Twenty-Two Moves Suffice for Rubik's Cube

Tomas Rokicki

June 20, 2009

The Rubik's Cube is a simple, inexpensive puzzle with only a handful of moving parts, yet some of its simplest properties remain unknown more than thirty years after its introduction. One of the most fundamental questions remains unsolved: how many moves are required to solve it in the worst case? We consider a single move to be a turn of any face, 90 degrees or 180 degrees in any direction (the 'face turn metric'). In this metric, there are more than 36,000 distinct positions known that require at least twenty moves to solve[9]. No positions are yet known that require twenty-one moves. Yet, the best theoretical approaches and computer searches to date have only been able to prove there are no positions that require more than twenty-six moves[4]; this gap is surprisingly large.

In this paper, we prove that all positions can be solved in twenty-two or fewer moves. We prove this new result by separating the cube space into two billion sets, each with 20 billion elements. We then divide our attention between finding an upper bound on the distance of positions in specific sets, and combining those results to calculate an upper bound on the full cube space.

The new contributions of this paper are:

1. We extend Kociemba's near-optimal solving algorithm to consider six transformations of a particular position simultaneously, so it finds near-optimal positions more quickly.

2. We convert his solving algorithm into a set solver that solves billions of positions at a time at a rate of more than 200 million positions a second.

3. We show how to eliminate a large number of the sets from consideration, because the positions in them only occur in conjunction with positions from other sets.

4. We combine the three contributions above with some simple greedy algorithms to pick sets to solve, and with a huge amount of computer power donated by Sony Pictures Imageworks, we actually run the sets, combine the results, and prove that every position in the cube can be solved in 22 moves or less.

Colors, Moves, and the Size of Cube Space

The Rubik's cube appears as a stack of 27 smaller cubes (cubies), with each visible face of the cubies colored one of six colors. Of these 27 cubies, seven form a fixed frame around which the other twenty move. The seven that form the fixed frame are the center cubies on each face and the central cubie.

Each move on the cube consists of grabbing the nine cubies that form a full face of the larger cube, and rotating them as a group 90 or 180 degrees around a central axis shared by the main cube and the nine cubies. Each move maintains the set of fully-visible cubie faces. The eight corner cubies each always have the same set of three faces visible, and the twelve edge cubies each always have the same set of two faces visible. We will frequently use the term 'corner' to mean 'corner cubie', and 'edge' to mean 'edge cubie'.

In the solved position, each face of the main cube has a single color. By convention, we associate these colors with their orientation on the solved cube: U(p), F(ront), R(ight), D(own), B(ack), and L(eft). Each move that uses a 90 degree clockwise twist is denoted by writing the face with no suffix; each move that uses a 90 degree counterclockwise twist is specified with the face followed by a prime symbol ($'$), and each move that uses a 180 degree twist is specified with the face followed by the digit 2. So a clockwise quarter turn of the right face is represented by R, and the move sequence R2L2U2D2F2B2 generates a pretty pattern known as *Pons Asinorum*. We write the set of all moves, containing the eighteen combinations of faces and twists, as S.

The way the moves in S can combine to generate different positions of the cube is not obvious, but is well known[1]; we state the relevant results here. The

corner cubies may be permuted arbitrarily, or the edge cubies arbitrarily, but not both at the same time; the parity of the two permutations must match. This contributes a factor of 12!8!/2 toward the total number of reachable positions.

Every corner cubie has exactly one face with either the U or D color. We define the default orientation for the corner cubies to be that where the U or D face is on the whole-cube u or d face; the corner cubies may also be twisted 120 degrees clockwise or counterclockwise with respect to this default orientation (looking toward the center of the cube). Note that these orientations for each cubie are preserved by the moves U, D, R2, L2, F2, B2, but not by the moves R, L, F, or B. This corner cubie orientation is fully arbitrary, except that the sum of all the twists for all the corner cubies must be a multiple of 360 degrees. These corner orientations contribute an additional $3^8/3$ factor toward the total number of reachable positions.

We define the default edge orientation to be that orientation in the solved state of the cube that is preserved by the moves U, D, R, L, F2, B2 (but changed by F and B). Each edge is either flipped from this orientation or not; the count of flipped edges must be even. These edge orientations contribute an additional $2^{12}/2$ factor toward the total number of reachable positions.

