

1. Introduction. This is a coset solver for Rubik's Cube, using the cosets generated by the group $\{U, F2, R2, D, B2, L2\}$. If you have not read `cubepos`, `kocsymm`, and `phase1prune`, you should read those before you try to understand this program.

```

⟨hcoset.cpp 1⟩ ≡
    const char *BANNER = "This is hcoset 1.1, (C) 2010-2012, Tomas Rokicki. All Right\
        s Reserved.";
#include "phase1prune.h"
#include <pthread.h>
#include <iostream>
#include <map>
#include <set>
#include <cstdio>
using namespace std;
⟨Data declarations 2⟩
⟨Utility functions 13⟩
⟨Threading objects 48⟩
void docoset(int seq, const char *movestring)
{
    ⟨Handle one coset from movestring 8⟩
}
int main(int argc, char *argv[])
{
    double progstart = walltime();
    duration();
    ⟨Parse arguments 3⟩
    ⟨Initialize the program 7⟩
    ⟨Handle the work 81⟩
    ⟨Cleanup 54⟩
    phase1prune::check_integrity();
    cout << "Completed in " << (walltime() - progstart) << endl;
}

```

2. The first thing we take up is argument parsing. Two arguments we know we need up front include a verbosity level (the default is 1, but the `-q` option makes it 0 and the `-v` option makes it 2), and a thread count.

```

⟨Data declarations 2⟩ ≡
    int verbose = 1;
    int numthreads = 1;
    const int MAX_THREADS = 32;

```

See also sections 4, 6, 10, 15, 17, 19, 21, 23, 29, 32, 35, 39, 43, 44, 49, 56, 61, 66, 75, 76, 79, and 82.

This code is used in section 1.

3. Parsing the arguments is boilerplate code.

```

⟨Parse arguments 3⟩ ≡
  int oargc = argc;
  char **oargv = argv;
  while (argc > 1 ∧ argv[1][0] ≡ '-') {
    argc--;
    argv++;
    switch (argv[0][1]) {
    case 'v': verbose++;
      break;
    case 'q': verbose = 0;
      break;
    case 't':
      if (argc < 2)
        error ("!not_enough_arguments_to_t");
      if (sscanf(argv[1], "%d", &numthreads) ≠ 1)
        error ("!bad_thread_count_argument");
      if (numthreads < 1 ∨ numthreads > MAX_THREADS)
        error ("!bad_value_for_thread_count");
      argc--;
      argv++;
      break;
    ⟨More arguments 5⟩
    default:
      error ("!bad_argument");
    }
  }

```

This code is used in section 1.

4. Usually the pruning tables are read from disk; if they don't exist, they are created, and then written to disk. If for some reason you do not want to write the pruning tables to disk, you can use the `-W` option to inhibit this.

```

⟨Data declarations 2⟩ +≡
  int skipwrite = 0;

```

5. Parsing this argument is easy.

```

⟨More arguments 5⟩ ≡
  case 'W': skipwrite++;
  break;

```

See also sections 11, 30, 53, 67, and 77.

This code is used in section 3.

6. This coset solver works by finding as many phase 1 solutions to a particular group position as it can; each phase 1 solution leads to a particular element of the coset. As soon as we have discovered all possible positions, we are done.

We start by writing the simplest possible coset solver; we can use this to verify results from faster, more optimized solvers. The simplest possible solver just keeps the element information in a `set`.

```

(Data declarations 2) +=
  moveseq repseq;
  set<permcube> world;
  kocsymm repkc;
  permcube reppc;
  cubepos repcp;

```

7. We always start by printing the arguments and the banner.

```

(Initialize the program 7) ≡
  if (verbose) cout << BANNER << endl << flush;
  for (int i = 0; i < oargc; i++) cout << " " << oargv[i];
  cout << endl;

```

See also sections 16, 22, 31, 33, 34, 36, 40, 47, 50, and 84.

This code is used in section 1.

8. Handling a coset starts by parsing the move string. Note that `phase1prune` is automatically protected against multiple initialization.

```

(Handle one coset from movestring 8) ≡
  int oldsingcount = singcount;
  const char *tmp = movestring;
  repseq = cubepos::parse_moveseq(tmp);
  if (*tmp)
    error ("!_extra_stuff_at_end_of_input_moveseq");
  cout << "Coset representative " << seq << " " << movestring << endl;
  phase1prune::init(skipwrite);
  double cosetstart = walltime();

```

See also sections 9, 12, 20, 26, 71, and 74.

This code is used in section 1.

9. Next, we execute the move sequence on our representative `kocsymm`.

```

(Handle one coset from movestring 8) +=
  repkc = identity_kc;
  reppc = identity_pc;
  repcp = identity_cube;
  for (unsigned int i = 0; i < repseq.size(); i++) {
    repkc.move(repseq[i]);
    reppc.move(repseq[i]);
    repcp.move(repseq[i]);
  }
#ifdef LEVELCOUNTS
  setup_levmul(repkc, repseq);
#endif

```

10. We want to support different levels of slowness. Level 0 is the slowest; level 1 the next fastest, and level 2, the default, is the fastest. In addition, we want to limit the maximum level to search with an option; this gives us a program that actually terminates. We also have a separate overall maximum level, which we will discuss later. We also maintain a global variable that indicates the depth we are currently exploring; this is only used by the prepass.

```
⟨Data declarations 2⟩ +≡
  int slow = 2;
  int maxsearchdepth = 35;
  int maxdepth = 35;
  int global_depth;
```

11. We use the `-s` option to set the slowness.

```
⟨More arguments 5⟩ +≡
case 's':
  if (argc < 2)
    error ("!not_enough_arguments_to_s");
  if (sscanf(argv[1], "%d", &slow) ≠ 1)
    error ("!bad_s_argument");
  if (slow < 0 ∨ slow > 2)
    error ("!bad_value_for_s");
  argc--;
  argv++;
  break;
case 'd':
  if (argc < 2)
    error ("!not_enough_arguments_to_d");
  if (sscanf(argv[1], "%d", &maxdepth) ≠ 1)
    error ("!bad_d_argument");
  if (maxdepth < 0)
    error ("!bad_value_for_d");
  if (maxdepth < maxsearchdepth) maxsearchdepth = maxdepth;
  argc--;
  argv++;
  break;
case 'S':
  if (argc < 2)
    error ("!not_enough_arguments_to_S");
  if (sscanf(argv[1], "%d", &maxsearchdepth) ≠ 1)
    error ("!bad_S_argument");
  if (maxsearchdepth < 0)
    error ("!bad_value_for_S");
  argc--;
  argv++;
  break;
```

12. If the `slow` value is zero, we use the slowest search.

```
⟨Handle one coset from movestring 8⟩ +≡
  if (slow ≡ 0) slowsearch1(repkc, reppc);
```

13. The slow search routine is much like the one in the two-phase solver.

```

<Utility functions 13> ≡
void slowsearch1(const kocsymm &kc, const permcube &pc, int togo, int movemask, int canon)
{
    if (togo == 0) {
        if (kc == identity_kc) {
            probes++;
            world.insert(pc);
        }
        return;
    }
    togo--;
    kocsymm kc2;
    permcube pc2;
    int newmovemask;
    while (movemask) {
        int mv = ffs(movemask) - 1;
        movemask &= movemask - 1;
        kc2 = kc;
        kc2.move(mv);
        int nd = phase1prune::lookup(kc2, togo, newmovemask);
        if (nd ≤ togo) {
            pc2 = pc;
            pc2.move(mv);
            int new_canon = cubepos::next_cs(canon, mv);
            slowsearch1(kc2, pc2, togo, newmovemask & cubepos::cs_mask(new_canon), new_canon);
        }
    }
}

```

See also sections 14, 18, 24, 25, 27, 28, 37, 38, 41, 42, 45, 46, 51, 52, 55, 57, 58, 59, 60, 62, 69, 78, 80, 83, and 85.

This code is used in section 1.

14. Finally, we do the search. We continue forever in this slow mode (there's no way it will ever finish, with just this logic). We put the slow search logic in a subroutine.

```

<Utility functions 13> +≡
void slowsearch1(const kocsymm &kc, const permcube &pc)
{
    duration();
    for (int d = phase1prune::lookup(repkc); d < maxsearchdepth; d++) {
        probes = 0;
        long long prevlev = uniq;
        slowsearch1(kc, pc, d, ALLMOVEMASK, CANONSEQSTART);
        uniq = world.size();
        long long thislev = uniq - prevlev;
        if (verbose) cout << "Tests_at_" << d << "_" << probes << "_in_" << duration() << "_uniq_" <<
            uniq << "_lev_" << thislev << endl;
    }
}

```

15. Congratulations! At this point we have a coset solver!

The slow search runs slowly, but more importantly, it typically runs out of memory rather quickly. To solve this, rather than using a **set**, we can use a **vector** and sort it to calculate uniqueness. But this would not make more than a factor of three improvement, probably, so we go ahead and take the next step—a big bitmap, one bit per element of the coset. With this data structure, memory usage, though large, is fixed.

The size of these cosets is 19,508,428,800 elements, so the bitmap needs to be about 2.5GB in size. On a 32-bit platform, it is unlikely we will be able to allocate that much memory contiguously. Furthermore, because of some optimizations we will make a little later, we will want two bitmaps.

