SIAM Review
Vol. 56, No. 4, pp. 645–670

The Diameter of the Rubik's Cube Group Is Twenty*

Tomas Rokicki†
Herbert Kociemba‡
Morley Davidson§
John Dethridge¶

Abstract. We give an expository account of our computational proof that every position of the Rubik's Cube can be solved in 20 moves or fewer, where a move is defined as any twist of any face. The roughly 4.3×10^{19} positions are partitioned into about two billion cosets of a specially chosen subgroup, and the count of cosets required to be treated is reduced by considering symmetry. The reduced space is searched with a program capable of solving one billion positions per second, using about one billion seconds of CPU time donated by Google. As a byproduct of determining that the diameter is 20, we also find the exact count of cube positions at distance 15.

Key words. group theory, algorithm performance, Rubik's Cube

AMS subject classifications. 20-04, 05C12, 20B40

DOI. 10.1137/140973499

1. Introduction. Few physical objects embody an abstract and complex mathematical entity as faithfully as, or with comparable popularity to, the Rubik's Cube. Positions of the cube correspond naturally to elements of an astronomically large group of permutations, the solved state corresponds to the identity element, and moves of the faces correspond to group generators. Solving the puzzle invokes, often in layman disguise, group theory concepts such as permutation parity, conjugations, and commutators, and indeed these notions are often independently discovered by those attempting to solve the puzzle without assistance. Now in its 40th year, the Rubik's Cube is in the midst of a remarkable resurgence. The World Cube Association now lists nearly 30,000 competitors worldwide, of whom over 5,000 have an official solve under fifteen seconds!

The Rubik's Cube is not just one puzzle, but many. If we take a solved cube and execute a single twist, even a small child will be able to solve it. If we do just two twists, a coworker can almost certainly solve it. But already by three twists, many people might only inadvertently scramble the puzzle further while attempting to solve it. On the other hand, given a thoroughly scrambled cube, many puzzlers will, with a

*Published electronically November 6, 2014. This paper originally appeared in *SIAM Journal on Discrete Mathematics*, Volume 27, Number 2, 2013, pages 1082–1105.
http://www.siam.org/journals/sirev/56-4/97349.html

†725 B Loma Verde, Palo Alto, CA 94303 (rokicki@gmail.com).
‡Bismarckstr. 31, 64293 Darmstadt, Germany (kociemba@t-online.de).
§Department of Mathematical Sciences, Kent State University, Kent, OH 44242 (davidson@math.kent.edu).
¶Google Inc., Mountain View, CA 94043 (johndethridge@gmail.com).

bit of effort, be able to complete a single face. Solving the puzzle requires a sequence of small insights that lead to tangible progress: first a single layer can be solved, then all eight corners, and so on.

Knowing how to solve the cube does not exhaust the puzzle. Can you solve it faster than your neighbor? How short a solution can you generally find? Can you calculate how many different positions are possible? How many positions are possible if you only move three faces? Can you write a computer program or build a robot that can solve it?

Determining short solutions to a given position by hand can be difficult, but this endeavor is a popular event known as "Fewest Moves" in the competitive cubing community—move counts in the mid-twenties for random positions are not uncommon. Remarkably, computer programs that could duplicate this feat did not exist until 1992 [3], nearly two decades after the cube was invented. A couple of years later, a program was finally written [7] that could determine a shortest possible solution for a given position, but it typically required a full day of computation for a single position.

We can define the difficulty of a position as the length of the shortest possible sequence of moves that solves the position. We call this the distance of the position, with the solved position at distance zero. Every position has a unique distance, and there are only a finite number of positions, so clearly there must be one or more "most difficult" positions—those that have the greatest distance. Can we determine that maximum distance, the so-called diameter of the puzzle, and perhaps calculate some or all of those positions? These are the primary questions with which this paper is concerned.

Ideally, the diameter question could be answered with some straightforward mathematical analysis, yielding a proof based on structural properties and how the moves affect them, or a short recursive proof like the well-known analysis of the Towers of Hanoi puzzle. While such a proof may yet exist, over thirty years of investigation by a large number of laymen and academicians alike have yet to uncover it. Our approach explores essentially all of the 4.3 quintillion positions of the cube, by leveraging symmetry and group structure to gain the computational efficiency needed to determine the diameter in a reasonable time frame. The insights that allowed us to calculate the diameter may yet prove useful in more general problems of state space exploration, for example, those used in formal verification (model checking) or planning in artificial intelligence.

Our interest in this problem stemmed from its challenge as a problem in pure mathematics and its numerical allure, akin to computing digits of pi or calculating large prime numbers. The cube has lived up to Ernő Rubik's intentions—as a form of recreational computing it has fomented interest, honed skills, and fostered visualization and practical experimentation in a generation of mathematicians, computer scientists, engineers, and even architects and artists.

While this paper does utilize group theory, the dependence is basic enough that we are able to illustrate its application using common-sense examples and explanations. For example, if one reads "subset" for "coset" throughout, the essential ideas can be understood.

In this paper we give an expository account of our proof that every position of Rubik's Cube[1] can be solved in 20 moves or fewer, where a move is defined as any twist of any face. This manner of counting moves, known as the half-turn metric

[1]Rubik's Cube is a registered trademark of Seven Towns Limited.

(HTM), is by far the most popular move-count metric for the cube and is understood to be the underlying metric when discussions around this problem, also known as finding "God's number" for the puzzle, arise in various online forums devoted to the cube. The problem has been of keen interest ever since the cube appeared on shelves three decades ago. In group theory language, the problem we solve is to determine the diameter, i.e., maximum edge-distance between vertices, of the HTM-associated Cayley graph of the Rubik's Cube group. As summarized in the next section, many researchers have found increasingly tight upper and lower bounds for the HTM diameter of the cube. The present work explains the computational aspects of our proof that it equals 20.

While the technological improvements of increased memory capacity and greater CPU power were necessary for this result, it was primarily a series of mathematical and algorithmic improvements that allowed us to settle the question in the current time frame. Our decomposition of the problem, along with certain insights, let us solve it with a program capable of doing about 65 billion group operations per second on a single common desktop CPU, which works out to be about 23 Rubik's Cube group operations per CPU cycle.

The decomposition is based on a subgroup H of the cube group. Following the section on the problem's history, our paper recommences with a description of this subgroup and its properties, including the notion of an H-wise *relabeling* of the cube for illustration purposes. We then introduce the notion of *canonical sequences*, relate them to the count of positions at a given distance from the solved position, and describe how we can use these ideas to simultaneously and quickly solve many positions at a time. We discuss the ideas and implementation of our solving program itself, first giving an overview of the two main phases of the coset solving program: search and *prepass*. We then present some empirically derived heuristic improvements that were key to the success of the final runs. Finally, we describe how we set up the actual final runs and give some statistics and details from those runs, followed by some ideas for future work.

In particular, the diameter question can be posed relative to move-count metrics other than the HTM. Both the quarter-turn metric (where a half-twist of a face constitutes two moves) and the slice-turn metric (where a middle slice may be rotated in a single move) are of significant popular interest and can be approached with techniques like the ones we use.

Our new and essential code optimizations were rather intricate and so are merely summarized, using short code excerpts, in the search and prepass sections. Our critical inner loop as compiled by the GNU Compiler Collection (GCC) is given as an appendix. The source code includes multiple implementations of its key algorithms at increasing levels of sophistication; this was critical to verify the correct operation of the code. In addition, a completely independent (sharing no code) high-performance version of the solver was created by one of the authors, and many comparisons were made between the results of the two for verification. Annotated source code and many further optimization details are available at http://cube20.org/src/.

It is worth mentioning one improvement not detailed here since it would take us too far afield, namely, our treatment of a set cover problem which allowed us to further reduce, beyond what standard combinatorial arguments give, the number of cosets of H that had to be processed. This gave an additional speedup factor of about 2.5, incremental in relative terms and of interest here only as our last major improvement prior to seeking a computing sponsor. The mathematical and computational derivation of our set cover solution will be treated in a separate publication, but the

actual solution is posted with our source code and may be easily verified, as three of us did independently from scratch. Despite all of our efforts to keep the total runtime low, the need for substantial computing resources was inevitable using the current approach on current hardware.

2. Previous Work. Despite widespread interest in the mathematics of the Rubik's Cube, not only among mathematicians and computer scientists but also in the popular press, the basic problem of finding the diameter of the group, in the half-turn or any other nontrivial metric, has remained unsolved for more than 30 years.

By 1980 it was understood from a very simple counting argument (see [26]) that at least 18 moves were required for some positions, but no positions had yet been proved to require more moves. Thistlethwaite [27] proved in July 1981 that 52 moves suffice. For most of the next decade, these two results were all that were published.

In 1990, Kloosterman lowered the upper bound to 42 [5]. In 1992, this bound was lowered to 39 by Reid [12], and then to 37 by Winter [28]. Also in 1992, Kociemba [3] introduced his two-phase algorithm, surprising the cube world with an algorithm that found near-optimal solutions rapidly on consumer hardware (the Atari ST) with only a megabyte of memory.

