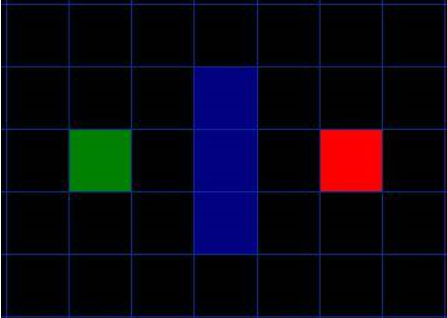
**AI课上代码实现 第二次试验 171491322 谷贺**

**A\*算法**

1.A\*算法是一种静态路网中求解最短路最有效的方法。

2.公式表示为： f(n)=g(n)+h(n), 其中 f(n) 是从初始点经由节点n到目标点的估价函数，g(n) 是在状态空间中从初始节点到n节点的实际代价，(n) 是从n到目标节点最佳路径的估计代价。

保证找到最短路径（最优解的）条件，关键在于估价函数h(n)的选取

？？？问题描述：有人想要从绿色的方格走到红色的方格，中间蓝色的是墙，怎么走才能使路途最短？

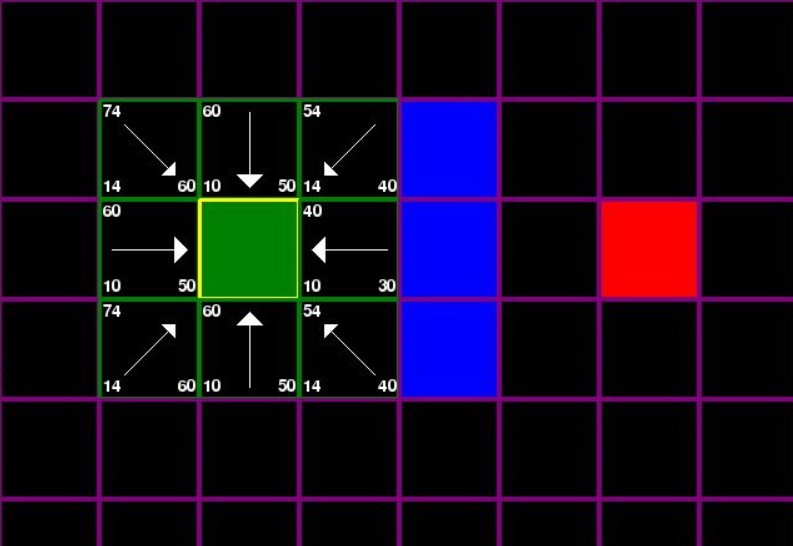
首先定义open list和close list，open list存放已知，但还没有探索过的区块，close list存放已经探索过的区块。

**最短路径肯定涉及到距离度量，在A\*算法中距离分为两个部分：G 和H，总距离F=G + H。**

**G等于从起点移动到指定方格的移动代价。在本例中，相邻节点间，横向和纵向的移动代价为 10 ，对角线的移动代价为 14 （10×根号2的近似）**

**H为从当前节点到终点的估计距离，是对剩余距离的估算值，而不是实际值**

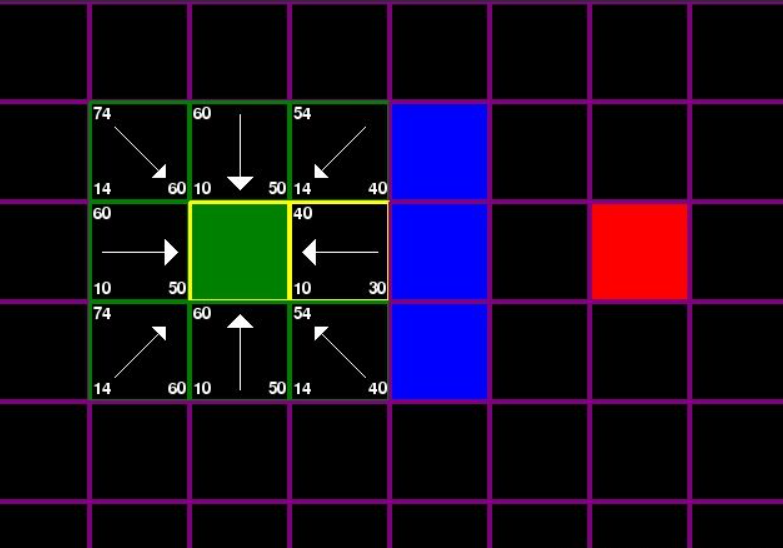
* \*绿色填充方块：起点
* \*蓝色填充方块：障碍
* \*红色填充方块：终点
* \*绿色边的方块：open list中的方块
* \*黄色边框方块：close list中的方块
* \*方块中白色箭头指向父亲节点
* \*方块中左上角数字代表F值，左下角G值，右下角H值



