December 12, 2017 at 15:26

1. Intro. This program is part of a series of "exact cover solvers" that I'm putting together for my own education as I prepare to write Section 7.2.2.1 of *The Art of Computer Programming*. My intent is to have a variety of compatible programs on which I can run experiments, in order to learn how different approaches work in practice.

Indeed, this is the first of the series. I've tried to write it as a primitive baseline against which I'll be able to measure various technical improvements and extensions. DLX1 is based on the program DANCE, which I wrote hastily in 1999 while preparing my paper about "Dancing Links." [See *Selected Papers on Fun and Games* (2011), Chapter 38, for a revised version of that paper, which first appeared in the book *Millennial Perspectives in Computer Science*, a festschrift for C. A. R. Hoare (2000).] That program, incidentally, was based on a program called XCOVER that I first wrote in 1994. After using DANCE as a workhorse for more than 15 years, and after extending it in dozens of ways for a wide variety of combinatorial problems, I'm finally ready to replace it with a more carefully crafted piece of code.

My intention is to make this program match Algorithm 7.2.2.1D, so that I can use it to make the quantitative experiments that will ultimately be reported in Volume 4B.

Although this is the entry-level program, I'm taking care to adopt conventions for input and output that will be essentially the same (or at least backward compatible) in all of the fancier versions that are to come.

We're given a matrix of 0s and 1s, some of whose columns are called "primary" while the other columns are "secondary." Every row contains a 1 in at least one primary column. The problem is to find all subsets of its rows whose sum is (i) *exactly* 1 in all primary columns; (ii) *at most* 1 in all secondary columns.

This matrix, which is typically very sparse, is specified on *stdin* as follows:

- Each column has a symbolic name, from one to eight characters long. Each of those characters can be any nonblank ASCII code except for '`:`' and '`|`'.
- The first line of input contains the names of all primary columns, separated by one or more spaces, followed by '`|`', followed by the names of all other columns. (If all columns are primary, the '`|`' may be omitted.)
- The remaining lines represent the rows, by listing the columns where 1 appears.
- Additionally, "comment" lines can be interspersed anywhere in the input. Such lines, which begin with '`|`', are ignored by this program, but they are often useful within stored files.

Later versions of this program solve more general problems by making further use of the reserved characters '`:`' and '`|`' to allow additional kinds of input.

For example, if we consider the matrix

$$\begin{pmatrix} 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 0 & 1 \end{pmatrix}$$

which was (3) in my original paper, we can name the columns `A`, `B`, `C`, `D`, `E`, `F`, `G`. Suppose the first five are primary, and the latter two are secondary. That matrix can be represented by the lines

```
| A simple example
A B C D E | F G
C E F
A D G
B C F
A D
B G
D E G
```

(and also in many other ways, because column names can be given in any order, and so can the individual rows). It has a unique solution, consisting of the three rows `A D` and `E F C` and `B G`.

2. After this program finds all solutions, it normally prints their total number on *stderr*, together with statistics about how many nodes were in the search tree, and how many "updates" were made. The running time in "mems" is also reported, together with the approximate number of bytes needed for data storage. One "mem" essentially means a memory access to a 64-bit word. (These totals don't include the time or space needed to parse the input or to format the output.)

Here is the overall structure:

```
#define o mems++ /* count one mem */
#define oo mems += 2 /* count two mems */
#define ooo mems += 3 /* count three mems */
#define O "%" /* used for percent signs in format strings */
#define mod % /* used for percent signs denoting remainder in C */
#define max_level 500 /* at most this many rows in a solution */
#define max_cols 100000 /* at most this many columns */
#define max_nodes 25000000 /* at most this many nonzero elements in the matrix */
#define bufsize (9 * max_cols + 3) /* a buffer big enough to hold all column names */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include "gb_flip.h"
  typedef unsigned int uint; /* a convenient abbreviation */
  typedef unsigned long long ullng; /* ditto */
  ⟨ Type definitions 5 ⟩;
  ⟨ Global variables 3 ⟩;
  ⟨ Subroutines 9 ⟩;
  main(int argc, char *argv[])
  {
    register int cc, i, j, k, p, pp, q, r, t, cur_node, best_col;
    ⟨ Process the command line 4 ⟩;
    ⟨ Input the column names 13 ⟩;
    ⟨ Input the rows 16 ⟩;
    if (vbose & show_basics) ⟨ Report the successful completion of the input phase 20 ⟩;
    if (vbose & show_tots) ⟨ Report the column totals 21 ⟩;
    imems = mems, mems = 0;
    ⟨ Solve the problem 22 ⟩;
  done: if (vbose & show_tots) ⟨ Report the column totals 21 ⟩;
    if (vbose & show_profile) ⟨ Print the profile 33 ⟩;
    if (vbose & show_basics) {
      fprintf(stderr, "Altogether␣"O"llu␣solution"O"s,␣"O"llu+"O"llu␣mems,", count,
          count ≡ 1 ? "" : "s", imems, mems);
      bytes = last_col * sizeof(column) + last_node * sizeof(node) + maxl * sizeof(int);
      fprintf(stderr, "␣"O"llu␣updates,␣"O"llu␣bytes,␣"O"llu␣nodes.\n", updates, bytes, nodes);
    }
  }
```

