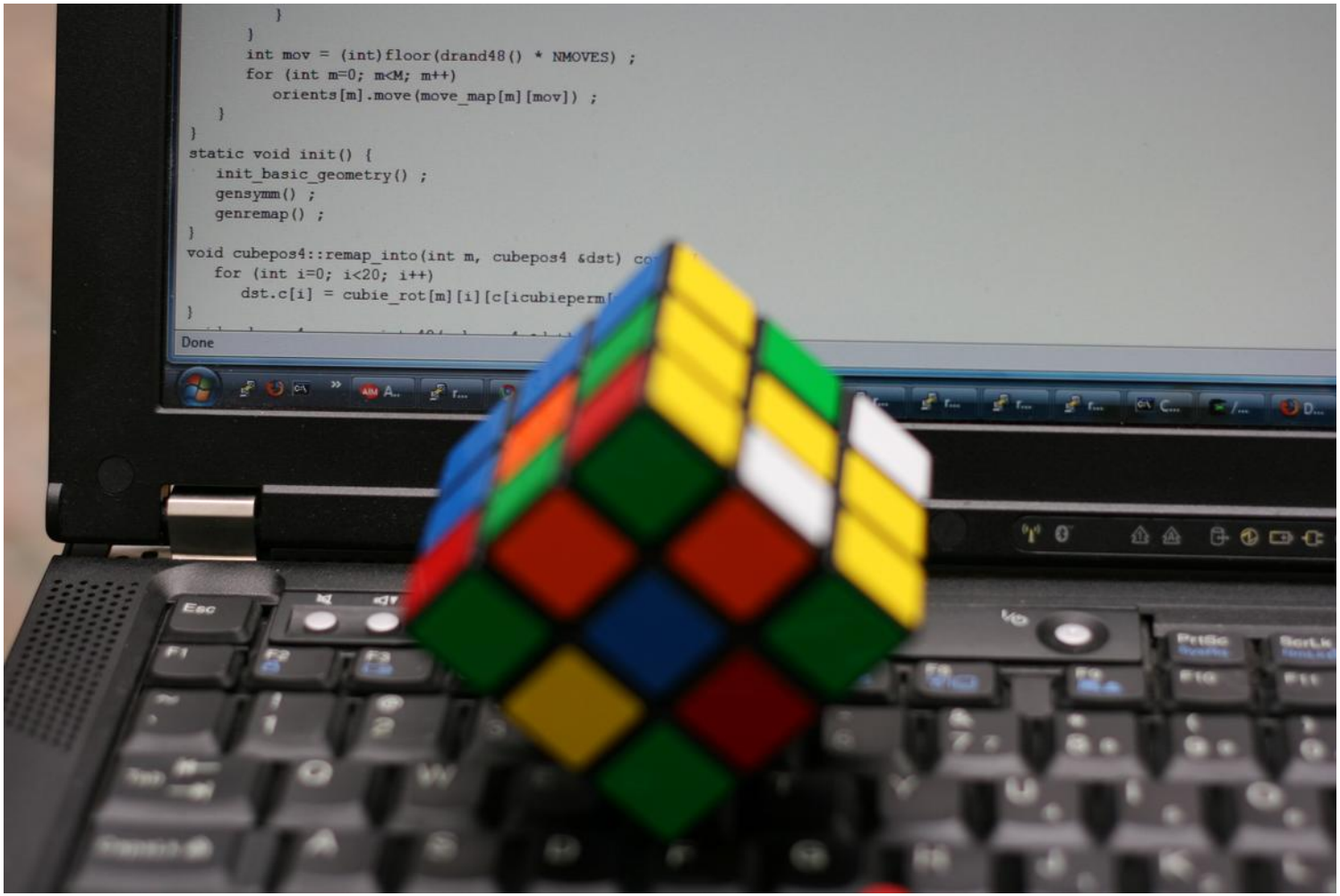
Computers and the Cube



Computer Cubing

Solving cube problems through programming:

- Graphical utilities
- Timers and practice software
- Big and impossible cube simulations
- Robotic solvers (Lego solver, Rubot)
- Scramblers
- Optimal and suboptimal solvers
- Subspace explorations

Why computer cubing?

- Widespread fascination with the cube: general public, speedsolvers, mathematicians
- Recreational computing is fun!
- Build skills; dabble with new technology
- Plenty of low-hanging fruit; gain some acclaim
- As computers get bigger and faster, problems get easier
- Rich space of questions and puzzles
- "That can't be done" is your challenge.

