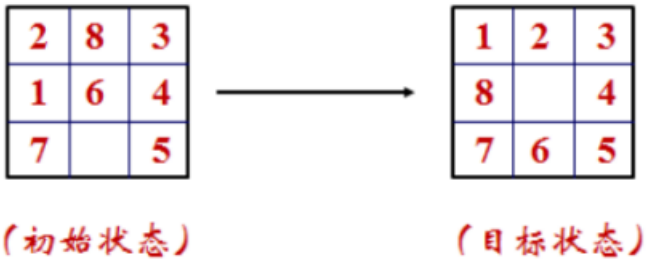
重排九宫问题

1. 问题描述

1. 在3×3的方格棋盘上放置分别标有数字1，2，3，4，5，6，7，8的8张牌
2. 初始状态为S_0，目标状态为 S_g，如下图所示



3. 可使用的算符有空格左移、空格上移、空格右移和空格 下移
4. 要求寻找从初始状态到目 标状态的路径。

2. 算法描述

2.1. 判断结点是否重复

1. 康托展开：

$$X = a[n]\cdot(n-1)! + a[n-1]\cdot(n-2)! + \cdots + a[i]\cdot(i-1)! + \cdots + a[1]\cdot 0!$$

其中，$a[i]$表示原数第i位在当前未出现元素中排在第几个，且$0 \le a[i] < i$，$1 \le i \le n$

2. 通过康托展开，可以将结点状态映射为$0 \to n!-1$间的整数，从而可判断结点的重复性

2.2. 判断问题是否有解

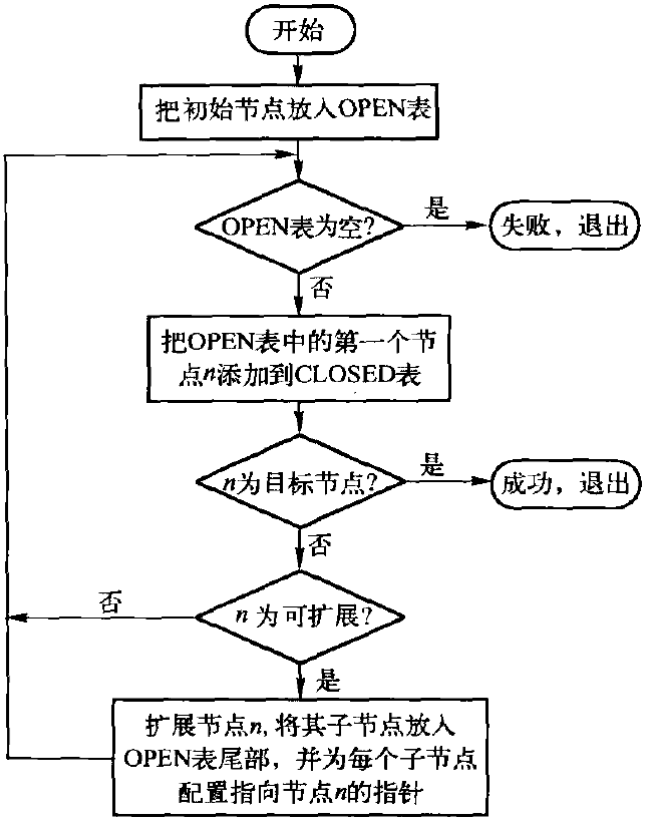
1. 个状态表示成一维的形式
2. 求出：除 0 之外所有数字的逆序数之和，称为这个状态的逆序
3. 若两个状态的逆序奇偶性相同，则可相互到达，否则不可相互到达。

2.3. 广度优先搜索

1. 概述

   1. 基本思想：初始节点开始，逐层扩展并考察(是否为目标节点)结点，一层结点全扩展考察完，再扩展下一层的结点
   2. OPEN表中的结点按照进入顺序排列
   3. 特点是：盲目性大，但一定能得到最短解(反之深度优先不一定得到的是最短解)

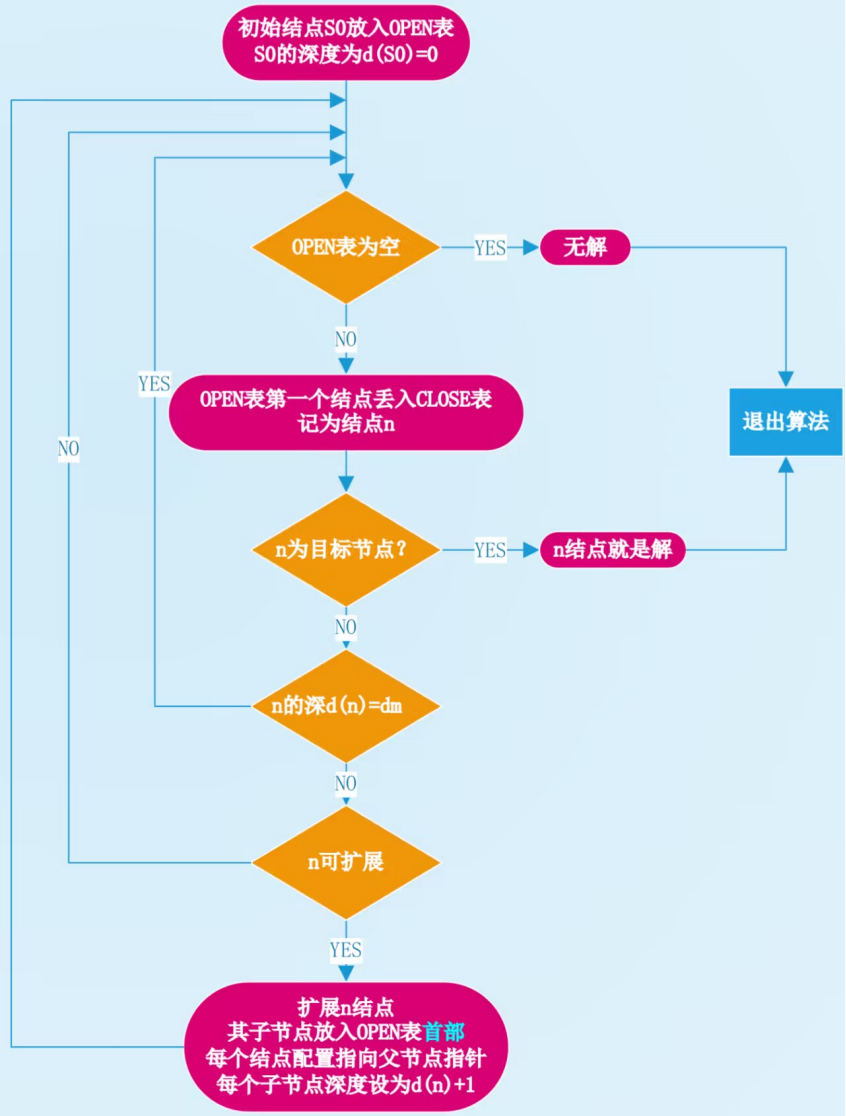
2. 广度优先搜索过程：注意以下将其子节点放入OPEN表尾部，尾部改为首部就是深度优先了