3. You can control the amount of output, as well as certain properties of the algorithm, by specifying options on the command line:

- '`v`⟨ integer ⟩' enables or disables various kinds of verbose output on *stderr*, given by binary codes such as *show_choices*;
- '`m`⟨ integer ⟩' causes every mth solution to be output (the default is `m0`, which merely counts them);
- '`s`⟨ integer ⟩' causes the algorithm to make random choices in key places (thus providing some variety, although the solutions are by no means uniformly random), and it also defines the seed for any random numbers that are used;
- '`d`⟨ integer ⟩' to sets *delta*, which causes periodic state reports on *stderr* after the algorithm has performed approximately *delta* mems since the previous report;
- '`c`⟨ positive integer ⟩' limits the levels on which choices are shown during verbose tracing;
- '`C`⟨ positive integer ⟩' limits the levels on which choices are shown in the periodic state reports;
- '`l`⟨ nonnegative integer ⟩' gives a *lower* limit, relative to the maximum level so far achieved, to the levels on which choices are shown during verbose tracing;
- '`t`⟨ positive integer ⟩' causes the program to stop after this many solutions have been found;
- '`T`⟨ integer ⟩' sets *timeout* (which causes abrupt termination if *mems* > *timeout* at the beginning of a level).

```
#define show_basics 1 /* vbose code for basic stats; this is the default */
#define show_choices 2 /* vbose code for backtrack logging */
#define show_details 4 /* vbose code for further commentary */
#define show_profile 128 /* vbose code to show the search tree profile */
#define show_full_state 256 /* vbose code for complete state reports */
#define show_tots 512 /* vbose code for reporting column totals at start and end */
#define show_warnings 1024 /* vbose code for reporting rows without primaries */
```

⟨ Global variables 3 ⟩ ≡

```
int random_seed = 0; /* seed for the random words of gb_rand */
int randomizing; /* has 's' been specified? */
int vbose = show_basics + show_warnings; /* level of verbosity */
int spacing; /* solution k is output if k is a multiple of spacing */
int show_choices_max = 1000000; /* above this level, show_choices is ignored */
int show_choices_gap = 1000000; /* below level maxl − show_choices_gap, show_details is ignored */
int show_levels_max = 1000000; /* above this level, state reports stop */
int maxl = 0; /* maximum level actually reached */
char buf[bufsize]; /* input buffer */
ullng count; /* solutions found so far */
ullng rows; /* rows seen so far */
ullng imems, mems; /* mem counts */
ullng updates; /* update counts */
ullng bytes; /* memory used by main data structures */
ullng nodes; /* total number of branch nodes initiated */
ullng thresh = 0; /* report when mems exceeds this, if delta ≠ 0 */
ullng delta = 0; /* report every delta or so mems */
ullng maxcount = #ffffffffffffffff; /* stop after finding this many solutions */
ullng timeout = #1fffffffffffffff; /* give up after this many mems */
```

See also sections 7 and 23.

This code is used in section 2.

