

在线魔方模拟求解器使用说明

这个网站主要分两部分，魔方模拟器和解法助手。分别点击添加魔方和解法助手即可，左右分两栏显示。

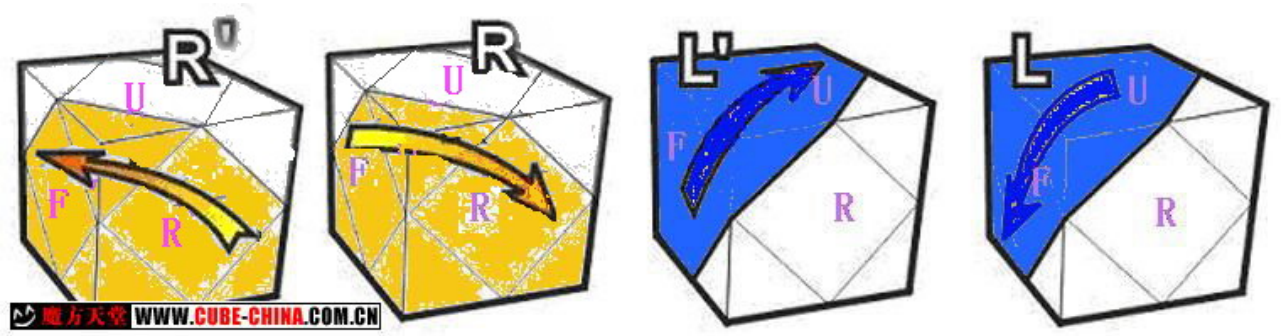
模拟器

支持二至七阶正阶魔方

魔方公式用RUBFLD等字母标记，转两层是小写，包括金字塔魔方。

具体标记意义是魔友熟知的，不再赘述，可参考任意魔方教程。

比如说斜转魔方的标记如下：



那么输入公式：

R' L' R L y2 R' L' R L

可达到效果：

斜转魔方

R' L' R L y2 R' L' R L

确认

模拟器目前不支持拖拽操作。

求解器


二阶魔方直接求解

对于一些简单的魔方，可以直接求出所有解法，
比如说二阶魔方，直接输入打乱：

F R2 U' F2 U2 F' U' F U' R2

F R2 U' F2 U2 F' U' F U' R2

确认



求解

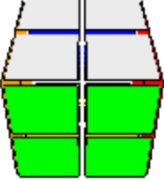
F' R2 F' U R' U2 F R'

F' R2 F' U R' U2 F R'

F' R2 F' U2 U' R' U2 F R'

F' R2 F' U' U2 R' U2 F R'

D2 F U2 B' U' F U' F2 D2 U2



点击求解后会出现一系列解法列表，点击其中的一行，即可进行相应动画演示。

CFOP

对于三阶魔方，不支持一步求解，但支持分步CFOP求解（直到F2L结束，后面算力不足，而且没有打表）
打乱一栏输入，例如：

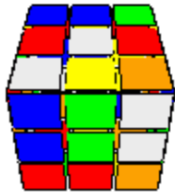
D' R L2 B' U L2 F' L2 D L2 B2 L' D2 R2 U2 D2 R' D2 F2 L'

然后点击求解

然后会显示一个解法列表，这只是第一步十字的各种做法，点击某一条，显示的魔方会进行相应转动

D' R L2 B' U L2 F' L2 D L2 B2 L' D2 R2 U2 D2 R' D2 F2 L'

确认



求解

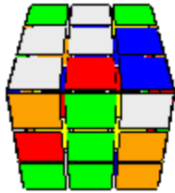
F2 U' B2 L' B R

F U' B2 L' B F R

U' B2 L' B U F2 R

R D' B L' B2 D' F2

R D' B L' B2 D' F2



求解

再次点击求解，即可进行十字下一步，即第一个棱角对，
一直到两层还原：



求解

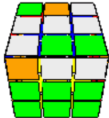
B2 U L U' L' B2 R' U' R

R2 U2 B' R B U2 R U2 R

R2 F' L F' L' F2 R U2 R

R2 U2 R B2 L' B' L B' R

R U L B2 R U L F U R2



求解

五阶求解3*3中心

五阶解法目前也只是求解一个3乘3中心。不过要搜索挺久的。

五阶魔方

U D2 d' u L' F' B b L' u' R' B b' D2 d2 r' u U B L2 B' l' L2 f u' r' u2 b r' f2 b' d r' U' B2 b r' l

确认



求解

R2 r U u' b f U' f2
R2 r f' b l' f2 U' f2
R2 f l2 F' r b u' f'
R2 u' r f B u' f2 l
R2 u' l B u' f u' f2 l



框架

下面讲一下主要的框架，方便大家理解代码和继续开发，因为魔方的种类有很多，在这个基础上继续工作也是不麻烦的。

构造

首先为了兼容各种魔方，我采取了通用性的想法，一个魔方的本质是一些面（或者说一些块，但由于染色是以面为单位，所以不妨把面看做最基本的单位），以及这些面所能进行的转动。

