二阶魔方还原

by Jianyu Zhou

时间限制: 1000 ms 内存限制: 16384 KB

问题描述

二阶魔方是 2x2x2 的立方体结构魔方，其只有 8 块，每个块有 3 个面露在外面，并被涂为不同的颜色。下图为二阶魔方的示意图和平面展开图。

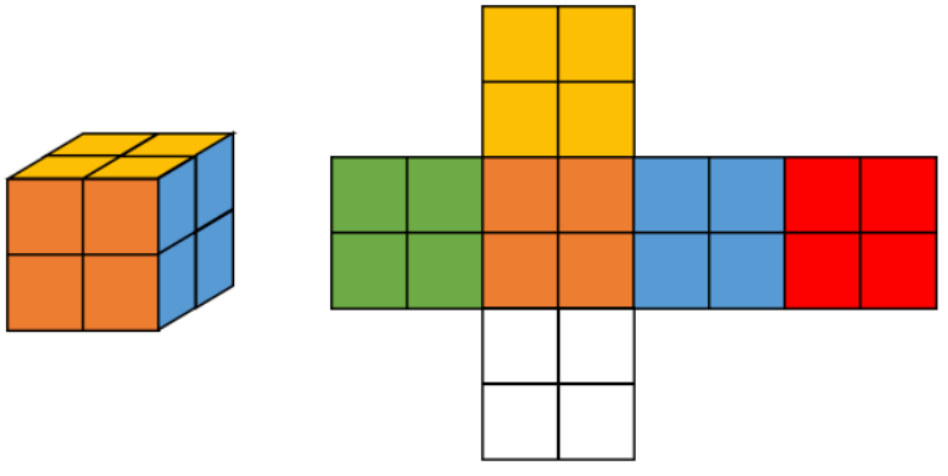

图1：二阶魔方与其平面展开图

我们定义 **魔方的状态是不受视角影响** 的，也就是说只有拧动魔方后，其状态才会改变。这里我们可以分析一下二阶魔方存在多少种不同的状态。由于视角的不同，二阶魔方的每一种状态会对应到 24 种不同的如图 1 所示的平面展开图（6 个面均可作为正面，同时又可以以正面为轴滚动 4 次，得到 6x4=24 种不同的平面展开图）。二阶魔方 8 个块的位置均可任意互换（8! 种情况）。如果固定一个块作为参考，那么另外 7 个块中每个都可以有 3 种不同的朝向（3^7 种情况）。于是我们得到了 $8!3^7$ 种不同的平面展开图。所以二阶魔方的状态总数为：

$$\frac{8!3^7}{24} = 3674160$$

在还原二阶魔方的过程中，“**FRU注释**”是一种比较通用的还原步骤注释。FRU注释只旋转魔方的 **正面（F: front）**、**右面（R: right）**和 **上面（U: up）**，并且可以分别 **顺时针旋转90°（用+表示）**、**180°（用2表示）**和 **逆时针旋转90°（用-表示）**。这样魔方每步就有 9 种的变化方式（注意到在可以调整视角的情况下其他的旋转方式是等价于这 9 种方式中的某一种的）。下图描述了FRU注释的操作过程：

