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.

Instead of actually solving an exact cover problem, DLX-PRE is a *preprocessor*: It converts the problem on *stdin* to an equivalent problem on *stdout*, removing any rows or columns that it finds to be unnecessary.

Here's a description of the input (and output) format, copied from DLX1: 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`.

DLX-PRE will simplify this drastically. First it will observe that every row containing `A` also contains `D`; hence column `D` can be removed from the matrix, as can the row `D E G`. Similarly we can removed column `F`; then column `C` and row `B C`. Now we can remove `G` and row `A G`. The result is a trivial problem, with three primary columns `A`, `B`, `E`, and three singleton rows `A`, `B`, `E`.

2. Furthermore, DLX2 extends DLX1 by allowing "color controls." Any row that specifies a "color" in a nonprimary column will rule out all rows that don't specify the same color in that column. But any number of rows whose nonprimary columns agree in color are allowed. (The previous situation was the special case in which every row corresponds to a distinct color.)

The input format is extended so that, if `xx` is the name of a nonprimary column, rows can contain entries of the form `xx:a`, where `a` is a single character (denoting a color).

Here, for example, is a simple test case:

```
| A simple example of color controls
A B C | X Y
A B X:0 Y:0
A C X:1 Y:1
X:0 Y:1
B X:1
C Y:1
```

The row `X:0 Y:1` will be deleted immediately, because it has no primary columns. The preprocessor will delete row `A B X:0 Y:0`, because that row can't be used without making column `C` uncoverable. Then column `C` can be eliminated, and row `C Y:1`.

3. These examples show that the simplified output may be drastically different from the original. It will have the same number of solutions; but by looking only at the simplified rows in those solutions, you may have no idea how to actually resolve the original problem! (Unless you work backward from the simplifications that were actually performed.)

The preprocessor for my SAT solvers had a counterpart called 'ERP', which converted solutions of the preprocessed problems into solutions of the original problems. DLX-PRE doesn't have that. But if you use the *show_orig_nos* option below, for example by saying '`v9`' when running DLX-PRE, you can figure out which rows of the original are solutions. The sets of rows that solve the simplified problem are the sets of rows that solve the original problem; the numbers given as comments by *show_orig_nos* provide the mapping between solutions.

For example, the simplified output from the first problem, using '`v9`', is:

```
A B C |
A
| (from 4)
B
| (from 5)
C
| (from 1)
```

And from the second problem it is similar, but not quite as simple:

```
A B | X Y
A X:1 Y:1
| (from 2)
B X:1
| (from 3)
```

4. Most of the code below, like the description above, has been cribbed from DLX2, with minor changes.

After this program does its work, it reports its running time in "mems"; one "mem" essentially means a memory access to a 64-bit word. (The given 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 7 ⟩;
  ⟨ Global variables 5 ⟩;
  ⟨ Subroutines 11 ⟩;
  main(int argc, char *argv[])
  {
    register int c, cc, dd, i, j, k, p, pp, q, qq, r, rr, rrr, t, uu, x, cur_node, best_col;
    ⟨ Process the command line 6 ⟩;
    ⟨ Input the column names 16 ⟩;
    ⟨ Input the rows 19 ⟩;
    if (vbose & show_basics) ⟨ Report the successful completion of the input phase 23 ⟩;
    if (vbose & show_tots) ⟨ Report the column totals 24 ⟩;
    imems = mems, mems = 0;
    ⟨ Reduce the problem 29 ⟩;
  finish: ⟨ Output the reduced problem 42 ⟩;
  done: if (vbose & show_tots) ⟨ Report the column totals 24 ⟩;
  all_done: if (vbose & show_basics) {
      fprintf(stderr,
          "Removed␣"O"d␣row"O"s␣and␣"O"d␣column"O"s,␣after␣"O"llu+"O"llu␣mems.\n",
          rows_out, rows_out ≡ 1 ? "" : "s", cols_out, cols_out ≡ 1 ? "" : "s", imems, mems);
    }
  }
```

5. 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*;
- '`d`⟨ integer ⟩' to sets *delta*, which causes periodic state reports on *stderr* after the algorithm has performed approximately *delta* mems since the previous report;
- '`t`⟨ positive integer ⟩' to specify the maximum number of rounds of row elimination that will be attempted.
- '`T`⟨ integer ⟩' sets *timeout* (which causes abrupt termination if *mems* > *timeout* at the beginning of a clause, but doesn't ruin the integrity of the output).

