中国科学院大学
University of Chinese Academy of Sciences

《高级软件工程》课题项目

研究项目报告

项目名称 Cube Solver

成　　员 顾子茵　陈子轩　王伟

指导教师 罗铁坚　教授

完成日期：2019 年 12 月

目　录

一、项目简介

众所周知，利用传统方式去探索博奕类游戏（比如魔方、棋牌等）的解是非常困难的，拿魔方来说，仅三阶的全部解空间就为4.33×10^{19}，靠传统人力是无法探索到全部的解。而借助机器学习我们可以探索博弈类游戏的解，帮助玩家更好地参与到游戏中。

本课题的研究对象是魔方。魔方作为一种益智玩具，它的娱乐性深受大众的喜爱，但是魔方的规则复杂，初玩魔方的新手总出现无从下手的境况。

本课题的目标为：借助开发的网页让魔方小白也能实现还原魔方。

本课题主要工作如下：

（1） 分析对比了现有的解魔方软件/网页存在的不足

（2） 设计并实现让魔方小白更快还原魔方的解决方案

① 前端：实体魔方输入方式以拍照为主，同时辅以涂色修正，针对高手玩家也可直接通过旋转序列输入；提供解魔方过程进度条功能

② 后台：通过实验对比当前主流的解魔方算法，同时兼顾了用户体验，选择改进的 Kociemba 算法作为 Cube Solver 的解魔方算法

（3） 针对已实现功能进行展望，提出下一步改进方案

通过我们的研究，实现了现有的解魔方软件/网页没有的功能。对于魔方小白，我们可以提供清晰的复原魔方的过程指导。

介绍视频：https://www.bilibili.com/video/av77772638/

演示网站：https://czx.ac.cn/cubesolver

GitHub：https://github.com/x5g/CubeSolver

二、需求分析

2.1 市场分析

首先我们对市面上已有的解魔方软件或是网页进行了分析，统计它们已有的功能，进而分析现存产品的存在的优点与不足。我们选择了十款主流的产品进行测评：Cube Station、超级魔方、极限魔方、智能魔方、Cube Explorer、cbsolver、3 阶魔方还原、alg.cubing.net、iamthecu.be、rubiksdiy.com、rubiks-cube-solver.com。

分析结果如表 1 所示。

表 1 市场上已有的解魔方软件/网页

	类型	自动打乱	初始状态输入方式			检查输入	还原魔方	步骤演示	结果记录	三维动画	配件及价格
			拍照	涂色	旋转						
Cube Station	手机 APP	√	√	×	√	√	√	√	√	√	Gan356i 智能魔方 淘宝 429R
超级魔方		√	√	×	√	√	√	√	√	√	计客 SUPERCUBE 淘宝 199R/289R
极限魔方		√	×	√	×	√	√	√	√	√	圣手魔方 淘宝 7.9R
智能魔方		√	×	×	√	√	√	√	√	√	小米智能魔方 小米商城 79R
Cube Explorer	电脑软件	√	√	√	√	√	√	×	√	×	无
cbsolver		√	×	√	√	√	√	√	√	×	无
3 阶魔方还原		√	×	√	×	√	√	√	×	√	无
alg.cubing.net	网页	×	×	×	√	×	×	√	×	√	无
iamthecu.be		√	×	×	√	√	×	×	×	√	无
rubiksdiy.com		×	×	√	×	×	√	√	×	×	无
rubiks-cube-solver.com		√	×	√	√	√	√	√	×	×	无
Cube Solver	网页	√	√	√	√	√	√	√	√	√	无