Distance questions

My main interest:

- 3x3 (larger puzzles are hard)
- Distance (diameter) questions (God's Number)
- Half-turn metric

Distance questions

A *position* p is some arrangement of the Rubik's cube.
A *sequence* s is some sequence of moves from some set A: $s \in A^*$
A *solution* to a position is some sequence s that, when applied to that position, results in the solved cube: $ps = 1$.
A *generator* of a position is some sequence s that yields the position when applied to the solved cube: $1s = p$.
The *distance* of a position p in moves from A is the length of the shortest sequence that solves (or generates) that position:

$$|p| = \min_{s \in A^*, ps=1} |s|$$

The *distance* of a set of positions g is the maximum distance of any position in g:

$$|g| = \max_{p \in g} |p|$$

Challenges

- Given a position, find a sequence that solves it.

Challenges

- Given a position, find a sequence that solves it.
- Find an algorithm to solve *any* given position.

Challenges

- Given a position, find a sequence that solves it.
- Find an algorithm to solve *any* given position.
- Given a position, find a *short* sequence that solves it. (Fewest Moves Challenge)

Challenges

- Given a position, find a sequence that solves it.
- Find an algorithm to solve *any* given position.
- Given a position, find a *short* sequence that solves it. (Fewest Moves Challenge)
- Find an algorithm to find a *short* solutions to any given position.

Challenges

- Given a position, find a sequence that solves it.
- Find an algorithm to solve *any* given position.
- Given a position, find a *short* sequence that solves it. (Fewest Moves Challenge)
- Find an algorithm to find a *short* solutions to any given position.
- Given a position, find its distance.

Challenges

- Given a position, find a sequence that solves it.
- Find an algorithm to solve *any* given position.
- Given a position, find a *short* sequence that solves it. (Fewest Moves Challenge)
- Find an algorithm to find a *short* solutions to any given position.
- Given a position, find its distance.
- Given a position, find a shortest (optimal) sequence that solves it.

Challenges

- Given a distance, find out how many different positions have that distance.

Challenges

- Given a distance, find out how many different positions have that distance.
- Given a group (or coset), find its distance.

Challenges

- Given a distance, find out how many different positions have that distance.
- Given a group (or coset), find its distance.
- Given a group (or coset), find the positions with the largest distance.

Challenges

- Given a distance, find out how many different positions have that distance.
- Given a group (or coset), find its distance.
- Given a group (or coset), find the positions with the largest distance.
- Find the distance of the entire cube group (God's Number).

Challenges

- Given a distance, find out how many different positions have that distance.
- Given a group (or coset), find its distance.
- Given a group (or coset), find the positions with the largest distance.
- Find the distance of the entire cube group (God's Number).
- Determine how many positions exist at each possible distance in the cube group.

Challenges

- Given a distance, find out how many different positions have that distance.
- Given a group (or coset), find its distance.
- Given a group (or coset), find the positions with the largest distance.
- Find the distance of the entire cube group (God's Number).
- Determine how many positions exist at each possible distance in the cube group.
- Find the distances of all positions in the cube group (God's Algorithm).

Before computers

How many positions at distance d?

Before computers

How many positions at distance d?
How many sequences of length d?

Before computers

How many positions at distance d?
How many sequences of length d?

$$18^d : (1, 18, 324, 5832, 104976 \ldots)$$

Before computers

How many positions at distance d?
How many sequences of length d?

$$18^d : (1, 18, 324, 5832, 104976 \ldots)$$

Never turn the same face twice:

$$18 * 15^{d-1} : (1, 18, 270, 4050, 60750, \ldots)$$

Before computers

How many positions at distance d?
How many sequences of length d?

$$18^d : (1, 18, 324, 5832, 104976 \ldots)$$

Never turn the same face twice:

$$18 * 15^{d-1} : (1, 18, 270, 4050, 60750, \ldots)$$

Can we do better?

Canonical sequences

How many positions are there really at distance 2?
$18 * 18 = 324$? No, this includes consecutive rotations of the same face.
$18 * 15 = 270$? No; why not?

Canonical sequences

How many positions are there really at distance 2?
$18 * 18 = 324$? No, this includes consecutive rotations of the same face.
$18 * 15 = 270$? No; why not?
This includes pairs of commuting moves: UD=DU. Solution: enforce an order on adjacent commuting moves. UFRDBL: U before D, F before B, R before L.
After a move of the U, F, or R face, can make $5 * 3 = 15$ following moves.
After a move of the D, B, or L face, can only make $4 * 3 = 12$ following moves.
These sequences are called *canonical* sequences.

