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 options or items 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 items are called "primary" while the other items are "secondary." Every option contains a 1 in at least one primary item. The problem is to find all subsets of its options whose sum is (i) *exactly* 1 in all primary items; (ii) *at most* 1 in all secondary items.

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

- Each item 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 items, separated by one or more spaces, followed by '`|`', followed by the names of all other items. (If all items are primary, the '`|`' may be omitted.)
- The remaining lines represent the options, by listing the items 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 items `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 item names can be given in any order, and so can the individual options). It has a unique solution, consisting of the three options `A D` and `E F C` and `B G`.

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

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

The input format is extended so that, if `xx` is the name of a nonprimary item, options 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 option `X:0 Y:1` will be deleted immediately, because it has no primary items. The preprocessor will delete option `A B X:0 Y:0`, because that option can't be used without making item `C` uncoverable. Then item `C` can be eliminated, and option `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 options 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 options of the original are solutions. The sets of options that solve the simplified problem are the sets of options 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 options in a solution */
#define max_itms 100000 /* at most this many items */
#define max_nodes 10000000 /* at most this many nonzero elements in the matrix */
#define bufsize (9 * max_itms + 3) /* a buffer big enough to hold all item 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_itm;
    ⟨ Process the command line 6 ⟩;
    ⟨ Input the item names 16 ⟩;
    ⟨ Input the options 19 ⟩;
    if (vbose & show_basics) ⟨ Report the successful completion of the input phase 23 ⟩;
    if (vbose & show_tots) ⟨ Report the item totals 24 ⟩;
    imems = mems, mems = 0;
    ⟨ Reduce the problem 29 ⟩;
  finish: ⟨ Output the reduced problem 43 ⟩;
  done: if (vbose & show_tots) ⟨ Report the item totals 24 ⟩;
  all_done: if (vbose & show_basics) {
      fprintf(stderr,
          "Removed␣"O"d␣option"O"s␣and␣"O"d␣item"O"s,␣after␣"O"llu+"O"llu␣mems,",
          options_out, options_out ≡ 1 ? "" : "s", itms_out, itms_out ≡ 1 ? "" : "s", imems, mems);
      fprintf(stderr, "␣"O"d␣round"O"s.\n", rnd, rnd ≡ 1 ? "" : "s");
    }
  }
```

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 (default 10000000000);
- '`t`⟨ positive integer ⟩' to specify the maximum number of rounds of option 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 options */
#define show_tots 512      /* vbose code for reporting item totals at start and end */
#define show_warnings 1024     /* vbose code for reporting options without primaries */
```

⟨ Global variables 5 ⟩ ≡

```
int vbose = show_basics + show_warnings;      /* level of verbosity */
char buf[bufsize];      /* input buffer */
ullng options;      /* options seen so far */
ullng imems, mems;      /* mem counts */
ullng thresh = 1000000000;      /* report when mems exceeds this, if delta ≠ 0 */
ullng delta = 10000000000;      /* report every delta or so mems */
ullng timeout = #1fffffffffffffff;      /* give up after this many mems */
int rounds = max_nodes;      /* maximum number of rounds attempted */
int options_out, itms_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 item of the input matrix is represented by a **item** struct, and each option 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 options appear sequentially, with "spacer" nodes between them. The nodes are also linked circularly within each item, in doubly linked lists. The item lists each include a header node, but the option lists do not. Item header nodes are aligned with a **item** struct, which contains further info about the item.

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 item 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 **item** 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 *itm* and *color* fields.

This program doesn't change the *itm* 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 item c, we use the *itm* 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 *itm* and *aux* for *color* are used in the code so that this nonstandard semantics will be more clear.

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

Spacer nodes are also used *within* an option, if that option has been shortened. The *up* and *down* fields in such spacers simply point to the next and previous elements. (We could optimize this by collapsing links, for example when several spacers are consecutive. But the present program doesn't do that.)

```
#define len itm    /* item 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 item */
  int itm;    /* the item 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 **item** struct contains three fields: The *name* is the user-specified identifier; *next* and *prev* point to adjacent items, when this item 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 itm_struct {
  char name[8];    /* symbolic identification of the item, for printing */
  int prev, next;    /* neighbors of this item */
} item;
```

9. ⟨ Global variables 5 ⟩ +≡

```
node *nd;      /* the master list of nodes */
int last_node;      /* the first node in nd that's not yet used */
item cl[max_itms + 2];      /* the master list of items */
int second = max_itms;      /* boundary between primary and secondary items */
int last_itm;      /* the first item in cl that's not yet used */
```

10. One **item** struct is called the root. It serves as the head of the list of items 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 items */
```

11. An option is identified not by name but by the names of the items it contains. Here is a routine that prints an option, given a pointer to any of its nodes. It also prints the position of the option in its item.

