CS4099
Rubik's Cube Reference Guide

Samuel Borg

October 9, 2024

1 Representing Cube State

First, the physical puzzle needs to be represented by a state that can be manipulated by a computer. For the 3x3, the state is represented by giving the position and orientation of each of the 8 corner cubies and 12 edge cubies (where the 6 centre cubies form a fixed reference for the cube). This is represented using **an array of 20 elements (one for each cubie) where each element encodes the position and orientation of the cubie** (8 positions $\times$ 3 orientations = 24 different values for the corners; 12 positions $\times$ 2 orientations = 24 different values for the edges). These days, it is more efficient to use separate arrays for the corner and edge cubies. The array elements should contain a position index (0-12 for edges; 0-7 for corners) and an orientation (0 or 1 for edges; 0-2 for corners). The order of the array is arbitrarily decided.

Primitive operators on the cube are defined as any twist of a single face (either 90 degrees clockwise/anticlockwise, or 180 degrees). This results in a search tree with a branching factor of 18 (as there are 6 different faces with 3 possible operators per face). We can reduce this by **disallowing any moves on the same face twice in a row** (e.g. turning a face 90 degrees twice is the same as turning the face 180 degrees once), which brings the branching factor after the first move down to 15. We can reduce this number even further by using the fact that twists of opposite faces are independent and commutative (i.e. twisting one face and then twisting the opposite face gives the same state as performing the same actions in the opposite order). **For each pair of opposite faces, we need to arbitrarily choose an order, and then disallow consecutive moves on the two faces in the opposite order.** This results in a branching factor of about 13.34847.

2 Search Algorithms & Heuristics

We need an admissible search algorithm in order to find optimal solutions. Since the problem space is very large, we can't use exponential-space algorithms like A*. Instead, we use **IDA* (iterative-deepening-A*)**. IDA* is a depth-first search that looks for increasingly longer solutions in a series of iterations, using a lower-bound heuristic to prune branches once their estimated length exceeds the current iteration bound.

The heuristic function relies on **pattern databases** in order to ensure a solution can be found in a timely manner. Three pattern databases are created - one for the corner

cubies, another for 6 of the edge cubies and the last for the other 6 edge cubies. Each pattern database stores the number of moves required to solve the set of cubies (corners / first six edges / other six edges) for any cube state. The pattern databases are generated (ahead of time) by using a breadth-first search from the goal state (i.e. from the solved cube), pruning off branches of the search tree where a state has been seen before with fewer moves. It should also be noted that the position and orientation of the final cubie in a set is determined by the other cubies in the set.

The pattern databases are represented as **hash tables**. In order to ensure the databases don't become too large, a "perfect hash" algorithm is required - one which takes a cube permutation and returns a hash index into the table, which is achieved using **Lehmer codes**. The algorithm used to create these hash indices is as follows (described by Korf):

> We scan the permutation from left to right, constructing a bit string of length n, indicating which elements of the permutation we've seen so far. Initially the string is all zeros. As each element of the permutation is encountered, we use it as an index into the bit string and set the corresponding bit to one. When we encounter element k in the permutation, to determine the number of elements less than k to its left, we need to know the number of ones in the first k bits of our bit string. We extract the first k bits by right shifting the string by n - k. This reduces the problem to: given a bit string, count the number of one bits in it.
>
> We solve this problem in constant time by using the bit string as an index into a precomputed table, containing the number of ones in the binary representation of each index.

To summarise:

- Generate three pattern databases (corners, first 6 edges, other 6 edges) ahead of time by using breadth-first-search.
  - Each pattern database is a hash table, where the values are the number of moves required to solve that set of cubies (position and orientation in correct place).
  - The hash indices are Lehmer codes (see above algorithm)
- When solving a cube, use IDA* (iterative-deepening-A*), which is a depth-first search that uses a lower bound heuristic to prune branches.
  - The lower bound heuristic is the maximum of the values in the three pattern databases for that cube state.

3 References

`https://www.cs.princeton.edu/courses/archive/fall06/cos402/papers/korfrubik.pdf`

`https://stackoverflow.com/questions/58860280/how-to-create-a-pattern-database-for-sol`

`https://cdn.aaai.org/AAAI/2005/AAAI05-219.pdf`

https://medium.com/@benjamin.botto/sequentially-indexing-permutations-a-linear-algori
(paywalled)