Canonical sequences

Can we solve this?
Break a sequence up into *syllables* (Hoey '82) of commuting moves:
UF'B2RUD' becomes [U] [F'B2] [R] [UD']
After any given syllable, only $4 * 3 = 12$ length one syllables are permitted; only $2 * 3 * 3 = 18$ length two syllables are permitted.

$$
\begin{aligned}
f(0) &= 1 \\
f(1) &= 18 \\
f(2) &= 243 \\
f(n) &= 12 * f(n-1) + 18 * f(n-2) \qquad (n > 2)
\end{aligned}
$$

Canonical Sequences

The count of canonical sequences of length d ($f(d)$) is close to the count of positions at distance d ($c(d)$):

d	18^n	$15 * 18^n$	$f(d)$	$c(d)$
0	1	1	1	1
1	18	18	18	18
2	324	270	243	243
3	5,832	4,050	3,240	3,240
4	104,976	60,750	43,254	43,239
5	1,889,568	911,250	577,368	574,908
6	34,012,224	13,668,750	7,706,988	7,618,438
7	612,220,032	205,031,250	102,876,480	100,803,036
8	11,019,960,576	3,075,468,750	1,373,243,544	1,332,343,288
9	198,359,290,368	46,132,031,250	18,330,699,168	17,596,479,795
10	3,570,467,226,624	691,980,468,750	244,686,773,808	232,248,063,316
11	64,268,410,079,232	10,379,707,031,250	3,266,193,870,720	3,063,288,809,012

Canonical Sequences

Is there a closed form for $f(d)$?

Canonical Sequences

Is there a closed form for $f(d)$?
Yes, using the approximation of $\pi \approx \sqrt{2} + \sqrt{3} = r$:

$$f(n) = \frac{\sqrt{3}}{4}\sqrt{18^n}(r^{n+1} - (-r)^{-n-1}) \qquad (n > 0)$$

This formula returns an integer for $n > 0$.

Canonical Sequences

Canonical sequences are very useful in computer cubing!

- Count of canonical sequences is upper bound on, and close to, count of positions at that distance; allows us to efficiently search positions.
- Easy to keep track of allowable next moves (just remember the previous move).
- Generating canonical positions in depth-first search is a core theme of many cube search programs.

Size of Cube Space

- Corner twists: $3^7 = 2,187$
- Corner permutations: $8! = 40,320$
- Edge flip: $2^{11} = 2,048$
- Edge permutations: $12! = 479,001,600$
- Edge/corner permutation match: $1/2$

Total: $|G| = 43,252,003,274,489,856,000$

- At $1\mu s$ per, that's 1.37 million years
- At 1 bit per, that's 5.4 million terabyte drives

Bounds on God's Number: Early 80's

$$f(17) = 18,476,969,736,848,122,368 < |G|$$

$$f(18) = 246,639,261,965,462,754,048 > |G|$$

So God's number is at least 18.
Upper bound? Human algorithms typically take more than 100 moves worst case. Even in 1979, there was a better bound.

Thistlethewaite's algorithm

Even before the cube came to American shores, computers were being used to solve it. Moren B. Thistlethwaite found an algorithm guaranteed to solve the cube in 52 or fewer moves during the 70's.
This algorithm is based on group theory but can also explained in non-mathematical terms. Essentially:

- Phase 1 fixes the edge flip.
- Phase 2 fixes the corner flip and puts the middle edges in the middle layer.
- Phase 3 puts the corners into their tetrads, the edges into their slices, and puts the corner permutations into the squares group.
- Phase 4 solves the cube.

These steps can be visualized through a restickering of the cube.

Phase 1: Edge flip

Remove all stickers from the corners.
Replace D stickers with U, and remove FRBL stickers form top and bottom cube layers.
Remove LR stickers from middle layer and replace FB stickers with U.
Now all edge cubies are identical and have a single U sticker on them.
This is an *edge flip* cube.
State space is 2048.
Note that the solved state is not affected by U, F2, R, D, B2, L.
Phase one: solve the "edge flip". Restrict further stages to the moves above.

Mathematics of Phase 1

Cube group: $G_0 = \{U, F, R, D, B, L\}$; size=43,252,003,274,489,856,000
Subgroup: $G_1 = \{U, F2, R, D, B2, L\}$; size=21,119,142,223,872,000
Cosets space size: 2,048
Coset space diameter: 7

Phase 2: Corner twist

Remove all FRBL stickers from the corners.
Now all corners are identical.
This is a *corner twist* cube.
State space is 2187.
Note that the solved state is not affected by U, F2, R2, D, B2, L2.
Unfortunately, those moves are not sufficient to solve all cubes that have a correct edge flip and corner twist; they cannot move edges out of the middle layer.

Phase 2: Place middle edges in middle layer

Remove FRBL stickers from top two layer edges; replace D stickers with U, B stickers with F, and L stickers with R (three color cube).
All middle edges are indistinguishable; all top/bottom cubies are indistinguishable.
Solve the resulting state (assume edge flip fixed already from phase 1).
State space is $\binom{12}{4} = 495$.
Note that the solved state is not affected by U, F2, R2, D, B2, L2.