2.4. 有界深度优先算法

1 基本思想：深度优先搜索的时候设置界限d_m，搜索达到深度界限后，不管找没找到都换个分支

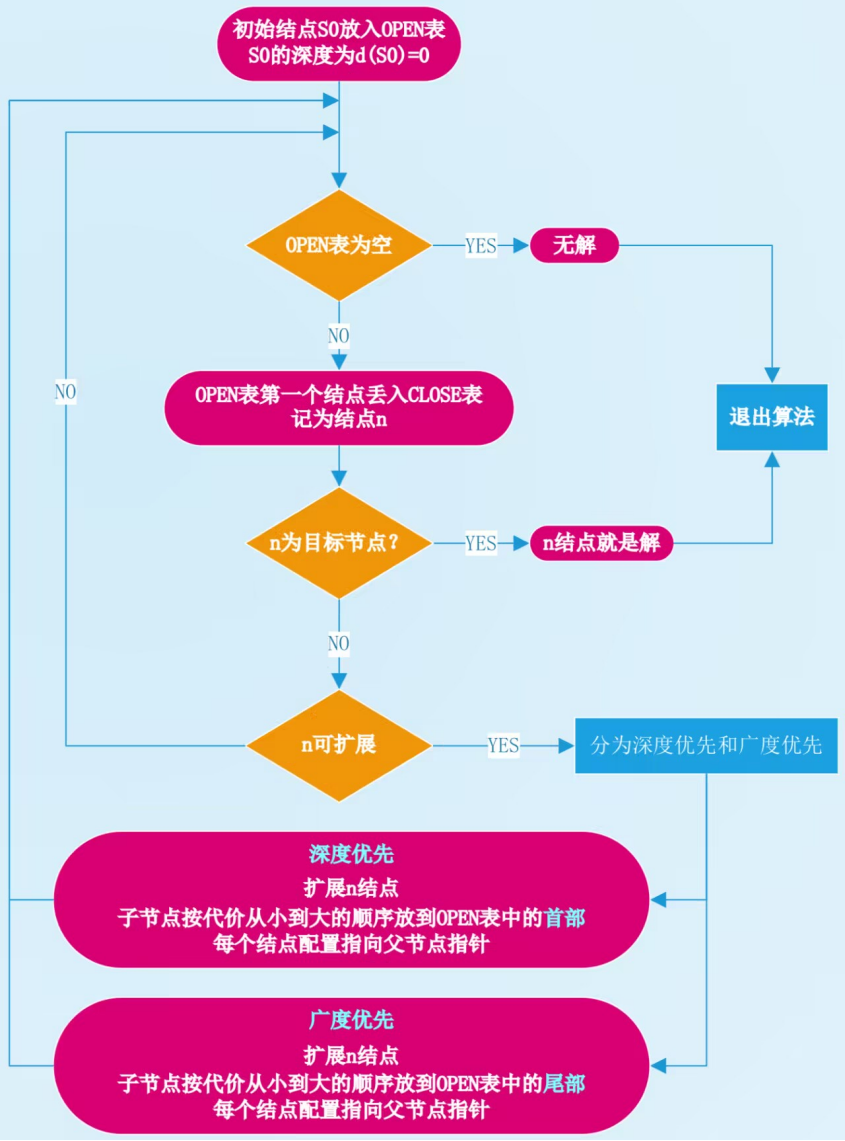
2 流程：



3. 特点：由于深度限制不一定能找到解，所以深度设置也困难，找到的也不一定是最优的
4. 改进1：先设置一较小d_m，当到达了d_m深度还未找到，就加深深度
5. 改进2：增加一个表R
    1. 每找到一个目标节点就，把他放进R并且设置d_m为其路径长
    2. 然后以d_m有限度的深度优先搜索
    3. 不断找到目标节点不断更新d_m，最后一定得到最优解

2.5. 代价树的广度/深度优先搜索(本实验中暂略)

1. 基本概念
    1. 代价树：边上标有代价的树
    2. 代价：$g(x)$是初始结点S_0到结点x的代价，$c(x_1,x_2)=g(x_2)-g(x_1)$是两结点之间的代价
2. 基本思想：OPEN表中的节点按代价从小到大排序，每次取出代价最小的结点
3. 特点：代价树的广度优先一定可以求得最优解
4. 过程



2.6. 启发式搜索

1️⃣启发性信息：指导搜索，与问题有关的信息

2️⃣估价函数：评估结点重要性，$f(x)=g(x)+h(x)$

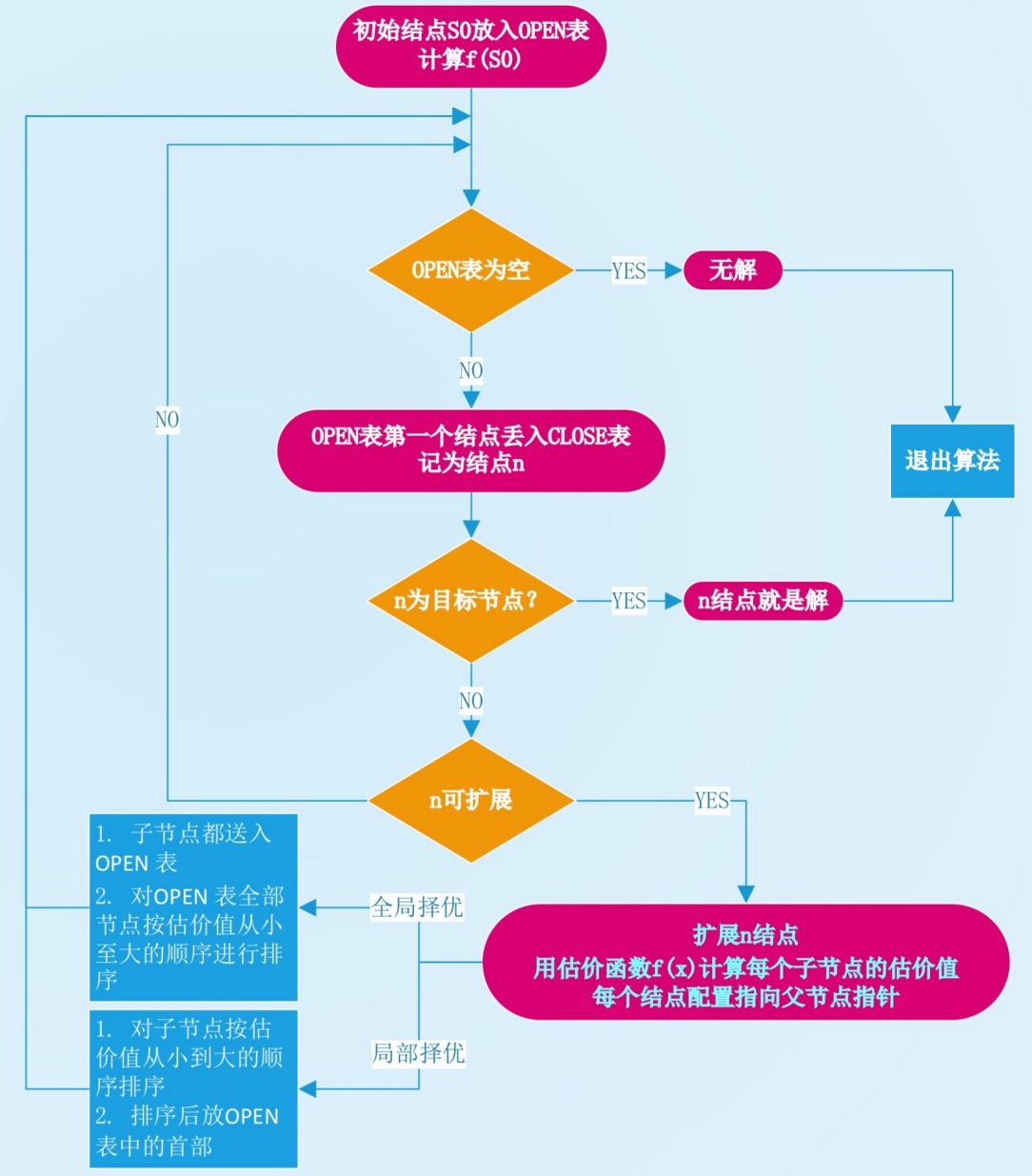
	$g(x)$	$h(x)$**启发函数**
含义	初始节点→x的代价	x→目标结点，最优路径的预估代价
特点	有利于完备性，不利于效率	不利于完备性，有利于效率