The total number of reachable positions, and thus the size of the cube group, is the product of these factors, which is about 4.33×10^{19}. We call the set of reachable positions G. For each of these positions, an infinite number of move sequences obtain that position. We define $d(p)$, the *distance* of a position p, to be the shortest length of any move sequence that obtains that position. We define the distance of a set of positions to be the maximum of the distances of all the positions in that set.

As a convention, we will denote the successive application of two move sequences by concatenation. We will also denote the application of a sequence to a position, or set of positions, by concatenation of the position and the sequence.

Symmetry

The Rubik's cube is highly symmetrical. There is no distinction among the faces except for the color; if we were to toss the cube in the air and catch it, the cube itself remains the same; only the color corresponding to the u face, the r face, and so on changes. Indeed, by tossing the cube, catching it, and noting the colors on the various faces in the new orientation, we can enumerate a total of 24 different ways we can orient the cube, each with a distinct mapping of colors to U, F, R, D, B, and L faces. Specifically, there are six different colors the up face can have, and for each of those six colors, there are four colors possible for the front face. These two face colors fully define the orientation of the normal physical cube.

If we peer in a mirror while performing this experiment, we notice that our alter-ego holds a cube with mirror-reversed orientations; these mirror-reversed orientations present an additional 24 possible mappings from colors to oriented faces. We further notice that whenever we do a clockwise move, our alter ego

does a counterclockwise move.

If we choose a canonical color representation, then each of these 48 orientations is a permutation of the cube colors. We call this set of color permutations M. If a particular cube position p is obtained by a move sequence s, we can obtain fully corresponding positions by applying one of the 48 color permutations (say, m), performing the sequence s, and then applying the inverse permutation of m. The resulting position shares many properties with the original one (especially, for us, distance). If we repeat this operation for all 48 permutations in M, we will obtain 48 positions. These positions are not always unique, but for the vast majority of cube positions they will be. Using this form of symmetry, we can reduce many explorations of the cube space by a factor of 48.

Each cube position has a single specific inverse position. If a position is reached by a move sequence s, then the inverse position is reached by inverse move sequence s'. To invert a move sequence, you reverse it and invert each move; the face remains the same, but clockwise becomes counterclockwise and vice versa. The set of symmetrical positions of the inverse of position p is the same as the inverses of the symmetrical positions of p. Some properties of a position are shared by its inverse position (specifically, distance).

We can partition the cube space into symmetry-plus-inverse reduced sets by combining each position with its symmetrical positions and their inverses; there are only 4.51×10^{17} such sets.

Calculating the Diameter

We are primarily interested in finding the maximum of the distance for all positions; this is known as the diameter of the group. For context, we review previous techniques for solving the cube using a computer, since our technique is derived from these.

Speedsolvers, cube aficionados who compete in how fast they can solve the cube and other related puzzles, have a wide variety of manual algorithms, from very simple beginner's methods to highly sophisticated methods that require the memorization of dozens of move sequences. Any of these algorithms are straightforward to implement on the computer, but the best of these tend to require many more moves than the actual distance of the position, so these techniques are, in general, useless in calculating the diameter.

Simple approaches to optimally solving a single position fail because the size of cube space is so large. A simple breadth-first search exhausts both memory and CPU time. Iterative deepening, which uses depth-first search limited by a given maximum depth that increases from 0 until the first solution is find, solves the memory exhaustion problem but still requires an impractical amount of CPU time.

A more practical algorithm is to compute all positions that are within some small distance of solved (say, seven moves, totalling 109,043,123 positions[10]), and store these positions and their distances in memory. Then, iterative deepening can be used from the position to be solved, at each node examining the

hash table to obtain a lower bound on the remaining distance and terminating that search branch if the sum of that bound and the current depth is greater than the current maximum depth. Various refinements are possible, such as only including one representative of the set of symmetrically equivalent positions in the hash table, or using a distance table of a subgroup of the cube rather than just the close positions, or only storing the distance mod 3 rather than the full distance. The first such program was written in 1997[5] and required several days per position on average, but a recent version by Kociemba using eight threads on an i7 920 processor can find about 300 optimal solutions an hour. If we were to use such a program to solve the reduced set of 4.51×10^{17} positions, one at a time, with today's hardware, we would require more than one million computers for more than one hundred thousand years. No better algorithm to optimally solve a single position is known.