In 1995, Reid lowered the upper bound to 29 [13] and also increased the lower bound to 20 [14] by proving that the "superflip" position (all corners solved, all edges flipped in their home positions) requires 20 moves to solve. Two years later, Korf [7] introduced the first practical computer program for optimally solving arbitrary positions in the Rubik's Cube group with a typical runtime of about a day.

The next seven years were relatively quiet, but the 2004 reincarnation of the "cube-lovers" mailing list as an Internet forum by Mark Longridge unleashed a flurry of activity. In 2005 Radu [11] lowered the upper bound to 28 and then in 2006 lowered it again to 27, both times using only a small amount of computer time. In 2007, Kunkle and Cooperman lowered the bound to 26 using a computer cluster; see [9] and [10].

It was during this period that we began the work presented in this paper [16]. In 2008, as our technique matured, we were able to lower the bound to 25 [17] (using only a few home machines) and then 23 [18] and 22 [21] (with John Welborn using idle time at Sony Pictures Imageworks). Further improvements in efficiency led to the present work.

To give an idea of the size of the problem, there are about 4.3×10^{19} (43 quintillion) positions in the cube group. We can immediately reduce this number by about a factor of 48 by considering the spatial symmetries of the cube: 24 rotational symmetries about various axes (e.g., the line through the centers of the up and down faces), 12 reflectional symmetries about various planes (e.g., the up-down "equatorial" plane), and 12 additional composite symmetries (e.g., up-down reflection followed by a 90-degree rotation about the up-down axis). Further reduction by about a factor of 2 is realized by group inversion, noting that when a solution sequence is inverted (i.e., the moves applied in reverse order with each move individually reversed), one generally obtains a solution sequence to a different scrambling. The fastest optimal solver we know of, based on Korf's ideas and requiring 33GB of RAM for tables, can solve positions on our Intel Nehalem baseline processor at a rate of 2.0 positions per second. Applying this optimal solver to each position after the reductions above would require seven billion CPU years. The fastest existing algorithm for finding near-optimal solutions, Kociemba's two-phase algorithm, when optimized carefully for batch speed, can find length-20 solutions for random positions at a rate of 3900

Table 2.1 *The solution rate, in positions per second, for four different algorithms for solving the cube. The first row shows existing position-at-a-time techniques, and the second row shows the rate when solving an entire coset at a time using our coset solver.*

	Optimal	Near-optimal
Individual position	2.0	3900
Coset of H	2×10^6	10^9

per second. The time required to solve all positions near-optimally, this way would be about 3.7 million CPU years. These two solvers are listed in the first row of Table 2.1. The second row gives our contribution. When solving an entire coset of positions at once, we can solve about two million per second optimally and about a billion per second near-optimally. One revealing aspect of our proof that the HTM diameter equals 20 is that it used fewer core-cycles (about 1.2×10^{19}) than there are cube positions.

3. Overview. Our determination of just how scrambled a cube can get is based on a single key idea: partition the space of cube positions into subsets such that each subset can be solved quickly and independently. In order to set the stage for the details of the actual program given in later sections, in this overview section we consider how our overall approach would apply to a much simpler and more familiar decomposition of the space.

Given a cube to play with, most people will attempt to solve the top layer first and proceed with the rest of the cube only after they have solved that layer. This subgoal is reasonably achievable by most people with a little bit of effort. It is also quite simple for a computer. There are only about 26 billion different "top-layer positions," i.e., configurations of the top eight *cubies* (or movable pieces) when distributed throughout the entire cube, so a table can be constructed in memory giving the number of moves each of these positions is from having the top layer solved. To find a solution to the top layer, we look up this top-layer distance for the current position and then try each move one by one on the current position to see which of them bring us closer to having the top layer solved. Such a table, called a pattern database [1], allows us to generate solution sequences, often including several "optimal" (i.e., shortest-length) solution sequences, for any top-layer position very rapidly.

The vast majority of those who solve the top layer for the first time will still have much work to do, with the remaining cubies in a state of disarray. But every so often one lucky person may find that their solution to the top layer also fortuitously solved all the remaining cubies as well. Each solution to the top layer is also a solution to some full-cube position, and different solution sequences to the top layer are typically full-cube solutions to different initial positions. Indeed, it is easy to compute which full-cube position is solved by a given solution to the top layer: simply start with a solved cube and "undo" the given solution, reversing each individual move and the order of the move sequence.

Since we can use a pattern database to generate many solutions to a given top-layer position, and since each such solution solves some full-cube position, it follows that we can keep track of which full-cube positions we have seen and therefore quickly find solutions to many different positions. This is the key idea behind our effort.

Given a set of move sequences that solve some particular top-layer position, the full-cube positions that they solve are related in that they all have the same initial position of the top cubies. Thus, we can use the position of the top-layer cubies to

partition the cube space into subsets, and then separately solve each subset, i.e., solve all positions in the subset. The number of subsets in the partition is the count of top-layer positions, or about 26 billion, and the number of positions in each subset is the count of reachable positions of the remaining cubies, or about 1.7 billion; the product of these two counts is the total number of reachable cube positions.

To generate nonoptimal solutions to the top layer using the pattern database, we need something slightly more complicated than we described above, since not all moves in such sequences will necessarily take us closer to the solution. However, we do know that at any point with, say, only n moves left, if our current position is more than n moves from the start according to the pattern database, we cannot find a solution from that position and should backtrack. This leads to a straightforward depth-first search with pruning:

```
function search(position p, sequence sofar, int togo) {
   if (dist[p] > togo)
      return ;
   if (togo == 0)
      record(sofar) ;
   else
      for (move m in moves)
         search(p.move(m), sofar.append(m), togo-1) ;
}
```

Because the average distance returned in our pattern database is about ten, this depth-first search can handle very deep searches quickly. Indeed, it takes only $O(d)$ time per solution, where d is the number of moves in the solution.

In order to solve an entire subset in our partition, we need to keep track of which positions in the subset have been solved and which remain unsolved. For all solutions generated, the above algorithm performs a `record()` operation, which is responsible for keeping track of the positions solved by the sequences generated by the search. We assign an index to each position in the subset and then use a Boolean array (stored densely as a bitmap) to track which positions we have seen. If we generate solution sequences ordered by length, then the first time we see a solution for a particular position, we know it is an optimal solution to that position. This is the principle behind the iterated deepening A* (IDA*) algorithm [6], except, instead of searching for the solution to a single position, we search for solutions to all the sequences in a large set. Once all the bits in the Boolean array are set, we have found a solution to all positions in the subset and we know precisely how many were of each length. In one run of this algorithm, for a given partition, searching out to whatever depth is needed, we solve 1.7 billion distinct cube positions.

The problem with this approach is that one might see millions of solutions to an easy position before seeing the first solution to one of the more difficult (deeper) positions. One fix is to just run the algorithm for a certain amount of time, or until a certain relatively small number of positions are left, and then solve all remaining positions using a different technique. For instance, one might enumerate all solutions in the subset through length 19 and then solve the very few remaining positions with another program.

But there is another fix that is even more effective. Once the top layer is solved, no twist of the bottom layer will change the top layer. Therefore, any solution to the top layer can be extended by a move of the bottom layer to create a new, different solution to the top layer. With some careful indexing and layout of the bitmap, we can extend an entire set of already-solved positions by one of these bottom-layer

moves using fast table lookups and logical bit operations. (How this can be done will be described in a later section.) For example, if we have explored all solutions through length 13, we can then extend this to all solutions of length 14 that end in a bottom-layer move in a single fast pass over the bitmap, with no need for additional search. For each original position in the bitmap, there are three possible bottom-layer moves to extend them, essentially multiplying the effectiveness of our search. We call this operation a prepass because we run it before our search, which then generates all sequences at each level:

```
for (int depth=0; ; depth++) {
   prepass() ; // extend position[a] set at depth d-1 to depth d
   search(inputpos, empty-sequence, depth) ;
               // search for new positions at distance d
}
```

With this structure we can also change our search routine to never generate solution sequences that end in a bottom-layer move; this is a somewhat subtle optimization but improves the search speed. In practice, the prepass eliminates a level of search; a full depth-18 search followed by the fast prepass operation solves a very large fraction of distance-19 positions. Without the prepass operation we would need a good fraction of a depth-19 search to solve an equivalent number of positions.

4. The Subgroup *H* and Its Cosets. The partitioning described above is a partitioning of a group into the cosets of a subgroup. The group is the full space of cube positions; the subgroup is the subset of positions that solve the subgoal (for instance, with the top layer solved). If the move sequence m solves the subgoal for an initial position p, then it also solves the subgoal for all positions that are in the same coset as p.

The primary difficulty with exploring any state space this large is keeping track of which states have been seen so far and which have not. By partitioning the space into subsets small enough to fit in memory and solving each subset independently, we can explore much larger state spaces.

Let us consider some subgroup H. To examine a specific right coset C of this subgroup we take an arbitrary representative p of C, so $C = H \cdot p$. With a search at depth d we find all move sequences m of length d that take the initial position p into the subgroup, that is, $p \cdot m \in H$. For each of these move sequences m there is exactly one cube position of C which is solved by this move sequence, namely, m', where m' is the inverse of m. The sequence m' is in C because $p \cdot m$ is some $h \in H$ with $p = h \cdot m'$, and therefore $h' \cdot p = m'$ so $m' \in H \cdot p$.

