Conquering the Cube with Cosets

Tomas Rokicki
rokicki@gmail.com

The group theoretic concept of cosets has proven effective in expanding our knowledge of Rubik's Cube, and promises to determine the God's number for the puzzle in the near future. We present four state of the art techniques to solve the Rubik's Cube. Two will find optimal solutions, and two will only find solutions of distance 20 or less. Two will solve arbitrary individual positions, and two, based on cosets, will solve large sets of related positions. The performance of these solvers varies by more than nine orders of magnitude, as shown in Table 1.

	Optimal	Near-optimal
Individual	0.36	3900
Coset	$2 \cdot 10^6$	10^9

Table 1: The solution rate, in positions per second, for four different algorithms for solving the cube.

1 Computer Cubing Milestones

To prepare the reader for the new techniques and results presented here, we must first recollect a bit of history. We give just enough detail to understand the key ideas in these algorithms. For more details on implementing puzzle solvers, see the Computer Puzzling page by Scherphuis [8]; for more details on how group theory applies to Rubik's Cube, see Joyner [2].

Two primary milestones in computer cubing are Kociemba's two-phase algorithm [3] from 1992, which was the first to quickly find near-optimal solutions for arbitrary positions, and Korf's optimal solver [4] from 1997, which could optimally solve arbitrary positions. These algorithms are based on iterated depth-first search. This search technique adds a depth limit to depth-first search, starting with a limit of zero and increasing it until the problem is solved. It can be considered a form of breadth-first search where the frontier is not stored explicitly, but is rather rediscovered as necessary. Both Kociemba's and Korf's algorithms explore the Cayley graph of the cube group, where the positions of the cube correspond to nodes of the graph and the moves of the cube correspond to edges of the graph.

Iterated depth-first search is identical to enumerating move sequences of the cube in order of their length. We can analyze the runtime of these algorithms by considering how many such sequences are so explored. We will primarily concern ourselves here with the half-turn metric, where there are 18 primary moves: a clockwise quarter twist, counter-clockwise quarter twist, and half-twist of each of the six faces. We use Singmaster notation, where the faces are labeled U, F, R, D, B, and L (for up, front, right, down, back, and left) and moves are represented by U, U^2, and U', for a clockwise twist, half twist, and counter-clockwise twist, respectively. The techniques we use apply similarly to the quarter-turn metric, where half twists are considered two moves.

Using the half-turn metric, any sequence with two consecutive quarter turns of the same face is not interesting, because there is a shorter sequence with a half turn of that face that leads to the same position. Thus, not all of the 18^d sequences of length d are worth exploring. Two simple rules are commonly used by cube programs to restrict themselves to interesting sequences. First, no turns of the same face can be adjacent. Second, any pair of adjacent commuting single moves (such as U and D) are always given in a particular order (we use U before D, F before B, and R before L). Any sequence that satisfies these two rules is considered a *canonical sequence* and cube programs will usually only explore such sequences. The number of canonical sequences of a given length is easily calculated with a simple recurrence, which we tabulate in Table 2.

Many interesting questions concern the distance of a position—the minimal number of moves required to attain that position from a solved cube (equivalently, the number of moves required to solve that position). For the values of d for which the count of positions at distance ($c(d)$) is known, the number of canonical sequences of that length ($f(d)$) is close to the number of positions at that distance; even at distance 13 (the deepest distance fully searched) they are within 10% of each other. Thus, cube programs based on canonical sequences are effective at exploring the space of the cube without exploring equivalent positions over and over

d	f(d)
0	1
1	18
2	243
3	3,240
4	43,254
5	577,368
6	7,706,988
7	102,876,480
8	1,373,243,544
9	18,330,699,168
10	244,686,773,808
11	3,266,193,870,720
12	43,598,688,377,184
13	581,975,750,199,168
14	7,768,485,393,179,328
15	103,697,388,221,736,960
16	1,384,201,395,738,071,424
17	18,476,969,736,848,122,368
18	246,639,261,965,462,754,048
19	3,292,256,598,848,819,251,200
20	43,946,585,901,564,160,587,264

Table 2: The number of canonical sequences of length d in the half-turn metric.

again; this is a major reason iterated depth-first search performs effectively.

