人工智能实验一

一、引言

15拼图游戏（15-Puzzle）是经典的滑块拼图游戏。该游戏由一个4x4的矩阵组成，其中包含15个编号为1至15的数字块，以及一个空白块。玩家的目标是通过移动数字块，将其排列成一个指定的目标状态。空白块可以用来交换其相邻位置的数字块，允许上下左右的滑动。每次移动，空白块所在位置的数字块与其交换，从而达到不同的状态。

15拼图游戏可以通过广度优先搜索（BFS）或深度优先搜索（DFS）等传统的搜索算法解决。然而，随着状态空间的迅速增大，传统的搜索算法效率较低，因此启发式搜索算法（如A*算法）成为一种常见且有效的解决方案。

本实验的主要目标是实现15拼图游戏的求解，并比较不同启发式策略（曼哈顿距离和对角线距离）对A*算法性能的影响。具体包括：

1. **实现A*算法**：基于启发式函数进行搜索，求解15拼图的最短路径。
2. **比较启发式策略**：分别使用曼哈顿距离和对角线距离作为启发式函数，比较它们在求解过程中的表现。
3. **分析代价函数与启发式选择的关系**：通过实验结果分析启发式函数对A*算法性能的影响。

二、问题的表示和求解算法

A*算法是一种典型的启发式搜索算法，广泛应用于路径规划问题。A*算法结合了广度优先搜索的全面性和贪婪算法的高效性。它通过引入一个评估函数 f(n)f(n) 来选择最优路径，该函数由两个部分组成：

- g(n)g(n)：从起始状态到当前状态 nn 的实际代价。
- h(n)h(n)：当前状态 nn 到目标状态的预估代价（即启发式函数）。

A*算法的评估函数为：

$$f(n)=g(n)+h(n)$$

在15拼图游戏中，A*算法通过评估每个状态的总代价来选择最优路径，从而有效地解决拼图问题。常用的启发式函数包括曼哈顿距离（Manhattan Distance）和对角线距离（Diagonal Distance），它们用于计算当前状态与目标状态之间的距离。

三、实验设计

在本实验中，使用A*算法来求解15拼图游戏。以下是具体的算法实现和相关分析。

1. 状态表示与类设计

本实验中，15拼图的状态使用`state`类进行表示。每个状态包含4x4的矩阵，表示当前的拼图布局，以及其他几个辅助信息：

- `d`：当前状态的深度值，表示从初始状态到当前状态所经历的步数。
- `p`：当前状态到目标状态的启发式估计值，用来帮助A*算法判断哪个状态最有可能是最优路径的一部分。
- `f`：当前状态的总代价，计算公式为：`f = g + h`，其中`g`为实际代价（深度值`d`），`h`为启发式代价（启发式函数`p`）。
- `matrix`：4x4矩阵，表示当前状态下各个拼图块的位置。

`state`类的主要功能包括状态比较、深度值和启发式估计值的计算，以及状态的移动操作。

2. 启发式函数设计

启发式函数需要满足以下两种性质：

1.可采纳的（admissible）

2.单调的（consistent）

启发式函数的准确性直接影响搜索算法的性能。由于搜索算法的目标是找到最优解，因此启发式函数应该尽可能接近实际距离。当估价函数低估实际距离时，搜索算法会产生过多的扩展节点，导致算法运行时间增加。相反，当估价函数高估实际距离时，搜索算法可能会错过更优解，导致算法不能找到最优解。

因此，分情况分析估价值和真实值的差距非常重要，可以设计更准确的启发式函数。使用多种启发式函数来评估问题可能是一个好的策略，**可以在算法性能和搜索时间之间找到一个平衡点。**

h(n) 的值	描述	性能变化
h(n) = 0	只有g(n)起作用，退化为Dijkstra算法	保证找到最短路径，但速度极慢
h(n) <= h*(n)		保证能找到最短路径，速度可能较慢
h(n) = h*(n)	只遵循最佳路径不会扩展其它节点	运行速度快并且能找到最短路径
h(n) > h*(n)		不能保证找到最短路径，但是速度将加快

