代码

1.首先构造坐标系使点有x,y两个属性，并重写equals方法使得后续易于判断结点是否是终点

1. public class Coord {
2. public int x;
3. public int y;
4. public Coord(int x, int y)
5. {
6. this.x = x;
7. this.y = y;
8. }
9. @Override
10. public boolean equals(Object obj)
11. {
12. if (obj == null) return false;
13. if (obj instanceof Coord)
14. {
15. Coord c = (Coord) obj;
16. return x == c.x && y == c.y;
17. }
18. return false;
19. }
20. }

2.赋予结点更多的属性，构建类似于链表的指针域的东西，方便最后回溯路径时用队列找上一个结点。

G表示起点到当前结点的cost，H表示当前结点到目的结点的估计代价

并重写compareTo函数，方便open队列（储存周围所以可以走的结点）的结点的cost比较

1. public class Node implements Comparable<Node>
2. {
3. public Coord coord; *// 坐标*
4. public Node parent; *// 父结点*
5. public double G; *// G：是个准确的值，是起点到当前结点的代价*
6. public double H; *// H：是个估值，当前结点到目的结点的估计代价*
7. public Node(int x, int y)
8. {
9. this.coord = new Coord(x, y);
10. }
11. public Node(Coord coord, Node parent, double g, double h)
12. {
13. this.coord = coord;
14. this.parent = parent;
15. G = g;
16. H = h;
17. }
18. @Override
19. public int compareTo(Node o)
20. {
21. if (o == null) return -1;
22. if (G + H > o.G + o.H)
23. return 1;
24. else if (G + H < o.G + o.H) return -1;
25. return 0;
26. }
27. }

3.生成整个地图的类，并得知相关数据

1. public class MapInfo {
2. public int[][] maps; *// 二维数组的地图*
3. public int width; *// 地图的宽*
4. public int hight; *// 地图的高*
5. public Node start; *// 起始结点*
6. public Node end; *// 最终结点*
8. public MapInfo(int[][] maps, int width, int hight, Node start, Node end)
9. {
10. this.maps = maps;
11. this.width = width;
12. this.hight = hight;
13. this.start = start;
14. this.end = end;
15. }
16. }

4.Astar算法实现

在地图上未走过的结点为0，障碍物为1，走过的路径为2

并用ArrayList（）存储走过的路径，PriorityQueue（）存储所有可以走的点

1. public class AStar {
2. public final static int BAR = 1; *// 障碍值*
3. public final static int PATH = 2; *// 路径*
4. public final static double DIRECT\_VALUE = 1; *// 横竖移动代价*
5. public final static double OBLIQUE\_VALUE = 1.4; *// 斜移动代价*
7. Queue<Node> openList = new PriorityQueue<Node>(); *// 优先队列(升序)*
8. List<Node> closeList = new ArrayList<Node>();

知道地图的相关数据，并在openList添加起始点，并通过moveNodeshan使当前点选择下一个节点

1. */\*\**
2. \* 开始算法
3. \*/
4. public void start(MapInfo mapInfo)
5. {
6. if(mapInfo==null) return;
7. *// clean*
8. openList.clear();
9. closeList.clear();
10. *// 开始搜索*
11. openList.add(mapInfo.start);*//在openList添加起点*
12. moveNodes(mapInfo);
13. }
14. */\*\**
15. \* 移动当前结点
16. \*/
17. private void moveNodes(MapInfo mapInfo)
18. {
19. while (!openList.isEmpty())
20. {
21. Node current = openList.poll();*//* 删除第一个元素,并返回这个元素
22. closeList.add(current);
23. addNeighborNodeInOpen(mapInfo,current);
24. if (isCoordInClose(mapInfo.end.coord))*//*判断是否找到终点
25. {
26. drawPath(mapInfo.maps, mapInfo.end);
27. break;
28. }
29. }
30. }

绘制路径的函数：通过closelist和parent找到上一个结点，将其中的值都改为2

1. */\*\**
2. \* 在二维数组中绘制路径
3. \*/
4. private void drawPath(int[][] maps, Node end)
5. {
6. if(end==null||maps==null) return;
7. System.out.println("总代价：" + end.G);
8. while (end != null)
9. {
10. Coord c = end.coord;
11. maps[c.y][c.x] = PATH;*//地图对应的位置改为2*
12. end = end.parent;*//?*
13. }
14. }