```
#define show_basics 1 /* vbose code for basic stats; this is the default */
#define show_choices 2 /* vbose code for general logging */
#define show_details 4 /* vbose code for further commentary */
#define show_orig_nos 8 /* vbose code to identify sources of output rows */
#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 5 ⟩ ≡

```
int vbose = show_basics; /* level of verbosity */
char buf[bufsize]; /* input buffer */
ullng rows; /* rows seen so far */
ullng imems, mems; /* mem counts */
ullng thresh = 0; /* report when mems exceeds this, if delta ≠ 0 */
ullng delta = 0; /* report every delta or so mems */
ullng timeout = #1fffffffffffff; /* give up after this many mems */
int rounds = max_nodes; /* maximum number of rounds attempted */
int rows_out, cols_out; /* this many reductions made so far */
```

See also sections 9 and 30.

This code is used in section 4.

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

⟨ Process the command line 6 ⟩ ≡

```
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 'd': k |= (sscanf(argv[j] + 1, ""O"lld", &delta) − 1), thresh = delta; break;
  case 't': k |= (sscanf(argv[j] + 1, ""O"d", &rounds) − 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>]␣[d<n>]␣[t<n>]␣[T<n>]␣<␣foo.dlx␣>␣bar.dlx\n", argv[0]);
  exit(−1);
}
```

This code is used in section 4.

7. 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 four important fields. Two are the pointers *up* and *down* of doubly linked lists, already mentioned. A third points directly to the column containing the node. And the last specifies a color, or zero if no color is specified.

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 $nd[c]$.

Notice that each **node** occupies two octabytes. We count one mem for a simultaneous access to the *up* and *down* fields, or for a simultaneous access to the *col* and *color* fields.

This program doesn't change the *col* fields after they've first been set up, except temporarily. 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 *color* field for special purposes. The alternative names *len* for *col* and *aux* for *color* 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.

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

⟨ Type definitions 7 ⟩ ≡

```
typedef struct node_struct {
  int up, down;   /* predecessor and successor in column */
  int col;   /* the column containing this node */
  int color;   /* the color specified by this node, if any */
} node;
```

See also section 8.

This code is used in section 4.

8. 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.

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

⟨ Type definitions 7 ⟩ +≡

```
typedef struct col_struct {
  char name[8];   /* symbolic identification of the column, for printing */
  int prev, next;   /* neighbors of this column */
} column;
```

9. ⟨ Global variables 5 ⟩ +≡

```
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 */
```

10. 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 */
```

11. 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.

This procedure differs slightly from its counterpart in DLX2: It uses '**while**' where DLX2 had '**if**'. The reason is that DLX-PRE sometimes deletes nodes, replacing them by spacers.

```
⟨ Subroutines 11 ⟩ ≡
  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);
      if (nd[q].color) fprintf(stream, ":"O"c", nd[q].color > 0 ? nd[q].color : nd[nd[q].col].color);
      q++;
      while (nd[q].col ≤ 0) q = nd[q].up;
      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 12, 13, 14, 27, and 28.

This code is used in section 4.

12. Another routine to print rows is used for diagnostics. It returns the original number of the row, and displays the not-yet-deleted columns in their original order. That original number (or rather its negative) appears in the spacer at the right of the row.

```
⟨ Subroutines 11 ⟩ +≡
  int dprow(int p, FILE *stream)
  {
    register int q, c;
    for (p--; nd[p].col > 0 ∨ nd[p].down < p; p--) ;
    for (q = p + 1; ; q++) {
      c = nd[q].col;
      if (c < 0) return -c;
      if (c > 0) {
        fprintf(stream, "␣"O".8s", cl[c].name);
        if (nd[q].color) fprintf(stream, ":"O"c", nd[q].color);
      }
    }
  }
```

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

```
⟨ Subroutines 11 ⟩ +≡
  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);
  }
```

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

```
#define sanity_checking 1   /* set this to 1 if you suspect a bug */
⟨ Subroutines 11 ⟩ +≡
  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 15 ⟩;
    }
  }
```

15. ⟨ Check column p 15 ⟩ ≡

```
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 14.

16. 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 16 ⟩ ≡
  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 17 ⟩;
    ⟨ Initialize last_col to a new column with an empty list 18 ⟩;
    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 4.

17. ⟨ Check for duplicate column name 17 ⟩ ≡

```
  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 16.

