que simplemente pedía alcanzar un objetivo.

Para empezar, el algoritmo seleccionado para este nivel ha sido el A*, con la función Manhattan para el cálculo de distancias. En este, nos basamos en la heurística del problema combinada con el costo a la hora de seleccionar el próximo nodo. La implementación de este algoritmo nos conduce a la creación de dos atributos más para el struct *nodo* que serían "distancia" (que almacena la suma de la heurística y el costo) y "heurística" (que es la distancia desde nuestra posición actual hasta el destino, calculada con Manhattan).

Este algoritmo pilla los nodos a expandir de forma óptima porque heurística siempre es una función que queda por debajo del camino completo, ya que ignora los caminos que sabe que van a ser más largos podando partes del árbol de combinaciones posibles. Uno de los problemas enfrentados ha sido el tiempo consumido, debido seguramente a una implementación no del todo correcta. Tras investigación y consultas a compañeros de otros cursos o grados, obtuve una forma de mejorar la velocidad del cálculo del camino y es multiplicar la heurística por algún valor. Esto disminuye el tiempo de cálculo y evita aparentemente que la ejecución termine por la condición del tiempo de cálculo.

En este caso se vuelve a utilizar como estructura de datos una cola con prioridad, lo que nos hace modificar el *operator* < ya previamente modificado. La nueva condición es comparar el atributo *distancia* de los nodos. Si uno es menor que el otro, significa que estos tienen valores diferentes, por tanto, al menos uno de ellos no es 0, dando a entender que se utiliza el algoritmo A* y efectivamente se basaría en el costo+heurística para el orden. Si ambos valores son iguales, significará que han dado con el mismo costo+heurística casualmente o que ambos son 0 y es que no se están actualizando esos valores (el algoritmo usado no es A* por lo que no debemos basarnos en esto, sino en el costo). Sea cual sea el caso que se dé, nos basamos en el atributo costo para la prioridad en la cola.

Tras la implementación de este algoritmo, los caminos obtenidos son rigurosos a la hora de esquivar tanto césped como agua, lo cual hace que la batería no sea un problema mayor, pero sí algo importante a tener en cuenta. Por ello, es por lo que se ha implementado la recogida de objetos que reducen el consumo cada vez que sea conveniente. Esto consiste en que en el momento en el que el personaje "vea" (lo tenga en el vector sensores.terreno) el bikini o las zapatillas, haga un pequeño desvío, los coja y luego continúe por donde iba. Se pierde algo de batería yendo a por ellos, pero es infinitamente menor que la que se ahorra cuando viste estas prendas. Una vez tiene alguno de los elementos, no vuelve a hacer desvíos hacia ellos, esto ocurre solamente cuando no lo tenga aún.

Tambien, relacionado con el gasto de la batería, existen ciertas condiciones que llevan al agente a recargar esta en las casillas especiales para ello (la letra X). Una condición que tiene que darse para que esto ocurra es haber "visto" la batería previamente, pues en el nivel 2 el mapa es desconocido y las coordenadas de la zona de recarga no se conocen. Tras ver la

batería y almacenar su localización, se tienen en cuenta diferentes criterios. En un principio, los criterios eran iguales, pero con la aparición de unos mínimos básicos en el número de objetivos a alcanzar realicé algunos cambios basados en las dimensiones del mapa (las cuales cojo de la variable mapaResultado).

En el caso del mapa30, al ser pequeño y requerir como mucho una recarga o dos a lo largo de su ejecución, se prioriza cargar poca batería en cada vez y hacerlo únicamente cuando el nivel de esta sea realmente bajo, pues al ser un mapa pequeño no hay problemas en que llegue de una punta a la otra de este si fuese necesario.

En el caso del mapa50, las dimensiones son algo más grandes, por lo que la batería fijada como mínimo con la que partir hacia la recarga es mayor, intentado evitar quedarse a mitad de camino. En cada recarga se obtiene una cantidad de batería mayor que en el caso anterior para reducir al máximo el número de veces que necesite recargar (lo cual nos da batería, pero nos quita tiempo restante de ejecución, que es irrecuperable).