通过两次重写函数把附近的点都加进openlist，第一次使用较为上层的方式写现结点的周围8个点， 第二个则为底层的算出结点的G和H，将G增加的值加入openlist（含是重点的特殊情况）（G越大，表明离起点越远，离终点靠近）

1. */\*\**
2. \* 添加所有邻结点到open表
3. \*/
4. private void addNeighborNodeInOpen(MapInfo mapInfo,Node current)
5. {
6. int x = current.coord.x;
7. int y = current.coord.y;
8. *// 左*
9. addNeighborNodeInOpen(mapInfo,current, x - 1, y, DIRECT\_VALUE);
10. *// 上*
11. addNeighborNodeInOpen(mapInfo,current, x, y - 1, DIRECT\_VALUE);
12. *// 右*
13. addNeighborNodeInOpen(mapInfo,current, x + 1, y, DIRECT\_VALUE);
14. *// 下*
15. addNeighborNodeInOpen(mapInfo,current, x, y + 1, DIRECT\_VALUE);
16. *// 左上*
17. addNeighborNodeInOpen(mapInfo,current, x - 1, y - 1, OBLIQUE\_VALUE );
18. *// 右上*
19. addNeighborNodeInOpen(mapInfo,current, x + 1, y - 1, OBLIQUE\_VALUE );
20. *// 右下*
21. addNeighborNodeInOpen(mapInfo,current, x + 1, y + 1, OBLIQUE\_VALUE );
22. *// 左下*
23. addNeighborNodeInOpen(mapInfo,current, x - 1, y + 1, OBLIQUE\_VALUE );
24. }
25. */\*\**
26. \* 添加一个邻结点到open表
27. \*/
28. private void addNeighborNodeInOpen(MapInfo mapInfo,Node current, int x, int y, double value)
29. {
30. if (canAddNodeToOpen(mapInfo,x, y))*//判断点是否可以走*
31. {
32. Node end=mapInfo.end;
33. Node start=mapInfo.start;
34. Coord coord = new Coord(x, y);
35. double G = calcH(start.coord,coord); *// 计算邻结点的G值（起点到当前点*
36. Node child = findNodeInOpen(coord);*//cood是否在open列表里*
37. if (child == null)
38. {
39. double H=calcH(end.coord,coord); *// 计算H值（当前点到重点*
40. if(isEndNode(end.coord,coord))
41. {
42. child=end;
43. child.parent=current;
44. child.G=calcH(start.coord,end.coord);
45. child.H=0;
46. }
47. else
48. {
49. child = new Node(coord, current, G, H);
50. }
51. openList.add(child);
52. }
53. else if (child.G > G)
54. {
55. child.G = G;
56. child.parent = current;
57. openList.add(child);
58. }
59. }
60. }

以下是上述函数进行比较还需的底层函数

1. */\*\**
2. \* 从Open列表中查找结点
3. \*/
4. private Node findNodeInOpen(Coord coord)
5. {
6. if (coord == null || openList.isEmpty()) return null;
7. for (Node node : openList)
8. {
9. if (node.coord.equals(coord))
10. {
11. return node;
12. }
13. }
14. return null;
15. }
16. */\*\**
17. \* 计算H的估值：“欧几里得”法，坐标分别取差值相加
18. \*/
19. private double calcH(Coord end,Coord coord)
20. {
21. return (Math.sqrt(Math.pow((end.x - coord.x),2) + Math.pow((end.y - coord.y),2) ))\* DIRECT\_VALUE;
22. }
24. */\*\**
25. \* 判断结点是否是最终结点
26. \*/
27. private boolean isEndNode(Coord end,Coord coord)
28. {
29. return coord != null && end.equals(coord);
30. }
31. */\*\**
32. \* 判断结点能否放入Open列表
33. \*/
34. private boolean canAddNodeToOpen(MapInfo mapInfo,int x, int y)
35. {
36. *// 是否在地图中*
37. if (x < 0 || x >= mapInfo.width || y < 0 || y >= mapInfo.hight) return false;
38. *// 判断是否是不可通过的结点*
39. if (mapInfo.maps[y][x] == BAR) return false;
40. *// 判断结点是否存在close表*
41. if (isCoordInClose(x, y)) return false;
42. return true;
43. }
44. */\*\**
45. \* 判断坐标是否在close表中
46. \*/
47. private boolean isCoordInClose(Coord coord)
48. {
49. return coord!=null&&isCoordInClose(coord.x, coord.y);
50. }
51. */\*\**
52. \* 判断坐标是否在close表中
53. \*/
54. private boolean isCoordInClose(int x, int y)
55. {
56. if (closeList.isEmpty()) return false;
57. for (Node node : closeList)
58. {
59. if (node.coord.x == x && node.coord.y == y)
60. {
61. return true;
62. }
63. }
64. return false;
65. }
66. }

