November 24, 2020 at 13:23

1. Intro. Counting closure operators on six elements that are nonisomorphic under permutations. (My program for $n = 5$ used a too-slow method; here I speed up by a factor of $n!$, I hope.)

I wrote this in a terrific hurry—sorry. The strategy is outlined in the next section below.

```
#define n 5
#define nn (1 ≪ n)
#define nfactorial 120
#define final_level (nn − 1)    /* the first element that is never in a solution */
#define verbose (n < 5)
#define log LOG    /* get around bug in clang */
#define logl LOGL    /* ditto */
#include <stdio.h>
  ⟨ Preprocessor definitions ⟩
  char f[nn];
  unsigned char perm[nfactorial][nn], iperm[nfactorial][nn];    /* perms and inverses */
  int link[nfactorial];    /* links in the lists of permutations */
  int wait[nn];    /* heads of those lists */
  int disc[nn];    /* permutations discarded at each level */
  int log0[nn], logl[nn];    /* where we began shuffling perms at each level */
  int log[nfactorial ∗ nn ∗ 2];
  int logptr;    /* current position in log table */
  int forced[nn];    /* is this entry forced to be zero? */
  int forcings[nfactorial];    /* how many cases has this perm forced? */
  unsigned int sols, tsols;
  ⟨ Subroutines 16 ⟩
  main()
  {
    register int d, j, k, l, m, p, q, t, auts;
    ⟨ Make the permutation tables 3 ⟩;
    ⟨ Put all permutations into wait[0] 4 ⟩;
    ⟨ Find all the solutions 2 ⟩;
    printf("Altogether␣%d␣solutions␣(reduced␣from␣%d).\n", sols + 1, tsols + 1);
  }
```

2. ⟨ Find all the solutions 2 ⟩ ≡

```
  l = logptr = 0;
  auts = nfactorial;
newlevel: if (l ≡ final_level) goto backtrack;
  logl[l] = logptr;
  if (verbose) {
    printf("Entering␣level␣%x␣(%d␣auts␣so␣far)\n", l, auts);
  }
  if (forced[l]) {
    if (verbose) printf("␣forced␣rejection␣of␣%x\n", l);
    goto reject;
  }
  ⟨ Reject l if it violates closure 5 ⟩;
  ⟨ Go through wait[l], trying to move it to wait[0]; but reject l if there's a conflict 9 ⟩;
  f[l] = 1;
  if (verbose) printf("␣accepting␣%x\n", l);
  ⟨ Update wait[0] and count the automorphisms 10 ⟩⟨ Record a solution 6 ⟩;
  l++;
  goto newlevel;
undo: ⟨ Downdate wait[0] 13 ⟩;
  ⟨ Reconstruct wait[l] 11 ⟩;
reject: f[l] = 0;
  ⟨ Check for new forced moves 14 ⟩;
  l++;
  goto newlevel;
backtrack: while (l > 0) {
    l--;
    if (f[l] ≡ 1) {
      if (verbose) printf("␣now␣rejecting␣%x\n", l);
      goto undo;
    }
    else ⟨ Uncheck for new forced moves 15 ⟩;
  }
  for (p = 1; p < nfactorial; p++)
    if (forcings[p]) printf("error:␣forcings[%d]␣not␣restored␣to␣zero!\n", p);
  for (k = 1; k < nn; k++)
    if (forced[k]) printf("error:␣forced[%x]␣not␣restored␣to␣zero!\n", k);
```

This code is used in section 1.

3. Algorithm 7.2.1.2T.

⟨ Make the permutation tables 3 ⟩ ≡

```
d = nfactorial >> 1, perm[d][0] = 1;
for (m = 2; m < n; ) {
  m++, d = d/m;
  for (k = 0; k < nfactorial; ) {
    for (k += d, j = m - 1; j > 0; k += d, j--) perm[k][0] = j;
    perm[k][0]++, k += d;
    for (j++; j < m; k += d, j++) perm[k][0] = j;
  }
}
for (j = 0; j < nn; j++) perm[0][j] = j;
for (k = 1; k < nfactorial; k++) {
  m = 1 << (perm[k][0] - 1);
  for (j = 0; j < nn; j++) {
    d = perm[k - 1][j];
    d ^= d >> 1;
    d &= m;
    d |= d << 1;
    perm[k][j] = perm[k - 1][j] ^ d;
  }
}
for (p = 0; p < nfactorial; p++)
  for (k = 0; k < nn; k++) iperm[p][perm[p][k]] = k;
```

This code is used in section 1.

