**实 验 报 告**

**学 院：** 计算机科学与技术学院

**班 级：** 计非211

**学 号：** 202131990522

**姓 名：** 嵇纹垲

**课 程：** 人工智能基础

**指导教师：** 周昌军

**完成时间：** 2024 **年** 3 **月** 27 **日**

**浙 江 师 范 大 学 制**

**A\*算法求解八数码问题**

**问题分析**

实验要求用A\*算法来实现将一个任意排列变换成一个指定排列。

解的存在性：A\*算法的本质是启发式搜索，可以以一个较快的速度找到解。但是需要注意的是，并不是任意一个排列都是有解的。我们不妨用0来代替原本状态里的空格，那么可以证明，如果两个排列的逆序数和奇偶性不同，该问题就是无解的，否则有解。

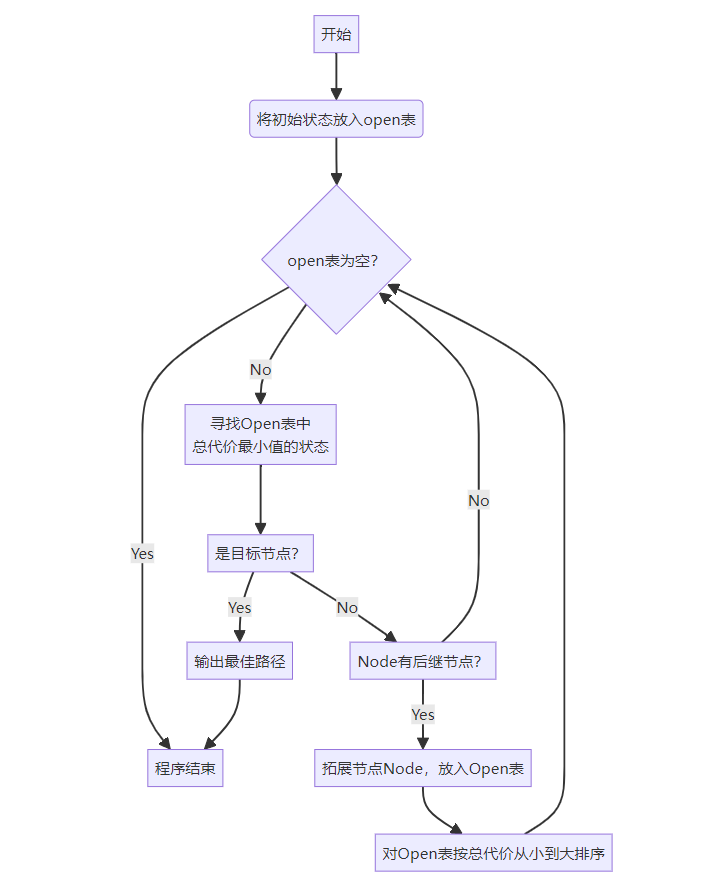
搜索策略：为了找到解决方案，我们需要一种有效的搜索策略。A\*算法是一种启发式搜索算法，它使用一个估计值来指导搜索的方向。在这个问题中，我们可以使用启发式函数来估计两个状态之间的距离，从而，每次转移到代价最小的点去，可以更快到达终点状态。

评估函数：可考虑每个状态n的估价函数值为两个分量：从起始状态到状态n的实际代价g(n)以及从状态n到达目标节点的估价代价法f(n)，这里我们将**g(n)定义为走到状态n的最小步数**，这是一个很自然的定义，因为它就代表了到达该状态的代价。定义**f(n)为状态n与目标状态的曼哈顿距离**，也就是对应数字的曼哈顿距离之和。

边界条件：在搜索过程中，我们需要设置一些边界条件来避免搜索不必要的状态。例如，如果一个状态已经证明不是解决方案的一部分，我们可以将其标记为不可达，并从搜索中排除。