It is not strictly necessary to optimally solve every position to compute the diameter. We know that some positions require at least twenty moves. The first such position found is called *superflip*; it has every cubie in the correct place, all corners correctly oriented, and all edges flipped[8]. Because we have a lower bound on the diameter, we need not optimally solve each position; once we find a solution of length twenty or less, we can move on to the next position. Herbert Kociemba devised an algorithm to quickly find reasonably short but not necessarily optimal solutions to arbitrary positions. That program (slightly improved as we shall describe) can find move sequences of length twenty or less at a rate of about 240 positions per second (subject to the condition that there is such a sequence; no exceptions have been found yet). Even with this kind of speed, proving all 4.51×10^{17} positions would require more than seven thousand computers for more than seven thousand years.

Rather than using a tremendous amount of CPU time, we can instead use a large amount of memory. If we have enough memory or disk space to store two bits for each of the 4.51×10^{17} positions, we can perform a breadth-first search; some clever bit twiddling and some nice fast multicore processors should allow us to extend this table at a rate of billions of positions a second. Unfortunately, this approach would require over one hundred petabytes of memory.

All hope is not lost. Technology marches onward; when we get to the point we can solve a billion positions a second, we will need only four computers for four years to finish the proof. In the meantime, we can come up with better techniques to refine the upper bound, and improve our techniques.

Kociemba's Algorithm

Several techniques have been used to find an upper bound on the diameter of the cube group. Thistlethwaite gave a four-stage algorithm that requires a maximum of 52 moves. Herbert Kociemba improved this to an algorithm that requires a maximum of 29 moves (as shown by Michael Reid[7]). Our work is based on Kociemba's algorithm, so we will describe it a bit further here. Kociemba himself has a much more detailed explanation on his web site[3]. In

2006, Silviu Radu reduced the upper bound to 27[6], and in 2007 Kunkle and Cooperman reduced it to 26[4].

Kociemba's algorithm identifies a subset of 20 billion positions, called H. Reid showed that every position in this subset is solvable in at most 18 moves, and further that every cube position is at most 12 moves from this subset. Phase one finds a move sequence that takes an arbitrary cube position to some position in the subset H, and phase two finds a move sequence that takes this new position to the fully solved state.

To describe this subset, we will introduce some new terminology. A cubie *belongs* in a particular place, if it is in that place in the solved cube. Thus, all cubies that have some face colored u belong in one of the top nine cubies. The middle layer consists of the nine cubies between the top and bottom layers; only four of these cubies (edges all) move.

The subset H is composed of all positions that have the following characteristics:

1. All corners and edges are in their default orientation (as defined earlier).
2. The edge cubies that belong in the middle layer are located in the middle layer.

The number of positions for which these conditions hold are the permissible permutations of the corners, the top and bottom edges, and the middle edges, with the condition that the parity between the edge permutation and the corner permutation must match. This is thus 8!8!4!/2 or 19.5 billion positions.

These characteristics are preserved by the moves U, U2, U′, D, D2, D′, R2, L2, F2, B2, which we call the set A. Further, these moves suffice to transform every position in H to the solved state. (This is a nontrivial result, but it can easily be shown by brute force enumeration.) For more than 95% of the positions in H, the shortest move sequence consisting only of moves from A is the same length as the shortest move sequence consisting only of moves from S, as shown in Table 1. Further, the worst case is 18 in both cases.

Fitting a distance table for all 20 billion positions of H into memory may seem challenging, but there are a few tricks we can use. Because the defining characteristics of this set treat the U and D faces differently than the L, R, F, and B faces, all 48 symmetries of the cube cannot be used; however, 16 can be used; we need only store one entry per equivalence class. Further, instead of storing the distance, which is an integer in $[0 \ldots 18]$ and would require more than four bits each entry, we can store only the distance mod 3, requiring only two bits each entry. This can be achieved by only performing lookups for positions that are adjacent to a position at a known depth. By maintaining a current position and a current distance, and updating the distance as we perform each move, the distance mod 3 of the new position gives us enough information to know whether that position has a distance less than, equal to, or greater than that of the previous position.

