1. Introduction. Phase one of Kociemba's two-phase algorithm involves finding a sequence of moves that takes an arbitrary position into the H group, generated by $\{U, F2, R2, D, B2, L2\}$. This pruning table is the heart of both the general Kociemba two-phase algorithm and the coset solver based on it, so careful attention to performance must be paid.

Most pruning tables just give an estimate of the depth of the current position. For this particular pruning table, we want to not only give the exact depth (to reduce false search paths), but also give information on precisely which moves take you closer to solved. By doing this, we can eliminate almost all false paths in the search. Making the pruning table larger by increasing the size of each entry does not slow things down much, because a cache miss is a cache miss whether you access a bit of the cache line or all 32 bytes, and the performance of this pruning table is dominated by the cache (and TLB) misses it incurs.

The Schreier coset graph, as implemented in the **kocsymm** class, has size $3^7 \cdot 2^{11} \cdot \binom{12}{4}$ which is 2,217,093,120 entries. This coset has 16-way symmetry, so we reduce the size of the pruning table to about 170,000,000 entries; the remapping of **kocsymm** is reasonably fast requiring only small tables. (This is a bit more than what you would expect, because we only do an approximate reduction by symmetry for speed, and thus have a few more table entries.)

Each entry should contain the depth (which can range from 0 to 12) as well as information on whether each move brings you closer to solved, keeps you the same distance from solved, or takes you further away from solved. (Solved here means in the group H, not completely solved). Normally in the half turn metric this would require more than 32 bits per entry ($3^{18} \cdot 13 > 2^{32}$) but we can reduce this pretty easily. For a particular face, you would expect there to be 27 possibilities, a factor of three contributed by each possible move of that face. Consider the moves U1 and U2. If U1 brings you closer to solved, then U2 cannot bring you further from solved, because the sequence U2U3 is identical to U1. Thus, moves that share the same face twist can differ in their impact on distance by at most one. This eliminates 12 of the possibilities, leaving only 15 possibilities for each face. This requires four bits, so we can fit each entry in 32 bits: 4 bits for each of the six faces, and another four bits for the distance.

In theory with this information we do not even need to store the distance, since we can solve each position using just the incremental information to obtain the initial distance, and then we can keep track of the actual distance incrementally. For the moment, for simplicity, we use four-byte entries with explicit distance. We'll consider how to take advantage of some of the extra bits later.

Our basic API is straightforward. We have an initialization routine, and a lookup method that takes the position to look up, the current number of moves remaining for the current search, and by reference a variable to return a bitmask of which moves to consider for the next level of search. Everything in this class is static; there are no instance methods or fields.

```
⟨ phase1prune.h 1 ⟩ ≡
#ifndef PHASE1PRUNE_H
#define PHASE1PRUNE_H
#include "kocsymm.h"
  ⟨ Constant declarations 6 ⟩;
  class phase1prune {
  public:
    static void init(int suppress_writing = 0);
    static int lookup(const kocsymm &kc, int &mask);
    ⟨ Method declarations 8 ⟩
    ⟨ Data declarations 3 ⟩
  };
#endif
```

2. Our initialization routine is the first component of our C++ file. It is protected against multiple invocation by a method-local static.

```
⟨ phase1prune.cpp 2 ⟩ ≡
#include "phase1prune.h"
#include <iostream>
#include <cstdio>
  using namespace std;
  ⟨ Data instantiations 4 ⟩
  ⟨ Utility functions 5 ⟩
  ⟨ Method bodies 9 ⟩
  void phase1prune::init(int suppress_writing)
  {
    static int initialized = 0;
    if (initialized) return;
    initialized = 1;
    ⟨ Initialize the instance 7 ⟩
  }
```

See also section 22.

3. We need a memory array to store the result, a static variable to hold the size of the array, and another static variable to hold the checksum of the array. We also need a filename for the file.

```
⟨ Data declarations 3 ⟩ ≡
  static unsigned int memsize;
  static unsigned char *mem;
  static int file_checksum;
  static const char *const filename;
```

This code is used in section 1.

