**代码开发文档：**

**1. 概述**

本程序实现了一个八数码问题求解的算法。八数码问题是一个经典的组合问题，要求在 3x3 的棋盘上通

过空格将一个数字移动到正确的位置，使得棋盘上的数字从左到右从上到下依次递增。本程序使用 A\*

算法来求解八数码问题。

**2. 输入输出**

程序通过命令行输入初始状态，输出 A\* 算法的搜索路径 path\_ 和移动序列 move\_path\_ 。

**3. 主要功能**

**3.1 初始化八数码问题**

通过输入初始状态，创建一个 EightDigital 对象 e1 ，并将其初始状态设置为输入的状态。

**3.2 判断问题是否可解**

**3.2.1 概述**

使用 HasSolution 函数判断初始状态是否可解（采用**逆序对数**判断八数码问题可解性）。bool EightDigital::HasSolution(string state) {

int num = 0;//计算逆序对数

for (int i = 1; i < int(state.size()); ++i) {

if (state[i] != '0') {

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

if (state.at(i) < state.at(j) && state[j != '0']) {

++num;

}

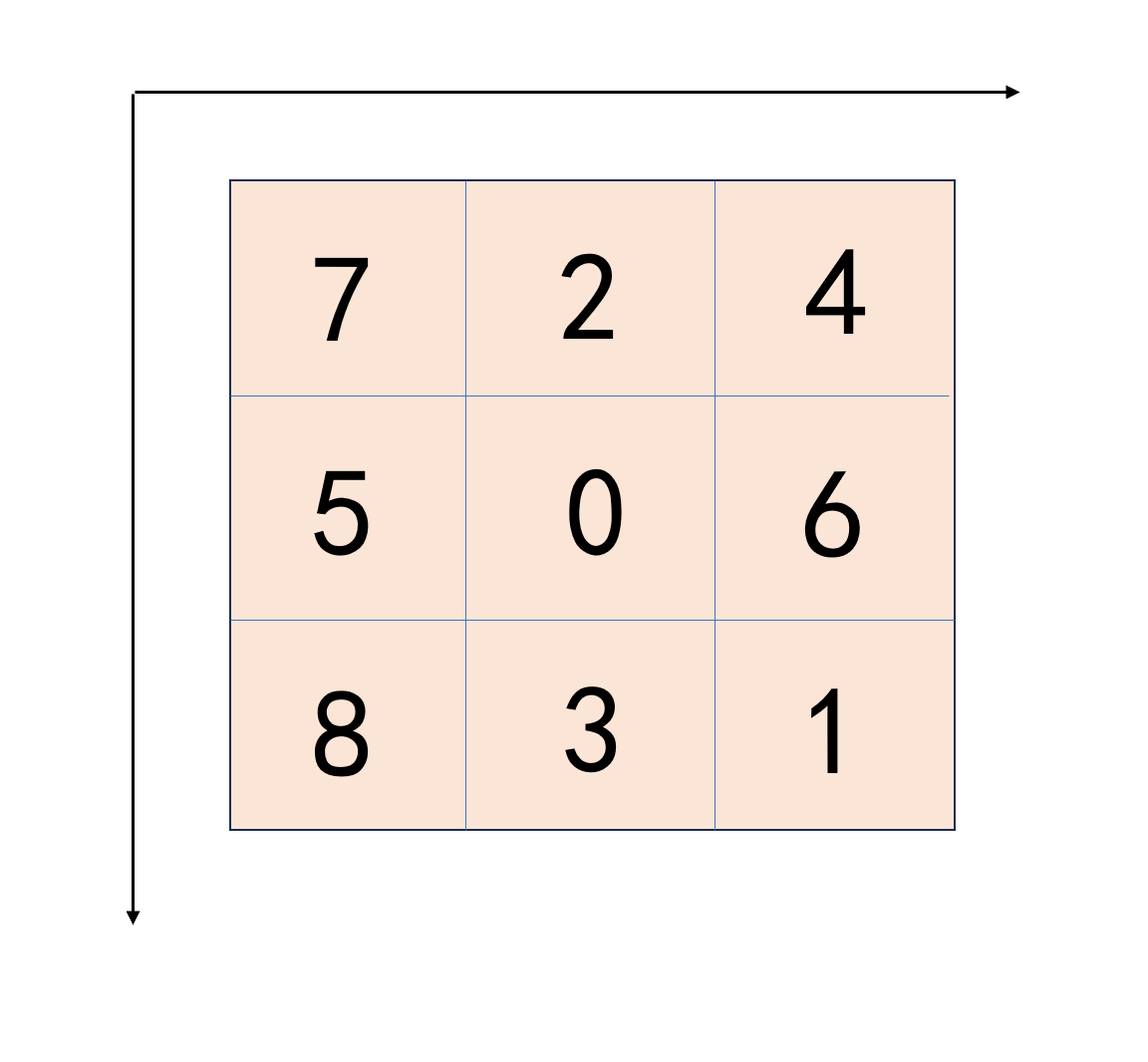
}

}

}

return num % 2 == 0;//判断奇偶性

}



八数码问题中除去空格（字符串中表示为'0'），如果一个数字的左边和上边的某个数字比它大，那么这

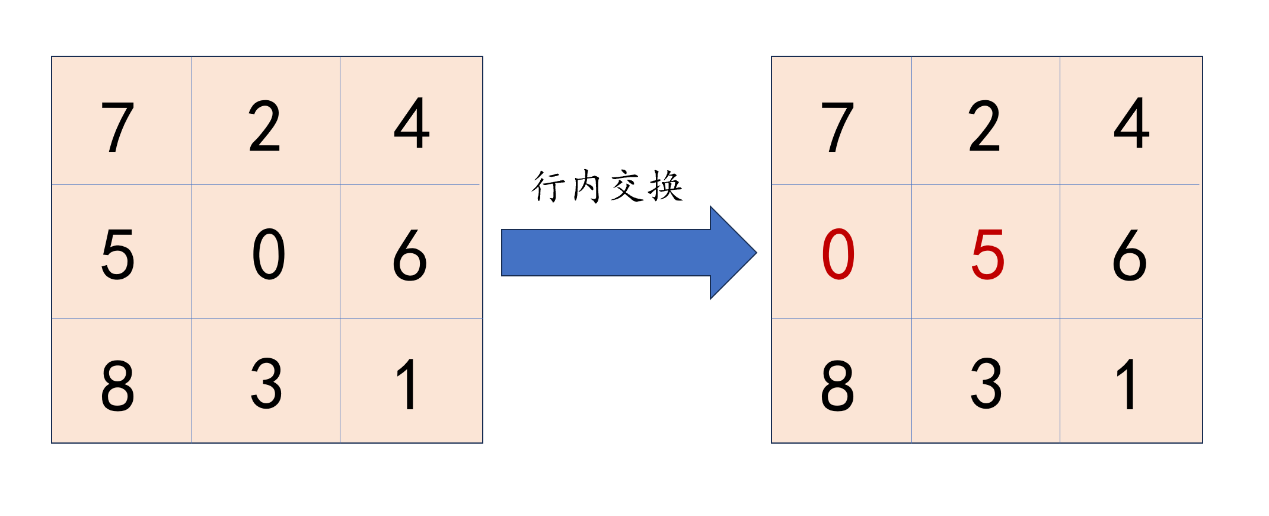
两个数字的组合就是一个逆序对。**一个数字序列的逆序对数如果为偶数，那么这个数字序列是可解的**。