We keep track of solved cube positions with a bitmap. When the search at depth d is complete, we know which cube positions of C are solvable within d moves. When we use the term *search* in what follows we mean this procedure. A carefully chosen subgroup H has the property that there are some single moves that, when applied to a position in H, result in a position still in H. We use these moves to extend the bitmap obtained from the depth-d search to find all cube positions in C which are solved by sequences of length $d+1$ which end in such a move. We call this a prepass because we apply it before the search at depth $d+1$.

When we talk about the subgroup H from now on, we refer to that well-known subgroup that is the second goal state of Thistlethwaite's four-stage algorithm [27] and the first goal state of Kociemba's two-phase algorithm [3]. As we shall see in this section, this subgroup choice provides unique advantages to our prepass technique, so much so that the bulk of the computational work of our proof was done with the

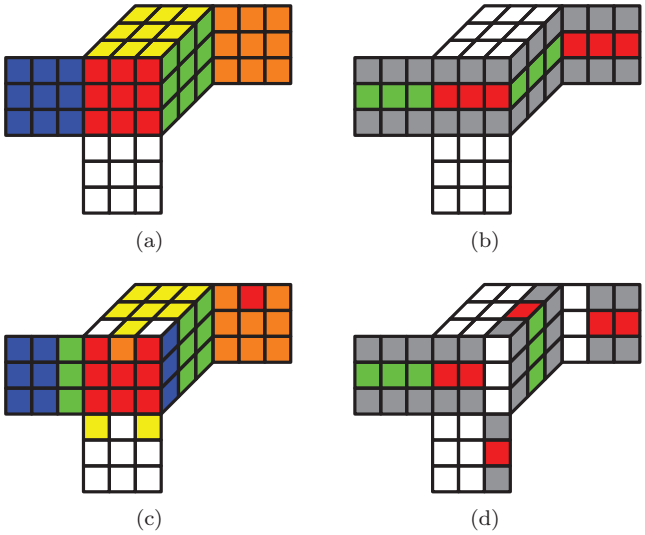

Fig. 4.1 (a) *The normal coloring of a solved cube,* (b) H*-wise relabeling of the solved position,* (c) *the position generated by* $R1L3U2L1R3$*, and* (d) *the coset* $H \cdot R1$.

prepass, and not with the search. Furthermore, this subgroup is small enough that the bitmap used in the search fits into memory.

The top-layer-solved subgroup we used in the prior section is easy to visualize; we just ignore cubies not originally from the top layer. An informal way to understand the H subgroup is to change a few of the stickers on the solved cube and remove some others, as shown in Figure 4.1(b). First, we give opposite faces of the cube the same color stickers, and second, we remove the stickers with a vertical orientation from the first and third layers of the cube. In this restickering, all the corner cubies are identical and there are only two distinguishable types of edges: middle edges and top/bottom edges. This relabeled puzzle is much simpler to solve than the full-cube puzzle, much like solving only the top layer is much easier than solving the whole cube.

We are relabeling cubies, and not positions; another description of the same relabeling that applies to any scrambled position goes as follows. Let us assume that the colors of opposite faces of the solved cube are yellow and white, blue and green, and red and orange and that yellow and white are oriented so they are the top and bottom faces, respectively. We relabel all yellow stickers to white, all blue stickers to green, and all orange stickers to red. Next, we remove any sticker that is adjacent to a white sticker on the same cubie.

The unique solved position of this relabeled cube is just the relabeled full cube: top and bottom faces are solid white, and the four other faces have a single horizontal stripe of a single color. The reason the prepass is so effective when this subgroup is used is that so few moves affect this solved position; no twist of the top or bottom face, nor any half-twist of any of the other four faces, changes the appearance of the relabeled cube. This means our prepass can use ten of the 18 moves to extend the set of found positions, rather than just three when using the top-layer subgroup. If our search phase examines n sequences to determine the found positions, then a single prepass operation on those positions is essentially generating $10n$ additional sequences, and possibly many additional positions, all in a single pass over the bitmap.

It is convenient to provide short names to the faces and moves. Viewing the puzzle's surface from its exterior as usual, a clockwise quarter-twist of the "up face" (U) is denoted here by U1, a half-twist by U2, and a counterclockwise quarter-twist by U3. We do likewise for the remaining faces: down (D), right (R), left (L), front (F), and back (B). This defines 18 moves; we refer to this set of moves as $\mathcal{S}$. The cube group G can then be viewed as the action of the free group $\langle \mathcal{S} \rangle$ on the set of cubie (individual block) stickers, also known as facelets, and the state of the puzzle is determined by the placement and orientation of the noncenter cubies (8 corners and 12 edges). The subgroup H is defined as being generated by all moves of the U and D faces and half-turns of the remaining faces. We define this subset of 10 moves as

$$\mathcal{A} = \{\text{U1,U2,U3,D1,D2,D3,F2,B2,L2,R2}\} \tag{4.1}$$

and the subgroup so generated as $H = \langle \mathcal{A} \rangle$. We divide the 43,252,003,274,489,856,000 distinct cube positions of the cube group G into 2,217,093,120 cosets of H in G. We will simply call them cosets of H since, as with all later coset spaces introduced in this paper, the parent subgroup will always be G. Each coset of H is composed of 19,508,428,800 cube positions, which is small enough that a check-off table can comfortably fit in the RAM on a good modern desktop system. We chose this approach because we are able to solve every element of a coset of H with at most 20 moves and, crucially, at an extremely rapid speed, by a method closely related to Kociemba's two-phase algorithm.

If we relabel a solved cube as described earlier, the moves from $\mathcal{A}$—which generate H—do not change the relabeled cube, so all cube positions in H show this color pattern after relabeling. But can we also be sure that all cube positions that show this color pattern after relabeling are elements of H? For example, the position generated by R1L3U2L1R3, shown in Figure 4.1(c), shows the color pattern of the solved cube after relabeling, but since its generator uses moves not in $\mathcal{A}$ it is not clear that it is an element of H. In fact, this position can also be generated by F2U2R2F2R2U2R2F2R2, so it is an element of H. The fact that any position with H-wise relabeling identical to that of the solved cube can be generated using only moves defining H is not obvious but can be verified either by a brute force calculation or by using stabilizer subgroup algorithms (e.g., as built into the computer algebra system GAP). It can also be proved with pen and paper alone by adapting well-known group-theoretic arguments, e.g., as presented in sections 11.2 and 12.1 of [2].

There are 8! ways to permute the corner (cubies) without changing the color pattern of the relabeled cube, while the 4 red/green edges of the middle slice can be permuted in 4! ways, and the remaining 8 edges with the white facelets can be permuted in 8! ways. So H would have 8!8!4! elements if we were allowed to scramble the puzzle by disassembling and then reassembling the cubies. Since we are constrained to only turning the faces, and since each face turn is an even permutation on the cubies, we find that $|H| = 8!8!4!/2 = 19{,}508{,}428{,}800$. As it is impossible to twist a corner or flip an edge on the relabeled cube without destroying the color pattern, flips and twists do not contribute to enumerating the elements of H.

The cosets of H can be visualized in a similar way, by a specific color pattern of the H-wise relabeled cube, such as in Figure 4.1(d). The three positions depicted in Figure 4.2 belong to the same coset because their H-wise relabelings are identical. In fact, they are identical to the same pattern shown in Figure 4.1(d), which depicts the right coset

$$H \cdot \text{R1} = \{h \cdot \text{R1} : h \in H\}.$$

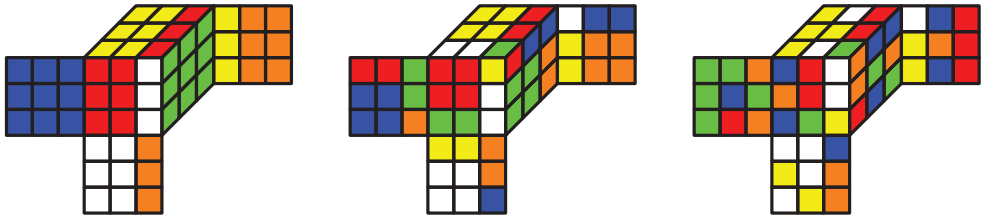

Fig. 4.2 *Three positions in the coset* $H \cdot R1$.

We now calculate the number of possible color patterns (i.e., the number of cosets of H) for an H-wise relabeled cube. First note that there are $\binom{12}{4}$ ways to assign the location of the 4 red-green edges. Flipping an edge or twisting a corner always gives a different pattern, as we noted above, so if disassembling the cube were permitted, we would get $2^{12}3^8$ possibilities for flips and twists. But, as before, the move constraints allow only $2^{11}3^7$ ways, since the flip/twist states of the final edge and corner are forced. As a result, there are $\binom{12}{4}2^{11}3^7 = 2{,}217{,}093{,}120$ different cosets of H.

The subgroup H exhibits 16-way symmetry. When we define an equivalence relation on the cosets of H by this symmetry, we find there are only 138,639,780 such equivalence classes, so we only need to solve a single representative of each one. Careful consideration of a decomposition of the cosets of H into cosets of a smaller subgroup permits us to take advantage of more of the natural 48-way symmetry of the cube and reduce the number of cosets to be solved to 55,882,296. Details of this reduction will be published separately.