18. ⟨ Initialize last_col to a new column with an empty list 18 ⟩ ≡

```
  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 16.

19. In DLX1 and its descendants, I put the row number into the spacer that follows it, but only because I thought it might be a possible debugging aid. Now, in DLX-PRE, I'm glad I did, because we need this number when the user wants to relate the simplified output to the original unsimplified rows.

```
⟨ Input the rows 19 ⟩ ≡
  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])) ∧ buf[p + j] ≠ ':'; j++)
        o, cl[last_col].name[j] = buf[p + j];
      if (¬j) panic("Empty column name");
      if (j ≡ 8 ∧ ¬isspace(buf[p + j]) ∧ 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] 20 ⟩;
      if (buf[p + j] ≠ ':') o, nd[last_node].color = 0;
      else if (k ≥ second) {
        if ((o, isspace(buf[p + j + 1])) ∨ (o, ¬isspace(buf[p + j + 2])))
          panic("Color must be a single character");
        o, nd[last_node].color = buf[p + j + 1];
        p += 2;
      } else panic("Primary column must be uncolored");
      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 22 ⟩;
        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 4.

20. ⟨ Create a node for the column named in $buf[p]$ 20 ⟩ ≡

```
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 21 ⟩;
```

This code is used in section 19.

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

⟨ Insert node *last_node* into the list for column k 21 ⟩ ≡

```
o, nd[k].len = t;      /* store the new length of the list */
nd[k].aux = last_node;      /* no mem charge for aux after len */
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;
```

This code is used in section 20.

22. ⟨ Remove *last_node* from its column 22 ⟩ ≡

```
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 19.

23. ⟨ Report the successful completion of the input phase 23 ⟩ ≡

```
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 4.

24. The column lengths after input should agree with the column lengths after this program has finished—unless, of course, we've successfully simplified the input! I print them (on request), in order to provide some reassurance that the algorithm isn't badly screwed up.

⟨ Report the column totals 24 ⟩ ≡

```
{
  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 4.

25. The dancing. Suppose p is a primary column, and c is an arbitrary column such that every row containing p also contains an uncolored instance of c. Then we can delete column c, and every row that contains c but not p. For we'll need to cover p, and then c will automatically be covered too.

More generally, if p is a primary column and r is a row such that $p \notin r$ but every row containing p is incompatible with r, then we can eliminate row r: That row can't be chosen without making p uncoverable.

This program exploits those two ideas, by systematically looking at all rows in the list for column c, as c runs through all columns.

This algorithm takes "polynomial time," but I don't claim that it is fast. I want to get a straightforward algorithm in place before trying to make it more complicated.

On the other hand, I've tried to use the most efficient and scalable methods that I could think of, consistent with that goal of relative simplicity. There's no point in having a preprocessor unless it works fast enough to speed up the total time of preprocessing plus processing.

26. The basic operation is "hiding a column." This means causing all of the rows in its list to be invisible from outside the column, except for the rows that color this column; they are (temporarily) deleted from all other lists.

As in DLX2, the neat part of this algorithm is the way the lists are maintained. No auxiliary tables are needed when hiding a column, or when unhiding it later. The nodes removed from doubly linked lists remember their former neighbors, because we do no garbage collection.

27. Hiding is much like DLX2's "covering" operation, but it has a new twist: If the process of hiding column c causes one or more other primary columns p, p', ... to become empty, we know that c can be eliminated (as mentioned above). Furthermore we know that we can delete every row that contains c but not *all* of p, p', etc. After those deletions, columns p, p', ... will become identical; therefore we can eliminate all but one of them. Therefore the *hide* procedure counts the number of such p, and puts them onto a stack for later processing. The stack is maintained via the *aux* fields in column headers.

```
⟨ Subroutines 11 ⟩ +≡
  int hide(int c)
  {
    register int cc, l, r, rr, nn, uu, dd, t, k = 0;
    for (o, rr = nd[c].down; rr ≥ last_col; o, rr = nd[rr].down)
      if (o, ¬nd[rr].color) {
        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;
          o, t = nd[cc].len − 1;
          o, nd[cc].len = t;
          if (t ≡ 0 ∧ cc < second) k++, nd[cc].aux = stack, stack = cc;
          nn++;
        }
      }
    return k;
  }
```

28. Unhiding has yet another twist: We may want to remove rows and/or columns after this operation has occurred. So we do *not* restore primary columns whose *len* is zero, except for the primary column identified by *stack*.

