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 programs DLX1 and DLX2; you should read that description now if you are unfamiliar with it.

This program modifies DLX2 by caching the results of partial solutions. Its output is not a list of solutions, but rather a ZDD that characterizes them. (The basic ideas are due to Masaaki Nishino, Norihito Yasuda, Shin-ichi Minato, and Masaaki Nagata, whose paper "Dancing with decision diagrams" appeared in the 31st AAAI Conference on Articial Intelligence (2017), pages 868–874. However, I've extended it from the exact cover problem to the considerably more general MCC problem, by adding color constraints and multiplicities.)

The ZDD is output in the text format accepted by the ZDDREAD programs, which I prepared long ago in connection with BDD15 and other software. A dummy node is placed at the root of the ZDD, so that ZDDREAD will know where to start. This ZDD is not properly ordered, in general; but I think the ZDDREAD programs will still work. (Knock on wood.)

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, how many "updates" and "cleansings" were made, how many ZDD nodes were created, and how many cache memos 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 items and spacers in all options */
#define max_inx 200000   /* at most this many items and item-color pairs */
#define max_cache 2000000000   /* octabytes in the cache */
        /* N.B.: max_cache must be less than 2^32, because of hashentry */
#define loghashsize 30
#define hashsize (1 << loghashsize)   /* octabytes in the hash table */
#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, znode, zsol, optionno, hit;
    ⟨Process the command line 4⟩;
    ⟨Input the item names 14⟩;
    ⟨Input the options 17⟩;
    ⟨Initialize the memo cache 27⟩;
    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 36⟩;
  done: if (sanity_checking) sanity();
    if (spacing) printf(""O"x:␣(~0?0:"O"x)\n", zddnodes, znode);   /* the root of the ZDD */
    if (vbose & show_tots) ⟨Report the item totals 22⟩;
    if (vbose & show_profile) ⟨Print the profile 50⟩;
    if (vbose & show_basics) {
      fprintf(stderr, "Altogether␣"O"llu␣solution"O"s,␣"O"llu+"O"llu␣mems,", count,
          count ≡ 1 ? "" : "s", imems, mems);
      bytes = last_itm * sizeof(item) + last_node * sizeof(node) + maxl * sizeof(int);
```

```
    bytes += sigptr * sizeof (inx) + cacheptr * sizeof(ullng);
    bytes += (2 * hashcount > hashsize ? hashsize : 2 * hashcount) * sizeof (hashentry);
    fprintf(stderr, "␣"O"llu␣updates,␣"O"llu␣cleansings,", updates, cleansings);
    fprintf(stderr, "␣"O"llu␣bytes,␣"O"llu␣search␣nodes,", bytes, nodes);
    fprintf(stderr, "␣"O"u␣ZDD␣node"O"s,␣"O"u+"O"u␣signatures,␣"O"llu␣hits.\n",
        zddnodes ≡ 2 ? 1 : zddnodes, zddnodes ≡ 2 ? "" : "s", memos − goodmemos, goodmemos + 1, hits);
      /* I added 1 because the book says the all-zero signature is in the cache */
  }
  ⟨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⟩', if nonzero, causes the ZDD to be output (the default is `m0`, which merely counts the solutions);
- '`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 searching for additional solutions, after this many have been found;
- '`T`⟨integer⟩' sets *timeout* (which causes abrupt termination if $mems > timeout$ at the beginning of a level);
- '`Z`⟨positive integer⟩' sets *maxzdd* (which causes early termination if $zddnodes > maxzdd$); `Z0` will give just the first solution;
- '`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_hits 8  /* vbose code to show cache hits */
#define show_secondary_details 16  /* vbose code to show active secondary lists */
#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;  /* a ZDD is output if spacing ≠ 0 */
  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;  /* 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 maxzdd = #ffffffffffffffff;  /* stop after finding this many ZDD nodes */
  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, 25, and 37.

