

November 24, 2020 at 13:23

1. Intro. This is a transcription of my “random matroid” program in #P72.

Standard input contains a sequence of integers. The first of these is the universe size, n , which should be at most 16. Then comes, for $r = 1, 2, \dots$, a list of sets that are stipulated to have rank $\leq r$. Sets are specified in hexadecimal notation, and each list is terminated by 0. Thus, the π -based example in my paper corresponds to the standard input

10 1a 222 64 128 288 10c

because #1a = $2^4 + 2^3 + 2^1$ represents the set $\{1, 3, 4\}$, and #222 represents $\{1, 5, 9\}$, etc. The program appends zeros to the data on standard input if necessary, so trailing zeros can be omitted. Similarly, the standard input

5 7 0 1e

specifies a five-point matroid in which $\{0, 1, 2\}$ has rank ≤ 2 and $\{1, 2, 3, 4\}$ has rank ≤ 3 .

#define nmax 16 /* to go higher, extend *print_set* to larger-than-hex digits */

#define lmax 25742 /* $2(\binom{16}{8} + 1)$, a safe upper bound on list size */

#include <stdio.h>

```
int n; /* number of elements in the universe */
int mask; /*  $2^n - 1$  */
int S[lmax + 1], L[lmax + 1]; /* list memory */
int r; /* the current rank */
int h; /* head of circular list of closed sets for rank  $r$  */
int nh; /* head of circular list being formed for rank  $r + 1$  */
int avail; /* beginning the list of available space */
int unused; /* the first unused slot in  $S$  and  $L$  arrays */
int x; /* a set used to communicate with the insert routine */
int rank[1 << nmax]; /* 100 + cardinality, or assigned rank */
```

<Subroutines 8>

main()

```
{
    register int i, j, k;
    if (scanf("%d", &n) != 1 || n > 16 || n < 0) {
        fprintf(stderr, "Sorry, I can't deal with a universe of size %d.\n", n);
        exit(-1);
    }
    mask = (1 << n) - 1;
    <Set initial contents of rank table 2>;
    <Initialize list memory to available 3>;
    rank[0] = 0, r = 0;
    while (rank[mask] > r) <Pass from rank  $r$  to  $r + 1$  4>;
    print_circuits();
}
```

2. <Set initial contents of *rank* table 2> \equiv

```
k = 1;
rank[0] = 100;
while (k <= mask) {
    for (i = 0; i < k; i++) rank[k + i] = rank[i] + 1;
    k = k + k;
}
```

This code is used in section 1.

3. The published paper had a comparatively inefficient algorithm here; it initialized thousands of links that usually remained unused.

```

⟨ Initialize list memory to available 3 ⟩ ≡
    L[1] = 2;
    L[2] = 1;
    S[2] = 0;
    h = 1; /* list containing the empty set */
    unused = 3;

```

This code is used in section 1.

```

4. ⟨ Pass from rank  $r$  to  $r + 1$  4 ⟩ ≡
{
    ⟨ Create empty list 5 ⟩;
    generate();
    if (r) enlarge();
    ⟨ Return list  $h$  to available storage 6 ⟩;
    r++;
    h = nh;
    sort(); /* optional */
    print_list(h);
    ⟨ Assign rank to sets and print independent ones 7 ⟩;
}

```

This code is used in section 1.

```

5. ⟨ Create empty list 5 ⟩ ≡
    nh = avail;
    if (nh) avail = L[nh];
    else nh = unused++;
    L[nh] = nh;

```

This code is used in section 4.

```

6. ⟨ Return list  $h$  to available storage 6 ⟩ ≡
    for (j = h; L[j] ≠ h; j = L[j]) ;
    L[j] = avail;
    avail = h;

```

This code is used in section 4.

```

7. ⟨ Assign rank to sets and print independent ones 7 ⟩ ≡
    printf("Independent sets for rank %d:", r);
    for (j = L[h]; j ≠ h; j = L[j]) mark(S[j]);
    printf("\n");

```

This code is used in section 4.

8. The *generate* procedure inserts minimal closed sets for rank $r + 1$ into a circular list headed by nh . (It corresponds to “Step 2” in the published algorithm.)