```
⟨ Subroutines 11 ⟩ +≡
  void unhide(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)
      if (o, ¬nd[rr].color) {
        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;
          }
          o, t = nd[cc].len;
          if (t ∨ cc ≥ second ∨ (o, cc ≡ stack)) {
            oo, nd[uu].down = nd[dd].up = nn;
            o, nd[cc].len = t + 1;
          }
          nn++;
        }
      }
  }
```

29. Here then is the main loop for each round of preprocessing.

```
⟨ Reduce the problem 29 ⟩ ≡
  for (c = 1; c < second; c++)
    if (o, nd[c].len ≡ 0) ⟨ Terminate with unfeasible column c 31 ⟩;
  for (rnd = 1; rnd ≤ rounds; rnd++) {
    if (vbose & show_choices) fprintf(stderr, "Beginning␣round␣"O"d:\n", rnd);
    for (change = 0, c = 1; c < last_col; c++)
      if (o, nd[c].len) ⟨ Try to reduce rows in column c's list 32 ⟩;
    if (¬change) break;
  }
```

This code is used in section 4.

30.

```
⟨ Global variables 5 ⟩ +≡
  int rnd;      /* the current round */
  int stack;    /* top of stack of columns that are hard to cover */
  int change;   /* have we removed anything on the current round? */
  int zeros;    /* the number of zeros found by hide */
```

31. We might find a primary column that appears in no rows. In such a case *all* of the rows can be deleted, and all of the other columns!

⟨ Terminate with unfeasible column c 31 ⟩ ≡

```
{
  if (vbose & show_details) fprintf(stderr, "Primary␣column␣"O".8s␣is␣in␣no␣rows!\n", cl[c].name);
  rows_out = rows;
  cols_out = last_col − 1;
  printf(""O".8s\n", cl[c].name);      /* this is the only line of output */
  goto all_done;
}
```

This code is used in sections 29 and 33.

32. In order to avoid testing a row repeatedly, we usually try to remove it only when c is its first element as stored in memory.

⟨ Try to reduce rows in column c's list 32 ⟩ ≡

```
{
  if (sanity_checking) sanity( );
  if (delta ∧ (mems ≥ thresh)) {
    thresh += delta;
    fprintf(stderr, "␣after␣"O"lld␣mems:␣"O"d."O"d,␣"O"d␣cols␣out,␣"O"d␣rows␣out\n",
        mems, rnd, c, cols_out, rows_out);
  }
  if (mems ≥ timeout) goto finish;
  stack = 0, zeros = hide(c);
  if (zeros) ⟨ Remove zeros columns, and maybe some rows 33 ⟩
  else {
    for (o, r = nd[c].down; r ≥ last_col; o, r = nd[r].down) {
      for (q = r − 1; o, nd[q].down ≡ q − 1; q−−) ;      /* bypass null spacers */
      if (o, nd[q].col ≤ 0)      /* r is the first (surviving) node in its row */
        ⟨ Mark row r for deletion if it leaves some primary column uncoverable 37 ⟩;
    }
    unhide(c);
    for (r = stack; r; r = rr) {
      oo, rr = nd[r].col, nd[r].col = c;
      ⟨ Actually delete row r 41 ⟩;
    }
  }
}
```

This code is used in section 29.

33. ⟨ Remove *zeros* columns, and maybe some rows 33 ⟩ ≡

```
{
  unhide(c);
  if (vbose & show_details)
    fprintf(stderr, "Deleting␣column␣"O".8s,␣which␣blocks␣"O".8s\n", cl[c].name, cl[stack].name);
  for (o, p = nd[stack].aux; p; o, p = nd[p].aux) {
    if (vbose & show_details)
      fprintf(stderr, "Column␣"O".8s␣equals␣column␣"O".8s\n", cl[p].name, cl[stack].name);
    cols_out++;
  }
  for (change = 2, o, r = nd[c].down; r ≥ last_col; r = rrr) {
    o, rrr = nd[r].down;
    ⟨ Delete or shorten row r 34 ⟩;
  }
  if (change ≠ 1) {
    c = stack;      /* all of c's rows have been deleted */
    ⟨ Terminate with unfeasible column c 31 ⟩;
  }
  o, nd[c].up = nd[c].down = c;
  o, nd[c].len = 0, cols_out++;      /* now column c is gone */
}
```

This code is used in section 32.

34. We're in the driver's seat here: If row r includes *stack* and the $zeros - 1$ columns that are going away, we keep it, but remove the redundant columns. Otherwise we delete it.

⟨ Delete or shorten row r 34 ⟩ ≡

