# A\*搜索算法求解

## 实验目的

用A\*算法求路径

## 二、算法分析

A\*算法主要与和实际代价g（n）和估计代价h（n）有关。A\*算法又包括两个性质，一个是不高估代价，意思就是每次我们估计的最小值总是比实际最小值要大，还有一个就是满足非递减性。

例如对于几何路网来说，可以取两节点间曼哈顿距离做为距离估计，这样估价函数f(n)在g(n)一定的情况下，会或多或少的受距离估计值h(n)的制约，节点距目标点近，h值小，f值相对就小，能保证最短路的搜索向终点的方向进行。明显优于Dijkstra算法的毫无方向的向四周搜索。

## 三、实验题目

已知A，B两个地点，中间设置障碍，周围设置墙壁（避免越界找不到结果）求从A点到B点的最短距离

‘$代表墙壁’

‘o代表空地’

‘#代表障碍物’

从A移动到B

## 四、实验代码

Map类：

**package** a\_star\_search;

**public** **class** Map {

**private** Node[][] map;

//节点数组

**private** Node startNode;

//起点

**private** Node endNode;

//终点

**public** Map() {

map = **new** Node[7][7];

**for** (**int** i = 0;i<7;i++){

**for** (**int** j = 0;j<7;j++){

map[i][j] = **new** Node(i,j,"o",**true**);

}

}

**for** (**int** d = 0;d<7;d++){

map[0][d].setValue("$");

map[0][d].setReachable(**false**);

map[d][0].setValue("$");

map[d][0].setReachable(**false**);

map[6][d].setValue("$");

map[6][d].setReachable(**false**);

map[d][6].setValue("$");

map[d][6].setReachable(**false**);

}

map[3][1].setValue("A");

startNode = map[3][1];

map[3][5].setValue("B");

endNode = map[3][5];

**for** (**int** k = 1;k<=3;k++){

map[k+1][3].setValue("#");

map[k+1][3].setReachable(**false**);

}

}

**public** **void** ShowMap(){

**for** (**int** i = 0;i<7;i++){

**for** (**int** j = 0;j<7;j++){

System.***out***.print(map[i][j].getValue()+" ");

}

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

}

}

**public** Node[][] getMap() {

**return** map;

}

**public** **void** setMap(Node[][] map) {

**this**.map = map;

}

**public** Node getStartNode() {

**return** startNode;

}

**public** **void** setStartNode(Node startNode) {

**this**.startNode = startNode;

}

**public** Node getEndNode() {

**return** endNode;

}

**public** **void** setEndNode(Node endNode) {

**this**.endNode = endNode;

}

}

Node类：

**package** a\_star\_search;

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

**private** **int** x; //x坐标

**private** **int** y; //y坐标

**private** String value; //表示节点的值

**private** **double** FValue = 0; //F值

**private** **double** GValue = 0; //G值

**private** **double** HValue = 0; //H值

**private** **boolean** Reachable; //是否可到达（是否为障碍物）

**private** Node PNode; //父节点

**public** Node(**int** x, **int** y, String value, **boolean** reachable) {

**super**();

**this**.x = x;

**this**.y = y;

**this**.value = value;

Reachable = reachable;

}

**public** Node() {

**super**();

}

**public** **int** getX() {

**return** x;

}

**public** **void** setX(**int** x) {

**this**.x = x;

}

**public** **int** getY() {

**return** y;

}

**public** **void** setY(**int** y) {

**this**.y = y;

}

**public** String getValue() {

**return** value;

}

**public** **void** setValue(String value) {

**this**.value = value;

}

**public** **double** getFValue() {

**return** FValue;

}

**public** **void** setFValue(**double** fValue) {

FValue = fValue;

}

**public** **double** getGValue() {

**return** GValue;

}

**public** **void** setGValue(**double** gValue) {

GValue = gValue;

}

**public** **double** getHValue() {

**return** HValue;

}

**public** **void** setHValue(**double** hValue) {

HValue = hValue;

}

**public** **boolean** isReachable() {

**return** Reachable;

}

**public** **void** setReachable(**boolean** reachable) {

Reachable = reachable;

}

**public** Node getPNode() {

**return** PNode;

}

**public** **void** setPNode(Node pNode) {

PNode = pNode;

}

}

AStar函数：

**package** a\_star\_search;

**import** java.util.ArrayList;

**public** **class** AStar {

/\*\*

\* 使用ArrayList数组作为“开启列表”和“关闭列表”

\*/

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

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

/\*\*

\* 获取H值

\* **@param** currentNode：当前节点

\* **@param** endNode：终点

\* **@return**

\*/

**public** **double** getHValue(Node currentNode,Node endNode){

**return** (Math.*abs*(currentNode.getX() - endNode.getX()) + Math.*abs*(currentNode.getY() - endNode.getY()))\*10;

}

/\*\*

\* 获取G值

\* **@param** currentNode：当前节点

\* **@return**

\*/

**public** **double** getGValue(Node currentNode){

**if**(currentNode.getPNode()!=**null**){

**if**(currentNode.getX()==currentNode.getPNode().getX()||currentNode.getY()==currentNode.getPNode().getY()){

//判断当前节点与其父节点之间的位置关系（水平？对角线）

**return** currentNode.getGValue()+10;

}

**return** currentNode.getGValue()+14;

}

**return** currentNode.getGValue();

}

/\*\*

\* 获取F值 ： G + H

\* **@param** currentNode

\* **@return**

\*/

**public** **double** getFValue(Node currentNode){

**return** currentNode.getGValue()+currentNode.getHValue();

}

/\*\*

\* 将选中节点周围的节点添加进“开启列表”

