Efficient 3-Phase 2x2x2x2 Solving Algorithm

By Anderson Taurence

Structure

The 2x2x2x2 Rubik's cube has 16 pieces with 4 stickers each.

Permutation

Pieces can be permuted in any even permutation. Since we define cube rotations to not change the puzzle state, we can always consider at least one piece to be solved. This reduces the number of pieces that may be permuted to 15. Therefore the total number of permutation states is $653,837,184,000$.

Orientation

The orientation of a piece can be thought of as a permutation of its stickers where only even permutations are possible. The orientation of all pieces is not completely independent. If the orientation of 15 pieces has been defined then there are only 4 possible orientations for the last piece. Again, due to cube rotations we may consider the orientation of a single piece to be solved. Therefore the total number of orientation states is $5,135,673,858,195,456$.

Total

The total number of permutation for the 2x2x2x2 is $3,357,894,533,384,932,272,635,904,000$ or about 3.4×10^{27}. This is nearly 100 million times larger than the classic 3x3.

Terms

Reference Axis 1: the axis which will be oriented first (W axis internally)
Reference Axis 2: the axis which will be oriented second (Z axis internally)
Fixed Piece: the piece on the negative side of all axes is considered fixed and all moves operate on the positive side of their axis

Phase 1

Details

Goal: Orient all pieces such that all stickers belonging to reference axis 1 are in reference axis 1.

Number of states: 1,07,37,41,824

Available moves: 92 (all)

Implementation

States and Moves

States are represented by an array of length 15 which holds the K4 orientation state as an integer (0-3) of the piece in each position. Moves are represented by 2 arrays of length 15. One holds the permutation of the move in the ‘moves to index’ format. The other holds the A4 action as an integer (0-11) that will be applied to the piece in that position. To compute a move you first take the state and permute the pieces according to the permutation array for the move. Then for every piece you replace its value with the value from a lookup table derived from the A4 group multiplication table. The lookup table looks like

[[0 0 0 1 1 1 2 2 2 3 3 3]
[1 2 3 0 3 2 3 0 1 2 1 0]
[2 3 1 3 2 0 0 1 3 1 0 2]
[3 1 2 2 0 3 1 3 0 0 2 1]]

and is indexed by [piece state, move action].

Index Encoding

The state array can be easily encoded and decoded from an integer by treating it as a base 4 encoding. This integer can be used as an index or key for the state in a pruning table array or hashmap.

Phase 2

Details

Goal: Orient all pieces such that all stickers belonging to reference axis 2 are in reference axis 2 and pieces are sorted to the correct side of reference axis 1.

Number of states: 30,778,405,515

Available moves: 44

Implementation

States and Moves

States are initially represented by 2 arrays of length 15. One holds the C3 orientation state as an integer (0-2) of the piece in each position. The other array (called IO permutation) holds a boolean for each position: 1 if the position is occupied by a piece from the negative reference axis 1 and 0 if the piece is from the positive reference axis 1. Moves are represented identically as in phase 1 but in phase 2 all moves are guaranteed to have an A4 action value less than 3 and they act exactly as cyclic orientations just like corners in 3D. To compute a move you first take the orientation state and permute the pieces according to the permutation array for the move. Then for every piece you replace its value with the value with (state value + move action) % 3. The permutation state can simply be permuted according to the permutation array for the move.

Coordinate Encoding

Since there are two array states that are completely independent, we can encode them into separate “coordinates” and then later combine them into an index. The C3 orientation array can be easily encoded as an integer by treating it as a base 3 number but throwing out the last value because it is determined by the first 14 values. For decoding, the last value can be determined by the condition that the sum of all entries mod 3 must be 0. Encoding the IO permutation array is significantly more challenging. First we convert the boolean array to a sorted array of 7 indices which contain true. Call this array x. The IO coordinate can be calculated as

$$6434 - \sum_{i=0}^{7} C(x[i],\ i + 1)$$

where $C(n, k)$ is the binomial coefficient n choose k. Reversing this coordinate to get the array is rather expensive and describing the algorithm would be complicated but it isn’t too hard to figure out. The solved state of both coordinates corresponds to 0. The index for pruning tables can be computed via (there are 2 ways to do it, I just chose this way)

$$IOcoord \times 3^{14} + C3coord.$$

Move Tables

Another way of computing moves when we have the luxury of multiple coordinates with relatively few states is to simply precompute the result of applying every move on every coordinate and use a lookup table to compute moves. Using move tables to compute a move on an index all we have to do is split up the index into its C3 and IO coordinate, perform a single lookup for each, then recombine them. This is blazingly fast.