4. If an option appears more than once on the command line, the first appearance takes precedence.

```
⟨ Process the command line 4 ⟩ ≡
  for (j = argc − 1, k = 0; j; j−−)
    switch (argv[j][0]) {
    case 'v': k |= (sscanf(argv[j] + 1, ""O"d", &vbose) − 1); break;
    case 'm': k |= (sscanf(argv[j] + 1, ""O"d", &spacing) − 1); break;
    case 's': k |= (sscanf(argv[j] + 1, ""O"d", &random_seed) − 1), randomizing = 1; break;
    case 'd': k |= (sscanf(argv[j] + 1, ""O"lld", &delta) − 1), thresh = delta; break;
    case 'c': k |= (sscanf(argv[j] + 1, ""O"d", &show_choices_max) − 1); break;
    case 'C': k |= (sscanf(argv[j] + 1, ""O"d", &show_levels_max) − 1); break;
    case 'l': k |= (sscanf(argv[j] + 1, ""O"d", &show_choices_gap) − 1); break;
    case 't': k |= (sscanf(argv[j] + 1, ""O"lld", &maxcount) − 1); break;
    case 'T': k |= (sscanf(argv[j] + 1, ""O"lld", &timeout) − 1); break;
    default: k = 1;      /* unrecognized command-line option */
    }
  if (k) {
    fprintf(stderr, "Usage:␣"O"s␣[v<n>]␣[m<n>]␣[s<n>]␣[d<n>]""␣[c<n>]␣[C<n>]␣[l<n\
        >]␣[t<n>]␣[T<n>]␣<␣foo.dlx\n", argv[0]);
    exit(−1);
  }
  if (randomizing) gb_init_rand(random_seed);
```

This code is used in section 2.

5. Data structures. Each column of the input matrix is represented by a **column** struct, and each row is represented as a list of **node** structs. There's one node for each nonzero entry in the matrix.

More precisely, the nodes of individual rows appear sequentially, with "spacer" nodes between them. The nodes are also linked circularly within each column, in doubly linked lists. The column lists each include a header node, but the row lists do not. Column header nodes are aligned with a **column** struct, which contains further info about the column.

Each node contains three important fields. Two are the pointers *up* and *down* of doubly linked lists, already mentioned. The third points directly to the column containing the node.

A "pointer" is an array index, not a C reference (because the latter would occupy 64 bits and waste cache space). The *cl* array is for **column** structs, and the *nd* array is for **node**s. I assume that both of those arrays are small enough to be allocated statically. (Modifications of this program could do dynamic allocation if needed.) The header node corresponding to $cl[c]$ is $nd[c]$.

We count one mem for a simultaneous access to the *up* and *down* fields. I've added a *spare* field, so that each **node** occupies two octabytes.

Although the column-list pointers are called *up* and *down*, they need not correspond to actual positions of matrix entries. The elements of each column list can appear in any order, so that one row needn't be consistently "above" or "below" another. Indeed, when *randomizing* is set, we intentionally scramble each column list.

This program doesn't change the *col* fields after they've first been set up. But the *up* and *down* fields will be changed frequently, although preserving relative order.

Exception: In the node $nd[c]$ that is the header for the list of column c, we use the *col* field to hold the *length* of that list (excluding the header node itself). We also might use its *spare* field for special purposes. The alternative names *len* for *col* and *aux* for *spare* are used in the code so that this nonstandard semantics will be more clear.

A *spacer* node has $col \le 0$. Its *up* field points to the start of the preceding row; its *down* field points to the end of the following row. Thus it's easy to traverse a row circularly, in either direction.

If all rows have length m, we can do without the spacers by simply working modulo m. But the majority of my applications have rows of variable length, so I've decided not to use that trick.

[*Historical note:* An earlier version of this program, DLX0, was almost identical to this one except that it used doubly linked lists for the rows as well as for the columns. Thus it had two additional fields, *left* and *right*, in each node. When I wrote DLX1 I expected it to be a big improvement, because I thought there would be fewer memory accesses in all of the inner loops where rows are being traversed. However, I failed to realize that the *col* and *right* fields were both stored in the same octabyte; hence the cost per node is the same—and DLX1 actually performs a few *more* mems, as it handles the spacer node transitions! This additional mem cost is compensated by the smaller node size, hence greater likelihood of cache hits. But the gain from pure sequential allocation wasn't as great as I'd hoped.]

```
#define len col   /* column list length (used in header nodes only) */
#define aux spare   /* an auxiliary quantity (used in header nodes only) */
```

⟨ Type definitions 5 ⟩ ≡

```
typedef struct node_struct {
  int up, down;   /* predecessor and successor in column */
  int col;   /* the column containing this node */
  int spare;   /* padding, not used in DLX1 */
} node;
```

See also section 6.

This code is used in section 2.

6. Each **column** struct contains three fields: The *name* is the user-specified identifier; *next* and *prev* point to adjacent columns, when this column is part of a doubly linked list.

As backtracking proceeds, nodes will be deleted from column lists when their row has been blocked by other rows in the partial solution. But when backtracking is complete, the data structures will be restored to their original state.