4. ⟨ Put all permutations into *wait*[0] 4 ⟩ ≡

```
for (p = 1; p < nfactorial; p++) link[p] = wait[0], wait[0] = p;
```

This code is used in section 1.

5. ⟨ Reject *l* if it violates closure 5 ⟩ ≡

```
for (j = 0; j < l; j++)
  if (f[j] && !(f[j & l])) {
    if (verbose) printf("␣rejecting␣%x␣for␣closure\n", l);
    goto reject;
  }
```

This code is used in section 2.

6. ⟨ Record a solution 6 ⟩ ≡

```
{
  sols++;
  tsols += nfactorial/auts;
  if (n < 6) {
    printf("%d:", sols);
    for (j = 0; j < nn; j++)
      if (f[j]) printf("␣%x", j);
    printf("␣(%d␣aut%s)\n", auts, auts > 1 ? "s" : "");
  }
}
```

This code is used in section 2.

7. The interesting part. When writing this program, I didn't have to work nearly as hard as I did in GROPESX (a program for algebraic structures that I wrote a few months ago). But still there are a few nontrivial points of interest as the permutations get shuffled from list to list.

In fact, I tried to get away with a more substantial simplification. It failed miserably.

In actual fact, I was tearing my hair out for awhile, because I couldn't believe that this would be so complicated. Maybe some day I'll learn the right way to tackle this problem.

8. The basic idea is simple: Each closure operation corresponds to a sequence $(f[0], \ldots, f[nn-2])$ with the property that $f[j] = f[k] = 1$ implies $f[j \mathbin{\&} k] = 1$. This program produces only canonical solutions, namely solutions with the property that $(f[0], \ldots, f[nn-1])$ is lexicographically greater than or equal to $(f[p_0], \ldots, f[p_{nn-1}])$ for all perms p. (These perms are permutations of the bits; for example, if $p_1 = 2$ and $p_2 = 4$ then $p_3 = 6$.)

At level l, I've set the values of $(f[0], \ldots, f[l-1])$. All perms live in various lists: If $(f[0], \ldots, f[l-1])$ is known to be lexicographically greater than $(f[p_0], \ldots, f[p_{l-1}])$, the perm p is in a discard list; otherwise p is in a waiting list. List $wait[0]$ has all the current automorphisms: These perms permute the current 1s $\{j \mid 0 \le j < l \text{ and } f[j] = 1\}$. Furthermore, for all subscripts $k < l$ such that $f[k] = 0$ and $p_k > l$, the future values p_k are marked so as to force $f[p_k] = 0$. Finally, the waiting lists $wait[k]$ for $l \le k < nn$ contain elements $j < l$ such that $f[j] = 1$ and $p_j = k$ and $f[i] = f[p_i]$ for $0 \le i < j$. When level k comes along, such perms will effectively be discarded if $f[k]$ is set to 0; but if $f[k]$ is set to 1, they will move to other lists.

The forcings were what caused me grief. I didn't want to have an elaborate data structure that showed exactly who was forcing whom, because that was very difficult to maintain under backtracking. The solution I found, shown below, is not terrifically easy, but it certainly is better than anything else I could think of. Basically $forcings[p]$ counts the number of places where p has forced a future value k; and $forced[k]$ counts the number of perms that have forced that value. These counts can, fortunately, be maintained by doing local operations, as we see for how many levels each perm remains relevant.

9. Okay, now let me write the most critical part of the program. At this point in the computation we are planning to set $f[l] = 1$. But we may have to abandon that plan, if "immediate rejection" would result. (Immediate rejection occurs when setting $f[l] = 1$ unhides a lexicographically superior solution.)

The *log* table records what we do here, so that it can be undone later. Entries on the log are of two kinds: A negative entry stands for a permutation that was "discarded" because it is no longer active. A nonnegative entry k stands for a permutation that moved to $wait[k]$. In either case, entry $log[t]$ identifies the destination of a permutation that came from $wait[0]$ if $t \ge log0[l]$, otherwise from $wait[l]$.