```
{
  for (o, t = zeros, q = r + 1; q ≠ r; ) {
    o, cc = nd[q].col;
    if (cc ≤ 0) {
      o, q = nd[q].up;
      continue;
    }
    if (cc ≥ second) {
      q++; continue;
    }
    if ((o, cc ≡ stack) ∨ (o, ¬nd[cc].len)) {
      t--;
      if (¬t) break;
    }
    q++;
  }
  if (q ≠ r) ⟨ Shorten and retain row r 35 ⟩
  else ⟨ Delete row r 36 ⟩;
}
```

This code is used in section 33.

35. ⟨ Shorten and retain row r 35 ⟩ ≡

```
{
  change = 1;
  if (vbose & show_details) {
    fprintf(stderr, "␣shortening");
    t = dprow(r, stderr), fprintf(stderr, "␣(row␣"O"d)\n", t);
  }
  o, nd[r].up = r + 1, nd[r].down = r − 1;      /* make node r into a spacer */
  o, nd[r].col = 0;
  for (o, t = zeros − 1, q = r − 1; t; ) {
    o, cc = nd[q].col;
    if (cc ≤ 0) {
      o, q = nd[q].down;
      continue;
    }
    if (cc ≥ second) {
      q−−;
      continue;
    }
    if (o, ¬nd[cc].len) {
      o, nd[q].up = q + 1, nd[q].down = q − 1;
      o, nd[q].col = 0;
      t−−;
    }
    q−−;
  }
}
```

This code is used in section 34.

36. Here we must remember that some nodes of row r might already be hidden, because of the (foolish?) exception to the rules that I made when writing *unhide*.

```
⟨ Delete row r 36 ⟩ ≡
  {
    if (vbose & show_details) {
      fprintf(stderr, "␣deleting");
      t = dprow(r, stderr), fprintf(stderr, "␣(row␣"O"d)\n", t);
    }
    rows_out ++;
    for (o, q = r + 1; q ≠ r; ) {
      o, cc = nd[q].col;
      if (cc ≤ 0) {
        o, q = nd[q].up;
        continue;
      }
      o, t = nd[cc].len − 1;
      if (t ≥ 0) {
        o, nd[cc].len = t;
        o, uu = nd[q].up, dd = nd[q].down;
        oo, nd[uu].down = dd, nd[dd].up = uu;
      }
      q++;
    }
  }
```

This code is used in section 34.

37. At this point we've hidden column c. Now we will hide also the other columns in row r; we'll delete r if this leaves some other primary column uncoverable. (As soon as such a column is encountered, we put it in pp and immediately back up.)

But before doing that test, we stamp the *aux* field of every non-c column of r with the number r. Then we'll know for sure whether or not we've blocked a column not in r.

When cc is a column in row r, with color x, the notion of "hiding column cc" means, more precisely, that we hide every row in cc's column list that clashes with row r. Row rr clashes with r if and only if either $x = 0$ or rr has cc with a color $\neq x$.

```
⟨ Mark row r for deletion if it leaves some primary column uncoverable 37 ⟩ ≡
  {
    for (q = r + 1; ; ) {
      o, cc = nd[q].col;
      if (cc ≤ 0) {
        o, q = nd[q].up;
        if (q > r) continue;
        break;      /* done with row */
      }
      o, nd[cc].aux = r, q++;
    }
    for (pp = 0, q = r + 1; ; ) {
      o, cc = nd[q].col;
      if (cc ≤ 0) {
        o, q = nd[q].up;
        if (q > r) continue;
        break;      /* done with row */
      }
      for (x = nd[q].color, o, p = nd[cc].down; p ≥ last_col; o, p = nd[p].down) {
        if (x > 0 ∧ (o, nd[p].color ≡ x)) continue;
        ⟨ Hide the entries of row p, or goto backup 38 ⟩;
      }
      q++;
    }
  backup: for (q = r − 1; q ≠ r; ) {
      o, cc = nd[q].col;
      if (cc ≤ 0) {
        o, q = nd[q].down;
        continue;
      }
      for (x = nd[q].color, o, p = nd[cc].up; p ≥ last_col; o, p = nd[p].up) {
        if (x > 0 ∧ (o, nd[p].color ≡ x)) continue;
        ⟨ Unhide the entries of row p 39 ⟩;
      }
      q−−;
    }
    if (pp) ⟨ Mark the unnecessary row r 40 ⟩;
  }
```

This code is used in section 32.

38. Long ago, in my paper "Structured programming with **go to** statements" [*Computing Surveys* **6** (December 1974), 261–301], I explained why it's sometimes legitimate to jump out of one loop into the midst of another. Now, after many years, I'm still jumping.

