Primera practica inteligencia Artificial

CERCA INFROMADA

Angelina Ruiz

Curso 2023/24

Profesor: Benet Manzanares

Índice

Código

El código implementado para esta practica se encuentra o bien en la carpeta de entrega o en el siguiente repositorio: https://github.com/angeruiizz/Primera-Practica-IA

Formalización del problema

Estado

Cada estado se define con las coordenadas de la situación en el mapa, representado con eje x y eje y. Además contiene información del coste y días acumulados.

Estado inicial

El estado inicial es el estado en el que el comerciante empieza el viaje. En este caso siempre empezara en la casilla 0,0, es decir el estado que contiene las coordenadas 0,0 y el total de monedas y días acumulados es zero.

Estado final

El estado final es el estado en el que el comerciante acaba el viaje. En este caso siempre acaba en la casilla cuyas coordenadas son 9,9 y el acumulado de monedas y días es superior a 0.

Operadores

Los operadores son los posibles movimientos que puede hacer el comerciante para desplazarse a la derecha, izquierda, abajo o arriba. Siempre y cuando la casilla no sea una montaña. Hay que tener en cuenta que por cada movimiento que hace tiene un coste de 5 monedas, pero el comerciante puede recuperar según la coordenada del siguiente estado.

Solución

La solución es el camino de estados que va desde el estado inicial hasta el estado final.

Coste a optimizar

El coste optimo es el camino que menos coste supone al comerciante. En este caso la solución optima es 33 monedas oros.

Heurísticas

Heurística 1

La primera heurística calcula la distancia euclidiana entre el estado actual y el estado final. Esta distancia se basa en el cálculo del teorema de Pitágoras.

Es decir que es la diferencia de cada coordenada (en el eje de las x y en el eje de las y) y luego aplicando el teorema de Pitágoras ($a^2 + b^2 = c^2 \rightarrow \sqrt{a^2 - b^2}$) .

De esta forma obtenemos la distancia euclidiana, que es la longitud de línea recta que une dos puntos en el plano.

En este caso, la solución es admisible pero no optima. Ya que la distancia calculada es real o más pequeña que la real (lo que cumple con la condición para que sea admisible) pero no es optima ya que el coste es superior a 33.

No es admisible respecto el dinero ya que no tiene en cuenta el coste del viaje si no la distancia

El código correspondiente es:

```
public static float Heuristic1(State currentState, State targetState,
float[][] map) {

        float x = currentState.getPosX() - targetState.getPosX();
        float y = currentState.getPosY() - targetState.getPosY();
        return (float) Math.sqrt(x * x - y * y); //hipotenusa
    }
```

Heurística 2

La segunda heurística implementada se basa en el calculo de la distancia Manhattan. Esta distancia básicamente es la distancia mínima que hay entre dos puntos del mapa.

Esta heurística es admisible y optima ya que la distancia es real o menor que la distancia real, y es óptima respecto el coste ya que el resultado es 33 que es el menor coste posible.

Pero no es admisible respecto el coste ya que no se tiene en cuenta a la hora de hacer el cálculo.

El código correspondiente a esta heurística és:

```
    public static float Heuristic2(State currentState, State targetState,
float[][] map) {
        return (Math.abs(currentState.getPosX() - targetState.getPosX())
              + Math.abs(currentState.getPosY() - targetState.getPosY()));
    }
```

Heurística 3

La última heurística implementada es una mezcla de la distancia manhattan y añadiendo el coste acumulado y real.

De esta forma, podemos estimar el coste del estado actual hasta el objetivo, ayudando al algoritmo a priorizar esos estados que están más cerca del objetivo y tienen un coste menor.

Esta heurística es admisible respecto distancia y coste. Por un lado tiene en cuenta distancia real (con el cálculo de la distancia manhattan) y por otro al obtener el coste acumulado más el coste que asumirá al escoger esa posición del tablero, es un coste real.

Pese a esto no es la solución optima ya que el resultado es mayor a 33.

La implementación del código es la siguiente;

```
    public static float Heuristic3(State currentState, State targetState,
float[][] map) {
                float dist = Math.abs(currentState.getPosX() -
targetState.getPosX())      +      Math.abs(currentState.getPosY()      -
targetState.getPosY());
        float cost = map[currentState.getPosX()][currentState.getPosY()];
//Coste de esa posición en mapa
```

```
         float costAcu = currentState.getContOr(); //Coste acumulado del
esatdo actual
        return dist + cost + costAcu;
    }
}
```

Decisiones de diseño

Programa principal

Para poder ejecutar el programa se ha creado un pequeño menú principal para poder seleccionar heurística deseada y tipo de búsqueda o de la ejecución de todo el programa.

Para poder llamar a los correspondientes métodos se ha seguido el guion y código base proporcionado.