故本实验中选择实现了两种启发式函数来计算状态的启发式估计值：

- **曼哈顿距离**：曼哈顿距离是指每个数字块与其目标位置的水平和垂直距离的总和。对于15拼图问题，曼哈顿距离能够较好地反映每个块与目标位置之间的局部距离。

```
int manhattanDistance(int t[ROW][COL]) {
    int dist = 0;
    for (int i = 0; i < ROW; i++) {
        for (int j = 0; j < COL; j++) {
            int value = matrix[i][j];
            if (value != 0) { // 0表示空白块
                int targetRow = (value - 1) / COL;
                int targetCol = (value - 1) % COL;
                dist += abs(i - targetRow) + abs(j - targetCol);  // 水平和垂直距离之和
            }
        }
    }
    return dist;
}
```

- **对角线距离**：对角线距离是计算每个数字块与目标位置之间的最大横向或纵向距离。该方法适合于考虑大范围的移动，可以在某些情况下比曼哈顿距离更有效。

```
int diagonalDistance(int t[ROW][COL]) {
    int dist = 0;
    for (int i = 0; i < ROW; i++) {
        for (int j = 0; j < COL; j++) {
            int value = matrix[i][j];
            if (value != 0) {
                int targetRow = (value - 1) / COL;
                int targetCol = (value - 1) % COL;
                dist += max(abs(i - targetRow), abs(j - targetCol));  // 计算对角线距离
            }
        }
    }
    return dist;
}
```

3. A*搜索过程

A*算法的核心是通过不断扩展当前状态的邻居状态，来找到最优解。我们使用`open`表和`close`表来分别存储待处理的状态和已处理的状态：

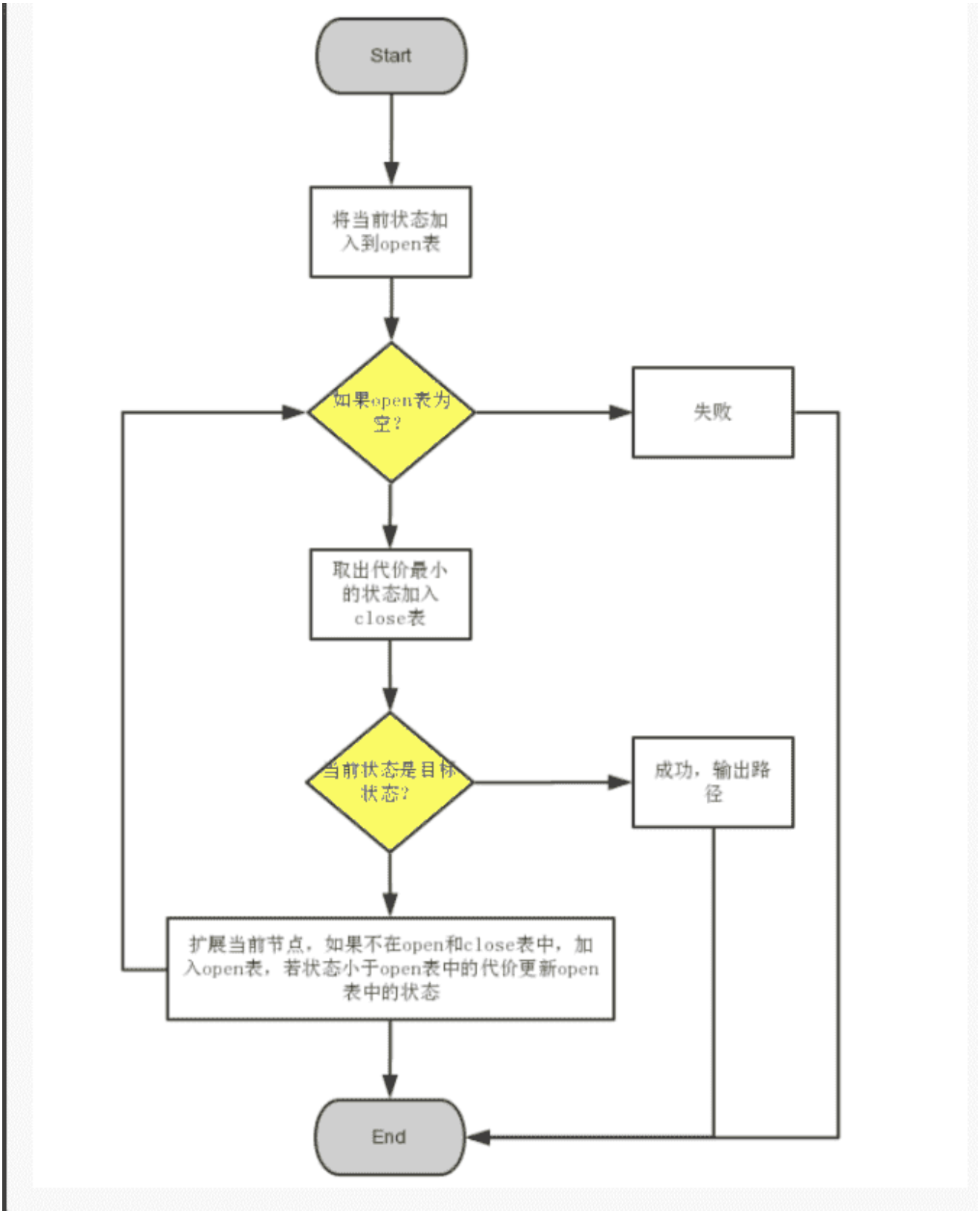
- `open`**表**：存储待处理的状态，按照`f`值（总代价）排序。
- `close`**表**：存储已处理的状态，避免重复处理。