➕重拍九宫中，$h(x)$是x的格局与目标节点格局不相同的牌数

3️⃣局部择优搜索：一个结点扩展后，按$f(x)$对子节点计算估计价值，选择价值最小的下一个考察

深度优先+代价树深度优先，均为局部择优搜索的特例

4️⃣全局择优搜索：一个结点扩展后，从OPEN表全体中，选一个估价最小的结点下一个考察

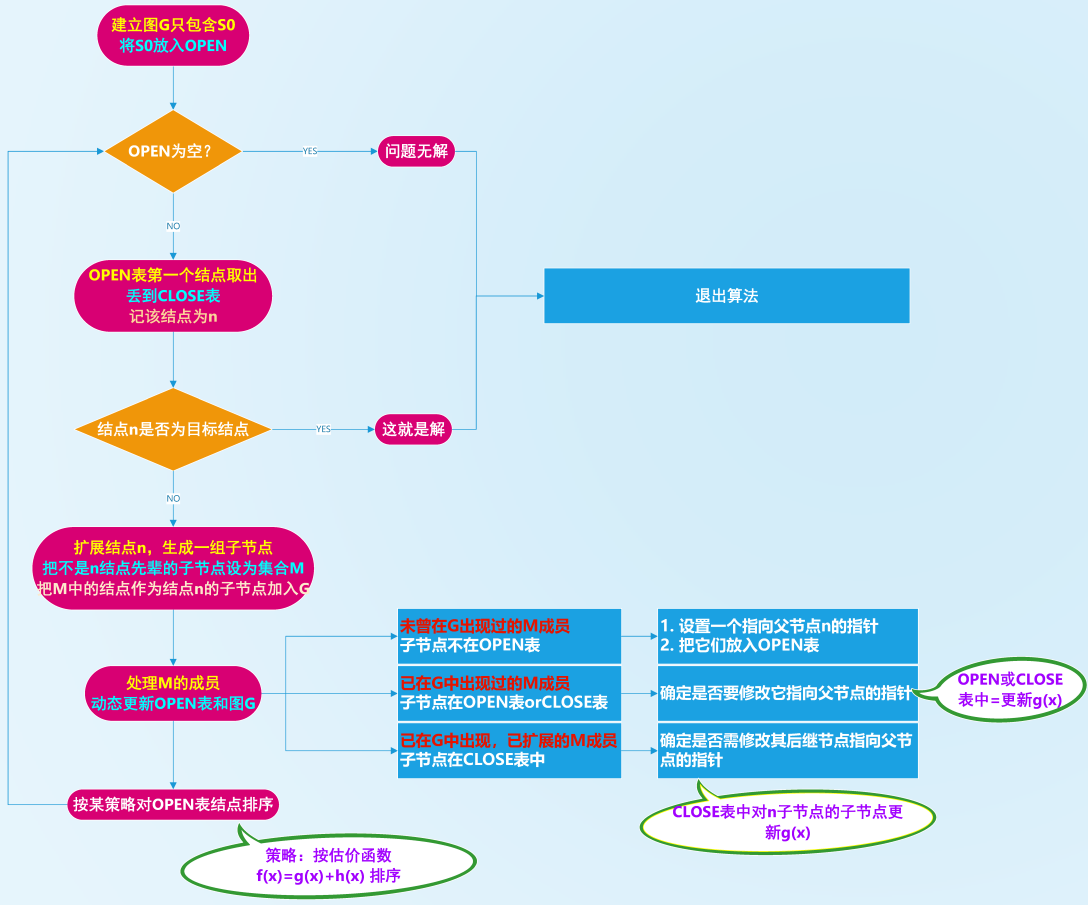


2.7. A*算法

1 算法思想：基于一般搜索过程，加上以下限制(S_0是初始节点)

1. OPEN表中节点按估价函数$f(x)=g(x)+h(x)$小至大排序
2. $g(x)$是$S_0 \to x$路径的代价，$g^*(x)$是$S_0 \to x$最小代价，$g(x)$可估计$g^*(x)$
3. $h^*(x)$是$x \to$ 目标节点最小代价，$h(x)$是$h^*(x)$下界

2 算法流程：气泡中为A^*算法相较于一般搜索过程的具象



3 $A*$算法的性质：

1. 可纳性：$A*$算法可纳
   - 可解状态空间图：初始→目标节点有路径存在
   - 可纳性：搜索算法可在有限步终止，并找到最优秀解
2. 最优性：$h(x)$越大启发信息就越多，搜索效率越高
3. 单调性：要求启发函数$h(x)$单调搜索代价会更低