**3.2.2 原因详陈如下：**

**初始状态的逆序对数为 0**，因为初始状态中没有逆序对。由于新状态仅由空格的移动产生，可分为两类

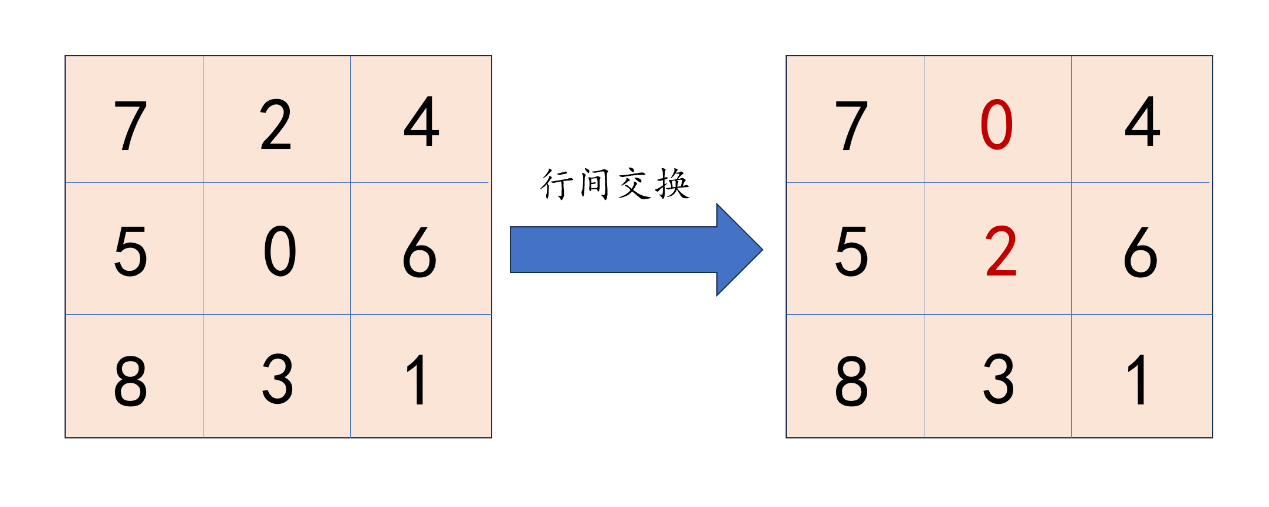
情况。

1. **行内移动**



我们不考虑元素 0 ，显然，行内交换**不改变**原序列的逆序对数。

1. **行间移动**



行间交换仅涉及3个元素，如上例 245 --> 452 ，逆序对数加2。

并可证明行间交换逆序对数的变化**仅限于 2 、0**。

**综上所述，仅在数字序列的逆序对数为偶的情形下，八数码问题可解。**

**3.3 A\* 算法实现**

如果问题可解，使用 Astar 函数实现 A\* 算法。

**3.4 输出搜索路径和移动序列**

使用 PrintAstarPath 函数输出搜索路径和移动序列。**4. 代码实现**

**4.1 定义 EightDigital 类**

定义一个 EightDigital 类，用于存储八数码问题的初始状态、目标状态、移动规则和 A\* 算法的相关属

性。

**4.2 实现 A\* 算法**

实现 A\* 算法的主要步骤如下：

1. 将初始状态加入 open 列表。（open表记作 open\_ 是一个**优先队列**，队头即 f 值最小的节点，见4.4

详细说明）

2. 当 open 列表不为空时，执行以下操作：

利用 top() 函数从open表取队头，将其作为当前状态。

如果当前状态是目标状态，则结束搜索。

否则，将当前状态从 open 列表中移除，并将其加入 close 列表。

对当前状态的相邻状态进行扩展，将可到达的空格移动到相邻状态，得到新的状态。

对于每个新状态，计算 g 值（**深度**）和 h 值（**曼哈顿距离**），并将它加入 open 列表（**曼哈顿**

**距离**和**深度**的代码实现，见4.5，4.6详细说明）。

重复以上步骤，直到找到目标状态。

**4.3 输出搜索路径和移动序列**

根据记录的父节点信息，从目标状态开始，依次回溯到初始状态，输出移动序列。

**4.4 优先队列**

在 A\* 算法中，需要使用优先队列来存储 open 表。优先队列的实现需要满足以下要求：

1. 支持插入、删除、查找、获取最小值等基本操作。

2. 支持按照优先级排序。

open 表的优先级由 f 值（**深度+曼哈顿值**）决定，f 值越小，优先级越高。其底层的实现是一个**最小**

**堆**，来维护open表中的元素按照优先级排序。open表的代码定义如下：

priority\_queue<pair<string, int>, vector<pair<string, int>>, Cmp>open\_;pair<string, int> ：表示队列中的元素是一个键值对，键是一个字符串（表示状态），值是一个整数