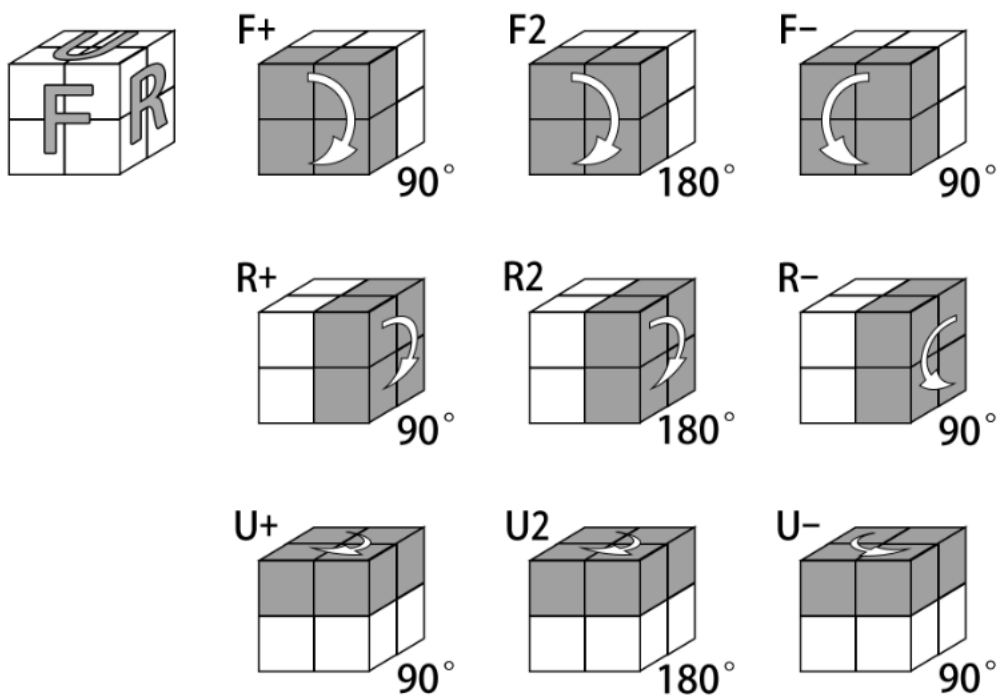

图2：FRU注释的 9 种魔方旋转方式

采用FRU注释的 9 种方式旋转魔方的话，对于二阶魔方的任意一种状态，最坏情况下只需要 11 步就可以将魔方还原。其中，在 3674160 种不同的魔方状态中，有一半左右（1887748 种）的状态最少需要 9 步才能还原魔方。下表给出了最少旋转次数和对应的状态总数的关系：

表1：还原魔方所需最少步数与状态个数的关系

最少转动次数	状态个数	最少转动次数	状态个数
0	1	6	50136
1	9	7	227536
2	54	8	870072
3	321	9	1887748
4	1847	10	623800
5	9992	11	2644

我们的问题就是：给出一个被打乱的二阶魔方，求出还原魔方的 **最少步数**（每步只能是FRU注释的 9 种操作的一种）并且按照 **FRU注释** 给出每步的具体操作。（注意：一个魔方被还原时仅要求每个面的颜色一致，与视角无关。）

输入格式

输入为魔方的某个状态。输入的格式是以图 1 的平面展开图的形式，按顺序给出每一块的颜色编号，其中每一块的颜色编号如下表所示：

表2：颜色编号

颜色	黄色	绿色	橙色	蓝色	红色	白色
编号	0	1	2	3	4	5

本题保证所有测试样例都是图 1 所示的魔方，即 **不同测试数据还原后的魔方配色一致**（但是视角可能不一样）。

输出格式

首先第一行输出一个正整数 n ，表示还原魔方所需的 **最少** 步数。

接下来 n 行每行一个字符串，表示还原魔方的每步操作。字符串要求是 9 种旋转操作（F+，F2，F-，R+，R2，R-，U+，U2，U-）中的一种。

注意：使用最少步数还原魔方可以有多种操作方式，请输出任意一种即可。前 4 组测试数据保证最少步数不超过 7 步，后 6 的测试数据无限制。

输入样例

```
颜色编号的平面展开图（本行不计为输入）
    0 4
    2 2
4 3 0 0 1 3 5 3
1 0 1 2 5 3 5 4
    4 1
    5 2
```