⟨ Go through $wait[l]$, trying to move it to $wait[0]$; but reject l if there's a conflict 9 ⟩ ≡

```
for (p = wait[l], wait[l] = 0; p; p = q) {
  q = link[p];
  for (k = iperm[p][l] + 1; k < l; k++) {
    if (f[k] ≡ 0 ∧ iperm[p][k] < iperm[p][l]) forcings[p]−−;
    j = perm[p][k];
    if (j < l) {
      if (f[j] ≡ f[k]) continue;
      if (f[k] ≡ 0) ⟨ Reject l immediately 12 ⟩;
      log[logptr++] = −j, link[p] = disc[l], disc[l] = p;      /* discard p */
      goto nextp;
    } else if (f[k] ≡ 1) {
      log[logptr++] = j, link[p] = wait[j], wait[j] = p;
      for (j = k − 1; j > iperm[p][l]; j−−)
        if (f[j] ≡ 0 ∧ perm[p][j] > k ∧ perm[p][j] < l) forcings[p]++;
      goto nextp;
    } else {
      if (verbose) printf(" f[%x]=1 will force f[%x]=0\n", j);
      forcings[p]++, forced[j]++;
    }
  }
  log[logptr++] = 0, link[p] = wait[0], wait[0] = p;
nextp: continue;
}
```

This code is used in section 2.

10. After we've made it through $wait[l]$, we are able to set $f[l] = 1$. The items of $wait[0]$ might now be automorphisms, or they might need to be moved to other waiting lists.

⟨ Update $wait[0]$ and count the automorphisms 10 ⟩ ≡

```
log0[l] = logptr;
for (auts = 1, p = wait[0], wait[0] = 0; p; p = q) {
  q = link[p];
  j = perm[p][l];
  if (j ≡ l) goto retain_it;
  else if (j > l) log[logptr++] = j, link[p] = wait[j], wait[j] = p;
  else if (f[j] ≡ 0) log[logptr++] = −1, link[p] = disc[l], disc[l] = p;
  else goto retain_it;
  continue;
retain_it: log[logptr++] = 0, link[p] = wait[0], wait[0] = p;
  auts++;
}
```

This code is used in section 2.

11. Here I've made a point to "undo" in precisely the reverse order of what I "did," so that lists are perfectly restored to their former condition.

The label *kludge* is one of my trademarks, I guess: It's a place in the middle of nested loops, which just happens to be the place we want to jump when doing an immediate rejection.

⟨ Reconstruct *wait*[*l*] 11 ⟩ ≡

```
t = 0;
while (logptr > logl[l]) {
  j = log[−−logptr];
  if (j < 0) {
    p = disc[l], disc[l] = link[p], k = iperm[p][−j];
    if (f[k] ≡ 0 ∧ iperm[p][k] < iperm[p][l]) forcings[p]++;
  } else if (j > 0) {
    p = wait[j], wait[j] = link[p], k = iperm[p][j];
    for (j = k − 1; j > iperm[p][l]; j−−)
      if (f[j] ≡ 0 ∧ perm[p][j] > k ∧ perm[p][j] < l) forcings[p]−−;
  } else p = wait[0], wait[0] = link[p], k = l;
  link[p] = t, t = p, k−−;
  while (k > iperm[p][l]) {
    j = perm[p][k];
    if (j > l ∧ f[k] ≡ 0) forcings[p]−−, forced[j]−−;
  kludge: if (f[k] ≡ 0 ∧ iperm[p][k] < iperm[p][l]) forcings[p]++;
    k−−;
  }
}
wait[l] = t;     /* I think it's "all together now" */
```

This code is used in section 2.

12. ⟨ Reject *l* immediately 12 ⟩ ≡

```
{
  t = p;
  goto kludge;
}
```

This code is used in section 9.

13. ⟨ Downdate *wait*[0] 13 ⟩ ≡

```
t = 0;
while (logptr > log0[l]) {
  j = log[−−logptr];
  if (j < 0) p = disc[l], disc[l] = link[p];
  else p = wait[j], wait[j] = link[p];
  link[p] = t, t = p;
}
wait[0] = t;
```

This code is used in section 2.

14. ⟨ Check for new forced moves 14 ⟩ ≡

```
for (auts = 1, p = wait[0]; p; p = link[p]) {
  j = perm[p][l];
  if (j > l) {
    if (verbose) printf("␣forcing␣f[%x]=0\n", j);
    forcings[p]++, forced[j]++;
  }
  if (iperm[p][l] < l) forcings[p]--;
  if (verbose) auts++;
}
```

This code is used in section 2.

15. ⟨ Uncheck for new forced moves 15 ⟩ ≡

```
for (p = wait[0]; p; p = link[p]) {
  j = perm[p][l];
  if (j > l) {
    forcings[p]--, forced[j]--;
  }
  if (iperm[p][l] < l) forcings[p]++;
}
```

This code is used in section 2.

16. Finally, here's a routine that documents the main invariant relations that I expect to be true when this program enters level l. (The *sanity* routine sure did prove to be useful when I was debugging the twisted logic above.)