5. Sequences of Moves. This section explains the relationship between move sequences and positions, defines the notion of *canonical sequences* from which an efficient search method is devised, and extends this notion to sequences ending in moves from $\mathcal{A}$. Our coset solver exploits the fact that the set $\mathcal{A}$ of generators of the subgroup H is a relatively large subset of the set $\mathcal{S}$ of all moves. For a set of positions in H, right-multiplication by moves in $\mathcal{A}$ results in positions in H. We can perform this operation (called a prepass) in bulk extremely efficiently, and this is the key to our overall result.

The performance of the coset solver is driven by practical considerations, so we must specify a hardware baseline. All performance numbers in this paper are measured on an Intel Nehalem X3460 CPU running at 2.8GHz with four cores and hyperthreading enabled. We distinguish core seconds (execution on one core) and CPU seconds (execution on the entire CPU). Our primary metric throughout is CPU seconds.

Our coset solver maintains a bitmap indicating which of the 19,508,428,800 elements of the coset have been solved at every point. Two primary techniques are used in the coset solver to check off positions in the bitmap. The first technique is an extremely fast search that enumerates canonical sequences of a particular length and marks the resulting positions in the bitmap. This search runs at about 25 million positions per second. At small lengths, when there are few canonical sequences (as described below), this completes quickly, while at larger lengths, with exponentially more positions, it can take much longer. The second technique is the prepass; it takes the set of positions already found and extends that set by all moves in $\mathcal{A}$ using a quick scan over memory and some logical operations. It is called a prepass because we use it for a particular depth before a search at that depth. The prepass is much faster than search on a per-position basis because it processes 24 positions and ten moves at a time in a short inner loop composed of just 61 machine instructions; this executes

group operations at a rate of 65 billion per CPU second. It forms the heart of our program, where most of the time is spent and where the most positions are found. Combined judiciously, these two techniques process a coset in, on average, about 20 seconds, or about a billion positions per second.

When searching for solutions to positions, it is important to minimize redundancy in the search tree. Separately searching sequences such as U1D2 and D2U1 that reach the same position in the same number of moves can slow down the search dramatically. In addition, sequences such as U1U2 are clearly suboptimal and should also be avoided. We define a canonical sequence as a sequence that obeys a specific order for adjacent individual moves that commute and is free of consecutive moves of the same face. These two restrictions alone are surprisingly effective.

We begin with a standard technique from linear algebra to determine the number, denoted here $q_n(\mathcal{S})$, of canonical sequences of length n from the set $\mathcal{S}$ of all 18 moves. (We could use a simpler approach here, but explicit matrix arithmetic makes it easier to extend these ideas to other sets of sequences, as is done later.) Ordering this set as

$$\mathcal{S} = \{\text{U1, U2, U3, F1, F2, F3, R1, R2, R3, D1, D2, D3, B1, B2, B3, L1, L2, L3}\},$$

we decree that a two-move sequence $s_i s_j$ of elements from $\mathcal{S}$ is canonical if and only if the (i, j)th entry of the following 18×18 incidence matrix C is 1:

$$C = \begin{pmatrix} Z & W & W & W & W & W \\ W & Z & W & W & W & W \\ W & W & Z & W & W & W \\ Z & W & W & Z & W & W \\ W & Z & W & W & Z & W \\ W & W & Z & W & W & Z \end{pmatrix},$$

where

$$W = \begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix} \text{ and } \quad Z = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}.$$

Any move is permitted as the first move in a canonical sequence and serves as the base case for the following inductive procedure to specify and enumerate the canonical sequences of a given length. To initialize the process, we let $\mathbf{a}$ be the 18-entry column vector composed solely of 1's, and denote its transpose by

$$\mathbf{a}^\top = (\underbrace{1, \ldots, 1}_{18 \ \text{1's}}).$$

The product $\mathbf{a}^\top C$ is then a row vector which encodes, for each of the 18 possible moves, the number of canonical two-move sequences ending with that move:

$$\mathbf{a}^\top C = (\underbrace{12, \ldots, 12}_{\text{nine 12's}}, \underbrace{18, \ldots, 18}_{\text{nine 18's}}).$$

For example, the first entry in this row vector indicates that there are 12 canonical two-move sequences that have U1 as the second move. To be canonical, their first move must be one of the 12 possible moves of F, R, B, or L, because consecutive moves

of the same (U) face must be avoided, and we require twists of the U face before twists of the D face whenever these faces occur consecutively. To get the total number $q_2(\mathcal{S})$ of two-move canonical sequences, we multiply $\mathbf{a}^\top C$ by the column vector $\mathbf{a}$, giving the 1×1 matrix $\mathbf{a}^\top C\mathbf{a} = (243)$. Extending this and dropping the matrix parentheses, as is usual for 1×1 matrices, we have

$$q_n(\mathcal{S}) = \mathbf{a}^\top C^{n-1}\mathbf{a}.$$

Since all of our restrictions can be categorized according to faces, we can collapse rows and columns together and instead redefine the matrices as

$$\mathbf{a}^\top = (\underbrace{1,\ldots,1}_{\text{six 1's}}), \qquad C = \begin{pmatrix} 0 & 3 & 3 & 3 & 3 & 3 \\ 3 & 0 & 3 & 3 & 3 & 3 \\ 3 & 3 & 0 & 3 & 3 & 3 \\ 0 & 3 & 3 & 0 & 3 & 3 \\ 3 & 0 & 3 & 3 & 0 & 3 \\ 3 & 3 & 0 & 3 & 3 & 0 \end{pmatrix},$$

in which case we get

$$q_n(\mathcal{S}) = 3\mathbf{a}^\top C^{n-1}\mathbf{a}.$$

Using this formula one can readily derive the recurrence relation

$$q_n(\mathcal{S}) = 12q_{n-1}(\mathcal{S}) + 18q_{n-2}(\mathcal{S}) \quad (n \geq 3),$$

which can also be seen combinatorially (see p. 34 of [26]). The matrix approach allows easy adaptation to further restrictions on allowed moves, which we use when analyzing the prepass.

When we compare the count of canonical sequences of length d to the known count of Rubik's Cube distance-d positions in Table 5.1, we find the distance 16; this indicates that basing a search on canonical sequences is sufficient to nearly eliminate any inefficient repeated search of the same position.

The count of canonical sequences allows us to estimate the runtime of a search through a particular depth. Empirically, our search implementation can visit about 25 million sequences per second. There are $2^{11}3^7\binom{12}{4} = 2{,}217{,}093{,}120$ cosets of H, and the canonical sequences are split among these cosets. Our time required to process a single coset by canonical sequences at depth n, in seconds, is therefore on average about $q_n(\mathcal{S})/(2{,}217{,}093{,}120 \times 25{,}000{,}000)$.

Below depth 14, the time is dominated by the setup time. For depths $d \geq 14$, we get the search times per coset given in the first two columns of Table 5.2 by summing the above for n from 1 to d.

One technique to prove a diameter of 20 for a given coset, then, would be to do a full search to depth 19 within the coset, thus finding all distance-19 positions in the coset. Any remaining positions must be of distance 20 or more. We then execute a single prepass, which will almost certainly prove that all remaining positions in the coset are of distance 20. But if it does not, and there are remaining positions, we can solve them separately using Kociemba's two-phase solver. This technique for a single coset would take about 16 CPU hours. All 138,639,780 distinct cosets of H would require about 280,000 CPU years, a substantial improvement over the 3.6 million CPU years calculated earlier, but clearly still not feasible.

Table 5.1 *The count of canonical sequences of length d and positions at distance d. Note how close the values are through d = 16. The count of cube positions at distance* 15 *is a new result from this work. The approximate values are from* [20] *and* [19].

d	Canonical sequences	Positions
0	1	1
1	18	18
2	243	243
3	3,240	3,240
4	43,254	43,239
5	577,368	574,908
6	7,706,988	7,618,438
7	102,876,480	100,803,036
8	1,373,243,544	1,332,343,288
9	18,330,699,168	17,596,479,795
10	244,686,773,808	232,248,063,316
11	3,266,193,870,720	3,063,288,809,012
12	43,598,688,377,184	40,374,425,656,248
13	581,975,750,199,168	531,653,418,284,628
14	7,768,485,393,179,328	6,989,320,578,825,358
15	103,697,388,221,736,960	91,365,146,187,124,313
16	1,384,201,395,738,071,424	≈1,100,000,000,000,000,000
17	18,476,969,736,848,122,368	≈12,000,000,000,000,000,000
18	246,639,261,965,462,754,048	≈29,000,000,000,000,000,000
19	3,292,256,598,848,819,251,200	≈1,500,000,000,000,000,000
20	43,946,585,901,564,160,587,264	≈300,000,000

Table 5.2 *Search time for one coset of H up to a given depth, count of sequences with n moves from $\mathcal{S}$ followed by m moves from $\mathcal{A}$, time for m prepass operations, and total time (prepass plus search). Note how the total time increases much more rapidly than the sequence count as n increases past* 15.

n	Search time	m	$q_{n,m}(\mathcal{S},\mathcal{A})$	Prepass time	Total time
14	0.14 seconds	6	8.10×10^{20}	18.6 seconds	18.7 seconds
15	1.9 seconds	5	1.59×10^{21}	15.5 seconds	17.4 seconds
16	25 seconds	4	3.13×10^{21}	12.4 seconds	37.4 seconds
17	5.6 minutes	3	6.18×10^{21}	9.3 seconds	5.8 minutes
18	1.2 hours	2	1.21×10^{22}	6.2 seconds	1.2 hours
19	16 hours	1	2.44×10^{22}	3.1 seconds	16 hours
20	9 days	0	4.39×10^{22}	0	9 days

There is a better approach than this for finding optimal solutions for every position in a coset. A full depth-18 search followed by a prepass finds all distance-18 positions and the majority of distance-19 positions in that coset. Any leftover positions are distance-19 or deeper, and they are thought to be very few [19]. On these leftover positions, we first use Kociemba's two-phase algorithm, limited to a few seconds per position. This succeeds a majority of the time. For the remaining positions, we apply an optimal solver, which runs at about 2×10^6 solutions per second. This combination of techniques reduces the search time for finding the overall diameter of the group to 21,000 CPU years.