The bitmaps will be indexed by corner permutation and edge permutation. The corner permutation has a smaller range of $8!$, so we use it to select a memory page. Each memory page must handle all possible edge permutations; they have a range of $8!4!/2$ (the division by two is because the corner and edge permutations must match).

```
<Data declarations 2> +=
  const int FACT8 = 40320;
  const int PAGESIZE = (FACT8 * FACT4/2/8);
  unsigned char **bitp1, **bitp2;
```

16. We must allocate these arrays at the start.

```
<Initialize the program 7> +=
  bitp1 = (unsigned char **) calloc(FACT8, sizeof(unsigned char *));
  bitp2 = (unsigned char **) calloc(FACT8, sizeof(unsigned char *));
  if (bitp1 == 0 || bitp2 == 0)
    error ("!_no_memory" );
```

17. When we set a bit in the large bitmap, we need to ensure that it is actually clear before we set it, in order to maintain the appropriate count. We need to use a 64-bit value to hold this count. We also define the target—how large each coset is.

```
<Data declarations 2> +=
  long long uniq = 0;
  long long probes = 0;
  const long long TARGET = FACT8 * (long long) FACT8 * (long long) FACT4/2;
#ifdef LEVELCOUNTS
  long long uniq_ulev = 0;
  long long sum_ulev[30];
#endif
```

18. Again looking ahead somewhat, we need a method to get a new cleared page. The reason we allocate an extra byte can be found in the section on the prepass, below; it is critical that we allocate that extra byte. We will assume we may be freeing and allocating pages, so we set up a queue for that purpose.

```

<Utility functions 13> +=
vector<unsigned char*> pageq;
unsigned char *getpage()
{
    unsigned char *r = 0;
    if (pageq.size() > 0) {
        r = pageq[pageq.size() - 1];
        pageq.pop_back();
    }
    else {
        r = (unsigned char *) malloc(PAGESIZE + 1);
        if (r == 0)
            error ("!no memory" );
    }
    return r;
}
unsigned char *getclearedpage()
{
    unsigned char *r = getpage();
    memset(r, 0, PAGESIZE);
    return r;
}
void freepage(unsigned char *r)
{
    pageq.push_back(r);
}

```

19. In some circumstances, we may want to run this program over many cosets but to a limited depth. In this case, freeing and clearing the bitmaps can dominate the runtime. To minimize this impact, we use a small array to indicate which pages were touched.

```

<Data declarations 2> +=
#ifdef FASTCLEAN
    unsigned char touched[FACT8];
    int did_a_prepass;
#endif

```

20. At the start of the program, we initialize our bitmap and some other variables. If we already have a page, we assume it is cleared (our cleanup routine will ensure this).

```

⟨Handle one coset from movestring 8⟩ +≡
    for (int i = 0; i < FACT8; i++)
        if (bitp1[i] == 0) bitp1[i] = getclearedpage();
    uniq = 0;
#ifdef FASTCLEAN
    memset(touched, 0, sizeof (touched));
    did_a_prepass = 0;
#endif
#ifdef LEVELCOUNTS
    uniq_ulev = 0;
#endif

```

21. The bit order of the lowest order bits will turn out to be absolutely critical later on. To support this we introduce an array that translates a **FACT4** value into a particular offset.

```

⟨Data declarations 2⟩ +≡
    unsigned char permtohit[FACT4];
    unsigned char bittoperm[FACT4];

```

22. For now we initialize it to just sequential order.

```

⟨Initialize the program 7⟩ +≡
    for (int i = 0; i < FACT4; i++) {
        permtohit[i] = i;
        bittoperm[i] = i;
    }

```

23. Modern computers have processors so fast they can execute hundreds of instructions during the time it takes to fetch a single memory value. Here, in this one case, we can actually defer the checking of the new bit that might be set until the next time this routine is called, and issue a prefetch to the memory address that it will need. Doing this makes a substantial improvement in the overall performance of the program.

```

⟨Data declarations 2⟩ +≡
    unsigned char saveb;
    unsigned char *savep;

```

24. We need a routine that takes a **permcube** and sets the appropriate bit in the bitmap. We already have code that handles most of the work for us. We actually check the previous bit, and set up for the next bit. Note how we do as much work as possible before the *flushbit()* call, to give the memory system as much time as possible to fetch the data. This is one of the major keys to our excellent search performance.

```

<Utility functions 13> +=
void flushbit()
{
    if (savep != 0) {
        if (0 == (*savep & saveb)) {
            *savep |= saveb;
            uniq++;
        }
        savep = 0;
    }
}

void setonebit(const permcube &pc)
{
    int cindex = (pc.c8_4 * FACT4 + pc.ctp) * FACT4 + pc.cbp;
    unsigned int eindex = (((permcube::c12_8[pc.et] * FACT4) + pc.etp) * FACT4 / 2 + (pc.ebp >>
        1)) * FACT4 + permtobit[pc.emp];
#ifdef FASTCLEAN
    touched[cindex] = 1;
#endif
    probes++;
    flushbit();
    savep = bitp1[cindex] + (eindex >> 3);
    _builtin_prefetch(savep);
    saveb = 1 << (eindex & 7);
}

```

25. With this written, the second search routine is very close to the first. We also introduce a small optimization; we do not do the final two lookups when there's only one more move to go because we know they are unnecessary. The unrolling of the loop in this fashion is another major key to our performance. Note that we count on *ffs()* performing well; that's something to check on your architecture.

⟨Utility functions 13⟩ +=

```

void slowsearch2(const kocsymm &kc, const permcube &pc, int togo, int movemask, int canon)
{
    if (togo ≡ 0) {
        if (kc ≡ identity_kc) setonebit(pc);
        return;
    }
    togo--;
    kocsymm kc2;
    permcube pc2;
    int newmovemask;
    while (movemask) {
        int mv = ffs(movemask) - 1;
        movemask &= movemask - 1;
        kc2 = kc;
        kc2.move(mv);
        int nd = phase1prune::lookup(kc2, togo, newmovemask);
        if (nd ≤ togo) {
            pc2 = pc;
            pc2.move(mv);
            int new_canon = cubepos::next_cs(canon, mv);
            int movemask3 = newmovemask & cubepos::cs_mask(new_canon);
            if (togo ≡ 1) { /* just do the moves. */
                permcube pc3;
                while (movemask3) {
                    int mv2 = ffs(movemask3) - 1;
                    movemask3 &= movemask3 - 1;
                    pc3 = pc2;
                    pc3.move(mv2);
                    setonebit(pc3);
                }
            }
            else {
                slowsearch2(kc2, pc2, togo, movemask3, new_canon);
            }
        }
    }
}

```

26. If the *slow* value is one, we use the next fastest search.

⟨Handle one coset from *movestring* 8⟩ +=

```

if (slow ≡ 1) slowsearch2(repkc, reppc);

```

27. Our outer routine for the second level search is here.

```

<Utility functions 13> +=
void slowsearch2(const kocsymm &kc, const permcube &pc)
{
    duration();
    for (int d = phase1prune::lookup(repkc); d < maxsearchdepth; d++) {
        probes = 0;
        long long prevlev = uniq;
        slowsearch2(kc, pc, d, ALLMOVEMASK, CANONSEQSTART);
        flushbit();
        long long thislev = uniq - prevlev;
        if (verbose) cout << "Tests_at_" << d << "_" << probes << "_in_" << duration() << "_uniq_" <<
            uniq << "_lev_" << thislev << endl;
    }
}

```

28. Many times coming up, we will need to store an edge coordinate into a **permcube**. This routine makes it easy.

```

<Utility functions 13> +=
void unpack_edgcoord(permcube &pc, int e8_4, int epp1, int epp2)
{
    pc.et = permcube::c8_12[e8_4];
    pc.etp = epp1;
    pc.ebp = epp2;
    pc.eb = kocsymm::epsymm_compress[#f0f - kocsymm::epsymm_expand[pc.et]];
    pc.em = 0;
}
void unpack_edgcoord(permcube &pc, int coord)
{
    unpack_edgcoord(pc, coord/(FACT4 * FACT4), coord/FACT4 % FACT4, coord % FACT4);
}

```

29. Prepass. We've got a tremendously fast search routine at this point, and efficient storage for the coset. There is one more major trick in our arsenal, one that expands the capabilities of this coset solver tremendously. That technique is the prepass.

Once a position has a phase 1 solution (that is, it is in the group H), it has a tendency to stay there; 10 of the 18 possible moves leave it in the coset. Most canonical sequences that are a solution to phsae one end in a move in H . The prepass allows us to handle all sequences ending in H , in a pass over the bitmap, without needing to explicitly search. This speeds up our search by a factor of two, typically, and for some cosets, like the trivial coset, speeds it up by a much larger factor. For instance, for the trivial coset, at distance 12, there are 16,019,916,192 canonical sequences that solve phase 1, but only 329,352,128 of those, or about one in fifty, end in a move not in H . This means a huge speedup for some cosets, and a good speedup for other cosets.

Use of the prepass also permits us to calculate bounds on the overall distance of the coset without finding optimal solutions for every single position.

The prepass works by taking the current set of found solutions, and extending each one by all ten moves in H , and adding those to the set. That's why we declared *bitp2* above; this is to hold a second copy of the bitmap, so we can maintain a separate previous and next bitmap during the prepass.

For proving a bound of 20, the prepass is the key operation in the coset solver, and the one that will take up the bulk of the time. The trick to doing this efficiently is to lay out the bits in memory strategically, so that almost all of the work can be done with simple logical operations and a lookup table or two.

For checking, we may want to disable the prepass.

```
<Data declarations 2> +≡
  int disable_prepass = 0;
```

30. The $-U$ option disables the prepass.

```
<More arguments 5> +≡
case 'U': disable_prepass++;
  break;
```

31. The layout is already implicit in the *setbit* routine above, although we have not finished all the code we need, nor have we set up the *bittoperm* and *permtobit* arrays appropriately.

Let's start with the least significant bits, the *emp* bits. Setting the mapping of middle edge permutations to bit locations. For a given set of corner and up/down edge permutations, there are twelve possible middle edge permutations that preserve parity. Six of the ten moves in *H* do not affect the middle edge permutation, so for those six moves, we can just copy the middle edge bits over; the assignment of bits to middle edge permutations is not affected by those six moves. For the remaining four moves (F2, R2, B2, L2), every move jumbles the bits in a specific way. We assign the bits to the permutations and vice versa in a way such that all the even permutations are in the low order bits, and such that the odd permutation bit assignment of the 12 is that from the corresponding even permutation after the move F2. This is how we assign bits to permutations; the following code does the work for us.

```

⟨Initialize the program 7⟩ +≡
  const int F2 = 1 + TWISTS;
  const int R2 = 1 + 2 * TWISTS;
  const int B2 = 1 + 4 * TWISTS;
  const int L2 = 1 + 5 * TWISTS;
  permcube pc;
  for (int i = 0; i < FACT4/2; i++) {
    permtobit[2 * i] = i;
    pc.emp = 2 * i;
    pc.move(F2);
    permtobit[pc.emp] = 12 + i;
  }
  for (int i = 0; i < FACT4; i++) bittoperm[permtobit[i]] = i;

```

32. For the remaining moves, we have to calculate the rearrangements of the 12 bits that can happen (for both even and odd values).

```

⟨Data declarations 2⟩ +≡
  const int SQMOVES = 3;
  short rearrange[2][SQMOVES][1 << 12];