⟨Subroutines 8⟩ \equiv

```

void insert(void);    /* details coming soon */
void generate(void)
{
    register int  $t, v, y, j, k$ ;
    for ( $j = L[h]; j \neq h; j = L[j]$ ) {
         $y = S[j]$ ;    /* a closed set of rank  $r$  */
         $t = \text{mask} - y$ ;
        ⟨Find all sets in list  $nh$  that already contain  $y$  and remove excess elements from  $t$  9⟩;
        ⟨Insert  $y \cup a$  for each  $a \in t$  10⟩;
    }
}

```

See also sections 11, 12, 13, 14, 15, 16, 17, and 18.

This code is used in section 1.

9. ⟨Find all sets in list nh that already contain y and remove excess elements from t 9⟩ \equiv

```

for ( $k = L[nh]; k \neq nh; k = L[k]$ )
    if ( $(S[k] \& y) \equiv y$ )  $t \&= \sim S[k]$ ;

```

This code is used in section 8.

10. ⟨Insert $y \cup a$ for each $a \in t$ 10⟩ \equiv

```

while ( $t$ ) {
     $x = y \mid (t \& -t)$ ;
    insert();    /* insert  $x$  into  $nh$ , possibly enlarging  $x$  */
     $t \&= \sim x$ ;
}

```

This code is used in section 8.

11. The following key procedure basically inserts the set x into list nh . But it augments x if necessary (and deletes existing entries of the list) so that no two entries have an intersection of rank greater than r . Thus it incorporates the idea of “Step 4,” but it is more efficient than a brute force implementation of that step.

```

⟨Subroutines 8⟩ +=
void insert(void)
{
    register int  $j, k$ ;
     $j = nh$ ;
    store:  $S[nh] = x$ ;
    loop:  $k = j$ ;
    continu:  $j = L[k]$ ;
    if ( $rank[S[j] \& x] \leq r$ ) goto loop;
    if ( $j \neq nh$ ) {
        if ( $x \equiv (x \mid S[j])$ ) { /* remove from list and continue */
             $L[k] = L[j], L[j] = avail, avail = j$ ;
            goto continu;
        } else { /* augment  $x$  and go around again */
             $x \mid= S[j], nh = j$ ;
            goto store;
        }
    }
     $j = avail$ ;
    if ( $j$ )  $avail = L[j]$ ;
    else  $j = unused++$ ;
     $L[j] = L[nh]$ ;
     $L[nh] = j$ ;
     $S[j] = x$ ;
}

```

12. The *enlarge* procedure inserts sets that are read from standard input until encountering an empty set. (It corresponds to “Step 3.”)

```

⟨Subroutines 8⟩ +=
void enlarge(void)
{
    while (1) {
         $x = 0$ ;
        scanf("%x", & $x$ );
        if ( $\neg x$ ) return;
        if ( $rank[x] > r$ ) insert();
    }
}

```

13. We don't output a set as a hexadecimal number according to the convention used on standard input; instead, we print an increasing sequence of hexadecimal digits that name the actual set elements. For example, the set that was input as **1a** would be output as **134**.

⟨Subroutines 8⟩ +≡

```
void print_set(int t)
{
    register int j, k;
    printf("_");
    for (j = 1, k = 0; j ≤ t; j <<= 1, k++)
        if (t & j) printf("%x", k);
}
```

14. ⟨Subroutines 8⟩ +≡

```
void print_list(int h)
{
    register int j;
    printf("Closed_sets_for_rank%d:", r);
    for (j = L[h]; j ≠ h; j = L[j]) print_set(S[j]);
    printf("\n");
}
```

15. The subroutine *mark*(*m*) sets $rank[m'] = r$ for all subsets $m' \subseteq m$ whose rank is not already $\leq r$, and outputs m' if it is independent (that is, if its rank equals its cardinality).