4. All these must be instantiated. For the filename, we use the name given below to indicate phase 1 pruning data, halfturn metric.

```
⟨ Data instantiations 4 ⟩ ≡
  unsigned int phase1prune::memsize;
  unsigned char *phase1prune::mem;
  int phase1prune::file_checksum;
  const char *const phase1prune::filename = "p1p1h.dat";
```

See also section 18.

This code is used in section 2.

5. We need a routine to do a checksum of the file, to verify integrity. We use a simplistic hash function. Note that this function expects an array pointing to **unsigned int**, not **unsigned char** like our *mem* array; we use a bit of coercion to make this work. These data files are not interchangeable across different endian architectures; the checksums will be different. We make it file static in case we use a different one somewhere else.

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

This code is used in section 2.

6. We use a constant to indicate the count of bytes needed per entry.

```
⟨ Constant declarations 6 ⟩ ≡
  const int BYTES_PER_ENTRY = 4;
```

This code is used in section 1.

7. Our initialization routine calculates the memory size and allocates the array. We use a single contiguous array; even on 32-bit architectures it should be straightforward to get about 700MB of contiguous memory—at least early in the program's lifetime.

```
⟨ Initialize the instance 7 ⟩ ≡
  memsize = BYTES_PER_ENTRY * CORNERRSYMM * EDGEOSYMM * EDGEPERM;
  mem = (unsigned char *) malloc(memsize);
  if (mem ≡ 0)
    error ("! no memory") ;
```

See also sections 15, 20, and 21.

This code is used in section 2.

8. We need methods to generate the table, read it, write it, and check it for integrity.

```
⟨ Method declarations 8 ⟩ ≡
  static void gen_table( );
  static int read_table( );
  static void write_table( );
  static void check_integrity( );
```

See also section 16.

This code is used in section 1.

9. Our generate table routine is next. We iterate over the distances, finding each position at the current depth, and extending it by one move.

```
⟨ Method bodies 9 ⟩ ≡
  void phase1prune::gen_table( )
  {
    memset(mem, −1, memsize);
    mem[0] = 0;
    int seen = 1;
    cout ≪ "Gen␣phase1" ≪ flush;
    for (int d = 1; ; d++) {
      int lastiter = (seen ≡ CORNERRSYMM ∗ EDGEOSYMM ∗ EDGEPERM);
      int seek = d − 1;
      int at = 0;
      for (int cs = 0; cs < CORNERRSYMM; cs++) {
        int csymm = kocsymm::cornersymm_expand[cs];
        for (int eosymm = 0; eosymm < EDGEOSYMM; eosymm++)
          for (int epsymm = 0; epsymm < EDGEPERM; epsymm++, at += BYTES_PER_ENTRY)
            if (mem[at] ≡ seek) {⟨ Handle one position 10 ⟩}
      }
      cout ≪ "␣" ≪ d ≪ flush;
      if (lastiter) break;
    }
    cout ≪ "␣done." ≪ endl ≪ flush;
  }
```

See also sections 12, 13, 14, 17, and 19.

This code is used in section 2.

10. For each position, we consider all possible moves, and track the distances of those moves. Then, we combine these results into three bytes. We keep track of which moves bring us closer, so later perhaps we can optimize the case where there's only one such possibility.

```
⟨ Handle one position 10 ⟩ ≡
  int deltadist[NMOVES];
  for (int mv = 0; mv < NMOVES; mv++) {
    int rd = 0;
    kocsymm kc(csymm, eosymm, epsymm);
    kc.move(mv);
    corner_mapinfo & cm = kocsymm::cornersymm[kc.csymm];
    for (int m = cm.minmap; cm.minbits ≫ m; m++)
      if ((cm.minbits ≫ m) & 1) {
        int deosymm = kocsymm::edgeomap[kocsymm::edgepxor[kc.epsymm][m ≫ 3] ⊕ kc.eosymm][m];
        int depsymm = kocsymm::edgepmap[kc.epsymm][m];
        int dat = ((cm.csymm * EDGEOSYMM + deosymm) * EDGEPERM + depsymm) * BYTES_PER_ENTRY;
        rd = mem[dat];
        if (rd ≡ 255) {
          rd = d;
          mem[dat] = rd;
          seen++;
        }
      }
    deltadist[mv] = rd − seek;
  }
  ⟨ Encode deltadist 11 ⟩
```

This code is used in section 9.

