24数码

1.算法说明

- 使用C++实现4个算法

  -A * + h1

  -IDA * + h1

  -A * + h2

  -IDA * + h2

- 自己设计的启发式函数h2

  -使用曼哈顿距离的变种，考虑四个联通通道，计算任意两个格子的最短距离，记录在一个25X25的二维数组distance中

  -可采纳性：因为h表示任意两点间的最小步数，故总小于实际路径耗散，且大于等于0，故可采纳

  -相比于h1, h2 在A*算法中取得了一定时间上的改进，但对IDA算法无改进

- 对时间和空间的优化

  -对于A*算法，用一个优先队列存储所有待扩展状态，本质是一个最小堆

  -对于重复的状态，不再生成新结点，而是规约到已有结点

2.关键代码说明

- distance数组

```
int distance[25][25]={
    {0,1,2,3,4, 1,2,3,4,4, 2,3,4,4,3, 3,4,4,5,4, 4,4,3,4,5},
    {1,0,1,2,3, 2,1,2,3,4, 3,2,3,4,4, 4,3,3,4,5, 4,3,2,3,4},
    {2,1,0,1,2, 3,2,1,2,3, 4,3,2,3,4, 4,3,2,3,4, 3,2,1,2,3},
    {3,2,1,0,1, 4,3,2,1,2, 4,4,3,2,3, 5,4,3,3,4, 4,3,2,3,4},
    {4,3,2,1,0, 4,4,3,2,1, 3,4,4,3,2, 4,5,4,4,3, 5,4,3,4,4},

    {1,2,3,4,4, 0,1,2,3,3, 1,2,3,3,2, 2,3,4,4,3, 3,4,4,5,4},
    {2,1,2,3,4, 1,0,1,2,3, 2,1,2,3,3, 3,2,3,4,4, 4,3,3,4,5},
    {3,2,1,2,3, 2,1,0,1,2, 3,2,1,2,3, 4,3,2,3,4, 4,3,2,3,4},
    {4,3,2,1,2, 3,2,1,0,1, 3,3,2,1,2, 4,4,3,2,3, 5,4,3,3,4},
    {4,4,3,2,1, 3,3,2,1,0, 2,3,3,2,1, 3,4,4,3,2, 4,5,4,4,3},

    {2,3,4,4,3, 1,2,3,3,2, 0,1,2,2,3, 1,2,3,3,2, 2,3,4,4,3},
    {3,2,3,4,4, 2,1,2,3,3, 1,0,1,2,2, 2,1,2,3,3, 3,2,3,4,4},
    {4,3,2,3,4, 3,2,1,2,3, 2,1,0,1,2, 3,2,1,2,3, 4,3,2,3,4},
    {4,4,3,2,3, 3,3,2,1,2, 2,2,1,0,1, 3,3,2,1,2, 4,4,3,2,3},
    {3,4,4,3,2, 2,3,3,2,1, 3,2,2,1,0, 2,3,3,2,1, 3,4,4,3,2},

    {3,4,4,5,4, 2,3,4,4,3, 1,2,3,3,2, 0,1,2,3,3, 1,2,3,4,4},
    {4,3,3,4,5, 3,2,3,4,4, 2,1,2,3,3, 1,0,1,2,3, 2,1,2,3,4},
    {4,3,2,3,4, 4,3,2,3,4, 3,2,1,2,3, 2,1,0,1,2, 3,2,1,2,3},
    {5,4,3,3,4, 4,4,3,2,3, 3,3,2,1,2, 3,2,1,0,1, 4,3,2,1,2},
    {4,5,4,4,3, 3,4,4,3,2, 2,3,3,2,1, 3,3,2,1,0, 4,4,3,2,1},

    {4,4,3,4,5, 3,4,4,5,4, 2,3,4,4,3, 1,2,3,4,4, 0,1,2,3,4},
    {4,3,2,3,4, 4,3,3,4,5, 3,2,3,4,4, 2,1,2,3,4, 1,0,1,2,3},
    {3,2,1,2,3, 4,3,2,3,4, 4,3,2,3,4, 3,2,1,2,3, 2,1,0,1,2},
    {4,3,2,3,4, 5,4,3,3,4, 4,4,3,2,3, 4,3,2,1,2, 3,2,1,0,1},
    {5,4,3,4,4, 4,5,4,4,3, 3,4,4,3,2, 4,4,3,2,1, 4,3,2,1,0}
};
```