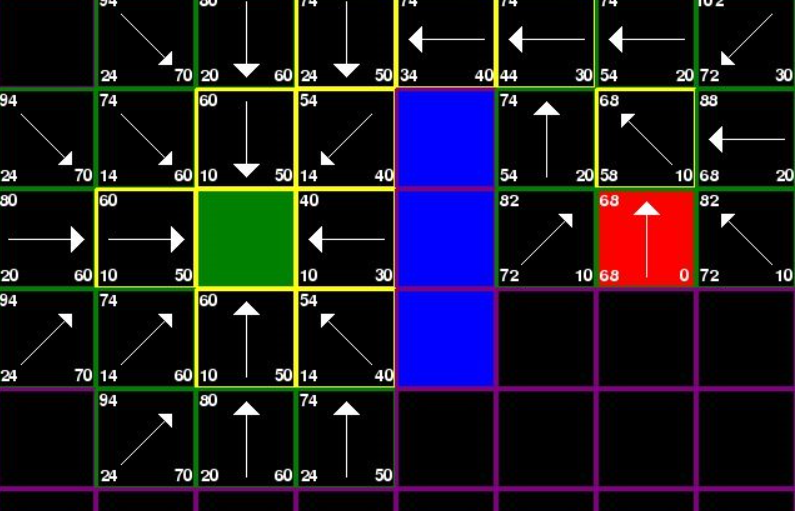
从open list中取出起始点，将起点加入close list。起点周围8个方格都可到达所以都加入到open list中，设置父节点为起点，并计算各自的F，G，H值。结果如上图所示。

从open list中找出F值最小的方格，起点右边的方格F值为40最小，暂且称该节点为A。将A从open list剔除，加入到close list。A右边为障碍物，忽略；其余方向的方格都已经在open list中且加入A并没有减小他们的G值，所以维持原样不变。

结果如下图所示，可见起点右边的方格加上了黄色框，代表进入close list，其余不变。



不断重复上述步骤，最后终点被加入到open list中，从终点开始，每个方格沿着父节点移动直至起点，就是最优路径。



**算法实现：**

**package** asd;

**import** java.util.Scanner;

**import** java.util.ArrayList;

**import** java.util.List;

**public** **class** wee {

**public** **static** **final** **int**[][] ***NODES*** = { { 0, 0, 0, 0, 0, 0, 0, 0, 0 }, { 0, 0, 0, 0, 0, 0, 0, 0, 0 },

{ 0, 0, 0, 0, 0, 0, 0, 0, 0 }, { 0, 0, 0, 1, 0, 0, 0, 0, 0 }, { 0, 0, 0, 1, 0, 0, 0, 0, 0 },

{ 0, 0, 0, 1, 0, 0, 0, 0, 0 }, { 0, 0, 0, 1, 0, 0, 0, 0, 0 }, { 0, 0, 0, 0, 0, 0, 0, 0, 0 },

{ 0, 0, 0, 0, 0, 0, 0, 0, 0 }, };

**public** **static** **final** **int** ***STEP*** = 10;

**private** ArrayList<Node> openList = **new** ArrayList<Node>();

**private** ArrayList<Node> closeList = **new** ArrayList<Node>();

**public** Node findMinFNodeInOpenList() {

Node tempNode = openList.get(0); // 先以第一个元素的F为最小值，然后遍历openlist的所有值，找出最小值

**for** (Node node : openList) {

**if** (node.F < tempNode.F) {

tempNode = node;