We count one mem for a simultaneous access to the *prev* and *next* fields.

```
⟨ Type definitions 5 ⟩ +≡
  typedef struct col_struct {
    char name[8];      /* symbolic identification of the column, for printing */
    int prev, next;      /* neighbors of this column */
  } column;
```

7.
```
⟨ Global variables 3 ⟩ +≡
  node nd[max_nodes];      /* the master list of nodes */
  int last_node;      /* the first node in nd that's not yet used */
  column cl[max_cols + 2];      /* the master list of columns */
  int second = max_cols;      /* boundary between primary and secondary columns */
  int last_col;      /* the first column in cl that's not yet used */
```

8. One **column** struct is called the root. It serves as the head of the list of columns that need to be covered, and is identifiable by the fact that its *name* is empty.

```
#define root 0      /* cl[root] is the gateway to the unsettled columns */
```

9. A row is identified not by name but by the names of the columns it contains. Here is a routine that prints a row, given a pointer to any of its nodes. It also prints the position of the row in its column.

```
⟨ Subroutines 9 ⟩ ≡
  void print_row(int p, FILE *stream)
  {
    register int k, q;
    if (p < last_col ∨ p ≥ last_node ∨ nd[p].col ≤ 0) {
      fprintf(stderr, "Illegal row "O"d!\n", p);
      return;
    }
    for (q = p; ; ) {
      fprintf(stream, " "O".8s", cl[nd[q].col].name);
      q++;
      if (nd[q].col ≤ 0) q = nd[q].up;      /* −nd[q].col is actually the row number */
      if (q ≡ p) break;
    }
    for (q = nd[nd[p].col].down, k = 1; q ≠ p; k++) {
      if (q ≡ nd[p].col) {
        fprintf(stream, " (?)\n"); return;      /* row not in its column! */
      } else q = nd[q].down;
    }
    fprintf(stream, " ("O"d of "O"d)\n", k, nd[nd[p].col].len);
  }
  void prow(int p)
  {
    print_row(p, stderr);
  }
```

See also sections 10, 11, 25, 26, 31, and 32.

This code is used in section 2.

10. When I'm debugging, I might want to look at one of the current column lists.

```
⟨ Subroutines 9 ⟩ +≡
  void print_col(int c)
  {
    register int p;
    if (c < root ∨ c ≥ last_col) {
      fprintf(stderr, "Illegal column "O"d!\n", c);
      return;
    }
    if (c < second)
      fprintf(stderr, "Column "O".8s, length "O"d, neighbors "O".8s and "O".8s:\n",
          cl[c].name, nd[c].len, cl[cl[c].prev].name, cl[cl[c].next].name);
    else fprintf(stderr, "Column "O".8s, length "O"d:\n", cl[c].name, nd[c].len);
    for (p = nd[c].down; p ≥ last_col; p = nd[p].down) prow(p);
  }
```

11. Speaking of debugging, here's a routine to check if redundant parts of our data structure have gone awry.

```
#define sanity_checking 0  /* set this to 1 if you suspect a bug */
⟨ Subroutines 9 ⟩ +≡
  void sanity(void)
  {
    register int k, p, q, pp, qq, t;
    for (q = root, p = cl[q].next; ; q = p, p = cl[p].next) {
      if (cl[p].prev ≠ q) fprintf(stderr, "Bad␣prev␣field␣at␣col␣"O".8s!\n", cl[p].name);
      if (p ≡ root) break;
      ⟨ Check column p 12 ⟩;
    }
  }
```

12. ⟨ Check column p 12 ⟩ ≡

```
  for (qq = p, pp = nd[qq].down, k = 0; ; qq = pp, pp = nd[pp].down, k++) {
    if (nd[pp].up ≠ qq) fprintf(stderr, "Bad␣up␣field␣at␣node␣"O"d!\n", pp);
    if (pp ≡ p) break;
    if (nd[pp].col ≠ p) fprintf(stderr, "Bad␣col␣field␣at␣node␣"O"d!\n", pp);
  }
  if (nd[p].len ≠ k) fprintf(stderr, "Bad␣len␣field␣in␣column␣"O".8s!\n", cl[p].name);
```

This code is used in section 11.

