UNIVERSIDAD DE GRANADA

ETSIIT
Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación

Práctica 2: Agentes Reactivos/Deliberativos: Los Extraños Mundos de BelKan

Inteligencia Artificial

Grupo C1

Autor:
Lugli, Valentino Glauco · YB0819879

1. Nivel 0 y Nivel 1

La realización del Nivel 0 fue seguida en su totalidad por el tutorial provisto.

En el Nivel 1 se implementó algoritmo de Búsqueda en Anchura. Como este algoritmo es muy similar al de Búsqueda en Profundidad, se tomó éste como base y se modificó, sobretodo, cambiando que la estructura de datos que mantiene a los nodos abiertos sea una cola en vez de una pila, de esta manera se pueden visitar todos los nodos de un nivel del árbol antes de visitar los nodos hijos que están en el siguiente nivel, siempre verificando que el nodo generado no se encuentre en cerrados.

Se toma como "expandir" un nodo, como un movimiento de los posibles del agente, es decir moverse hacia delante, girar a la derecha y a la izquierda de una posición en particular se toman como la expansión del nodo padre que sería la posición actual con una orientación en particular; dos nodos son considerados idénticos si poseen la misma fila, columna y orientación.

Para este nivel no se vio la necesidad de implementar nuevos métodos o atributos de la clase.

2. Nivel 2

Para el Nivel 2 se decidió implementar el algoritmo de búsqueda A*, para poder adaptar el algoritmo al problema se expandió la clase **`nodo`**, creándose otra llamada **`aStarNode`**, esta clase almacena además de las coordenadas y orientación del nodo y el camino que lo ha llevado hasta allí, también el coste total de ese nodo, el valor de la heurística y del costo ya acumulado además también mantiene si en ese nodo se tiene puesto el bikini y las zapatillas.

Para la heurística, se utilizó la Distancia Manhattan debido a que se está trabajando con un mundo cuadriculado dónde ir en diagonal es más costoso. Es una heurística admisible porque el costo que predice es el de ir en línea recta o de hacer giros de 90º grados hacia el objetivo por una casilla con el coste mínimo de 1 unidad de batería.

La estructura de datos que mantiene los nodos abiertos es un **`std::multiset`** de **`aStarNode`** que compara cada nodo por el coste mínimo, para que sea más optimizada la ordenación pero que se permita acceder a los miembros para comprobar si un nodo ya se encuentra en la estructura de datos por medio de iteradores ya que realizar una búsqueda utilizando el método de la estructura de datos no es factible debido a que el comparador de ordenación es el coste y no otras características del nodo. Realizar esto con una **`std::priority_queue`** es mucho más engorroso. La lista de cerrados se mantiene también como un **`std::multiset`** ya que las búsquedas son eficientes con la función de comparación.

Para la función de comparación de cerrados, además de comparar el estado, se compara si los nodos poseen bikinis y si poseen zapatillas.

En este nivel se definieron dos miembros de la clase para mantener si al momento que se llama el A*, se tienen ya puesto el bikini o las zapatillas.

3. Nivel 3

Para el Nivel 3 se modificó el algoritmo A* para que realice la búsqueda del camino más óptimo de 3 objetivos (la implementación soporta *n* objetivos en teoría). La diferencia con el A* original, es que, los nodos almacenan también una lista de objetivos a los que les falta por ir, se utiliza la misma clase **`aStarNode`** ya que esta clase tiene también un miembro que almacena la lista de objetivos del nodo actual, este miembro no se utiliza si se realiza el A* normal.

Además, cuando es momento de generar los hijos, en ciertas ocasiones no se generan los usuales 3 hijos, sino 9 hijos. ¿Cuándo? Se generan cuando se está en el nodo raíz, al principio de las iteraciones de esta manera se tendrán 3 tríos de hijos que cada uno está yendo a un primer objetivo diferente (entiéndase que los nodos se dirigen al objetivo que poseen al frente de la lista de objetivos que poseen). Se vuelven a generar más hijos cuando algún nodo llega al objetivo que tiene actualmente, una vez lo quitan de su lista de objetivos, los hijos vuelven a generarse cada trío con un objetivo diferente de los que restan, así hasta que algún nodo llega a los 3 objetivos, debido al uso del **`std::multiset`** para la ordenación se asegura que ese es el nodo con el coste óptimo. Al principio son 9 hijos, luego son 6 cuando un nodo llega al segundo objetivo que tenga, luego cuando llega al último objetivo termina el bucle porque ha encontrado el camino a los tres.

Para que la lista de cerrados funcionase, ahora se ha cambiado la función de comparación, para que además de tener en cuenta el estado, si posee bikini o zapatillas, tome en cuenta qué objetivos posee cada nodo y en qué orden.

Para la heurística, nuevamente se utiliza la distancia Manhattan, pero esta vez esta distancia es la suma de la distancias Manhattan de las coordenadas del nodo hacia el primer objetivo y de ese objetivo al segundo y del segundo al tercero, en el orden en que se encuentran en la lista.