Para poder imprimir el resultado, se hace un bucle recorriendo la lista con los estados y obteniendo ambas coordenadas de cada uno de ellos printando los por pantalla. Además de imprimir el coste acumulado del estado final y los días (es decir el número de estados que tiene el camino).

Algoritmos de búqueda

Para ambos algoritmos de búsqueda se ha seguido el pseudocódigo proporcionado en clase.

Ambos algoritmos extienden de la clase Search. Gracias a esto, se consigue que cada uno tenga su correspondiente implementación para la búsqueda pero comparten la forma de obtener los sucesores (es decir el método EvaluateOperators).

Este método básicamente busca 4 posibles sucesores; arriba, abajo, izquierda y derecha. Para ello comprueba que la casilla no este en el limite (por tal de no obtener posiciones inexistentes) y que no sea una montaña.

Para saber que no es una montaña, miro el coste del posible sucesor y si es menor que 100 significa que es cualquier casilla a excepción de la montaña.

Luego simplemente actualizo la información de este y lo añado a la lista de posibles sucesores. Es decir, obtengo el gasto de esa casilla (ósea hago 5 – el coste de la casilla) y se lo sumo al coste acumulado, además de sumar un día al contador de días.

Best-first

Para el primer algoritmo de búsqueda, utilizo tres listas; la lista de pendientes, de tratados y el resultado. La lista de pendientes se encarga de almacenar los posibles sucesores que están por tratar, la lista de tratados són aquellos que van formando el camino y la lista resultado es la que forma el camino correspondiente a la solución.

Básicamente la lista ‘resultado’ lo obtenemos recorriendo la lista de tratados a través del puntero que tiene cada estado apuntando a su padre/ anterior. Y luego utilizamos la librería collections, para girar los estados de tal manera que los primeros de la lista sean los últimos, correspondiendo al orden correcto del camino.

El algoritmo best-first sigue la siguiente lógica;

Se añade el primer nodo a la lista de pendientes y empieza un bucle que no para hasta que se encuentre o bien la lista de pendientes este vacía (con lo que habrá recorrido todo los estados) sin encontrar la solución.

A continuación, ordena la lista de pendientes según la heurística, esto es fundamental para que la lista de pendientes este actualizada. De esta forma nos aseguramos que los siguientes estados pendientes son los sucesores del estado que estamos tratando.

Y seleccionamos el estado actual, que es el estado con el que obtendremos los siguientes sucesores. Para obtener este estado, se usa la función remove(), que saca el último estado añadido recientemente de la lista de pendientes.

Seguidamente comprueba que el estado que estamos tratando no sea el final y busca sus hijos mediante la función evaluateOperators (explicada en el anterior apartado). Una vez obtenemos los posibles sucesores, añadimos la referencia del padre (anterior) a cada uno de ellos mediante un bucle for y los añadimos a la lista de pendientes.

Una vez ya ha encontrado el estado final, el bucle inicial finaliza y empieza otro para recorrer la lista de tratados y encontrar la solución, tal y como he explicado anteriormente.

A*

El segundo método de búsqueda, el A* es muy similar al anterior.

Siguiendo la misma mecánica que en el best-first, la parte diferente a este es a la hora de mirar posibles sucesores, ya que podemos encontrar otros caminos más económicos o con menos distancia (según la heurística).

Después de obtener la lista de posibles sucesores, analizamos cada sucesor. Si no ha estado tratado anteriormente tenemos dos opciones:

La primera es que este sucesor no este en la lista de pendientes, con lo que igual que antes añadiríamos la referencia del anterior estado y lo añadiríamos a la lista de pendientes.

Y la segunda, que es la gracia de este algoritmo, si esta en la lista de pendientes trata de mirar si desde la posición en la que esta ese nodo dentro de la lista de pendientes su heurística + el coste acumulado es mayor que ese mismo estado encontrado con otro padre. Si esto sucede, quitamos de la lista de pendientes al sucesor, actualizamos su nuevo padre y lo volvemos a añadir a la lista de pendientes.

Luego sigue igual que el anterior algoritmo, el bucle se para hasta encontrar el estado final y se recorre la lista de tratados vía los punteros a los padres de cada estado obteniendo la solución.

Esta vez, para ordenar la lista de pendientes, he decidido implementar la función Quicksort utilizada en otra asignatura y adaptando para este caso. Resumidamente la función Quicksort se encarga en partir la lista de pendientes, y recursivamente la función hasta que ordene la lista. Cuando se parte la lista, se selecciona un ‘pivote’ que en este caso es el último elemento, y se recorre la parte de la lista comparándolo con el pivote. Una vez recorremos toda la parte correspondiente, podemos intercambiar el pivote de tal forma que tendremos a la izquierda los estados cuya heurística es menor que el pivote y a la derecha el valor mayor. Esto se va repitiendo hasta conseguir que la lista esta ordenada.