3D框架使用了three.js，利用顶点集合v和面f的构造函数如下：

```
class Tcube {
  constructor(v, f) {
    this.restorev = v;
    this.restoref = f;
    this.initState = [];
    this.pieces = [];
    . . . . .
    for (let face of f) {
      let geo = new THREE.Geometry();
      let mat = [new THREE.MeshBasicMaterial({color: 0}),
        new THREE.MeshBasicMaterial({color: 0, wireframe: true})];
      geo.vertices = [];
      let abcd = [];
      if (face instanceof THREE.Face4) {
        abcd = [face.a, face.b, face.c, face.d];
```

```
        geo.faces = [new THREE.Face4(0, 1, 2, 3)];
    }
    else . . . . .
}
}
```

利用这个框架，如果我们要新建一个n阶魔方， 只需这么写：

```
function NewCube(n) {
    let v = new THREE.CubeGeometry(1, 1, 1, n, n, n).vertices;
    let f = new THREE.CubeGeometry(1, 1, 1, n, n, n).faces;
    let cube = new Tcube(v, f);
    cube.fill(1, 0, 0, 1 / 2, red);
    . . .
    cube.addmove(-1, 0, 0, -thre, 'M', thre, 4);
    . . .
    let chara = ['二', '三', '四', '五', '六', '七'];
    cube.name = chara[n - 2] + '阶魔方';
    return cube;
}
```

因为three框架已经为我们提供了标准的网格立方体。

染色

然后是染色，这个很简单，一般的魔方都是在同一平面的色片为同一颜色，所以函数：

```
fill(x, y, z, v, c) {
    for (let i = 0; i < this.pieces.length; i++) {
        let cal = this.centers[i].dot(new THREE.Vector3(x, y, z));
        if (Math.abs(cal - v) < 0.0001) {
            . . . . .
        }
    }
}
```

即把

$$x * x_c + y * y_c + z * z_c = v$$

的所有面片上色，其中 x_c, y_c, z_v 是面片的几何中心。

```
cube.fill(1, 0, 0, 1 / 2, red);
cube.fill(1, 0, 0, -1 / 2, orange);
cube.fill(0, 0, 1, 1 / 2, white);
cube.fill(0, 0, 1, -1 / 2, yellow);
cube.fill(0, 1, 0, 1 / 2, blue);
cube.fill(0, 1, 0, -1 / 2, green);
```

转动

类似于染色，函数addmove()即把

$$x * x_c + y * y_c + z * z_c \geq v$$

的所有面片绕轴 (x, y, z) 顺时针旋转，不过对于旋转的度数，我采取了自动计算公约数的方法，就是算出最小的能够使几何形状重合的角度。当然必要时也能手动指定。

所以，比如说为正阶魔方添加转动，只需：

```
let thre = 1 / 2 - 1 / n - 0.01;
cube.addmove(-1, 0, 0, -thre, 'M', thre, 4);
cube.addmove(1, 0, 0, thre, 'R');
cube.addmove(-1, 0, 0, thre, 'L');
cube.addmove(0, 0, 1, thre, 'U');
cube.addmove(0, 0, -1, thre, 'D');
cube.addmove(0, -1, 0, thre, 'F');
```

这个函数对于每次调用同时加入了R,R2,R'等三个方向的不同转动。

求解

求解是在双向bfs的基础上，是一个通用架构，不管对于异形还是正阶代码通用。

```
function solve(curSet, aimSet, moveSet, num) {
    if (num === undefined) num = 1;
    for (let i of aimSet) {
        for (let j of curSet) {
            if (j.toString() === i.toString()) return [[]];
        }
    }
    let inverseSet = {};
    for (let move in moveSet) {
        inverseSet[move] = inversePerm(moveSet[move]);
    }
    ret = [];
    let stateMap1 = {};
    let stateMap2 = {};
    let q1 = [];
    for (let cur of curSet) {
        q1.push({state: cur, step: []});
        stateMap1[cur] = [];
    }
    let q2 = [];
    for (let aim of aimSet) {
        q2.push({state: aim, step: []});
        stateMap2[aim] = [];
    }
    let stepHalf = 5;
    let stepT = 5;
    while (q1.length !== 0 || q2.length !== 0) {
        if (q1.length !== 0) {
            . . . .
        }
        if (q2.length !== 0) {
            . . . .
        }
    }
    return ret;
}
```

在moveSet所指定的转动内，求解从curSet到aimSet的在num步以内的转动。

比如说要求解二阶魔方，

```
function New222Solution() {
```

```

let sol = new solver();
let samp = NewCube(2);
let fin = samp.initState;
sol.initialCopy = [fin];
sol.initial = [fin];
sol.aimSet = [fin];
sol.moveSet = {};
for (let i of ['R', 'R2', 'R\'', 'U', 'U\'', 'U2', 'F', 'F\'', 'F2']) {
    sol.moveSet[i] = samp.permutation[i];
}
sol.cube = NewCube(2);
let solution = [sol];
solution.cube = NewCube(2);
solution.name = '二阶魔方直接求解';
return solution;
}

```

solver是一个类，封装了初和终态，调用solver函数即可：

```
sol.got = solve(sol.initial, sol.aimSet, sol.moveSet, 10);
```

具体的细节处理涉及到多阶段求解时网页的交互，就不赘述了。