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_option(int p, FILE *stream)
{
  register int k, q;
  if (p < last_itm ∨ p ≥ last_node ∨ nd[p].itm ≤ 0) {
    fprintf(stderr, "Illegal option "O"d!\n", p);
    return;
  }
  for (q = p; ; ) {
    fprintf(stream, " "O".8s", cl[nd[q].itm].name);
    if (nd[q].color) fprintf(stream, ":"O"c", nd[q].color > 0 ? nd[q].color : nd[nd[q].itm].color);
    q++;
    while (nd[q].itm ≤ 0) q = nd[q].up;
    if (q ≡ p) break;
  }
  for (q = nd[nd[p].itm].down, k = 1; q ≠ p; k++) {
    if (q ≡ nd[p].itm) {
      fprintf(stream, " (?)\n"); return;      /* option not in its item! */
    } else q = nd[q].down;
  }
  fprintf(stream, " ("O"d of "O"d)\n", k, nd[nd[p].itm].len);
}
void prow(int p)
{
  print_option(p, stderr);
}
```

See also sections 12, 13, 14, 27, and 28.

This code is used in section 4.

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

```
⟨ Subroutines 11 ⟩ +≡
  int dpoption(int p, FILE *stream)
  {
    register int q, c;
    for (p--; nd[p].itm > 0 ∨ nd[p].down < p; p--) ;
    for (q = p + 1; ; q++) {
      c = nd[q].itm;
      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 item lists.

```
⟨ Subroutines 11 ⟩ +≡
  void print_itm(int c)
  {
    register int p;
    if (c < root ∨ c ≥ last_itm) {
      fprintf(stderr, "Illegal␣item␣"O"d!\n", c);
      return;
    }
    if (c < second)
      fprintf(stderr, "Item␣"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, "Item␣"O".8s,␣length␣"O"d:\n", cl[c].name, nd[c].len);
    for (p = nd[c].down; p ≥ last_itm; 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 0      /* 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␣itm␣"O".8s!\n", cl[p].name);
      if (p ≡ root) break;
      ⟨ Check item p 15 ⟩;
    }
  }
```

15. ⟨ Check item 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].itm ≠ p) fprintf(stderr, "Bad itm field at node "O"d!\n", pp);
}
if (nd[p].len ≠ k) fprintf(stderr, "Bad len field in item "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 item names 16 ⟩ ≡
  nd = (node *) calloc(max_nodes, sizeof(node));
  if (¬nd) {
    fprintf(stderr, "I␣couldn't␣allocate␣space␣for␣"O"d␣nodes!\n", max_nodes);
    exit(-666);
  }
  if (max_nodes ≤ 2 * max_itms) {
    fprintf(stderr, "Recompile␣me:␣max_nodes␣must␣exceed␣twice␣max_itms!\n");
    exit(-999);
  }     /* every item 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_itm = 1;
    break;
  }
  if (¬last_itm) panic("No␣items");
  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␣item␣name");
      o, cl[last_itm].name[j] = buf[p + j];
    }
    if (j ≡ 8 ∧ ¬isspace(buf[p + j])) panic("Item␣name␣too␣long");
    ⟨ Check for duplicate item name 17 ⟩;
    ⟨ Initialize last_itm to a new item with an empty list 18 ⟩;
    for (p += j + 1; o, isspace(buf[p]); p++) ;
    if (buf[p] ≡ '|') {
      if (second ≠ max_itms) panic("Item␣name␣line␣contains␣|␣twice");
      second = last_itm;
      for (p++; o, isspace(buf[p]); p++) ;
    }
  }
  if (second ≡ max_itms) second = last_itm;
  o, cl[root].prev = second − 1;      /* cl[second − 1].next = root since root = 0 */
  last_node = last_itm;      /* reserve all the header nodes and the first spacer */
  o, nd[last_node].itm = 0;
```

This code is used in section 4.

17. ⟨ Check for duplicate item name 17 ⟩ ≡

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

This code is used in section 16.

18. ⟨ Initialize *last_itm* to a new item with an empty list 18 ⟩ ≡

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

This code is used in section 16.

19. In DLX1 and its descendants, I put the option 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 options.

⟨ Input the options 19 ⟩ ≡