Phase 2: Corner twist and middle edges

Phase 2 is a combination of fixing the corner twist, and getting the middle edges in place.
End of phase 1: $G_1 = \{U, F2, R, D, B2, L\}$; size=21,119,142,223,872,000
End of phase 2: $G_2 = \{U, F2, R2, D, B2, L2\}$: size=19,508,428.800
Cosets space size: $2,187 * 495 = 1,082,565$
Thistlethwaite found a solution to this coset space in 13 moves.
Real diameter is 10.

Phase 3: Getting into the squares group

With the edge flip and corner twist solved, only the cubie permutations remain.
Restickering: replace all D stickers with U, B stickers with F, and L stickers with R.
Two different corner cubie colors: UFR and URF.
Three different edge cubie colors: UF, FR, and RU.
Solve to single-colored faces.
Result unaffected by U2, F2, R2, D2, B2, L2 (square moves).
With twists and flips solved, and middle edges already placed, this has a state space of $\binom{8}{4}^2$ or 4900.
This does not quite work, though; there is a further restriction on the permutations attainable within each corner tetrad in the squares group.

Phase 3: Getting into the squares group

Phase 3 is the process of getting to the squares group: edges in the appropriate slice, corners in the appropriate tetrad, and into a "solvable" corner permutation position.

End of phase 2: $G_2 = \{U, F2, R2, D, B2, L2\}$: size=19,508,428.800

End of phase 3: $G_3 = \{U2, F2, R2, D2, B2, L2\}$: size=663,552

Cosets space size: $\binom{8}{4}^2 * 6 = 29,400$

Thistlethwaite found a solution to this coset space in 15 moves.

Real diameter is 13.

Phase 4: Solving the cube.

Phase 4 solves the permutations of each edge slice and each corner tetrad.
End of phase 3: $G_3 = \{U2, F2, R2, D2, B2, L2\}$
Group size: $24^5/2/6 = 663,552$
Thistlethwaite found a solution to this group in 17 moves.
Real diameter is 15.

Thistlethwaite's Algorithm

$G_0 = \{U, F, R, D, B, L\}$; size=43,252,003,274,489,856,000
$|G_0 \backslash G_1| = 2,048$
$G_1 = \{U, F2, R, D, B2, L\}$; size=21,119,142,223,872,000
$|G_1 \backslash G_2| = 1,082,565$
$G_2 = \{U, F2, R2, D, B2, L2\}$: size=19,508,428,800
$|G_2 \backslash G_3| = 29,400$
$G_3 = \{U2, F2, R2, D2, B2, L2\}$: size=663,552
$|G_3| = 663,552$.
Thistlethwaite found solutions of lengths 7, 13, 15, and 17, for a total of 52 moves.
This provided the best upper bound to God's number for a long time.
Later improved to 50, and then to 45, through computer searches; the actual diameters are 7, 10, 13, and 15.

Programming the cube

The late 80's were not very productive in computer cubing. We will take advantage of the break in the historical timeline to discuss how one might code a computer program.

- Representation of the cube
- Search
- Tables

Facelet representation

Number each facelet; encode each move by a facelet permutation.

```
                1  2  3
                4  5  6
                7  8  9

10 11 12   13 14 15   16 17 18   19 20 21
21 22 23   24 25 26   27 28 29   30 31 32
33 34 35   36 37 38   39 40 41   42 43 44

               45 46 47
               48 49 50
               50 51 52
```

- Do not need center facelets, except for whole cube moves.
- Each move affects 20 facelets: tedious but finite.
- Requires 54 (48) bytes storage.

Cubie representation

Number the cubies and the slots they go in:

```
            Top          Middle          Bottom
           0   1                           4   5
Corners
           2   3                           6   7

             0              4   5              8
Edges      1   2                          9   10
             3              6   7             11
```

Slot to cubie representation

```
struct cornerslot {
    int cubie ; // 0..7
    int twist ; // 0..2
} ;
struct edgeslot {
    int cubie ; // 0..11
    int flip ; // 0..1
} ;
struct cubeslots {
    cornerslot corners[8] ;
    edgeslot edges[12] ;
} ;
```

Slot to cubie representation: moves

```
cubeslots::moveU() {
    cornerslot c = corners[0] ;
    corners[0] = corners[2] ;
    corners[2] = corners[3] ;
    corners[3] = c ;
    edgeslot e = edges[0] ;
    edges[0] = edges[1] ;
    edges[1] = edges[3] ;
    edges[3] = e ;
}
```

Twist moves must also update twists.