```

33. Initializing these is just a slog through the possibilities. We first set all the single-bit values, and then we combine them.

```

⟨Initialize the program 7⟩ +≡
  const int mvs[] = {R2, B2, L2};
  for (int mvi = 0; mvi < SQMOVES; mvi++)
    for (int p = 0; p < FACT4; p++) {
      pc.emp = p;
      pc.move(mvs[mvi]);
      rearrange[p & 1][mvi][1 << (permtobit[p] % 12)] = 1 << (permtobit[pc.emp] % 12);
    }
  for (int p = 0; p < 2; p++)
    for (int mvi = 0; mvi < SQMOVES; mvi++)
      for (int i = 1; i < (1 << 12); i++) {
        int lowb = i & -i;
        rearrange[p][mvi][i] = rearrange[p][mvi][lowb] | rearrange[p][mvi][i - lowb];
      }

```

34. It turns out, with all the choices we have made (all of which were deterministic), the values for B2 are the same going forwards or backwards. We take advantage of that to reduce cache misses in the inner loop, but we check this still holds here.

```

⟨Initialize the program 7⟩ +=
  for (int i = 0; i < (1 << 12); i++)
    if (rearrange[0][1][i] ≠ rearrange[1][1][i])
      error ("!_mismatch_in_rearrange");

```

35. For the next most significant bits, we use the up/down edge permutation. We use a lookup array to perform this mapping. For indexing, we drop the least significant bit of the up/down edge mapping, since it can be reconstructed from the parity of the other permutation components. We take advantage of the fact that consecutive pairs of permutations indexed by $(2k, 2k + 1)$, when right-multiplied by any element of S_4 , yields another pair, either $(2j, 2j + 1)$ or $(2j + 1, 2j)$. (See **kocsymm** for more information.) (This is also true for groups of 6, which help make this algorithm particularly cache-friendly.) Note that this array is reasonably large, but we access it sequentially. We premultiply by 3 to get an actual page offset. Our pages are $8! * 3/2$ or 60,480 bytes long, so these indices barely fit into an unsigned short.

```

⟨Data declarations 2⟩ +=
  const int PREPASS_MOVES = 10;
  unsigned short eperm_map[FACT8/2][PREPASS_MOVES];

```

36. The initialization is long but straightforward.

```

⟨Initialize the program 7⟩ +=
  int ind = 0;
  for (int e8_4 = 0; e8_4 < C8_4; e8_4++)
    for (int epp1 = 0; epp1 < FACT4; epp1++)
      for (int epp2 = 0; epp2 < FACT4; epp2 += 2, ind++) {
        int mvi = 0;
        for (int mv = 0; mv < NMOVES; mv++) {
          if (¬kocsymm::in_Kociemba_group(mv)) continue;
          unpack_edgcoord(pc, e8_4, epp1, epp2);
          pc.move(mv);
          eperm_map[ind][mvi] = ((permcube::c12_8[pc.et] * FACT4 + pc.etp) * FACT4 + pc.ebp)/2 * 3;
          mvi++;
        }
      }
}

```

37. The inner loop. We are ready now for the innermost loop of the program, the one that accounts for probably two-thirds of the runtime in our effort to prove 20. Examine the assembly generated by your compiler for this routine very carefully. The inner loop should have about 50 instructions, straight line code; if you see anything that could be improved, change it here.

Input to this function is a bit complicated. We have a destination page we are going to write (but surprisingly, not read). We have a set of ten source pages, one for each of the instructions in H , from which we will read bits, transform them into the new locations in our destination page, and write them back. Finally, we have the base, which is the portion of the pages to do. This routine does $12 \cdot 24 \cdot 24$ bits from each page, which corresponds to 2,304 bytes from each page. One full execution of this routine does 69,120 group multiplies. The inner body is executed 288 times on each call, and each inner body execution does 240 group multiplies.

The code below uses unaligned reads and writes. When I was originally writing this code, I used a bunch of byte reads and writes, but it turned out to be significantly faster to just use unaligned memory accesses. On modern Intel processors, there is almost no penalty for unaligned reads and writes. On other processors, this code may perform substantially suboptimally.

Note that this routine smashes one byte past the end of each page. For this reason, we allocate each page one byte larger than it really needs to be. The code restores the smashed value at the end.

This routine does not read the source. With the exception of the empty sequence, every sequence that ends in the trivial H group either ends with a move in the H group, or ends with one of the eight moves F1, F3, R1, R3, B1, B3, L1, L3. These eight moves always occur in pairs, as do the sequences that use them; any sequence ending in F1 that ends in H has a matching sequence that ends in F3 that also ends in H . What this means is that for every position that has already been found, there is a matching position also already found that is either F2, R2, B2, or L2 away. So, except for the trivial coset and the empty sequence, we never need to consider the source bitmap in the inner loop; just considering the ten adjacent bitmaps always suffices and never loses any bits.

(Utility functions 13) +=

```
void innerloop3(unsigned char *dst, unsigned char **srcs, int base)
{
    dst += 3 * base;
    unsigned char *end = dst + 12 * 24 * 3;
    unsigned char tval = *end;    /* save this byte */
    unsigned short *cpp = eperm_map[base];
    for (; dst < end; dst += 3, cpp += PREPASS_MOVES) {
        int wf2 = *(int *)(srcs[3] + cpp[3]);
        int wr2 = *(int *)(srcs[4] + cpp[4]);
        int wb2 = *(int *)(srcs[8] + cpp[8]);
        int wl2 = *(int *)(srcs[9] + cpp[9]);

        *(int *) dst = (((wf2 & #fff) |      /* F2, even to odd */
            rearrange[0][0][wr2 & #fff] |    /* R2, even to odd */
            rearrange[0][1][wb2 & #fff] |    /* B2, even to odd */
            rearrange[0][2][wl2 & #fff] << 12) | /* L2, even to odd */
            ((wf2 >> 12) & #fff) |          /* F2, odd to even */
            rearrange[1][0][(wr2 >> 12) & #fff] | /* R2, odd to even */
            rearrange[0][1][(wb2 >> 12) & #fff] | /* B2, odd to even */
            rearrange[1][2][(wl2 >> 12) & #fff] | /* L2, odd to even */
            *(int *)(srcs[0] + cpp[0]) |      /* U1 */
            *(int *)(srcs[1] + cpp[1]) |      /* U2 */
            *(int *)(srcs[2] + cpp[2]) |      /* U3 */
            *(int *)(srcs[5] + cpp[5]) |      /* D1 */
            *(int *)(srcs[6] + cpp[6]) |      /* D2 */
            *(int *)(srcs[7] + cpp[7]));      /* D3 */
    }
}
```

```

    }
    *end = tval;    /* restore smashed value */
}

```

38. After updating a page we want to count the bits set. This routine does a fairly good job. This can probably be replaced with some intrinsics that might do a better job.

⟨Utility functions 13⟩ +≡

```

int countbits(unsigned int *a)
{
    int r = 0;
    const unsigned int mask1 = #55555555;
    const unsigned int mask2 = #33333333;
    const unsigned int mask3 = #0f0f0f0f;
    for (int i = 0; i < PAGESIZE; i += 24) {
        unsigned int w1 = *a++;
        unsigned int w2 = *a++;
        unsigned int w3 = *a++;

        w1 = (w1 & mask1) + ((w1 >> 1) & mask1) + (w2 & mask1);
        w2 = ((w2 >> 1) & mask1) + (w3 & mask1) + ((w3 >> 1) & mask1);
        unsigned int s1 = (w1 & mask2) + ((w1 >> 2) & mask2) + (w2 & mask2) + ((w2 >> 2) & mask2);
        s1 = (s1 & mask3) + ((s1 >> 4) & mask3);
        w1 = *a++;
        w2 = *a++;
        w3 = *a++;
        w1 = (w1 & mask1) + ((w1 >> 1) & mask1) + (w2 & mask1);
        w2 = ((w2 >> 1) & mask1) + (w3 & mask1) + ((w3 >> 1) & mask1);
        unsigned int s2 = (w1 & mask2) + ((w1 >> 2) & mask2) + (w2 & mask2) + ((w2 >> 2) & mask2);
        s1 += (s2 & mask3) + ((s2 >> 4) & mask3);
        r += 255 & ((s1 >> 24) + (s1 >> 16) + (s1 >> 8) + s1);
    }
    return r;
}

```

39. Sometimes, like when we are doing level counting, we need to do a bit count as above, but we also need to calculate a weighted sum. This routine handles that; it is slower than the above routine, but it should only very rarely be called. It depends on an array containing bit counts.

⟨Data declarations 2⟩ +≡

```

#ifdef LEVELCOUNTS
    unsigned char bc[1 << 12];
#endif

```

40. Setting up the bit count array is easy.

⟨Initialize the program 7⟩ +≡

```

#ifdef LEVELCOUNTS
    for (int i = 1; i < (1 << 12); i++) bc[i] = 1 + bc[i & (i - 1)];
#endif

```

41. Finally, we have our alternative bitcount function. This does nasty things with pointers; it will only work, for instance, on a little endian machine (just like our prepass).

```

<Utility functions 13> +=
#ifdef LEVELCOUNTS
    int parity(int coord)
    {
        return (permcube::c8_4_parity[coord/(FACT4 * FACT4)] ⊕ (coord/24) ⊕ coord) & 1;
    }
    int countbits2(int cperm, int *a, int &rv2)
    {
        int coparity = parity(cperm);
        int r = 0, r2 = 0;
        int ind = 0;
        for (int e8_4 = 0; e8_4 < C8_4; e8_4++) {
            int p2 = coparity ⊕ permcube::c8_4_parity[e8_4];
            for (int epp1 = 0; epp1 < FACT4; epp1++) {
                int off1 = (p2 ⊕ epp1) & 1;
                int off2 = 1 - off1;
                for (int epp2 = 0; epp2 < FACT4; epp2 += 2, ind += 2) {
                    int w = *a;
                    int v1 = bc[(w & #fff)];
                    int v2 = bc[((w >> 12) & #fff)];
                    r += v1 + v2;
                    r2 += v1 * levmul[ind + off1] + v2 * levmul[ind + off2];
                    a = (int *)(((char *) a) + 3);
                }
            }
        }
        rv2 = r2;
        return r;
    }
#endif

```

42. For even more cache-friendliness, we do sets of related pages as a group. These pages are typically related by the up and down face moves. To store data about every collection of pages, we use the following structure, which contains pointers to the pages, as well as a mapping of the order to do the various offsets in.

```

<Utility functions 13> +=
struct elemdata {
    unsigned char *dst;
    unsigned char *from[PREPASS_MOVES];
    unsigned char e8_4map[70];
};

```

43. The number of pages in a collection is defined by this constant.

```

<Data declarations 2> +=
const int STRIDE = 16;