   单调的条件：$h(S_g)=0$，$h(x_i)\le h(x_j)+c(x_i,x_j)$

3. 结果与分析

3.1. 几个用例之下的测试结果

```
dann_hiroaki@DESKTOP-QANEDCT:~$ g++ a.cpp
dann_hiroaki@DESKTOP-QANEDCT:~$ ./a.out
input:
2 8 3
1 0 4
7 6 5
BFS: time = 2ms, 4 step(s):
------
2 0 3
1 8 4
7 6 5
------
0 2 3
1 8 4
7 6 5
```

```
------
1 2 3
0 8 4
7 6 5
------
1 2 3
8 0 4
7 6 5

depths for DFS: 10
DFS: time = 2ms, 8 step(s):
------
2 8 3
0 1 4
7 6 5
------
0 8 3
2 1 4
7 6 5
------
8 0 3
2 1 4
7 6 5
------
8 1 3
2 0 4
7 6 5
------
8 1 3
0 2 4
7 6 5
------
0 1 3
8 2 4
7 6 5
------
1 0 3
8 2 4
7 6 5
------
1 2 3
8 0 4
7 6 5

A*: time = 1ms, 4 step(s):
------
2 0 3
1 8 4
7 6 5
------
0 2 3
1 8 4
7 6 5
------
1 2 3
0 8 4
```

```
7 6 5
------
1 2 3
8 0 4
7 6 5

Press 'q' to exit:
q
dann_hiroaki@DESKTOP-QANEDCT:
```

```
dann_hiroaki@DESKTOP-QANEDCT:~$ ./a.out
input:
3 1 5
4 0 8
2 6 7
No solution!
Press 'q' to exit:
q
dann_hiroaki@DESKTOP-QANEDCT:
```

```
input:
2 1 6
4 0 8
7 5 3
BFS: time = 240ms, 18 step(s):
------
2 0 6
4 1 8
7 5 3
------
0 2 6
4 1 8
7 5 3
------
4 2 6
0 1 8
7 5 3
------
4 2 6
1 0 8
7 5 3
------
4 2 6
1 8 0
7 5 3
------
4 2 0
1 8 6
7 5 3
------
4 0 2
1 8 6
7 5 3
------
```

```
0 4 2
1 8 6
7 5 3
------
1 4 2
0 8 6
7 5 3
------
1 4 2
8 0 6
7 5 3
------
1 4 2
8 6 0
7 5 3
------
1 4 2
8 6 3
7 5 0
------
1 4 2
8 6 3
7 0 5
------
1 4 2
8 0 3
7 6 5
------
1 0 2
8 4 3
7 6 5
------
1 2 0
8 4 3
7 6 5
------
1 2 3
8 4 0
7 6 5
------
1 2 3
8 0 4
7 6 5

depths for DFS: 100
DFS: time = 3784ms, 100 step(s):
------
2 1 6
0 4 8
7 5 3
------
0 1 6
2 4 8
7 5 3
------
1 0 6
```

```
2 4 8
7 5 3
------
1 6 0
2 4 8
7 5 3
------
1 6 8
2 4 0
7 5 3
------
1 6 8
2 0 4
7 5 3
------
1 6 8
0 2 4
7 5 3
------
0 6 8
1 2 4
7 5 3
------
6 0 8
1 2 4
7 5 3
------
6 8 0
1 2 4
7 5 3
------
6 8 4
1 2 0
7 5 3
------
6 8 4
1 0 2
7 5 3
------
6 8 4
0 1 2
7 5 3
------
0 8 4
6 1 2
7 5 3
------
8 0 4
6 1 2
7 5 3
------
8 4 0
6 1 2
7 5 3
------
8 4 2
```

```
6 1 0
7 5 3
------
8 4 2
6 0 1
7 5 3
------
8 4 2
0 6 1
7 5 3
------
0 4 2
8 6 1
7 5 3
------
4 0 2
8 6 1
7 5 3
------
4 2 0
8 6 1
7 5 3
------
4 2 1
8 6 0
7 5 3
------
4 2 1
8 0 6
7 5 3
------
4 0 1
8 2 6
7 5 3
------
0 4 1
8 2 6
7 5 3
------
8 4 1
0 2 6
7 5 3
------
8 4 1
2 0 6
7 5 3
------
8 0 1
2 4 6
7 5 3
------
8 1 0
2 4 6
7 5 3
------
8 1 6
```

```
2 4 0
7 5 3
------
8 1 6
2 0 4
7 5 3
------
8 1 6
0 2 4
7 5 3
------
8 1 6
7 2 4
0 5 3
------
8 1 6
7 2 4
5 0 3
------
8 1 6
7 0 4
5 2 3
------
8 1 6
0 7 4
5 2 3
------
0 1 6
8 7 4
5 2 3
------
1 0 6
8 7 4
5 2 3
------
1 6 0
8 7 4
5 2 3
------
1 6 4
8 7 0
5 2 3
------
1 6 4
8 0 7
5 2 3
------
1 6 4
8 2 7
5 0 3
------
1 6 4
8 2 7
0 5 3
------
1 6 4
```

```
0 2 7
8 5 3
------
1 6 4
2 0 7
8 5 3
------
1 0 4
2 6 7
8 5 3
------
0 1 4
2 6 7
8 5 3
------
2 1 4
0 6 7
8 5 3
------
2 1 4
6 0 7
8 5 3
------
2 0 4
6 1 7
8 5 3
------
0 2 4
6 1 7
8 5 3
------
6 2 4
0 1 7
8 5 3
------
6 2 4
1 0 7
8 5 3
------
6 2 4
1 5 7
8 0 3
------
6 2 4
1 5 7
0 8 3
------
6 2 4
0 5 7
1 8 3
------
0 2 4
6 5 7
1 8 3
------
2 0 4
```

```
6 5 7
1 8 3
------
2 4 0
6 5 7
1 8 3
------
2 4 7
6 5 0
1 8 3
------
2 4 7
6 5 3
1 8 0
------
2 4 7
6 5 3
1 0 8
------
2 4 7
6 5 3
0 1 8
------
2 4 7
0 5 3
6 1 8
------
2 4 7
5 0 3
6 1 8
------
2 0 7
5 4 3
6 1 8
------
2 7 0
5 4 3
6 1 8
------
2 7 3
5 4 0
6 1 8
------
2 7 3
5 0 4
6 1 8
------
2 7 3
5 1 4
6 0 8
------
2 7 3
5 1 4
0 6 8
------
2 7 3
```

```
0 1 4
5 6 8
------
0 7 3
2 1 4
5 6 8
------
7 0 3
2 1 4
5 6 8
------
7 1 3
2 0 4
5 6 8
------
7 1 3
0 2 4
5 6 8
------
0 1 3
7 2 4
5 6 8
------
1 0 3
7 2 4
5 6 8
------
1 3 0
7 2 4
5 6 8
------
1 3 4
7 2 0
5 6 8
------
1 3 4
7 0 2
5 6 8
------
1 3 4
7 6 2
5 0 8
------
1 3 4
7 6 2
5 8 0
------
1 3 4
7 6 0
5 8 2
------
1 3 4
7 0 6
5 8 2
------
1 3 4
```

```
7 8 6
5 0 2
------
1 3 4
7 8 6
0 5 2
------
1 3 4
0 8 6
7 5 2
------
1 3 4
8 0 6
7 5 2
------
1 3 4
8 5 6
7 0 2
------
1 3 4
8 5 6
7 2 0
------
1 3 4
8 5 0
7 2 6
------
1 3 4
8 0 5
7 2 6
------
1 3 4
8 2 5
7 0 6
------
1 3 4
8 2 5
7 6 0
------
1 3 4
8 2 0
7 6 5
------
1 3 0
8 2 4
7 6 5
------
1 0 3
8 2 4
7 6 5
------
1 2 3
8 0 4
7 6 5

A*: time = 7ms, 18 step(s):
```

```
------
2 0 6
4 1 8
7 5 3
------
0 2 6
4 1 8
7 5 3
------
4 2 6
0 1 8
7 5 3
------
4 2 6
1 0 8
7 5 3
------
4 2 6
1 8 0
7 5 3
------
4 2 0
1 8 6
7 5 3
------
4 0 2
1 8 6
7 5 3
------
0 4 2
1 8 6
7 5 3
------
1 4 2
0 8 6
7 5 3
------
1 4 2
8 0 6
7 5 3
------
1 4 2
8 6 0
7 5 3
------
1 4 2
8 6 3
7 5 0
------
1 4 2
8 6 3
7 0 5
------
1 4 2
8 0 3
7 6 5
```

```
------
1 0 2
8 4 3
7 6 5
------
1 2 0
8 4 3
7 6 5
------
1 2 3
8 4 0
7 6 5
------
1 2 3
8 0 4
7 6 5

Press 'q' to exit:
q
dann_hiroaki@DESKTOP-QANEDCT:
```

3.2. 算法性能对比与分析

1️⃣BFS：BFS能够找到最短路径（即最少步骤的解），但在空间和时间上可能不是最高效的，尤其是当状态空间非常大时

2️⃣DFS：DFS在这个例子中表现出较长的计算时间，原因是它可能探索许多无效路径。DFS不保证找到最短路径，但通常在内存使用上更有效。这里的步骤数超过了BFS，表明找到的解不是最优解

3️⃣A*：Astar搜索在这个例子中表现最优，找到了最短路径，且计算时间远低于BFS和DFS。Astar通过启发式函数优化搜索过程，从而提高了效率。它通常能在合理的时间内找到最优解

4. 源代码解析，及其Python版本(顺带写了)

4.1. `#include<>`

```cpp
#include <iostream>
#include <algorithm>
#include <ctime>
#include <limits>
#include <vector>
#include <queue>
#include <variant>
#include <stack>
#include <functional> // 用于 std::function
#include <type_traits>
```

4.2. 初始化 `fact[]`

```
/*
** 函数 'f' 是一个递归函数，用于计算给定数字 'n' 的阶乘。
** fact存储0!--(9-1)!
*/
int f(int n) {
if (n == 0 || n == 1) {
  return 1;
}
return n * f(n - 1);
}
int fact[] = { f(0), f(1), f(2), f(3), f(4), f(5), f(6), f(7), f(8) };
```

对应的Python

```
def f(n):
 # 计算阶乘的递归函数
 if n == 0 or n == 1:
     return 1
 return n * f(n - 1)
# 创建一个列表来存储0! 到 8! 的值
fact = [f(i) for i in range(9)]
```

4.3. 康托展开及其逆

```
/*
** 函数 'cantor' 实现康托展开。
** 康托展开是一种将数组映射到一个唯一整数的方法。
** 它遍历数组 'board' 的每个元素，对于每个元素，计算数组中后续元素中小于当前元素的个数
'a'。
** 然后，使用 'a' 乘以 (8 - i) 的阶乘，并累加到 'ans'。
** 这样，每个数组都可以映射到一个唯一的整数 'ans'。
*/
int cantor(int board[]) {
int ans = 0;
int i = 0;
while (i < 9) {
int a = 0;
for (int j = i + 1; j < 9; ++j) {
   if (board[i] > board[j]) a++;
}
ans += a * fact[8 - i];
++i;
}
return ans;
}
/*
** 函数 'rev_cantor' 实现逆康托展开。
** 它根据给定的整数 'num'，重构原始的数组。
** 首先，创建一个包含数字0到8的向量 'vec'。
** 然后，对于每个位置 'i'，通过 'num' 除以 (8 - i) 的阶乘来确定该位置的元素。
** 从 'vec' 中移除已确定的元素，并用求模运算更新 'num'。
```

```
** 通过这种方式，可以从整数 'num' 重构原始数组 'board'。
*/
void rev_cantor(int num, int board[]) {
std::vector<int> vec;
int i = 0;
while (i < 9) {
vec.push_back(i);
++i;
}
i = 0;
while (i < 9) {
int pos = num / fact[8 - i];
board[i] = vec[pos];
vec.erase(vec.begin() + pos);
num %= fact[8 - i];
++i;
}
}
```