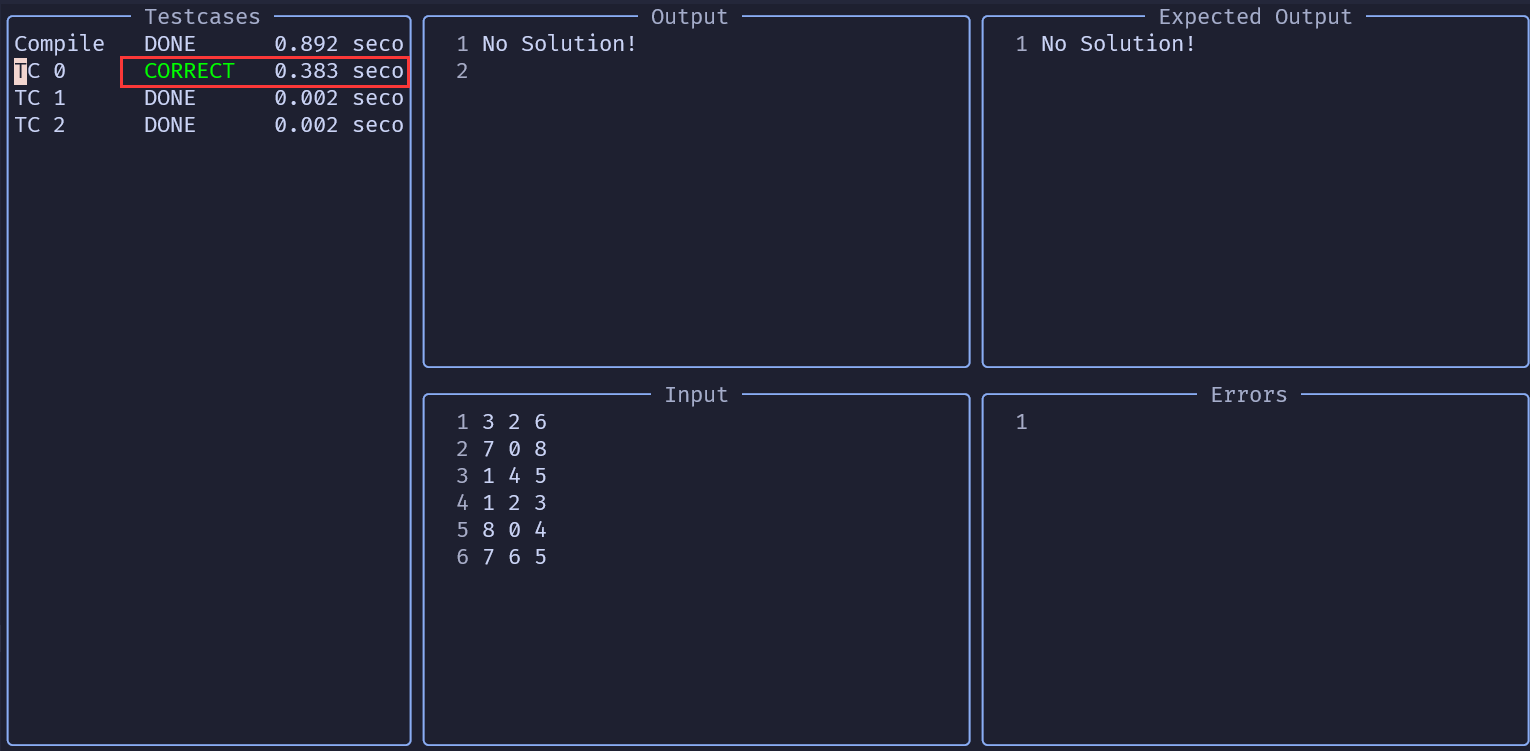
**算法设计**

A\*算法可以通过宽度优先搜索来实现。以下是程序的大致流程图



为了方便，我们可以使用一个优先队列来实现Open表。这需要我们自定义排序方式，那么就按照之前所分析的，取f(n)+g(n)的尽可能小的状态放在队头即可

另外一开始就可以把是否有解给判断出来，可以避免不必要的搜索，比如下图中无解的情况耗时是有解的情况的数百倍，因为无解的情况搜索了所有的状态



**代码**

#include <algorithm>

#include <array>

#include <iostream>

#include <map>

#include <queue>

#include <utility>

constexpr int INF = 0x3f3f3f3f;

constexpr int N = 3;

constexpr int dir[4][2] = {

{ 0, 1},

{ 1, 0},

{ 0, -1},

{-1, 0},

};

using Table = std::array<std::array<int, N>, N>;

