1. Introduction. Phase two of Kociemba's two-phase algorithm involves finding a solution to a position in the group H generated by $\{U, F2, R2, D, B2, L2\}$. This file constructs a pruning table for that group.

This group has size $12! \cdot 12! \cdot 4!/2$ or 19,508,428,800 and distance values from 0 to 18. If we allocate a byte per entry, we'd need 20GB of core for the pruning table—this is unreasonably large.

The purpose of this pruning table in the two-phase algorithm is two-fold. Remember that during the phase one search, every sequence found leads to some position p in the H group. The first use of the phase two table is to look up an exact or conservative estimate of the distance remaining to solved, and add that to the length of the phase one sequence, and see if that sum is less than the length of the current best solution so far. If it is not, we can immediately reject that phase one solution and go on to the next.

If the phase one solution was not rejected, then we need to use the pruning table to find a phase two solution, completing the overall solution to the original position. If the pruning table always has exact distances, this is guaranteed to succeed. If the pruning table only gives conservative approximations of the distance, this may fail.

The phase two solution is found using standard iterated depth-first search, at each node of the tree looking up the current position in the pruning table and rejecting that branch of the search if the distance from the table indicates there is no solution in the given number of moves.

Our main concern is reducing the size of the table. The first technique is to use the 16-way symmetry of H to reduce the table by only storing a single representative of each symmetry class. This reduces the memory requirements from about 20GB down to about 1.2GB; this is still quite large.

We do not require an exact table, however, so we can hash the state down into a smaller range. Remember that all positions in H have a solved orientation (subject to the conventions we laid out in **cubepos**) and the middle four cubies are in the middle four slots. The group is defined by the permutation of the eight corners, the permutation of the eight edges whose home position is not in the middle layer, and the permutation of the four edges whose home position is the middle layer, subject to the requirement that the parity of the overall permutation of the edges must match the parity of the permutation of the corners.

If you hash the state into a smaller range, thus mapping multiple elements of the group to the same table entry, you must put the smallest distance of any of the elements that map into that entry, because the pruning table must give a conservative estimate of the distance. For most ways to reduce the size of the table, this gives a marked reduction in average depth, and thus a marked decrease in the effectiveness of the table; this is the tradeoff for saving space. In this case, however, we can eliminate the permutation of the middle edges from consideration without significantly decreasing the average distance; the resulting table is $4!/2$ or 12 times smaller.

The reason this is so is that most optimal sequences that solve a particular position in H can be transformed into other sequences of the same length that solve other positions in H, but positions that differ only in the middle edge permutation These other sequences are not always optimal, but frequently they are. To understand why this is so, consider that the moves U1, U2, U3, D1, D2, and D3 do not affect the middle edges at all. Indeed, the move U1 is the same as the move D1 followed by a whole cube rotation clockwise from the top, if we only consider the top and bottom cubies and ignore the middle edges and center cubies. So any sequence that looks like $\alpha U1\beta$ can be transformed into a different sequence $\alpha D1\beta'$ that has the same overall effect on the top and bottom edges and corners (plus a whole cube rotation). Since a typical solution to a position in H has many U and D moves that can be so rotated, there are many distinct sequences that can be directly derived from the optimal sequence that generate other moves in H that differ only in whole cube rotations and effect on the middle layer. There are only four possible whole cube rotation states if you limit yourself to rotations around the up-down axis, so even when you limit yourself to those sequences that end up with the center cubies where you started, you frequently have many sequences left.

The following table gives the distance distribution both for the group H and the smaller group H' generated by actions from H on a representation that omits the middle edges altogether. You'll note that the percentages and the average depth is reasonably close.

Distance	H		H'	
0	1		1	
1	10		10	
2	67		67	
3	456		420	
4	3,079		2,335	
5	19,948		12,260	
6	123,074		61,038	
7	736,850		291,004	
8	4,185,118		1,327,429	0.1%
9	22,630,733	0.1%	5,821,374	0.4%
10	116,767,872	0.6%	24,141,784	1.5%
11	552,538,680	2.8%	89,480,354	5.5%
12	2,176,344,160	11.2%	262,907,144	16.2%
13	5,627,785,188	28.8%	485,409,604	29.9%
14	7,172,925,794	36.8%	508,704,668	31.3%
15	3,608,731,814	18.5%	232,904,952	14.3%
16	224,058,996	1.1%	14,508,468	0.9%
17	1,575,608		129,376	
18	1,352		112	
Average	13.58		13.29	

A 12-fold decrease in table size, with only a 0.29 reduction in average distance, is remarkable. The only similar reduction I'm aware of with Rubik's cube is the reduction from corners-with-centers to corners-without-centers (the 2x2x2), and the reasoning is similar.