输出样例

```
最少步数和每步的操作（本行不计为输入）
3
F+
R+
U+
```

提示

1. 关于魔方状态数的分析可以参考 维基百科，同时大家也可以使用 二阶魔方模拟器 熟悉二维魔方的操作和求解。
2. 下图给出了关于输入样例和输出样例的解释。

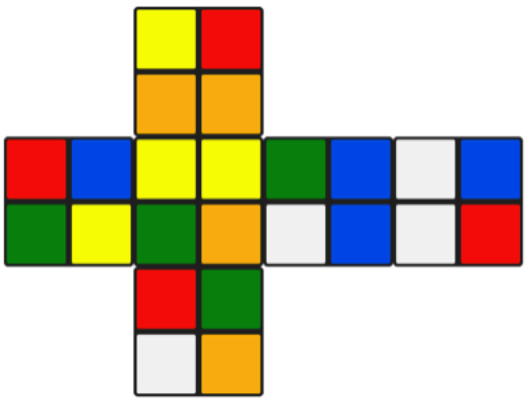

图3：样例对应的魔方平面展开图

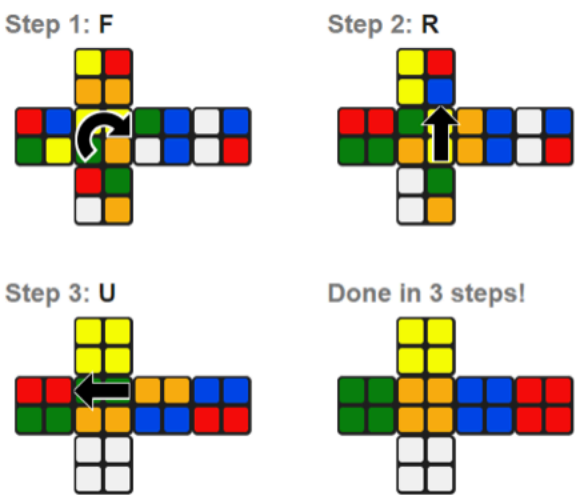

图4：样例的求解方式

3. 通过本题可能需要了解“状态空间搜索”、“状态压缩”和“双向广度优先搜索”。请先参考百度百科的 搜索算法。

3.1 状态空间搜索

状态空间搜索就是将问题的求解过程转化为从初始状态到目标状态寻找路径的过程。状态空间可以看成一个图，图上的点就是某个 **状态** ，边则是某个状态到另一个状态的 **状态转移** 关系。使用状态空间搜索求解问题的关键就是：利用有效的数据结构对状态进行存储，实现状态间的转移关系，进而使用 **深度优先搜索** 或 **广度优先搜索** 算法计算初始状态到目标状态路径。

本题中，状态就是魔方的状态，状态转移关系则是FRU注释中的 9 种操作。这里我们可以使用一个长度为 24 的一维数组来表示魔方的一个状态。比如我们对魔方的每块的每个面按顺序编号

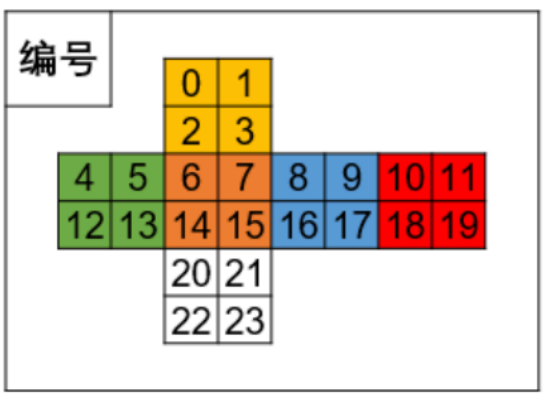

图 5：对魔方的面进行编号

由图 5 所示，2 号位置的面是黄色（颜色编号 0），8 号位置的面是蓝色（颜色编号 3），所以我们可以用以下数组表示该状态（注意颜色的编号是数组的值，取值 0~5，面的位置编号是数组的下标，取值 0~23）：

{0, 0, 0, 0, 1, 1, 2, 2, 3, 3, 4, 4, 1, 1, 2, 2, 3, 3, 4, 4, 5, 5, 5, 5}

接下来看状态转移如何进行。以 F+ 操作为例，对于图 5 的状态，进行 F+ 操作后，每个块的面会发生如下图所示的变化

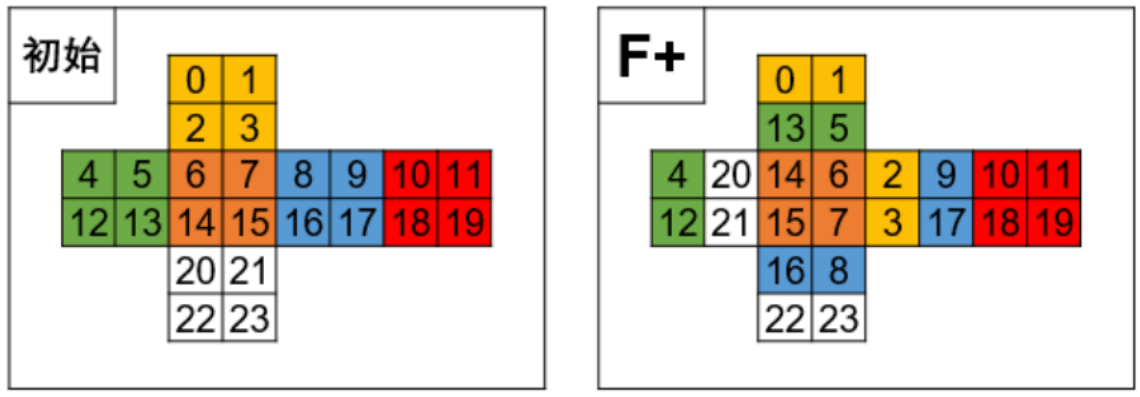

图 6：F+ 操作的状态转移关系

由图 6 所示，有 8 个面发生了移动，比如 2 号位置变成了原来 13 号位置的颜色，3 号位置变成了原来 5 号位置的颜色，所以经过 F+ 操作后，新的状态由以下数组表示（粗体为发生变化的位置）