Table tar{};

struct Node {

Table cur{};

int ManhattanDis = 0, minStep = INF, lst;

Node() : cur({}) {}

Node(Table c, int step, int l) : cur(c), minStep(step), lst(l) {

for (int x0 = 0; x0 < N; ++x0)

for (int y0 = 0; y0 < N; ++y0)

for (int x1 = 0; x1 < N; ++x1)

for (int y1 = 0; y1 < N; ++y1)

if (cur[x0][y0] && cur[x0][y0] == tar[x1][y1])

ManhattanDis += abs(x1 - x0) + abs(y1 - y0);

}

std::pair<int, int> getPos() const {

for (int i = 0; i < N; ++i)

for (int j = 0; j < N; ++j)

if (!cur[i][j]) return {i, j};

return {};

}

int getHashValue() const {

int value = 0;

for (int i = 0; i < N; ++i)

for (int j = 0; j < N; ++j)

value = value \* 10 + cur[i][j];

return value;

}

bool operator<(const Node &rhs) const {

return ManhattanDis + minStep > rhs.ManhattanDis + rhs.minStep;

}

bool isTarget() const { return cur == tar; }

void print() const {

for (int i = 0; i < N; ++i)

for (int j = 0; j < N; ++j)

std::cout << cur[i][j] << " \n"[j == N - 1];

}

};

std::map<int, int> dis;

std::map<int, Node> node;

int main() {

std::ios::sync\_with\_stdio(false);

std::cin.tie(nullptr);

Table t;

for (int i = 0; i < N; ++i)

for (int j = 0; j < N; ++j)

std::cin >> t[i][j];

for (int i = 0; i < N; ++i)

for (int j = 0; j < N; ++j)

std::cin >> tar[i][j];

std::priority\_queue<Node> q;

q.push(Node(t, 0, 0));

Node Ans;

while (!q.empty()) {

Node cur = q.top();

q.pop();

if (cur.isTarget()) {

dis[cur.getHashValue()] = cur.minStep;

Ans = cur;

break;

}

if (dis.count(cur.getHashValue()) && dis[cur.getHashValue()] <= cur.minStep)

continue;

node[cur.getHashValue()] = cur;

dis[cur.getHashValue()] = cur.minStep;

auto [x, y] = cur.getPos();

for (int i = 0; i < 4; ++i) {

int nx = x + dir[i][0];

int ny = y + dir[i][1];

if (nx < 0 || ny < 0 || nx >= N || ny >= N) continue;

Table pre = cur.cur;

std::swap(pre[x][y], pre[nx][ny]);

Node nxt = Node(pre, cur.minStep + 1, cur.getHashValue());

q.push(nxt);

}

}

if (Ans.getHashValue()) {

std::cout << "Find Solution!\n";

std::cout << "MinStep is " << Ans.minStep << '\n';

std::cout << '\n';

std::vector<Node> vec;

while (Ans.getHashValue()) {

vec.push\_back(Ans);

Ans = node[Ans.lst];

}

std::reverse(vec.begin(), vec.end());

for (auto i : vec) {

std::cout << "Step " << i.minStep << '\n';

for (int j = 0; j < N; ++j)

for (int k = 0; k < N; ++k)

std::cout << i.cur[j][k] << " \n"[k == N - 1];

std::cout << '\n';

}

} else {

std::cout << "No Solution!\n";