对应的Python代码

```
def cantor(board, fact):
 ans = 0
 for i in range(9):
     a = sum(1 for j in range(i + 1, 9) if board[i] > board[j])
     ans += a * fact[8 - i]
 return ans
def rev_cantor(num, fact):
 vec = list(range(9))
 board = [0] * 9
 for i in range(9):
     pos = num // fact[8 - i]
     board[i] = vec.pop(pos)
     num %= fact[8 - i]
 return board
```

4.4. 判断是否有解

```
/*
** 函数 'access' 用于判断两个数组是否具有相同的“可解性”。
** 它通过计算每个数组的逆序数来实现这一点。
** 逆序数是数组中一对元素的数量，其中较大的元素出现在较小的元素之前。
** 这个函数首先计算 'board1' 的逆序数 'n1'，然后计算 'board2' 的逆序数 'n2'。
** 为了计算逆序数，它遍历数组中的每对元素，如果找到逆序对（即前面的元素大于后面的元
素），逆序数增加。
** 最后，这个函数检查两个逆序数是否同为奇数或偶数。
** 如两个逆序数同为奇数或偶数，则这两个数组被认为是具有相同的“可解性”，函数返回 true；
否则返回 false。
*/

bool access(int board1[], int board2[]) {
int n1 = 0, n2 = 0;
int i = 0, j;
```

```
// 处理board1
while (i < 9) {
  j = i + 1;
  while (j < 9) {
      switch (board1[i] != 0 && board1[j] != 0 && board1[i] > board1[j])
{
      case true:
          ++n1;
          break;
      default:
          break;
      }
      ++j;
  }
  ++i;
}
// 重置索引，处理board2
i = 0;
while (i < 9) {
  j = i + 1;
  while (j < 9) {
      switch (board2[i] != 0 && board2[j] != 0 && board2[i] > board2[j])
{
      case true:
          ++n2;
          break;
      default:
          break;
      }
      ++j;
  }
  ++i;
}
// 判断两个逆序数是否同为奇数或偶数
if ((n1 % 2) == (n2 % 2)) {
  return true;
}
else {
  return false;
}
}
```

对应的Python代码

```
def access(board1, board2):
 def count_inversions(board):
     inversions = 0
     for i in range(9):
         for j in range(i + 1, 9):
             if board[i] != 0 and board[j] != 0 and board[i] > board[j]:
                 inversions += 1
     return inversions

 n1 = count_inversions(board1)
 n2 = count_inversions(board2)
```

```
    # 判断两个逆序数是否同为奇数或偶数
    return (n1 % 2) == (n2 % 2)
```

4.5. 搜索树结点

```
/*
** 结构体 'Node' 用于表示搜索树中的一个节点。
** 它包含几个重要的属性：
** 'num' - 代表当前状态的康托展开值，是状态的唯一表示。
** 'G' - 代表从初始节点到当前节点的代价（或深度）。
** 'H' - 代表启发式估计，即从当前节点到目标节点的预估代价。
** 'self' - 通常指向节点自身，用于管理节点的生命周期或进行特定操作。
** 'parent' - 指向父节点，用于在找到解决方案时追溯路径。

** 构造函数中，如果 'G' 大于0，计算 'H' 的值。这里使用的是曼哈顿距离，
** 它是每个数字从当前位置到目标位置的横向和纵向移动步数的总和。

** 重载 '<' 操作符用于比较两个节点。在优先队列中，节点根据 'G' + 'H' 的值排序，
** 这样可以优先处理估计总代价最低的节点。

** 'goal' 是一个静态数组，存储目标状态的康托展开值。
*/
struct Node {
Node(int n = 0, Node* p1 = NULL, Node* p2 = NULL, int g = 0) :
num(n), self(p1), parent(p2), G(g), H(0) {
if (g > 0) {
   //计算启发函数值
   int goal_pos[9][2] = { {1, 1}, {0, 0}, {0, 1}, {0, 2}, {1, 2}, {2, 2},
{2, 1}, {2, 0}, {1, 0} };
   int board[9];
   rev_cantor(num, board);
   for (int i = 0; i < 9; ++i) {
       int x = i / 3, y = i % 3;
       H += std::abs(x - goal_pos[board[i]][0]) + std::abs(y -
goal_pos[board[i]][1]);
   }
}
}
bool operator<(const Node& node) const {
return (this->G + this->H) > (node.G + node.H);
}
int num;         //康托展开值
int G;           //初始结点到当前结点的代价（当前结点深度）
int H;           //当前结点到目标结点的代价（曼哈顿距离）
Node* self;
Node* parent;
static int goal[9];
};
```

对应的Python代码

```
class Node:
 goal = [1, 2, 3, 8, 0, 4, 7, 6, 5]  # 目标状态的数组，需要根据实际情况进行设置
```

```
 def __init__(self, num=0, self_node=None, parent=None, g=0):
     self.num = num
     self.self_node = self_node  # Python中不需要，但为保持结构一致性保留
     self.parent = parent
     self.G = g
     self.H = 0

     if g > 0:
         # 计算启发式估计值（曼哈顿距离）
         goal_pos = [(1, 1), (0, 0), (0, 1), (0, 2), (1, 2), (2, 2), (2,
1), (2, 0), (1, 0)]
         board = rev_cantor(num)  # 假设 rev_cantor 函数已经被定义
         self.H = sum(abs(i // 3 - goal_pos[board[i]][0]) + abs(i % 3 -
goal_pos[board[i]][1]) for i in range(9))

 def __lt__(self, other):
     # 定义节点间的比较方法，用于优先队列
     return (self.G + self.H) > (other.G + other.H)
```

4.6. 输入输出

```
int Node::goal[9] = { 1, 2, 3, 8, 0, 4, 7, 6, 5 };
/*
** 函数 'input' 用于从用户那里输入一个数字板的状态。
** 数字板是一个包含9个数字（0到8）的数组，其中0通常表示空白格。
** 这个函数首先显示一个提示信息，然后读取用户输入的9个数字。
** 如果输入的数字不在0到8的范围内，或者有重复的数字，则显示错误信息并要求用户重新输入。
*/
void input(int board[]) {
int flag[9];
begin:
std::cout << "input:" << std::endl;
for (int i = 0; i < 9; ++i)
flag[i] = 0;
for (int i = 0; i < 9; ++i) {
std::cin >> board[i];
if (board[i] >= 0 && board[i] < 9) {
   ++flag[board[i]];
   if (flag[board[i]] > 1) {
       std::cout << "Error input!" << std::endl;
       std::cin.ignore(std::numeric_limits<std::streamsize>::max(),
'\n');
       goto begin;
   }
}
else {
   std::cout << "Error input!" << std::endl;
   std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n');
   goto begin;
}
}
}
```

```
/*
** 函数 'output' 用于输出从初始状态到目标状态的步骤序列。
** 它接收一个 'Node' 类型的对象作为参数，该对象表示搜索树中的一个节点。
** 函数首先将从目标状态回溯到初始状态的节点序列存入一个向量 'ans'。
** 然后，它逆序遍历这个向量，并使用 'rev_cantor' 函数将每个节点的康托展开值转换回数字板状态，
** 最后输出每一步的状态。
*/
void output(Node node) {
std::vector<Node> ans;
while (node.parent) {
ans.push_back(node);
node = *(node.parent);
}
std::cout << ans.size() << " step(s):" << std::endl;
for (int i = ans.size() - 1; i >= 0; --i) {
std::cout << "------" << std::endl;
int board[9];
rev_cantor(ans[i].num, board);
for (int j = 0; j < 3; ++j) {
   for (int k = 0; k < 3; ++k)
       std::cout << board[3 * j + k] << ' ';
   std::cout << std::endl;
}
}
}
```