```
while (1) {
  if (¬fgets(buf, bufsize, stdin)) break;
  if (o, buf[p = strlen(buf) − 1] ≠ '\n') panic("Option 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 option */
  for (pp = 0; buf[p]; ) {
    for (j = 0; j < 8 ∧ (o, ¬isspace(buf[p + j])) ∧ buf[p + j] ≠ ':'; j++)
      o, cl[last_itm].name[j] = buf[p + j];
    if (¬j) panic("Empty item name");
    if (j ≡ 8 ∧ ¬isspace(buf[p + j]) ∧ buf[p + j] ≠ ':') panic("Item name too long");
    if (j < 8) o, cl[last_itm].name[j] = '\0';
    ⟨ Create a node for the item 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 = (unsigned char) buf[p + j + 1];
      p += 2;
    } else panic("Primary item must be uncolored");
    for (p += j + 1; o, isspace(buf[p]); p++) ;
  }
  if (¬pp) {
    if (vbose & show_warnings) fprintf(stderr, "Option ignored (no primary items):␣"O"s", buf);
    while (last_node > i) {
      ⟨ Remove last_node from its item 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");
    options++;
    o, nd[last_node].up = i + 1;
    o, nd[last_node].itm = −options;
  }
}
```

This code is used in section 4.

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

```
for (k = 0; o, strncmp(cl[k].name, cl[last_itm].name, 8); k++) ;
if (k ≡ last_itm) panic("Unknown␣item␣name");
if (o, nd[k].aux ≥ i) panic("Duplicate␣item␣name␣in␣this␣option");
last_node++;
if (last_node ≡ max_nodes) panic("Too␣many␣nodes");
o, nd[last_node].itm = k;
if (k < second) pp = 1;
o, t = nd[k].len + 1;
⟨ Insert node last_node into the list for item 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 items above will be correct.

⟨ Insert node *last_node* into the list for item 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 item 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 item 22 ⟩ ≡

```
o, k = nd[last_node].itm;
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␣options,␣" O "d+" O "d␣items,␣" O "d␣entries␣successfully␣read)\n",
        options, second − 1, last_itm − second, last_node − last_itm);
```

This code is used in section 4.

24. The item lengths after input should agree with the item 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 item totals 24 ⟩ ≡

```
{
  fprintf(stderr, "Item␣totals:");
  for (k = 1; k < last_itm; 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 item, and c is an arbitrary item such that every option containing p also contains an uncolored instance of c. Then we can delete item c, and every option 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 item and r is an option such that $p \notin r$ but every option containing p is incompatible with r, then we can eliminate option r: That option can't be chosen without making p uncoverable.

This program exploits those two ideas, by systematically looking at all options in the list for item c, as c runs through all items.

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 an item." This means causing all of the options in its list to be invisible from outside the item, except for the options that color this item; 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 an item, 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 item c causes at least one primary item p to become empty, we know that c can be eliminated (as mentioned above). Furthermore we know that we can delete every option that contains c but not p.

Therefore the *hide* procedure puts the value of such p in a global variable, for use by the caller. That global variable is called '*stack*' for historical reasons: My first implementation had an unnecessarily complex mechanism for dealing with several primary items that simultaneously become empty, so I used to put them onto a stack.

```
⟨ Subroutines 11 ⟩ +≡
  void hide(int c)
  {
    register int cc, l, r, rr, nn, uu, dd, t, k = 0;
    for (o, rr = nd[c].down; rr ≥ last_itm; 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].itm;
          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) stack = cc;
          nn++;
        }
      }
  }
```

28. ⟨ Subroutines 11 ⟩ +≡

```
void unhide(int c)
{
  register int cc, l, r, rr, nn, uu, dd, t;
  for (o, rr = nd[c].down; rr ≥ last_itm; 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].itm;
        if (cc ≤ 0) {
          nn = uu;
          continue;
        }
        o, t = nd[cc].len;
        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 (cc = 1; cc < last_itm; cc++)
  if (o, nd[cc].len ≡ 0) ⟨ Take note that cc has no options 41 ⟩;
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_itm; c++)
    if (o, nd[c].len) ⟨ Try to reduce options in item c's list 31 ⟩;
  if (¬change) break;
}
```

This code is used in section 4.

30. ⟨ Global variables 5 ⟩ +≡

```
int rnd;      /* the current round */
int stack;    /* a blocked item; or top of stack of options to delete */
int change;   /* have we removed anything on the current round? */
```

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

Note (after correcting a bug, 02 January 2023): If c is secondary and has a nonzero color in option r, we should *not* try to remove r, because r has not been hidden by the *hide* routine. Thus we might miss some potential deletions. Users can avoid this by putting all of the colored secondary items last in every option.