}

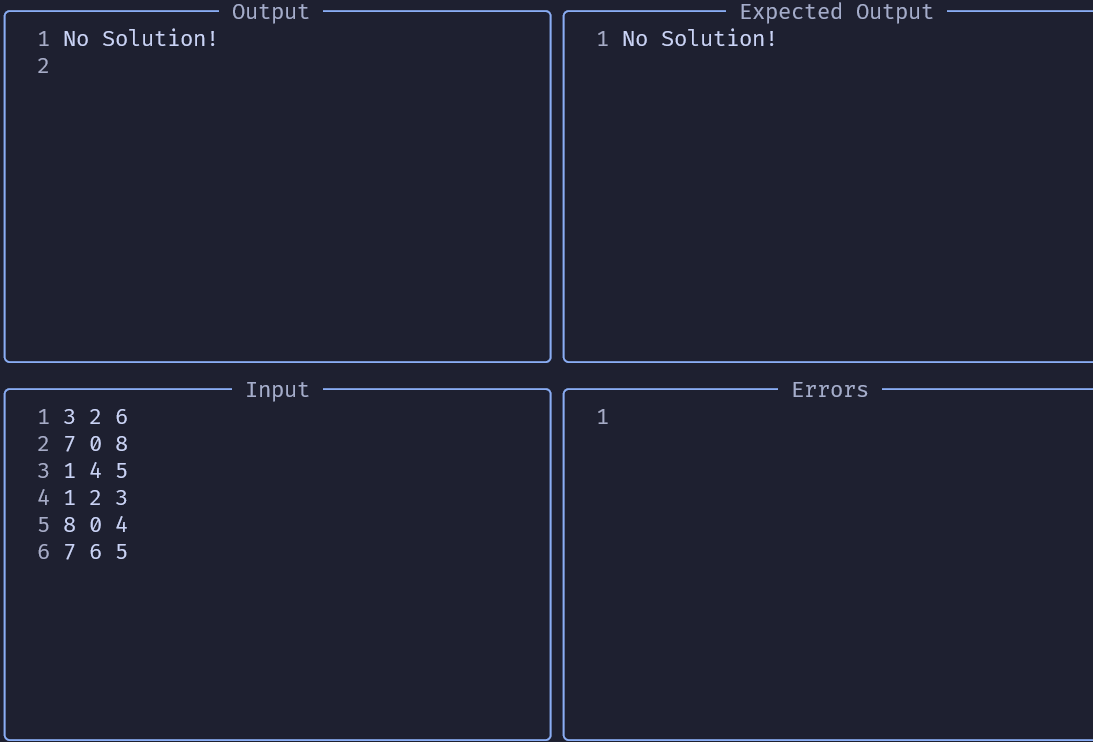
return 0;

}

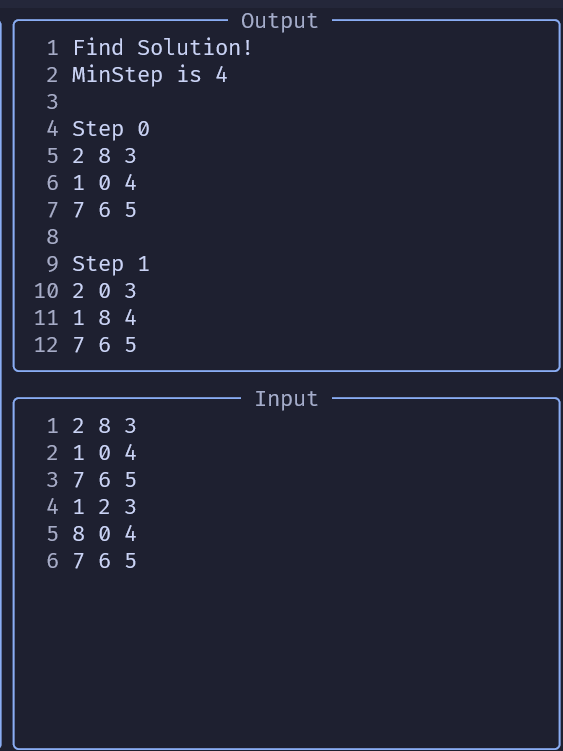
**运算结果**

输入是一个初始状态以及终止状态。如果状态有解，输出最小步骤以及每一步的变化。否则输出No Solution！表示无解

对于给定的样例，输出结果如下。这是因为它们的逆序对奇偶性不同



我们换一个样例



输入：

2 8 3

1 0 4

7 6 5

1 2 3

8 0 4

7 6 5

输出

Find Solution!