对表 1 的分析，可以得出现有的竞争品各自的最大不足为：

（1）手机 APP：需要特定实体价钱贵的特定魔方实体配件

（2）电脑软件：需要特定操作系统平台的支持

（3）已有网页：功能不全，自定输入、解魔方、3D 动画没有兼得

2.2 用户故事

作为一个魔方小白，希望不用复杂的过程就可以还原一个被打乱的魔方。

2.3 基本需求

综上，我们提出以下基本需求：

(1) 基于网页，无需特定魔方实体配件

(2) 以尽可能快的方式准确的将实体魔方输入到解魔方页面中

(3) 尽可能快的给出解魔方步骤，解魔方的过程可以自由前进后退

(4) 实现魔方的 3D 转动效果

(5) 允许按进度条查看详细步骤分解

(6) 结果记录支持离线保存，方便查看。

三、系统设计

3.1 整体功能设计

系统整体功能设计如图 1 所示。

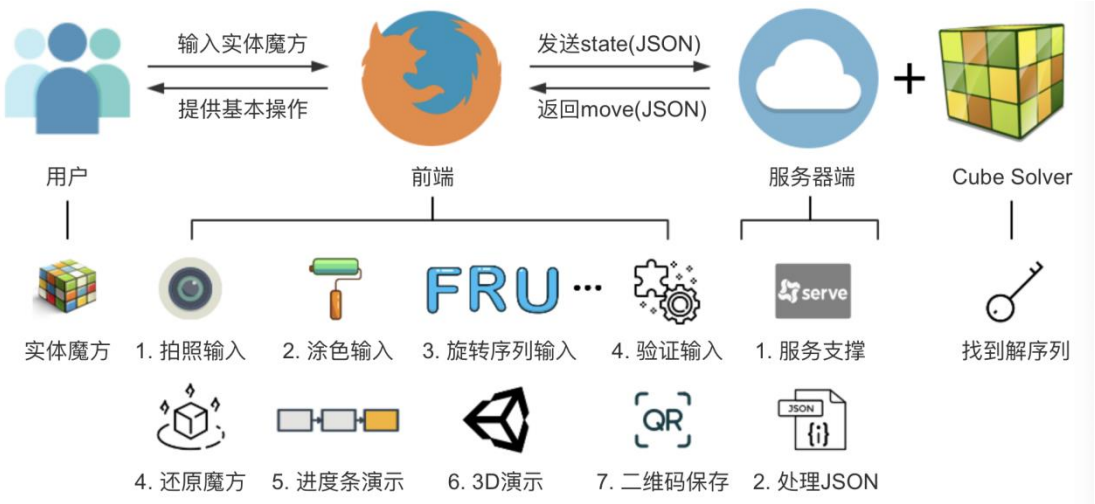


图 1 系统整体功能设计图

3.2 具体功能设计

3.2.1 实体魔方输入方式

传统网页输入方式为涂色和旋转序列，但在实际体验过程中如下问题：

（1） 涂色需要涂 54 个面，涂错后需要一点一点的查错，且这种方式需要一定的空间想象力

（2） 旋转序列输入对于魔方小白是极其困难的，需要提前学习一定的魔方旋转序列知识

综上，传统的两种输入方式对魔方小白而言是不友好的，有没有一种快速的便捷的输入方式呢？答案是肯定的，借鉴手机 APP 和电脑软件的功能，可以想到用摄像头扫描输入。

通过摄像头输入后，需要对输入图像进行识别，这里采用 RGB 转换为 HSV 的方式，对 HSV 进行区间设置，从而对应不同的颜色。

RGB 是目前运用最广的色彩模型（如图 2 所示），RGB 即是代表红、绿、蓝三个通道的颜色。其他颜色通过对红(R)、绿(G)、蓝(B)三个颜色通道的变化以及它们相互之间的叠加来得到。而 HSV 是一种比较直观的颜色模型，这个模型中颜色的参数分别是：色调（Hue, H），饱和度（Saturation, S），明度

（Value, V）。色调 H 用角度度量，取值范围为 0°～360°；饱和度 S 表示颜色接近光谱色的程度；明度 V 表示颜色明亮的程度。

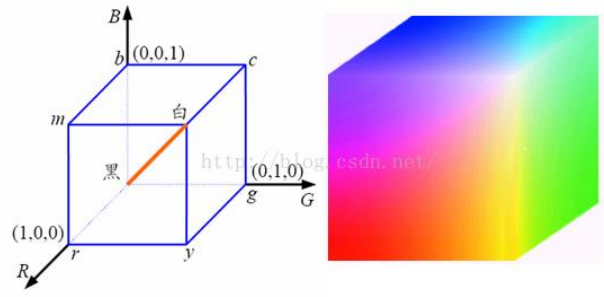


图 2 RGB 模型

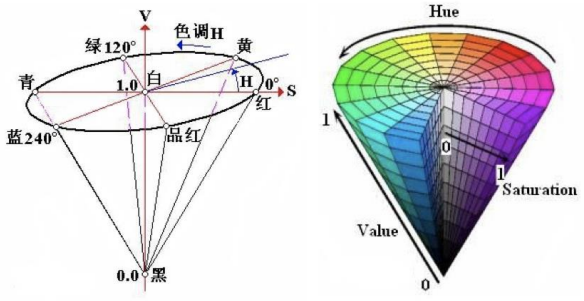


图 3 HSV 模型

RGB 到 HSV 的变换公式为

$$R' = \frac{R}{255} \quad G' = \frac{G}{255} \quad B' = \frac{B}{255}$$

$$C_{max} = \max(R', G', B') \quad C_{min} = \min(R', G', B') \quad \Delta = C_{max} - C_{min}$$

$$H = \begin{cases} 0°, & \Delta = 0 \\ 60° \times \left(\frac{G' - B'}{\Delta} + 0\right), & C_{max} = R' \\ 60° \times \left(\frac{B' - R'}{\Delta} + 2\right), & C_{max} = G' \\ 60° \times \left(\frac{R' - G'}{\Delta} + 4\right), & C_{max} = B' \end{cases} \quad S = \begin{cases} 0, & C_{max} = 0 \\ \frac{B' - R'}{\Delta}, & C_{max} \neq 0 \end{cases} \quad V = C_{max}$$