}

}

**return** tempNode;

}

// 考虑周围节点的时候，就不把节点值为1的节点考虑在内，所以自然就直接避开了障碍物

**public** ArrayList<Node> findNeighborNodes(Node currentNode) {

ArrayList<Node> arrayList = **new** ArrayList<Node>();

// 只考虑上下左右，不考虑斜对角

**int** topX = currentNode.x;

**int** topY = currentNode.y - 1;

// canReach方法确保下标没有越界 exists方法确保此相邻节点不存在于closeList中，也就是之前没有遍历过

**if** (canReach(topX, topY) && !*exists*(closeList, topX, topY)) {

arrayList.add(**new** Node(topX, topY));

}

**int** bottomX = currentNode.x;

**int** bottomY = currentNode.y + 1;

**if** (canReach(bottomX, bottomY) && !*exists*(closeList, bottomX, bottomY)) {

arrayList.add(**new** Node(bottomX, bottomY));

}

**int** leftX = currentNode.x - 1;

**int** leftY = currentNode.y;

**if** (canReach(leftX, leftY) && !*exists*(closeList, leftX, leftY)) {

arrayList.add(**new** Node(leftX, leftY));

}

**int** rightX = currentNode.x + 1;

**int** rightY = currentNode.y;

**if** (canReach(rightX, rightY) && !*exists*(closeList, rightX, rightY)) {

arrayList.add(**new** Node(rightX, rightY));

}

**return** arrayList;

}

**public** **boolean** canReach(**int** x, **int** y) {

**if** (x >= 0 && x < ***NODES***.length && y >= 0 && y < ***NODES***[0].length) {

**return** ***NODES***[x][y] == 0; // 避过障碍物，如果节点值不为0，说明不可到达。

}

**return** **false**;

}

**public** Node findPath(Node startNode, Node endNode) {

// 把起点加入 open list

openList.add(startNode);

**while** (openList.size() > 0) {

// 遍历 open list ，查找 F值最小的节点，把它作为当前要处理的节点

Node currentNode = findMinFNodeInOpenList();

// F值最小的节点从open list中移除

openList.remove(currentNode);

// 把这个节点移到 close list，closelist就是存储路径的链表

closeList.add(currentNode);

// 查找不存在于close list当中的周围节点（不考虑斜边的邻居）

ArrayList<Node> neighborNodes = findNeighborNodes(currentNode);

// openlist其实就是存储的外围的节点集合

**for** (Node node : neighborNodes) {// 总之要把邻居节点添加进openlist当中

**if** (*exists*(openList, node)) { // 如果邻居节点在openlist当中

foundPoint(currentNode, node);

} **else** {

// 如果邻居节点不在openlist中，那就添加进openlist

notFoundPoint(currentNode, endNode, node);

}

}

// 如果在openlist中找到了终点，那么就说明已经找到了路径，返回终点

**if** (*find*(openList, endNode) != **null**) {

**return** *find*(openList, endNode);

}

}

**return** *find*(openList, endNode);

}

// 此种情况就是发现周围的F值最小的节点是之前已经遍历过了的，所以这个节点的G,H,F值都是已经计算过了的

// 此时H值肯定不会变，所以要比较G值，如果现在的G值比之前的小，说明现在的路径更优

// 接着就重置此节点的父指针，G值和F值

**private** **void** foundPoint(Node tempStart, Node node) {

**int** G = calcG(tempStart, node);

**if** (G < node.G) {

node.parent = tempStart;

node.G = G;

node.calcF();

}

}

// 这种情况是之前没有计算过此节点的值，所以在这里要计算一遍G,H,F值，然后确认父指针指向，然后加入openlist当中

**private** **void** notFoundPoint(Node tempStart, Node end, Node node) {

node.parent = tempStart;

node.G = calcG(tempStart, node);

node.H = calcH(end, node);

node.calcF();

openList.add(node);

}

**private** **int** calcG(Node start, Node node) {

**int** G = ***STEP***;

**int** parentG = node.parent != **null** ? node.parent.G : 0;

**return** G + parentG;

}

// 这个是最简单粗暴的计算H值得方法，当然还有其它方法，这里先理解AStar的思想先，以后可以自己改进这个计算H值得方法

**private** **int** calcH(Node end, Node node) {

**int** step = Math.*abs*(node.x - end.x) + Math.*abs*(node.y - end.y);

**return** step \* ***STEP***;

}

**public** **static** **void** main(String[] args) {

Node startNode = **new** Node(5, 1);