The remaining problem is how we can transform an arbitrary cube position into a position in H in 12 or fewer moves. To illustrate how this can be done,

d	moves in S	moves in A
0	1	1
1	10	10
2	67	67
3	456	456
4	3,079	3,079
5	20,076	19,948
6	125,218	123,074
7	756,092	736,850
8	4,331,124	4,185,118
9	23,639,531	22,630,733
10	122,749,840	116,767,872
11	582,017,108	552,538,680
12	2,278,215,506	2,176,344,160
13	5,790,841,966	5,627,785,188
14	7,240,785,011	7,172,925,794
15	3,319,565,322	3,608,731,814
16	145,107,245	224,058,996
17	271,112	1,575,608
18	36	1,352
	19,508,428,800	19,508,428,800

Table 1: The number of positions in H at a given distance using moves from S and moves from A; the numbers are strikingly similar.

we describe a way to relabel the cube so that all positions in H have the same appearance, and all positions not in H have a different appearance.

Consider an arbitrary position p. To be in H, the permutations of the corners are irrelevant; only the orientation matters. To represent this, we remove all colored stickers from the corners, replacing the stickers colored U or D with U and leaving the other faces, say, the underlying black plastic. (To make it easy to follow, we also replace the D sticker in the center of the bottom face with U.) All corner cubies are now interchangeable, but we have sufficient information to note the orientation of the corners.

The permutation of the middle edges does not matter either, but they must lie in the middle layer and be oriented correctly. We thus remove the colored stickers from four edge cubies that belong in the middle layer, replacing the F and B colors with F and leaving the L and R colors as black. (We also replace the B center sticker with F for convenience.)

The permutations of the top and bottom edges also does not matter; for these we do the same color change we did for the corners (U and D get turned into U, and the other four colors get removed).

With this transformation, all positions in H get turned into the same solved cube: eight corners, each with a U sticker on either the up or down face; four middle edges, each with a F sticker on either the front or back face; eight top/bottom edges, each with a U sticker on the top or bottom face. Every position not in H has a different appearance.

This relabeled puzzle has a much smaller state space than the full cube space. Specifically, the space consists of $3^8/3$ corner orientations multiplied by $2^{12}/2$ edge orientations multiplied by $\binom{12}{4}$ ways to distribute four middle edges among twelve edge positions, for a total of 2.22×10^9 positions. We call this set of positions R. With 16 ways to reduce this by symmetry and using only two bits per state, a full distance table is easy to fit in memory, and the full state space can be explored easily. We shall call this relabeling process r; it takes a position in G and transforms it into a position in R.

Kociemba's algorithm, then, is to take the original position, call it a, compute $r(a)$, the relabeling; solve the relabeled puzzle with some sequence $b \in S^*$, apply those moves to an original cube yielding ab which lies in H, and then finish the solution with another sequence $c \in A^*$ such that abc is the solved cube. The final solution sequence is bc.

Kociemba's algorithm splits the problem into two roughly equal subproblems, each of which is easy to exhaustively explore, using a lookup table that fits in memory, yielding a fairly good solution to the much larger problem. This algorithm can find a solution of distance 29 or less almost instantaneously (in well under a millisecond). This defines an upper bound on the worst-case position distance.

Kociemba extended this algorithm for another purpose: to quickly find near-optimal solutions for a given position. He proposed finding many phase one solutions, starting with the shortest and increasing in length, and for each finding the shortest phase 2 solution. By considering dozens, thousands, or even millions of such sequences, he has found that in practice nearly optimal solu-

tions are found very quickly. Given an input which is the initial cube position denoted by a, his algorithm is given as Algorithm 1. The algorithm can either run to completion, or it can be terminated by the user or when a solution of a desired length is attained.

Algorithm 1 Kociemba's Algorithm.

```
1: d ← 0
2: l ← ∞
3: while d < b do
4:    for b ∈ S^d, r(ab) = e do
5:       if d + d_2(ab) < l then
6:          Solve phase two; report new better solution
7:          l = d + d_2(ab)
8:       end if
9:    end for
10:   d ← d + 1
11: end while
```

In Kociemba's algorithm, d_2 is a table lookup that takes a position in H and returns the distance to the identity element (ϵ) using moves in A. (Kociemba actually uses a smaller, faster table that gives a bound on this value; see [3] for details.) The for loop is implemented by a depth-first recursive routine that maintains ab incrementally and has a number of further refinements, such as not permitting b to end in a move in A. The phase two solution process is omitted both because it is straightforward and because it takes much less time than enumerating phase one solutions.

