# IDA\*搜索算法求解

## 实验目的

用IDA\*算法完成目标的搜索

## 二、算法分析

IDA\*算法是A\*的一个变形，和A\*算法不同的是他不保存之前的搜索状态(意味着同一个节点可能会被搜索多次)，它的搜索效率会稍微低于A\*算法。它的本质是深度优先搜索，每次找到最小的节点展开它的所有子节点，然后深度+1进行搜索

## 三、实验题目

设想有这样的一个场景，有老板A和员工B，老板A有一个拼图游戏需要求解，老板观察了一下问题，根据问题的manhattan距离得到了一个可能的最小步骤解n。于是把员工B叫过来，说你给我在n步之内解出题目。于是员工B开始解题，他的方法很简单采用深搜的方法搜索所有符合该manhattan距离限制的解，此时，可能当员工B遍历了所以的结果后还是没有找到可行解，于是他只能把这个结果和老板A说了，老板A看了以后，允许解题步骤限制+1，然后员工B在以这个n+1的步骤限制，从头开始做一个遍历，可能他还没有找到可行解，他就会再去找老板，老板说再加1吧，如此循环，直到员工B求得可行解。

## 四、实验代码

**package** LAB\_5;

**public** **class** IDAstar {

//分别代表左、上、右、下四个移动方向的操作数

**private** **int**[] up = {-1,0};

**private** **int**[] down = {1,0};

**private** **int**[] left = {0,-1};

**private** **int**[] right = {0,1};

**private** **final** **int** UP = 0;

**private** **final** **int** DOWN = 2;

**private** **final** **int** LEFT = 1;

**private** **final** **int** RIGHT = 3;

**private** **int** SIZE;

//各个目标点坐标

**private** **int**[][] targetPoints;

//用于记录移动步骤，存储0,1,2,3,对应上，下，左，右

**private** **static** **int**[] *moves* = **new** **int**[100000];

**private** **static** **long** *ans* = 0;; //当前迭代的"设想代价"

//目标状态

**private** **static** **int**[][] *tState* = {

{1 ,2 ,3 ,4 } ,

{5 ,6 ,7 ,8 } ,

{9 ,10,11,12} ,

{13,14,15,0 }

};

**private** **static** **int**[][] *sState* = {

{2 ,10 ,3 ,4 } ,

{1 ,0,6 ,8 } ,

{5 ,14,7,11} ,

{9,13,15,12 }

};

//初始状态

// private static int[][] sState = {

// {12,1 ,10,2 } ,

// {7 ,11,4 ,14} ,

// {5 ,0 ,9 ,15} ,

// {8 ,13,6 ,3}

// };

**private** **static** **int** *blank\_row*,*blank\_column*;

**public** IDAstar(**int**[][] state) {

SIZE = state.length;

targetPoints = **new** **int**[SIZE \* SIZE][2];

**this**.*sState* = state;

//得到空格坐标

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

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

**if**(state[i][j] == 0) {

*blank\_row* = i;

*blank\_column* = j;

**break**;

}

}

}

//得到目标点坐标数组

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

**for**(**int** j=0;j<state.length;j++) {

targetPoints[*tState*[i][j]][0] = i; //行信息

targetPoints[*tState*[i][j]][1] = j; //列信息

}

}

}

/\*\*

\* 讨论问题的可解性

\* **@param** state 状态

\*/

**private** **boolean** canSolve(**int**[][] state) {

**if**(state.length % 2 == 1) { //问题宽度为奇数

**return** (getInversions(state) % 2 == 0);

} **else** { //问题宽度为偶数

**if**((state.length - *blank\_row*) % 2 == 1) { //从底往上数,空格位于奇数行

**return** (getInversions(state) % 2 == 0);

} **else** { //从底往上数,空位位于偶数行

**return** (getInversions(state) % 2 == 1);

}

}

}

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

IDAstar idaAlgorithm = **new** IDAstar(*sState*);