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 'Z': k |= (sscanf(argv[j] + 1, ""O"lld", &maxzdd) − 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>]␣[Z<n]␣<␣foo.dlx\n", argv[0]);
  exit(−1);
}
if (randomizing) gb_init_rand(random_seed);
else gb_init_rand(0);
```

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 sections 7, 23, and 24.

This code is used in section 2.

7. Each **item** struct contains five 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; *sig* and *offset* are part of the memo-cache mechanism explained below.

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; also one mem for a simultaneous access to both *sig* and *offset*.

⟨ Type definitions 6 ⟩ +≡
```
typedef struct itm_struct {
  char name[8];  /* symbolic identification of the item, for printing */
  int prev, next;  /* neighbors of this item */
  int sig, offset;  /* fields for constructing signatures for the memo cache */
} 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, cc;
    if (p < last_itm ∨ p ≥ last_node ∨ nd[p].itm ≤ 0) {
      fprintf(stderr, "Illegal␣option␣"O"d!\n", p);
      return;
    }
    for (q = p, cc = nd[q].itm; ; ) {
      fprintf(stream, "␣"O".8s", cl[cc].name);
      if (nd[q].color) fprintf(stream, ":"O"c",
            nd[q].color > 0 ? siginx[cl[cc].sig + nd[q].color].orig : siginx[cl[cc].sig + nd[cc].color].orig);
      q++;
      cc = nd[q].itm;
      if (cc ≤ 0) q = nd[q].up, cc = nd[q].itm;    /* −cc 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, 39, 40, 43, 44, 48, and 49.

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 memo cache. This program has special data structures by which we can tell if the current covering-and-purification status matches a previous status. Each status is converted to a multibit signature, with one bit for each primary item, and possibly several bits for each second item that can be colored in several ways. Every potential contribution to the signature is specified by an 8-byte **inx** structure.

⟨ Type definitions 6 ⟩ +≡
```
typedef struct inx_struct {
  int hash;      /* bits used to randomize the signature */
  short code;    /* what bits should be set in that octabyte? */
  char shift;    /* by how much should be code be shifted? */
  char orig;     /* the original character used for a color */
} inx;
```

24. A large hash table is used to help decide which signatures are currently known. Its entries are octabytes with two fields:

⟨ Type definitions 6 ⟩ +≡
```
typedef struct hash_struct {
  int sig;       /* where the signature can be found in the cache array */
  int zddref;    /* the ZDD node that corresponds to this signature */
} hashentry;
```

25. A multibit signature consists of one or more octabytes, all but the last of which have the sign bit set. It is preceded in *cache* by an octabyte that contains the count of all solutions represented by its ZDD node.

⟨ Global variables 3 ⟩ +≡
```
inx siginx[max_inx];      /* indexes for making signatures */
int sigptr;      /* this many siginx entries are in use */
int sigsiz;      /* this many octabytes per signature */
hashentry *hash;      /* hash table for locating signatures */
int hashcount;      /* this many items are in the hash table */
ullng *cache;      /* the memo cache */
unsigned int cacheptr;      /* this many octabytes of cache are in use */
unsigned int oldcacheptr;      /* this many were in use a moment ago */
unsigned int zddnodes = 2;      /* total ZDD nodes created */
unsigned int memos;      /* this many configurations were cached */
unsigned int goodmemos;      /* of which this many had solutions */
ullng hits;      /* total number of cache hits */
char usedcolor[256], colormap[256];      /* tables for color code renumbering */
```

26. The colors of a secondary item are mapped into small positive integers, so that the signature will be compact. For example, if the colors are `a` and `b`, we'll change them to 1 and 2; but the original names will be remembered in the *orig* field. In this case there will be three *code* values, occupying two bits of the signature: $code = 1$ when the item is unpurified; $code = 2$ when it has been purified to 1; $code = 3$ when it has been purified to 2.

The *siginx* table entry for item k is accessed by $cl[k].sig$ when k is primary, or by $cl[k].sig + nd[k].color$ when k is secondary. That entry will tell us what bits should be contributed to octabyte $cl[k].offset$ of the overall multibit signature, and it will also contribute to the 32-bit hash code of the full signature.