The final improvement we make is to use only four bits for each entry, which we use to represent values 1 through 16. A value of zero can be inferred by inspection, and there are very few positions at 17 or greater.

This class depends on **cubepos** and **kocsymm**, so if you haven't read those yet, now might be a good time.

⟨ `phase2prune.h` 1 ⟩ ≡
```
#ifndef PHASE2PRUNE_H
#define PHASE2PRUNE_H
#include "kocsymm.h"
```
See also section 2.

2. We have an initialization routine and a lookup routine. The initialization routine is not called at construction time, so you can declare this class statically but control when it is initialized. (Initialization may include generation or loading of the pruning table, which can be a lengthy operation.) There are no actual fields or non-static methods; everything in this class is static.

⟨ `phase2prune.h` 1 ⟩ +≡
```
  const int FACT8 = 40320;
  class phase2prune {
  public:
    static void init(int suppress_writing = 0);
    static int lookup(const cubepos &cp);
    static int lookup(const permcube &pc);
    ⟨ Method declarations 12 ⟩
    ⟨ Data declarations 6 ⟩
  };
#endif
```

3. For the body of the initialization routine, we need to declare the C++ file at last. In our initialization routine, we pass a flag indicating whether or not to suppress the writing of the pruning table to disk.

```
⟨ phase2prune.cpp 3 ⟩ ≡
#include "phase2prune.h"
#include <iostream>
#include <cstdio>
  using namespace std;
  ⟨ Data instantiations 4 ⟩
  ⟨ Utility methods 5 ⟩
  ⟨ Method bodies 11 ⟩
  void phase2prune::init(int suppress_writing)
  {
    static int initialized = 0;
    if (initialized) return;
    initialized = 1;
    ⟨ Initialize the instance 8 ⟩
  }
```

4. When we lookup a **cubepos** in this pruning table, the first thing to do is to compute a canonical representative. We cannot use the normal **cubepos** canonicalization, because that takes orientation into account, and this pruning table must not. Instead, we use the corner permutation (as calculated by **permcube**) to select a canonical coordinate. For each corner permutation, we need to store the $m \in M$ remapping, the reduced corner permutation coordinate, and the set of bits that give what remappings generate that minimum coordinate. This is like *corner_mapinfo* in **kocsymm**, except the minimum coordinate will not fit in an **unsigned char**. This array has size 160K, but this is dwarfed by the actual pruning table itself.

When we generate the pruning table, we will use the corner calculates as the outer loop (since that's what we are remapping by) and use the edge permutation as the inner loop. To make this reasonably fast, we need a table that can remap an edge up/down permutation. This is not a small table, but it's also dwarfed by the pruning table.

```
⟨ Data instantiations 4 ⟩ ≡
  struct corner_reduce {
    unsigned char m, parity;
    lookup_type c, minbits;
  } corner_reduction[FACT8];
  lookup_type edgeud_remap[KOCSYMM][FACT8];
```

See also sections 7 and 18.

This code is used in section 3.

5. Filling out the corner reduction array is fairly straightforward; we use the existing classes **cubepos** and **permcube** to do the work. First we need a particular ordering of the corner elements of **permcube**; this is somewhat arbitrary.

```
⟨ Utility methods 5 ⟩ ≡
  inline int corner_coordinate(const permcube &pc)
  {
    return (pc.c8_4 * FACT4 + pc.ctp) * FACT4 + pc.cbp;
  }
  inline int edge_coordinate(const permcube &pc)
  {
    return (permcube::c12_8[pc.et] * FACT4 + pc.etp) * FACT4 + pc.ebp;
  }
```

See also section 19.

This code is used in section 3.