Cubie to slot representation

```
struct cornercubie {
    int slot ; // 0..7
    int twist ; // 0..2
} ;
struct edgecubie {
    int slot ; // 0..11
    int flip ; // 0..1
} ;
struct cubecubies {
    cornercubie corners[8] ;
    edgecubie edges[12] ;
} ;
```

Cubie to slot representation: moves

```
int cornerslotmove[8][18] = { ... } ; fill in
int edgeslotmove[12][18] = { ... }
cubeslots::moveU() {
    for (int i=0; i<8; i++)
        corners[i].slot = cornerslotmove[corners[i].slot][U] ;
    for (int i=0; i<12; i++)
        edges[i].slot = edgeslotmove[edges[i].slot][U] ;
}
// Twist moves must also update twists
```

Calculating the inverse of a position

Calculating the inverse of a position

- Converting from cubie-to-slot to slot-to-cubie is a simple loop

Calculating the inverse of a position

- Converting from cubie-to-slot to slot-to-cubie is a simple loop
- Converting from slot-to-cubie to cubie-to-slot is the identical loop

Calculating the inverse of a position

- Converting from cubie-to-slot to slot-to-cubie is a simple loop
- Converting from slot-to-cubie to cubie-to-slot is the identical loop
- Evaluating the inverse in either format is again, the same loop

Cubie optimizations

- All cubie or slot structures can be represented by a byte with a value of 0..23.
- Total size of representation: 20 bytes.
- Small lookup tables can be used to perform each move and twist at the same time.

Coordinate representation

For top speed in some problems, it's best to split the cube representation into as few simple objects as possible, especially when not dealing with full permutation information.
Example: Phase 1 of Thistlethwaite's algorithm: only edge flip matters.
Can use a single integer 0..2047, one bit per edge (omitting the final parity bit).
Moves then can be done with a single small lookup table very quickly.

Your first cube program

- Write it the simplest possible way you can think of! No thoughts to speed whatsoever.
- Read and write standard representations (Singmaster for positions, standard move notation for sequences) for interoperability.
- Do not optimize.
- If you must optimize, do not optimize yet.
- Simple elegant algorithms trump complicated convoluted optimizations.
- As you build faster and more optimized versions, always test them against each other.
- Get a friend to implement your idea—and check programs against each other.
- "Retracting" a paper/result is much harder than publishing one!

Search strategies

Searching the space of a group or a coset space is an example problem we may want to solve.
We can represent either a group or a coset space as a graph, with positions as nodes and moves as edges.
We use graph algorithms to search such graphs; these include

- Depth-first search
- Iterated depth-first search
- Breadth-first search

Limited depth depth-first search

```
boolean dfs(cubepos cp, int togo) {
   if (togo == 0)
      return (cp == identity) ;
   for (m : moves)
      if (dfs(cp.move(m), togo-1))
         return true ;
   return false ;
}
```

In this particular version, we return true if the cube has a solution at that depth, otherwise we return false.
Note that this is not a normal depth-first search where we record each position as we go, and terminate the recursion when we visit the same position again. Instead we depth-limit the search.
Typically you will want to embed our "canonical sequences" ideas here to reduce the identical positions explored.

Iterated depth-first search

```
int depth(cubepos cp) {
   for (int d=0; ; d++)
      if (dfs(cp, d))
         return d ;
}
```

We simply try each depth one by one until we find one that solves the cube.
Note that we do not ever explicitly represent the graph; these are called "implicit graph searches" because we are searching the graph that is implied by the move function, rather than an explicit graph that lives in memory.

Breadth-first search

Breadth-first search is useful when the whole graph, or at least that part we are interested in, fits in memory, and we do not want to visit any state more than once.

```
int bfs(cubepos cp) {
   map<cubepos, int> seen ;
   queue<cubepos> q ;
   q.add(cp) ;
   seen[cp] = 0 ;
   while (!q.empty()) {
      cp = q.removefront() ;
      for (m : moves) {
         cubepos cp2 = cp.move(m) ;
         if (cp2 == identity)
            return seen[cp] + 1 ;
         if (seen[cp2] == null) {
            seen[cp2] = seen[cp] + 1 ;
            q.addback(cp2) ;
         }
      }
```

Memory-efficient breadth-first search

There is a variation of breadth-first search that works well when a hash table or list of all states would be too large to fit into memory, but two-bits-per-state fits into memory. This works by storing the distance mod three in those two bits, and reserving a value (3) for unexplored. The algorithm looks like:

```
seen[startstate] = 0 ;
states = 0 ;
for (int d=1; states < statesize; d++) {
   for (s : states)
      if ((seen[s] + 1) % 3 == d % 3)
         for (m : moves)
            if (seen[move(s, m)] == 3) {
               seen[move(s, m)] = d % 3 ;
               states++ ;
            }
}
```

Memory-efficient breadth-first search

Despite the fact that this code expands states multiple times, it can be significantly faster than "normal" breadth-first search because of the efficiency of its operations.

