Inteligencia Artificial

Documentación de la práctica 2: Agentes Reactivos/Deliberativos: Los extraños mundos de Belkan

Nivel 1: Algoritmo de búsqueda por anchura

Teniendo la búsqueda por profundidad, hacer la búsqueda por anchura es sencillo. Simplemente cambio la pila LIFO por una cola FIFO.

Se va iterando por todos el arbol, generando para cada nodo sus tres posibles hijos, es decir, los tres posibles estados que puede realizar Belkan: girar derecha, girar izquierda, ir hacia delante. Cada vez que se abre un nodo, se mete en la lista de abiertos, que como hemos dicho es una cola FIFO. Cuando se explora, se comprueba si es el nodo objetivo, y si no lo es, se cierra y se mete en la lista de cerrados, y se sigue al siguiente nodo de la lista de abiertos (que es el primero, al ser una cola FIFO).

El último nodo contendrá una secuencia de acciones, que son las acciones que se han recorrido en el árbol desde el nodo inicial hasta este, con lo que se genera el plan para llegar al objetivo. Además, se llama a la función que pinta el plan para poder verlo de forma gráfica.

Nivel 1: Algoritmo de búsqueda por costo uniforme

El algoritmo de búsqueda por costo uniforme sigue una idea parecida a la de anchura. He basado la solución en el siguiente pseudocódigo:

```
procedure UniformCostSearch(Graph, root, goal)
  node := root, cost = 0
  frontier := priority queue containing node only
  explored := empty set
  do
    if frontier is empty
      return failure
    node := frontier.pop()
    if node is goal
      return solution
    explored.add(node)
    for each of node's neighbors n
      if n is not in explored
        if n is not in frontier
          frontier.add(n)
        else if n is in frontier with higher cost
          replace existing node with n
```

Para la lista de abiertos, en este caso hay que usar una cola FIFO, pero por prioridad. Esta prioridad la marca el costo de los nodos, estando ordenados por menor costo (el algoritmo debe coger los nodos de menor coste). Con la función *CalculaCosto* se calcula el costo de un nodo, pasándole la fila y la columna en la que está: 5 si es bosque, 10 si es agua y 2 si es tierra, si no el costo es 1.

El contenedor utilizado para la lista de abiertos es un multimap. Es necesario eliminar los nodos antiguos una vez que se encuentra que ese nodo tiene otro costo mayor (no se pueden tener nodos repetidos), y si fuese una priority_queue de la stl no se podría, puesto que no tiene iteradores, y sería bastante engorroso buscar el nodo para borrarlo y poner el nuevo. Es un multimap porque puede haber dos nodos con el mismo costo, pero que sean distintos, y eso en un map normal no se podría meter. El tipo del multimap es nodo, y como está ordenado por el costo, de menor a mayor, entonces la clave es el costo, siendo este campo un int. Se ha añadido además un atributo costo al struct nodo para que el propio nodo conozca su costo y no tener que iterar hasta ese nodo en el multimap para saber su costo.

Al igual que antes, se va iterando por todos el arbol, cogiendo el primer valor de la lista de abiertos cada vez que se generan para cada nodo sus tres posibles hijos: girar derecha, girar izquierda e ir hacia delante. Para cada uno de los nodos se hace lo mismo. Se calcula su orientación, su costo (+1 si es solo girar, +el costo de la casilla a la que va si es moverse hacia delante) y entonces, si el nodo no se ha cerrado aún, se comprueba si ya existe en la lista de abiertos para sobreescribirlo por el actual si este tiene un costo más actualizado, es decir, un costo mayor. Un nodo es igual que otro si tiene misma fila, columna y orientación. En el caso de que este nodo aún no se haya abierto, se añade a la lista de abiertos igual que en los algoritmos anteriores.

Nivel 2: Agente reactivo/deliberativo complejo

El algoritmo usado en esta parte es el algoritmo de búsqueda por costo uniforme, puesto que es el más eficiente de los tres implementados. Los otros dos tardan más y dan soluciones menos buenas.

En la función *think*, que es la que implementa el comportamiento necesario para descubrir el mapa y poder recorrerlo de forma correcta, distingo entre dos partes: nivel 1 y nivel 2, mediante el sensor de nivel que tiene el agente para así evitar confusiones. Dentro del nivel 2 tenemos dos partes bien diferenciadas: una en la que aún el agente no conoce el mapa, y por tanto centrará sus esfuerzos en buscar un PK para saber donde está, y otra en la que ya sabe donde está y se limita a pintar el mapa conforme anda y a realizar una búsqueda de la misma manera que en el nivel 1 (costo uniforme en este caso).

Antes de distinguir entre esos dos casos, se actualizan los efectos de la última acción y se comprueba si el destino ha cambiado, igual que se hizo en el tutorial al principio de la práctica. Es entonces cuando se distingue entre los dos casos: - En el principio lógicamente el agente no estará en un punto de referencia, por

tanto mientras haga acciones ejecutará un comportamiento totalmente reactivo como el del tutorial, en el que simplemente evitará los obstáculos y evitará morir. Una vez que encuentre el punto de referencia, activará el flag *puedoDescubrir*, a partir del cual ya empezará a pintar el mapa, y seguirá así hasta el final de la ejecución. - Una vez que pueda descubrir, lo primero que hará será empezar a pintar el mapa simplemente iterando sobre el vector de superficie y rellenando con los valores que contiene en el *mapaResultado*. Es importante aclarar que una vez que encuentra un punto de referencia, su siguiente acción será quedarse quieto, para así en la siguiente acción poder rellenar bien el mapa y no rellenarlo mal con una casilla desplazada.
Se calcula un plan hacia el destino, al igual que en el nivel 1, con la excepción de que cada 3 pasos el agente recalculará el plan (desactiva el flag de que *hayPlan*) para así maximizar el encontrar mejores soluciones conforme se va descubriendo el mapa y evitar crear planes a través de sitios por los que no se puede pasar ya que están inexplorados aún, como muros. En el caso de que lo que el agente tenga delante sea un aldeano, no cogerá el plan (no lo borrará de la lista de acciones) y en su lugar la acción que ejecutará será quedarse quieto hasta que el aldeano se vaya. Por último, se tienen en cuenta los precipicios, para evitar que el agente cree un plan por ellos. Simplemente, si detecta que tiene precipicio delante, recalcula el plan y gira a la derecha por defecto, ya que si sigue hacia delante morirá.