13. Inputting the matrix. Brute force is the rule in this part of the code, whose goal is to parse and store the input data and to check its validity.

```
#define panic(m)
        { fprintf(stderr,""O"s!\n"O"d:␣"O".99s\n",m,p,buf); exit(-666); }
⟨ Input the column names 13 ⟩ ≡
  if (max_nodes ≤ 2 * max_cols) {
    fprintf(stderr,"Recompile␣me:␣max_nodes␣must␣exceed␣twice␣max_cols!\n");
    exit(-999);
  }    /* every column will want a header node and at least one other node */
  while (1) {
    if (¬fgets(buf,bufsize,stdin)) break;
    if (o,buf[p = strlen(buf) − 1] ≠ '\n') panic("Input␣line␣way␣too␣long");
    for (p = 0; o,isspace(buf[p]); p++) ;
    if (buf[p] ≡ '|' ∨ ¬buf[p]) continue;      /* bypass comment or blank line */
    last_col = 1;
    break;
  }
  if (¬last_col) panic("No␣columns");
  for ( ; o,buf[p]; ) {
    for (j = 0; j < 8 ∧ (o,¬isspace(buf[p + j])); j++) {
      if (buf[p + j] ≡ '|' ∨ buf[p + j] ≡ ':') panic("Illegal␣character␣in␣column␣name");
      o,cl[last_col].name[j] = buf[p + j];
    }
    if (j ≡ 8 ∧ ¬isspace(buf[p + j])) panic("Column␣name␣too␣long");
    ⟨ Check for duplicate column name 14 ⟩;
    ⟨ Initialize last_col to a new column with an empty list 15 ⟩;
    for (p += j + 1; o,isspace(buf[p]); p++) ;
    if (buf[p] ≡ '|') {
      if (second ≠ max_cols) panic("Column␣name␣line␣contains␣|␣twice");
      second = last_col;
      for (p++; o,isspace(buf[p]); p++) ;
    }
  }
  if (second ≡ max_cols) second = last_col;
  o,cl[root].prev = second − 1;      /* cl[second − 1].next = root since root = 0 */
  last_node = last_col;      /* reserve all the header nodes and the first spacer */
  o,nd[last_node].col = 0;
```

This code is used in section 2.

14. ⟨ Check for duplicate column name 14 ⟩ ≡

```
  for (k = 1; o,strncmp(cl[k].name,cl[last_col].name,8); k++) ;
  if (k < last_col) panic("Duplicate␣column␣name");
```

This code is used in section 13.

15. ⟨ Initialize *last_col* to a new column with an empty list 15 ⟩ ≡

```
  if (last_col > max_cols) panic("Too␣many␣columns");
  if (second ≡ max_cols) oo,cl[last_col − 1].next = last_col,cl[last_col].prev = last_col − 1;
  else o,cl[last_col].next = cl[last_col].prev = last_col;      /* nd[last_col].len = 0 */
  o,nd[last_col].up = nd[last_col].down = last_col;
  last_col++;
```

This code is used in section 13.

16. I'm putting the row number into the spacer that follows it, as a possible debugging aid. But the program doesn't currently use that information.

```
⟨ Input the rows 16 ⟩ ≡
  while (1) {
    if (¬fgets(buf, bufsize, stdin)) break;
    if (o, buf[p = strlen(buf) − 1] ≠ '\n') panic("Row␣line␣too␣long");
    for (p = 0; o, isspace(buf[p]); p++) ;
    if (buf[p] ≡ '|' ∨ ¬buf[p]) continue;      /* bypass comment or blank line */
    i = last_node;      /* remember the spacer at the left of this row */
    for (pp = 0; buf[p]; ) {
      for (j = 0; j < 8 ∧ (o, ¬isspace(buf[p + j])); j++) o, cl[last_col].name[j] = buf[p + j];
      if (j ≡ 8 ∧ ¬isspace(buf[p + j])) panic("Column␣name␣too␣long");
      if (j < 8) o, cl[last_col].name[j] = '\0';
      ⟨ Create a node for the column named in buf[p] 17 ⟩;
      for (p += j + 1; o, isspace(buf[p]); p++) ;
    }
    if (¬pp) {
      if (vbose & show_warnings) fprintf(stderr, "Row␣ignored␣(no␣primary␣columns):␣"O"s", buf);
      while (last_node > i) {
        ⟨ Remove last_node from its column 19 ⟩;
        last_node−−;
      }
    } else {
      o, nd[i].down = last_node;
      last_node++;      /* create the next spacer */
      if (last_node ≡ max_nodes) panic("Too␣many␣nodes");
      rows++;
      o, nd[last_node].up = i + 1;
      o, nd[last_node].col = −rows;
    }
  }
```

This code is used in section 2.