27. We give the smallest offsets to the items with the largest numbers, hoping that many of the signatures will be cached after all of the small-numbered items have been covered.

```
#define overflow(p, pname)
          { fprintf(stderr, "Overflow␣in␣cache␣memory␣(%s=%d)!\n", pname, p); exit(−667); }
⟨Initialize the memo cache 27⟩ ≡
  hash = (hashentry ∗) malloc(hashsize ∗ sizeof(hashentry));
  if (¬hash) {
    fprintf(stderr, "Couldn't␣allocate␣the␣hash␣table␣(hashsize=%d)!\n", hashsize);
    exit(−68);
  }
  cache = (ullng ∗) malloc(max_cache ∗ sizeof(ullng));
  if (¬cache) {
    fprintf(stderr, "Couldn't␣allocate␣the␣cache␣memory␣(max_cache=%d)!\n", max_cache);
    exit(−69);
  }
  q = 1, r = 0;    /∗ offset and position within the multibit signature ∗/
  for (k = last_itm − 1; k; k−−)
    if (k < second) ⟨Prepare for a primary item signature 28⟩
    else ⟨Prepare for a secondary item signature 29⟩;
  sigsiz = q + 1;
```

This code is used in section 2.

28. ⟨Prepare for a primary item signature 28⟩ ≡

```
  {
    if (r ≡ 63) q++, r = 0;    /∗ the sign bit is used for continuations ∗/
    o, siginx[sigptr].shift = r, siginx[sigptr].code = 1;
    mems += 4, siginx[sigptr].hash = gb_next_rand();
    o, cl[k].sig = sigptr++, cl[k].offset = q;
    if (sigptr ≥ max_inx) overflow(max_inx, "max_inx");
    r++;
  }
```

This code is used in section 27.

29. ⟨ Prepare for a secondary item signature 29 ⟩ ≡

```
{
  if (o, nd[k].down ≡ k) {      /* unused secondary item */
    register l, r;
    o, l = cl[k].prev, r = cl[k].next;
    oo, cl[l].next = r, cl[r].prev = l;
    continue;     /* it disappears */
  }
  o, nd[k].color = 0;
  cc = 1;
  for (p = nd[k].down; p > k; o, p = nd[p].down) {
    o, i = nd[p].color;
    if (i) {
      o, t = usedcolor[i];
      if (¬t) oo, colormap[cc] = i, usedcolor[i] = cc++;
      o, nd[p].color = usedcolor[i];     /* the original color is permanently changed */
    }
  }
  for (t = 1; cc ≥ (1 ≪ t); t++) ;     /* t = ⌊lg cc⌋ + 1 slots in the signature */
  if (sigptr + t ≥ max_inx) overflow(max_inx, "max_inx");
  if (r + t ≥ 63) q++, r = 0;
  for (i = 0; i < cc; i++) {
    o, siginx[sigptr + i].shift = r, siginx[sigptr + i].code = 1 + i;
    oo, siginx[sigptr + i].orig = colormap[i], usedcolor[colormap[i]] = 0;
    mems += 4, siginx[sigptr + i].hash = gb_next_rand();
    o, cl[k].sig = sigptr, cl[k].offset = q;
  }
  sigptr += cc, r += t;
}
```

This code is used in section 27.

30. **#define** *signbit* #8000000000000000

⟨ Look for the current status in the memo cache 30 ⟩ ≡

```
{
  register ullng sigacc;
  register unsigned int sighash;
  register int off, sig, offset;
  if (cacheptr + sigsiz ≥ max_cache) overflow(max_cache, "max_cache");
  sighash = 0, off = 1, sigacc = 0;
  for (o, k = cl[last_itm].prev; k ≠ last_itm; o, k = cl[k].prev)
    ⟨ Contribute a secondary item to the signature 32 ⟩;
  for (o, k = cl[root].prev; k ≠ root; o, k = cl[k].prev) ⟨ Contribute a primary item to the signature 31 ⟩;
  o, cache[cacheptr + off] = sigacc;
  ⟨ Do the hash lookup 33 ⟩;
}
```

This code is used in section 36.

31. ⟨Contribute a primary item to the signature 31⟩ ≡

```
{
  o, sig = cl[k].sig, offset = cl[k].offset;
  while (off < offset) {
    o, cache[cacheptr + off] = sigacc | signbit;
    off++, sigacc = 0;
  }
  o, sighash += siginx[sig].hash;
  sigacc += 1LL << siginx[sig].shift;    /* siginx[sig].code = 1 */
}
```

This code is used in section 30.