{0, 0, **1**, **1**, 1, **5**, **2**, **2**, **0**, 3, 4, 4, 1, **5**, **2**, **2**, **0**, 3, 4, 4, **3**, **3**, 5, 5}

所以，我们需要把FRU注释中的 9 中操作对应的颜色变换列举出来（即变换后的颜色对应原来的哪个位置的颜色），然后依据输入给的初始状态，通过广度优先搜索遍历可能转移到的状态，直到搜索到最终状态（每个面的四个小面颜色一致），便求出了还原魔方的最少步数，并且可以通过回溯得到每步的操作。

3.2 状态压缩

在搜索的过程中，我们需要标记哪些状态已经被访问过了。对于简单的搜索，我们可以直接用 bool 数组来标记状态访问与否。但是，本题的状态是一个长度为 24 的数组，我们需要建立哈希表来存储。为了简化程序，这里建议大家直接使用 C++ STL 中的 map 或 unordered_map 来实现哈希表，并且状态的表示也用 STL 中的 vector 而不是数组（因为 vector 可以直接用在 map 中，而数组不行）。不过，用数组或 vector 表示一个状态会占用较多的内存，并且会延长哈希表定位键值的时间。

状态压缩要考虑的问题是如何用 **更有效的方式对状态进行编码**。对本题而言，注意到只有 6 种颜色，所以可以使用 **六进制** 对长度为 24 的状态数组编码：六进制的第 i 位，表示第 i 个位置的颜色。所以我们需要一个取值范围在 0 ~ 6^{24}-1 的数字来表示一个状态，这个取值范围恰好在无符号 8 字节整数（C++中的 unsigned long long）的取值范围（0 ~ 2^{64}-1）内。于是，我们在使用 map 标记状态时，就不需要使用 vector 作为键值，而是使用这个 unsigned long long 的编码值了。比如最初状态的数组使用六进制状态压缩的表示为

$$
\begin{aligned}
& hash\left(\{0,0,0,0,1,1,2,2,3,3,4,4,1,1,2,2,3,3,4,4,5,5,5,5\}\right) \\
= \; & 0\times 6^0+0\times 6^1+\cdots a_i\times 6^i+\cdots+5\times 6^{22}+5\times 6^{23} \\
= \; & 4737629216041086961
\end{aligned}
$$

而我们要获取第 i 个位置的颜色时，便可直接让压缩后的编码对 6^{i+1} 取模后，再对 6^i 取整即可。

当然，六进制状态压缩只是一个例子，存在更好的压缩方式，因为总状态数仅有 3674160 种。

3.3 双向广度优先搜索

广度优先搜索的过程中需要维护一个存储待搜索状态的队列，因此广度优先搜索的问题就是会占用很大的内存。而对于目标状态确定的问题来说，双向广度优先搜索是减少内存开销的一个有效手段。双向广度优先搜索的思想就是 **从 初始状态 和 目标状态 同时进行广度优先搜索**，保持搜索的层数同步，最终两个方向的搜索所遍历的状态会在中间相遇，从而求出了最短的路径。这样一来便减去了很多不必要的搜索状态，因为搜索的过程中每层的状态数是近似成倍增加的。

使用双向广度优先搜索需要确定初始状态和目标状态。对于本题而言，初始状态是直接输入的，而目标状态看上去应该有 24 种可能（视角因素）。实际上，本题的目标状态应该是唯一的，因为 **FRU注释的操作过程中，魔方中有一块从来没有移动过**（后左下）。由于题目保证全部测试点的魔方配色方案一致，因此，魔方还原后的状态便由这一个不动的块唯一确定了。

PS：建议尽可能精简代码，重复的代码封装成函数调用，充分利用数组，以减少出错的可能性。