**AI课上代码实现 第四次试验 硬件一班 王倩倩 171491121**

1. **IDA\***

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。IDA\*的翻译是迭代深度的A\*算法，它是一种有信息提示的搜索。

**代码实现：**

**package** chess;

/\*\*

\* IDA\*求解15puzzle问题

\* IDA\*整合了IDDFS和A\*算法。其中IDDFS控制了求解过程中的内存开销，A\*算法意味着"启发式"搜索。

\* IDA\*可以理解成迭代深度的A\*算法，其中这里的"深度"指的是问题的解的"假定损耗"。

\* 使"假定损耗"不断迭代增加，检验是否能在"假定损耗"的前提找到可行解，如果不行的话，就继续迭代。

\* 这里和A\*算法不同的是没有开放列表，由于采用了IDDFS的策略，IDA\*是深度优先搜索的，故此没有开放列表。

\* **@author** wly

\* **@date** 2013-12-20

\*

\*/

**public** **class** chess {

//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};

/\*\*注意，这里UP和DOWN，LEFT和RIGHT必须是两两相对的，因为后面代码中使用

\* ((dPrev != dCurr) && (dPrev%2 == dCurr%2))

\* 来判断前后两个移动方向是否相反

\*/

**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** chess(**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) {

chess idaAlgorithm = **new** chess(*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**;

}

**运行结果截图：**