This algorithm is extremely effective. Some reasons are:

1. Phase one solutions are found very fast, and mostly access the portions of the phase one lookup table near the solved position; this locality enhances the utility of caches significantly.

2. When searching for a phase two solution, almost always the very first lookup shows that the distance to the solved position would make the total solution longer than the best found so far; thus, almost all phase one solutions are rejected with a single lookup in the phase two table.

3. Kociemba has found that in practice the algorithm runs considerably faster if he does not consider phase one solutions that contain a strict prefix that is also a phase one solution. This is motivated by the fact that we had already explored that prefix earlier (since we consider phase one solutions by increasing length).

4. The last move at the end of phase one is always a quarter turn of F, B, R, or L; the inverse move is also a solution of phase one, so candidate solutions are always found in pairs at the leaves of the phase one search tree.

5. There are a number of optimizations that can be performed for the phase one search when the distance to H is small, such as storing specifically which moves decrease the distance from that point.

Kociemba's algorithm can be run as described above, or it can be run in triple-axis mode. Note how the algorithm treats the u and d faces differently than the other four. Instead of just exploring a single given position a, in triple-axis mode we explore three rotated positions, one with the cube rotated such that the r and l faces correspond to u and d, one such that the b and f faces correspond to u and d, and the original unrotated position. We try each rotation for a given phase one depth before moving on to the next phase one depth. Our tests show that this finds smaller positions much faster than the standard single-axis mode; when trying to find solutions of length 20 or less, this works approximately six times faster on average than single-axis search.

We have taken this idea one step further; we also consider the inverse position in three orientations for a new six-axis mode. We find this gives on average a further factor of two speed increase when trying to find positions of twenty moves or less.

Our Set Solver

Reid showed a bound of 30 by proving it takes no more than 12 moves to bring an arbitrary cube position to the H set (by solving the restickered cube), and then showing that every cube position in H can be solved in 18 moves. (He then reduced that to 29 with a clever insight we omit for brevity[7].) Our proof of 22 is similar, but instead of using just the H set, we use a union of over a million sets all related to H.

Consider how Kociemba's solver solves an arbitrary position to find a near-optimal solution. It first brings the position (a) into H, by solving the restickered puzzle using some sequence of moves (b). It applies that sequence of moves to the original cube, then looks up how far that position is from solved using a sequence c containing only moves from A (those moves that stay within H), and determines if the total sequence is better than the best known. It then finds another way to bring the position into H, and checks how close it is to solved at that point. It does this dozens, or hundreds, or thousands, millions, or even billions of times, each time checking for a shorter solution.

We turn this technique inside out. Each sequence b that solves the restickered position $r(a)$ is a solution to some full cube position that has the same restickering as the given input position; so is each sequence bc where $c \in A^*$. Rather than throwing most of these solutions away, we keep track of what full cube position each bc sequence solves, marking them off in a table, until we've found some solution for every position that has the same restickering as the original position. Where Kociemba's algorithm searches for b and c such that $abc = \epsilon$, we instead search for b and c such that $r(abc) = r(\epsilon)$. This way we find optimal solutions to 20 billion positions at a time. We are careful to do this in

order of increasing length of bc, so that every time we find a bc that leads to a position we haven't seen before, we know we have a optimal solution to that position.

Since $abc \in H$, we can implement this by simply replacing the lookup table d_2 on H with a bitmap on H indicating whether the position abc has already been seen. When every bit in the table has been set, we know we have found an optimal solution to every position in Ha.

For our purposes, we do not need an optimal solution to every position; all we need is a bound on the distance of the entire set. Just as in Kociemba's solver, the deeper the phase one search is allowed to go, the longer the program takes; yet, a shallow phase one search will still find a solution to every position in the set. We use a tunable parameter m that limits the depth of the phase one search to trade off execution time against the optimality of the solutions found.