6. Once we know how many symmetry-reduced coordinates there are, we also know how much memory we need. We declare a variable to hold that value here, as well as our memory array pointer.

```
⟨ Data declarations 6 ⟩ ≡
  static int cornermax;
  static unsigned int memsize;
  static unsigned int *mem;
```

See also section 17.

This code is used in section 2.

7. We need to declare all of these instances.

```
⟨ Data instantiations 4 ⟩ +≡
  int phase2prune::cornermax;
  unsigned int phase2prune::memsize;
  unsigned int *phase2prune::mem;
```

8. Now we try all possibilities.

```
⟨ Initialize the instance 8 ⟩ ≡
  cornermax = 0;
  for (int c8_4 = 0; c8_4 < C8_4; c8_4++)
    for (int ctp = 0; ctp < FACT4; ctp++)
      for (int cbp = 0; cbp < FACT4; cbp++) {
        permcube pc;
        pc.c8_4 = c8_4;
        pc.ctp = ctp;
        pc.cbp = cbp;
        int oc = corner_coordinate(pc);
        int minc = oc;
        int minm = 0;
        int minbits = 1;
        cubepos cp;
        pc.set_perm(cp);
        for (int m = 1; m < 16; m++) {
          cubepos cp2;
          cp.remap_into(m, cp2);
          permcube pc2(cp2);
          int tc = corner_coordinate(pc2);
          if (tc < minc) {
            minc = tc;
            minm = m;
            minbits = 1 ≪ m;
          }
          else if (tc ≡ minc) minbits |= 1 ≪ m;
        }
        corner_reduce &cr = corner_reduction[oc];
        if (oc ≡ minc) cr.c = cornermax++;
        cr.m = minm;
        cr.c = corner_reduction[minc].c;
        cr.minbits = minbits;
        cr.parity = (permcube :: c8_4_parity[c8_4] + ctp + cbp) & 1;
      }
```

See also sections 9, 10, and 23.

This code is used in section 3.

9. Next we initialize the remapping of the edge coordinates.

```
⟨ Initialize the instance 8 ⟩ +≡
  int at = 0;
  cubepos cp, cp2;
  for (int e8_4 = 0; e8_4 < C8_4; e8_4++) {
    permcube pc;
    pc.et = permcube::c8_12[e8_4];
    pc.eb = kocsymm::epsymm_compress[#f0f − kocsymm::epsymm_expand[pc.et]];
    for (int etp = 0; etp < FACT4; etp++) {
      pc.etp = etp;
      for (int ebp = 0; ebp < FACT4; ebp++, at++) {
        pc.ebp = ebp;
        for (int m = 0; m < KOCSYMM; m++) {
          pc.set_edge_perm(cp);
          cp.remap_into(m, cp2);
          permcube pc2(cp2);
          edgeud_remap[m][at] = edge_coordinate(pc2);
        }
      }
    }
  }
```

10. We continue our initialization with allocation of the memory array. We store two bytes per entry.

```
⟨ Initialize the instance 8 ⟩ +≡
  memsize = cornermax * (FACT8/2);
  mem = (unsigned int *) malloc(memsize);
  if (mem ≡ 0)
    error ("!␣no␣memory␣in␣phase2prune") ;
```

11. Looking up a position. We write our lookup routine carefully, inlining the portions of the *remap* and coordinate calculation code we really need. Even with this care, this code will probably encounter numerous cache misses in a single lookup because of the large tables in use.

⟨ Method bodies 11 ⟩ ≡

```
int phase2prune::lookup(const cubepos &cp)
{
  permcube pc(cp);
  return lookup(pc);
}
int phase2prune::lookup(const permcube &pc)
{
  int cc = corner_coordinate(pc);
  corner_reduce &cr = corner_reduction[cc];
  int off = cr.c * FACT8 + edgeud_remap[cr.m][edge_coordinate(pc)];
  int r = (mem[off >> 3] >> (4 * (off & 7))) & #f;
  if (r == 0 ∧ pc == identity_pc) return 0;
  else return r + 1;
}
```

See also sections 13, 20, 21, 22, and 25.

This code is used in section 3.