For the purposes of proving that the diameter is 20, positions need only be solved in at most 20 moves each, that is, we do not need to find optimal solutions for every position. We do this by searching to a lesser depth (say, n) and then performing $20-n$

prepasses to find distance-20 solutions. Choosing the right value for n is critical; a smaller n will run more quickly but increases the chance of having many remaining positions without solutions of 20 moves or fewer.

We know that a depth-19 search eliminates almost all positions, and fewer than one in a billion positions have distance 20. We also know that the number of length-19 canonical sequences is about 3.29×10^{21}. If we do not search past depth n, but only use prepasses, this restricts the canonical sequences to those where all moves past the nth move must come from $\mathcal{A}$. We can use our matrix representation to compute how many such canonical sequences there are and compare this with $q_{19}(\mathcal{S})$ to find out how deep we need to search to likely cover the space adequately. To this end we create a modified transition matrix $C_{\mathcal{A}}$ from C by removing those moves not in $\mathcal{A}$, i.e., the quarter-twists of F, R, B, and L. The resulting matrix, corresponding to the element ordering in (4.1), is

$$C_{\mathcal{A}} = \begin{pmatrix} 0 & 1 & 1 & 3 & 1 & 1 \\ 3 & 0 & 1 & 3 & 1 & 1 \\ 3 & 1 & 0 & 3 & 1 & 1 \\ 0 & 1 & 1 & 0 & 1 & 1 \\ 3 & 0 & 1 & 3 & 0 & 1 \\ 3 & 1 & 0 & 3 & 1 & 0 \end{pmatrix}.$$

As before, we see that the number of canonical sequences with the first n moves from $\mathcal{S}$ and the next m moves from $\mathcal{A}$, denoted $q_{n,m}(\mathcal{S}, \mathcal{A})$, is given by

$$q_{n,m}(\mathcal{S}, \mathcal{A}) = 3\mathbf{a}^{\top} C^{n-1} C_{\mathcal{A}}^{m} \mathbf{a}.$$

The results of this calculation are shown in Table 5.2. Since we are proving a diameter of 20, we have $n + m = 20$. A complete search at depth 16, extended by four moves from $\mathcal{A}$, may generate about as many positions as a complete search to depth 19. So we only need do a search to depth 16, requiring about 27 seconds total, followed by four prepasses, requiring about 3 seconds each, to solve the vast majority of the positions in each coset.

Empirically, we have found that doing a full search to depth 15 followed by five prepasses usually eliminates all but a few dozen positions. The canonical sequences are not distributed evenly over the cosets, so for some cosets there are significantly fewer length-15 canonical sequences; for these cosets, we do a partial depth-16 search. In sum, we average about 3s per coset on the search and 15s for the five prepasses, plus about 1.5s of overhead, for a total of about 19.5s per coset. On average we find 345 positions left per coset, which we can then quickly solve in 20 or fewer moves using Kociemba's two-phase algorithm. Using this technique, our CPU time requirements are reduced to about 87 CPU years. We were able to further reduce this to 35 CPU years by reducing the count of cosets needed. We plan to describe this improvement in a separate publication.

6. The Coset Solver.

6.1. Search Phase. Our coset solver relies on a fast IDA* search routine to seed the prepass. This search routine must enumerate canonical sequences of a given length that bring a particular coset representative p into the group H, calculate which element of H the representative p brings it to, and check and set bits in the subgroup bitmap corresponding to that element. We use a large exact pattern database to guide the search. The pattern database is indexed by the coset of H represented by the current position, and it gives the number of moves required to reach the subgroup H.

6.2. Search Optimizations. The set of canonical sequences can be thought of as a tree, with the empty sequence at the root, the sequences of length 1 at the next level down, and so on. Every node represents a particular sequence. Iterated depth-first search walks this tree at increasingly deeper levels. For our coset search with representative position p, we associate with each node the position reached by executing the sequence for that node from the position p. Each of these positions has a distance from the group H. Our pattern database is used to look up that distance. For a given search depth, we consider all the nodes in the search tree at that depth as leaves, because we do not visit deeper nodes in that iteration of the search.

The pattern database is indexed by cosets of H. There are 2,217,093,120 such cosets, but we exploit the 16-way symmetry of the H group and its cosets to reduce the size of the table to about 170 million entries. For fast move execution and fast indexing, we use a coordinate representation for the coset position, much as Kociemba does with his Cube Explorer program [4].

For a search to a particular depth, there are only certain canonical sequences of that length that take a coset representative p to the group H. If we enumerate all of these sequences, we must visit all of the corresponding nodes and their ancestors in the search tree. Any time we visit a node that is not one of those solving sequences or a prefix of a solving sequence, we waste execution time. We want to minimize the number of *false nodes*, whose sequences are neither solving sequences nor prefixes of solving sequences.

Normal IDA* search, using a pattern database to provide the heuristic function, will usually explore many more false nodes than useful ones. Every useful node must be visited, and normal IDA* search visits all immediate successors of every useful node, most of which are not useful.

The dominant cost of visiting a node in our type of IDA* search is querying the pattern database for the value of the heuristic function; the table is relatively large, and thus accesses will likely miss the processor cache and perhaps the translation lookaside buffer (TLB) as well. Every main memory access fetches 64 bytes on our Intel Nehalem processor, so we might as well make use of a few more bytes. We do this by including information on the distance of all neighboring nodes, as well as the node itself; we use this distance information to decide which neighbors are useful even before we calculate their position.

Unlike a normal IDA* search where the first successful sequence terminates the search, we want to enumerate all successful sequences. Using the information about the neighbors allows us to only visit useful nodes, walking down the tree to precisely those leaves that correspond to sequences that bring positions p into H. For the leaves, we do not perform a heuristic function lookup as we know all useful leaves correspond to positions in H.

One advantage of using H as the subgroup from which to compute cosets is that the useful subtree of the search tree has a much higher ratio of leaves to nonleaf nodes than other subgroups. This means the ratio of successful sequences (one per leaf node, each of which is a new candidate solution) to useful nodes (a basic unit of work in our tree search) is much higher. Consider, by comparison, the subgroup E that fixes edges, so the cosets are defined by the edge positions. The shortest canonical sequence between any two different positions in the subgroup E is quite large, because every move from a position in E messes up the edges, which then need to be put back into place.

In the useful subtree of the search tree for any coset of H, any ancestor node that has useful leaves has at least two useful leaves. The sequence for a given leaf must either end in a move in $\mathcal{A}$ or a move not in $\mathcal{A}$. If the sequence ends in a move in $\mathcal{A}$,

that means the position associated with the ancestor node is also in H, and, thus, it has several children in H, one for each move in $\mathcal{A}$ that is permitted after the move that precedes that move. If the predecessor move is a move of the U face, it will have seven useful leaves from moves in $\mathcal{A}$ of F2, R2, D2, B2, L1, L2, and L3. If it is a move of the D face, it will have four useful leaves from the moves in $\mathcal{A}$ of F2, R2, B2, and L2. If it is a move of F or R, it will have eight useful leaves from any move of U or of D, or a half-twist of the opposite face. If it is a move of B or L, it will have six useful leaves from any move of U or D.

If the sequence for a given leaf does not end in a move in $\mathcal{A}$, there are only eight possible moves (those not in $\mathcal{A}$) it could end with: quarter-twists of the faces F, R, B, and L. Let us assume without loss of generality (by symmetry) that the last move from the ancestor node to this leaf is F1. Then the move F3 from the ancestor node is also a position in H and hence is a leaf. This is because F1 and F3 differ by F2, which leaves an H position in H. Thus, if the sequence does not end in a move in $\mathcal{A}$, but there are useful leaves (that is, child nodes in H), then there must be exactly two of them.

