November 24, 2020 at 13:23

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.

The basic input format for all of these solvers is described at the beginning of program DLX1, and you should read that description now if you are unfamiliar with it.

DLX2 extends DLX1 by allowing "color controls," which give considerably more flexibility to nonprimary items: Any option that specifies a "color" in a nonprimary item will rule out all options that don't specify the same color for 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, because it has no primary items. The unique solution consists of options `A C X:1 Y:1` and `B X:1`.

If the input contains no color specifications, the behavior of DLX2 will almost exactly match that of DLX1, except for having a slightly longer program and taking a bit longer to input the options.

[*Historical note:* My first program for color-controlled exact covering was GDANCE, written in November 2000 when I was thinking about two-dimensional de Bruijn sequences. Later I came gradually to realize that the idea has many, many other applications. Indeed, in 2016 I noticed that the general constraint satisfaction problem can actually be regarded as a special case, when the allowable joint constraints are explicitly listed.]

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" and "cleansings" were made. The running time in "mems" is also reported, together with the approximate number of bytes needed for data storage. (An "update" is the removal of an option from its item list. A "cleansing" is the removal of a satisfied color constraint from its option. One "mem" essentially means a memory access to a 64-bit word. The reported 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 5000     /* at most this many options in a solution */
#define max_cols 100000     /* at most this many items */
#define max_nodes 10000000     /* at most this many nonzero elements in the matrix */
#define bufsize (9 * max_cols + 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 6⟩;
  ⟨Global variables 3⟩;
  ⟨Subroutines 10⟩;
  main(int argc, char *argv[])
  {
    register int cc, i, j, k, p, pp, q, r, t, cur_node, best_itm;
    ⟨Process the command line 4⟩;
    ⟨Input the item names 14⟩;
    ⟨Input the options 17⟩;
    if (vbose & show_basics) ⟨Report the successful completion of the input phase 21⟩;
    if (vbose & show_tots) ⟨Report the item totals 22⟩;
    imems = mems, mems = 0;
    ⟨Solve the problem 23⟩;
  done: if (sanity_checking) sanity();
    if (vbose & show_tots) ⟨Report the item totals 22⟩;
    if (vbose & show_profile) ⟨Print the profile 37⟩;
    if (vbose & show_basics) {
      fprintf(stderr, "Altogether␣"O"llu␣solution"O"s,␣"O"llu+"O"llu-"O"llu␣mems,", count,
          count ≡ 1 ? "" : "s", imems, mems, lmems);
      bytes = last_itm * sizeof(item) + last_node * sizeof(node) + maxl * sizeof(int);
      fprintf(stderr, "␣"O"llu␣updates,␣"O"llu␣cleansings,", updates, cleansings);
      fprintf(stderr, "␣"O"llu␣bytes,␣"O"llu␣nodes.\n", bytes, nodes);
    }
    ⟨Close the files 5⟩;
  }
```

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 ⟩' sets *delta*, which causes periodic state reports on *stderr* after the algorithm has performed approximately *delta* mems since the previous report (default 10000000000);
- '`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);
- '`S`⟨ filename ⟩' to output a "shape file" that encodes the search tree.

```
#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 item totals at start and end */
#define show_warnings 1024  /* vbose code for reporting options 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 options;  /* options seen so far */
ullng imems, mems, lmems;  /* mem counts */
ullng updates;  /* update counts */
ullng cleansings;  /* cleansing counts */
ullng bytes;  /* memory used by main data structures */
ullng nodes;  /* total number of branch nodes initiated */
ullng thresh = 10000000000;  /* report when mems exceeds this, if delta ≠ 0 */
ullng delta = 10000000000;  /* report every delta or so mems */
ullng maxcount = #ffffffffffffffff;  /* stop after finding this many solutions */
ullng timeout = #1fffffffffffffff;  /* give up after this many mems */
FILE *shape_file;  /* file for optional output of search tree shape */
char *shape_name;  /* its name */
```

See also sections 8 and 24.

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;
  case 'S': shape_name = argv[j] + 1, shape_file = fopen(shape_name, "w");
    if (¬shape_file)
      fprintf(stderr, "Sorry,␣I␣can't␣open␣file␣`"O"s'␣for␣writing!\n", shape_name);
    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>]␣[S<bar>]␣<␣foo.dlx\n", argv[0]);
  exit(-1);
}
if (randomizing) gb_init_rand(random_seed);
```

This code is used in section 2.

5. ⟨ Close the files 5 ⟩ ≡

```
if (shape_file) fclose(shape_file);
```

This code is used in section 2.