考虑到扫描输入方式会有一定的误差，所以必要的传统的输入方式应予以保留。扫描顺序设计为：

第一步，任选一面作为初始面，扫描初始面。

第二步，以俯视角顺时针旋转魔方 90°，扫描该面，并回到初始面。

第三步，以俯视角逆时针旋转魔方 90°，扫描该面，并回到初始面。

第四步，以右手为基点，向下旋转魔方 90°，扫描该面，并回到初始面。

第五步，以右手为基点，向上旋转魔方 90°，扫描该面，并回到初始面。

第六步，以俯视角旋转魔方 180°，扫描该面。

由此得到实体魔方输入界面设计：

（1） 一个摄像头扫描输入框，如图 4 所示

（2） 六个面的扫描结果展示框，如图 5 所示

（3） 保留传统的魔方输入方式：涂色区，如图 6 所示；旋转序列输入框

（4） 必要的展示魔方，如图 7 所示

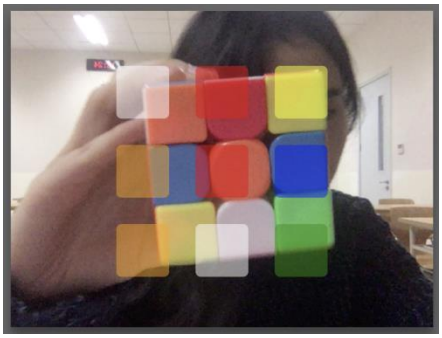


图 4 摄像头扫描输入框

图 5 六个面的扫描结果展示框

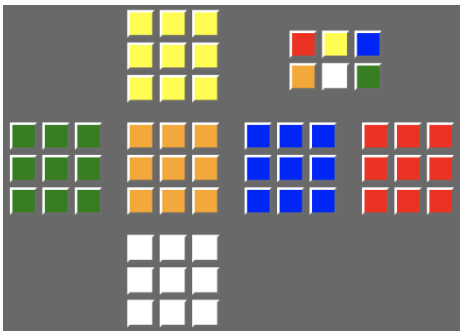


图 6 涂色区

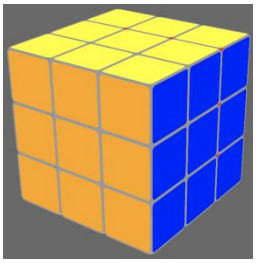


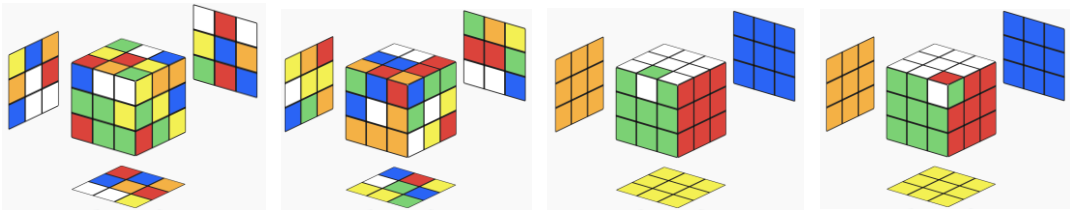
图 7 展示魔方

3.2.2 验证输入

不难想到，用户输入的魔方不一定是可解的，可能出现的错误包括：

（1） 缺少涂色或某一色块总数不对，如图 8(a)所示状态，其中白色块为 10 个，橙色块为 8 个。

（2） 中心块、棱块、角块的涂色为不可能状态，如图 8(b)(c)(d)所示。图 8(b)中不可能出现 6 个中心色块出现相同颜色的状态，图 8(c)中棱块颜色为不可解状态，图 8(d)中角块颜色为不可解状态。



(a) (b) (c) (d)