For depth-16 searches or deeper, we use the prepass to handle all those canonical sequences that end in moves in $\mathcal{A}$, rather than explicitly enumerating them. At the beginning of a search at depth d, the coset bitmap represents all those elements of H that can be reached by move sequences of length $d-1$ applied to the coset representative p, so the prepass calculates exactly the same set of positions as a normal search phase at depth d restricted to sequences that end in moves from H. As a consequence, at depth d, we only need to explore sequences that end in a move not in H. This typically saves about 10/18 or 56% of the sequences (since there are ten moves in $\mathcal{A}$ but 18 in $\mathcal{S}$). It does not save quite that much execution time because the search nodes it eliminates are exactly those that have more than two useful leaves per ancestor node. Despite this, it is a useful optimization for the deeper searches.

In order to confine our search to sequences not ending in a move from H, we observe that the shortest canonical sequences that take a position that is in H out of H and then back into H have length 5 (such a sequence is R1L3U2L1R3). Any sequence that ends with a move not in $\mathcal{A}$, but ends with a position in H, must enter it on the last move. Thus, any time we enter H early in the search, we must have at least five moves remaining for us to exit H and reenter it; further, we must exit H with at least four moves remaining or we will not be able to get back into H in time.

There are two technical details worthy of mention. First, the use of 2MB TLB pages rather than the default 4K TLB pages can significantly reduce the cost of TLB misses; the libhugetblfs library can enable this easily on modern Linux kernels. Second, search required to find the next position can be overlapped with the memory access for the previous bit. Normally our search-driven bitmap updating might look like something like this:

```
void checkpos(position p) {
  unsigned char *addr = calcaddr(p) ;
  unsigned char b = calcbit(p) ;
  if (0 == (*addr & b)) { // new position found!
     count++ ;
     *addr |= b ;
  }
}
```

Modern processors include prefetching instructions that tell the memory system that the processor intends to use a particular memory location in the near future, so

it should be loaded into the cache but without stalling the execution pipeline to do so. We can take advantage of this by issuing such an instruction when we have found a new address to check, but not actually performing the check until we have found another position:

```
unsigned char *addr = 0 ;
unsigned char b = 0 ;
void checkpos(position p) {
  if (addr != 0 && 0 == (*addr & b)) { // new position found!
     count++ ;
     *addr |= b ;
  }
  addr = calcaddr(p) ;
  b = calcbit(p) ;
  prefetch(addr) ;
}
```

This idea can be extended to enable good scalability when multithreading the search. The pattern database itself does not need any sort of locking because it is read-only during the search, but updating the coset bitmap must be carefully managed so threads do not obliterate bits set by other threads. To do this, each thread builds a local queue of addresses and bits to check in the table. As it calculates each address, it prefetches the memory address to get the data into the cache. After a thread has accumulated, say, 64 such addresses, it then obtains a global lock, checks and sets all those 64 addresses, and releases the global lock. This eliminates any possibility of concurrent access to the coset bitmap, which guarantees correctness, yet permits the actual search for those addresses to occur in parallel, yielding high scalability. By making the queue reasonably short, no thread is held up too long waiting for the global lock, but by making it as long as we do, we amortize the cost of getting the global lock. In our testing, we see almost perfect scalability in the search phase up to eight threads.

6.3. Prepass Phase. Our coset solver maintains a large bitmap that represents the set of positions in that coset for which a solution has already been found. Every position in the coset is assigned an ordinal number, and we use an indexing algorithm to convert the position to that assigned ordinal number and another algorithm to convert an ordinal number to that position.

The coset positions that are being solved are elements of the coset $H \cdot p$, where p is a coset representative. Our search constructs sequences m such that $p \cdot m$ is in H (i.e., the sequence m brings the position p back into the coset H). It is much easier to index the elements of the group H than it is to index an arbitrary coset of H, because all positions in H share some common properties that are not shared by the positions of a coset of H. So instead of indexing the position solved by the sequence m, we index the position reached by $p \cdot m$, which is in H.

Positions in H all have the solved orientation on both edges and corners, so we do not need to consider orientation in our indexing operation. Further, all positions in H have the middle edges always in the middle slots. Thus, we can separate the state of a position in H into three portions: the permutation of the corners, the permutation of the up/down edges, and the permutation of the middle edges. These have sizes 8!, 8!, and 4!, respectively. The overall parities of these three portions must have an even sum, so the total size of the group is 8!8!4!/2; a bitmap of this size requires 2.3GB of

memory. We index each of these components separately and combine the results into a single index.

Overall, our prepass simply iterates over the range of the ordinals for the positions in H, and for each bit set in the bitmap, computes all neighbors to that position (by moves in H) and marks them in a new copy of the bitmap. The code essentially does the following:

```
newbitmap = oldbitmap ;
for (i : ordinals of H) {
  if (oldbitmap[i])
     for (move m : A) {
        cubepos p = unindex(i) ;
        p.move(m) ;
        newbitmap[index(p)] = 1 ;
     }
}
```

With careful organization of the bitmaps and a handful of lookup tables, we can optimize this routine to execute at a rate of about 65 billion potentially new positions per second on a single CPU. The next section describes how this is done.

6.4. Prepass optimizations. Making the prepass fast primarily involves organizing the bitmap in such a way that group operations can be performed in parallel with bitwise logical operations and relatively small lookup tables. To enable this, we split the overall ordinal indexing of the bitmap into three coordinates: the permutation of the corners, the permutation of the up and down edges, and the permutation of the middle edges (from most significant to least significant).

Because $8! = 40{,}320$ is so small, for the corners we simply construct an array `nextcorner[i][m]` that gives the ordinal value for the position of the corners after taking the position for ordinal i and making move m. We do the same thing for the eight up/down edges if we restrict moves to H, and the same thing for the four middle edges via arrays `nextudedge[i][m]` and `nextmedge[i][m]`. To reflect parity considerations, where the corner permutation parity must match the edge permutation parity, we do not distinguish the permutation of two of the up/down edges (chosen arbitrarily), since these are forced by the parity of the permutation.

With these three arrays in place, our prepass now looks like this:

```
for (c : ordinals of corners) {
  for (eud: ordinals of up/down edges) {
     for (em: ordinals of middle edges) {
        int i = (c * 8!/2 + eud) * 4! + em ;
        if (oldbitmap[i])
           for (move m : A)
               newbitmap[
        (nextcorner[c][m] * 8!/2 + nextudedge[eud][m]) * 4! +
                  nextmedge[em][m]] = 1 ;
     }
  }
}
```

If we premultiply the `nextcorner` array by $8!/2 * 4!$ and the `nextudedge` array by $4!$, the inner expression simplifies to

```
newbitmap[nextcorner[c][m] + nextudedge[eud][m] +
                                  nextmedge[em][m]] = 1 ;
```

We can hoist the indexing of `newbitmap` and `nextcorner` using a temporary array that holds the following sums:

```
for (c : ordinals of corners) {
  char *cptrs[] ;
  for (move m : A)
     cptrs[m] = newbitmap + nextcorner[c][m] ;
  for (eud: ordinals of up/down edges) {
     for (em: ordinals of middle edges) {
        int i = (c * 8!/2 + eud) * 4! + em ;
        if (oldbitmap[i])
           for (move m : A)
              cptrs[m][nextudedge[eud][m]+nextmedge[em][m]] = 1 ;
     }
  }
}
```

We now focus on moves of the U and D faces, which are six of the ten moves in $\mathcal{A}$. These moves do not affect the middle layer, so `nextmedge[em][m] == em`. There are 24 values of `em`, and we want one bit per value, so we recast our bitmap as an array of 24-bit integers, one integer per (`c`, `eud`) combination, with one bit of each integer corresponding to a value of `em`. We use the bitwise logical "or" operation to perform the test of the source and conditional setting of the destination. We modify the constant multiplications we have done in `nextudedge` and `nextcorner` to take this into account, and for the up/down moves only, we end up with

```
for (c : ordinals of corners) {
  int24 *cptrs[] ;
  for (move m : A)
     cptrs[m] = newbitmap + nextcorner[c][m] ;
  for (eud: ordinals of up/down edges) {
     int emset = oldbitmap[c*8!/2 + eud] ;
     for (move m : {U1, U2, U3, D1, D2, D3})
        cptrs[m][nextudedge[eud][m]] |= emset ;
  }
}
```

For the remaining moves in $\mathcal{A}$, which are F2, R2, B2, and L2, the middle edge position is modified, so we need to compute this modification. This modification to the middle edges, when a subset of 24 permutations is represented as a bitmask, is just a permutation of those bits. In practice, one must consider that modern commodity CPUs do not include instructions for arbitrary bit permutations. We could use a large lookup table (2^{24} entries for each of these four moves with each entry three bytes, requiring a total of 200MB) to perform the permutation of bits, but the resulting cache misses would destroy the performance we hope to attain. So instead we split the source value into two 12-bit subwords and use several lookup tables, each much smaller, to perform the bit permutation.

Our mapping of permutations to bits is arbitrary, so we put all the even permutations in the low 12 bits of the 24-bit word, and the odd permutations in the high 12 bits. Since all of F2, R2, B2, and L2 are odd permutations on the middle edges, this means low 12 bits from the source will all map to high 12 bits from the destination, and vice versa. We use eight arrays to perform this operation—an odd array and an even array for each of the four moves we handle.