Double-ended breadth-first search

For many search problems, as in a general cube solver, depth-first search runs out of time, and breadth-first search runs out of space.
If we are looking for a solution for a single position, double-ended search can be effective.
Double-ended search is the same as breadth-first search, but instead of starting with only the start position in the queue (at depth 0), we also insert the goal position (putting it at a huge depth, say, 1000). Then we proceed searching not only from the start position but also from the goal position, until we encounter a state that is reachable from both.
To solve a depth-10 position using standard bfs or dfs would require us to examine about 100,000,000,000 states. To do the same with double-ended search requires us to examine only about 500,000 states.

Kloosterman's Investigations

Computer cubing started up with a vengeance again in the early 1990's, so we return to our historical timeline.
In 1992, Hans Kloosterman replaced G_3 with a different subgroup, the subgroup of G_2 such that all U cubies are in the U face and all D cubies are in the D face (and of course, all flips and twists are corrected). He found solutions in (7, 10, 8, and 18) moves for a new bound on God's algorithm of 43 moves.
He also found a way to always *remove* a move between stages 3 and 4, to prove a new bound of 42 moves.

Kociemba shocks the world

In 1992, Herbert Kociemba implemented the famous *Two Phase* algorithm that forms the heart of his current Cube Explorer program. This amazing algorithm would easily find much shorter solutions to almost any cube position than any program written before it. With this program, shorter solutions to many difficult positions were quickly found. Indeed, no one could find a position that this algorithm could not quickly solve in 22 moves or less.

Two-Phase Algorithm

Kociemba's algorithm was based on the same essential ideas as Thistlethwaite's, but with two major additions:

- Combine phases 1 and 2 into a single phase, and phases 3 and 4 into a single phase.
- Instead of finding a single solution in each phase, instead find many solutions in phase 1 and evaluate each in phase 2 to find shorter and shorter solutions.

Since Kociemba's algorithm only had two phases, it is known as the "Two Phase" algorithm. The target group of the first phase is the same as Thistlethwaite's G_2:

$$H = \{U, F2, R2, D, B2, L2\}$$

The size of the coset space from G (the full cube group) to H is

$$|G \backslash H| = 2,217,093,120$$

The size of the group H itself is:

$$|H| = 19,508,428,800$$

Two Phase Algorithm

Each phase of Kociemba's algorithm is too large to easily explore using standard dfs or bfs. Even double-ended search was too much for the memory of computers of the time.
Kociemba used a refinement of double-ended search to make the problem tractable. Rather than search from the goal state each time, he computed a large table that contained the results of a breadth-first search from the goal state, and stored it on disk. This was then re-read every time the program was run.

Pruning tables

This pruning table approach of Kociemba's was made much more effective through two important techniques.

- Rather than storing key→value pairs in a hashtable, he computed an index for each position, and used a simple array. Where two positions shared the same index, the smaller distance was entered into the hash table. Thus his pruning tables were conservative, rather than exact.
- He combined positions that were related by symmetry into a single canonical symmetry representative, and only put those positons in the table. This enabled a much smaller table to represent many more positions.

So his pruning tables took an existing position and returned a lower bound on the number of moves to the goal state.

Depth-first search with pruning tables

```
boolean dfs(cubepos cp, int togo) {
    if (depth_estimate[cp] > togo)
        return false ;
    for (m : moves)
        if (dfs(cp.move(m), togo-1))
            return true ;
    return false ;
}
```

This is a general depth-first search using pruning tables. The "canonical sequence" code should be added to this to make it more effective.

He used depth-first search twice, once for phase one, and once for phase two. On every successful phase one solution, he would invoke the phase two depth-first search algorithm see if a solution better than the best one known would be found.

```
int best = 1000000 ;
int p1d = 0 ;
while (true) {
   dfsp1(cp, p1d) ;
   p1d++ ;
}
boolean dfsp2(cubepos cp, int togo) {
   if (depth_estimatep2[cp] > togo)
      return ;
   ...
}
void dfsp1(cubepos cp, int togo) {
   if (togo == 0 && in_H(cp))
      dfsp2(cp, best-p1d) ;
   ...
```

Kociemba's Algorithm

In practice, Kociemba's algorithm returns a solution very quickly, but it may be of length 25 or longer. But very quickly, the phase one distance gets high enough (14 or 15) that very few of the positions reached by the phase one search actually need consideration in phase two, since the pruning table will directly give proof that the overall solution length would be too long. Since the phase one algorithm can generate phase one solutions extremely quickly, shorter and shorter solutions are found.
If you allow the Two Phase algorithm to continue, eventually the phase one length will grow to the length of the best sequence found, at which point you will have an optimal solution (depending on whether certain other optimizations have been applied). But this takes too long for it to be a pratical optimal solver.