A*算法的主要步骤如下：

1. 从起始状态开始，计算其启发式代价`f = g + h`（`g`为实际代价，`h`为启发式估计）。

2. 将起始状态加入 `open` 表。
3. 从 `open` 表中选择 `f` 值最小的状态，进行扩展。
4. 对于每个扩展出的新状态，计算其 `f` 值，并根据情况更新 `open` 表或 `close` 表。
5. 重复以上步骤，直到找到目标状态或 `open` 表为空。

.



四、实验结果

1.实验结果实例

本实验使用了一个示例状态（起始状态）和目标状态进行测试，具体的测试结果如下：

- **测试起始状态**：

```
3  2  7  8
1  6  4 12
5 10 11 15
9 13 0  14
```

- **测试目标状态**：

```
1  2  3  4
5  6  7  8
9 10 11 12
13 14 15  0
```

实验结果展示
通过A*算法的搜索过程，可以得到不同启发式函数下的搜索路径。以下是两种启发式函数的比较：

- **曼哈顿距离**:

```
1 2 3 4
5 6 0 7
9 10 11 8
13 14 15 12
d:22 h:3 f:25

step: 23
1 2 3 4
5 6 7 0
9 10 11 8
13 14 15 12
d:23 h:2 f:25

step: 24
1 2 3 4
5 6 7 8
9 10 11 0
13 14 15 12
d:24 h:1 f:25

step: 25
1 2 3 4
5 6 7 8
9 10 11 12
13 14 15 0
d:25 h:0 f:25

find the target!
A* algorithm execution time: 1730 milliseconds
```

- 搜索路径长度：25step
- 总运行时间：1730millisecond

- **对角线距离**:

```
step: 22
1 2 3 4
5 6 0 7
9 10 11 8
13 14 15 12
d:22 h:3 f:25

step: 23
1 2 3 4
5 6 7 0
9 10 11 8
13 14 15 12
d:23 h:2 f:25

step: 24
1 2 3 4
5 6 7 8
9 10 11 0
13 14 15 12
d:24 h:1 f:25

step: 25
1 2 3 4
5 6 7 8
9 10 11 12
13 14 15 0
d:25 h:0 f:25

find the target!
A* algorithm execution time: 16837 milliseconds
```

- 搜索路径长度：25step
- 总运行时间：16837millisecond

2.最终结果

曼哈顿距离

样例	成功所用步数	所用时间
1	1	0
2	13	1
3	21	4

样例	成功所用步数	所用时间
4	25	1730

线性冲突

样例	成功所用步数	所用时间
1	1	0
2	13	1
3	21	38
4	25	16837

五、实验分析

1. 启发式策略比较

在本实验中，分别使用了**曼哈顿距离**和**对角线距离**作为启发式函数进行A*算法的求解。以下是两种启发式函数的分析：

- **曼哈顿距离**：该启发式函数基于水平和垂直方向上的距离，适用于方块移动过程中通常的局部调整。由于每个方块与目标位置的水平和垂直距离总是加和的，因此这种距离计算方法可以保证解的可达性和高效性。
- **对角线距离**：对角线距离考虑了每个方块在横向或纵向上的最大移动量。与曼哈顿距离相比，对角线距离在某些情况下可能更有利，尤其是当方块的移动涉及大范围的交换时。