2 An Optimal Solver

Korf's optimal solving algorithm is somewhat simpler than Kociemba's two-phase solver, so we will discuss it first. The key idea is to use a large table that gives a lower bound on the distance of a particular position. On every new node explored by depth-first search, if the distance given by that table for the current position is greater than the remaining depth permitted to be explored by this iteration of the search, we *prune* this node from the search; we backtrack. Thus these tables are called *pruning tables* (or *pattern databases*). Every pruning table has an *effective distance* which is roughly the distance at which search is typically cut off. The larger the table, the greater the effective distance and the faster the overall algorithm.

We use the term *effective distance* without giving a formal definition; it is intended to reflect the efficacy of the pruning table more than its average distance or some other more easily measured quantity, to simplify our analysis. The effective distance is usually not far from the average distance; indeed, if it is far, that indicates a defect in the chosen pruning table.

With a pruning table of effective distance d, during iteration n of the depth-first search in Korf's algorithm, we will explore all canonical sequences of length $n-d+1$. The runtime of the algorithm is directly proportional to the number of sequences we must explore before we find a solution; for typical positions, a solution is found at depth 18. For simplicity, we will assume the solution is found halfway through a depth 18 search. Using a reasonable pruning table of, say, 8GB, we can attain an effective distance of about 12, so the runtime for a typical position should be proportional to the exploration of the number of canonical sequences of length 6 or less. Table 2 shows about eight million such sequences so we expect the average runtime to be that required to explore about that many positions.

We have implemented Korf's algorithm using such a large pruning table, and we require about 40 million evaluations on average, which is close enough for this rough analysis. This program is able to solve random positions at a rate of about 0.36 positions per second on an i7-920 CPU running at 2.66GHz. This is about 4 1/2 orders of magnitude faster than Korf's original program on his Sun workstation, and reflects primarily technology improvements in computer hardware. This problem of optimally solving cube positions is interesting in that its solution speed is proportional to both the CPU speed and to the memory available, both of which have been improving according to Moore's law; thus, the overall runtime of this algorithm has been improving, approximately, according to Moore's law squared.

We solved 1,000,000 randomly selected positions in both the half-turn and the quarter-turn metric; the distance results are shown in Table 3. Note that even with one million samples, all the positions were solved in under twenty moves in the half-turn metric and under twenty-four moves in the quarter-turn metric.

3 The Two-Phase Solver

Finding an optimal solution is much harder than finding a near-optimal solution. Kociemba's two-phase algorithm does this latter very effectively, but to understand it we need to know just a tiny bit of group theory.

Let us assume we are exploring the Rubik's Cube group, but we disallow all quarter moves on the faces F, R, B, and L; thus, we only permit 10 of the usual 18 moves. With this subset of moves, we find we cannot attain all positions of the cube; in fact, we can only attain 19,508,428,800 of them. This set of positions forms a subgroup of the larger cube group; we will call this subgroup H. H contains approximately one in every two billion cube positions. We can build two tables, both small enough to fit in memory, which enable us to rapidly find good solutions to arbitrary positions. The first table, called the phase one table, tells for a given arbitrary cube position how many moves it would take to convert that position to one in H. The second table tells for all of the positions in H how far that position is from solved, using only moves in H.

	12h	13h	14h	15h	16h	17h	18h	19h	
15q	1	1	3	2					7
16q		2	18	48	35				103
17q		3	23	143	347	354			870
18q		5	40	305	1,713	4,520	2,034		8,617
19q		1	40	505	5,190	29,711	33,363	474	69,284
20q		2	39	674	9,932	100,164	212,466	7,213	330,490
21q			9	345	7,697	104,052	301,668	16,371	430,142
22q				41	1,533	28,173	120,449	9,720	159,916
23q					1	53	427	90	571
	1	14	172	2,063	26,448	267,027	670,407	33,868	1,000,000

Table 3: Distances of a random set of 1,000,000 positions, in both half-turn metric and quarter-turn metric.