12. Generating the pruning table. We need a routine to generate the pruning table. To do this, we initialize the solved position to the value 0 and all other positions to the value 15. Then for values of d from 0 to 13, we find all positions at that depth, compute their neighbors, and if their neighbors are so far unseen, set the depth to $d+1$. Since we share the representations of distances 0 and 1 using the value 0 in the array, we actually initialize the start with a value of 1, and after the first iteration, we reset that back to 0.

```
⟨ Method declarations 12 ⟩ ≡
  static void gen_table( );
  static int read_table( );
  static void write_table( );
  static void check_integrity( );
```

See also section 24.

This code is used in section 2.

13. There is one major subtlety when generating pruning tables that depend on symmetry like this one: we don't want to have to do a full symmetry reduction on every lookup; we just want to reduce symmetry by the corner permutation. The tricky thing then is whenever our destination corner permutation has any symmetry, we must be sure to compute and update all relevant symmetry values for that element.

```
⟨ Method bodies 11 ⟩ +≡
  void phase2prune::gen_table( )
  {
    memset(mem, 255, memsize);
    cout ≪ "Gen␣phase2" ≪ flush;
    mem[0] &= ∼14;
    int seen = 1;
    for (int d = 0; d < 15; d++) {
      unsigned int seek = (d ? d − 1 : 1);
      int newval = d;
      for (int c8_4 = 0; c8_4 < C8_4; c8_4 ++)
        for (int ctp = 0; ctp < FACT4; ctp++)
          for (int cbp = 0; cbp < FACT4; cbp++) {
            permcube pc;
            pc.c8_4 = c8_4;
            pc.ctp = ctp;
            pc.cbp = cbp;
            int oc = corner_coordinate(pc);
            corner_reduce &cr = corner_reduction[oc];
            if (cr.minbits & 1) {
              ⟨ Iterate over all moves 14 ⟩;
            }
          }
#ifndef QUARTER
      if (d ≡ 0) mem[0] &= ∼15;
#endif
      cout ≪ "␣" ≪ d ≪ flush;
    }
    cout ≪ "␣done." ≪ endl ≪ flush;
  }
```

14. Try all the different moves from this corner position. Note that we only handle half turn metric at the moment. In any case, hoist the destination corner permutation computation to the top of the loop. We also calculate offsets from both the source and the destination rows.

```
⟨ Iterate over all moves 14 ⟩ ≡
  permcube pc2, pc3, pc4;
  cubepos cp2, cp3;
  int off = corner_reduction[oc].c * (FACT8/8);
  for (int mv = 0; mv < NMOVES; mv++) {
    if (¬kocsymm::in_Kociemba_group(mv)) continue;
    pc2 = pc;
    pc2.move(mv);
    int dest_off = corner_coordinate(pc2);
    corner_reduce &cr = corner_reduction[dest_off];
    int destat = cr.c * (FACT8/8);
    for (int m = cr.m; (1 ≪ m) ≤ cr.minbits; m++)
      if ((cr.minbits ≫ m) & 1) {⟨ Scan one row 15 ⟩}
  }
```

This code is used in section 13.

15. When we scan a row, we need to work on the 8! possible permutations of the edge cubies, doing a move and a remapping on each. For efficiency we embed parts of the **permcube** move routine in here. We accelerate the scan if we see a bunch of unset values.

```
⟨ Scan one row 15 ⟩ ≡
  int at = 0;
  for (int e8_4 = 0; e8_4 < C8_4; e8_4++) {
    int et = permcube::c8_12[e8_4];
    int t1 = permcube::eperm_move[et][mv];
    int eb = kocsymm::epsymm_compress[#f0f − kocsymm::epsymm_expand[et]];
    int t2 = permcube::eperm_move[eb][mv] & 31;
    int dst1 = permcube::c12_8[t1 ≫ 5] * 24 * 24;
    t1 &= 31;
    for (int etp = 0; etp < FACT4; etp++)
      for (int ebp = 0; ebp < FACT4; ebp++, at++) {
        if (mem[off + (at ≫ 3)] ≡ #ffffffff) {
          ebp += 7;
          at += 7;
        }
        else if (((mem[off + (at ≫ 3)] ≫ (4 * (at & 7))) & #f) ≡ seek) {⟨ Handle one position 16 ⟩}
      }
  }
```

This code is used in section 14.