We need additional code to extract the high and low words from the source, do the array lookups, and then combine the result. The code for just these four moves then looks like

```
int evenempermute[4][1<<12], oddempermute[4][1<<12] ;
for (c : ordinals of corners) {
  int24 *cptrs[] ;
  for (move m : A)
     cptrs[m] = newbitmap + nextcorner[c][m] ;
  for (eud: ordinals of up/down edges) {
     int emset = oldbitmap[c*8!/2 + eud] ;
     int emodd = emset >> 12 ;
     int emeven = (emset & 0xfff) ;
     for (move m : {F2, R2, B2, L2})
        cptrs[m][nextudedge[eud][m]] |=
                     (evenpermute[m][emeven] << 12) |
                      oddpermute[m][emodd] ;
  }
}
```

With tiny innermost loop bodies like this, it is advantageous to combine the two loops and unroll the inner bodies. This inner loop reads the source bitmap, calculates from it ten different parts of the destination bitmap, and modifies them in the process. This generates a lot of memory bus traffic to write the updated values back to memory, so we instead turn the loop around and compute the new value for the destination bitmap based on ten different portions of the source bitmap it reaches. We can do this transformation because $\mathcal{A}$ is closed under inversion, which significantly reduces the write traffic. We integrate this operation with the bitmap copy, yielding the following code:

```
int evenempermute[4][1<<12], oddempermute[4][1<<12] ;
for (c : ordinals of corners) {
  int24 *cptrs[] ;
  for (move m : A)
     cptrs[m] = oldbitmap + nextcorner[c][m] ;
  for (eud: ordinals of up/down edges) {
     int F2set = cptrs[F2][nextudedge[eud][F2]] ;
     int R2set = cptrs[R2][nextudedge[eud][R2]] ;
     int B2set = cptrs[B2][nextudedge[eud][B2]] ;
     int L2set = cptrs[L2][nextudedge[eud][L2]] ;
     newbitmap[c * 8! / 2 + eud] =
        oldbitmap[c * 8! / 2 + eud] |
        cptrs[U1][nextedge[eud][U1]] |
        cptrs[U2][nextedge[eud][U2]] |
        cptrs[U3][nextedge[eud][U3]] |
        cptrs[D1][nextedge[eud][D1]] |
        cptrs[D2][nextedge[eud][D2]] |
        cptrs[D3][nextedge[eud][D3]] |
        ((evenpermute[F2][F2set & 0xfff] |
          evenpermute[R2][R2set & 0xfff] |
          evenpermute[B2][B2set & 0xfff] |
          evenpermute[L2][L2set & 0xfff]) << 12) |
        oddpermute[F2][F2set >> 12] |
```

```
            oddpermute[R2][R2set >> 12] |
            oddpermute[B2][B2set >> 12] |
            oddpermute[L2][L2set >> 12] ;
    }
  }
```

The straight-line code in the inner loop above seems lengthy, but it performs ten moves on 24 different positions, and it does so with very simple instructions that execute quickly, using small lookup arrays that fit in either the cache or are accessed sequentially.

There are two more improvements we have made to the above code. First, since our arrangement of the mapping of middle edge permutation to bit position is arbitrary, except that the even positions must be in the lower 12 bits, we arrange it so that the `evenpermute` and `oddpermute` arrays for one of the moves represent the identity operation. This allows us to remove two of the eight arrays and their lookups. Further, we do not need to include the `oldbitmap[]` lookup (the first element of the large disjunction), because at any depth greater than 0, whenever a bit is set in the array, so also is a different bit that is related to this bit by a move in H. This is due to the shape of the search trees (as discussed in the previous section), where all leaf nodes always have immediate siblings related by moves in $\mathcal{A}$.

Modern CPU architectures do not have a native 24-bit integer value, but a 32-bit integer works fairly well as its replacement. This yields a bitmap that is too large, as eight of the 32 bits are wasted. In our testing on modern Intel and AMD processors, unaligned 32-bit accesses on an array with the 24-bit integers packed densely actually perform somewhat better than using the sparser layout.

Our final prepass consists of an inner loop containing 61 instructions including the test and branch at the end. For each three-second prepass operation, this inner loop is executed $8! * 8!/2$ or 812,851,200 times. On our Intel Nehalem processor, this is executed across eight logical threads in four physical cores, giving an approximate execution rate of 1.5 instructions per cycle per core, which is fairly good, especially considering how much memory traffic is generated. This inner loop handles 240 group operations each execution, so our operation rate is 65 billion group operations per second on this single CPU. The GNU-compiled code for this crucially fast loop is provided as an appendix.

There are a few additional but minor optimizations we have performed. We allocate portions of the new bitmap only as they are needed, and we free portions of the old bitmap upon their last use, so the overall memory impact is significantly lower than twice the size of a single bitmap. We evaluate the corner positions not in ordinal order, but in a different order that enhances the cache hit rate and also minimizes the maximum amount of memory allocated at once.

7. Heuristics. Much of the speed of our approach comes from using only the prepass at greater depths, but the prepass only considers moves in $\mathcal{A}$. Thus, not finding solutions of length 20 or less for some positions does not mean those positions require more than 20 moves to solve. We can call these positions that remain after a coset search *leftover* positions; we must consider them separately.

Luckily, the Kociemba two-phase search algorithm, especially when extended to six-axis search [21], can find length-20 or shorter solutions quickly, at an average rate of about 3900 positions per second. Thus, we want to balance the amount of time spent doing search in the coset solver against the expected count of leftover positions. In this section we discuss empirically derived heuristics that gave us a reasonable balance.

So far we have done our analysis based on the count of canonical sequences of a particular length divided by the count of cosets, that is, based on averages. In reality, some cosets have many more sequences of a particular length, while some have many fewer. For most cosets, a full search to depth 15, followed by five prepasses, finds solutions of length 20 or less to all positions except a few.

For some cosets this strategy leaves an impractical number of positions remaining to be solved with the two-phase algorithm. So for about a third of the cosets we do a partial depth-16 search. We performed a full search to at least depth 15 to compute, as a secondary and corroborating result, the total number of positions at a distance of exactly 15.

On average, a full depth-16 search would require about 27 seconds. Since without the depth-16 search the time per coset is less than 20 seconds, doing a full depth-16 search would more than double the time for those cosets.

For depth 16, we always first do a prepass. In our initial testing, we tried to find a particular count of total positions found at which to terminate the depth-16 search. However, we found that for the same total position count, the number of leftover positions differed significantly across cosets. Let us explain why.

After the depth-16 search, we only do prepasses. These prepasses only execute moves from $\mathcal{A}$. Let us determine how many next moves there are for a canonical sequence for moves in $\mathcal{A}$ and moves not in $\mathcal{A}$. For U1, U2, and U3, we have 7 possible successor moves. For F2 and R2, we have 10 possible successors. For D1, D2, and D3, we have only 4 possible successors, and for B2 and L2, we have 8 possible successors. The average over these 10 moves is 6.9 successors.

For moves not in $\mathcal{A}$, we have F1, F3, R1, and R3, each with 9 successors; for B1, B3, L1, and L3, we have 8 successors. The average here is 8.5 successors. So moves ending in $\mathcal{A}$ have, on average, fewer successors, and are therefore less valuable toward our goal of solving all positions in at most 20 moves.

As a solution, we derate positions already found by the depth-16 prepass when calculating the number of positions at which to terminate the search. As a result, almost all positions found at depth 16 were found by the prepass. We performed a sequence of experiments to determine by how much to derate these positions and at what point to terminate the search. If x denotes the total number of positions found after the depth-16 prepass, we perform a depth-16 search until the total number of positions found is $167{,}000{,}000 + x/3$.

8. The Big Run. Managing nearly 56 million independent coset runs on a large cluster can be difficult. To amortize the overhead of calculating or loading the required pattern database, we program the coset solver to iteratively execute a sequence of cosets in a single execution. By executing enough cosets in sequence in a single execution, the time to calculate the pattern database becomes negligible and all considerations of loading, storing, or distributing the large table are eliminated. To improve the credibility of our results, we calculate a hash of each pattern database a priori and validate that hash in memory at the beginning and end of each run.

To ease management of which cosets need to be solved, which are pending, and which are completed, we integrate the list of the cosets needed directly into the source of the coset solver, assign each an ordinal, and make each execution responsible for a contiguous range in that ordinal.

We also use the integrated coset list to compute the exact count of positions at each distance from 0 to 15. As Korf's article [8] indicates, at depth 10 a breadth-first search approach is already challenging. We include logic that, for the set of positions at

every level, computes those positions for which this particular coset was the canonical coset (the lowest-indexed coset), and which of the positions are duplicated in an earlier coset.

Since the count of positions through distance 14 is already known [24], verifying the sums of these counts across all 56 million runs provides a certain amount of validation of our results. We always perform a depth-15 search to completion, so we can also compute precisely the number of positions at that distance.

Almost all positions solved in one coset have a corresponding inverse position of the same distance in a different coset. This additional redundancy also helps validate our result.

All cosets and leftover positions were run on Google servers during July 2010. There were a total of 19,260,301,834 leftover positions (an average of 345 leftover positions per coset) that were individually solved using our six-axis two-phase algorithm. Using a simple Perl script to iterate through the logs and sum up the positions at each depth, we were relieved to find that the counts confirmed the known results for distances 0 through 14. We announced our new result for the count of positions at distance 15 (91,365,146,187,124,313) in July 2010 [22]. This result was independently confirmed by Scheunemann in August 2010 [25].