11. We now have the delta distances for all `NMOVES` moves. We need to encode this data into the remaining bits of the lookup entry. If the most significant bit of a nybble is set, that means the distances are all nonnegative; if the most significant bit is clear, then the distances are all nonpositive. The least significant three bits indicate for each move whether the higher or lower value is appropriate; a zero means the lower value, and a one means the higher value. If none of the three moves have an impact, we choose the value 8 (and not the value 7, which encodes the same semantics). We collect the information from opposite faces into the same byte value to make later remapping more efficient.

```
⟨ Encode deltadist 11 ⟩ ≡
  for (int b = 0; b < 3; b++) {
    int v = 0;
    int clim = 1;
    for (int c = clim; c ≥ 0; c−−) {
      int vv = 0;
      int cnts[3];
      cnts[0] = cnts[1] = cnts[2] = 0;
      for (int t = 2; t ≥ 0; t−−) {
        vv = 2 ∗ vv + deltadist[3 ∗ b + 9 ∗ c + t];
        cnts[1 + deltadist[3 ∗ b + 9 ∗ c + t]]++;
      }
      if (cnts[0] > 0 ∧ cnts[2] > 0)
        error ("! bad delta distance values within one face turn set") ;
      if (cnts[0])      /∗ combination of zeros and -1's ∗/
        vv += 7;
      else      /∗ combination of zeros and ones, or just zeros ∗/
        vv += 8;
      v = 16 ∗ v + vv;
    }
    mem[at + b + 1] = v;
  }
```

This code is used in section 10.

12. Input and Output. 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 9 ⟩ +≡
  const int CHUNKSIZE = 65536;
  int phase1prune::read_table( )
  {
    FILE *f = fopen(filename, "rb");
    if (f ≡ 0) return 0;
    int togo = memsize;
    unsigned char *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((unsigned int *) p, siz, seed);
      togo −= siz;
      p += siz;
    }
    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;
  }
```

13. 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 9 ⟩ +≡
  void phase1prune::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);
  }
```

14. 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 9 ⟩ +≡
  void phase1prune::check_integrity( )
  {
    if (file_checksum ≠ datahash((unsigned int *) mem, memsize, 0))
      error ("! integrity of pruning table compromised") ;
    cout ≪ "Verified integrity of phase one pruning data: " ≪ file_checksum ≪ endl;
  }
```

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

```
⟨ Initialize the instance 7 ⟩ +≡
  if (read_table( ) ≡ 0) {
    gen_table( );
    file_checksum = datahash((unsigned int *) mem, memsize, 0);
    if (¬suppress_writing) write_table( );
  }
```

16. Lookup routines, and a solve routine.

```
⟨ Method declarations 8 ⟩ +≡
  static int lookup(const kocsymm &kc);
  static int lookup(const kocsymm &kc, int togo, int &nextmovemask);
  static moveseq solve(kocsymm kc);
```

17. Looking up the distance is quick, based on the code we've already seen.

```
⟨ Method bodies 9 ⟩ +≡
  int phase1prune::lookup(const kocsymm &kc)
  {
    corner_mapinfo & cm = kocsymm::cornersymm[kc.csymm];
    int m = cm.minmap;
    int r = mem[BYTES_PER_ENTRY * (((cm.csymm * EDGEOSYMM) +
        kocsymm::edgeomap[kocsymm::edgepxor[kc.epsymm][m ≫
        3] ⊕ kc.eosymm][m]) * 495 + kocsymm::edgepmap[kc.epsymm][m])];
    return r;
  }
```

18. A more important lookup routine, though, is the one that not only gives us the distance, but also tells us which moves take us closer to solved. At the first level, we have that information directly in the table. Unfortunately, because we remap our position to perform the lookup, we also have to remap the bitmap from the table. To do this, we need to take into account the remapping (and how it reorders the faces, and possibly negates the twists) and whether the number of moves left is equal to or in excess of the number of moves to go. We use a single array *map_phase1* that is indexed by all of this data and returns the relevant bits indicating what moves are valid. For each of the sixteen possible reorientations, we also have an additional array that gives the appropriate offsets. We only use this here so we make it file static rather than class static.