The lookup table for C3 would contain 4,782,969 32 bit integers for 44 moves. Without any kind of compression this is about 850 MB which is admittedly large. You could save virtually all this space by using the array based computation which would only be slightly slower since the coordinate encoding and decoding is fast for the C3 coordinate. For the C3 coordinate you could sacrifice the move table if ram is an issue and if not then you might as well use it.

For the IO coordinate, using move tables is a must. The lookup table is only 6,435 16 bit integers for 44 moves. Without any kind of compression this is about 550 KB. There is no reason not to use this.

Phase 3

Details

Goal: Solve the puzzle.

Number of states: 101,606,400

Available moves: 12

Implementation

States and Moves

States are initially represented by an array of length 15 representing the piece permutation. Since the pieces are now sorted along the W axis, the piece numbers can be represented as their original value mod 8 since pieces will not be exchanged along the W axis. For example the solved state looks like

[0 1 2 3 4 5 6 7 0 1 2 3 4 5 6].

Moves are represented identically as in phase 1 but we don't need the A4 orientation array. To compute a move, the permutation state can simply be permuted according to the permutation array for the move.

Coordinate Encoding

There are too many states to encode the permutation into a single coordinate and use a move table so we have to break up the state into two separate arrays. One array for the positive W axis pieces (called the I coordinate) and one for the negative W axis pieces (called the O coordinate). There is a slight sacrifice in doing that. We know that the parity of the overall permutation must be even but for the sub-arrays, they may be either both even or both odd. Since we want to compute the coordinates with separate move tables, we must make the tables twice as large since we have both even and odd permutations possible for each coordinate (just

never different parity at the same time). We have to pay this price of redundancy if we want to use move tables. A bit of care needs to be taken when converting from coordinates to arrays to correctly account for the parity.

I will go through the example for the I coordinate but the same process works for encoding the O coordinate. Start with the subarray for the I state and call it x. First we encode it ignoring the parity by skipping the first 2 indices and computing

$$c = \sum_{i=2}^{8} \frac{i!}{2} \left(\sum_{j=0}^{i} (x[j] > x[i]) \right).$$

Loop through the indices starting at 2 and multiply the number of values at smaller indices that are greater than the value at index i and multiply it by $\frac{i!}{2}$. Then you compute whether x is an even permutation or odd permutation. If it is odd then you add $8!/2$ to c and that is the I coordinate. The same thing can be done with the O sub-array but replace any 8 with 7.

The reason we compute them this way is so that all the even permutations are ordered first and then all the odd permutations. This lets us perform a really cheap check to see if a coordinate represents an even or odd permutation.

When we convert the coordinates to an index we use the formula

$$index = Ocoord \times \frac{8!}{2} + (Icoord \,\%\, 2).$$

This throws away the information about the parity of the I coordinate (since we know it must match the O coordinate). When we decode this we just need to add $8!/2$ if the O coordinate is an odd permutation.

Move Tables

Using move tables for this stage is necessary. For the I coordinate the lookup table would contain 40,320 16 bit integers for 12 moves which without compression is about 1 MB. For the O coordinate the lookup table would contain 5,040 16 bit integers for 12 moves which without compression is about 120 KB.

Pruning Table

For phase 3 there are only 101,606,400 states and with 12 moves available it is possible to compute the complete pruning table. Storing an 8 bit integer at each index, the table is only 100 MB. The maximum depth is 21 moves STM so this phase is very inefficient but this phase will rarely need anywhere near this number of moves since we can search other phase 1 and phase 2 solution combinations and we will almost instantly find very short phase 3 solutions.

Pruning Table Computation

The pruning table is designed to be a lookup table which gives a lower bound on the number of moves required to solve a state.

For pruning table computation we use a simple depth first search. First, a pruning table is created. This can either be an array of 8 bit integers (in which case you need to choose a value to represent an empty entry and initialize the pruning table with it. I recommend using the value 1 greater than the maximum pruning depth so that these values represent a lower bound on the solution length) or a hashmap with 8 bit integers are the values. Then 2 queues are created. A binary variable is created to keep track of which queue is the working queue. The solved state is marked as 0 depth in the pruning table and is initialized in the working queue. An integer variable is created with value 1 to keep track of the current depth of the search. While there are states in either queue and the depth is less than or equal to the maximum pruning depth: while there are items in the current queue take a state out of the working queue and iterate through all states generated by the available moves. If you have extra space in ram, store the axis of the previous move done to the state in the queue with the state so that you can filter out moves that have the same axis when generating new states since these will always result in already explored states and don't need to be computed. For every new state check the pruning table to see if it has been seen before. If it has not then write the current depth to the index of that state and add it to the other queue. Once the current queue is empty, switch which queue is the working queue and increment the current depth. Once the outermost while loop terminates (either by surpassing the maximum depth or by finding all the states) return or save the pruning table.

