# AStar4SudokuReshuffle

王子义　2022010912020

# Announcement

**Code** will be released at **tailorbrue/AStar4SudokuReshuffle (github.com)** with **MIT License**.

# Introduction

九宫重排是指给定一个具有 **8** 个图块的 **3×3** 板(每个图块都有一个 **1 到 8** 的数字)和一个**空白图块**(用 **0** 代表), 以及一个**初始状态**和一个**目标状态**, 期望通过**只交换空白图块与相邻的数字图块**, 将**初始状态变成目标状态**的任务.

初始

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

答案

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

此次试验尝试给出一个具有**完备性, 高效率**和**最优性**的算法, 即解决判断某给定的九宫重排问题**是否有解**的问题, 以及在有解的前提下, 如何**高效**得到**最短**移动路径(**Moving Path**)的问题.

# Method

## 置换, 对换与逆序数

### Def. 置换

一个有限集合 S 到自身的双射称为 S 的一个**置换**, 可表示为:

$$f = \begin{pmatrix} a_1, a_2, \ldots, a_n \\ a_{p_1}, a_{p_2}, \ldots, a_{p_n} \end{pmatrix}$$

### Def. 对换

如果把 1,2,3,...,n 的排列中，任意两个数 i 和 j 交换，其余数保持不动, 就得到一个新排列. 对于排列施加这样一个变换叫做一个**对换**, 用 (i, j) 表示。

- **定理**: 任意置换都可分解为对换的乘积. 注意九宫重排问题中仅保留了部分可能的对换, 从而可能**无法组合得到全部的可能的置换**, 这就意味着此任务有可能存在**无解的情况**.
- **Def.** 有偶数个逆序的排列叫做一个**偶排列,** 有奇数个逆序的排列叫做一个**奇排列.**
  - **定理:** 每一个对换都**改变**排列的奇偶性。

### Def. 逆序数

在一个排列中, 如果某一个较大的数排在某一个较小的数前面, 就说这两个数构成一个逆序. 在一个排列里出现的逆序的总个数, 叫做这个排列的**逆序数**.

- **验证:** 考虑集合 S = {1, 2,..., 8}, 九宫重排中出现的对换都**不会改变**其排列的奇偶.

## A* 算法

A* 算法是一种求解最短路径的启发式搜索方法. 伪代码如下:

```
OPEN = priority queue containing START
CLOSED = empty set

while lowest rank in OPEN is not the GOAL:
  current = remove lowest rank item from OPEN
  add current to CLOSED
  for neighbors of current:
    cost = g(current) + movementcost(current, neighbor)
    if neighbor in OPEN and cost less than g(neighbor):
      remove neighbor from OPEN, because new path is better
    if neighbor in CLOSED and cost less than g(neighbor): (2)
      remove neighbor from CLOSED
    if neighbor not in OPEN and neighbor not in CLOSED:
```

```
        set g(neighbor) to cost
        add neighbor to OPEN
        set priority queue rank to g(neighbor) + h(neighbor)
        set neighbor's parent to current

reconstruct reverse path from goal to start
by following parent pointers
```

### 启发函数

启发函数由两部分组成, 定义如下:

$$f = g + h$$

其中 g(n)表示从**初始结点到任意结点n**的代价, h(n)表示从**结点n到目标点**的启发式评估代价, f(n) 权衡这两者.

- 如果 h(n) 是0，则只有g(n)起作用，此时A演变成 Dijkstra 算法，这**保证能找到最短路径**. 另一种极端情况，如果h(n)比g(n)大很多，则只有h(n)起作用，A star演变成BFS算法.
- 如果h(n)经常都比从n移动到目标的实际代价小（或者相等），则A**保证**能找到一条最短路径。
- h(n)越小，A扩展的结点越多，运行**就得越慢**。如果h(n)有时比从n移动到目标的实际代价高，则A star不能保证找到一条最短路径，但它**运行得更快**。

### Open list & Closed list

为了避免搜索路径中**出现环**, 我们将经过过的节点放入 Closed list, 每次搜索都只在 Open list 中寻找. 每次取 Open list 中 F 值**最小**的点, 且**更新**其可达点的 G 值. 当然, 在特别的搜索树中, 不需要 G 值的更新.