17. ⟨ Create a node for the column named in buf[p] 17 ⟩ ≡

```
  for (k = 0; o, strncmp(cl[k].name, cl[last_col].name, 8); k++) ;
  if (k ≡ last_col) panic("Unknown␣column␣name");
  if (o, nd[k].aux ≥ i) panic("Duplicate␣column␣name␣in␣this␣row");
  last_node++;
  if (last_node ≡ max_nodes) panic("Too␣many␣nodes");
  o, nd[last_node].col = k;
  if (k < second) pp = 1;
  o, t = nd[k].len + 1;
  ⟨ Insert node last_node into the list for column k 18 ⟩;
```

This code is used in section 16.

18. Insertion of a new node is simple, unless we're randomizing. In the latter case, we want to put the node into a random position of the list.

We store the position of the new node into $nd[k].aux$, so that the test for duplicate columns above will be correct.

As in other programs developed for TAOCP, I assume that four mems are consumed when 31 random bits are being generated by any of the GB_FLIP routines.

```
⟨ Insert node last_node into the list for column k 18 ⟩ ≡
  o, nd[k].len = t;      /* store the new length of the list */
  nd[k].aux = last_node;      /* no mem charge for aux after len */
  if (¬randomizing) {
    o, r = nd[k].up;      /* the "bottom" node of the column list */
    ooo, nd[r].down = nd[k].up = last_node, nd[last_node].up = r, nd[last_node].down = k;
  } else {
    mems += 4, t = gb_unif_rand(t);      /* choose a random number of nodes to skip past */
    for (o, r = k; t; o, r = nd[r].down, t--) ;
    ooo, q = nd[r].up, nd[q].down = nd[r].up = last_node;
    o, nd[last_node].up = q, nd[last_node].down = r;
  }
```

This code is used in section 17.

19.

```
⟨ Remove last_node from its column 19 ⟩ ≡
  o, k = nd[last_node].col;
  oo, nd[k].len--, nd[k].aux = i − 1;
  o, q = nd[last_node].up, r = nd[last_node].down;
  oo, nd[q].down = r, nd[r].up = q;
```

This code is used in section 16.

20.

```
⟨ Report the successful completion of the input phase 20 ⟩ ≡
  fprintf(stderr, "(" O"lld␣rows,␣" O"d+" O"d␣columns,␣" O"d␣entries␣successfully␣read)\n",
      rows, second − 1, last_col − second, last_node − last_col);
```

This code is used in section 2.

21. The column lengths after input should agree with the column lengths after this program has finished. I print them (on request), in order to provide some reassurance that the algorithm isn't badly screwed up.

```
⟨ Report the column totals 21 ⟩ ≡
  {
    fprintf(stderr, "Column␣totals:");
    for (k = 1; k < last_col; k++) {
      if (k ≡ second) fprintf(stderr, "␣|");
      fprintf(stderr, "␣" O"d", nd[k].len);
    }
    fprintf(stderr, "\n");
  }
```

This code is used in section 2.

22. The dancing. Our strategy for generating all exact covers will be to repeatedly choose always the column that appears to be hardest to cover, namely the column with shortest list, from all columns that still need to be covered. And we explore all possibilities via depth-first search.

The neat part of this algorithm is the way the lists are maintained. Depth-first search means last-in-first-out maintenance of data structures; and it turns out that we need no auxiliary tables to undelete elements from lists when backing up. The nodes removed from doubly linked lists remember their former neighbors, because we do no garbage collection.

The basic operation is "covering a column." This means removing it from the list of columns needing to be covered, and "blocking" its rows: removing nodes from other lists whenever they belong to a row of a node in this column's list.

```
⟨ Solve the problem 22 ⟩ ≡
  level = 0;
forward: nodes++;
  if (vbose & show_profile) profile[level]++;
  if (sanity_checking) sanity( );
  ⟨ Do special things if enough mems have accumulated 24 ⟩;
  ⟨ Set best_col to the best column for branching 29 ⟩;
  cover(best_col);
  oo, cur_node = choice[level] = nd[best_col].down;
advance: if (cur_node ≡ best_col) goto backup;
  if ((vbose & show_choices) ∧ level < show_choices_max) {
    fprintf(stderr, "L"O"d:", level);
    print_row(cur_node, stderr);
  }
  ⟨ Cover all other columns of cur_node 27 ⟩;
  if (o, cl[root].next ≡ root) ⟨ Record solution and goto recover 30 ⟩;
  if (++level > maxl) {
    if (level ≥ max_level) {
      fprintf(stderr, "Too␣many␣levels!\n");
      exit(−4);
    }
    maxl = level;
  }
  goto forward;
backup: uncover(best_col);
  if (level ≡ 0) goto done;
  level−−;
  oo, cur_node = choice[level], best_col = nd[cur_node].col;
recover: ⟨ Uncover all other columns of cur_node 28 ⟩;
  oo, cur_node = choice[level] = nd[cur_node].down; goto advance;
```

This code is used in section 2.