**if**(idaAlgorithm.canSolve(*sState*)) {

System.***out***.println("--问题可解，开始求解--");

//以曼哈顿距离为初始最小代价数

**int** j = idaAlgorithm.getHeuristic(*sState*);

System.***out***.println("初始manhattan距离:" + j);

**int** i = -1;//置空默认移动方向

**long** time = System.*currentTimeMillis*();

//迭代加深"最小代价数"

**for**(*ans*=j;;*ans*++) {

**if**(idaAlgorithm.solve(*sState*

,*blank\_row*,*blank\_column*,0,i,j)) {

**break**;

}

}

System.***out***.println("求解用时:"+(System.*currentTimeMillis*() - time));

idaAlgorithm.printMatrix(*sState*);

**int**[][] matrix = idaAlgorithm.move(*sState*,*moves*[0]);

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

matrix = idaAlgorithm.move(matrix, *moves*[k]);

}

} **else** {

System.***out***.println("--抱歉！输入的问题无可行解--");

}

}

**public** **int**[][] move(**int**[][]state,**int** direction) {

**int** row = 0;

**int** column = 0;

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

**for**(**int** j=0;j<state.length;j++) {

**if**(state[i][j] == 0) {

row = i;

column = j;

}

}

}

**switch**(direction) {

**case** UP:

state[row][column] = state[row-1][column];

state[row-1][column] = 0;

**break**;

**case** DOWN:

state[row][column] = state[row+1][column];

state[row+1][column] = 0;

**break**;

**case** LEFT:

state[row][column] = state[row][column-1];

state[row][column-1] = 0;

**break**;

**case** RIGHT:

state[row][column] = state[row][column+1];

state[row][column+1] = 0;

**break**;

}

printMatrix(state);

**return** state;

}

**public** **void** printMatrix(**int**[][] matrix) {

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

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

**for**(**int** j=0;j<matrix.length;j++) {

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

}

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

}

}

/\*\*

\* 求解方法

\* \* **@param** state 当前状态

\* **@param** blank\_row 空位的行坐标

\* **@param** blank\_column 空格的列坐标

\* **@param** dep 当前深度

\* **@param** d 上一次移动的方向

\* **@param** h 当前状态估价函数

\* **@return**

\*/

**public** **boolean** solve(**int**[][] state,**int** blank\_row,**int** blank\_column,

**int** dep,**long** d,**long** h) {

**long** h1;

//和目标矩阵比较，看是否相同，如果相同则表示问题已解

**boolean** isSolved = **true**;

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

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

**if**(state[i][j] != *tState*[i][j]) {

isSolved = **false**;

}

}

}

**if**(isSolved) {

**return** **true**;

}

**if**(dep == *ans*) {

**return** **false**;

}

//用于表示"空格"移动后的坐标位置

**int** blank\_row1 = blank\_row;

**int** blank\_column1 = blank\_column;

**int**[][] state2 = **new** **int**[SIZE][SIZE];

**for**(**int** direction=0;direction<4;direction++) {

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

**for**(**int** j=0;j<state.length;j++) {

state2[i][j] = state[i][j];

}

}

//本地移动方向和上次移动方向刚好相反，跳过这种情况的讨论

**if**(direction != d && (d%2 == direction%2)) {

**continue**;

}

**if**(direction == UP) {

blank\_row1 = blank\_row + up[0];

blank\_column1 = blank\_column + up[1];

} **else** **if**(direction == DOWN) {

blank\_row1 = blank\_row + down[0];

blank\_column1 = blank\_column + down[1];

} **else** **if**(direction == LEFT) {

blank\_row1 = blank\_row + left[0];

blank\_column1 = blank\_column + left[1];

} **else** {

blank\_row1 = blank\_row + right[0];

blank\_column1 = blank\_column + right[1];

}

//边界检查

**if**(blank\_column1 < 0 || blank\_column1 == SIZE

|| blank\_row1 < 0 || blank\_row1 == SIZE) {

**continue** ;

}

//交换空格位置和当前移动位置对应的单元格

state2[blank\_row][blank\_column] = state2[blank\_row1][blank\_column1];

state2[blank\_row1][blank\_column1] = 0;