16. We've found a single position at the distance we seek. Find all of its neighbors, and check if this is a newly reached value.

```
⟨ Handle one position 16 ⟩ ≡
  int etp1 = permcube::s4mul[etp][t1];
  int ebp1 = permcube::s4mul[ebp][t2];
  int dat = edgeud_remap[m][dst1 + etp1 * 24 + ebp1];
  int val = (mem[destat + (dat >> 3)] >> (4 * (dat & 7))) & #f;
  if (val == #f) {
    mem[destat + (dat >> 3)] -= (#f - newval) << (4 * (dat & 7));
    seen++;
  }
```

This code is used in section 15.

17. Disk I/O. The pruning table takes a fair amount of time to generate (about 40 seconds on modern hardware), and I'm frequently impatient, so we add some routines to read and write the pruning table to a file on disk.

```
⟨ Data declarations 6 ⟩ +≡
  static const char *const filename;
  static int file_checksum;
```

18. We choose the filename below, to indicate version 1 of the phase 2 pruning data, halfturn metric.

```
⟨ Data instantiations 4 ⟩ +≡
  const char *const phase2prune::filename = "p2p1h.dat";
  int phase2prune::file_checksum;
```

19. We need a routine to do a checksum of the file, to verify integrity. We use a simplistic hash function. We make it file static; we might use a different one in a different file.

```
⟨ Utility methods 5 ⟩ +≡
  static int datahash(unsigned int *dat, int sz, int seed)
  {
    while (sz > 0) {
      sz -= 4;
      seed = 37 * seed + *dat++;
    }
    return seed;
  }
```

20. Our read routine is straightforward; we return 1 on success, and 0 on failure. We could read the whole thing at once and then checksum it afterwards, but we choose to do it in chunks that fit in cache. The `"rb"` in the *fopen* call is to force binary mode on Windows platforms.

```
⟨ Method bodies 11 ⟩ +≡
  const int CHUNKSIZE = 65536;
  int phase2prune::read_table( )
  {
    FILE *f = fopen(filename, "rb");
    if (f ≡ 0) return 0;
    int togo = memsize;
    unsigned int *p = mem;
    int seed = 0;
    while (togo > 0) {
      unsigned int siz = (togo > CHUNKSIZE ? CHUNKSIZE : togo);
      if (fread(p, 1, siz, f) ≠ siz) {
        cerr ≪ "Out␣of␣data␣in␣" ≪ filename ≪ endl;
        fclose(f);
        return 0;
      }
      seed = datahash(p, siz, seed);
      togo −= siz;
      p += siz/sizeof(unsigned int);
    }
    if (fread(&file_checksum, sizeof(int), 1, f) ≠ 1) {
      cerr ≪ "Out␣of␣data␣in␣" ≪ filename ≪ endl;
      fclose(f);
      return 0;
    }
    fclose(f);
    if (file_checksum ≠ seed) {
      cerr ≪ "Bad␣checksum␣in␣" ≪ filename ≪ ";␣expected␣" ≪ file_checksum ≪ "␣but␣saw␣" ≪
          seed ≪ endl;
      return 0;
    }
    return 1;
  }
```

21. Our write routine is the converse of the above. We checksum as we write. Any error is fatal. The `"wb"` in the *fopen* call is to force binary mode on Windows platforms.

```
⟨ Method bodies 11 ⟩ +≡
  void phase2prune::write_table( )
  {
    FILE *f = fopen(filename, "wb");
    if (f ≡ 0)
      error ("! cannot write pruning file to current directory") ;
    if (fwrite(mem, 1, memsize, f) ≠ memsize)
      error ("! error writing pruning table") ;
    if (fwrite(&file_checksum, sizeof(int), 1, f) ≠ 1)
      error ("! error writing pruning table") ;
    fclose(f);
  }
```

22. We add a routine to check the integrity of the pruning table, perhaps at the end of a long run. Any error is fatal.

```
⟨ Method bodies 11 ⟩ +≡
  void phase2prune::check_integrity( )
  {
    if (file_checksum ≠ datahash(mem, memsize, 0))
      error ("! integrity of pruning table compromised") ;
    cout ≪ "Verified integrity of phase two pruning data: " ≪ file_checksum ≪ endl;
  }
```