Node endNode = **new** Node(5, 5);

Node parent = **new** wee().findPath(startNode, endNode);// 返回的是终点，但是此时父节点已经确立，可以追踪到开始节点

**for** (**int** i = 0; i < ***NODES***.length; i++) {

**for** (**int** j = 0; j < ***NODES***[0].length; j++) {

System.***out***.print(***NODES***[i][j] + ", ");

}

System.***out***.println();

}

ArrayList<Node> arrayList = **new** ArrayList<Node>();

**while** (parent != **null**) {// 遍历刚才找到的路径。没问题

// System.out.println(parent.x + ", " + parent.y);

arrayList.add(**new** Node(parent.x, parent.y));

parent = parent.parent;

}

System.***out***.println("\n");

**for** (**int** i = 0; i < ***NODES***.length; i++) {

**for** (**int** j = 0; j < ***NODES***[0].length; j++) {

**if** (*exists*(arrayList, i, j)) {// 把路径经过的点用@表示

System.***out***.print("@, ");

} **else** {

System.***out***.print(***NODES***[i][j] + ", ");

}

}

System.***out***.println();

}

}

**public** **static** Node find(List<Node> nodes, Node point) {

**for** (Node n : nodes)

**if** ((n.x == point.x) && (n.y == point.y)) {

**return** n;

}

**return** **null**;

}

**public** **static** **boolean** exists(List<Node> nodes, Node node) {

**for** (Node n : nodes) {

**if** ((n.x == node.x) && (n.y == node.y)) {

**return** **true**;

}

}

**return** **false**;

}

**public** **static** **boolean** exists(List<Node> nodes, **int** x, **int** y) {

**for** (Node n : nodes) {

**if** ((n.x == x) && (n.y == y)) {

**return** **true**;

}

}

**return** **false**;

}

**public** **static** **class** Node {

**public** Node(**int** x, **int** y) {

**this**.x = x;

**this**.y = y;

}

**public** **int** x;

**public** **int** y;

**public** **int** F;

**public** **int** G;

**public** **int** H;

**public** **void** calcF() {

**this**.F = **this**.G + **this**.H;

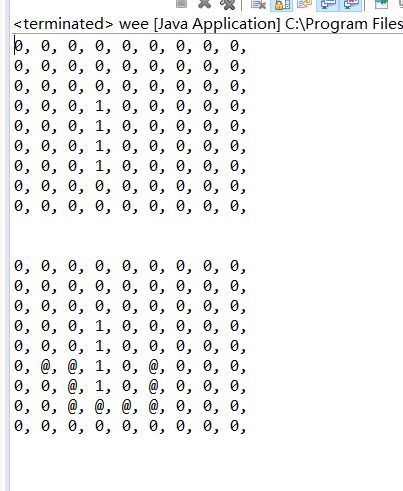
}

**public** Node parent;

}

}

运行结果：



A\*方法总结：

1.把起始格添加到开启列表。

2.重复如下的工作:

a)寻找开启列表中F值最低的格子。我们称它为当前格。

b)把它切换到关闭列表。

c)对相邻的8格中的每一个?

1.如果它不可通过或者已经在关闭列表中，略过它。反之如下。

2.如果它不在开启列表中，把它添加进去。把当前格作为这- -格的父节点。记录这一格的F,G 和H值。

3.如果它已经在开启列表中，用G值为参考检查新的路径是否更好。更低的G值意味着更好的路径。如果是这样，就把这一-格的父节点改成当前格，并且重新计算这-格的G和F值。如果你保持你的开启列表按F值排序，改变之后你可能需要重新对开启列表排序。

d)停止，当你

1.把目标格添加进了开启列表,这时候路径被找到，或者

2.没有找到目标格，开启列表已经空了。这时候，路径不存在。

3.保存路径。从目标格开始，沿着每一格的父节点移动直到回到起始格。