\* **@param** node

\* **@param** map

\*/

**public** **void** inOpen(Node node,Map map){

**int** x = node.getX();

**int** y = node.getY();

**for** (**int** i = 0;i<3;i++){

**for** (**int** j = 0;j<3;j++){

//判断条件为：节点为可到达的（即不是障碍物，不在关闭列表中），开启列表中不包含，不是选中节点

**if**(map.getMap()[x-1+i][y-1+j].isReachable()&&!open.contains(map.getMap()[x-1+i][y-1+j])&&!(x==(x-1+i)&&y==(y-1+j))){

map.getMap()[x-1+i][y-1+j].setPNode(map.getMap()[x][y]);

//将选中节点作为父节点

map.getMap()[x-1+i][y-1+j].setGValue(getGValue(map.getMap()[x-1+i][y-1+j]));

map.getMap()[x-1+i][y-1+j].setHValue(getHValue(map.getMap()[x-1+i][y-1+j],map.getEndNode()));

map.getMap()[x-1+i][y-1+j].setFValue(getFValue(map.getMap()[x-1+i][y-1+j]));

open.add(map.getMap()[x-1+i][y-1+j]);

}

}

}

}

/\*\*

\* 使用冒泡排序将开启列表中的节点按F值从小到大排序

\* **@param** arr

\*/

**public** **void** sort(ArrayList<Node> arr){

**for** (**int** i = 0;i<arr.size()-1;i++){

**for** (**int** j = i+1;j<arr.size();j++){

**if**(arr.get(i).getFValue() > arr.get(j).getFValue()){

Node tmp = **new** Node();

tmp = arr.get(i);

arr.set(i, arr.get(j));

arr.set(j, tmp);

}

}

}

}

/\*\*

\* 将节点添加进”关闭列表“

\* **@param** node

\* **@param** open

\*/

**public** **void** inClose(Node node,ArrayList<Node> open){

**if**(open.contains(node)){

node.setReachable(**false**);

//设置为不可达

open.remove(node);

close.add(node);

}

}

**public** **void** search(Map map){

//对起点即起点周围的节点进行操作

inOpen(map.getMap()[map.getStartNode().getX()][map.getStartNode().getY()],map);

close.add(map.getMap()[map.getStartNode().getX()][map.getStartNode().getY()]);

map.getMap()[map.getStartNode().getX()][map.getStartNode().getY()].setReachable(**false**);

map.getMap()[map.getStartNode().getX()][map.getStartNode().getY()].setPNode(map.getMap()[map.getStartNode().getX()][map.getStartNode().getY()]);

sort(open);

//重复步骤

**do**{

inOpen(open.get(0), map);

inClose(open.get(0), open);

sort(open);

}

**while**(!open.contains(map.getMap()[map.getEndNode().getX()][map.getEndNode().getY()]));

//知道开启列表中包含终点时，循环退出

inClose(map.getMap()[map.getEndNode().getX()][map.getEndNode().getY()], open);

showPath(close,map);

}

/\*\*

\* 将路径标记出来

\* **@param** arr

\* **@param** map

\*/

**public** **void** showPath(ArrayList<Node> arr,Map map) {

**if**(arr.size()>0){

@SuppressWarnings("unused")

Node node = **new** Node();

node = map.getMap()[map.getEndNode().getX()][map.getEndNode().getY()];

**while**(!(node.getX() ==map.getStartNode().getX()&&node.getY() ==map.getStartNode().getY())){

node.getPNode().setValue("\*");

node = node.getPNode();

}

}

}

}

主函数：

**package** a\_star\_search;

**public** **class** MainTest {

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

// **TODO** Auto-generated method stub

Map map = **new** Map();

AStar aStar = **new** AStar();

map.ShowMap();

aStar.search(map);

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

System.***out***.println("经过A\*算法计算后");

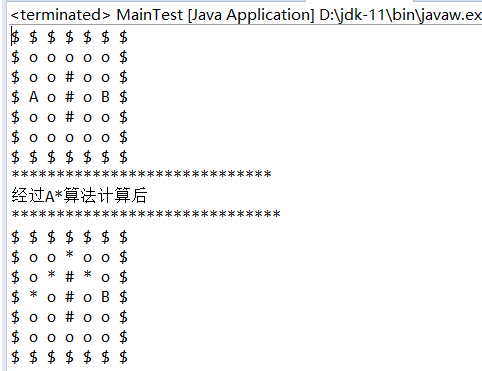
System.***out***.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");

map.ShowMap();

}

}

## 五、实验结果：



## 六、实验总结

A\*算法的重点就是关于估计代价h（n）的查找，每次都要找到那个从当前节点到目标节点路径最短的节点，然后从该节点继续查找到目标节点路径最短的节点，到目标的实际距离，就是已经走过的路径加上估计的距离。