2. 代价函数与启发式选择的关系

A*算法的性能依赖于启发式函数的选择。以下是代价函数与启发式选择的关系分析：

- **代价函数评估**：启发式函数直接影响A*算法的搜索效率和求解时间。较为精确的启发式函数能够更快地指引搜索方向，减少不必要的搜索。
- **曼哈顿距离 vs 对角线距离**：曼哈顿距离通常会导致A*算法在局部移动时效率较高，但对于大范围的移动，可能会需要更多的扩展步骤。相比之下，对角线距离的启发式函数能更好地处理方块大范围的移动，因此在某些情况下可能会减少状态扩展的数量。

3. 时间和空间复杂度

A*算法的时间复杂度受启发式函数的影响。曼哈顿距离虽然计算简单，但可能会导致更多的状态扩展。而对角线距离尽管计算略为复杂，但在某些情形下能够加快搜索过程。因此，选择合适的启发式函数对于提高算法效率至关重要。

六、结果

由上述数据可以看出，曼哈顿距离、线性冲突这两个个启发式函数中**最好的是曼哈顿距离**，可以在满足可采纳的基础上比较快的计算出最短路径。
本实验为进一步探索启发式搜索算法提供了基础，对于其他类似的拼图问题或路径规划问题也具有一定的参考价值。

七、附录

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <math.h>
#include <chrono>  // 用于计时

using namespace std;
using namespace std::chrono;

#define ROW 4
#define COL 4
#define N 0     //north
#define S 1     //south
#define W 2
#define E 3

class state{

public:

    int status = 0;
    int d = 0;       //当前深度值
    int p = 0;       //与目标状态的总距离
    int f = 0;       //函数估算值
    int direction = 0;       // 记录方向
    int index;           //下标
    int pre;             //记录前一状态
    int matrix[ROW][COL];

    state(){}
    state(int matrix[ROW][COL]){         // 构造函数初始化
        memcpy(this->matrix, matrix, sizeof(int)*ROW*COL);
        //pre = NULL;
    }

    bool compare(int t[ROW][COL]) {      //判断当前矩阵状态与矩阵t是否一样
        int sum = 0;
        for(int i = 0; i < ROW; i++)
            for (int j = 0; j < COL; j++) {
                if (matrix[i][j] != t[i][j])
                    return false;
```

```
            }
        return true;
    }

    void setS0(int d, int f) {           //设置起始状态的深度和评价函数估计值
        this->d = d;
        this->f = f;
    }

    void setD(int t[ROW][COL]) {    // 计算当前状态深度
        d += 1;     //深度值加1
        setP(t);

    }
    int diagonalDistance(int t[ROW][COL]) {//对角线距离（Diagonal Distance）
    int dist = 0;
    for (int i = 0; i < ROW; i++) {
        for (int j = 0; j < COL; j++) {
            int value = matrix[i][j];
            if (value != 0) {
                int targetRow = (value - 1) / COL;
                int targetCol = (value - 1) % COL;
                dist += max(abs(i - targetRow), abs(j - targetCol));  // 计
算对角线距离
            }
        }
    }
    return dist;
}

// 计算启发式函数
void setP(int t[ROW][COL]) {

    // 使用对角线距离作为启发式函数

    p = diagonalDistance(t);

    setF();  // 更新估值

}
    // void setP(int t[ROW][COL]) {     // 计算当前状态各点与目标状态t的各点的最
短距离之和，曼哈顿距离（Manhattan Distance）

    //  int num = 0;

    //  for(int i = 0; i < ROW; i++)