对应的Python

```
def input_board():
 while True:
     print("input:")
     board = list(map(int, input().split()))
     if len(board) != 9 or len(set(board)) != 9 or any(b < 0 or b >= 9
for b in board):
         print("Error input!")
         continue
     return board

def output(node):
 ans = []
 while node.parent:
     ans.append(node)
     node = node.parent
 print(f"{len(ans)} step(s):")
 for node in reversed(ans):
     board = rev_cantor(node.num)  # 假设 rev_cantor 函数已经被定义
     for i in range(3):
         print(" ".join(str(board[3 * i + j]) for j in range(3)))
     print("------")
```

4.7. 三种算法的集成移动函数(待完善)

```
/*
** 函数 'moveAndCheckUnified' 是一个模板函数，用于执行移动操作并检查新状态。
** 它接受以下参数：
** 'board' - 当前状态的数组表示。
** 'pos' - 空白（0）在数组中的位置。
** 'move' - 移动的方向和步数（例如，在一维数组中表示为左移或右移的步数）。
** 'dataStructure' - 存储节点的数据结构，如队列、栈或优先队列。
** 'parent' - 指向当前节点父节点的指针。
** 'depth' - 指向深度的指针，仅在深度优先搜索中使用。
** 'G' - 当前节点的代价（深度）。

** 函数首先交换 'pos' 和 'pos + move' 的位置，模拟移动操作。
** 然后，使用 'cantor' 函数计算新状态的康托展开值 'num'。
** 如果新状态未被访问（即 'visited[num]' 为 false），则创建一个新的 'Node'，
** 并将其添加到 'dataStructure' 中。同时，标记这个状态为已访问。
** 如果 'depth' 不为空（即在DFS中使用），则将深度减一。
** 最后，再次交换 'pos' 和 'pos + move' 的位置，恢复原始状态。
*/
template <typename Container>
void moveAndCheckUnified(int board[], int pos, int move, Container&
dataStructure, Node* parent, int* depth = nullptr, int G = 0) {
std::swap(board[pos], board[pos + move]);
int num = cantor(board);
if (!visited[num]) {
  Node* node_ptr = new Node(num, NULL, parent, (depth == nullptr) ? G + 1
: *depth);
  node_ptr->self = node_ptr;
  dataStructure.push(*node_ptr);
  visited[num] = true;

  if (depth != nullptr) { // 专门为DFS设计
      --(*depth);
  }
}
std::swap(board[pos], board[pos + move]);
}
```

对应的Python

```python
def move_and_check_unified(board, pos, move, data_structure, parent, visited, depth=None, G=0):
 board[pos], board[pos + move] = board[pos + move], board[pos]  # 交换位置，模拟移动
 num = cantor(board)  # 计算康托展开值
 if num not in visited:
     node = Node(num, None, parent, G + 1 if depth is None else depth)  # 创建新节点
     data_structure.put(node)  # 添加到数据结构中
     visited.add(num)  # 标记为已访问

     if depth is not None:  # 如果提供了深度参数
         depth -= 1  # 减少深度

 board[pos], board[pos + move] = board[pos + move], board[pos]  # 恢复原始状态
```

4.8. 三种算法的分立集成函数

```cpp
/*
** 函数 'moveAndCheck' 用于BFS算法中的移动和检查操作。
** 它接收当前状态的数组 'board'，空白位置 'pos'，移动方向和距离 'move'，
** 节点队列 'Q' 和指向当前节点的父节点的指针 'parent'。
** 函数通过交换 'pos' 和 'pos + move' 来模拟移动操作，
** 然后检查新状态是否已访问过。如果没有，它会创建一个新的节点并加入队列 'Q'。

** 函数 'moveAndCheckDFS' 为DFS算法实现类似的功能。
** 不同之处在于，它使用了栈 'S' 并管理了一个额外的 'depth' 参数，
** 用于控制搜索的深度。如果函数执行了一个有效的移动，它返回 true；否则返回 false。

** 函数 'moveAndCheckAStar' 为A*搜索算法实现类似的功能。
** 它使用优先队列 'Q' 并接受当前节点的代价 'G'。
** 新创建的节点的代价是 'G + 1'，代表从初始节点到当前节点的步数。
*/

bool visited[362880] = { false };
//BFS移动函数
void moveAndCheck(int board[], int pos, int move, std::queue<Node>& Q, Node* parent) {
std::swap(board[pos], board[pos + move]);
int num = cantor(board);
if (!visited[num]) {
  Node* node_ptr = new Node(num, NULL, parent);
  node_ptr->self = node_ptr;
  Q.push(*node_ptr);
  visited[num] = true;
}
std::swap(board[pos], board[pos + move]);
}
//DFS移动函数
bool moveAndCheckDFS(int board[], int pos, int move, std::stack<Node>& S, Node* parent, int& depth) {
std::swap(board[pos], board[pos + move]);
```

```
int num = cantor(board);
if (!visited[num]) {
  Node* node_ptr = new Node(num, NULL, parent);
  node_ptr->self = node_ptr;
  S.push(*node_ptr);
  visited[num] = true;
  --depth;
  std::swap(board[pos], board[pos + move]);
  return true; // 表示移动并添加了新节点
}
std::swap(board[pos], board[pos + move]);
return false; // 表示没有移动或添加新节点
}
//A*移动函数
void moveAndCheckAStar(int board[], int pos, int move,
std::priority_queue<Node>& Q, Node* parent, int G) {
std::swap(board[pos], board[pos + move]);
int num = cantor(board);
if (!visited[num]) {
  Node* node_ptr = new Node(num, NULL, parent, G + 1);
  node_ptr->self = node_ptr;
  Q.push(*node_ptr);
  visited[num] = true;
}
std::swap(board[pos], board[pos + move]);
}
```

对应的Python

```
def move_and_check(board, pos, move, queue, parent, visited):
 board[pos], board[pos + move] = board[pos + move], board[pos]
 num = cantor(board)
 if num not in visited:
     node = Node(num, None, parent)
     queue.put(node)
     visited.add(num)
 board[pos], board[pos + move] = board[pos + move], board[pos]

def move_and_check_dfs(board, pos, move, stack, parent, visited, depth):
 board[pos], board[pos + move] = board[pos + move], board[pos]
 num = cantor(board)
 if num not in visited:
     node = Node(num, None, parent)
     stack.append(node)
     visited.add(num)
     depth -= 1
     board[pos], board[pos + move] = board[pos + move], board[pos]
     return True
 board[pos], board[pos + move] = board[pos + move], board[pos]
 return False

def move_and_check_astar(board, pos, move, priority_queue, parent,
visited, G):
 board[pos], board[pos + move] = board[pos + move], board[pos]
 num = cantor(board)
```

```
 if num not in visited:
     node = Node(num, None, parent, G + 1)
     priority_queue.put(node)
     visited.add(num)
 board[pos], board[pos + move] = board[pos + move], board[pos]
```