⟨ Subroutines 16 ⟩ ≡

```
int timestamp;
int stamp[nfactorial];

void sanity(int l)
{
  register c, j, jj, k, p;
  if (l ≡ 0) return;
  timestamp++;
  ⟨ Sanity check the wait lists 17 ⟩;
  ⟨ Sanity check the discard lists 18 ⟩;
  ⟨ Sanity check wait[0] 19 ⟩;
  for (p = 1; p < nfactorial; p++)
    if (stamp[p] ≠ timestamp) {
      printf("error:␣perm␣%d␣has␣disappeared!\n", p);
      goto error_exit;
    }
  return;
error_exit: printf("(Detected␣at␣level␣%x)\n", l); return;
}
```

This code is used in section 1.

17. ⟨ Sanity check the wait lists 17 ⟩ ≡

```
for (k = l; k < nn; k++)
  for (p = wait[k]; p; p = link[p]) {
    stamp[p] = timestamp;
    jj = iperm[p][k];
    if (f[jj] ≠ 1) {
      printf("error:␣wait[%x]␣contains␣noncritical␣perm␣%d!\n", k, p);
      goto error_exit;
    }
    for (j = c = 0; ; j++) {
      if (perm[p][j] > jj) {
        if (f[j] ≡ 0) c++;
        else if (perm[p][j] ≡ k) break;
      } else if (f[j] ≠ f[perm[p][j]]) {
        printf("error:␣perm␣%d␣on␣wait[%x]␣contains␣early␣mismatch␣f[%x]!=f[%x]!\n", p, k, j,
            perm[p][j]);
        goto error_exit;
      }
    }
    if (c ≠ forcings[p]) {
      printf("error:␣forcings[%d]␣is␣%d,␣not␣%d,␣in␣wait[%x]!\n", p, forcings[p], c, k);
      goto error_exit;
    }
  }
```

This code is used in section 16.

18. The wait lists *wait*[k] for $1 \le k < l$ are essentially discards too, because we've set $f[k] = 0$.
I don't check the *forcings* count in *disc*[k], because the perms in such lists don't satisfy the same invariants as other perms.

⟨ Sanity check the discard lists 18 ⟩ ≡

```
for (k = 1; k < l; k++) {
  for (p = disc[k]; p; p = link[p]) {
    stamp[p] = timestamp;
    for (jj = 0; jj < l; jj++)
      if (f[jj] ≠ f[perm[p][jj]]) break;
    if (jj ≡ l) {
      printf("error:␣disc[%x]␣contains␣the␣nondiscardable␣perm␣%d!\n", k, p);
      goto error_exit;
    }
    if (f[jj] ≡ 0) {
      printf("error:␣disc[%x]␣contains␣the␣counterexample␣perm␣%d!\n", k, p);
      goto error_exit;
    }
  }
  for (p = wait[k]; p; p = link[p]) {
    stamp[p] = timestamp;
    for (jj = 0; jj < l; jj++)
      if (f[jj] ≠ f[perm[p][jj]]) break;
    if (jj ≡ l) {
      printf("error:␣wait[%x]␣contains␣the␣nondiscardable␣perm␣%d!\n", k, p);
      goto error_exit;
    }
    if (f[jj] ≡ 0) {
      printf("error:␣wait[%x]␣contains␣the␣counterexample␣perm␣%d!\n", k, p);
      goto error_exit;
    }
    for (j = c = 0; j < jj; j++)
      if (perm[p][j] > jj ∧ f[j] ≡ 0) c++;
    if (c ≠ forcings[p]) {
      printf("error:␣forcings[%d]␣is␣%d,␣not␣%d,␣in␣wait[%x]!\n", p, forcings[p], c, k);
      goto error_exit;
    }
  }
}
```

This code is used in section 16.

19. ⟨ Sanity check *wait*[0] 19 ⟩ ≡

```
for (p = wait[0]; p; p = link[p]) {
  stamp[p] = timestamp;
  for (c = j = 0; j < l; j++) {
    if (f[j] ≠ f[perm[p][j]]) {
      if (f[j] ≡ 0) {
        printf("error:␣wait[0]␣contains␣the␣counterexample␣perm␣%d!\n", k, p);
        goto error_exit;
      }
      printf("error:␣wait[0]␣contains␣the␣discardable␣perm␣%d!\n", k, p);
    }
    if (perm[p][j] ≥ l) c++;
  }
  if (c ≠ forcings[p]) {
    printf("error:␣forcings[%d]␣is␣%d,␣not␣%d,␣in␣wait[0]!\n", p, forcings[p], c);
    goto error_exit;
  }
}
```

This code is used in section 16.

20. Index.

HORN-COUNT