⟨Subroutines 8⟩ +≡

```
void mark(int m)
{
    register int t, v;
    if (rank[m] > r) {
        if (rank[m] ≡ 100 + r) print_set(m);
        rank[m] = r;
        for (t = m; t; t = v) {
            v = t & (t - 1);
            mark(m - t + v);
        }
    }
}
```

16. I've added a *tl* array to the data structure, to speed up and shorten this routine.

⟨Subroutines 8⟩ +≡

```

void sort()
{
    register int i, j, k;
    int hd[101 + nmax], tl[101 + nmax];
    for (i = 100; i ≤ 100 + n; i++) hd[i] = -1;
    j = L[h];
    L[h] = h;
    while (j ≠ h) {
        i = rank[S[j]];
        k = L[j];
        L[j] = hd[i];
        if (L[j] < 0) tl[i] = j;
        hd[i] = j;
        j = k;
    }
    for (i = 100; i ≤ 100 + n; i++)
        if (hd[i] ≥ 0) L[tl[i]] = L[h], L[h] = hd[i];
}

```

17. The parameter *card* is 100 plus the cardinality of *m* in the following subroutine.

⟨Subroutines 8⟩ +≡

```

void unmark(int m, int card)
{
    register t, v;
    if (rank[m] < 100) {
        rank[m] = card;
        for (t = mask - m; t; t = v) {
            v = t & (t - 1);
            unmark(m + t - v, card + 1);
        }
    }
}

```

18. ⟨Subroutines 8⟩ +≡

```

void print_circuits(void)
{
    register int i, k;
    printf("The_circuits_are:");
    for (k = 1; k ≤ mask; k += k)
        for (i = 0; i < k; i++)
            if (rank[k + i] ≡ rank[i]) {
                print_set(k + i);
                unmark(k + i, rank[i] + 101);
            }
    printf("\n");
}

```

19. Index.

avail: [1](#), [5](#), [6](#), [11](#).

card: [17](#).

continuu: [11](#).

enlarge: [4](#), [12](#).

exit: [1](#).

fprintf: [1](#).

generate: [4](#), [8](#).

h: [1](#), [14](#).

hd: [16](#).

i: [1](#), [16](#), [18](#).

insert: [1](#), [8](#), [10](#), [11](#), [12](#).

j: [1](#), [8](#), [11](#), [13](#), [14](#), [16](#).

k: [1](#), [8](#), [11](#), [13](#), [16](#), [18](#).

L: [1](#).

lmax: [1](#).

loop: [11](#).

m: [15](#), [17](#).

main: [1](#).

mark: [7](#), [15](#).

mask: [1](#), [2](#), [8](#), [17](#), [18](#).

n: [1](#).

nh: [1](#), [4](#), [5](#), [8](#), [9](#), [10](#), [11](#).

nmax: [1](#), [16](#).

print_circuits: [1](#), [18](#).

print_list: [4](#), [14](#).

print_set: [1](#), [13](#), [14](#), [15](#), [18](#).

printf: [7](#), [13](#), [14](#), [18](#).

r: [1](#).

rank: [1](#), [2](#), [11](#), [12](#), [15](#), [16](#), [17](#), [18](#).

S: [1](#).

scanf: [1](#), [12](#).

sort: [4](#), [16](#).

stderr: [1](#).

store: [11](#).

t: [8](#), [13](#), [15](#), [17](#).

tl: [16](#).

unmark: [17](#), [18](#).

unused: [1](#), [3](#), [5](#), [11](#).

v: [8](#), [15](#), [17](#).

x: [1](#).

y: [8](#).

- ⟨ Assign rank to sets and print independent ones 7 ⟩ Used in section 4.
- ⟨ Create empty list 5 ⟩ Used in section 4.
- ⟨ Find all sets in list nh that already contain y and remove excess elements from t 9 ⟩ Used in section 8.
- ⟨ Initialize list memory to available 3 ⟩ Used in section 1.
- ⟨ Insert $y \cup a$ for each $a \in t$ 10 ⟩ Used in section 8.
- ⟨ Pass from rank r to $r + 1$ 4 ⟩ Used in section 1.
- ⟨ Return list h to available storage 6 ⟩ Used in section 4.
- ⟨ Set initial contents of *rank* table 2 ⟩ Used in section 1.
- ⟨ Subroutines 8, 11, 12, 13, 14, 15, 16, 17, 18 ⟩ Used in section 1.

ERECTION

	Section	Page
Intro	1	1
Index	19	7