Kociemba's algorithm, given an arbitrary position, uses iterated depth-first search to enumerate the sequences that transform that position to one in H. For each such sequence found, he considers the sum of the length of that sequence and the distance to solved returned from the phase two table; the sum is the length of a solution to the original position starting with the sequence under scrutiny. If this is the best so far, he saves it, and continues the search. In practice this is able to find distance-20 or better solutions to arbitrary positions in, typically, under a millisecond.

The key idea is that the phase one table is a perfect pruning table; for every position, it gives the exact count of moves (not just a lower bound) necessary to transform the position into one in H. Similarly, the phase two table is also exact. This means no time is wasted enumerating incorrect sequences; each sequence is found in constant time, at a rate of millions per second. Only those sequences that lead to H are generated and explored; this is approximately one out of every two billion sequences at each length. Phase two sequence exploration is only done if the new solution is definitely better than the one we have in hand, so the time for that exploration is negligible. The runtime, therefore, is proportional to how many phase one sequences are generated in order to find a distance-20 solution.

Let us consider a search through distance-d phase one solutions. There are approximately $f(d)/(2 \cdot 10^9)$ such sequences for a typical starting position; thus, there are about that many probes to the phase two table, which has the distribution shown in Table 4.

Let us start our consideration at distance nine, which is the first distance that likely contains a phase one solution (for an arbitrary given position). At this distance, the number of canonical sequences is about $2 \cdot 10^{10}$ so there will typically be only a few phase-one solutions, and the expected few probes in the phase two table will probably give us a value around 13, for a total distance of 22. At a phase one distance of 13, there are about $6 \cdot 10^{14}$ canonical solutions, only about 300,000 of which our phase one search will generate (since the others do not leave us with a position in H). Those 300,000

d	count	d	count
0	1	10	116,767,872
1	10	11	552,538,680
2	67	12	2,176,344,160
3	456	13	5,627,785,188
4	3,079	14	7,172,925,794
5	19,948	15	3,608,731,814
6	123,074	16	224,058,996
7	736,850	17	1,575,608
8	4,185,118	18	1,352
9	22,630,733		
			19,508,428,800

Table 4: The phase two distance table distribution for Kociemba's two-phase algorithm.

probes in the phase two table will probably give us at least one that is of distance seven or less, for a total solution length of 20.

This is a rough analysis, and in practice, enough positions require phase one searches of 14, 15, and 16 that the average count of phase one sequences that need to be considered could be substantially higher. This is mitigated with some additional techniques that take into account different orientations of the position. A modern implementation of Kociemba's algorithm on an i7-920 can find distance-20 solutions for about 3,900 random positions per second, requiring an average of only 785 phase one evaluations each. This is more than 10,000 times faster than the optimal solver.

4 A Coset Solver

Kociemba's algorithm shows how effective a bit of group theory can be in practice. Another example of this is the problem of counting the cube positions at a given distance. Jerry Bryan [1] has done this to distance 11 in the half-turn metric and 13 in the quarter-turn metric, but we can extend these results using cosets.

The most straightforward approach to this problem is to use iterated depth-first search, and store all the found positions in a big table. If we explore a sequence that just happens to reach a position we have already seen before, we don't count it twice. But this table rapidly gets too large to fit into memory, and if you spool the table out to disk, the performance drops significantly. What we need is some way to partition the cube space so we can explore it one small section at a time, and add up the results from these individual, smaller explorations to get our final counts.

As a thought experiment, let us imagine that we remove the stickers from all of the edge cubies; only the corners and centers remained stickered. "Solving" this cube is much easier; indeed, there are only 88,179,840 distinct positions, so we can use a tiny, fast lookup table. Indeed, many of the positions are equivalent by symmetry; this cuts the size of the table down to about 1.8 million. In addition, any given position and its inverse have the same distance, so we can reduce this down to about one million distinct corner positions that need to be considered.