Pruebas

Mapa 1

Best-first

Heurística 1

- Solución:
  (0,1)(1,1)(1,2)(2,2)(2,3)(3,3)(3,4)(4,4)(5,4)(5,5)(6,5)(6,6)(6,7)(7,7)(7,8)(8,8)(8,9)(9,9)
  - Tiempo: 18 días
  - Coste: 42,5 monedas

Heurística 2

- Solución:
  (0,1)(0,2)(0,3)(0,4)(0,5)(1,5)(2,5)(3,5)(3,4)(4,4)(5,4)(5,5)(6,5)(6,6)(6,7)(6,8)(6,9)(7,9)(8,9)(9,9)
  - Tiempo: 20 días
  - Coste: 33,0 monedas

Heurística 3

- Solución:
  (0,1)(0,2)(0,3)(1,3)(2,3)(3,3)(4,3)(5,3)(6,3)(7,3)(8,3)(9,3)(9,4)(9,5)(9,6)(9,7)(9,8)(9,9)
  - Tiempo: 18 días
  - Coste: 62,5 monedas

A*

Heurística 1

- Solución:
  (1,0)(2,0)(3,0)(4,0)(5,0)(6,0)(7,0)(8,0)(9,0)(9,1)(9,2)(9,3)(9,4)(9,5)(9,6)(9,7)(9,8)(9,9)
  - Tiempo: 18 días
  - Coste: 53,5 monedas

Heurística 2

- Solución:
  (0,1)(0,2)(0,3)(0,4)(0,5)(1,5)(2,5)(3,5)(3,4)(4,4)(5,4)(5,5)(6,5)(6,6)(6,7)(6,8)(6,9)(7,9)(8,9)(9,9)
  - Tiempo: 20 días
  - Coste: 33,0 monedas

Heurística 3

- Solución:
  (0,1)(0,2)(0,3)(1,3)(2,3)(3,3)(4,3)(5,3)(6,3)(7,3)(8,3)(9,3)(9,4)(9,5)(9,6)(9,7)(9,8)(9,9)
  - Tiempo: 18 días
  - Coste: 62,5 monedas

Análisis

Como podemos observar, la heurística 2 es la que nos da la solución óptima en este mapa. En ambos recorridos obtenemos el mismo tiempo y monedas, es el mismo recorrido.

Sin embargo, las otras dos heurísticas nos ofrecen menos tiempo, pero más coste. Además, que según el tipo de cerca utilizado la heurística 1 varia, siendo mucho más económica el best First.

Mapa 2

El mapa que se ha utilizado es:

```
    { 'N', 'P', 'N', 'N', 'N' },
    { 'A', 'C', 'M', 'A', 'N' },
    { 'N', 'M', 'M', 'C', 'C' },
    { 'N', 'C', 'N', 'N', 'C' },
    { 'N', 'N', 'P', 'C', 'N' },
```

Best-first

Heurística 1

- Solución: (0,1)(0,2)(0,3)(1,3)(2,3)(3,3)(3,4)(4,4)
  - Tiempo: 8 días
  - Coste: 30,0 monedas

Heurística 2

- Solución: (0,1)(0,2)(0,3)(0,4)(1,4)(2,4)(3,4)(4,4)
  - Tiempo: 8 días
  - Coste: 31,5 monedas

Heurística 3

- Solución: (1,0)(2,0)(3,0)(4,0)(4,1)(4,2)(4,3)(4,4)
  - Tiempo: 8 días
  - Coste: 28,0 monedas

A*

Heurística 1

- Solución: (1,0)(2,0)(3,0)(4,0)(4,1)(4,2)(4,3)(4,4)
  - Tiempo: 8 días
  - Coste: 28,0 monedas

Heurística 2

- Solución: (0,1)(0,2)(0,3)(0,4)(1,4)(2,4)(3,4)(4,4)
  - Tiempo: 8 días
  - Coste: 31,5 monedas

Heurística 3

- Solución: (1,0)(2,0)(3,0)(4,0)(4,1)(4,2)(4,3)(4,4)
  - Tiempo: 8 días
  - Coste: 28,0 monedas

Análisis

En este caso, no sigue igual que en el primer mapa. La heurística que marca el camino optimo es la 3 en ambos tipos de búsqueda. Además, podemos observar que la heurística 1 según el tipo de búsqueda hace un recorrido u otro.

Algoritmo Hill Climbing

El algoritmo Hill Climbing se diferencia de los otros dos porque ahorra coste de almacenamiento ya que solo mira los hijos del estado actual y sigue si los sucesores son mejores que el padre, si no se detiene.

En el caso de esta práctica, desde mi punto de vista si que considero que es capaz de llegar a la solución con las tres heurísticas. Pesé a que esta búsqueda no mira todas las posibilidades y tampoco garantiza encontrar la solución, al ser en los dos mapas un espacio de búsqueda pequeño en ambos casos encontrara solución.