    //      for (int j = 0; j < COL; j++) {

    //          for(int m = 0; m < ROW; m++)
```

```
    //              for (int k = 0; k < COL; k++) {
    //                  if (matrix[i][j] == t[m][k] && matrix[i][j] != 0) {
    //                      num += abs(i - m) + abs(j - k);
    //                  }
    //              }
    //      }
    //  p = num;
    //  setF();         //接着设计估评价函数估计值（p+d）
    // }
    void setF() {
        f = p + d;
    }


    bool operator<(const state &temp) const {
        return f < temp.f;
    }


    bool up(int t[ROW][COL]) {
        if (direction != S) {//上一步不是向下移  防止出现来回移动
            int temp;

            direction = N;//这步是向上移
            for (int i = 0; i < ROW; i++) {
                for (int j = 0; j < COL; j++) {
                    if (matrix[i][j] == 0 && i - 1 >= 0) {          //在i j
位置是0
                        temp = matrix[i][j];
                        matrix[i][j] = matrix[i - 1][j];
                        matrix[i - 1][j] = temp;
                        setD(t);        //设置当前状态d p f
                        return true;
                        break;
                    }
                }
            }
        }
        setD(t);
        return false;
```

```
    }
    bool down(int t[ROW][COL]) {

        if (direction != N) {//上一步不是向上移
            int temp;
            direction = S;//这步是向下移

            for (int i = 0; i < ROW; i++) {
                for (int j = 0; j < COL; j++) {
                    if (matrix[i][j] == 0 && i + 1 < ROW) {
                        temp = matrix[i][j];
                        matrix[i][j] = matrix[i + 1][j];
                        matrix[i + 1][j] = temp;
                        setD(t);
                        return true;
                        break;

                    }

                }

            }

        }
        setD(t);
        return false;
    }

    bool left(int t[ROW][COL]) {
        if (direction != E) {//上一步不是向右移

            int temp;
            direction = W;//这步是向左移

            for (int i = 0; i < ROW; i++) {
                for (int j = 0; j < COL; j++) {
                    if (matrix[i][j] == 0 && j - 1 >= 0) {

                        temp = matrix[i][j];
                        matrix[i][j] = matrix[i][j - 1];
                        matrix[i][j - 1] = temp;
                        setD(t);

                        return true;

                        break;
                    }
                }
            }
        }
```

```
        setD(t);
        return false;
    }

    bool right(int t[ROW][COL]) {
        if (direction != W) {//上一步不是向左移

            int temp;
            direction = E;//这步是向右移
            for (int i = 0; i < ROW; i++) {
                for (int j = 0; j < COL; j++) {
                    if (matrix[i][j] == 0 && j + 1 < COL) {
                        temp = matrix[i][j];
                        matrix[i][j] = matrix[i][j + 1];
                        matrix[i][j + 1] = temp;
                        setD(t);
                        return true;
                        break;

                    }
                }
            }
        }

        setD(t);
        return false;
    }

    void show() {           //显示当前状态    并且显示对应的深度值 与目标的差距 评价函数估计值

        for (int i = 0; i < ROW; i++) {
            for (int j = 0; j < COL; j++) {
                cout << matrix[i][j] <<" ";

            }
            cout << endl;
        }
        cout << "d:" << d << "   " << "h:" << p << "   " << "f:" << f << endl;
        cout << endl;
    }
};

vector<state> store;

class A_star{

public:
```

```
    vector<state> open;
    vector<state> close;
    state start;
    state end;
    int count = 0;
    A_star() {}

    A_star(state start, state end){          //构造函数

        this->start = start;

        this->end = end;

        this->start.setS0(-1, 0);

        this->start.setD(end.matrix);

        open.push_back(this->start);         //首状态进入open表

    }



    bool inOpen(state temp) {        // 判断是否在open表

        vector<state>::iterator it;

        int count = 0;

        for (it = open.begin(); it != open.end(); it++) {

            if (temp.compare(it[0].matrix) && temp.f < it[0].f) {    //如果新
节点在open表中 用新节点替换旧节点 IF 新节点的f值小于旧节点

                open.erase(open.begin() + count);        //删除旧结点

                open.push_back(temp);        //插入新节点

                return true;

            }

            count++;

        }

        return false;

    }
```

```
bool inClose(state temp) {      // 判断是否在close表

    vector<state>::iterator it;

    int count = 0;

    for (it = close.begin(); it != close.end(); it++) {

        if (temp.compare(it[0].matrix) && temp.f < it[0].f) {

            close.erase(close.begin() + count);//删除旧结点

            open.push_back(temp);//插入新节点

            return true;