32. ⟨Contribute a secondary item to the signature 32⟩ ≡

```
{
  if (o, nd[k].len == 0) continue;
  o, sig = cl[k].sig, offset = cl[k].offset;
  while (off < offset) {
    o, cache[cacheptr + off] = sigacc | signbit;
    off++, sigacc = 0;
  }
  o, sig += nd[k].color;
  o, sighash += siginx[sig].hash;
  sigacc += ((long long) siginx[sig].code) << siginx[sig].shift;
}
```

This code is used in section 30.

33. Here I use Algorithm 6.4D in the hash table, "open addressing with double hashing," because I want to refresh my brain's memory of that technique. (It conserves my computer's memory nicely, and avoids the primary clustering of simpler methods.)

#define *hashmask* $((1 \ll loghashsize) - 1)$

⟨ Do the hash lookup 33 ⟩ ≡

```
{
  register int h, hh, s, l;
  hh = (sighash ≫ (loghashsize − 1)) | 1;
  for (h = sighash & hashmask; ; h = (h + hh) & hashmask) {
    o, s = hash[h].sig;
    if (¬s) break;
    for (l = 0; ; l++) {
      if (oo, cache[s + l] ≠ cache[cacheptr + 1 + l]) break;
      if (cache[s + l] & signbit) continue;
      goto cache_hit;
    }
  }
  if (++hashcount ≥ hashsize) overflow(hashsize, "hashsize");
  o, hash[h].sig = cacheptr + 1;    /* cache[cacheptr] will hold a count */
  oldcacheptr = cacheptr, cacheptr += q + 1;
  memos++;
  o, hashloc[level] = h;
  hit = 0;
  goto cache_miss;
cache_hit: hit = 1 + h;
cache_miss: ;
}
```

This code is used in section 30.

34. The following code is executed after completing the computation on a level that has found at least one solution. The memo cache entry for that level is *hashloc*[*level*], and the ZDD node representing all those solutions is *znode*.

⟨ Cache the successful *znode* 34 ⟩ ≡

```
{
  register int h;
  o, h = hashloc[level];
  o, hash[h].zddref = znode;
  goodmemos++;
  ooo, cache[hash[h].sig − 1] = count − entrycount[level];
}
```

This code is used in section 36.

35. To celebrate a cache hit, we emulate all of the relevant previous computation at high speed.

⟨ Use previous ZDD data in place of this level's computation 35 ⟩ ≡

```
{
  register ullng c;
  o, znode = hash[hit − 1].zddref;
  if (vbose & show_hits) fprintf(stderr, "Hit[%x]␣(zdd="O"x,␣sols="O"lld)\n",
        hash[hit − 1].sig − 1, znode, cache[hash[hit − 1].sig − 1]);
  if (znode) {
    o, c = cache[hash[hit − 1].sig − 1];    /* this many new solutions are hereby found */
    count += c;
    if (count ≥ maxcount) timeout = 0;    /* exit as soon as possible */
    if (count < c) fprintf(stderr, "(the␣solution␣count␣has␣overflowed!)\n");
  }
  hits++;
  goto backdown;
}
```

This code is used in section 36.

36. 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 36⟩ ≡
  level = 0;
forward: nodes++;
  if (vbose & show_profile) profile[level]++;
  if (sanity_checking) sanity();
  ⟨Do special things if enough mems have accumulated 38⟩;
  ⟨Look for the current status in the memo cache 30⟩;
  if (hit) ⟨Use previous ZDD data in place of this level's computation 35⟩;
  o, entrycount[level] = count;
  znode = 0;
  ⟨Set best_itm to the best item for branching 45⟩;
  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 41⟩;
  if (o, cl[root].next ≡ root) ⟨Register a solution and goto recover 46⟩;
  o, savez[level] = znode;
  if (++level > maxl) {
    if (level ≥ max_level) {
      fprintf(stderr, "Too␣many␣levels!\n");
      exit(−4);
    }
    maxl = level;
  }
  goto forward;
backup: uncover(best_itm);
  if (znode) ⟨Cache the successful znode 34⟩;
backdown: if (level ≡ 0) goto done;
  level−−;
  oo, cur_node = choice[level], best_itm = nd[cur_node].itm;
  o, zsol = znode, znode = savez[level];
recover: ⟨Uncover all other items of cur_node 42⟩;
  if (zsol) ⟨Make a new ZDD node 47⟩;
  if (timeout ≡ 0) goto backup;
  oo, cur_node = choice[level] = nd[cur_node].down; goto advance;
```

This code is used in section 2.

37. ⟨ Global variables 3 ⟩ +≡

