

May 16, 2016 at 15:34

1* This is a quick program to find all canonical forms of reflection networks for small n .

Well, when I wrote that paragraph I believed it, but subsequently I have added lots of bells and whistles because I wanted to compute more stuff. At present this code determines the number B_n of equivalence classes of reflection networks (i.e., irredundant primitive sorting networks); also the number of weak equivalence classes, either with (C_{n+1}) or without (D_{n+1}) anti-isomorphism; and the number of preweak equivalence classes (E_{n+1}) , which is the number of simple arrangements of $n + 1$ pseudolines in a projective plane. For each representative of D_{n+1} it also computes the “score,” which is the number of ways to add another pseudoline crossing the network.

If compiled without the `NOPRINT` switch, each member of B_n is printed as a string of transposition numbers, generated in lexicographic order. This is followed by `*` if the string is also a representative of C_{n+1} when prefixed by $01 \dots n$. And if the string is also a representative of D_{n+1} , you also get the score in brackets, followed by `#` if it is a representative of E_{n+1} . If not a representative of D_{n+1} , the symbol `>` is printed followed by the string of an anti-equivalent network.

If compiled with the `DEBUG` switch, you also get intermediate output about the backtrack tree and the networks generated while searching for anti-equivalence and preweak equivalence.

I wrote this program to allow n up to 10; but integer overflow will surely occur in $B_{10} \approx 2 \times 10^{10}$, if I ever get a computer fast enough to run that case. When $n = 7$, this program took 48 seconds to run, on January 12, 1991; the running time for $n = 6$ was 1 second, and for $n = 8$ it was 57 minutes. Therefore I made a stripped-down version to enumerate only B_n when $n = 9$. In fact, this program is that stripped down version, contrary to what is said above. This program does $n = 7$ in 4 seconds, $n = 8$ in 4:42 minutes, and I think it will do $n = 9$ in about 10 hours. I tried several experiments for benchmarking, since this program is clearly compute-bound: Compiling with `-g` instead of with `-O` increased the running time for $n = 8$ to 6:19; if I also removed the `register` hints on variables i, ii, iii, j , it went up to 9:09. With optimization and no register hints it took 6:38. (When I actually computed $B_9 = 112018190$, I used the slowest version, with no register hints and the `-g` switch; that took 19:50:37.)

```
#include <stdio.h>
```

2* There’s an array $a[1 \dots n]$ containing k inversions; an index j showing where we are going to try to reduce the inversions by swapping $a[j]$ with $a[j + 1]$; and two arrays for backtracking. At choice-level l we set $t[l]$ to the current j value, and we also set $c[l]$ to 1 if we swapped, 0 if we didn’t.

```
#define swap(j)
    { int tmp = a[j]; a[j] = a[j + 1]; a[j + 1] = tmp; }
#define npairs 120 /* should be greater than  $2\binom{n+1}{2}$  */
#define ncycle 240 /* should be greater than  $4\binom{n+1}{2}$  */
< Global variables 2* > ≡
    int n; /* number of elements to be reflected */
    int a[10]; /* array that shows progress */
    int k; /* number of inversions yet to be removed */
    int l; /* current choice level */
    int c[npairs]; /* code for choices made */
    int t[npairs]; /* j values where choices were made */
    int bn, cn, dn, en; /* counters for  $B_n, C_{n+1}, D_{n+1}, E_{n+1}$  */
    int smin, smax; /* counters for “scores” */
    float stot; /* grand total of scores */
```

See also sections 8 and 13.

This code is used in section 3*.

3* The value of n is supposed to be an argument.

```
#define abort(s)
    { fprintf(stderr,s); exit(1); }
⟨Global variables 2*⟩
main(argc,argv)
    int argc; /* number of args */
    char **argv; /* the args */
{ register int j; /* current place in array */
  register int i, ii, iii; /* general-purpose indices */
  if (argc ≠ 2) abort("Usage: _reflect_n\n");
  if (sscanf(argv[1], "%d", &n) ≠ 1 ∨ n < 2 ∨ n > 10) abort("n _should _be _in _the _range _2..10!\n");
  ⟨Initialize 4⟩;
  ⟨Run through all canonical reflection networks 5*⟩;
  printf("B=%d\n", bn);
}
```

5* ⟨Run through all canonical reflection networks 5*⟩ ≡

moveleft: $j--$;

loop:

```
if (j ≡ 0) {
  if (k ≡ 0)
    if ((++bn % 1000000) ≡ 0) {
      for (i = 1; i < l; i++)
        if (c[i]) putchar('0' - 1 + t[i]);
        putchar('\n');
    }
}
```

⟨Backtrack, either going to *loop* or to *finished* when all possibilities are exhausted 6⟩;

```
}
if (a[j] < a[j + 1]) goto moveleft;
t[l] = j;
c[l++] = 0;
goto moveleft;
```

finished: ;

This code is used in section 3*.