# Experiment

## g(n) & h(n)

此次实验中, g(n) 和 h(n) 被设置如下:

```
def setH(self, endNode):
    for x in range(0, 3):
        for y in range(0, 3):
            for m in range(0, 3):
                for n in range(0, 3):
                    if self.array2d[x][y] == endNode.array2d[m]
                        self.h += abs(x - m) + abs(y - n)
def setG(self, steps):
      self.g = steps
```

## 初始状态与目标状态

```
    start = [[4,5,7],[0,2,3],[1,6,8]]
    end = [[1,2,3],[4,0,5],[6,7,8]]
```

## 搜索树与移动路径

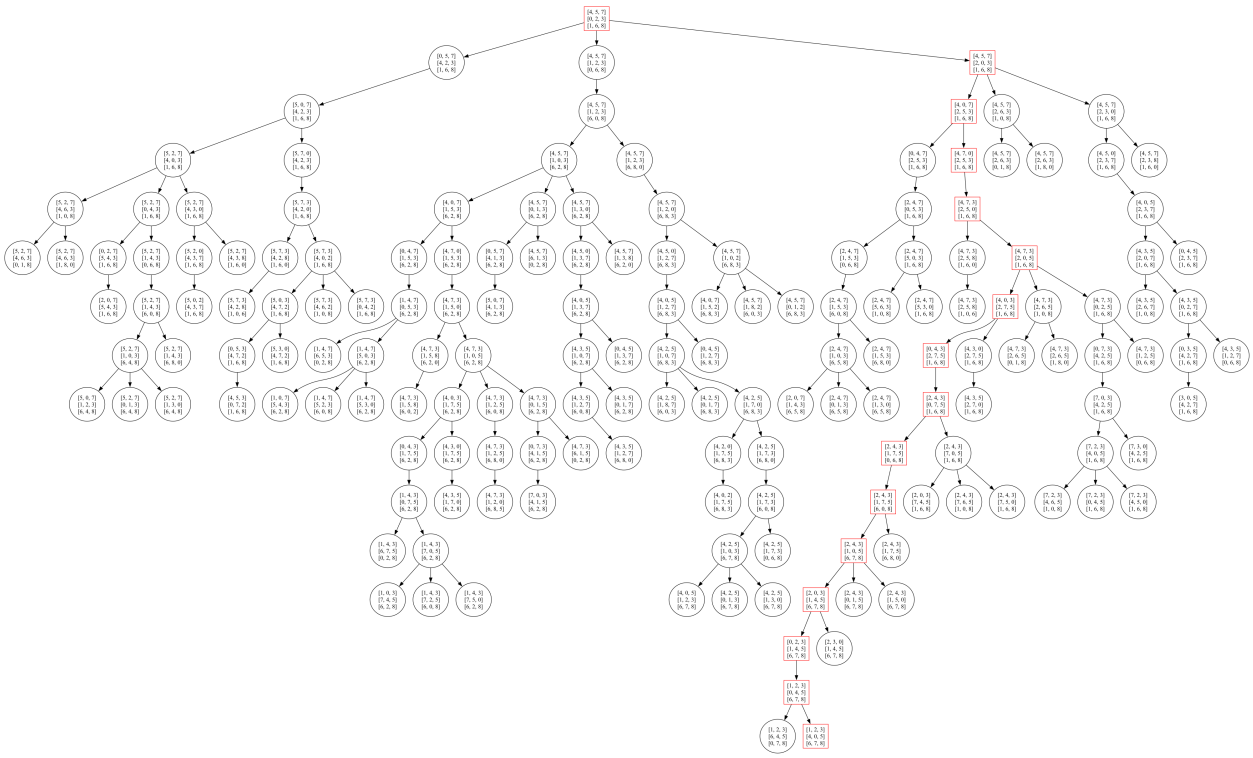

Figure.1. Solving tree. Moving path is show with red boxes.