23. We now finish our initialization with the routines that read and/or generate the file.

```
⟨ Initialize the instance 8 ⟩ +≡
  if (read_table( ) ≡ 0) {
    gen_table( );
    file_checksum = datahash(mem, memsize, 0);
    if (¬suppress_writing) write_table( );
  }
```

24. We need a solver for random positions. It takes a maximum distance for which the solution is useful. If there is no solution, it returns an empty vector (it's up to you to distinguish the case where the position is already solved). We also declare a utility routine that actually does the recursion.

```
⟨ Method declarations 12 ⟩ +≡
  static moveseq solve(const permcube &pc, int maxlen = 30);
  static moveseq solve(const cubepos &cp, int maxlen = 30)
  {
    permcube pc(cp);
    return solve(pc, maxlen);
  }
  static int solve(const permcube &pc, int togo, int canonstate, moveseq &seq);
```

25. And here we have the standard implementation of iterated depth-first search. The magic $^\circ 227227227$ below filters out moves that are not in H all at once in the half turn metric.

```
⟨ Method bodies 11 ⟩ +≡
  moveseq phase2prune::solve(const permcube &pc, int maxlen)
  {
    moveseq r;
    for (int d = lookup(pc); d ≤ maxlen; d++)
      if (solve(pc, d, CANONSEQSTART, r)) {
        reverse(r.begin( ), r.end( ));
        break;
      }
    return r;
  }
  int phase2prune::solve(const permcube &pc, int togo, int canonstate, moveseq &r)
  {
    if (lookup(pc) > togo) return 0;
    if (pc ≡ identity_pc) return 1;
    if (togo-- ≤ 0) return 0;
    permcube pc2;
    int mask = cubepos::cs_mask(canonstate) & ^227227227;
    while (mask) {
      int ntogo = togo;
      int mv = ffs(mask) − 1;
      mask &= mask − 1;
      pc2 = pc;
      pc2.move(mv);
      if (solve(pc2, ntogo, cubepos::next_cs(canonstate, mv), r)) {
        r.push_back(mv);
        return 1;
      }
    }
    return 0;
  }
```

26. Test routine.

```
⟨ phase2prune_test.cpp 26 ⟩ ≡
#include "phase2prune.h"
#include <iostream>
  using namespace std;
  char buf[4096];
  int main(int argc, char *argv[])
  {
    if (lrand48() ≡ 0) srand48(time(0));
    phase2prune::init(0);
    phase2prune::check_integrity();
    cubepos cp;
    for (int i = 0; i < 100000; i++) {
      char *tmp;
      int mv = random_move();
      if (kocsymm::in_Kociemba_group(mv)) {
        cp.movepc(mv);
      }
      int lookd = phase2prune::lookup(cp);
      cout ≪ "Distance␣" ≪ lookd ≪ endl;
      moveseq s = phase2prune::solve(cp);
      cubepos cpt = cp;
      for (unsigned int j = 0; j < s.size(); j++) cpt.movepc(s[j]);
      cubepos::append_moveseq(tmp = buf, s);
      cout ≪ "Solution␣length␣" ≪ s.size() ≪ "␣" ≪ buf ≪ endl;
      if (cpt ≠ identity_cube)
        error ("!␣bad␣solve") ;
      if ((unsigned int) lookd > s.size())
        error ("!␣solution␣too␣short") ;
    }
  }
```

⟨ Data declarations 6, 17 ⟩ Used in section 2.
⟨ Data instantiations 4, 7, 18 ⟩ Used in section 3.
⟨ Handle one position 16 ⟩ Used in section 15.
⟨ Initialize the instance 8, 9, 10, 23 ⟩ Used in section 3.
⟨ Iterate over all moves 14 ⟩ Used in section 13.
⟨ Method bodies 11, 13, 20, 21, 22, 25 ⟩ Used in section 3.
⟨ Method declarations 12, 24 ⟩ Used in section 2.
⟨ Scan one row 15 ⟩ Used in section 14.
⟨ Utility methods 5, 19 ⟩ Used in section 3.
⟨ `phase2prune.cpp` 3 ⟩
⟨ `phase2prune.h` 1, 2 ⟩
⟨ `phase2prune_test.cpp` 26 ⟩

PHASE2PRUNE