6. Data structures. Each item of the input matrix is represented by an **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 with respect to 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 an **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.

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

This program doesn't change the *itm* 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 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.

The *color* field of a node is set to -1 when that node has been cleansed. In such cases its original color appears in the item header. (The program uses this fact only for diagnostic outputs.)

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

⟨Type definitions 6⟩ ≡

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

See also section 7.

This code is used in section 2.

7. 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.

As backtracking proceeds, nodes will be deleted from item lists when their option has been hidden by other options 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 6⟩ +≡

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

8. ⟨ 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 */
item cl[max_cols + 2];  /* the master list of items */
int second = max_cols;  /* boundary between primary and secondary items */
int last_itm;  /* the first item in cl that's not yet used */
```

9. 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 */
```

10. 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 list.

⟨ Subroutines 10 ⟩ ≡

```
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++;
    if (nd[q].itm ≤ 0) q = nd[q].up;  /* −nd[q].itm is actually the option number */
    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 list! */
    } 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 11, 12, 26, 27, 30, 31, 35, and 36.

This code is used in section 2.

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

```
⟨ Subroutines 10 ⟩ +≡
  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);
  }
```

12. 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 10 ⟩ +≡
  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 13 ⟩;
    }
  }
```

13.

```
⟨ Check item p 13 ⟩ ≡
  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 12.

14. 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 14 ⟩ ≡
  if (max_nodes ≤ 2 ∗ max_cols) {
    fprintf(stderr, "Recompile␣me:␣max_nodes␣must␣exceed␣twice␣max_cols!\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 15 ⟩;
    ⟨ Initialize last_itm to a new item with an empty list 16 ⟩;
    for (p += j + 1; o, isspace(buf[p]); p++) ;
    if (buf[p] ≡ '|') {
      if (second ≠ max_cols) panic("Item␣name␣line␣contains␣|␣twice");
      second = last_itm;
      for (p++; o, isspace(buf[p]); p++) ;
    }
  }
  if (second ≡ max_cols) second = last_itm;
  oo, cl[last_itm].prev = last_itm − 1, cl[last_itm − 1].next = last_itm;
  oo, cl[second].prev = last_itm, cl[last_itm].next = second;
    /∗ this sequence works properly whether or not second = last_itm ∗/
  oo, cl[root].prev = second − 1, cl[second − 1].next = root;
  last_node = last_itm;    /∗ reserve all the header nodes and the first spacer ∗/
    /∗ we have nd[last_node].itm = 0 in the first spacer ∗/
```

This code is used in section 2.

15.

```
⟨ Check for duplicate item name 15 ⟩ ≡
  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 14.

16. ⟨Initialize *last_itm* to a new item with an empty list 16⟩ ≡

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

This code is used in section 14.

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

⟨Input the options 17⟩ ≡

```
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] 18⟩;
    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 list 20⟩;
      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 2.

18. ⟨ Create a node for the item named in *buf* [*p*] 18 ⟩ ≡

```
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 19⟩;
```

This code is used in section 17.

19. 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 items 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 item *k* 19 ⟩ ≡

```
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 item 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 18.

20. ⟨ Remove *last_node* from its item list 20 ⟩ ≡

```
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 17.

21. ⟨ Report the successful completion of the input phase 21 ⟩ ≡

```
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 2.

22. The item lengths after input should agree with the item 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 item totals 22⟩ ≡

```
{
  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 2.

23. The dancing. Our strategy for generating all exact covers will be to repeatedly choose always the item that appears to be hardest to cover, namely the item with shortest list, from all items 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 an item." This means removing it from the list of items needing to be covered, and "hiding" its options: removing nodes from other lists whenever they belong to an option of a node in this item's list.

```
⟨Solve the problem 23⟩ ≡
  level = 0;
forward: nodes++;
  if (vbose & show_profile) profile[level]++;
  if (sanity_checking) sanity();
  ⟨Do special things if enough mems have accumulated 25⟩;
  ⟨Set best_itm to the best item for branching 32⟩;
  cover(best_itm);
  oo, cur_node = choice[level] = nd[best_itm].down;
advance: if (cur_node ≡ best_itm) goto backup;
  if ((vbose & show_choices) ∧ level < show_choices_max) {
    fprintf(stderr, "L"O"d:", level);
    print_option(cur_node, stderr);
  }
  ⟨Cover all other items of cur_node 28⟩;
  if (o, cl[root].next ≡ root) ⟨Visit a solution and goto recover 33⟩;
  if (++level > maxl) {
    if (level ≥ max_level) {
      fprintf(stderr, "Too␣many␣levels!\n");
      exit(−4);
    }
    maxl = level;
  }
  goto forward;
backup: uncover(best_itm);
  if (level ≡ 0) goto done;
  level−−;
  oo, cur_node = choice[level], best_itm = nd[cur_node].itm;
recover: ⟨Uncover all other items of cur_node 29⟩;
  oo, cur_node = choice[level] = nd[cur_node].down; goto advance;
```

This code is used in section 2.