With such a pruning table, we can quickly enumerate, for any particular corner position, all the distinct canonical sequences of length d that "solve" that corner position, using iterated depth-first search and using the table of distances to prune ineffective moves. Why do we care about multiple solutions to the same corner position? Because if we apply those solutions to the original, fully stickered cube, we find we are enumerating a set of positions in the full cube group that have a particular corner position, but various edge positions. This is one example of a coset.

The positions of the cube form a group with many subgroups. One of those subgroups is the set of positions with the corners in the solved position, but the edges arranged arbitrarily; this subgroup is of size $12!/2 \cdot 2^{11}$ or 490,497,638,400. For any arbitrary position of the cube, we can apply each of the positions from the subgroup to obtain a new set; this new set is called a coset. For each position in one of these cosets, all the corners are in the same positions, but the edges are moved around. Every coset of a subgroup has the same size, and the number of cosets is just the size of the group divided by the size of the subgroup.

So to enumerate all the cube positions at distance d, we can do the following for each of the approximately million distinct interesting corner positions. First we enumerate all the positions at a distance less than d (using this same technique, recursively). Then, we enumerate all sequences of length d that solve the corner position; for each of those sequences, we apply it to the normally stickered cube, check if it's not in the table containing positions at distance less than d, and if not, add it to the table and increment our counter (being careful to take into account the symmetry of the position). After we have enumerated all such sequences, we add the count to our global count, clear the table, and move on to the next interesting corner position.

d	HTM positions (f(d))	QTM positions
0	1	1
1	18	12
2	243	114
3	3,240	1,068
4	43,239	10,011
5	574,908	93,840
6	7,618,438	878,880
7	100,803,036	8,221,632
8	1,322,343,288	76,843,595
9	17,596,479,795	717,789,576
10	232,248,063,316	6,701,836,858
11	3,063,288,809,012	62,549,615,248
12	40,374,425,656,248	583,570,100,997
13	531,653,418,284,628	5,442,351,625,028
14		50,729,620,202,582
15		472,495,678,811,004

Table 5: The number of Rubik's Cube positions at distance d in both the half-turn metric (HTM) and the quarter-turn metric (QTM).

Using this technique, we break the problem down into about a million smaller problems, each of which fit into memory more easily. The total runtime is directly proportional to the number of canonical sequences of length d, which for the depths explored is also proportional to the result. The results from this exploration are shown in Table 5; the overall runs took about ten days of CPU time on an i7-920 for the last level in both the half-turn metric and the quarter-turn metric. The results reflect about 6.4 million new positions found per second; these are all optimal solves, but not arbitrary positions. Only the last two rows in each column are new results.

5 Another Coset Solver

The above technique partitions the cube group by cosets of the subgroup that fixes corners. For other cube explorations, other subgroups can be more effective. For example, Kociemba's subgroup H can be used.

One reason the subgroup H was so effective in Kociemba's algorithm is that it neatly splits the cube group into two subproblems of roughly equal size; phase one has a size of about 2 billion, and phase two has a size of about 20 billion. Tables describing these subproblems can easily fit into memory for quick access.

Kociemba's algorithm motors through millions of sequences per second, but it discards the vast majority of them in search of short solutions for an arbitrary position. But, if our interest is to optimally solve many distinct (but related) positions at once, as was the case for our corners-fixed solver above, we do not need to discard those solutions; we can retain them as solutions to specific positions we are interested in. Indeed, let us execute Kociemba's algorithm changed in only one major respect: instead of looking up distances in a phase two table, let us just mark off entries in a phase two bitmap indicating that that particular entry has been reached. The full phase two bitmap itself represents a particular coset we are interested in solving, and setting a bit that was previously unset is specifically finding an optimal solution to a position in that coset.

For a typical coset, a search at level d requires consideration of about $f(d)/2 \cdot 10^9$ sequences. My current implementation can explore about 10 million sequences per second on an i7-920. A full exploration out to depth 19 takes about 46 hours but, for most cosets, solves all positions, at a rate of about 120,000 optimal solutions a second. We can do much better than even this.