25* ⟨If debugging, print the active region of x 25*⟩ ≡

```
#ifndef DEBUG
    printf("\n_ _");
    for (m = s; m < ss; m++) putchar(x[m] + '0' - 1);
#endif
```

This code is used in sections 16, 17, and 20.

The following sections were changed by the change file: 1, 2, 3, 5, 25.

<i>a</i> : 2*	<i>cn</i> : 2*, 4, 11.
<i>abort</i> : 3*	<i>d</i> : 8.
<i>acc</i> : 23.	<i>DEBUG</i> : 1*, 7, 16, 24, 25*
<i>argc</i> : 3*	<i>delta</i> : 23.
<i>argv</i> : 3*	<i>dn</i> : 2*, 4, 22.
<i>b</i> : 8.	<i>done</i> : 12, 15, 18.
<i>bn</i> : 2*, 3*, 4, 5*, 7.	<i>e</i> : 13.
<i>c</i> : 2*	<i>en</i> : 2*, 4, 12.

exit: 3*
finished: 5*, 6.
fprintf: 3*
i: 3*
ii: 1*, 2*, 21, 23.
iii: 1*, 3*, 20, 21.
j: 3*
jj: 13, 20, 21.
k: 2*
l: 2*
loop: 5*, 6.
m: 13.
main: 3*
moveleft: 5*
n: 2*
ncycle: 2*, 8, 13.
NOPRINT: 1*, 7, 11, 12, 15, 22.
npairs: 2*, 8, 13.
okay: 17, 18.
p: 23.
printf: 3*, 22, 24, 25*
putchar: 5*, 7, 11, 12, 15, 16, 24, 25*
r: 8.
ref: 12, 13.
rep: 13, 20.
rr: 8, 9, 10, 11, 13, 14, 20, 22, 23.
rrr: 8, 9, 12, 15, 16, 20, 22, 24.
s: 13.
score: 22.
smax: 2*, 4, 22.
smin: 2*, 4, 22.
ss: 13, 15, 16, 17, 18, 19, 20, 25*
sscanf: 3*
stderr: 3*
stot: 2*, 4, 22.
swap: 2*, 6.
t: 2*
tmp: 2*
x: 13.
y: 13.

- ⟨ Backtrack, either going to *loop* or to *finished* when all possibilities are exhausted 6 ⟩ Used in section 5*.
- ⟨ Check if it gives a new CC system on $n + 1$ elements 9 ⟩ Used in section 7.
- ⟨ Compute the score for this weak equivalence/antiequivalence class rep 22 ⟩ Used in section 12.
- ⟨ End-around shift x 19 ⟩ Used in sections 15, 16, and 17.
- ⟨ Fill in the cell counts $x[i]$ for cases when $b[i] = j$ 23 ⟩ Used in section 22.
- ⟨ Global variables 2*, 8, 13 ⟩ Used in section 3*.
- ⟨ If debugging, print the active region of b 24 ⟩ Used in section 20.
- ⟨ If debugging, print the active region of x 25* ⟩ Used in sections 16, 17, and 20.
- ⟨ If the new network is weakly equivalent to a lexicographically smaller one, **goto done** 17 ⟩ Used in section 12.
- ⟨ If the x network is weakly equivalent to an earlier one, **goto done**; if weakly equivalent to the present one, **goto okay** 18 ⟩ Used in section 17.
- ⟨ Initialize 4 ⟩ Used in section 3*.
- ⟨ Insert the value $j + 1$ canonically into x 21 ⟩ Used in section 20.
- ⟨ Make the big test for pre-weak equivalence 12 ⟩ Used in section 11.
- ⟨ Move the “pole” into the cell preceding the first transposition module 20 ⟩ Used in section 12.
- ⟨ Print a solution 7 ⟩
- ⟨ Replace the present x by the reverse of y 16 ⟩ Used in section 12.
- ⟨ Reset b to a double cycle 14 ⟩ Used in section 12.
- ⟨ Run through all canonical reflection networks 5* ⟩ Used in section 3*.
- ⟨ Shift the first transposition to the other end 10 ⟩ Used in section 9.
- ⟨ Test lexicographic order; **break** if equal or less 11 ⟩ Used in section 9.
- ⟨ Test the reverse of b for weak equivalence; **goto done** if weakly equivalent to a previous case 15 ⟩ Used in section 12.