        }

        count++;

    }

    return false;

}



state transfer(int t[ROW][COL]) {   //从起始状态开始拓展 t是最终状态的矩阵值

    while (true) {
        if (open.empty()){// open表为空查询失败
            cout<<"find error!"<<endl;
            return 0;
        }

        state cur = open.front();
        int st;     //储存p在vector中的下标
        store.push_back(cur);                //将所有经历的状态进行储存

        for(int i=0;i<store.size();i++){

            int sum=0;

            for(int j=0;j<ROW;j++){
                for(int k=0;k<COL;k++){
                    if(cur.matrix[j][k]==store[i].matrix[j][k])
                            sum++;
                }
            }
```

```
            if(sum==ROW*COL){
                st=i;
                break;          //代表找到p在store中对应的下标
            }
        }

        cur.index=st;

        if (cur.compare(t)){        // 判断是否为目标状态
            return cur;               //若是 则代表找到 并且返回当前state
        }
        //针对于open表的首元素进行状态拓展

        state tempup = cur;

        if (tempup.up(t)) {     // 上移
            tempup.pre=st;          //当前tempup已经发生了变化
            if (inOpen(tempup) == 0 && inClose(tempup) == 0)    //若新节
点在open表和close中均未出现
                open.push_back(tempup);     //则open表加入新节点

        }

        state tempdown = cur;
        if (tempdown.down(t)) {     // 下移
            tempdown.pre=st;
            if (inOpen(tempdown) == 0 && inClose(tempdown) == 0)
                open.push_back(tempdown);

        }

        state templeft = cur;

        if (templeft.left(t)) {     // 左移
            templeft.pre=st;
            if (inOpen(templeft) == 0 && inClose(templeft) == 0)
                open.push_back(templeft);
        }

        state tempright = cur;

        if (tempright.right(t)) {       // 右移
            tempright.pre=st;
            if (inOpen(tempright) == 0 && inClose(tempright) == 0)
                open.push_back(tempright);

        }

        open.erase(open.begin());       // 从open表删除评价函数估计值最小的
```

```
元素
            close.push_back(cur);       // 将评价函数估计值最小的元素加入close表
            sort(open.begin(), open.end()); //根据评价函数估计值进行排序
        }
    }



    void printStep(int index){      //路径输出

        vector<state> print;        //储存输出路径
        print.push_back(store[index]);          //将状态路径储存
        index=store[index].pre;

        while(index!=0){                //起始状态的下标在store中对应0 因此在
print数组中没有加入起始状态
            print.push_back(store[index]);          //将状态路径储存
            index=store[index].pre;

        }

        cout<<"Print steps:"<<endl;

        for(int i=print.size()-1;i>=0;i--){
            cout<<"step: "<<print.size()-i<<endl;
            for(int j=0;j<ROW;j++){
                for(int k=0;k<COL;k++){
                    cout<<print[i].matrix[j][k]<<" ";    //打印状态对应的矩阵
                }
                cout<<endl;

            }
            cout<<"d:"<<print[i].d<<" h:"<<print[i].p<<" f:"
<<print[i].f<<endl<<endl;

        }
        cout<<"find the target!"<<endl;

    }

};


int main(){

    int s[ROW][COL]={ { 3,  2,  7,  8 },

                      { 1,  6,  4, 12 },

                      { 5, 10, 11, 15 },
```

```
                    { 9, 13, 0,  14 } };

    int e[ROW][COL]={{1, 2, 3, 4},

                     {5, 6, 7, 8},

                     {9, 10, 11, 12},

                     {13, 14, 15, 0}};

    state start(s);              //设置起始状态
    state end(e);                //设置终止状态
    cout<<"start state:"<<endl;
    start.show();

    A_star arrive(start,end);       //设置起始状态的评价函数估计值

    auto start_time = high_resolution_clock::now();
    state last=arrive.transfer(e);
    auto end_time = high_resolution_clock::now();
    auto duration = duration_cast<milliseconds>(end_time - start_time);

    arrive.printStep(last.index);            //输出路径状态
    cout << "A* algorithm execution time: " << duration.count() << "
milliseconds" << endl;
    return 0;
}
```