图 8 各种错误状态

这就需要一个验证输入模块来在实际解魔方之前进行验证是否可解。考虑到需要减轻服务器负担，因此我们将该模块设计为前台模块，由 JavaScript 进行验证。

当验证通过后，再由前台发送 json 格式的 state 编码给后台解魔方模块；否则向用户报错，并指出错误类型，以便用户根据错误反馈进行重新输入。

3.2.3 进度条及 3D 演示

考虑到用户需要根据网站一步一步的还原手中的实体魔方，显然需要设计可进退的进度条，同时辅助以 3D 魔方进行演示。

进度条及 3D 演示界面设计：

（1） 前台页面在接收到后台解魔方 move(JSON)后，可以出现解魔方的进度条，而且可以单击前进、后退按钮查看前后步骤，如图 9 所示。

（2） 魔方支持可旋转视角，配合进度条实现一面 3D 旋转，如图 10 所示。

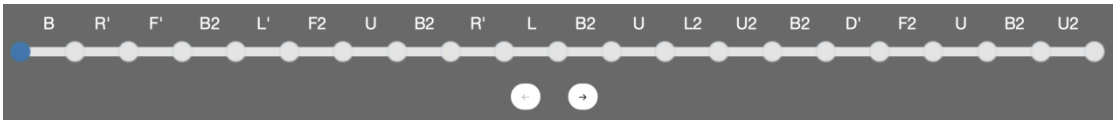


图 9 进度条演示

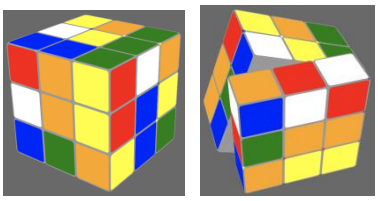


图 10 3D 演示

3.2.4 二维码保存

为方便手机用户可以查看或跟进别人或自己的解魔方过程，设计二维码保存模块，方便传播。具体设计为将魔方的待解状态和解序列保存到二维码内，由前台页面提供该二维码，使用手机或其他设备扫描该二维码，可以出现魔方及解法 3D 动画。如 11 所示。

图 11 二维码保存

3.2.5 架构模式选择

Django 框架为 models、template、views、urls 四个部分。

（1）Model（模型）：负责业务对象与数据库的对象（ORM）

（2）Template（模板）：负责如何把页面展示给用户

（3）View（视图）：负责业务逻辑，并在适当的时候调用 Model 和 Template

（4）urls 分发器，它将一个个 URL 的页面请求分发给不同的 view 处理，view 再调用相应的 Model 和 Template

3.2.6 解魔方算法及 API 设计

解魔方算法作为 Cube Solver 的核心功能，需要选择适合魔方小白的算法。首要的准则就是解法返回的解序列要尽可能的短，换言之，解序列包含的动作要尽可能的少。第二准则是要考虑到用户体验，试想一个解法需要 10 秒来反馈 15 步的解序列和另一个解法需要 2 秒来反馈 20 步的解序列，从用户体验的角度而言显然后一种算法更适合。

考虑到解耦设计原则，在后台中设计 API 接口。API 的输入参数为魔方的状态序列，输出为魔方的解序列。

四、系统实现

4.1 颜色识别

本课题最大的挑战之一就是颜色识别。计算机具有数字模型，找出像素在显示器上显示的颜色很容易，但是要读取摄像头的输入来弄清楚魔方的颜色分布是比较棘手的。因为魔方表面会反光，考虑到照明，如果亮度过高，魔方表面会显示眩光，因此我们能做的就是猜测颜色。

在设计阶段，我们选择通过将 RGB 转换为 HSV 进行识别颜色。HSV 转换到具体颜色的对应关系见表 2 所示。

表 2 HSV 转换到具体颜色的对应关系

	White	Red		Orange	Yellow	Green	Blue
H_{min}	0	0	$\frac{151}{180}$	$\frac{6}{180}$	$\frac{20}{180}$	$\frac{45}{180}$	$\frac{100}{180}$
H_{max}	1	$\frac{6}{180}$	1	$\frac{20}{180}$	$\frac{45}{180}$	$\frac{100}{180}$	$\frac{151}{180}$
S_{min}	0	$\frac{30}{255}$					
S_{max}	$\frac{100}{255}$	1					
V_{min}	$\frac{130}{255}$	$\frac{50}{255}$					
V_{max}	1	1					

4.2 魔方验证

采用测试用例驱动开发。首先，编写好测试输入，然后在 GitHub 上找寻通过全部测试用例的项目。

测试样例包括图 8 中的各种错误状态及其他合法随机状态，如图 12 所示。

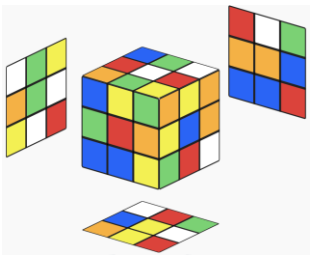


图 12 合法随机状态

经我们测试，许许多多的验证模块仅是对六种颜色的数目进行验证，而非真正验证用户输入是否合法。存在一种错误状态输入，普通的验证模块是排除

不了的：仅交换任意两个面的中心块颜色，如图 13 所示。这一状态显然是错误输入的，但很多验证模块依然验证通过，交给后台算法，同样后台算法也可产生所谓的还原解序列 U D R2 L2 U D' R2 L2 D2，但实际上并无有效解，如图 14 所示。