Michael Reid

In May of 1992, Michael Reid burst onto the scene. Using a three phase algorithm (with two subgroups) and exhaustive analysis, he was able to lower the bound on God's number to 39 (the previous value was 42). He performed this feat with a three-phase algorithm using the intermediate groups $\{U, R, F\}$ and $\{U, R2, F2\}$, and a computer analysis of the groups and their coset spaces.

Dik Winter

Exactly one day later, Dik Winter presented his analysis of the coset space of phase one of Kociemba's algorithm that reduced the maximum distance bound to 12; this in conjunction with the earlier Kloosterman result of 25 for the last two phases of the Thistlethwaite algorithm gave a new lower bound of 37.
An error was found in Winter's work, but he had it corrected in a few days; Reid had held the bound on God's number for less than a week!

Jerry Bryan

In July of 1994, Jerry Bryan posted the count of cube positions up to a depth of 7:

n	c(n)
0	1
1	18
2	243
3	3,240
4	43,239
5	574,908
6	7,618,438
7	100,803,036

This was the first large computer search of the full cube space.

Michael Reid

In January of 1995, Michael Reid stole back the bound on God's algorithm, smashing it from 37 all the way down to 29! He accomplished this with an exhaustive computer analysis of both phase one and phase two of Kociemba's algorithm.
His analysis showed the phase one coset space had a maximum distance of 12, and the phase two group had a maximum distance of 18. Further he showed how to "save a move" in the wost cases, yielding a brand new bound on God's number of 29. This time, rather than hold the bound for less than a week, he would hold it for more than a decade.
Also in 1995, Michael Reid showed that the position *superflip* required 20 moves, thus setting a lower bound on God's number.

Richard Korf and "Pattern Databases"

To this date, no one had written a practical solver for the cube that would return a sequence with the minimum number of moves. In May of 1997, Richard Korf presented exactly such a program, to world-wide popular acclaim.
The technique he used was depth-first search with pruning tables. Neither technique was new, but his selection of the pruning tables to use, together with his programming abilities and faster machines finally made the problem reasonably tractable. Each cube took from several hours to several days to solve. His paper included the solutions to ten random cubes: 1 at distance 16, 3 at distance 17, and 6 at distance 18.
Numerous optimal solvers followed with various tradeoffs in memory space, pruning tables selected, and various optimizations.

Further progress in counting positions

In January of 1998, Jerry Bryan announced his results for the count of cube positions at a depth of 9 (1,332,343,288). In July the same year, he pushed that out to a depth of 10 (17,596,479,795). Both of these results derived from a technique where he stored a tree of all positions to depth 5, and then combined this tree with itself in an intricate way that generated all product positions in lexicographical order. This allowed him to scan and remove duplicates without ever storing the billions of positions in memory or on disk.

Silviu Radu's results

Computer cubing took a break for nearly half a decade, with few new results. The next major contributor, Silviu Radu, through computer search and some careful insight, extended Reid's work to decrease the bound on God's number to 28 (in December of 2005) and then to 27 (in April of 2006).
Perhaps more impressive was the work he presented in July 2006, where he calculated an optimal solution to all 164,604,041,664 symmetric positions of the cube. Notable was the fact that though there were 1,091,994 distance-20 positions, there was not a single position that took more than 20 moves.
I think this may be the most impressive computer cubing feat ever.
These results were made possible by a new technique he and I had been playing with for some time: coset solvers.

Coset Solvers

At this point, there were position-at-a-time optimal and near-optimal solvers. They typically took from milliseconds (for near-optimal) to hours (for optimal, some positions) for each position. Clearly this was still much too slow.

In addition, for groups and coset spaces small enough to fit into memory, there were exhaustive solvers; they usually worked by some efficient version of breadth-first search (as we showed earlier). But it will be a long time until we can fit 4e19 states in memory.

Is there a way to build a solver intermediate between the two?

Coset Solver

Yes, indeed there is, and quite effectively too.
Consider the edges group of the cube. There are about one trillion positions; with symmetry reductions and other tricks, we can build a full exact pruning table for this group using two gigabytes of RAM. This makes it very easy for us to solve any edges position nearly instantaneously (in tens of microseconds).
Now, for any given edges position, there are many full cube positions with the edges in that state (specifically, cosets of the edges-fixed group)—indeed, there are 44,089,920 such full cube positions.
In addition, for every given edges positions, there are many sequences that solve that edges positions—and our large pruning table enables us to enumerate them quickly.

Coset Solver

Every sequence that solves a given edges position puts the corners into some position. If we want to solve all 44 million corners positions that share that same edge position, we need to:

- Enumerate solutions to the edges positions in order of increasing length, and
- Mark off seen corner positions in a bitmap so we know when we've found the first solution to the full cube position.