Because performance and specifications of the Google computers are confidential, and also to partially validate the results, we ran a subset of the run (80,100 of the cosets, or about one in every 700) on an individual Intel Nehalem machine and compared the results against those from the Google runs. We used performance numbers from these local runs to extrapolate the overall runtime had the full run been done on Intel Nehalem machines. This is the basis for the performance numbers throughout.

Our sample cosets averaged 19.622 seconds each to run, including the time required to solve the leftover positions, so we extrapolate the total run as 55,882,296 times this average or 34.75 CPU years. Of this time, 98.99% was spent in the coset solver, and 1.01% was spent in the two-phase solver for the leftover positions. About 16.0% of the total time was used for search, with 9.45% of the total time used for just the depth-15 search. Prepasses used 81.14% of the total time, with the average single prepass taking 3.119s. We note that leftover positions generally took a bit longer than random positions for the two-phase solver to solve, with a rate of only 1660 (vs. 3900) per second.

9. Future Work and Conclusions. We have computed the diameter of the Rubik's Cube group, but there is much still unknown about this group. In particular, the exact counts of positions at distances 16 through 20 are still unknown. Without an exact distance distribution, many may consider this puzzle still unsolved.

The diameter of the group in both the quarter-turn and the slice-turn metrics is still unknown. The best known results on the quarter-turn metric are a lower bound of 26 [15] and an upper bound of 28 [23]; for the slice-turn metric we know a lower bound of 18 and an upper bound of 20 (this paper). We anticipate God's number for the quarter-turn metric to be 26, and for the slice-turn metric to be 18; we are presently running a variation of the coset solver presented in this paper at the Ohio Supercomputing Center to resolve the quarter-turn metric.

Our approach of partitioning a large search space by cosets of a subgroup provides significant breathing room over standard in-memory breadth-first search, without introducing the latency or bandwidth limitations of keeping the bitmap in secondary storage. We plan to use this technique on additional problems.

10. Appendix. The inner loop of the prepass phase as compiled by GCC is shown here. By handling 65 billion group operations per second per CPU, its efficiency was key to our main result.

```
.L13:
        movzwl  6(%r10), %eax
        movq    -48(%rsp), %rdx
        movzwl  2(%r10), %ecx
        movl    (%rdx,%rax), %r9d
        movzwl  8(%r10), %eax
        movq    -40(%rsp), %rdx
        movl    (%rdx,%rax), %esi
        movzwl  16(%r10), %eax
        movq    -32(%rsp), %rdx
        movl    (%rdx,%rax), %edi
        movzwl  18(%r10), %eax
        movq    -24(%rsp), %rdx
        movl    (%rdx,%rax), %r8d
        movzwl  4(%r10), %eax
        movl    (%r15,%rax), %edx
        movq    -16(%rsp), %rax
        orl     (%rax,%rcx), %edx
        movzwl  (%r10), %eax
        orl     (%r14,%rax), %edx
        movzwl  10(%r10), %eax
        orl     (%r13,%rax), %edx
        movzwl  12(%r10), %eax
        orl     (%r12,%rax), %edx
        movzwl  14(%r10), %eax
        addq    $20, %r10
        orl     (%rbp,%rax), %edx
        movl    %r9d, %eax
        andl    $4095, %r9d
        sarl    $12, %eax
        andl    $4095, %eax
        orl     %eax, %edx
        movl    %esi, %eax
        andl    $4095, %esi
        sarl    $12, %eax
        andl    $4095, %eax
        movswl  rearrange+24576(%rax,%rax),%eax
        orl     %eax, %edx
        movl    %edi, %eax
        andl    $4095, %edi
        sarl    $12, %eax
        movswl  rearrange+8192(%rdi,%rdi),%ecx
        andl    $4095, %eax
        movswl  rearrange+8192(%rax,%rax),%eax
        orl     %eax, %edx
        movl    %r8d, %eax
        andl    $4095, %r8d
        sarl    $12, %eax
        andl    $4095, %eax
        movswl  rearrange+40960(%rax,%rax),%eax
```

```
orl     %eax, %edx
movswl  rearrange(%rsi,%rsi),%eax
orl     %ecx, %eax
movswl  rearrange+16384(%r8,%r8),%ecx
orl     %r9d, %eax
orl     %ecx, %eax
sall    $12, %eax
orl     %eax, %edx
movl    %edx, (%r11)
addq    $3, %r11
cmpq    %rbx, %r11
jne     .L13
```

Acknowledgments. We extend our sincere thanks to Google for providing the CPU power required to finally put this question to rest. We also owe a debt of gratitude to the two anonymous referees for their advice, corrections, and suggested improvements to this article.

REFERENCES

[1] J. Culberson and J. Schaeffer, *Pattern databases*, Comput. Intelligence, 14 (1998), pp. 318–334.
[2] D. Joyner, *Adventures in Group Theory: Rubik's Cube, Merlin's Magic & Other Mathematical Toys*, The Johns Hopkins University Press, Baltimore, MD, 2008.
[3] H. Kociemba, *Close to God's algorithm*, Cubism for Fun, 28 (1992), pp. 10–13.
[4] H. Kociemba, *Cube Explorer*, Windows program, http://kociemba.org/cube.htm.
[5] H. Kloosterman, *Rubik's cube in* 42 *moves*, Cubism for Fun, 25 (1990), pp. 19–22.
[6] R.E. Korf, *Depth-first iterative-deepening: An optimal admissible tree search*, Artificial Intelligence, 27 (1985), pp. 97–109.
[7] R.E. Korf, *Finding optimal solutions to Rubik's cube using pattern databases*, in Proceedings of the Fourteenth National Conference on Artificial Intelligence (AAAI-97), Providence, RI, 1997, pp. 700–705.
[8] R.E. Korf, *Linear-time disk-based implicit graph search*, J. ACM, 55 (2008), pp. 1–40.
[9] D. Kunkle and G. Cooperman, *Twenty-six moves suffice for Rubik's cube*, in Proceedings of the International Symposium on Symbolic and Algebraic Computation (ISSAC '07), ACM Press, 2007, pp. 235–242.
[10] D. Kunkle and G. Cooperman, *Harnessing parallel disks to solve Rubik's cube*, J. Symbolic Comput., 44 (2009), pp. 872–890.
[11] S. Radu, *Solving Rubik's Cube in* 28 *Face Turns*, http://cubezzz.dyndns.org/drupal/?q=node/view/37 (2005).
[12] M. Reid, *New Upper Bound*, http://cube20.org/cubelovers/CL08/075.txt (1992).
[13] M. Reid, *New Upper Bounds*, http://cube20.org/cubelovers/CL14/054.txt (1995).
[14] M. Reid, *Superflip Requires* 20 *Face Turns*, http://cube20.org/cubelovers/CL15/002.txt (1995).
[15] M. Reid, *Superflip Composed with Four Spot*, http://www.math.ucf.edu/~reid/Rubik/Cubelovers/ (1998).
[16] T. Rokicki, *In Search of:* 21*f*s and* 20*f*s; A Four Month Odyssey*, http://cubezzz.dyndns.org/drupal/?q=node/view/56 (2006).
[17] T. Rokicki, *Twenty-Five Moves Suffice for Rubik's Cube*, preprint, arxiv.org/abs/0803.3435, 2008.
[18] T. Rokicki, *Twenty-Three Moves Suffice*, http://cubezzz.dyndns.org/drupal/?q=node/view/117 (2008).
[19] T. Rokicki, *New estimate for* 20*f*:* 300,000,000, http://cubezzz.dyndns.org/drupal/?q=node/view/167 (2009).
[20] T. Rokicki, 1,000,000 *Cubes Optimally Solved*, http://cubezzz.dyndns.org/drupal/?q=node/view/172 (2010).
[21] T. Rokicki, *Twenty-two moves suffice for Rubik's cube*, Math. Intelligencer, 32 (2010), pp. 33–40.

[22] T. Rokicki, $15f^* = 91365146187124313$, http://cubezzz.dyndns.org/drupal/?q=node/view/197 (2010).
[23] T. Rokicki, *Twenty-Eight QTM Moves Suffice*, http://cubezzz.dyndns.org/drupal/?q=node/view/532 (2014).
[24] T. Scheunemann, *God's Algorithm out to* $14f^*$, http://cubezzz.dyndns.org/drupal/?q=node/view/191 (2010).
[25] T. Scheunemann, *God's Algorithm out to* $15f^*$, http://cubezzz.dyndns.org/drupal/?q=node/view/201 (2010).
[26] D. Singmaster, *Notes on Rubik's Magic Cube*, 5th ed., Enslow Publishers, Hillside, NJ, 1981.
[27] M. Thistlethwaite, *Thistlethwaite's* 52*-move algorithm*, http://www.jaapsch.net/puzzles/thistle.htm (1981).
[28] D.T. Winter, *New upper bound on God's algorithm for Rubik's cube*, http://www.math.ucf.edu/~reid/Rubik/Cubelovers/ (1992).