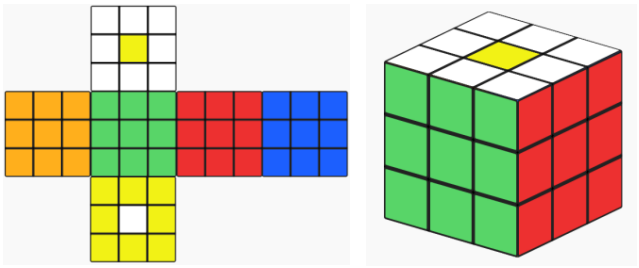


图 13 错误状态输入

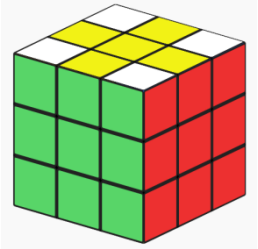


图 14 无效解

我们选择了通过该测试的一个项目，作者为 mfeather1，在此表示感谢。

完整的代码在 verify.js 和 rclib.js 中，可以在我们的 GitHub 中找到。

4.3 解魔方算法及 API

4.3.1 对比实验

针对系统设计中提到的两个原则，我们首先对比了目前已有的 6 种解魔方算法。实验设计为 1000 组输入，对比输出解序列的长度。现将实验过程及结果反思详细说明。

（1） 两种计数方式

在选择解魔方算法时，我们发现针对相同的输入，不同的算法输出格式会有差别，例如

输入旋转序列为：

Input: D U' B' U' B D' B' D' L2 U B F' D U' B R' U' D2 U' F' U L'

其对应魔方状态如图 15 所示。

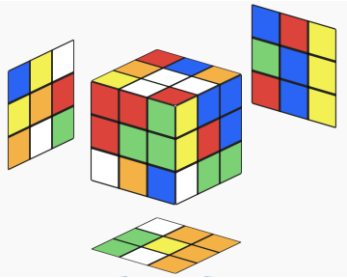


图 14 无效解

选择两个不同的算法 A、算法 B，

算法 A 的实际输出为：

OutputA0: F B' L' F2 B' L D' F2 D2 F' U L' F2 R U2 F2 U2 R L D2

算法 B 的实际输出为：

OutputB0: L U' F U D D U R B' U D' F B' U' L L D B D B' U B U D'

经检测，两个算法的输出均可达到还原魔方的效果。

但有不同之处在于算法 A 中出现了类似“D2”的步骤，而算法 B 中无此步骤，取而代之的是“D D”，为统一标准，不妨以两种不同的计数方式分别统计：计数方式 1 将“D2”视为步数为 1，计数方式 2 将“D2”视为步数 2。

其意义在于在两种不同的计数方式下，两种算法所表现的优劣不同。例如，在方式 1 下：

算法 A 的输出为：

OutputA1: F B' L' F2 B' L D' F2 D2 F' U L' F2 R U2 F2 U2 R L D2 (20f)

算法 B 的输出为：

OutputB1: L U' F U D2 U R B' U D' F B' U' L2 D B D B' U B U D' (22f)

显然在方式 1 下，算法 A 是优于算法 B 的。而在方式 2 下：

算法 A 的输出为：

OutputA2: F B' L' F F B' L D' F F D D F' U L' F F R U U F F U U R L D D (28f)

算法 B 的输出为：

OutputB2: L U' F U D D U R B' U D' F B' U' L L D B D B' U B U D' (24f)

显然在方式 2 下，算法 B 是优于算法 A 的。

由此在两种不同的计数方式下，分别进行实验。

（2） 实验过程及结果

输入文件在我们的 GitHub 中 CubeSolver/test/compare/result.xls 文件中可以找到。

对比的 6 种算法为：O-Kociemba(18, 5), O-Kociemba(20, 2), DeepCubeA, LayerFirst, CFOP, Kociemba。

实验结果为：在计数方式 1 中，图 15 中的曲线是 6 种算法的比较结果，图 16 是放大图 15 中步数为 15-40 的曲线。在计数方式 2 中，图 17 中的曲线是 6 种算法的比较，图 18 是放大图 17 中步长为 15-45 的曲线。表 3 给出了具体的量化对比数据。

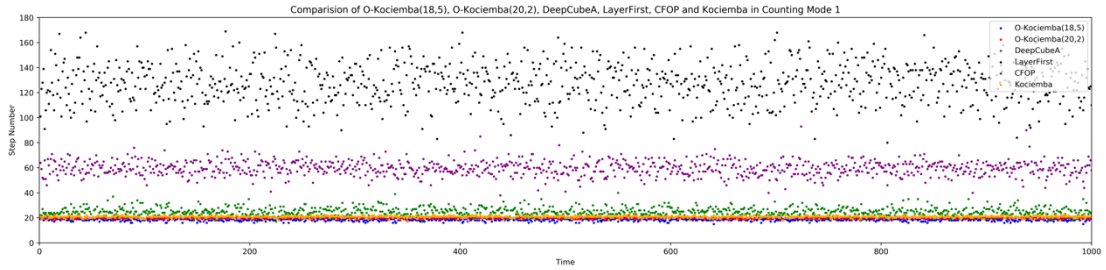


图 15 O-Kociemba(18, 5)、O-Kociemba(20, 2)、DeepCubeA、LayerFirst、CFOP 和 Kociemba 算法的在计数方式 1 中的步数对比