//查看当前空格是否正在靠近目标点

**if**(direction == DOWN && blank\_row1

> targetPoints[state[blank\_row1][blank\_column1]][0]) {

h1 = h - 1;

} **else** **if**(direction == UP && blank\_row1

< targetPoints[state[blank\_row1][blank\_column1]][0]){ h1 = h - 1;

} **else** **if**(direction == RIGHT && blank\_column1

> targetPoints[state[blank\_row1][blank\_column1]][1]) {

h1 = h - 1;

} **else** **if**(direction == LEFT && blank\_column1

< targetPoints[state[blank\_row1][blank\_column1]][1]) {

h1 = h - 1;

} **else** {

//这种情况发生在任意可能的移动方向都会使得估价函数值变大

h1 = h + 1;

}

**if**(h1+dep+1>*ans*) { //剪枝

**continue**;

}

*moves*[dep] = direction;

//迭代深度求解

**if**(solve(state2, blank\_row1, blank\_column1, dep+1, direction, h1)) {

**return** **true**;

}

}

**return** **false**;

}

/\*\*

\* 得到估价函数值

\*/

**public** **int** getHeuristic(**int**[][] state) {

**int** heuristic = 0;

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

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

**if**(state[i][j] != 0) {

heuristic = heuristic +

Math.*abs*(targetPoints[state[i][j]][0] - i)

+ Math.*abs*(targetPoints[state[i][j]][1] - j);

}

}

}

**return** heuristic;

}

/\*\*

\* 计算问题的"倒置变量和"

\* **@param** state

\*/

**private** **int** getInversions(**int**[][] state) {

**int** inversion = 0;

**int** temp = 0;

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

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

**int** index = i\* state.length + j + 1;

**while**(index < (state.length \* state.length)) {

**if**(state[index/state.length][index%state.length] != 0

&& state[index/state.length]

[index%state.length] < state[i][j]) {

temp ++;

}

index ++;

}

inversion = temp + inversion;

temp = 0;

}

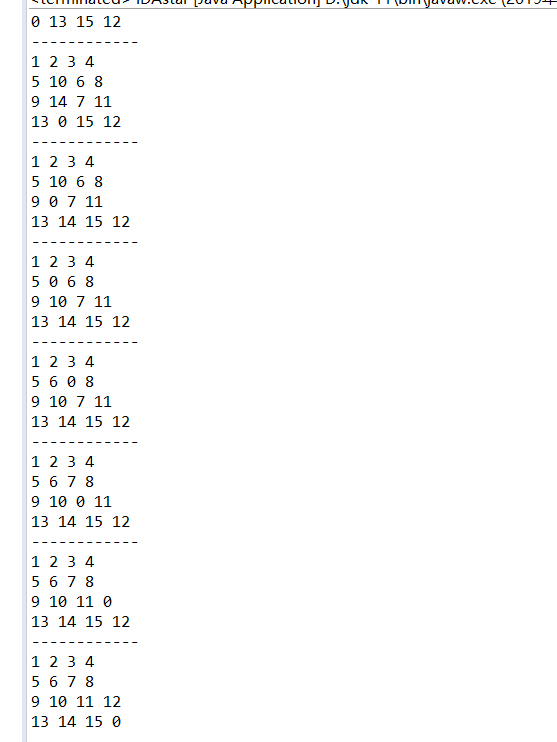
}

**return** inversion;

}

}

## 五、实验结果：



## 六、实验总结

IDA\*的搜索策略是深度优先，不过它搜索最小的节点的子节点时是广度优先，它与IDS算法类似，也是每次搜索完没找到结果就深度+1，但IDS是全部的深度+1，而IDA\*是找到的节点的子节点深度+1，它的时间复杂度为O（B^d）空间复杂度是O（b\*d）。