```

44. The ordering of pages to solve was defined by an external optimizer, which we do not go into here. Suffice it to say that *cornerorder* contains a permutation of the integers 0...40319; any order will work, but some orders may cause this program to use a fair bit less memory. The output of this routine is in an include file named *corner_order.h* and its values are stored in an array called *cornerorder*.

```
<Data declarations 2> +≡
#include "corner_order.h"
```

45. The following routine computes the neighbors of a corner.

```
<Utility functions 13> +≡
void calcneighbors(int cperm, int *a)
{
    permcube pc, pc2;
    pc.c8_4 = cperm / (FACT4 * FACT4);
    pc.ctp = cperm / FACT4 % FACT4;
    pc.cbp = cperm % FACT4;
    for (int mv = 0; mv < NMOVES; mv++) {
        if (!kocsymm::in_Kociemba_group(mv)) continue;
        pc2 = pc;
        pc2.move(mv);
        *a++ = (pc2.c8_4 * FACT4 + pc2.ctp) * FACT4 + pc2.cbp;
    }
}
```

46. Our main routine to do a collection of STRIDE pages is below. The magic array *moveseq16* is the sequence of moves that generates the relevant values in a single group of *corder_order*; we check that this holds.

```

<Utility functions 13> +=
int moveseq16[] = {2, 9, 0, 9, 2, 9, 0, 0, 0, 11, 2, 11, 0, 11, 2, 0};
unsigned char initorder[70];
void doouter(int r)
{
    elemdata edata[16];
    int neighbors[PREPASS_MOVES];
    int tperm = cornerorder[r];
    permcube pc, pc2;
    for (int i = 0; i < STRIDE; i++) {
        elemdata *e = edata + i;
        int cperm = cornerorder[r + i];
        if (cperm ≠ tperm)
            error ("!_inconsistent_corner_order" );
        calcneighbors(cperm, neighbors);
        int dp = 0;
        for (int mv = 0; mv < NMOVES; mv++) {
            if (-kocsymm::in_Kociemba_group(mv)) continue;
            e-from[dp] = bitp2[neighbors[dp]];
            if (mv ≡ moveseq16[i]) tperm = neighbors[dp];
            dp++;
        }
        if (i ≡ 0) {
            for (int j = 0; j < 70; j++) e-e84map[j] = initorder[j];
        }
        else {
            pc = identity_pc;
            for (int j = 0; j < 70; j++) {
                pc.c8_4 = (e - 1)-e84map[j];
                pc.move(moveseq16[i - 1]);
                e-e84map[j] = pc.c8_4;
            }
        }
        e-dst = bitp1[cperm];
    }
    for (int i = 0; i < 70; i++)
        for (int j = 0; j < STRIDE; j++) {
            elemdata *e = edata + j;
            innerloop3(e-dst, e-from, e-e84map[i] * 12 * 24);
        }
}

```

47. We need to initialize the *initorder* array.

```

<Initialize the program 7> +≡
  unsigned char used[70];
  memset(initorder, 255, sizeof (initorder));
  memset(used, 255, sizeof (used));
  int at = 0;
  for (int i = 0; i < 70; i++)
    if (used[i] ≡ 255) {
      permcube pc2;
      used[i] = 0;
      initorder[at++] = i;
      unpack_edgcoord(pc, i, 0, 0);
      for (int j = 0; j < 15; j++) {
        pc2 = pc;
        pc2.move(moveseq16[j]);
        int e84 = permcube::c12_8[pc.et];
        if (used[e84] ≡ 255) {
          used[e84] = 0;
          initorder[at++] = e84;
        }
      }
    }
  if (at ≠ 70)
    error ("!bad_setup_in_setorder()");

```

48. Threading. The fastest search routine is threaded, so we need some objects to manage local state so it doesn't interfere with global state. We also build variations of some of the above routines that work better in a threaded context. Where safe to do so, we leave the routines global, but we do need some local state. Any routines that use the worker thread, we declare in this CWEB macro. Everything past here, we write with a careful eye towards threading.

```

<Threading objects 48> ≡
struct worker_thread {
    <Worker thread object contents 63>;
    char pad[128]; /* make sure these objects don't share cache lines */
} workers[MAX_THREADS];

```

See also sections 72 and 73.

This code is used in section 1.

49. Access to the output stream, or any other shared state, requires a global mutex.

```

<Data declarations 2> +≡
    pthread_mutex_t mutex;

```

50. We initialize the mutex.

```

<Initialize the program 7> +≡
    pthread_mutex_init(&mutex, Λ);

```

51. We call these methods to acquire and release the mutex.

```

<Utility functions 13> +≡
void get_global_lock()
{
    pthread_mutex_lock(&mutex);
}
void release_global_lock()
{
    pthread_mutex_unlock(&mutex);
}

```

52. If we need to display a position from a bitmap, this is how we do it. Note that sometimes the data may need to go to a separate file. In addition, we support setting of the level at which sing positions are written.

```

<Utility functions 13> +≡
FILE *singfile;
int singcount;
int singlevel = 99;
void showsing(const permcube &pc)
{
    cubepos cp, cp2;
    pc.set_perm(cp);
    cubepos :: mul(cp, repcp, cp2);
    if (singfile) fprintf(singfile, "SING□%s\n", cp2.Singmaster_string());
    else cout << "SING□" << cp2.Singmaster_string() << endl;
    singcount ++;
}

```

53. We need to parse an argument to set the singfile. In addition

```

⟨ More arguments 5 ⟩ +≡
case 'f':
  if (argc < 2)
    error ("!not_enough_arguments_to_f" );
  singfile = fopen(argv[1], "w");
  if (singfile ≡ 0)
    error ("!can't_open_singfile" );
  argc--;
  argv++;
  break;
case 'L':
  if (argc < 2)
    error ("!not_enough_arguments_to_L" );
  if (sscanf(argv[1], "%d", &singlelevel) ≠ 1)
    error ("!non-numeric_argument_to_L" );
  argc--;
  argv++;
  break;

```

54. At the end we close the singfile and report how many positions were written.

```

⟨ Cleanup 54 ⟩ ≡
  if (singfile) {
    cout << "Wrote_" << singcount << "_total_positions_to_singfile" << endl;
    fclose(singfile);
  }

```

See also section 86.

This code is used in section 1.

55. Sometimes we need to scan a page for unset bits, and display the corresponding positions.

```

<Utility functions 13> +=
void showunset(int cperm)
{
    int *pbits = (int *) bitp1[cperm];
    permcube pc;
    pc.c8_4 = cperm/(FACT4 * FACT4);
    pc.ctp = cperm/FACT4 % FACT4;
    pc.cbp = cperm % FACT4;
    int coparity = (permcube::c8_4_parity[pc.c8_4] ⊕ pc.ctp ⊕ pc.cbp) & 1;
    for (int i = 0; i < FACT8/8 * 3; i++) {
        if (pbits[i] ≠ -1) {
            int t = ~pbits[i];
            while (t) {
                int j = ffs(t) - 1;
                t &= t - 1;
                int k = (i << 5) + j;
                int ep8_4 = k/(24 * 24 * 12);
                int epp1 = k/(24 * 12) % 24;
                int epp2 = k/24 % 12 * 2;
                int emperm = bittoperm[k % 24];
                int par0 = (permcube::c8_4_parity[ep8_4] ⊕ epp1 ⊕ emperm ⊕ coparity) & 1;
                unpack_edgcoord(pc, ep8_4, epp1, epp2 + par0);
                pc.emp = emperm;
                get_global_lock();
                showing(pc);
                release_global_lock();
            }
        }
    }
}

```

56. The purpose of the corner order array is to save some memory while we are running, so we don't actually need two complete bitmaps in memory at a given time. To make this work, we need to track which pages have been used how many times; as soon as a page has been used as many times as we expect, we can free it. This array helps us keep track of that. We also maintain a variable that says where we are in the *cornerorder* array.

```

<Data declarations 2> +=
unsigned char use_count[FACT8];
int work_done;

```

57. When we want to find the next chunk of pages to work on, we call *get_prepass_work()*. If it returns -1 , we are done. This may be multithreaded, so we get the lock. We allocate the destination pages as we need them; note that we do not need to clear these pages, as the prepass just smashes whatever values it finds in them.

```

<Utility functions 13> +=
int get_prepass_work()
{
    get_global_lock();
    int r = work_done;
    if (r < FACT8) {
        for (int i = 0; i < STRIDE; i++) bitp1[cornerorder[r + i]] = getpage();
        work_done += STRIDE;
    }
    release_global_lock();
    if (r ≥ FACT8) return -1;
    return r;
}

```

58. When we are done with a chunk of work, we track which pages are possibly freeable. We notice a special *uniq* value of -1 that indicates we do not need to count bits.

```

<Utility functions 13> +=
void finish_prepass_work(int cperm)
{
    int thisblock = 0;
#ifdef LEVELCOUNTS
    int this_ulev = 0;
#endif
    if (need_count_bits) {
#ifdef LEVELCOUNTS
        if (need_count_bits > 1) thisblock = countbits2(cperm, (int *) bitp1[cperm], this_ulev);
        else
#endif
        thisblock = countbits((unsigned int *) bitp1[cperm]);
    }
    if (global_depth + 1 ≥ singlelevel ∧ maxsearchdepth < global_depth) showunset(cperm);
    int neighbors[PREPASS_MOVES];
    calcneighbors(cperm, neighbors);
    get_global_lock();
    uniq += thisblock;
#ifdef LEVELCOUNTS
    uniq_ulev += this_ulev;
#endif
    for (int i = 0; i < PREPASS_MOVES; i++) {
        if ((--use_count[neighbors[i]]) ≡ 0) {
            freepage(bitp2[neighbors[i]]);
            bitp2[neighbors[i]] = 0;
        }
    }
    release_global_lock();
}

```

59. Our main prepass work goes like this. We declare it as it is, so it meets the requirements of *pthread*.

```

<Utility functions 13> +=
void *do_prepass_work(void *)
{
    while (1) {
        int r = get_prepass_work();
        if (r < 0) break;
        doouter(r);
        for (int i = 0; i < STRIDE; i++) finish_prepass_work(cornerorder[r + i]);
    }
    return 0;
}