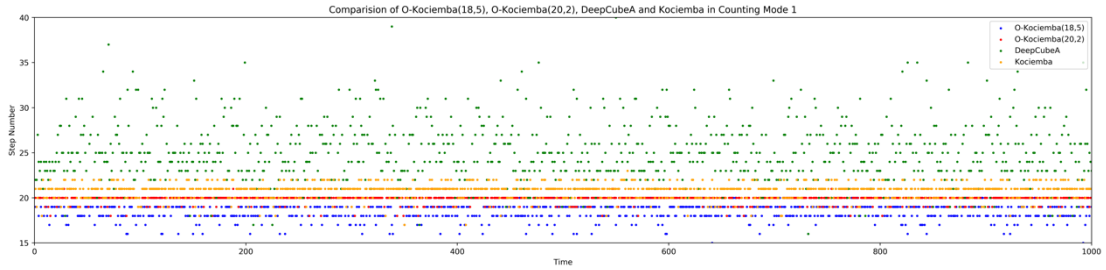


图 16 O-Kociemba(18, 5)、O-Kociemba(20, 2)、DeepCubeA 和 Kociemba 算法的在计数模式 1 中的步数对比

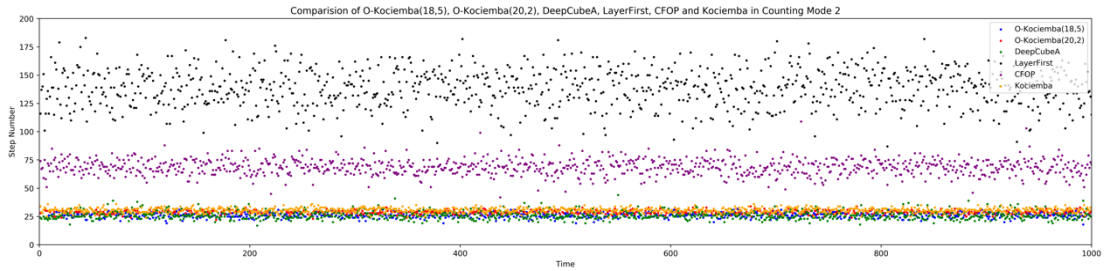


图 17 O-Kociemba(18, 5)、O-Kociemba(20, 2)、DeepCubeA、LayerFirst、CFOP 和 Kociemba 算法的在计数方式 2 中的步数对比

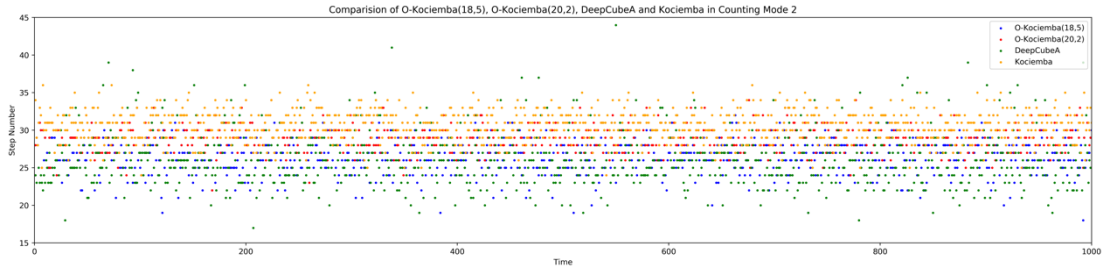


图 18 O-Kociemba(18, 5)、O-Kociemba(20, 2)、DeepCubeA、LayerFirst、CFOP 和 Kociemba 算法的在计数方式 2 中的步数对比

表 3 O-Kociemba(18, 5)、O-Kociemba(20, 2)、DeepCubeA、LayerFirst、CFOP 和 Kociemba 算法的详细对比数据

Comparison of O-Kociemba(18,5), O-Kociemba(20,2), DeepCubeA, LayerFirst, CFOP and Kociemba in Two Different Counting Modes						
	Average step number		Minimum step number		Maximum step number	
	Mode 1	Mode 2	Mode 1	Mode 2	Mode 1	Mode2
O-Kociemba(18,5)	18.597	26.707	15	18	20	33
O-Kociemba(20,2)	19.802	28.991	17	22	21	34
DeepCubeA	24.934	26.292	16	17	40	44
LayerFirst	127.684	138.774	80	87	169	183
CFOP	59.635	68.781	38	42	93	109
Kociemba	20.792	30.554	17	24	23	36

（3） 结果分析与反思

实验比较了 O-Kociemba(18, 5)、O-Kociemba(20, 2)、DeepCubeA、LayerFirst、CFOP 和 Kociemba 算法在相同的 1000 组输入下的输入长度。其中在人类手动解魔方算法中，应用最广泛的为 LayerFirst 法（层先法、公式法）及高阶算法 CFOP 算法，显然经过对比实验，这两种算法是远远不如其他算法的。

在机器解魔方算法中，Kociemba（又称 TwoPhase）算法作为其基础算法，其原理是根据魔方的色块性质将魔方的状态集划分为两个子集 G1 和 G2，状态机 G1 是为魔方的初始状态经过若干步规定以内的旋转所组成的状态集。该状态集中状态数量较小。状态集 G2 表示以 G1 的所有状态为原始状态，经过若干次任意旋转后的状态集。Kociemba 算法首先使用 IDA*搜索转换到状态集 G1，然后使用另一个 IDA*搜索转换为原始状态。