结果实现

1. public class Test extends AStar {
2. public static void main(String[] args)
3. {
4. int[][] maps = {
5. { 0, 0, 0, 0, 0, 0, 0, 0 },
6. { 0, 0, 0, 0, 0, 0, 0, 0 },
7. { 0, 0, 0, 1, 0, 0, 0, 0 },
8. { 0, 0, 0, 0, 0, 0, 0, 0 },
9. { 0, 0, 0, 0, 0, 0, 0, 0 },
10. { 0, 0, 0, 0, 1, 1, 1, 1 },
11. { 0, 0, 0, 0, 0, 0, 0, 0 },
12. { 0, 0, 0, 0, 0, 0, 0, 0 }
13. };
14. MapInfo info=new MapInfo(maps,maps[0].length, maps.length,new Node(7, 0), new Node(0, 4));
15. new AStar().start(info);
16. //new Dijkstra().start(info);
17. //new Bfs().start(info);
18. printMap(maps);
19. }
21. /\*\*
22. \* 打印地图
23. \*/
24. public static void printMap(int[][] maps)
25. {
26. for (int i = 0; i < maps.length; i++)
27. {
28. for (int j = 0; j < maps[i].length; j++)
29. {
30. System.out.print(maps[i][j] + " ");
31. }
32. System.out.println();
33. }
34. }
35. }

文本

中度可信度描述已自动生成

Dijkstra算法

相比于Astar算法，Dijkstra算法无启发式即其G和H的计算方式不同，修改相关代码即可

1. private void addNeighborNodeInOpen(MapInfo mapInfo,Node current, int x, int y, double value)
2. {
3. if (canAddNodeToOpen(mapInfo,x, y))
4. {
5. Node end=mapInfo.end;
6. Coord coord = new Coord(x, y);
7. double G = current.G + value; *// 计算邻结点的G值*
8. Node child = findNodeInOpen(coord);

G = 已经走过的cost， H = 下一步花费的cost

1. */\*\**
2. \* 计算H的估值：“曼哈顿”法，坐标分别取差值相加
3. \*/
4. private double calcH(Coord end,Coord coord)
5. {
6. return (Math.abs(end.x - coord.x) + Math.abs(end.y - coord.y)) \* DIRECT\_VALUE;
7. }

H的值计算方式也不太相同

结果：

电脑萤幕

中度可信度描述已自动生成

Bfs算法

Bfs的openlist与前两者不同其没有cost的比较，而秉承着先进先出的队列思想

1. List<Node> openList = new ArrayList<Node>(); *// 优先队列(升序)*
2. List<Node> closeList = new ArrayList<Node>();

所以用不着PriorityQueue（）进行排序

1. Node current = openList.remove(0);*//删除第一个元素*

Poll()方法也用不了改为remove（）

结果：

电脑萤幕

描述已自动生成

解释

1.因为三个算法计算距离的方式不同，所以Astar的总代价为欧几里得距离，而Dijkstra和Bfs算法用的是曼哈顿距离

2.用以上算法实现和网页上的路径不太一样，但总距离是一样的，个人感觉是队列排序时无法完全模拟网页结果所致

空间复杂度和时间复杂度

因为算法中的循环和储存空间都与整个地图的边长由着直接关系，所以把其边长设为n

则空间复杂度为O（n^2）（主要引导的是openlist），时间复杂度为也为O（n^2）