```

60. We are finally ready for our full prepass routine. Threading it is straightforward; we don't need special state, we just spawn the workers, have them all get work, and do it. Each worker is identical.

```

<Utility functions 13> +=
void doprepass()
{
    uniq = 0;
#ifdef FASTCLEAN
    did_a_prepass = 1;
#endif
#ifdef LEVELCOUNTS
    uniq_ulev = 0;
#endif
    swap(bitp1, bitp2);
    memset(use_count, PREPASS_MOVES, sizeof (use_count));
    work_done = 0;
    pthread_t p_thread[MAX_THREADS];
    for (int ti = 1; ti < numthreads; ti++) pthread_create(&(p_thread[ti]), Λ, do_prepass_work, 0);
    do_prepass_work(0);
    for (int ti = 1; ti < numthreads; ti++) pthread_join(p_thread[ti], 0);
    if (need_count_bits == 0) cout << "Prepass_at_" << global_depth << "_done_in_" << duration() <<
        ";_unique?" << endl << flush;
    else cout << "Prepass_at_" << global_depth << "_done_in_" << duration() << ";_unique" <<
        uniq << endl << flush;
}

```

61. Our prepass must be used with a search routine that knows it is being used, so it does not repeat the work that the prepass does. We need a flag to indicate that a prepass is going to be done on this level.

```

<Data declarations 2> +=
int this_level_did_prepass = 0;

```

62. Each search worker thread maintains its own separate pool of statistics, so there is no contention on shared variables. We also maintain a queue of memory addresses and bits to examine, because we do not want to get the global lock each time. We define a constant that says how big a batch to use for the address queue.

```
<Utility functions 13> +=  
    const int CHECKABITSIZE = 64;  
    struct checkabit {  
        unsigned char *p;  
#ifdef LEVELCOUNTS  
        unsigned char weight;  
#endif  
        unsigned char b;  
    };
```

63. Our flush routine and its data is now thread-specific.

```

⟨ Worker thread object contents 63 ⟩ ≡
    long long local_probes;
#ifdef LEVELCOUNTS
    long long local_ulev;
#endif
    int bitcount;
    checkabit q[CHECKABITSIZE];
    void local_bitflush()
    {
        get_global_lock();
        int this_uniq = 0;
#ifdef LEVELCOUNTS
        int this_ulev = 0;
#endif
        for (int i = 0; i < bitcount; i++) {
            checkabit &c = q[i];
            if (0 ≡ (*c.p & c.b)) {
                *c.p |= c.b;
#ifdef LEVELCOUNTS
                this_ulev += c.weight;
#endif
                this_uniq++;
            }
        }
        uniq += this_uniq;
        if (uniq > enoughbits ∧ global_depth ≡ maxsearchdepth) search_terminated_early = 1;
#ifdef LEVELCOUNTS
        local_ulev += this_ulev;
#endif
        release_global_lock();
        local_probes += bitcount;
        bitcount = 0;
    }
    void local_initialize()
    {
        local_probes = 0;
#ifdef LEVELCOUNTS
        local_ulev = 0;
#endif
        bitcount = 0;
        ⟨ More thread initialization 65 ⟩;
    }

```

See also sections 64, 68, and 70.

This code is used in section 48.

64. We use a local set a bit routine and a local touched array.

```

< Worker thread object contents 63 > +=
  unsigned char local_touched[FACT8];
  void local_setonebit(const permcube &pc)
  {
    int cindex = (pc.c8_4 * FACT4 + pc.ctp) * FACT4 + pc.cbp;
    int epindex = ((permcube::c12_8[pc.et] * FACT4) + pc.etp) * FACT4 + pc.ebp;
    int eindex = (epindex >> 1) * FACT4 + permtobit[pc.emp];
#ifdef FASTCLEAN
    local_touched[cindex] = 1;
#endif
    if (bitcount >= CHECKABITSIZE) local_bitflush();
    checkabit &c = q[bitcount];
    c.p = bitp1[cindex] + (eindex >> 3);
    _builtin_prefetch(c.p);
#ifdef LEVELCOUNTS
    c.weight = levmul[epindex];
#endif
    c.b = 1 << (eindex & 7);
    bitcount++;
  }

```

65. We need to clear our *local_touched* array.

```

< More thread initialization 65 > ≡
#ifdef FASTCLEAN
  memset(local_touched, 0, sizeof (local_touched));
#endif

```

This code is used in section 63.

66. Our fast search routine also supports the ability to exit early, if a particular count of bits set is reached. We also support the ability to not count bits exactly, requested by *-a*. Finally, the *-F* option sets up *fast20* mode, which sets the max search depth to 16, the last level to 20, disables counting of the sets that don't matter, and sets *enoughbits* at level 15 based on a heuristic.

```

< Data declarations 2 > +=
  int search_terminated_early = 0;
  int dont_count_after_max_search = 0;
  int need_count_bits = 0;
  long long enoughbits = TARGET;
  int fast20;

```

67. We set this value with the `-e` option.

⟨More arguments 5⟩ +≡

```
case 'e':
    if (argc < 2)
        error ("!_not_enough_arguments_to_e");
    if (sscanf(argv[1], "%lld", &enoughbits) ≠ 1)
        error ("!_bad_arguments_to_e");
    argc--;
    argv++;
    break;
case 'a': dont_count_after_max_search++;
    break;
case 'F': fast20++;
    dont_count_after_max_search++;
    maxsearchdepth = 16;
    maxdepth = 20;
    break;
```

68. And now we have our search routine. This is the same as the previous search routine, except it does not consider solutions that end in moves from H . It turns out that any sequence that does not end in a move from H , but still ends in the H group, must be at a certain distance from H late in the sequence. For instance, right before the last move, it must be exactly one move from H . Right before the second to the last move, it must be exactly two moves from H . Right before the third move, it can be at either two or three moves from H . Right before the fourth move, it must be at least one move away from H , but no more than four moves. Otherwise, this is pretty much the same as *slowsearch2*.

(Worker thread object contents 63) +≡

```

void search(const kocsymm &kc, const permcube &pc, int togo, int movemask, int canon)
{
  if (togo == 0) {
    if (kc == identity_kc) local_setonebit(pc);
    return;
  }
  togo--;
  kocsymm kc2;
  permcube pc2;
  int newmovemask;
  if (search_terminated_early) return;
  while (movemask) {
    int mv = ffs(movemask) - 1;
    movemask &= movemask - 1;
    kc2 = kc;
    kc2.move(mv);
    int nd = phase1prune::lookup(kc2, togo, newmovemask);
    if (nd <= togo & (togo == nd || togo + nd >= 5 || !this_level.did_prepass)) {
      pc2 = pc;
      pc2.move(mv);
      int new_canon = cubepos::next_cs(canon, mv);
      int movemask3 = newmovemask & cubepos::cs_mask(new_canon);
      if (togo == 1) { /* just do the moves. */
        permcube pc3;
        while (movemask3) {
          int mv2 = ffs(movemask3) - 1;
          movemask3 &= movemask3 - 1;
          pc3 = pc2;
          pc3.move(mv2);
          local_setonebit(pc3);
        }
      }
      else {
        search(kc2, pc2, togo, movemask3, new_canon);
      }
    }
  }
}

```

69. Our worker threads get work parcels. A work parcel is an initial set of two moves to work with. Note that if the starting depth is three or less, we do not even bother to spread the work.

```

<Utility functions 13> +=
int search_work_seq;
int get_search_work()
{
    get_global_lock();
    int r = -1;
    while (1) {
        r = search_work_seq++;
        if (r ≥ NMOVES * NMOVES) {
            r = -1;
            break;
        }
        int mv1 = r/NMOVES;
        int mv2 = r % NMOVES;
        int s = cubepos::next_cs(CANONSEQSTART, mv1);
        int mask = cubepos::cs_mask(s);
        if ((mask >> mv2) & 1) break;
    }
    release_global_lock();
    return r;
}

```

70. The worker thread gets work until there's no more to be gotten.

⟨Worker thread object contents 63⟩ +≡

```

void dowork()
{
    local_initialize();
    if (global_depth ≤ 3) {
        search(repkc, reppc, global_depth, ALLMOVEMASK, CANONSEQSTART);
    }
    else {
        while (1) {
            int movepair = get_search_work();
            if (movepair < 0) break;
            int mv1 = movepair / NMOVES;
            int mv2 = movepair % NMOVES;
            kocsymm kc(repkc);
            permcube pc(reppc);
            kc.move(mv1);
            pc.move(mv1);
            kc.move(mv2);
            pc.move(mv2);
            int s = cubepos::next_cs(CANONSEQSTART, mv1);
            s = cubepos::next_cs(s, mv2);
            int mask;
            phase1prune::lookup(kc, global_depth - 2, mask);
            search(kc, pc, global_depth - 2, mask & cubepos::cs_mask(s), s);
        }
    }
    local_bitflush();
    get_global_lock();
    probes += local_probes;
#ifdef LEVELCOUNTS
    uniq_ulev += local_ulev;
#endif
#ifdef FASTCLEAN
    for (int i = 0; i < FACT8; i++) touched[i] |= local_touched[i];
#endif
    release_global_lock();
}

```

71. If the *slow* value is greater than 1, we use the normal fast search.

⟨Handle one coset from *movestring* 8⟩ +≡

```

if (slow > 1) search();

```

72. This is the *pthread*-compatible worker.

⟨Threading objects 48⟩ +≡

```
void *do_search_work(void *s)
{
    worker_thread *w = (worker_thread *) s;
    w->dowork();
    return 0;
}
```

73. Our outer routine for the fast search is here. The rules for doing prepass are moderately complex. We require a minimum count of six million set bits (otherwise search is faster than prepass). For once, we have a routine that might terminate, so we need to check for that.

```

⟨Threading objects 48⟩ +=
  void search()
  {
    duration();
    for (int d = phase1prune::lookup(repkc); d ≤ maxdepth; d++) {
      global_depth = d;
      probes = 0;

      long long prevlev = uniq;
#ifdef LEVELCOUNTS
      long long prev_ulev = uniq_ulev;
#endif
      this_level.did_prepass = ¬disable_prepass ∧ d > 1 ∧ (uniq > 6000000 ∨ d > maxsearchdepth);
      if (this_level.did_prepass) {
        need_count_bits = (d ≤ maxsearchdepth ∨ ¬dont_count_after_max_search);
#ifdef LEVELCOUNTS
        if (d ≤ maxsearchdepth) need_count_bits++;
#endif
      }
      doprepass();
    }
    else {
      need_count_bits = 1;
    }
    if (uniq ≡ TARGET) break;
    search_work_seq = 0;
    search_terminated_early = 0;
    int did_full_search = 0;
    if (fast20 ∧ d ≡ 16) enoughbits = 167000000 + uniq/3;
    if (d ≤ maxsearchdepth) {
      did_full_search = 1;
      if (d ≤ 3) {
        workers[0].dowork();
      }
      else {
        pthread_tp_thread[MAX_THREADS];
        for (int ti = 1; ti < numthreads; ti++)
          pthread_create(&(p_thread[ti]), Λ, do_search_work, &workers[ti]);
        workers[0].dowork();
        for (int ti = 1; ti < numthreads; ti++) pthread_join(p_thread[ti], 0);
      }
    }
    if (search_terminated_early) {
      cout << "Terminated_search_at_level_" << d << "_early_due_to_enoughbits" << endl;
      if (dont_count_after_max_search) need_count_bits = 0;
      did_full_search = 0;
    }
    long long thislev = uniq - prevlev;
#ifdef LEVELCOUNTS
    long long thisulev = uniq_ulev - prev_ulev;