（表示f值）。

vector<pair<string, int>> ：表示队列的底层实现是一个vector容器，用于存储上述键值对。

Cmp ：是一个**比较类**（compare class），用于定义优先队列的元素比较规则。其代码定义如下：

//以下结构用于实现运算符重载，用以实现最小优先队列

struct Cmp {

bool operator()(pair<string, int>& a, pair<string, int>& b) {

return a.second > b.second;

}

};

它实现了C++的仿函数 operator() （**lambda表达式**），用于定义一个**比较函数**。这个比较函数接收两

个pair类型的参数，并返回一个布尔值，表示第一个参数的第二个元素是否大于第二个参数的第二个元

素。优先队列 priority\_queue 将依照此规则对元素进行排序。

**4.5 计算曼哈顿距离 h 值**

在 A\* 算法中，需要计算当前状态到目标状态的曼哈顿距离。代码实现如下：

int EightDigital::Inspire(string state) {

int cost = 0;//记录代价，曼哈顿距离

for (int i = 0; i < 9; ++i) {

if (state[i] != '0') {

cost += abs((state[i] - '0') / 3 - i / 3) + abs((state[i] - '0') % 3 - i

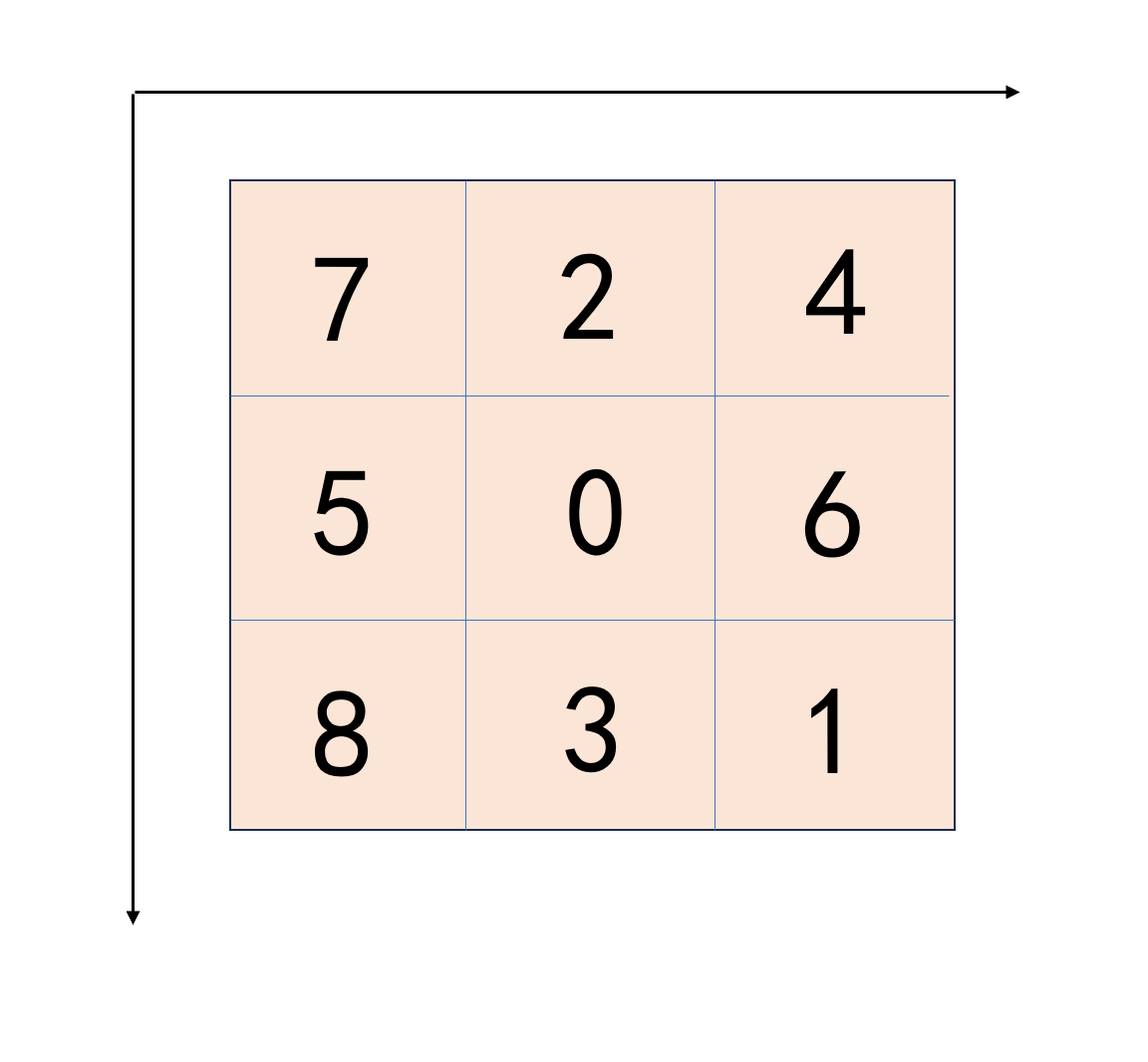
}

}

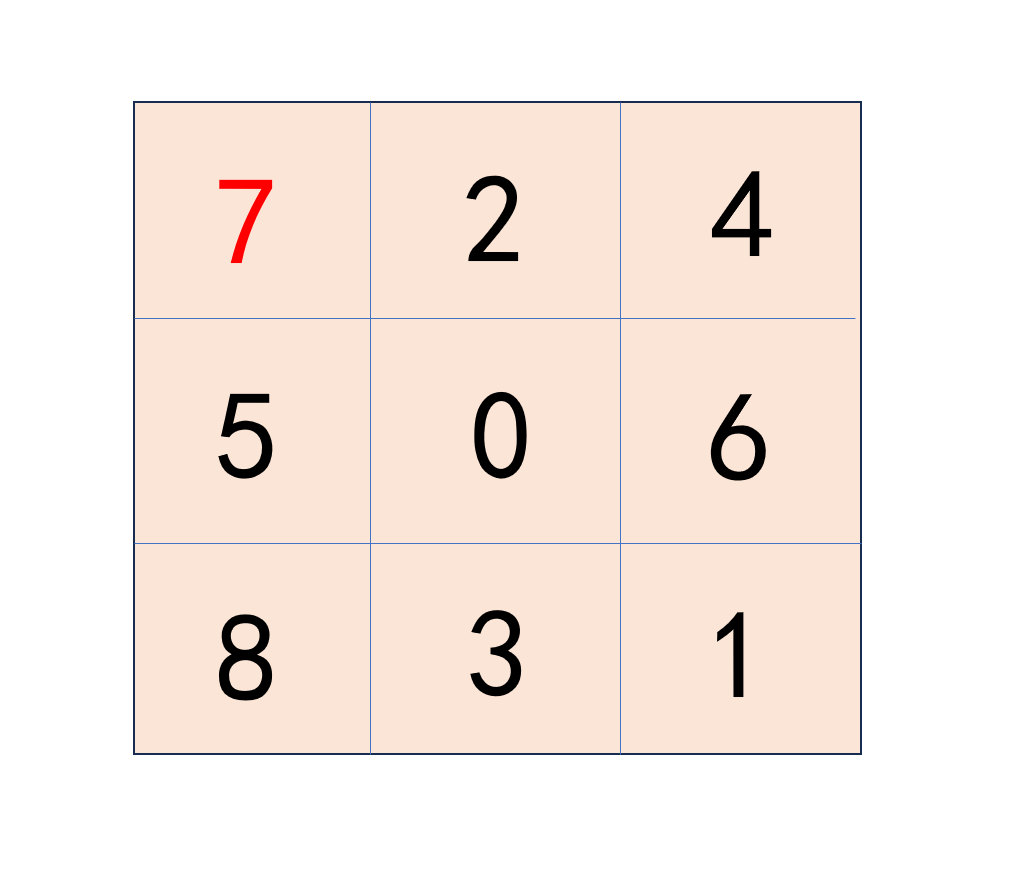
return cost;