```
int level;   /* number of choices in current partial solution */
int choice[max_level];   /* the node chosen on each level */
int savez[max_level];   /* current znode on each level */
ullng profile[max_level];   /* number of search tree nodes on each level */
ullng entrycount[max_level];   /* count when a new level commences */
int hashloc[max_level];   /* hash location for cached computations at each level */
```

38. ⟨ Do special things if enough *mems* have accumulated 38 ⟩ ≡

```
if (delta ∧ (mems ≥ thresh)) {
  thresh += delta;
  if (vbose & show_full_state) print_state();
  else print_progress();
}
if (mems ≥ timeout) {
  fprintf(stderr, "TIMEOUT!\n");
  timeout = 0;
}
```

This code is used in section 36.

39. 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.

Program DLX2 improved its performance by not removing nodes from secondary items that have been purified. In DLX6 we don't want to do this, because we want the *len* field of secondary items to drop to zero when none of the active options use them. (Such items are irrelevant to the cached status.) But we can save part of the work, by decreasing *len* without altering *up* or *down*.

Furthermore, when the *len* field of a secondary item does drop to zero, we want to remove it from the list of "active" secondary items.

⟨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; ) {
      o, cc = nd[nn].itm;
      if (cc ≤ 0) {
        o, nn = nd[nn].up; continue;
      }
      if (nd[nn].color ≥ 0) {
        o, uu = nd[nn].up, dd = nd[nn].down;
        oo, nd[uu].down = dd, nd[dd].up = uu;
      }
      updates++;
      o, t = nd[cc].len − 1;
      o, nd[cc].len = t;
      if (t ≡ 0 ∧ cc ≥ second) {
        o, l = cl[cc].prev, r = cl[cc].next;
        oo, cl[l].next = r, cl[r].prev = l;
      }
      nn++;
    }
}
```

40. Here we uncover an item by processing its options from bottom to top, thus undoing in the reverse order of doing.

⟨ Subroutines 10 ⟩ +≡

```
void uncover(int c)
{
  register int cc, l, r, rr, nn, uu, dd, t;
  for (o, rr = nd[c].up; rr ≥ last_itm; o, rr = nd[rr].up)
    for (nn = rr − 1; nn ≠ rr; ) {
      o, cc = nd[nn].itm;
      if (cc ≤ 0) {
        o, nn = nd[nn].down; continue;
      }
      if (nd[nn].color ≥ 0) {
        o, uu = nd[nn].up, dd = nd[nn].down;
        oo, nd[uu].down = nd[dd].up = nn;
      }
      o, t = nd[cc].len + 1;
      o, nd[cc].len = t;
      if (t ≡ 1 ∧ cc ≥ second) {
        o, l = cl[cc].prev, r = cl[cc].next;
        oo, cl[l].next = cl[r].prev = cc;
      }
      nn−−;
    }
  o, l = cl[c].prev, r = cl[c].next;
  oo, cl[l].next = cl[r].prev = c;
}
```

41. A subtle point arises here: When *best_itm* was covered, or when a previous item in the option for *cur_node* was covered or purified, we may have removed all of the remaining nodes for some secondary item, and deleted that item from the list of active secondaries. We don't want to cover or purify it in such cases, since that would delete it twice.

⟨ Cover all other items of *cur_node* 41 ⟩ ≡

```
for (pp = cur_node + 1; pp ≠ cur_node; ) {
  o, cc = nd[pp].itm;
  if (cc ≤ 0) o, pp = nd[pp].up;
  else {
    if (cc < second ∨ (o, nd[cc].len)) {
      if (¬nd[pp].color) cover(cc);
      else if (nd[pp].color > 0) purify(pp);
    }
    pp++;
  }
}
```

This code is used in section 36.