Putting these two tricks together, and you have a single program that can solve 44 million positions at a time, extremely quickly.
When there are only a handful of positions left, those positions should then be handled by a separate optimal solver.
This technique allows us to find optimal solutions to positions within a coset at a rate of thousands per second (or even faster, much faster).

Faster Computers

In November of 2006, Jerry Bryan announced the count of positions at distance 11 (3,063,288,809,012); this took three months on his PC. Coset techniques were about to make a huge impact.

Kunkle and Cooperman: 26

In August of 2007, Kunkle and Cooperman announced they had lowered the bound on God's number to 26. This hit major media outlets. Their technique was a two-phase approach, using a large cluster of machines and many disks in parallel to gain the bandwidth required.
An error was found in their approach, but it was corrected and their result held.
Their approach combined the first three phases of the Thistlethwaite algorithm into one huge coset space exploration, along with a number of smaller squares group coset explorations.

Kociemba-group coset solver

From 2004 to about 2006 off and on, Silviu and I had been (separately) writing, honing, testing, and sharing results from Kociemba-group coset solvers. These solvers use the Kociemba group as an intermediate group, thus solving 19,508,428,800 cube positions at a given time. In addition, they have the following advantages over (for instance) a corners-group coset solver:

- The search phase is much faster because the terminal moves that bring a position into the group always occur in pairs—you get two positions for the effort of one.
- Phase one sequences that end in the moves that are in the group can be computed separately, and extremely quickly, with a simple quick pass through memory and some bitwise operations; this reduces the search by another factor of two.
- When you do not need optimal solutions for all sequences, but rather just a bound on the distance of the coset, you can terminate the search phase early (say, after a depth of 16) and just use the trick from the previous item to finish out the coset.

Coset space graph

By using the Kociemba group as the target group, not only do we solve an enormous (19,508,428,800) number of positions at a time, but the coset space graph itself (with only about 138 million nodes once it is reduced by symmetry) is small enough to fit in memory and be manipulated efficiently. Because of that, my approach for lowering God's number was just to solve thousands and thousands of cosets, enough so that the maximum coset distance was shown to be below a given number. That is, I did not simply focus on the furthest cosets and try to reduce them, but instead solved cosets throughout the graph, each solved coset bring more and more nodes in the coset space below the current bound.

Kociemba-group coset solver results

This program has shown the following results:

- December 2007: confirmed Kunkle and Cooperman's 26 bound.
- March 2008: Using only my home PCs, lowered the bound on God's number to 25.
- April 2008: Using idle time on the Sony Pictures Imageworks computers, lowered the bound on God's number to 23.
- August 2008: Using more idle time, lowered the bound on God's number to 22. This required the solution of 1.28 million cosets, each of size 19.5B positions; overall this program has found solutions of length 20 or less to more than 3% of cube space.

Next Goals

At this point, the program can find optimal solutions to positions on a single i7-920 CPU at a rate of 250ns each, or four million a second. It can find near-optimal (distance of 20 or less) solutions to positions at a rate of 3ns each (330 million a second).
We intend to continue refining this program until multi-core, 64-bit computers with more than 4 GB of RAM are commonplace. Then we will probably use a distributed computing infrastructure to finally prove God's number is 20.

Other Coset Solver Results

Using the corners group of the cube, we wrote a coset solver that enumerated the positions at a given depth. Within a day of writing the program, we had set new records on discovering the count of positions at a given depth.

- In June 2009, we showed there are 40,374,425,656,248 positions at depth 12.
- In July 2009, we showed there are 531,653,418,284,628 positions at depth 13. This is about one in every 80,000 positions.

Low hanging fruit

With today's machines and the techniques that have been developed, there are many opportunities for making a mark in computer cubing.

- All symmetric positions in the QTM need to be solved.
- I am sure we can push the count of cubes at a given depth to 14f* and 16q* fairly easily.
- God's number in the QTM can probably be reduced (using these same techniques, I have already reduced it from 34 down to 29).
- How many distance 20f* positions can you find? Is there a better way to find them?
- How many distance 25q* and 26q* positions are there? I conjecture there are only 3 25q* and only 1 26q*.
- How many moves does it take to scramble a 4x4? A pyraminx? Is there a reasonable compromise between quality of scramble and difficulty of setting up the puzzles for competition?

Low hanging fruit

- Solve the combination of Thistlethwaite's first three phases.
- Solve the combination of Thistlethwaite's last three phases.
- Write a solver that uses a different metric, such as one customized for Rubot. For the Lego solver. For human fingers.

In this, as in so much else, the ideas are out there. It is the sweat, the implementation, the testing, the experimentation, that gets the results.

Acknowledgements

All of the work I have done was inspired by so many others, too many to count. But I have received direct assistance and had many fruitful discussions with both Silviu Radu and Herbert Kociemba. They truly have the shoulders of giants.