Kociemba 算法经过多年发展已出现优化版，优化过程体现在 G1 和 G2 阶段中的连接选择，将两个局部最优过程经过训练融合成整体。这里将改进版的

Kociemba 算法称为 O-Kociemba 算法，详细训练过程不做展示，感谢作者 hkociemba。需要指出的是，O-Kociemba 算法的第一个参数是期望步数，第二个参数是极限时间。显然，当时间限制较大时，O-Kociemba 算法可以探索更多的解状态。

DeepCubeA 算法是由 UCI 的计算机科学教授们研究的可以自动学习如何破解魔方的人工智能系统。DeepCubeA 的任务是需要尽快的找到最低的“功能成本”来破解魔方，其中包括计算成本和移动量。研究人员选择反向训练，将 DeepCubeA 放置于序列的的一个特殊状态，让它从已经破击的部分中开始进行深度强化学习。

结果反思。为什么在两种不同的计数方式下会各有优劣呢？事实上，在（1）中的算法 A 即是 O-Kociemba(20,2)算法，算法 B 即是 DeepCubeA 算法。我们考虑实际算法应用，DeepCubeA 可以说是目前为止解魔方算法里最快的了，这个“快”体现在使用机器解魔方，应用对象类似于图 19 所示的机器手臂。

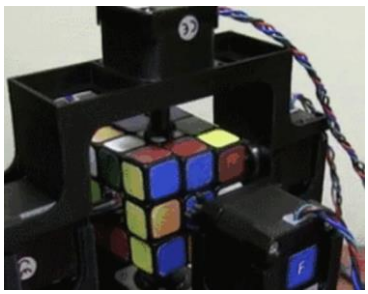


图 19 机器手臂

针对机器手臂，将魔方任意一面旋转 90° 所耗费时间是固定的，与机器机电转动速度有关。所以计数方式 2 是应用于机器手臂解魔方的，越少的单步操作，机器解魔方所耗时间即越短。

而作为人类来说，并不关心以单面旋转 90° 还是 180° 所耗费时间，大可将其认为是一步，所追求的应该是在计数方式 1 下的最少步数。因此，我们以计数方式 1 为用户的实际体验标准。考虑到用户体验的影响，我们选择 O-Kociemba(20,2)作为我们的解魔方算法。

这里给出 O-Kociemba(20,2)和 DeepCubeA 的对比测试样例视频：https://www.bilibili.com/video/av77760383/及对比测试网站：https://czx.ac.cn/cubesolvertest

4.3.2 O-Kociemba 算法及 API

首先，该算法将魔方的六个面状态编码为表 4 的形式。

算法的输入为魔方状态 state 字符串 UBL...，意思为在当前在 U1 位置的状态是初始魔方为 U 的状态，在 U2 位置的状态是初始魔方为 B 的状态，在 U3 的位置的状态是初始状态为 L 的状态等。state 字符串长度为 54，顺序为 U1 U2 U3 U4 U5 U6 U7 U8 U9 R1 R2 R3 R4 R5 R6 R7 R8 R9 F1 F2 F3 F4 F5 F4 F5 F6 F7 F6 F7 F8 F9 D1 D2 D3 D4 D5 D6 D7 D8 D9 L1 L2 L3 L4 L5 L6 L7 L8 L9 B1 B2 B3 B4 B5 B6 B7 B8 B9。例如，一个已求解魔方的 state 字符串定义就是 UUUUUUUUURRRRRRRRRFFFFFFFFFDDDDDDDDDLLLLLLLLLBBBBBBBBB。

表 4 O-Kociemba 算法下魔方的六个面状态编码形式

O-Kociemba 算法下魔方的六个面状态编码形式			
	U1 U2 U3 U4 U5 U6 U7 U8 U9		
L1 L2 L3 L4 L5 L6 L7 L8 L9	F1 F2 F3 F4 F5 F6 F7 F8 F9	R1 R2 R3 R4 R5 R6 R7 R8 R9	B1 B2 B3 B4 B5 B6 B7 B8 B9
	D1 D2 D3 D4 D5 D6 D7 D8 D9		

算法的输出为 move 字符串，该字符串每个部分由空格分隔，每个部分表示单个移动：单个字母意味着顺时针旋转 90 度；后跟单引号的字母表示将该面逆时针旋转 90 度；后面带有数字 2 的字母表示将该面旋转 180 度。

例如 R U R' U R U2 R' U

其次，我们实现了将用户的颜色输入方式转换为算法的状态输入方式。具体过程为：首先获取每个面的中心色块对应的颜色，然后按照算法魔方的输入序列顺序转换并重排。其主要伪代码如表 5 所示。