The main input to our set solver is a sequence $a \in S^*$, which takes the solved cube into some position; the set that will be solved is Ha. Another input is the maximum depth m to run the phase one search; we have found the value $m = 16$ is usually sufficient to prove an upper bound for the distance of the set to be 20. To find the exact distance, m should be set to ∞. Our algorithm is given as Algorithm 2. At the end of each iteration of the main loop, f contains

Algorithm 2 Set Solver

```
 1: f ← ∅
 2: d ← 0
 3: loop
 4:    f ← f ∪ fA {—prepass}
 5:    if f = H then
 6:       return d
 7:    end if
 8:    if d ≤ m then
 9:       for b ∈ S^d, r(ab) = e do {–search}
10:          f ← f ∪ ab
11:       end for
12:    end if
13:    if f = H then
14:       return d
15:    end if
16:    d ← d + 1
17: end loop
```

all positions abc such that $|bc| < d$. The prepass (line 4), corresponding to Kociemba's phase two, extends the set f by sequences ending with a move from A; the search (lines 9–11), corresponding to Kociemba's phase one, extends the set f with move sequences not ending in a move from A.

Unlike Kociemba's algorithm, we do permit our phase one search to enter and

then leave the H group; we do this in order to compute the precise set bound. We have not yet explored the performance impact of this on our running time.

The set f is represented by a bitmap, one bit per position. For the prepass (line 4), we need to have both a source and destination set, so we need to have two of these bitmaps in memory at once. Our memory requirements are completely dominated by these bitmaps.

The indexing of f is done by splitting the cube position into independent coordinates, representing the permutation of the corners, the permutation of the up/down edges, and finally the permutation of the middle edges.

The time spent in the code is split between the prepass and the search phases. The prepass is a simple scan over the entire f set, multiplying by the ten moves in A; this can be done efficiently by handling the coordinates from most significant to least significant in a recursive algorithm so that the inner loop only need deal with the permutation of the middle edges, and the more expensive corner coordinate computation is performed early in the recursion and thus substantially fewer times. In the innermost loop, we perform the move and bitmap update on all possible middle edge permutations using a lookup table and some bit-parallel logic operations.

The time in the search phase (lines 9–11) is very small for low d, because there are few sequences s that satisfy the conditions, but as d grows, so does the time for the search phase, exponentially. Typically a search at level $d+1$ will require ten times as much time as a search at level d. By limiting m to 16 in the typical case, we limit the total time in the search phase, and the whole program runs fast. For values of m of 17 or higher, the search phase will dominate the total runtime.

The Set Graph

The set R of relabeled positions of G has about two billion elements. Consider a position $a \in R$; we can define the parent set of a to be all elements $g \in G$ such that $r(g) = a$. Let us pick a single one of the elements i in the parent set of a; the entire parent set can be represented by Hi. Each such set has precisely the same number of elements, about 20 billion; every pair of sets is either identical or disjoint; and the union of all of the sets is G, the full cube space. (This can be shown with elementary group theory because H is a subgroup of G and each set Hi is a coset.)

These sets are all related by the full set of cube moves (S). Consider a cube position a and its set Ha. The set Hab for $b \in S$ is adjacent to the set Ha. We can consider R as a graph, where the vertices are the sets represented by the positions of R, and the edges are moves in S. Clearly for any given position $|d(ab) - d(a)| \leq 1$, and therefore the same is true for sets as a whole: $|d(Hab) - d(Ha)| \leq 1$. If we have shown that $d(Ha) \leq c$ for some value of c, we have also shown that $d(Has) \leq c + |s|$ where s is a sequence of moves of length s. This allows us to find an upper bound for one set, and use it to infer constraints on upper bounds of neighboring sets in the graph of R.

The relabeled puzzle shows 16-way symmetry, so there are only about 139 million relabeled positions when reduced by this symmetry. This reduced graph easily fits into memory, and operations on this graph can be performed reasonably quickly. For each vertex, we maintain a value which is the least upper bound we have proved to date. These values are initialized to 30, since we know every position and thus every set has a distance of no more than that. As we solve new sets, we update the value for the vertex associated with that set, and update adjacent vertices recursively with the new upper bound implied by this value.