Ya en el mapa75 esta forma de evolucionar las condiciones dejó de permitir llegar a los objetivos mínimos para la nota máxima (en este caso 40). Tras varias pruebas, la única forma en la que conseguí cumplirlo es haciendo recargas aún más altas, lo cual, si mantiene la evolución de un mapa a otro, pero con la restricción de que cuando quede poco tiempo de ejecución se olvide de la batería (que por lo general llegaba sobrado a ese punto) y se centre en conseguir objetivos. El margen de batería que tiene para el regreso a esta para recargar también es algo mayor que en el anterior con miras a evitar lo mismo.

En los mapas de dimensiones de 100 hay dos casos. Los momentos en los que el tiempo de ejecución restante es mayor que 2000 (se presupone que a esas alturas aún no conoce mucho mapa) y en los que es menor que esa cantidad. Sea cual sea el caso, se le hace recargar bastante batería porque son mapas en los que se necesita ir bien cargado siempre. Ya, si nos encontramos en la fase en la que aun no tiene por qué conocer el mapa lo suficientemente bien, se le hace recargar más a menudo que en la otra fase. Es la principal diferencia entre estas.

No son ni mucho menos las mejores condiciones posibles para que se consiga el máximo número de requisitos, pero son casos en los que al menos una vez me ha dado el mayor objetivo pedido o más. El algoritmo es capaz de llegar a esos puntos que se piden, solamente tienen que darse algunas situaciones concretas relacionadas con la aparición aleatoria de objetivos sobre el mapa, cuya indeterminación impide comprobar con 100% de certeza que siempre te los vaya a hacer.

Una idea para reducir el número de viajes a recargar ha sido implementar que, al principio, cuando no conoce el mapa (en mapas de dimensión 75 y 100), recalcule a cada paso. Esto hace que, durante ese periodo de tiempo, aunque no conozca el mapa, no pise nada verde o azul, lo que le va a reservar una gran cantidad de batería para el resto de ejecución. También permite que el margen de batería necesario para que vaya a recargar sea menor, pues sabes que cuando tenga que ir ya habrá conocido muchísimo terreno esquivando costes altos. Además de esto, también recalcula aleatoriamente cuando el nivel de vida que queda es múltiplo de 8. Esto solo hace que cada 8 unidades de ejecución recalculo, pudiendo salvar al agente de

atravesar mucha zona costosa desconocida cuando se termine el periodo inicial en el que la esquiva. Esto puede hacer algo ineficiente el código, pero la multiplicación de la heurística ya le da la velocidad suficiente como para que estas comprobaciones aleatorias den más ventajas que desventajas.

Existían alternativas como empezar yendo a puntos aleatorios, diferentes a los destinos, solamente con la intención de explorar mapa o de ir a lugares cercanos de la batería para conocer sus alrededores y facilitar el acceso a la recarga. Estas ideas han supuesto problemas, pero han quedado comentadas en el código para el caso en el que sean útiles.

Como idea auxiliar a la de ir cuando no queda otra, está que, si pasas cerca de la batería, dependiendo de cuanta tenga el agente, le conviene recargar antes de continuar, lo cual puede aplazar la recarga obligatoria a instantes más tarde, cuando ya tenga más objetivos.

Ya para reducir los tiempos de ejecución gastados en cosas inútiles, está implementada una parte en la que se esquiva al aldeano encontrado en el camino. Esto ocurre de una forma muy básica que no siempre asegura que se esquive, pero sí en muchísimas ocasiones sirve para rodearlo o dejarle pasar por el lugar en el que estabas tú, lo cual evita tiempos de ejecución parados sin hacer nada.

Para terminar, otro caso finalmente descartado, pero igualmente implementado es el caso de ver unas zapatillas y en el camino encontrar un bikini o viceversa. Este código gestionaría el orden de acceso, controlaría que ambos destinos se cumplan y que tras estos se vuelva al trascurso regular. Fue desechada por el hecho de que no sé si ese caso se llega a dar alguna vez y es código que puede interferir en otras funciones que realiza el programa.

Finalmente, y sin aún tener muy claro como funcionan las coordenadas para las comprobaciones (en Windows pone X e Y en lugar de Filas y Columnas, además las coordenadas del destino estaban intercambiadas y en la ejecución del mapa50 en BelkanSG el destino introducido sale al revés, entre otras cosas un poco ambiguas), creo que mi programa puede satisfacer los destinos buenos en al menos 3 mapas de los 6 propuestos, y dependiendo de los objetivos incluso más.

A pesar de los inconvenientes que pueda haber tenido, los cuales son normales; valoro y felicito la difícil gestión de la asignatura en estas circunstancias.