```

```

#endif
    if (global_depth + 1 ≥ singlelevel ∧ maxsearchdepth ≥ global_depth)
        for (int cperm = 0; cperm < FACT8; cperm++) showunset(cperm);
    if (verbose) {
#ifdef LEVELCOUNTS
        if (need_count_bits ≡ 0) cout << "Tests_at_" << d << "_" << probes << "_in_" << duration() <<
            "_uniq?_lev?" << endl;
        else if (did_full_search)
            cout << "Tests_at_" << d << "_" << probes << "_in_" << duration() << "_uniq_" << uniq <<
                "_lev_" << thislev << "_utot_" << uniq_ulev << "_thisulev_" << thislev << endl;
        else cout << "Tests_at_" << d << "_" << probes << "_in_" << duration() << "_uniq_" << uniq <<
            "_lev_" << thislev << endl;
#else
        if (did_full_search) cout << "Tests_at_" << d << "_" << probes << "_in_" << duration() <<
            "_uniq_" << uniq << "_lev_" << thislev << endl;
        else cout << "Tests_at_" << d << "_" << probes << "_in_" << duration() << "_uniq?" << endl;
#endif
    }
#ifdef LEVELCOUNTS
    if (did_full_search) sum_ulev[d] += thisulev;
#endif
}
}
}

```

74. Finally, at the end of the *docoset* routine, we free up the memory we allocated. In the case of FASTCLEAN, we only free the touched pages in the first bitmap.

```

⟨Handle one coset from movestring 8⟩ +≡
    int delta = singcount - oldsingcount;
    if (singfile) cout << "Wrote_" << delta << "_sing_positions." << endl;
    for (int i = 0; i < FACT8; i++) {
#ifdef FASTCLEAN
        if (bitp1[i] ≠ 0 ∧ (touched[i] ∨ did_a_prepass)) {
            freepage(bitp1[i]);
            bitp1[i] = 0;
        }
#else
        if (bitp1[i] ≠ 0) {
            freepage(bitp1[i]);
            bitp1[i] = 0;
        }
#endif
        if (bitp2[i] ≠ 0) {
            freepage(bitp2[i]);
            bitp2[i] = 0;
        }
    }
    cout << "Finished_in_" << (walltime() - cosetstart) << endl;

```

75. Coset cover. This program works with the H coset, which only has 16-way symmetry. The full cube has 48-way symmetry, however, and it would be nice to take advantage of the full symmetry of the cube in reducing the problem size of proving a 20 bound. In order to make this happen, we need to compute a small covering set of H cosets that, when all the positions and their reorientations are taken into account, cover the complete cube space.

We have computed such a cover, and by solving only about 56 million out of the 138 million symmetry-unique H -cosets, we can prove a bound of $20f^*$; this reduces our total work by about 60%, so it is an important optimization.

The actual generation of this cover is beyond the scope of this program. However, we do embed the cover into the program. The cover is stored as a file called *bestsol.h* and creates an array *bestsol* that contains a value 0 if that coset is not to be computed, 1 if that coset should be completed on the first pass, and a 2 if that coset should be completed on the second pass. The first pass cosets, taken together, are enough to prove a bound of $21f^*$; first and second pass together cover the entire space to prove a bound of $20f^*$.

```
<Data declarations 2> +≡
#include "bestsol.h"
```

76. The *bestsol* array is in lexicographical order of symmetry-reduced edge permutation plus orientation; thus, there are about 65,000 entries. We index the cosets that are represented by these solutions, putting the phase one cosets first, and then the phase two cosets. We index the cosets so managing large batch runs of this program is easier; we can simply specify a range of cosets to run, and similarly, keep track of the range of cosets that have been completed. We use the *-r* option to pick up that range. Note that the *TOTALCOSETS* value is a magic number from our cover solution.

```
<Data declarations 2> +≡
int first_coset;
const int TOTALCOSETS = 55882296;
int coset_count;
```

77. This is where we parse the option.

```
<More arguments 5> +≡
case 'r':
    if (argc < 3)
        error ("!_not_enough_arguments_to_r");
    if (sscanf(argv[1], "%d", &first_coset) ≠ 1 ∨ sscanf(argv[2], "%d", &coset_count) ≠ 1)
        error ("!_bad_arguments_to_r");
    argc -= 2;
    argv += 2;
    if (first_coset < 0 ∨ first_coset ≥ TOTALCOSETS)
        error ("!_bad_first_value_to_r");
    if (coset_count ≤ 0)
        error ("!_bad_coset_count_specified_to_r");
    if (coset_count + first_coset > TOTALCOSETS) coset_count = TOTALCOSETS - first_coset;
    break;
```

78. As we generate the cosets, we will generate **kocsymm** objects indicating the coset to solve. This routine evaluates the coset, turns it into a move sequence, and calls the main *docoset* routine with that move string. We need to forward-declare the main *docoset* routine here, too.

```

⟨Utility functions 13⟩ +=
  const int U2 = 1;
  void docoset(int seq, const char *movestring);    /* forward declaration */
  void docoverelement(int seq, const kocsymm &kc)
  {
    int d = phase1prune::lookup(kc);
    if (d > maxdepth)    /* skip if too deep */
      return;
    moveseq moves = phase1prune::solve(kc);
    moves = cubepos::invert_sequence(moves);
    if (moves.size() == 0) {
      moves.push_back(U2);
      moves.push_back(U2);
    }
    char buf[160];
    strcpy(buf, cubepos::moveseq_string(moves));
    docoset(seq, buf);
  }

```

79. To make the indexing go faster, we keep track of the count of symmetry-reduced cosets for any single value of combined edge coordinate.

```

⟨Data declarations 2⟩ +=
  map<int, int> symcount;

```

80. The subroutine that generates the cosets is here. We need to make sure the phase 1 pruning table is initialized at the start.

```

⟨Utility functions 13⟩ +=
  int bestsollookup[EDGEOSYMM * EDGEPERM];
  unsigned int orderkc(const kocsymm &kc)
  {
    return (kc.epsymm << 11) + kc.eosymm;
  }
  int lookupkc(const kocsymm &kc)
  {
    return bestsollookup[(kc.epsymm << 11) + kc.eosymm];
  }
  void genseqs(int lo, int hi)
  {
    phase1prune::init();
    cubepos cp, cp2;
    int tot = 0;
    int c = 0;
    for (int eo = 0; eo < EDGEOSYMM; eo++)
      for (int ep = 0; ep < EDGEPERM; ep++) {
        kocsymm kc(0, eo, ep);
        kocsymm kc2;
        kc.canon_into(kc2);
        if (kc ≡ kc2) bestsollookup[orderkc(kc)] = bestsol[c++];
        else bestsollookup[orderkc(kc)] = bestsollookup[orderkc(kc2)];
      }
    for (int match = 2; match > 0; match--) {
      int c = 0;
      for (int eo = 0; eo < EDGEOSYMM; eo++) {
        for (int ep = 0; ep < EDGEPERM; ep++) {
          kocsymm kc(0, eo, ep);
          kocsymm kc2;
          kc.canon_into(kc2);
          if (¬(kc ≡ kc2)) continue;
          if (bestsol[c] ≠ match) {
            c++;
            continue;
          }
        }
        int cnt = 1;
        kc.set_coset(cp);
        for (int m = 1; m < 16; m++) {
          cp.remap_into(m, cp2);
          kocsymm kc2(cp2);
          if (kc2 ≡ kc) cnt |= 1 << m;
        }
        if (tot + CORNERSYMM < lo ∨ tot ≥ hi) {
          if (cnt ≡ 1) {
            tot += CORNERSYMM;
            c++;
          }
        }
      }
    }
  }

```

```

        continue;
    }
    else if (symcount.find(cnt) ≠ symcount.end()) {
        tot += symcount[cnt];
        c++;
        continue;
    }
}
int tcnt = 0;
for (int co = 0; co < CORNERSYMM; co++) {
    kc.csymm = co;
    int okay = 1;
    kc.set_coset(cp);
    for (int m = 1; okay ∧ m < 16; m++) {
        if (0 ≡ ((cnt ≫ m) & 1)) continue;
        cp.remap_into(m, cp2);
        kocsymm kc2(cp2);
        if (kc2 < kc) okay = 0;
    }
    if (okay) {
        if (tot ≥ lo ∧ tot < hi) docoverelement(tot, kc);
        tcnt++;
        tot++;
    }
}
symcount[cnt] = tcnt;
c++;
}
}
}
if (tot ≠ TOTALCOSETS)
    error ("!mistake_in_computation_of_total_cosets" );
}

```

81. We are ready to fill in our work handling routine. If the `-r` option was not specified, we expect a single coset on the command line. If there is no argument, we assume we want to run all cosets.

⟨Handle the work [81](#)⟩ ≡

```

if (argc > 1) {
    docoset(0, argv[1]);
}
else {
    if (coset_count ≡ 0) coset_count = TOTALCOSETS;
    gensesq(first_coset, first_coset + coset_count);
}

```

This code is used in section [1](#).

82. Our next major concern is using this coset solver to compute the exact count of cube positions at each distance—overall. If this matches the known results, that provides some validation that this program, the cover, and the execution are all correct. In addition, we hope to extend the known results. The prior known results were through 11f*. I extended those using a coset approach to 12f* and 13f*, and just today as I write this, someone else who goes by the username *tscheunemann* on the cube lovers forum extended the results to 14f*. With this program, I hope to take it to 15f*.

This is all a bit involved and tricky, however. We need to make sure we don't double count positions that exist in more than one of the cosets we plan to solve. Further, we need to make sure to handle symmetrical positions correctly. Luckily, our cover solution is built-in to this program, so we can do both.

Temporarily, we bracket this code with preprocessor directives so we can benchmark the program both with and without this feature, to see what the overall impact is.