4.9. 终止查找函数重载

```
/*
** 每个 'finalizeSearch' 函数都是为特定的搜索算法和相应的数据结构设计的：
** 队列用于BFS，栈用于DFS，优先队列用于A*搜索。

** 这些函数执行以下任务：
** 1. 计算自搜索开始以来经过的时间。
** 2. 使用 'output' 函数输出搜索结果。
** 3. 清理动态分配的节点以防止内存泄漏。
**    这是通过删除数据结构中的每个节点来完成的。

** 参数：
** 'node' - 搜索算法找到的最终节点。
** 'searchMethod' - 正在使用的搜索方法的名称。
** 'start' - 搜索的开始时间。
** 'dataStructure' - 搜索算法使用的数据结构。

** 函数首先计算经过的时间并打印出来，同时显示搜索方法的名称。
** 然后它调用 'output' 来展示从起始到最终节点的路径。
** 最后，它清理数据结构，删除其中的所有节点。
*/

void finalizeSearch(Node& node, const std::string& searchMethod, clock_t
start, std::queue<Node>& dataStructure) {
clock_t end = clock();
std::cout << searchMethod << ": " << "time = " << end - start << "ms, ";
output(node);

while (!dataStructure.empty()) {
  delete dataStructure.front().self;
  dataStructure.pop();
}
}
void finalizeSearch(Node& node, const std::string& searchMethod, clock_t
start, std::stack<Node>& dataStructure) {
clock_t end = clock();
std::cout << searchMethod << ": " << "time = " << end - start << "ms, ";
output(node);

while (!dataStructure.empty()) {
  delete dataStructure.top().self;
  dataStructure.pop();
}
}
void finalizeSearch(Node& node, const std::string& searchMethod, clock_t
start, std::priority_queue<Node>& dataStructure) {
clock_t end = clock();
std::cout << searchMethod << ": " << "time = " << end - start << "ms, ";
```

```
output(node);

while (!dataStructure.empty()) {
  delete dataStructure.top().self;
  dataStructure.pop();
}
}
```

对应的Python

```
import time

def finalize_search(node, search_method, start_time, data_structure):
 end_time = time.time()
 elapsed_time = (end_time - start_time) * 1000  # 转换为毫秒
 print(f"{search_method}: time = {elapsed_time:.2f}ms")
 output(node)
 # 在Python中无需手动删除节点
```

4.10. 初始化函数(待优化)

```
/*
** 函数 'initializeSearch' 用于初始化搜索算法。
** 它接受以下参数:
** 'Board' - 表示初始状态的数组。
** 'ans' - 用于存储目标状态的康托展开值。
** 'start' - 用于记录搜索开始的时间。

** 函数首先复制 'Board' 数组到 'board'，然后记录当前时间到 'start'。
** 接下来，它计算目标状态的康托展开值，并存储在 'ans' 中。
** 然后，它清理 'visited' 数组，确保所有状态都标记为未访问。

** 函数创建一个新的 'Node'，表示初始状态，并将其添加到数据结构中。
** 这个数据结构可以是队列、栈或优先队列，取决于具体的搜索算法。

** 最后，函数返回一个包含数据结构和初始节点指针的pair。
*/
template<typename T>
std::pair<T, Node*> initializeSearch(int Board[], int& ans, clock_t&
start) {
int board[9];
for (int i = 0; i < 9; ++i)
  board[i] = Board[i];

start = clock();

ans = cantor(Node::goal);
for (int i = 0; i < 362880; ++i)
  visited[i] = false;

T dataStructure;
Node* node_ptr = new Node(cantor(board), NULL, NULL);
```

```
node_ptr->self = node_ptr;
dataStructure.push(*node_ptr);
visited[dataStructure.front().num] = true;

return { dataStructure, node_ptr };
}
```

对应的Python

```
import time

def initialize_search(board, goal_state):
 start_time = time.time()

 goal_num = cantor(goal_state)  # 计算目标状态的康托展开值
 visited = set()  # 使用集合来跟踪已访问的状态

 initial_node = Node(cantor(board), None, None)  # 创建初始节点
 data_structure = [initial_node]  # 可以是队列、栈或其他类型的数据结构
 visited.add(initial_node.num)

 return data_structure, initial_node, goal_num, start_time
```

4.11. 三种搜索算法

4.11.1. 广度优先搜索

```
/*
** 函数 'BFS' 实现了广度优先搜索算法。
** 它接受一个表示初始状态的数组 'Board'。

** 首先，它复制 'Board' 数组到 'board'，并记录搜索开始的时间。
** 然后，计算目标状态的康托展开值 'ans'，并初始化访问标记数组 'visited'。

** 使用队列 'Q' 来存储待探索的节点。初始节点（表示初始状态）被添加到队列中。
** 然后，搜索算法开始执行，直到队列为空或找到目标状态。

** 在每一步，算法从队列中取出一个节点，并检查它是否是目标状态。
** 如果是，调用 'finalizeSearch' 函数输出结果并结束搜索。
** 否则，根据可行的移动方向生成新状态，并将对应的节点添加到队列中。

** 移动操作是通过 'moveAndCheck' 函数完成的，它检查新状态是否已被访问过，
** 如果没有，则生成新的节点并加入队列。
*/
void BFS(int Board[]) {
int board[9];
for (int i = 0; i < 9; ++i)
  board[i] = Board[i];

clock_t start, end;
start = clock();

int ans = cantor(Node::goal);
for (int i = 0; i < 362880; ++i)
```

```
  visited[i] = false;

std::vector<Node> tree;
std::queue<Node> Q;
Node* node_ptr = new Node(cantor(board), NULL, NULL);
node_ptr->self = node_ptr;
Q.push(*node_ptr);
visited[Q.front().num] = true;

while (!Q.empty()) {
  Node node = Q.front();
  Q.pop();
  tree.push_back(node);

  if (node.num == ans) { finalizeSearch(node, "BFS", start, Q); break;
}

  rev_cantor(node.num, board);
  int pos = 0;
  for (/**/; board[pos] != 0; ++pos);
  int row = pos / 3;
  int col = pos % 3;

  if (col > 0) moveAndCheck(board, pos, -1, Q, node.self); // 左移
  if (row > 0) moveAndCheck(board, pos, -3, Q, node.self); // 上移
  if (col < 2) moveAndCheck(board, pos, +1, Q, node.self); // 右移
  if (row < 2) moveAndCheck(board, pos, +3, Q, node.self); // 下移
}
}
```

对应的Python

```
import time

def bfs(board, goal_state):
 start_time = time.time()

 goal_num = cantor(goal_state)
 visited = set()

 queue = [Node(cantor(board), None, None)]
 visited.add(queue[0].num)

 while queue:
     node = queue.pop(0)
     if node.num == goal_num:
         finalize_search(node, "BFS", start_time, queue)  # 假设
'finalize_search' 已定义
         break

     board_state = rev_cantor(node.num)  # 假设 'rev_cantor' 已定义
     pos = board_state.index(0)
     row, col = pos // 3, pos % 3

     # 尝试不同的移动并检查
```

```
    for new_pos in [pos - 1, pos - 3, pos + 1, pos + 3]:  # 左、上、右、下移
        if 0 <= new_pos < 9 and not (col == 0 and new_pos == pos - 1) and not (col == 2 and new_pos == pos + 1):
            new_board = board_state.copy()
            new_board[pos], new_board[new_pos] = new_board[new_pos], new_board[pos]
            num = cantor(new_board)
            if num not in visited:
                queue.append(Node(num, None, node))
                visited.add(num)
```