The way you store the states in the queue is decided by the amount of ram you have available. If you have to reduce ram usage then store the states in the queue as their pruning table index and convert them to array or coordinate form depending on the phase when they are pulled from the queue. If you have more ram available, store the states in the queue directly as an array or pair of coordinates depending on the phase. This allows you to compute new states without having to convert from an index every time.

Pruning table uncompressed sizes
Phase 1: ~1 GB
Phase 2: ~31 GB
Phase 3: ~100 MB

Since the phase 3 table is small and can be fully generated, it makes sense to store it as a simple array. For phase 2, the vast majority of states will be unexplored so it makes sense to implement it as a hash table so you can actually store it in memory. For phase 1 you could go either way. It depends on how many states you are exploring with the pruning table.

It is possible to compress the pruning table values from 8 bits to 4 bits if you are willing to give up being able to look up exactly how far a state is from solved. The lookup value would only tell

you if the state takes more moves to solve than the maximum depth. The table will still be able to tell you if, after applying a move, the state got closer or farther from solved.
(See: http://kociemba.org/math/pruning.htm “To reduce memory size”…)

Full Solution Methods

Iterative Solution Shortening

This method will find at least one solution almost instantly. Then it will very quickly return much shorter solutions before slowing down eventually and becoming unusably slow at finding shorter solutions. If given enough time however, the algorithm will return an optimal solution and eventually verify that it is optimal. This method is good if you just want a reasonably short solution quickly.

This method iterates through all phase 1 solutions in order of increasing solution length. For every phase 1 solution, it will iterate through all phase 2 solutions in order of increasing solution length (with solutions that start with a move on the same axis as the last move from the previous phase so that a move can be canceled). For every phase 2 solution it will check the optimal phase 3 solution (again accounting for move canceling). If this solution is shorter than the shortest solution found yet then it will save the new solution. At every phase, if the solution so far is longer than the shortest solution then terminate that branch. Once the phase 1 solution is longer than the shortest solution, the shortest solution must be optimal and it can be returned, verified as optimal.

Iterative Deepening Search (IDS)

This method will return all solutions in order of increasing length. It is impossibly slow to use for any real scrambles. For scrambles which are known to have a very short solution this method might be useful. However the iterative solution shortening method will likely perform almost as well and it will give suboptimal solutions faster. This method is only useful if you want to verify that no solution exists for a state shorter than some small number of moves or if you know that there is a very short solution and want to find it.

This method begins by starting at depth 0 and for every depth it will iterate through all phase 1 solutions in order of increasing solution length. For every phase 1 solution, it will iterate through all phase 2 solutions in order of increasing solution length (with solutions that start with a move on the same axis as the last move from the previous phase so that a move can be canceled). For every phase 2 solution it will iterate through every phase 3 solution (again accounting for move canceling). If the total solution length is equal to depth then it will be yielded. At every phase, if the solution so far is longer than the shortest solution then that branch will be terminated. The first solution found by this method will be optimal and all subsequent solutions will be returned in order of increasing solution length.

Further Improvements

Parallel Search

Before solving the cube we treat one of the pieces as solved. However the orientation in which we consider the piece to be solved is arbitrary. There are 12 orientations for the piece in its place which can be accessed via cube rotations. What can be done is to create 12 copies of the cube we want to solve and cube rotate them such that the piece we are considering solved has a different orientation in each of them. Then we can run the solving algorithm in parallel on each of the 12 cubes. The iterative deepening search may not benefit from this as much but it would likely significantly speed up the iterative solution shortening method. The shortest solution could be shared between all the threads, allowing each solver to take advantage of the knowledge of any of the others.

Symmetry Reduction

There are almost certainly ways to reduce the number of unique states for coordinates in different phases by using symmetry reductions. However I fear that the lookup tables required for mapping between the symmetry reductions of these large groups would be prohibitively expensive.

Quarter Turn Metric (QTM)

This solver was created with STM in mind due to phase 3 requiring mostly double moves. This program could be modified to solve in QTM with a couple complications. Firstly, the pruning table search for phase 3 would need to use Dijkstra's algorithm (so the double moves can be weighted twice the single moves) rather than a depth first search which could be done using a bucket queue with 2 buckets. This would be less efficient than using breadth first search for STM but it should work. For the other phases, all you would have to do is undefined double moves and remove any optimizations that take advantage of moves with the same axis being always redundant. This would slow things down a little but perhaps some kind of lookup table could be used to check which moves are redundant.

For a simple adaptation for QTM, the full solution algorithm could be changed to measure the length of solutions in QTM. No more guarantees can be made about optimality of solutions and more solutions will need to be checked before terminating a branch but it would generate solutions shorter than the STM-based solutions.