24. ⟨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 */
```

25. ⟨ Do special things if enough *mems* have accumulated 25 ⟩ ≡

```
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 23.

26. When an option is hidden, it leaves all lists except the list of the item that is being covered. Thus a node is never removed from a list twice.

We can save time by not removing nodes from secondary items that have been purified. (Such nodes have *color* < 0. Note that *color* and *itm* are stored in the same octabyte; hence we pay only one mem to look at them both.)

We could save even more time by not updating the *len* fields of secondary items. Instead of suppressing that calculation, this program calculates how much would be saved.

⟨ Subroutines 10 ⟩ +≡

```
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_itm; o, rr = nd[rr].down)
    for (nn = rr + 1; nn ≠ rr; ) {
      if (o, nd[nn].color ≥ 0) {
        o, uu = nd[nn].up, dd = nd[nn].down;
        cc = nd[nn].itm;
        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;
        if (cc ≥ second) lmems += 2;
      }
      nn++;
    }
}
```

27. I used to think that it was important to uncover an item by processing its options 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 options 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 10⟩ +≡
```
void uncover(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)
    for (nn = rr + 1; nn ≠ rr; ) {
      if (o, nd[nn].color ≥ 0) {
        o, uu = nd[nn].up, dd = nd[nn].down;
        cc = nd[nn].itm;
        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;
        if (cc ≥ second) lmems += 2;
      }
      nn++;
    }
  o, l = cl[c].prev, r = cl[c].next;
  oo, cl[l].next = cl[r].prev = c;
}
```

28. ⟨Cover all other items of *cur_node* 28⟩ ≡
```
for (pp = cur_node + 1; pp ≠ cur_node; ) {
  o, cc = nd[pp].itm;
  if (cc ≤ 0) o, pp = nd[pp].up;
  else {
    if (¬nd[pp].color) cover(cc);
    else if (nd[pp].color > 0) purify(pp);
    pp++;
  }
}
```
This code is used in section 23.

29. We must go leftward as we uncover the items, because we went rightward when covering them.

⟨ Uncover all other items of *cur_node* 29 ⟩ ≡

```
for (pp = cur_node − 1; pp ≠ cur_node; ) {
  o, cc = nd[pp].itm;
  if (cc ≤ 0) o, pp = nd[pp].down;
  else {
    if (¬nd[pp].color) uncover(cc);
    else if (nd[pp].color > 0) unpurify(pp);
    pp−−;
  }
}
```

This code is used in section 23.

30. When we choose an option that specifies colors in one or more items, we "purify" those items by removing all incompatible options. All options that want the chosen color in a purified item are temporarily given the color code −1 so that they won't be purified again.

⟨ Subroutines 10 ⟩ +≡

```
void purify(int p)
{
  register int cc, rr, nn, uu, dd, t, x;
  o, cc = nd[p].itm, x = nd[p].color;
  nd[cc].color = x;    /* no mem charged, because this is for print_option only */
  cleansings++;
  for (o, rr = nd[cc].down; rr ≥ last_itm; o, rr = nd[rr].down) {
    if (rr ≡ p) fprintf(stderr, "confusion!\n");
    if (o, nd[rr].color ≠ x) {
      for (nn = rr + 1; nn ≠ rr; ) {
        if (o, nd[nn].color ≥ 0) {
          o, uu = nd[nn].up, dd = nd[nn].down;
          cc = nd[nn].itm;
          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;
          if (cc ≥ second) lmems += 2;
        }
        nn++;
      }
    } else cleansings++, o, nd[rr].color = −1;
  }
}
```