4.11.2. 有界深度优先搜索

```
/*
** 函数 'DFS' 实现深度优先搜索算法。
** 它接受表示初始状态的数组 'Board' 和深度限制 'd'。

** 首先，它复制 'Board' 数组到 'board'，记录搜索开始的时间，并设置深度限制。
** 接着，它计算目标状态的康托展开值 'ans' 并初始化访问数组 'visited'。

** 使用栈 'S' 来存储待探索的节点。初始节点被推入栈中。
** 然后，算法开始执行，直到栈为空或找到目标状态。

** 在每一步，算法检查当前深度，并从栈中弹出一个节点。
** 如果节点的状态等于目标状态，调用 'finalizeSearch' 函数输出结果并结束搜索。
** 否则，根据可行的移动方向生成新状态，并将对应的节点推入栈中。

** 移动操作是通过 'moveAndCheckDFS' 函数完成的，它检查新状态是否已被访问过，
** 如果没有，则生成新的节点并加入栈。深度也会相应地减小。
*/
void DFS(int Board[], int d) {
int board[9];
for (int i = 0; i < 9; ++i)
  board[i] = Board[i];

clock_t start, end;
start = clock();

int depth = d;
int ans = cantor(Node::goal);
for (int i = 0; i < 362880; ++i)
  visited[i] = false;

std::vector<Node> tree;
std::stack<Node> S;
Node* node_ptr = new Node(cantor(board), NULL, NULL);
node_ptr->self = node_ptr;
S.push(*node_ptr);
visited[S.top().num] = true;

while (!S.empty()) {
  if (depth < 0) {
      ++depth;
```

```
        tree.push_back(S.top());
        S.pop();
    }

    Node node = S.top();

    if (node.num == ans) { finalizeSearch(node, "DFS", start, S); break;
}

    rev_cantor(node.num, board);
    int pos = 0;
    for (/**/; board[pos] != 0; ++pos);
    int row = pos / 3;
    int col = pos % 3;

    // 使用 moveAndCheckDFS 函数处理四个方向的移动
    if (col > 0 && moveAndCheckDFS(board, pos, -1, S, node.self, depth))
continue; // 左移
    if (row > 0 && moveAndCheckDFS(board, pos, -3, S, node.self, depth))
continue; // 上移
    if (col < 2 && moveAndCheckDFS(board, pos, +1, S, node.self, depth))
continue; // 右移
    if (row < 2 && moveAndCheckDFS(board, pos, +3, S, node.self, depth))
continue; // 下移

    ++depth;
    tree.push_back(S.top());
    S.pop();
}
if (depth == d + 1) {
    for (int i = tree.size() - 1; i >= 0; --i)
        delete tree[i].self;
    std::cout << "No solution at current depth!" << std::endl;
}
}
```

对应的Python

```
import time

def dfs(board, goal_state, depth_limit):
 start_time = time.time()

 goal_num = cantor(goal_state)
 visited = set()

 stack = [Node(cantor(board), None, None)]
 visited.add(stack[-1].num)
 depth = depth_limit

 while stack:
     if depth < 0:
         depth += 1
         stack.pop()
         continue
```

```
    node = stack[-1]

    if node.num == goal_num:
        finalize_search(node, "DFS", start_time, stack)  # 假设
'finalize_search' 已定义
        return

    board_state = rev_cantor(node.num)  # 假设 'rev_cantor' 已定义
    pos = board_state.index(0)
    row, col = pos // 3, pos % 3

    moved = False
    # 尝试不同的移动并检查
    for new_pos in [pos - 1, pos - 3, pos + 1, pos + 3]:  # 左、上、右、
下移
        if 0 <= new_pos < 9 and not (col == 0 and new_pos == pos - 1)
and not (col == 2 and new_pos == pos + 1):
            new_board = board_state.copy()
            new_board[pos], new_board[new_pos] = new_board[new_pos],
new_board[pos]
            num = cantor(new_board)
            if num not in visited:
                stack.append(Node(num, None, node))
                visited.add(num)
                moved = True
                depth -= 1
                break

    if not moved:
        depth += 1
        stack.pop()

print("No solution at current depth!")
```

4.11.3. A*算法

```
/*
** 函数 'Astar' 实现了 A* 搜索算法。
** 它接受一个表示初始状态的数组 'Board'。

** 首先，它复制 'Board' 数组到 'board'，记录搜索开始的时间。
** 接着，它计算目标状态的康托展开值 'ans' 并初始化访问数组 'visited'。

** 使用优先队列 'Q' 来存储待探索的节点，按照启发式估计值排序。
** 初始节点（表示初始状态并带有启发式估计值）被添加到优先队列中。
** 然后，算法开始执行，直到队列为空或找到目标状态。

** 在每一步，算法从队列中取出估计代价最低的节点，
** 并检查它是否是目标状态。如果是，调用 'finalizeSearch' 函数输出结果并结束搜索。
** 否则，根据可行的移动方向生成新状态，并将对应的节点添加到优先队列中。

** 移动操作是通过 'moveAndCheckAStar' 函数完成的，它检查新状态是否已被访问过，
** 如果没有，则生成新的节点并加入优先队列。节点的估计代价考虑了到目标状态的启发式估
计。
```

```
*/
void Astar(int Board[]) {
int board[9];
for (int i = 0; i < 9; ++i)
  board[i] = Board[i];

clock_t start, end;
start = clock();

int ans = cantor(Node::goal);
for (int i = 0; i < 362880; ++i)
  visited[i] = false;

std::vector<Node> tree;
std::priority_queue<Node> Q;
Node* node_ptr = new Node(cantor(board), NULL, NULL, 0);
node_ptr->self = node_ptr;
Q.push(*node_ptr);
visited[Q.top().num] = true;

while (!Q.empty()) {
  Node node = Q.top();
  Q.pop();
  tree.push_back(node);

  if (node.num == ans) { finalizeSearch(node, "A*", start, Q); break; }

  rev_cantor(node.num, board);
  int pos = 0;
  for (/**/; board[pos] != 0; ++pos);
  int row = pos / 3;
  int col = pos % 3;

  // 使用 moveAndCheckAStar 函数处理四个方向的移动
  if (col > 0) moveAndCheckAStar(board, pos, -1, Q, node.self, node.G);
// 左移
  if (row > 0) moveAndCheckAStar(board, pos, -3, Q, node.self, node.G);
// 上移
  if (col < 2) moveAndCheckAStar(board, pos, +1, Q, node.self, node.G);
// 右移
  if (row < 2) moveAndCheckAStar(board, pos, +3, Q, node.self, node.G);
// 下移
}
}
```

对应的Python

```
import time
import queue

def astar(board, goal_state):
 start_time = time.time()

 goal_num = cantor(goal_state)
 visited = set()
```

```
    priority_queue = queue.PriorityQueue()
    initial_node = Node(cantor(board), None, None, 0)
    priority_queue.put(initial_node)
    visited.add(initial_node.num)

    while not priority_queue.empty():
        node = priority_queue.get()

        if node.num == goal_num:
            finalize_search(node, "A*", start_time, priority_queue)  # 假设 'finalize_search' 已定义
            break

        board_state = rev_cantor(node.num)  # 假设 'rev_cantor' 已定义
        pos = board_state.index(0)
        row, col = pos // 3, pos % 3

        # 尝试不同的移动并检查
        for new_pos, cost in [(pos - 1, -1), (pos - 3, -3), (pos + 1, 1), (pos + 3, 3)]:
            if 0 <= new_pos < 9 and not (col == 0 and new_pos == pos - 1) and not (col == 2 and new_pos == pos + 1):
                new_board = board_state.copy()
                new_board[pos], new_board[new_pos] = new_board[new_pos], new_board[pos]
                num = cantor(new_board)
                if num not in visited:
                    new_node = Node(num, None, node, node.G + cost)
                    priority_queue.put(new_node)
                    visited.add(num)
```

5.12. 主函数

```
int main(void) {
    int board[9];
    char ch;
    do {
        input(board);
        if (access(board, Node::goal)) {
            BFS(board);
            std::cout << std::endl;
            int depth = 10;
            std::cout << "depths for DFS: ";
            std::cin >> depth;
            DFS(board, depth);
            std::cout << std::endl;
            Astar(board);
            std::cout << std::endl;
        }
        else
            std::cout << "No solution!" << std::endl;
        std::cout << "Press 'q' to exit:" << std::endl;
        std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n');
```

```
 } while (std::cin >> ch && ch != 'q');
}
```

对应的Python代码

```python
def main():
    while True:
        board = input_board()  # 假设 input_board 函数已定义
        if access(board, Node.goal):  # 假设 access 函数已定义
            bfs(board, Node.goal)  # 假设 bfs 函数已定义
            print()

            depth = int(input("depths for DFS: "))
            dfs(board, Node.goal, depth)  # 假设 dfs 函数已定义
            print()

            astar(board, Node.goal)  # 假设 astar 函数已定义
            print()
        else:
            print("No solution!")

        ch = input("Press 'q' to exit, any other key to continue: ")
        if ch.lower() == 'q':
            break

if __name__ == "__main__":
    main()
```