Improving the Bound

Some sets we solve have relatively few positions in the furthest distance. Since for lower values of m our set solver only gives us an upper bound on the set distance, in many cases the true distance of all these positions is less than the calculated upper bound. By solving these explicitly using a single position cube solver, and proving they do not require as many moves as our set solver found, we can frequently reduce our bound on the distance for the set by 1. To facilitate this, if the count of unsolved positions in one of the sets falls below 65,536 at the top of the loop, we print each of these positions to a log file.

To solve these positions, we first use our six-axis implementation of Kociemba's solution algorithm. Since the solution distance we seek is almost always 19 or 20, this algorithm finds solutions very quickly, usually in a fraction of a second. For those positions that resist Kociemba's solver, we solve them using our optimal solver.

Reducing Memory Use

During the prepass, we compute $f \leftarrow f \cup fA$, where both the original and the new f is represented by a bitmap with one bit per position. Since the set size is almost 20 billion, this would normally require 2.4GB per set for a total of about 4.8GB. This is more memory than can be allocated on 32-bit operating systems, and is more memory than can be added to many modern computers. We can reduce the memory requirements substantially by only keeping a portion of the source and destination bitmaps in memory at any given time.

We do this by splitting the bitmap index into two parts, one calculated from the corner permutation and the other calculated by the edge permutation. We then split each bitmap into pieces, one piece per corner permutation; there are 8! such pieces. For every source bitmap part, corresponding to some source corner permutation, and every single move from A, there is only a single destination bitmap part, and this is found by performing the move from A on the corner permutation corresponding to the source bitmap part. As we proceed through the prepass, we consider each corner permutation in turn, allocating destination bitmap parts only as we need them, and freeing source bitmap parts as soon as

we are finished with them. With a small program that performs a randomized search guided by some ad-hoc heuristics, we have found a good ordering of the corner permuations such that the maxiumum amount of memory required at any one time during the prepass is only 3.2GB, which enables the set solver to be run on machines with only 4GB of physical memory.

Choosing Sets to Solve

This work grew out of a search for distance 21 positions[9] that involved solving a number of these sets exactly. We thus started this work with a few thousand sets already solved; we used those as our base set. At every point during this exploration we maintained the symmetry-reduced graph R on disk annotated with the best upper bound we had proven for each corresponding set. To select a new set to solve, we used a simple greedy strategy. We chose a vertex that, when we pushed its bound down to 20, and propagated its implications over the graph R, would reduce the maximum number of vertices from above 22 to 22 or less; we call this value the 'impact' of the vertex. We evaluated the impact of a few hundred vertices, and chose the one with the greatest impact to solve. Once we had selected a vertex, we added it to the list of sets to solve, updated the relevant vertices on the in-memory copy of the graph (not the persistent one on disk), and repeated this process to select another vertex.

We typically generated lists of a few thousand sets to solve in this manner. Since some of the sets actually were found to have a bound of 19 or even 18, and this changed the graph in different ways than our above algorithm assumed, we generated a brand new list of vertices to solve every few days based on the updated R graph.

Results

Approximately 6,000 sets, sufficient to prove an upper bound of 25, were all computed on home machines between October 2007 and March 2008. When those results were announced, we were contacted by John Welborn of Sony Pictures Imageworks, offering some idle computer time on a large render farm to push the computation further. Using these machines, we were quickly able to solve sets to prove bounds of 24 (26,380 sets requiring approximately one core year) and 23 (180,090 sets requiring approximately seven core years). With some additional time, we managed to finally prove a bound of 22 (1,265,326 sets requiring fifty core years). The sets were run on a heterogeneous collection of machines, some multi-core, some single-core, some older and slower and some more modern. Since these sets were run, the set solver has seen significant performance improvement and processor technology has advanced; on a single Intel i7 920 processor we believe we can reproduce all these results in only sixteen core years (four CPU years on this processor).

All of these sets were shown to have a distance of 20 or less, using searches

through depth $d = 16$ or depth $d = 15$. Approximately 4.2% were shown to have a distance of 19.

We continue to execute sets, and we are making progress toward proving a bound of 21. Once this is done, we believe that with only a few more core centuries, we can show a new bound of 20 on the diameter of the cube group.