表 5 将用户的颜色输入方式转换为算法的状态输入方式主要伪代码

两种输入方式的状态转换伪代码
var U5 = facelets_arr[04], R5 = facelets_arr[28], F5 = facelets_arr[25], D5 = facelets_arr[49], L5 = facelets_arr[22], B5 = facelets_arr[31] var solve_facelets_arr_temp = []; for (i = 0; i < facelets_arr.length; i++){ swich (facelets_arr[i]) { case U5: solve_facelets_arr_temp.push('U'); case L5: solve_facelets_arr_temp.push('L'); case F5: solve_facelets_arr_temp.push('F'); case R5: solve_facelets_arr_temp.push('R'); case B5: solve_facelets_arr_temp.push('B'); case D5: solve_facelets_arr_temp.push('D'); } }

最后我们给出该算法的调用 API。在我们的网页中的代码为表 6 所示，对外调用方式为：输入参数为 state(JSON)，返回为 move(JSON)，样例如表 7 所示。

表 6 前台页面中调用后台解法 API 的代码

前台页面中调用后台解法 API 的代码
$.get("solver", { 'state' : state }, //state is a cube definition string function(data) { //data.move is the sequence of soling cube sol = data.move.replace(/^\s+\|\s+$/g, "").replace(/\s+/g, " "); } }

表 7 对外调用 API 方式

对外调用 API 方式
get:https://czx.ac.cn/solver?state=LFBDUULLFULUFRRBLLUFRLFDLRRD BDBDBDBDBRFFLDRUBRUUUBRFDF return: {"move": "L2 B2 U2 L2 D2 L2 R' D R' D2 L' F D2 U2 R' D2 R' D' F'"}

五、系统测试

5.1 自动化测试

我们选择浏览器自动化测试框架——Selenium 进行测试。

该测试框架具有以下功能：

（1） 框架底层使用 JavaScript 模拟真实用户对浏览器进行操作。测试脚本执行时，浏览器自动按照脚本代码做出点击，输入，打开，验证等操作，从终端用户的角度测试应用程序。

（2） 使浏览器兼容性测试自动化成为可能，尽管在不同的浏览器上依然有细微的差别。

（3） 使用简单，可使用 Java，Python 等多种语言编写用例脚本。

我们选择 Python 语言编写自动化测试。测试内容主要为：

（1） 模拟用户输入，测试在瞬间和进度条分步执行下魔方能否正确还原

（2） 输入无解的状态，测试检验模块是否能完成判断

其中测试魔方打乱及还原采用 WCA 魔方打乱公式分别进行涂色输入和旋转序列输入。特别说明旋转测试的方案：将旋转序列保存在输入文件 input.txt 中，并运行测试文件 automaticTest.py 进行测试。可以从我们 GitHub 上的 test 文件夹中找到 automationTest.py 和 input.txt。其中 input.txt 的内容样例如表 8 所示。

表 8 input.txt 的内容样例

input.txt 的内容样例
D2 L R2 B' F' U2 F L' D' B' F' D' U' B2 F'
D2 U L2 R D L2 R2 U F2 L D2 F2 L R2 F
L' R' B2 F2 U2 F2 D2 B2 F' L2 D2 U2 B R2 B
D' U F' D2 U' L R2 D' U2 R D2 L R' D U2

自动化测试样例录像：https://www.bilibili.com/video/av77760268/

5.2 其他测试

因技术所限，扫描输入无法进行自动化测试，我们选择了市面上最常见的两种实体魔方，实色魔方和贴纸魔方，如图 19 所示。经过在不同光线下的测试，实色魔方明显比贴纸魔方识别成功率要高。这与我们的 HSV 识别区间设计有关。

图 19 两种实体魔方

六、不足与展望

6.1 不足

如果进度条单击得太快，演示魔方将无法恢复。这是因为 Cube.js 在还原时会有一定的延迟。如果间隔短于此延迟，则将发生此问题。

6.2 展望

在扫描输入阶段，可以添加一些过滤规则以帮助猜测标签的颜色。例如，一个不错的选择是隐马尔可夫模型（Hidden Markov Model, HMM）

七、致谢

1. hkociemba，我们解魔方算法文件 RubiksCube-TwophaseSovler 的作者。
2. petr-lee，我们扫描魔方模块文件 RubikScan 的作者。
3. mfeather1，我们涂色及验证魔方模块文件 3ColorCube 的作者。
4. kmh0228，我们展示魔方模块文件 Cube.js 的作者。
5. iyangyuan，我们解魔方进度条模块文件 ystep 的作者。
6. carbon-app，我们代码高亮图片文件 carbon 的作者。
7. Forest Agostinelli, Stephen McAleer, Alexander Shmakov, Pierre Baldi，感谢你们将 DeepCubeA 算法开源。
8. 彭前程、齐兵、段旭，感谢你们在分析 DeepCubeA 算法时的帮助。
9. 阿里巴巴云计算，我们的服务器提供商，感谢你提供的低价服务器。
10. 罗铁坚教授，感谢您提出的魔方课题。