```
⟨ Data instantiations 4 ⟩ +≡
  static unsigned char map_phase1_offsets[KOCSYMM][3];
  static int map_phase1[2][12][256];
```

19. Assuming those arrays are filled in appropriately, our main lookup routine looks like this.

```
⟨ Method bodies 9 ⟩ +≡
  int phase1prune::lookup(const kocsymm &kc, int togo, int &nextmovemask)
  {
    corner_mapinfo & cm = kocsymm::cornersymm[kc.csymm];
    int m = cm.minmap;
    int off = BYTES_PER_ENTRY * (((cm.csymm * EDGEOSYMM) +
        kocsymm::edgeomap[kocsymm::edgepxor[kc.epsymm][m >>
        3] ⊕ kc.eosymm][m]) * 495 + kocsymm::edgepmap[kc.epsymm][m]);
    int r = mem[off];
    if (togo < r) {
      nextmovemask = 0;
    }
    else if (togo > r + 1) {
      nextmovemask = ALLMOVEMASK;
    }
    else {
      int(*p)[256] = map_phase1[togo − r];
      unsigned char *o = map_phase1_offsets[m];
      nextmovemask = p[o[0]][mem[off + 1]] + p[o[1]][mem[off + 2]] + p[o[2]][mem[off + 3]];
    }
    return r;
  }
```

20. Initializing the two arrays is a little tricky. First we initialize the offsets; we use a key that uses the least significant bit to indicate a negative orientation, the next bit to indicate that the faces are flipped, and the next two bits to indicate what axis the remapping maps this axis to.

```
⟨ Initialize the instance 7 ⟩ +≡
  for (int m = 0; m < KOCSYMM; m++) {
    for (int f = 0; f < 3; f++) {
      int mv = f * TWISTS;
      int mv2 = cubepos::move_map[m][mv];
      int f2 = mv2/TWISTS;
      int key = 0;
      if (mv2 % TWISTS ≡ TWISTS − 1)      /* negative orientation */
        key++;
      if (f2 ≥ 3)      /* flip the faces */
        key += 2;
      key += 4 * (f2 % 3);      /* where the low order face maps */
      map_phase1_offsets[cubepos::invm[m]][f] = key;
    }
  }
```

21. Now we fill in the actual bit array itself. We essentially just perform the operations described in the key. We calculate the low nybble first; then we iterate to combine it with the high nybble.

```
⟨ Initialize the instance 7 ⟩ +≡
  for (int slack = 0; slack < 2; slack++) {
    for (int key = 0; key < 12; key++) {
      int nv[16];
      for (int nyb = 0; nyb < 16; nyb++) {
        int bits = 0;
        if (slack ∧ nyb ≤ 7) {      /* always, any move */
          bits = 7;
        }
        else if (slack ≡ 0 ∧ nyb ≥ 7) {
          bits = 0;
        }
        else {
          bits = 7 − (nyb & 7);
        }
        if (key & 1)      /* negate the twist if key says so */
          bits = ((bits & 1) ≪ 2) + (bits & 2) + (bits ≫ 2);
        if (key & 2)      /* flip the faces if key says so */
          bits ≪= 3 * TWISTS;
        bits ≪= TWISTS * (key ≫ 2);      /* shift to correct face */
        nv[nyb] = bits;
      }
      int *a = map_phase1[slack][key];
      for (int byte = 0; byte < 256; byte++)
        a[byte] = nv[byte&15] | (((nv[byte ≫ 4] ≪ (3*TWISTS)) | (nv[byte ≫ 4] ≫ (3*TWISTS)))&°777777);
    }
  }
```

22. We also provide a quick and dirty solve routine.

```
⟨ phase1prune.cpp 2 ⟩ +≡
  moveseq phase1prune::solve(kocsymm kc)
  {
    moveseq r;
    int d = phase1prune::lookup(kc);
    while (d > 0) {
      int nmm = 0;
      int t = phase1prune::lookup(kc, d, nmm);
      if (t ≡ 0) break;
      if (t ≠ d)
        error ("! did not make progress") ;
      if (nmm ≡ 0)
        error ("! no solution?") ;
      int mv = ffs(nmm) − 1;
      r.push_back(mv);
      kc.move(mv);
      d−−;
    }
    return r;
  }
```

23. Our test routines.