The Quarter-Turn Metric

These general ideas apply nearly equally well to the quarter-turn metric, where each 180 degree twist requires two (quarter) moves. The fundamental algorithms remain the same, except each 180 degree move (half move) has weight two. Implementing this in our set solver did introduce one complication: the prepass operation $f \leftarrow f \cup fA$ does not properly handle the half moves. This problem can be solved by considering permutation parity.

Every permutation is either of odd or even parity; it is of odd parity if an odd number of element swaps is needed to restore the permutation to the identity, and even if an even number of swaps is needed. Every quarter move performs a permutation of odd parity on the corners and also on the edges; every half move performs a permutation of even parity. Thus, the parity of the corner permutation always matches the parity of the edge permutation, and this is always equal to the parity of the number of quarter turns performed from the solved state.

The positions in the set H are evenly divided between those of odd parity (H_1) and those of even parity (H_0). Similarly, we can consider our intermediate set of positions f to be split into odd (f_1) and even (f_0) parity, and the moves in A to be split into quarter moves (A_1) and half moves (A_0). At step d in the quarter turn metric, we can only find positions whose parity is the same as the parity of d. Thus, before the prepass, the half of f that has the opposite parity to d represents positions at distance $d - 1$ or less, but the half that has the same parity as d represents positions at $d - 2$ or less. To reflect newly reachable positions at distance d, we can apply the half moves (A_0) to the half with the same parity as d, and apply the quarter moves (A_1) to the half with the other parity. Line four in Algorithm 2 must be replaced by the code shown in Algorithm 3.

Algorithm 3 Prepass for the Quarter-Turn Metric

if $odd(d)$ **then**
  $f_1 \leftarrow f_1 \cup f_1 A_0 \cup f_0 A_1$
else
  $f_0 \leftarrow f_0 \cup f_0 A_0 \cup f_1 A_1$
end if

The distances in the phase one pruning table (d_2) are of course different in the quarter-turn metric, and in general Kociemba's algorithm is somewhat less effective; the solutions found quickly tend to be somewhat further from optimal

than with the half-turn metric. Similarly, the quarter-turn metric version of our set solver requires searching deeper in phase one. Specifically, for almost all sets, searching through $d = 19$, taking about five minutes on our i7 920, proves almost all positions in that set can be solved in 25 or fewer moves. Typically only one or two positions are left, and these are very quickly solved by Kociemba's algorithm in 24 moves, leaving a bound of 25 for the whole set.

In the quarter turn metric, there is only one position known that has a distance of 26; this position was found by Michael Reid. We solved 24,759 sets in the quarter turn metric to a depth of 19; each of these was found to have a bound of 25 or less, except for the single set which included Reid's position. These sets sufficed to show that there is no cube position that requires 30 or more moves, lowering the upper bound in the quarter turn metric from 34[6] to 29.

Acknowledgements

This work was greatly helped by discussions with Silviu Radu; it was he who directed us to the subgroup (called H here) used by Kociemba. We are also grateful to Herbert Kociemba for both his original 1992 algorithm (and its implementation in Cube Explorer) and for ongoing email discussions that have led to significant simplifications and performance improvements in the set solver. Many thanks also to John Welborn and Sony Pictures Imageworks, who donated massive computer time toward this project. The list of cosets and our calculated distance bounds are available at `http://johnwelborn.com/rubik22/`.

References

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

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

[3] Kociemba, Herbert. Cube Explorer (Windows program). `http://kociemba.org/cube.htm`

[4] 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.

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

[6] Radu, Silviu. "New Upper Bounds on Rubik's cube." http://www.risc.uni-linz.ac.at/publications/download/risc_3122/uppernew3.ps

[7] Reid, Michael. "New upper bounds." Cube lovers, 7 January 1995. http://www.math.ucf.edu/~reid/Rubik/Cubelovers/

[8] Reid, Michael. "Superflip requires 20 face turns." Cube lovers, 18 January 1995. http://www.math.ucf.edu/~reid/Rubik/Cubelovers/

[9] Rokicki, Tomas. "In search of: 21f*s and 20f*s; a four month odyssey." 7 May 2006. http://cubezzz.homelinux.org/drupal/?q=node/view/56

[10] Sloane, N. J. A. "Online Encyclopedia of Integer Sequences," Sequence A080601 (sum of the first seven terms).