}

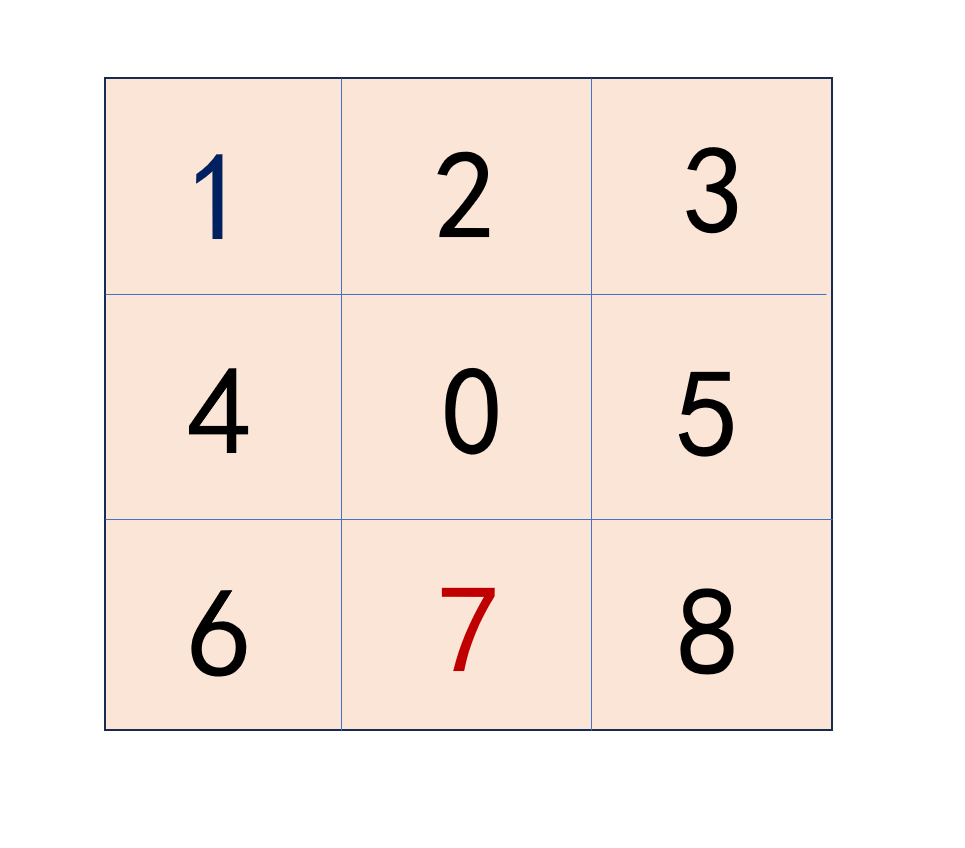
再以前面列举的例子说明：



**当前状态**元素 7 的行序为 i/3 = 0 ,列序为 i%3 = 0 。



**目标状态**元素 7 的行序为 (state[i] - '0') / 3 = 2 , 列序为 (state[i] - '0') % 3 = 1 。



所以**行间的绝对距离** abs((state[i] - '0') / 3 - i / 3) 为 2 - 0 = 2。

**列间的绝对距离** abs((state[i] - '0') % 3 - i % 3) 为 1 - 0 = 1。

曼哈顿距离等于**行间的绝对距离**加上**列间的绝对距离**，即 2 + 1 = 3 。

**4.6 计算深度 g 值**

此算法中，我们定义了一个**全局深度表** map\_depth\_ ，其中key为当前状态，value为当前状态的深度。

map<string, int> map\_depth\_;

当我们扩展得到新状态时，需要判断该状态是否已经扩展过，即是否在 map\_depth\_ 中。if (map\_depth\_.find(state) == map\_depth\_.end()) {

......

}

如果不在深度表中，即当前状态**未扩展**过，则需要将当前状态的深度值 depth 加一，存入深度表，压入

open表，在open表中按照 f = g + h 的规则进行排序。

如果在深度表，但是 depth 值**小于**当前状态的深度值，则需要更新深度表，移出close表，压入open

表，按照 f = g + h 的规则进行排序。

在此过程中，我们还定义了一个全局的**父子关系表** map\_path\_ ，用以记录当前状态的父状态，以便回溯

搜索路径。该表伴随深度表的更新而更新。

**5. 写在最后**

添加了测试文件，分别是**test**文件夹内的 test.py 和主目录下的 test\_main.cpp 文件。

test.py 文件用以生成一些随机状态，测试的样例数可自行指定。

test\_main.cpp 文件用以测试八数码的搜索算法。

同一项目内包含了多个**main**函数入口，使用时需进行**更名**。