31. Just as *purify* is analogous to *cover*, the inverse process is analogous to *uncover*.

⟨ Subroutines 10 ⟩ +≡

```
void unpurify(int p)
{
  register int cc, rr, nn, uu, dd, t, x;
  o, cc = nd[p].itm, x = nd[p].color;    /* there's no need to clear nd[cc].color */
  for (o, rr = nd[cc].up; rr ≥ last_itm; o, rr = nd[rr].up) {
    if (rr ≡ p) fprintf(stderr, "confusion!\n");
    if (o, nd[rr].color < 0) o, nd[rr].color = x;
    else {
      for (nn = rr − 1; nn ≠ rr; ) {
        if (o, nd[nn].color ≥ 0) {
          o, uu = nd[nn].up, dd = nd[nn].down;
          cc = nd[nn].itm;
          if (cc ≤ 0) {
            nn = dd; continue;
          }
          oo, nd[uu].down = nd[dd].up = nn;
          o, t = nd[cc].len + 1;
          o, nd[cc].len = t;
          if (cc ≥ second) lmems += 2;
        }
        nn−−;
      }
    }
  }
}
```

32. The "best item" is considered to be an item 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_itm* to the best item for branching 32 ⟩ ≡

```
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; t ∧ 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_itm = k, t = nd[k].len, p = 1;
    else {
      p++;   /* this many items achieve the min */
      if (randomizing ∧ (mems += 4, ¬gb_unif_rand(p))) best_itm = 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_itm].name, t);
if (shape_file) {
  fprintf(shape_file, ""O"d␣"O".8s\n", t, cl[best_itm].name);
  fflush(shape_file);
}
```

This code is used in section 23.

33. ⟨ Visit a solution and **goto** *recover* 33 ⟩ ≡

```
{
  nodes++;   /* a solution is a special node, see 7.2.2–(4) */
  if (level + 1 > maxl) {
    if (level + 1 ≥ max_level) {
      fprintf(stderr, "Too␣many␣levels!\n");
      exit(−5);
    }
    maxl = level + 1;
  }
  if (vbose & show_profile) profile[level + 1]++;
  if (shape_file) {
    fprintf(shape_file, "sol\n"); fflush(shape_file);
  }
  ⟨ Record solution and goto recover 34 ⟩;
}
```

This code is used in section 23.

34. ⟨ Record solution and **goto** *recover* 34 ⟩ ≡

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

This code is used in section 33.

35. ⟨ Subroutines 10 ⟩ +≡

```
void print_state(void)
{
  register int l;
  fprintf(stderr, "Current␣state␣(level␣"O"d):\n", level);
  for (l = 0; l < level; l++) {
    print_option(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);
}
```

36. 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

$$\texttt{0},\ \texttt{1},\ \ldots,\ \texttt{9},\ \texttt{a},\ \texttt{b},\ \ldots,\ \texttt{z},\ \texttt{A},\ \texttt{B},\ \ldots,\texttt{Z}.$$

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 10 ⟩ +≡

```
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]].itm, 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);
}
```

37. ⟨ Print the profile 37 ⟩ ≡

```
{
  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.

38. Index.

⟨ Check for duplicate item name 15 ⟩ Used in section 14.
⟨ Check item p 13 ⟩ Used in section 12.
⟨ Close the files 5 ⟩ Used in section 2.
⟨ Cover all other items of *cur_node* 28 ⟩ Used in section 23.
⟨ Create a node for the item named in $buf[p]$ 18 ⟩ Used in section 17.
⟨ Do special things if enough *mems* have accumulated 25 ⟩ Used in section 23.
⟨ Global variables 3, 8, 24 ⟩ Used in section 2.
⟨ Initialize *last_itm* to a new item with an empty list 16 ⟩ Used in section 14.
⟨ Input the item names 14 ⟩ Used in section 2.
⟨ Input the options 17 ⟩ Used in section 2.
⟨ Insert node *last_node* into the list for item k 19 ⟩ Used in section 18.
⟨ Print the profile 37 ⟩ Used in section 2.
⟨ Process the command line 4 ⟩ Used in section 2.
⟨ Record solution and **goto** *recover* 34 ⟩ Used in section 33.
⟨ Remove *last_node* from its item list 20 ⟩ Used in section 17.
⟨ Report the item totals 22 ⟩ Used in section 2.
⟨ Report the successful completion of the input phase 21 ⟩ Used in section 2.
⟨ Set *best_itm* to the best item for branching 32 ⟩ Used in section 23.
⟨ Solve the problem 23 ⟩ Used in section 2.
⟨ Subroutines 10, 11, 12, 26, 27, 30, 31, 35, 36 ⟩ Used in section 2.
⟨ Type definitions 6, 7 ⟩ Used in section 2.
⟨ Uncover all other items of *cur_node* 29 ⟩ Used in section 23.
⟨ Visit a solution and **goto** *recover* 33 ⟩ Used in section 23.

DLX2