MinStep is 4

Step 0

2 8 3

1 0 4

7 6 5

Step 1

2 0 3

1 8 4

7 6 5

Step 2

0 2 3

1 8 4

7 6 5

Step 3

1 2 3

0 8 4

7 6 5

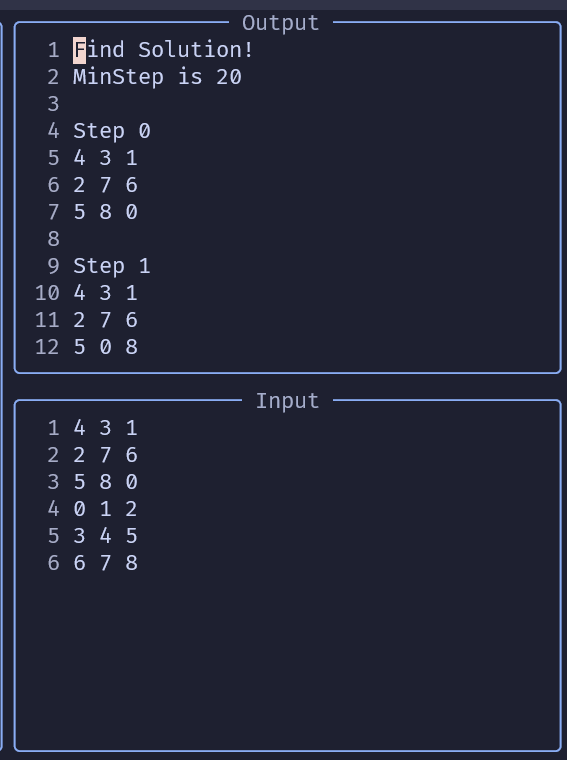
Step 4

1 2 3

8 0 4

7 6 5

可以发现这个确实有解，并且只需要四步即可。可以证明4步确实是最优解



输入：

4 3 1

2 7 6

5 8 0

0 1 2

3 4 5

6 7 8

输出：

Find Solution!

MinStep is 20

Step 0

4 3 1

2 7 6

5 8 0

Step 1

4 3 1

2 7 6

5 0 8

Step 2

4 3 1

2 0 6

5 7 8

Step 3

4 3 1

2 6 0

5 7 8

Step 4

4 3 1

2 6 8

5 7 0

Step 5

4 3 1

2 6 8

5 0 7

Step 6

4 3 1

2 0 8

5 6 7

Step 7

4 3 1

0 2 8

5 6 7

Step 8

4 3 1

5 2 8

0 6 7

Step 9

4 3 1

5 2 8

6 0 7

Step 10

4 3 1

5 2 8

6 7 0

Step 11

4 3 1

5 2 0

6 7 8

Step 12

4 3 1

5 0 2

6 7 8

Step 13

4 3 1

0 5 2

6 7 8

Step 14

0 3 1

4 5 2

6 7 8

Step 15

3 0 1

4 5 2

6 7 8

Step 16

3 1 0

4 5 2

6 7 8

Step 17

3 1 2

4 5 0

6 7 8

Step 18

3 1 2

4 0 5

6 7 8

Step 19

3 1 2

0 4 5

6 7 8

Step 20

0 1 2

3 4 5

6 7 8

**实验总结**

通过本次实验，我对A\*算法以及广度优先搜索有了更深的理解。A\*算法的本质是在搜索的基础上加以优化，通过引入代价函数的概念来实现在搜索过程中动态调整搜索策略，从而能够更快地找到最优解。