```
⟨ Try to reduce options in item c's list 31 ⟩ ≡
  {
    if (sanity_checking) sanity();
    if (delta ∧ (mems ≥ thresh)) {
      thresh += delta;
      fprintf(stderr, "␣after␣"O"lld␣mems:␣"O"d."O"d,␣"O"d␣items␣out,␣"O"d␣options␣out\n",
          mems, rnd, c, itms_out, options_out);
    }
    if (mems ≥ timeout) goto finish;
    stack = 0, hide(c);
    if (stack) ⟨ Remove item c, and maybe some options 32 ⟩
    else {
      for (o, r = nd[c].down; r ≥ last_itm; o, r = nd[r].down) {
        for (q = r − 1; o, nd[q].down ≡ q − 1; q−−) ;      /* bypass null spacers */
        if (o, nd[q].itm ≤ 0 ∧ (o, ¬nd[r].color))
            /* r is the first (surviving, uncolored) node in its option */
          ⟨ Stack option r for deletion if it leaves some primary item uncoverable 36 ⟩;
      }
      unhide(c);
      for (r = stack; r; r = rr) {
        oo, rr = nd[r].itm, nd[r].itm = c;
        ⟨ Actually delete option r 40 ⟩;
      }
    }
  }
```

This code is used in section 29.

32.

```
⟨ Remove item c, and maybe some options 32 ⟩ ≡
  {
    unhide(c);
    if (vbose & show_details)
      fprintf(stderr, "Deleting␣item␣"O".8s,␣forced␣by␣"O".8s\n", cl[c].name, cl[stack].name);
    for (o, r = nd[c].down; r ≥ last_itm; r = rrr) {
      o, rrr = nd[r].down;
      ⟨ Delete or shorten option r 33 ⟩;
    }
    o, nd[c].up = nd[c].down = c;
    o, nd[c].len = 0, itms_out++;      /* now item c is gone */
    change = 1;
  }
```

This code is used in section 31.

33. We're in the driver's seat here: If option r includes *stack*, we keep it, but remove item c. Otherwise we delete it.

```
⟨ Delete or shorten option r 33 ⟩ ≡
  {
    for (q = r + 1; q ≠ r; ) {
      o, cc = nd[q].itm;
      if (cc ≤ 0) {
        o, q = nd[q].up;
        continue;
      }
      if (cc ≡ stack) break;
      q++;
    }
    if (q ≠ r) ⟨ Shorten and retain option r 34 ⟩
    else ⟨ Delete option r 35 ⟩;
  }
```

This code is used in section 32.

34.

```
⟨ Shorten and retain option r 34 ⟩ ≡
  {
    if (vbose & show_details) {
      fprintf(stderr, "␣shortening");
      t = dpoption(r, stderr), fprintf(stderr, "␣(option␣"O"d)\n", t);
    }
    o, nd[r].up = r + 1, nd[r].down = r − 1;      /* make node r into a spacer */
    o, nd[r].itm = 0;
  }
```

This code is used in section 33.

35.

```
⟨ Delete option r 35 ⟩ ≡
  {
    if (vbose & show_details) {
      fprintf(stderr, "␣deleting");
      t = dpoption(r, stderr), fprintf(stderr, "␣(option␣"O"d)\n", t);
    }
    options_out++;
    for (o, q = r + 1; q ≠ r; ) {
      o, cc = nd[q].itm;
      if (cc ≤ 0) {
        o, q = nd[q].up;
        continue;
      }
      o, t = nd[cc].len − 1;
      if (t ≡ 0) ⟨ Take note that cc has no options 41 ⟩;
      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 33.

36. At this point we've hidden item c and option r. Now we'll hide also the other items in that option; and we'll delete r if this leaves some other primary item uncoverable. (As soon as such an item 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 item of r with the number r. Then we'll know for sure whether or not we've blocked an item not in r.