Most of the sequences will end in one of the ten moves from H (since these are 10 of the 18 normal moves). The bitmap represents a coset of moves all related by moves in H; that is, the position reached by the prefix of the sequence omitting the last move is also in H, and was found by the previous-depth search. If we use two bitmaps, one for depth $d-1$ and another for the current depth d, we can scan the bitmap at depth $d-1$, and for any bits set, extract the relevant position, apply each of the ten moves from H to this position and set the corresponding bit in the bitmap for depth d. Furthermore, we can do this operation on all potential 19,508,428,800 potential positions at $d-1$ with all ten generators of H in about four seconds, using small lookup tables and bitmap tricks. This is about 50 billion group operations per second. We call this a *prepass* since it is typically run before a search pass. This prepass saves us about a factor of two at each search depth, which brings our time per coset down to about 23 hours.

Indeed, we do not need to complete a depth 19 search at all. Instead, we complete a depth 18 search (using the prepass idea above) in about 100 minutes. This will set all but about 600 million bits, on average. Then, we execute the prepass once, marking all positions that are at depth 19 in a move sequence that ends in one of the moves from H. This leaves on average only about 400 positions, positions that could be at distance 19, 20, or more. We then optimally solve these remaining 400 positions by first trying the two-phase solver to find a solution at distance 19 (since we know these positions are distance 19 or deeper), and if that fails, using the optimal solver. On average handling these remaining 400 positions takes another 60 minutes, so overall we can solve all 19,508,428,800 positions in the coset in 160 minutes total, for a rate of about two million optimal solutions per second. This is about six million times faster than solving the positions individually.

We applied this technique to 250 randomly selected H-cosets with a total of nearly five trillion positions. The final distance distribution is shown in Table 6. Of these 250 cosets, only 18 had any distance-20 positions,

d	count
8	30
9	990
10	18,603
11	291,034
12	4,191,486
13	57,843,281
14	779,147,743
15	10,312,034,775
16	131,362,659,765
17	1,315,341,261,754
18	3,261,117,757,529
19	158,131,992,975
20	35
	4,877,107,200,000

Table 6: The distance distribution of the positions optimally solved from 250 random H-cosets.

and there were only 35 such positions. Extrapolating this to the entire cube group, we estimate there are approximately 300,000,000 distance-20 positions. Picking positions at random, a distance-20 position might only be encountered every 140 billion positions.

6 God's Number

God's number is the distance of the farthest position from start, or the diameter of the Cayley graph. Prior to the coset solver we are describing, the best results were a lower bound of 20 (by example position at that distance) and an upper bound of 26 [5].

To find God's number, we do not need to find optimal solutions for all positions, but instead find a distance bound for the coset as a whole. If we do a search to depth 16 (typically taking about 35 seconds), then do four prepasses to extend these solutions to depth 17, 18, 19, and 20, this alone will typically eliminate all but a few positions. Essentially, we are implicitly enumerating all the sequences that use any of the 18 face turns for the first 16 moves, and then any of the 10 generators of H for the last four moves (and of course all prefixes of these sequences). The key here is *implicitly*; we only explicitly enumerate the prefixes up to length 16, and then use the prepass technique to extend our sets by the generators of H for four steps.

Indeed, we don't even need to do a full depth 16 search; once about 240 million bits are set at depth 16, we can terminate that search early and still end up having found a solution to all the positions but a few dozen. Using the two-phase solver will eliminate these positions in well under a second, for a total runtime of twenty seconds. Effectively, we are able to use this technique to find a length-20 or shorter solution for all 19 billion members of an H-coset at a rate of a billion positions per second. This is about 250,000 times faster than the standard two-phase algorithm.

Proving a bound of 20 on a coset provides a nice concise representation of a bound on a large number of positions. Cosets are related to each other by the same generators that form the group itself; applying a single move to all the positions in a coset by left multiplication takes the entire coset into an adjacent coset; the graph of the cosets related by these moves forms a Schreier coset graph. If we prove a bound of 20 on a coset, we implicitly prove a bound of 21 on its neighboring cosets, and 22 on the neighbors of those cosets, and so on. The coset graph has two billion elements (about 138 million when reduced by symmetry) so it easily fits into memory. By computing bounds of 20 on scattered cosets, I have been able to lower the bound on God's number from the 26 cited above all the way down to 22 [7]. Computer time for this search was generously donated by Sony Pictures Imageworks, using the same computers that rendered *Spiderman* and *Surf's Up*.