The cover we use is based on the edge orientation, and on the location of the four middle edge cubies; we do not distinguish by corner orientation. Our overall edge permutation information is based on top, middle, and edge cubies; this does not exhibit the same 48-way symmetry as the cube itself. For efficiency, we provide an array that lets us convert our edge up/down permutation information to sets of edge permutations that share the same left/right edge occupancy and front/back edge occupancy, since this is what defines the subgroups that correspond to intersections of rotated Kociemba-group subsets. We also declare the array that will contain the eventual contribution of each edge permutation to our overall level count.

```
<Data declarations 2> +=
#ifdef LEVELCOUNTS
    long long levprev, levuniq, levsum;
    char levmul[FACT8];
    int k3map[FACT8];
#endif
```

83. The routine that initialies the *k3map* is next.

```
<Utility functions 13> +=
#ifdef LEVELCOUNTS
void setupk3map()
{
    int lookaside[1 << 12];
    memset(lookaside, -1, sizeof (lookaside));
    int ind = 0;
    cubepos cp;
    permcube pc;
    for (int e8_4 = 0; e8_4 < C8_4; e8_4++) {
        for (int epp1 = 0; epp1 < FACT4; epp1++) {
            for (int epp2 = 0; epp2 < FACT4; epp2++, ind++) {
                unpack_edgcoord(pc, e8_4, epp1, epp2);
                pc.set_perm(cp);
                int key = (1 << (cp.e[0]/2)) | (1 << (cp.e[3]/2)) | (1 << (cp.e[8]/2)) | (1 << (cp.e[11]/2));
                if (lookaside[key] < 0) lookaside[key] = ind;
                k3map[ind] = lookaside[key];
            }
        }
    }
}
#endif
```

84. We only need to call this once.

```
< Initialize the program 7 > +≡  
#ifdef LEVELCOUNTS  
    setupk3map();  
#endif
```

85. Next, we have the code that calculates the weight for the edge permutations from a particular coset.

```

<Utility functions 13> +=
#ifdef LEVELCOUNTS
void setup_levmul(const kocsymm &kc, const moveseq &moves)
{
    int x = kc.calc_symm();
    memset(levmul, 48/x, sizeof (levmul));
    int ind = 0;
    for (int e8_4 = 0; e8_4 < 70; e8_4++)
        for (int epp1 = 0; epp1 < FACT4; epp1++)
            for (int epp2 = 0; epp2 < FACT4; epp2++, ind++) {
                permcube pc;
                if (k3map[ind] != ind) {
                    levmul[ind] = levmul[k3map[ind]];
                }
                else {
                    cubepos cpt, cp2, cp3;
                    unpack_edgcoord(pc, e8_4, epp1, epp2);
                    pc.set_perm(cp2);
                    for (unsigned int i = 0; i < moves.size(); i++) cp2.move(moves[i]);
                    for (int i = 0; i < 8; i++) cp2.c[i] = cubepos::corner_val(i, 0);
                    cp2.invert_into(cpt);
                    cpt.remap_into(16, cp2);
                    kocsymm kc3(cp2);
                    kocsymm kc1;
                    kocsymm kc2(cpt);
                    kc2.canon_into(kc1);
                    kc3.canon_into(kc2);
                    cpt.remap_into(32, cp2);
                    kocsymm kct(cp2);
                    kct.canon_into(kc3);
                    if ((orderkc(kc2) < orderkc(kc1) & lookupkc(kc2)) ∨ (orderkc(kc3) <
                        orderkc(kc1) & lookupkc(kc3))) {
                        levmul[ind] = 0;
                    }
                    else {
                        int d = 1;
                        if (kc1 ≡ kc2) d++;
                        if (kc1 ≡ kc3) d++;
                        levmul[ind] = 48/(x * d);
                    }
                }
            }
    }
#endif

```

86. If we counted levels, give the results here right at the end.

```
<Cleanup 54> +=
```

```
#ifndef LEVELCOUNTS
```

```
    for (unsigned int i = 0; i < sizeof (sum_ulev)/sizeof (sum_ulev[0]); i++)
```

```
        if (sum_ulev[i] cout << "Level" << i << " count" << sum_ulev[i] << endl;
```

```
#endif
```

__builtin_prefetch: [24](#), [64](#).

a: [38](#), [41](#), [45](#).

ALLMOVEMASK: [14](#), [27](#), [70](#).

argc: [1](#), [3](#), [11](#), [53](#), [67](#), [77](#), [81](#).

argv: [1](#), [3](#), [11](#), [53](#), [67](#), [77](#), [81](#).

at: [47](#).

b: [62](#).

BANNER: [1](#), [7](#).

base: [37](#).

bc: [39](#), [40](#), [41](#).

bestsol: [75](#), [76](#), [80](#).

bestsollookup: [80](#).

bitcount: [63](#), [64](#).

bitp1: [15](#), [16](#), [20](#), [24](#), [46](#), [55](#), [57](#), [58](#), [60](#), [64](#), [74](#).

bitp2: [15](#), [16](#), [29](#), [46](#), [58](#), [60](#), [74](#).

bittoperm: [21](#), [22](#), [31](#), [55](#).

buf: [78](#).

B2: [31](#), [33](#).

c: [63](#), [64](#), [80](#).

calc_symm: [85](#).

calcneighbors: [45](#), [46](#), [58](#).

calloc: [16](#).

canon: [13](#), [25](#), [68](#).

canon_into: [80](#), [85](#).

CANONSEQSTART: [14](#), [27](#), [69](#), [70](#).

cbp: [24](#), [45](#), [55](#), [64](#).

check_integrity: [1](#).

checkabit: [62](#), [63](#), [64](#).

CHECKABITSIZE: [62](#), [63](#), [64](#).

cindex: [24](#), [64](#).

cnt: [80](#).

co: [80](#).

coord: [28](#), [41](#).

coparity: [41](#), [55](#).

corder_order: [46](#).

corner_order: [44](#).

corner_val: [85](#).

cornerorder: [44](#), [46](#), [56](#), [57](#), [59](#).

CORNERSYMM: [80](#).

coset_count: [76](#), [77](#), [81](#).

cosetstart: [8](#), [74](#).

countbits: [38](#), [58](#).

countbits2: [41](#), [58](#).

cout: [1](#), [7](#), [8](#), [14](#), [27](#), [52](#), [54](#), [60](#), [73](#), [74](#), [86](#).

cp: [52](#), [80](#), [83](#).

cperm: [41](#), [45](#), [46](#), [55](#), [58](#), [73](#).

cpp: [37](#).

cpt: [85](#).

cp2: [52](#), [80](#), [85](#).

cp3: [85](#).

cs_mask: [13](#), [25](#), [68](#), [69](#), [70](#).

csymm: [80](#).

ctp: [24](#), [45](#), [55](#), [64](#).

cubepos: [6](#), [8](#), [13](#), [25](#), [52](#), [68](#), [69](#), [70](#), [78](#), [80](#), [83](#), [85](#).

c12_8: [24](#), [36](#), [47](#), [64](#).

c8_12: [28](#).

C8_4: [36](#), [41](#), [83](#).

c8_4: [24](#), [45](#), [46](#), [55](#), [64](#).

c8_4_parity: [41](#), [55](#).

d: [14](#), [27](#), [73](#), [78](#), [85](#).

delta: [74](#).

did_a_prepas: [19](#), [20](#), [60](#), [74](#).

did_full_search: [73](#).

disable_prepas: [29](#), [30](#), [73](#).

do_prepas_work: [59](#), [60](#).

do_search_work: [72](#), [73](#).

docoset: [1](#), [74](#), [78](#), [81](#).

docoverelement: [78](#), [80](#).

dont_count_after_max_search: [66](#), [67](#), [73](#).

doouter: [46](#), [59](#).

doprepas: [60](#), [73](#).

dowork: [70](#), [72](#), [73](#).

dp: [46](#).

dst: [37](#), [42](#), [46](#).

duration: [1](#), [14](#), [27](#), [60](#), [73](#).

e: [46](#).

eb: [28](#).

ebp: [24](#), [28](#), [36](#), [64](#).

edata: [46](#).

EDGEOSYMM: [80](#).

EDGEPERM: [80](#).

eindex: [24](#), [64](#).

elemdata: [42](#), [46](#).

em: [28](#).

emp: [24](#), [31](#), [33](#), [55](#), [64](#).

emperm: [55](#).

end: [37](#), [80](#).

endl: [1](#), [7](#), [8](#), [14](#), [27](#), [52](#), [54](#), [60](#), [73](#), [74](#), [86](#).

enoughbits: [63](#), [66](#), [67](#), [73](#).

eo: [80](#).

eosymm: 80.
ep: 80.
eperm_map: 35, 36, 37.
epindex: 64.
epp1: 28, 36, 41, 55, 83, 85.
epp2: 28, 36, 41, 55, 83, 85.
epsymm: 80.
epsymm_compress: 28.
epsymm_expand: 28.
ep8_4: 55.
et: 24, 28, 36, 47, 64.
etp: 24, 28, 36, 64.
e8_4: 28, 36, 41, 83, 85.
e84: 47.
e84map: 42, 46.
FACT4: 15, 17, 21, 22, 24, 28, 31, 33, 36, 41, 45, 55, 64, 83, 85.
FACT8: 15, 16, 17, 19, 20, 35, 55, 56, 57, 64, 70, 73, 74, 82.
FASTCLEAN: 19, 20, 24, 60, 64, 65, 70, 74.
fast20: 66, 67, 73.
fclose: 54.
ffs: 13, 25, 55, 68.
find: 80.
finish_prepass_work: 58, 59.
first_coset: 76, 77, 81.
flush: 7, 60.
flushbit: 24, 27.
fopen: 53.
fprintf: 52.
freepage: 18, 58, 74.
from: 42, 46.
F2: 31.
genseqs: 80, 81.
get_global_lock: 51, 55, 57, 58, 63, 69, 70.
get_prepass_work: 57, 59.
get_search_work: 69, 70.
getclearedpage: 18, 20.
getpage: 18, 57.
global_depth: 10, 58, 60, 63, 70, 73.
hi: 80.
i: 7, 9, 20, 22, 31, 33, 34, 38, 40, 46, 47, 55, 57, 58, 59, 63, 70, 74, 85, 86.
identity_cube: 9.
identity_kc: 9, 13, 25, 68.
identity_pc: 9, 46.
in_Kociemba_group: 36, 45, 46.
ind: 36, 41, 83, 85.
init: 8, 80.
initorder: 46, 47.
innerloop3: 37, 46.
insert: 13.
invert_into: 85.
invert_sequence: 78.
j: 46, 47, 55.
k: 55.
kc: 13, 14, 25, 27, 68, 70, 78, 80, 85.
kct: 85.
kc1: 85.
kc2: 13, 25, 68, 80, 85.
kc3: 85.
key: 83.
kocsymm: 6, 9, 13, 14, 25, 27, 28, 35, 36, 45, 46, 68, 70, 78, 80, 85.
k3map: 82, 83, 85.
LEVELCOUNTS: 9, 17, 20, 39, 40, 41, 58, 60, 62, 63, 64, 70, 73, 82, 83, 84, 85, 86.
levmul: 41, 64, 82, 85.
levprev: 82.
levsum: 82.
levuniq: 82.
lo: 80.
local_bitflush: 63, 64, 70.
local_initialize: 63, 70.
local_probes: 63, 70.
local_setonebit: 64, 68.
local_touched: 64, 65, 70.
local_ulev: 63, 70.
lookaside: 83.
lookup: 13, 14, 25, 27, 68, 70, 73, 78.
lookupkc: 80, 85.
lowb: 33.
L2: 31, 33.
m: 80.
main: 1.
malloc: 18.
map: 79.
mask: 69, 70.
mask1: 38.
mask2: 38.
mask3: 38.
match: 80.
MAX_THREADS: 2, 3, 48, 60, 73.
maxdepth: 10, 11, 67, 73, 78.
maxsearchdepth: 10, 11, 14, 27, 58, 63, 67, 73.
memset: 18, 20, 47, 60, 65, 83, 85.
move: 9, 13, 25, 31, 33, 36, 45, 46, 47, 68, 70, 85.
movemask: 13, 25, 68.
movemask3: 25, 68.
movepair: 70.
moves: 78, 85.
moveseq: 6, 78, 85.
moveseq_string: 78.
moveseq16: 46, 47.