23.

```
⟨ Global variables 3 ⟩ +≡
  int level;      /* number of choices in current partial solution */
  int choice[max_level];      /* the node chosen on each level */
  ullng profile[max_level];      /* number of search tree nodes on each level */
```

24. ⟨ Do special things if enough *mems* have accumulated 24 ⟩ ≡

```
if (delta ∧ (mems ≥ thresh)) {
  thresh += delta;
  if (vbose & show_full_state) print_state( );
  else print_progress( );
}
if (mems ≥ timeout) {
  fprintf(stderr, "TIMEOUT!\n"); goto done;
}
```

This code is used in section 22.

25. When a row is blocked, it leaves all lists except the list of the column that is being covered. Thus a node is never removed from a list twice.

Note: I could have saved some mems in this routine, and in similar routines below, by not updating the *len* fields of secondary columns. But I chose not to make such an optimization because it might well be misleading: The insertion of a mem-free new branch '**if** (*cc* < *second*)' can be costly since it makes hardware branch prediction less effective. Furthermore those *len* fields are in column header nodes, which tend to remain in cache memory where they're readily accessible.

⟨ Subroutines 9 ⟩ +≡

```
void cover(int c)
{
  register int cc, l, r, rr, nn, uu, dd, t;
  o, l = cl[c].prev, r = cl[c].next;
  oo, cl[l].next = r, cl[r].prev = l;
  updates++;
  for (o, rr = nd[c].down; rr ≥ last_col; o, rr = nd[rr].down)
    for (nn = rr + 1; nn ≠ rr; ) {
      o, uu = nd[nn].up, dd = nd[nn].down;
      o, cc = nd[nn].col;
      if (cc ≤ 0) {
        nn = uu;
        continue;
      }
      oo, nd[uu].down = dd, nd[dd].up = uu;
      updates++;
      o, t = nd[cc].len − 1;
      o, nd[cc].len = t;
      nn++;
    }
}
```

26. I used to think that it was important to uncover a column by processing its rows from bottom to top, since covering was done from top to bottom. But while writing this program I realized that, amazingly, no harm is done if the rows are processed again in the same order. So I'll go downward again, just to prove the point. Whether we go up or down, the pointers execute an exquisitely choreographed dance that returns them almost magically to their former state.

```
⟨ Subroutines 9 ⟩ +≡
  void uncover(int c)
  {
    register int cc, l, r, rr, nn, uu, dd, t;
    for (o, rr = nd[c].down; rr ≥ last_col; o, rr = nd[rr].down)
      for (nn = rr + 1; nn ≠ rr; ) {
        o, uu = nd[nn].up, dd = nd[nn].down;
        o, cc = nd[nn].col;
        if (cc ≤ 0) {
          nn = uu;
          continue;
        }
        oo, nd[uu].down = nd[dd].up = nn;
        o, t = nd[cc].len + 1;
        o, nd[cc].len = t;
        nn++;
      }
    o, l = cl[c].prev, r = cl[c].next;
    oo, cl[l].next = cl[r].prev = c;
  }
```

27.

```
⟨ Cover all other columns of cur_node 27 ⟩ ≡
  for (pp = cur_node + 1; pp ≠ cur_node; ) {
    o, cc = nd[pp].col;
    if (cc ≤ 0) o, pp = nd[pp].up;
    else cover(cc), pp++;
  }
```

This code is used in section 22.

28. When I learned that the covering of individual columns can be done safely in various orders, I almost convinced myself that I'd be able to blithely ignore the ordering—I could apparently undo the covering of column a then b by uncovering a first. However, that argument is fallacious: When a is uncovered, it can resuscitate elements in column b that would mess up the uncovering of b. The choreography is delicate indeed.

(Incidentally, the *cover* and *uncover* routines both went to the right. That was okay. But we must then go left here.)

```
⟨ Uncover all other columns of cur_node 28 ⟩ ≡
  for (pp = cur_node − 1; pp ≠ cur_node; ) {
    o, cc = nd[pp].col;
    if (cc ≤ 0) o, pp = nd[pp].down;
    else uncover(cc), pp−−;
  }
```

This code is used in section 22.