```
⟨ Hide the entries of row p, or goto backup 38 ⟩ ≡
  for (qq = p + 1; qq ≠ p; ) {
    o, cc = nd[qq].col;
    if (cc ≤ 0) {
      o, qq = nd[qq].up;
      continue;
    }
    o, t = nd[cc].len − 1;
    if (¬t ∧ cc < second ∧ nd[cc].aux ≠ r) {
      pp = cc;
      goto midst;      /* with fingers crossed */
    }
    o, nd[cc].len = t;
    o, uu = nd[qq].up, dd = nd[qq].down;
    oo, nd[uu].down = dd, nd[dd].up = uu;
    qq++;
  }
```

This code is used in section 37.

39.

```
⟨ Unhide the entries of row p 39 ⟩ ≡
  for (qq = p − 1; qq ≠ p; ) {
    o, cc = nd[qq].col;
    if (cc ≤ 0) {
      o, qq = nd[qq].down;
      continue;
    }
    oo, nd[cc].len++;
    o, uu = nd[qq].up, dd = nd[qq].down;
    oo, nd[uu].down = nd[dd].up = qq;
  midst: qq−−;
  }
```

This code is used in section 37.

40. When I first wrote this program, I reasoned as follows: "Row r has been hidden. So if we remove it from list c, the operation *unhide*(c) will keep it hidden. (And that's precisely what we want.)"

Boy, was I wrong! This change to list c fouled up the *unhide* routine, because things were not properly restored/undone after the list no longer told us to undo them. (Undeleted rows are mixed with deleted ones.)

The remedy is to mark the row, for deletion *later*. The marked rows are linked together via their *col* fields, which will no longer be needed for their former purpose.

```
⟨ Mark the unnecessary row r 40 ⟩ ≡
  {
    if (vbose & show_details) {
      fprintf(stderr, "␣"O".8s␣blocked␣by", cl[pp].name);
      t = dprow(r, stderr), fprintf(stderr, "␣(row␣"O"d)\n", t);
    }
    rows_out++, change = 1;
    o, nd[r].col = stack, stack = r;
  }
```

This code is used in section 37.

41.

```
⟨ Actually delete row r 41 ⟩ ≡
  for (p = r + 1; ; ) {
    o, cc = nd[p].col;
    if (cc ≤ 0) {
      o, p = nd[p].up;
      continue;
    }
    o, uu = nd[p].up, dd = nd[p].down;
    oo, nd[uu].down = dd, nd[dd].up = uu;
    oo, nd[cc].len--;
    if (p ≡ r) break;
    p++;
  }
```

This code is used in section 32.

42. The output phase. Okay, we're done!

```
⟨ Output the reduced problem 42 ⟩ ≡
  ⟨ Output the column names 43 ⟩;
  ⟨ Output the rows 44 ⟩;
```

This code is used in section 4.

43.
```
⟨ Output the column names 43 ⟩ ≡
  for (c = 1; c < last_col; c++) {
    if (c ≡ second) printf("␣|");
    if (o, nd[c].len) printf("␣"O".8s", cl[c].name);
  }
  printf("\n");
```

This code is used in section 42.

44.
```
⟨ Output the rows 44 ⟩ ≡
  for (c = 1; c < last_col; c++)
    if (o, nd[c].len) {
      for (o, r = nd[c].down; r ≥ last_col; o, r = nd[r].down) {
        for (q = r − 1; o, nd[q].down ≡ q − 1; q−−) ;
        if (o, nd[q].col ≤ 0) {      /* r was the leftmost survivor in its row */
          t = dprow(r, stdout);
          printf("\n");
          if (vbose & show_orig_nos) printf("|␣(from␣"O"d)\n", t);
        }
      }
    }
```

This code is used in section 42.

45. Index.

DLX-PRE