```
⟨ phase1prune_test.cpp 23 ⟩ ≡
#include "phase1prune.h"
#include <iostream>
  using namespace std;
  int main(int argc, char ∗argv[])
  {
    if (lrand48() ≡ 0) srand48(time(0));
    phase1prune::init();
    int t[10000];
    for (int i = 0; i < 10000; i++) t[i] = random_move();
    kocsymm kc;
    ⟨ Check next move info 24 ⟩
    ⟨ Check basic lookup performance 25 ⟩
    ⟨ Check extended lookup performance 26 ⟩
    ⟨ Check solution performance 27 ⟩
  }
```

24. Check that the next move information is correct.

```
⟨ Check next move info 24 ⟩ ≡
  for (int i = 0; i < 10000; i++) {
    kc.move(t[i]);
    int d = phase1prune::lookup(kc);
    for (int d2 = d − 1; d2 ≤ d + 2; d2++) {
      int nextmovemask = 0;
      kocsymm kc2;
      int dt = phase1prune::lookup(kc, d2, nextmovemask);
      if (dt ≠ d)
        error ("mismatch on lookup") ;
      int bc = 0;
      for (int mv = 0; mv < NMOVES; mv++) {
        kc2 = kc;
        kc2.move(mv);
        dt = phase1prune::lookup(kc2);
        if (dt < d2) bc |= (1 ≪ mv);
      }
      if (nextmovemask ≠ bc)
        error ("! move mask error") ;
    }
  }
```

This code is used in section 23.

25. Check how fast we can look up.

```
⟨ Check basic lookup performance 25 ⟩ ≡
  int sum = 0;
  duration( );
  for (int i = 0; i < 10000; i++)
    for (int j = 0; j < 10000; j++) {
      kc.move(t[j]);
      sum += phase1prune::lookup(kc);
    }
  cout ≪ "Did 100M basic lookups in " ≪ duration( ) ≪ " sum " ≪ sum ≪ endl;
```

This code is used in section 23.

26. Check how fast we can look up with extended info.

```
⟨ Check extended lookup performance 26 ⟩ ≡
  int prev = 10;
  int nextmovemask = 0;
  for (int i = 0; i < 10000; i++)
    for (int j = 0; j < 10000; j++) {
      kc.move(t[j]);
      int r = phase1prune::lookup(kc, prev, nextmovemask);
      sum += r + nextmovemask;
      prev = r;
    }
  cout ≪ "Did 100M extended lookups in " ≪ duration( ) ≪ " sum " ≪ sum ≪ endl;
```

This code is used in section 23.

27. Check how fast we can solve.

```
⟨ Check solution performance 27 ⟩ ≡
  for (int i = 0; i < 1000; i++)
    for (int j = 0; j < 10000; j++) {
      kc.move(t[j]);
      phase1prune::solve(kc);
    }
  cout ≪ "Did␣10M␣solves␣in␣" ≪ duration() ≪ "␣sum␣" ≪ sum ≪ endl;
```

This code is used in section 23.

⟨ Check basic lookup performance 25 ⟩ Used in section 23.
⟨ Check extended lookup performance 26 ⟩ Used in section 23.
⟨ Check next move info 24 ⟩ Used in section 23.
⟨ Check solution performance 27 ⟩ Used in section 23.
⟨ Constant declarations 6 ⟩ Used in section 1.
⟨ Data declarations 3 ⟩ Used in section 1.
⟨ Data instantiations 4, 18 ⟩ Used in section 2.
⟨ Encode *deltadist* 11 ⟩ Used in section 10.
⟨ Handle one position 10 ⟩ Used in section 9.
⟨ Initialize the instance 7, 15, 20, 21 ⟩ Used in section 2.
⟨ Method bodies 9, 12, 13, 14, 17, 19 ⟩ Used in section 2.
⟨ Method declarations 8, 16 ⟩ Used in section 1.
⟨ Utility functions 5 ⟩ Used in section 2.
⟨ `phase1prune.cpp` 2, 22 ⟩
⟨ `phase1prune.h` 1 ⟩
⟨ `phase1prune_test.cpp` 23 ⟩

PHASE1PRUNE