29. The "best column" is considered to be a column that minimizes the number of remaining choices. If there are several candidates, we choose the leftmost — unless we're randomizing, in which case we select one of them at random.

```
⟨ Set best_col to the best column for branching 29 ⟩ ≡
  t = max_nodes;
  if ((vbose & show_details) ∧ level < show_choices_max ∧ level ≥ maxl − show_choices_gap)
    fprintf(stderr, "Level␣"O"d:", level);
  for (o, k = cl[root].next; k ≠ root; o, k = cl[k].next) {
    if ((vbose & show_details) ∧ level < show_choices_max ∧ level ≥ maxl − show_choices_gap)
      fprintf(stderr, "␣"O".8s("O"d)", cl[k].name, nd[k].len);
    if (o, nd[k].len ≤ t) {
      if (nd[k].len < t) best_col = k, t = nd[k].len, p = 1;
      else {
        p++;      /* this many columns achieve the min */
        if (randomizing ∧ (mems += 4, ¬gb_unif_rand(p))) best_col = k;
      }
    }
  }
  if ((vbose & show_details) ∧ level < show_choices_max ∧ level ≥ maxl − show_choices_gap)
    fprintf(stderr, "␣branching␣on␣"O".8s("O"d)\n", cl[best_col].name, t);
```

This code is used in section 22.

30.

```
⟨ Record solution and goto recover 30 ⟩ ≡
  {
    count++;
    if (spacing ∧ (count mod spacing ≡ 0)) {
      printf(""O"lld:\n", count);
      for (k = 0; k ≤ level; k++) print_row(choice[k], stdout);
      fflush(stdout);
    }
    if (count ≥ maxcount) goto done;
    goto recover;
  }
```

This code is used in section 22.

31.

```
⟨ Subroutines 9 ⟩ +≡
  void print_state(void)
  {
    register int l;
    fprintf(stderr, "Current␣state␣(level␣"O"d):\n", level);
    for (l = 0; l < level; l++) {
      print_row(choice[l], stderr);
      if (l ≥ show_levels_max) {
        fprintf(stderr, "␣...\n");
        break;
      }
    }
    fprintf(stderr, "␣"O"lld␣solutions,␣"O"lld␣mems,␣and␣max␣level␣"O"d␣so␣far.\n", count,
        mems, maxl);
  }
```

32. During a long run, it's helpful to have some way to measure progress. The following routine prints a string that indicates roughly where we are in the search tree. The string consists of character pairs, separated by blanks, where each character pair represents a branch of the search tree. When a node has d descendants and we are working on the kth, the two characters respectively represent k and d in a simple code; namely, the values 0, 1, ..., 61 are denoted by

`0, 1, ..., 9, a, b, ..., z, A, B, ...,Z.`

All values greater than 61 are shown as '`*`'. Notice that as computation proceeds, this string will increase lexicographically.

Following that string, a fractional estimate of total progress is computed, based on the naïve assumption that the search tree has a uniform branching structure. If the tree consists of a single node, this estimate is .5; otherwise, if the first choice is 'k of d', the estimate is $(k-1)/d$ plus $1/d$ times the recursively evaluated estimate for the kth subtree. (This estimate might obviously be very misleading, in some cases, but at least it grows monotonically.)

```
⟨ Subroutines 9 ⟩ +≡
  void print_progress(void)
  {
    register int l, k, d, c, p;
    register double f, fd;
    fprintf(stderr, "␣after␣"O"lld␣mems:␣"O"lld␣sols,", mems, count);
    for (f = 0.0, fd = 1.0, l = 0; l < level; l++) {
      c = nd[choice[l]].col, d = nd[c].len;
      for (k = 1, p = nd[c].down; p ≠ choice[l]; k++, p = nd[p].down) ;
      fd *= d, f += (k − 1)/fd;      /* choice l is k of d */
      fprintf(stderr, "␣"O"c"O"c", k < 10 ? '0' + k : k < 36 ? 'a' + k − 10 : k < 62 ? 'A' + k − 36 : '*',
          d < 10 ? '0' + d : d < 36 ? 'a' + d − 10 : d < 62 ? 'A' + d − 36 : '*');
      if (l ≥ show_levels_max) {
        fprintf(stderr, "...");
        break;
      }
    }
    fprintf(stderr, "␣"O".5f\n", f + 0.5/fd);
  }
```

33. ⟨ Print the profile 33 ⟩ ≡

```
  {
    fprintf(stderr, "Profile:\n");
    for (level = 0; level ≤ maxl; level++) fprintf(stderr, ""O"3d:␣"O"lld\n", level, profile[level]);
  }
```

This code is used in section 2.

34. Index.

DLX1