movestring: [1](#), [8](#), [78](#).
mul: [52](#).
mutex: [49](#), [50](#), [51](#).
mv: [13](#), [25](#), [36](#), [45](#), [46](#), [68](#).
mvi: [33](#), [36](#).
mvs: [33](#).
mv1: [69](#), [70](#).
mv2: [25](#), [68](#), [69](#), [70](#).
nd: [13](#), [25](#), [68](#).
need_count_bits: [58](#), [60](#), [66](#), [73](#).
neighbors: [46](#), [58](#).
new_canon: [13](#), [25](#), [68](#).
newmovemask: [13](#), [25](#), [68](#).
next_cs: [13](#), [25](#), [68](#), [69](#), [70](#).
NMOVES: [36](#), [45](#), [46](#), [69](#), [70](#).
numthreads: [2](#), [3](#), [60](#), [73](#).
oargc: [3](#), [7](#).
oargv: [3](#), [7](#).
off1: [41](#).
off2: [41](#).
okay: [80](#).
oldsingcount: [8](#), [74](#).
orderkc: [80](#), [85](#).
p: [33](#), [62](#).
p_thread: [60](#), [73](#).
pad: [48](#).
pageq: [18](#).
PAGESIZE: [15](#), [18](#), [38](#).
parity: [41](#).
parse_moveseq: [8](#).
par0: [55](#).
pbits: [55](#).
pc: [13](#), [14](#), [24](#), [25](#), [27](#), [28](#), [31](#), [33](#), [36](#), [45](#), [46](#), [47](#),
[52](#), [55](#), [64](#), [68](#), [70](#), [83](#), [85](#).
pc2: [13](#), [25](#), [45](#), [46](#), [47](#), [68](#).
pc3: [25](#), [68](#).
permcube: [6](#), [13](#), [14](#), [24](#), [25](#), [27](#), [28](#), [31](#), [36](#), [41](#),
[45](#), [46](#), [47](#), [52](#), [55](#), [64](#), [68](#), [70](#), [83](#), [85](#).
permtobit: [21](#), [22](#), [24](#), [31](#), [33](#), [64](#).
phase1prune: [1](#), [8](#), [13](#), [14](#), [25](#), [27](#), [68](#), [70](#), [73](#), [78](#), [80](#).
pop_back: [18](#).
PREPASS_MOVES: [35](#), [37](#), [42](#), [46](#), [58](#), [60](#).
prev_ulev: [73](#).
prevlev: [14](#), [27](#), [73](#).
probes: [13](#), [14](#), [17](#), [24](#), [27](#), [70](#), [73](#).
progstart: [1](#).
pthread: [72](#).
pthread_create: [60](#), [73](#).
pthread_join: [60](#), [73](#).
pthread_mutex_init: [50](#).
pthread_mutex_lock: [51](#).
pthread_mutex_t: [49](#).
pthread_mutex_unlock: [51](#).
pthread_t: [60](#), [73](#).
pthreads: [59](#).
push_back: [18](#), [78](#).
p2: [41](#).
q: [63](#).
r: [18](#), [38](#), [41](#), [46](#), [57](#), [59](#), [69](#).
rearrange: [32](#), [33](#), [34](#), [37](#).
release_global_lock: [51](#), [55](#), [57](#), [58](#), [63](#), [69](#), [70](#).
remap_into: [80](#), [85](#).
repcp: [6](#), [9](#), [52](#).
repkc: [6](#), [9](#), [12](#), [14](#), [26](#), [27](#), [70](#), [73](#).
reppc: [6](#), [9](#), [12](#), [26](#), [70](#).
repseq: [6](#), [8](#), [9](#).
rv2: [41](#).
r2: [41](#).
R2: [31](#), [33](#).
s: [69](#), [70](#), [72](#).
saveb: [23](#), [24](#).
savep: [23](#), [24](#).
search: [68](#), [70](#), [71](#), [73](#).
search_terminated_early: [63](#), [66](#), [68](#), [73](#).
search_work_seq: [69](#), [73](#).
seq: [1](#), [8](#), [78](#).
set: [6](#), [15](#).
set_coset: [80](#).
set_perm: [52](#), [83](#), [85](#).
setbit: [31](#).
setonebit: [24](#), [25](#).
setup_levmul: [9](#), [85](#).
setupk3map: [83](#), [84](#).
showsing: [52](#), [55](#).
showunset: [55](#), [58](#), [73](#).
singcount: [8](#), [52](#), [54](#), [74](#).
singfile: [52](#), [53](#), [54](#), [74](#).
singlevel: [52](#), [53](#), [58](#), [73](#).
Singmaster_string: [52](#).
size: [9](#), [14](#), [18](#), [78](#), [85](#).
skipwrite: [4](#), [5](#), [8](#).
slow: [10](#), [11](#), [12](#), [26](#), [71](#).
slowsearch1: [12](#), [13](#), [14](#).
slowsearch2: [25](#), [26](#), [27](#), [68](#).
solve: [78](#).
SQMOVES: [32](#), [33](#).
srcs: [37](#).
sscanf: [3](#), [11](#), [53](#), [67](#), [77](#).
std: [1](#).
strcpy: [78](#).
STRIDE: [43](#), [46](#), [57](#), [59](#).
sum_ulev: [17](#), [73](#), [86](#).
swap: [60](#).
symcount: [79](#), [80](#).

s1: [38](#).
s2: [38](#).
t: [55](#).
TARGET: [17](#), [66](#), [73](#).
tcnt: [80](#).
tcpemr: [46](#).
this_level_did_prepas: [61](#), [68](#), [73](#).
this_ulev: [58](#), [63](#).
this_uniq: [63](#).
thisblock: [58](#).
thislev: [14](#), [27](#), [73](#).
thisulev: [73](#).
ti: [60](#), [73](#).
tmp: [8](#).
togo: [13](#), [25](#), [68](#).
tot: [80](#).
TOTALCOSETS: [76](#), [77](#), [80](#), [81](#).
touched: [19](#), [20](#), [24](#), [70](#), [74](#).
tscheunemann: [82](#).
tval: [37](#).
TWISTS: [31](#).
uniq: [14](#), [17](#), [20](#), [24](#), [27](#), [58](#), [60](#), [63](#), [73](#).
uniq_ulev: [17](#), [20](#), [58](#), [60](#), [70](#), [73](#).
unpack_edgcoord: [28](#), [36](#), [47](#), [55](#), [83](#), [85](#).
use_count: [56](#), [58](#), [60](#).
used: [47](#).
U2: [78](#).
vector: [15](#), [18](#).
verbose: [2](#), [3](#), [7](#), [14](#), [27](#), [73](#).
v1: [41](#).
v2: [41](#).
w: [41](#), [72](#).
walltime: [1](#), [8](#), [74](#).
wb2: [37](#).
weight: [62](#), [63](#), [64](#).
wf2: [37](#).
wl2: [37](#).
work_done: [56](#), [57](#), [60](#).
worker_thread: [48](#), [72](#).
workers: [48](#), [73](#).
world: [6](#), [13](#), [14](#).
wr2: [37](#).
w1: [38](#).
w2: [38](#).
w3: [38](#).
x: [85](#).

- ⟨ Cleanup 54, 86 ⟩ Used in section 1.
- ⟨ Data declarations 2, 4, 6, 10, 15, 17, 19, 21, 23, 29, 32, 35, 39, 43, 44, 49, 56, 61, 66, 75, 76, 79, 82 ⟩ Used in section 1.
- ⟨ Handle one coset from *movestring* 8, 9, 12, 20, 26, 71, 74 ⟩ Used in section 1.
- ⟨ Handle the work 81 ⟩ Used in section 1.
- ⟨ Initialize the program 7, 16, 22, 31, 33, 34, 36, 40, 47, 50, 84 ⟩ Used in section 1.
- ⟨ More arguments 5, 11, 30, 53, 67, 77 ⟩ Used in section 3.
- ⟨ More thread initialization 65 ⟩ Used in section 63.
- ⟨ Parse arguments 3 ⟩ Used in section 1.
- ⟨ Threading objects 48, 72, 73 ⟩ Used in section 1.
- ⟨ Utility functions 13, 14, 18, 24, 25, 27, 28, 37, 38, 41, 42, 45, 46, 51, 52, 55, 57, 58, 59, 60, 62, 69, 78, 80, 83, 85 ⟩ Used in section 1.
- ⟨ Worker thread object contents 63, 64, 68, 70 ⟩ Used in section 48.
- ⟨ `hcset.cpp` 1 ⟩

HCOSET

	Section	Page
Introduction	1	1
Prepass	29	12
The inner loop	37	15
Threading	48	21
Coset cover	75	36