- A*算法的实现

```
bool Astart(Picture *&beginPicture, Picture *&endPicture) {
    priority_queue<Picture *, vector<Picture *>, cmp> openQueue;
    vector<Picture *> openTable, closeTable;
    beginPicture->setHValue(*endPicture);
    beginPicture->updateFvalue();

    openQueue.push(beginPicture);
    openTable.push_back(beginPicture);
    int move[4][2] = { {-1, 0},
                      {1,  0},
                      {0,  -1},
                      {0,   1}};
   int counter=0;
    while (!openQueue.empty()) {
        if(counter==5000){
            cout<<"5000"<<endl;
            counter=0;
        }
        counter++;
        Picture *bestPicture = openQueue.top();

        if (*bestPicture == *endPicture) {
            cout<<"find the result"<<endl;
            delete endPicture;
            endPicture = bestPicture;
            return true;
        }

        closeTable.push_back(bestPicture);
        openQueue.pop();
        deleteElement(openTable, bestPicture);
        //4 directions
        for (int i = 0; i < 4; ++i){
            int row = bestPicture->zeroRow + move[i][0];
            int column = bestPicture->zeroColumn + move[i][1];

            if(row==2 && column==-1)    {row=2; column=Picture::columnSize-1;}
            if(row==2 && column == Picture::columnSize )    {row=2; column=0;}
            if(row==-1 && column==2)    {row=Picture::rowSize-1; column==2;}
            if(row==Picture::rowSize && column==2)  {row=0; column==2;}
            if (row >= 0 && row < Picture::rowSize && column >= 0 && column <
Picture::columnSize)
            {
                Picture *successor = new Picture(*bestPicture);
                int **theArray = successor->getPicturePoint();

                    theArray[successor->zeroRow][successor->zeroColumn] =
theArray[row][column];
                    theArray[row][column] = 0;
                    successor->zeroRow = row;
                    successor->zeroColumn = column;
                    successor->parente = bestPicture;

                switch(i)
                {
                    case 0: successor->parent='U'; break;
                    case 1: successor->parent='D'; break;
                    case 2: successor->parent='L'; break;
                    case 3: successor->parent='R'; break;
                }

                successor->setGvalue(bestPicture->getGvalue() + 1);
                int flag = inVector(openTable, *successor);
                if (flag >= 0) {
                    if (successor->getGvalue() < openTable[flag]->getGvalue()) {
                        openTable[flag]->setGvalue(successor->getGvalue());
                        openTable[flag]->parente = bestPicture;
                        openTable[flag]->parent = successor->parent;
                        openTable[flag]->updateFvalue();
                        delete successor;
                    }
                }
                flag = inVector(closeTable, *successor);
                if (flag >= 0) {
                    if (successor->getGvalue() < closeTable[flag]->getGvalue()) {
                        closeTable[flag]->setGvalue(successor->getGvalue());
                        closeTable[flag]->parente = bestPicture;
                        closeTable[flag]->parent = successor->parent;
                        closeTable[flag]->updateFvalue();
                        delete successor;
                        openQueue.push(closeTable[flag]);
                        openTable.push_back(closeTable[flag]);
                        closeTable.erase(closeTable.begin() + flag);
                    }
                } else {
                    //cout<<"here"<<endl;
                    successor->setHValue(*endPicture);
                    successor->updateFvalue();
                    openQueue.push(successor);
                    openTable.push_back(successor);
                }
            }
```

```
        }
    }

    return false;
}
```

- IDA*算法的实现

```
int IDAstar(){
    int h = H2(start,target);
    cout<<"h:"<<h<<endl;
    maxDepth = h;
    while (!dfs(start,0, h, '\0'))
        maxDepth++;
    return maxDepth;
}


//IDAstar算法开始
bool dfs(STATUS cur,int depth,int h,char preDir){

    //IDA*估值函数剪枝
    //当前局面的估价函数值+当前的搜索深度 > 预定义的最大搜索深度时剪枝
    if( (depth+h>maxDepth)||depth>100) return false;

     if(memcmp(&cur, &target, sizeof(STATUS)) == 0 )
    {
        path[depth] = '\0';
        return true;
    }

    STATUS next;
    for(int i=0;i<4;++i){
        if(dirCode[i]==preDir) continue;//不能回到上一状态
        next=cur;
        int row,column;
        row = cur.zeroRow + dir[i][0];
        column = cur.zeroColumn + dir[i][1];
        //处理4个缺口
        if(row==2 && column==-1)
            {row=2; column=COLUMNSIZE-1;}
        if(row==2 && column == COLUMNSIZE )
            {row=2; column=0;}
        if(row==-1 && column==2)
            {row=ROWSIZE-1; column==2;}
        if(row==ROWSIZE && column==2)
            {row=0; column==2;}
        next.zeroRow=row;
        next.zeroColumn=column;
        //状态不合法则回退
        if( !( next.zeroRow >= 0 && next.zeroRow < ROWSIZE && next.zeroColumn >= 0
&& next.zeroColumn < COLUMNSIZE ) )
            continue;
        swap(next.data[cur.zeroRow][cur.zeroColumn], next.data[next.zeroRow]
[next.zeroColumn]); //置换变成新的状态
        int nexth=H2(next,target);//重新计算h值
        path[depth] = dirCode[i];
        if(dfs(next, depth + 1, nexth, rDirCode[i]))
            return true;
    }
    return false;
}
```

3.实验结果

1.A*+h1

样例编号	运行时间	移动序列	总步数
1	0.000000s	DDLDD	5
2	0.001000s	LULDLULLDD	10
3	0.013000s	LLUURRRUURDDDDLUURDD	20

2.IDA*+h1

样例编号	运行时间	移动序列	总步数
1	0.001000s	DDLDD	5
2	0.001000s	LULDLULLDD	10
3	0.000000s	LLUURRRUURDDDDLUURDD	20
4	0.032000s	RDRURRDRUUULDLDLDDLUUUURRURR	28
5	656.719000s	RDDDLLDRRURURDDRRRDLLDRRDDDDDRUURULLUURR	40

3.A*+h2

样例编号	运行时间	移动序列	总步数
1	0.000000s	DDLDD	5
2	0.001000s	LULDLULLDD	10
3	0.000000s	LLUURRRUURDDDDLUURDD	20
4	0.771000s	RDRDLUUUURRUUURDRUUULDLDDDRR	28

4.IDA*+h2

样例编号	运行时间	移动序列	总步数
1	0.001000s	DDLDD	5
2	0.017000s	LULDLULLDD	10
3	85.786000s	LLUURRRUURDDDDLUURDD	20