7 The Search for 20's and 21's

Since distance-20 positions are so rare, one interesting challenge is to find as many as you can. Silviu Radu and Herbert Kociemba did this successfully in 2006; they calculated optimal solutions to all 164,604,041,664 symmetrical positions of the cube [6]. They hypothesized that the density of distance-20 positions would be higher in the set of symmetrical positions, and they were able to identify all of the 1,091,994 positions that are at distance 20 and exhibit any symmetry.

But most distance-20 positions are not symmetrical. Is there a way we can identify these positions more efficiently than searching random H-cosets one after the other, picking up on average less than one distance-20 positions per coset solved? Yes there is. A dynamic programming approach that evaluates the count of sequences of length d that result in positions from each H-coset requires a fair amount of memory to execute, but is otherwise straightforward. This computation allows us to identify those cosets of H that have the fewest length-19 sequences. If we assume that such cosets are likely to have the greatest count of distance-20 positions, we can solve only those cosets and expect to get a much higher success rate.

We have identified and solved 346 of the cosets that have the fewest distance-19 sequences. From these cosets, we have found 108,316 distance-20 positions, each of which expands by rotation and inversion to up to 95 additional positions. As of this writing, we have a grand total of 10,592,538 distance-20 positions, which is probably about three or four percent of the total.

If there are any distance-21 positions, they might more likely occur in one of the cosets with the greatest number of distance-20 positions (since any coset with a distance-21 position must, by necessity, contain at least 10 distance-20 positions.) But to date we have not found any distance-21 positions.

Current machines can prove a bound of 20 for an arbitrary H-coset in about 20 seconds. Proving a bound of 21 for God's Number would require about 7.5 million cosets to be bounded at 20; proving a strict bound of 20 for God's number would require about 56 million cosets to be bounded at 20 (or, a position at distance 21 might be found). Both of these are within the capabilities of a large cluster of computers or a distributed attack; it should be at most a couple of years before God's number is finally proved.

8 Acknowledgements

Many of the ideas used in implementing these programs originated in discussions with Silviu Radu and Herbert Kociemba; I am grateful for their time and energy. I am also thankful for Martin Gardner, who sparked my interest in recreational math.

References

[1] Bryan, Jerry. "Distance from Start, Standard 3x3x3 Rubik's Cube." `http://home.comcast.net/~c24m48/math/rubikresults.html`

[2] Joyner, David. *Adventures in Group Theory: Rubik's Cube, Merlin's Magic & Other Mathematical Toys.* Baltimore: The John Hopkins University Press, 2008.

[3] Kociemba, Herbert. "Close to God's Algorithm" *Cubism For Fun* 28 (April 1992) pp. 10-13.

[4] Korf, Richard E. "Finding Optimal Solutions to Rubik's Cube Using Pattern Databases." *Proceedings of the Workshop on Computer Games (W31) at IJCAI-97.*

[5] Kunkle, D.; Cooperman, G. "Twenty-six Moves Suffice for Rubik's Cube." *Proceedings of the International Symposium on Symbolic and Algebraic Computation (ISSAC '07)*, ACM Press.

[6] Radu, Silviu. "How to Compute Optimal Solutions for All 164,604,041,664 Symmetric Positions of Rubik's Cube." June, 2006. `http://cubezzz.homelinux.org/drupal/?q=node/view/63`

[7] Rokicki, Tomas G. "Twenty-two Moves Suffice for Rubik's Cube." *The Mathematical Intelligencer* 32 (1) (March 2010) pp. 33-40.

[8] Scherphuis, Jaap. "Computer Puzzling." `http://www.jaapsch.net/puzzles/compcube.htm`