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

```
⟨ Stack option r for deletion if it leaves some primary item uncoverable 36 ⟩ ≡
  {
    for (q = r + 1; ; ) {
      o, cc = nd[q].itm;
      if (cc ≤ 0) {
        o, q = nd[q].up;
        if (q > r) continue;
        break;      /* done with option */
      }
      o, nd[cc].aux = r, q++;
    }
    for (pp = 0, q = r + 1; ; ) {
      o, cc = nd[q].itm;
      if (cc ≤ 0) {
        o, q = nd[q].up;
        if (q > r) continue;
        break;      /* done with option */
      }
      for (x = nd[q].color, o, p = nd[cc].down; p ≥ last_itm; o, p = nd[p].down) {
        if (x > 0 ∧ (o, nd[p].color ≡ x)) continue;
        ⟨ Hide the entries of option p, or goto backup 37 ⟩;
      }
      q++;
    }
  backup: for (q = r − 1; q ≠ r; ) {
      o, cc = nd[q].itm;
      if (cc ≤ 0) {
        o, q = nd[q].down;
        continue;
      }
      for (x = nd[q].color, o, p = nd[cc].up; p ≥ last_itm; o, p = nd[p].up) {
        if (x > 0 ∧ (o, nd[p].color ≡ x)) continue;
        ⟨ Unhide the entries of option p 38 ⟩;
      }
      q−−;
    }
    if (pp) ⟨ Mark the unnecessary option r 39 ⟩;
  }
```

This code is used in section 31.

37. 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 option p, or goto backup 37 ⟩ ≡
  for (qq = p + 1; qq ≠ p; ) {
    o, cc = nd[qq].itm;
    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 36.

38.

```
⟨ Unhide the entries of option p 38 ⟩ ≡
  for (qq = p − 1; qq ≠ p; ) {
    o, cc = nd[qq].itm;
    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 36.

39. When I first wrote this program, I reasoned as follows: "Option 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 options are mixed with deleted ones.)

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

```
⟨ Mark the unnecessary option r 39 ⟩ ≡
  {
    if (vbose & show_details) {
      fprintf(stderr, "␣"O".8s␣blocked␣by", cl[pp].name);
      t = dpoption(r, stderr), fprintf(stderr, "␣(option␣"O"d)\n", t);
    }
    options_out++, change = 1;
    o, nd[r].itm = stack, stack = r;
  }
```

This code is used in section 36.

40.

```
⟨ Actually delete option r 40 ⟩ ≡
  for (p = r + 1; ; ) {
    o, cc = nd[p].itm;
    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 (nd[cc].len ≡ 0) ⟨ Take note that cc has no options 41 ⟩;
    if (p ≡ r) break;
    p++;
  }
```

This code is used in section 31.

41.

```
⟨ Take note that cc has no options 41 ⟩ ≡
  {
    itms_out++;
    if (cc ≥ second) {
      if (vbose & show_details) fprintf(stderr, "␣"O".8s␣is␣in␣no␣options\n", cl[cc].name);
    } else ⟨ Terminate with unfeasible item cc 42 ⟩;
  }
```

This code is used in sections 29, 35, and 40.

42. We might find a primary item that appears in no options. In such a case *all* of the options can be deleted, and all of the other items!

```
⟨ Terminate with unfeasible item cc 42 ⟩ ≡
  {
    if (vbose & show_details)
      fprintf(stderr, "Primary␣item␣"O".8s␣is␣in␣no␣options!\n", cl[cc].name);
    options_out = options;
    itms_out = last_itm − 1;
    printf(""O".8s\n", cl[cc].name);      /* this is the only line of output */
    goto all_done;
  }
```

This code is used in section 41.

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

```
⟨ Output the reduced problem 43 ⟩ ≡
  ⟨ Output the item names 44 ⟩;
  ⟨ Output the options 45 ⟩;
```

This code is used in section 4.

```
44. ⟨ Output the item names 44 ⟩ ≡
  for (c = 1; c < last_itm; c++) {
    if (c ≡ second) printf("␣|");
    if (o, nd[c].len) printf("␣"O".8s", cl[c].name);
  }
  printf("\n");
```

This code is used in section 43.

```
45. ⟨ Output the options 45 ⟩ ≡
  for (c = 1; c < last_itm; c++)
    if (o, nd[c].len) {
      for (o, r = nd[c].down; r ≥ last_itm; o, r = nd[r].down) {
        for (q = r − 1; o, nd[q].down ≡ q − 1; q−−) ;
        if (o, nd[q].itm ≤ 0) {      /* r was the leftmost survivor in its option */
          t = dpoption(r, stdout);
          printf("\n");
          if (vbose & show_orig_nos) printf("|␣(from␣"O"d)\n", t);
        }
      }
    }
```

This code is used in section 43.

46. Index.

DLX-PRE