4. Nivel 4

Para el Nivel 4 se reutiliza el algoritmo A* utilizado en el Nivel 2.

4.1. Consideraciones generales

- Un registro **`objective`** que tiene dentro el **`estado`** dónde está un objetivo y su distancia Manhattan al Agente, de esta manera se pueden ordenar por su cercanía al mismo. También se utiliza este registro para almacenar las estaciones de recarga que estén en el mapa.
- Un registro **`enemy`** que tiene dentro el **`estado`** de dónde está un Aldeano y la posición en el vector de **`sensores.superficie`**
- Un miembro de la clase que mantiene la batería actual del agente, **`batteryLevel`**
- Un miembro de la clase que mantiene los objetivos actuales, **`actualObjectives`**, es diferente del miembro original **`objetivos`**, ya que el miembro original en cada "tick" de ejecución obtiene los 3 objetivos que se encuentran en el mapa, esta lista difiere porque se va reduciendo conforme el agente pasa por los objetivos y además está ordenada con el objetivo primero más cercano al agente; una vez que está vacía se copian los 3 nuevos objetivos de la lista original.

- Se tiene un método **clearFog** la cual se encarga de tomar la información de **sensores.terreno** y con ella actualizar el mapa, además devuelve un booleano dependiendo si se está rellenando una sección nueva del mapa, de esta manera el agente puede replanificar si al descubrir una sección nueva del mapa se encuentra con Agua, Bosques, Muros o Precipicios
- Se tiene un método **checkForEnergyStations** el cual se llama al detectar que el nivel de batería es menor de 1500 unidades. Esta función escanea el mapa como está actualmente en búsqueda de estaciones de recarga, que almacena como **struct objective** dentro de la lista **energyStations**, se ordenan también por su cercanía al agente y es allí dónde el agente replanificará la ruta para ir. Una vez el agente ha recargado 2000 unidades, retoma los objetivos pendientes. Si el Agente no consigue en ese "tick" del mapa estaciones, continúa con los objetivos hasta que aparezca una estación o se quede sin batería.
- Se tiene una función **checkIfOnObjective** que es la que se encarga de quitar los objetivos del mapa, se llama en cada "tick" ya que es posible que en la ruta hacia una estación de recarga, el agente pase por un objetivo, de esa manera se logra que el Agente lo tenga en cuenta y ya cuando tenga baterías sabe que ya lo ha obtenido y va por otro.
- Se tiene un método **checkEnemies** el cual está analizando los **sensores.superficie** en búsqueda de aldeanos latosos cerca del Agente, almacena en el vector **enemiesNearby** su posición en el mapa calculada dependiendo de en qué posición del sensor fue detectado el Aldeano.
- El método **evadeEnemies** es llamado si se encontraron aldeanos cerca del Agente, esta función permite al Agente esquivar los aldeanos ya que hace que en ese "tick" de ejecución, las posiciones dónde estén los Aldeanos sean tomadas como un muro, de esta manera si un Agente entra en colisión con un aldeano, se replanificará la ruta y el Agente le pasará por un lado al Aldeano. Esto se realizó así debido a que los Aldeanos no posan un peligro mortal al Agente, simplemente estorban, por lo que hacer contacto con ellos no es algo que se evite activamente, además, así se reducen las llamadas al A*, que puede ralentizar mucho la exploración si se está en un mapa muy grande.
- Debido a esto, se tiene también un método **restoreMap** el cual restaura el mapa a como estaba, es decir, donde estaba un aldeano, vuelve a colocar lo que el **sensor.terreno** detecta, por esto es que se almacena la posición en la que el aldeano fue detectado, se utiliza para restaurar el mapa en ese sitio exacto.
- El método **think** posee una sección que se activa únicamente con el nivel 4 dónde se encuentran la lógica descrita anteriormente. El Agente obtiene objetivos, se va detectando si la batería se reduce, si se encuentra algún aldeano enfrente, etc. Se optó por modificar ligeramente el método **pathFinding** para tener un parámetro independiente que sea el objetivo que será utilizado por el A*

4.2. Consideraciones para el A*

- Se optó que el agente sea "curioso", es decir, preferirá investigar partes desconocidas del mapa si se encuentra que lo que está conocido posee un coste muy alto, por esto, las casillas marcadas como **?** tienen un coste de 1.

- Como se sabe que los mapas siempre están rodeados de 3 casillas de precipicios o bien, muros, en el constructor del Agente para el nivel 4, estas casillas se rellenan automáticamente con precipicios, de esta manera también se evita que el agente curioso dé muchas vueltas pensando que puede encontrar un mejor camino y se termina encontrando siempre con un precipicio, que si bien no muere, replanifica mucho. Se sabe que no hay mapas que no cumplan esta condición porque, experimentalmente se creó un mapa sin obstáculos en los bordes y se obtuvo que ocurriría un error de segmentación porque los sensores estarían leyendo información fuera de las barreras del mapa.