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

And the logic above requires another subtle point: We must not allow *purify*(*pp*) to change the length of *nd*[*pp*].*itm* from nonzero to zero. (Otherwise we couldn't unpurify it.)

⟨ Uncover all other items of *cur_node* 42 ⟩ ≡

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

This code is used in section 36.

43. 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.

The purified item's list stays intact, so that we can unpurify it later. But we adjust the *len*, so that only active options are counted.

⟨ Subroutines 10 ⟩ +≡

```
void purify(int p)
{
  register int cc, rr, nn, uu, dd, t, x, tt;
  o, cc = nd[p].itm, x = nd[p].color;
  o, nd[cc].color = x;
  o, tt = nd[cc].len;
  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) {
      tt--;
      for (nn = rr + 1; nn ≠ rr; ) {
        o, cc = nd[nn].itm;
        if (cc ≤ 0) {
          o, nn = nd[nn].up; continue;
        }
        if (nd[nn].color ≥ 0) {
          o, uu = nd[nn].up, dd = nd[nn].down;
          oo, nd[uu].down = dd, nd[dd].up = uu;
        }
        updates++;
        o, t = nd[cc].len − 1;
        o, nd[cc].len = t;
        if (t ≡ 0 ∧ cc ≥ second) {
          register int l, r;
          o, l = cl[cc].prev, r = cl[cc].next;
          oo, cl[l].next = r, cl[r].prev = l;
        }
        nn++;
      }
    } else cleansings++, o, nd[rr].color = −1;
  }
  if (tt > 0) o, cc = nd[p].itm, nd[cc].len = tt;    /* no mem for fetching cc again */
  else {
    register int l, r;
    o, cc = nd[p].itm, nd[cc].len = −1;    /* store a signal for unpurification */
    o, l = cl[cc].prev, r = cl[cc].next;
    oo, cl[l].next = r, cl[r].prev = l;
  }
}
```

44. 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, tt;
  oo, cc = nd[p].itm, x = nd[p].color, nd[cc].color = 0;
  o, tt = nd[cc].len;
  if (tt < 0) {
    register int l, r;
    tt = 0;      /* tt was artificially negative, to give a signal */
    o, l = cl[cc].prev, r = cl[cc].next;
    oo, cl[l].next = cl[r].prev = cc;
  }
  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 {
      tt++;
      for (nn = rr − 1; nn ≠ rr; ) {
        o, cc = nd[nn].itm;
        if (cc ≤ 0) {
          o, nn = nd[nn].down; continue;
        }
        if (nd[nn].color ≥ 0) {
          o, uu = nd[nn].up, dd = nd[nn].down;
          oo, nd[uu].down = nd[dd].up = nn;
        }
        o, t = nd[cc].len + 1;
        o, nd[cc].len = t;
        if (t ≡ 1 ∧ cc ≥ second) {
          register int l, r;
          o, l = cl[cc].prev, r = cl[cc].next;
          oo, cl[l].next = cl[r].prev = cc;
        }
        nn−−;
      }
    }
  }
  o, cc = nd[p].itm, nd[cc].len = tt;
}
```

45. 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 45 ⟩ ≡

```
t = max_nodes;
if ((vbose & show_details) ∧ level < show_choices_max ∧ level ≥ maxl − show_choices_gap) {
  fprintf(stderr, "Level␣"O"d:", level);
  if (vbose & show_hits) fprintf(stderr, "["O"x]", oldcacheptr);
}
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_secondary_details) ∧ level < show_choices_max ∧ level ≥ maxl − show_choices_gap) {
  fprintf(stderr, ";");
  for (k = cl[last_itm].next; k ≠ last_itm; k = cl[k].next)
    fprintf(stderr, "␣"O".8s("O"d)", cl[k].name, nd[k].len);
}
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 36.

46. ⟨ Register a solution and **goto** *recover* 46 ⟩ ≡

```
{
  nodes++;     /* a solution is a special node, see 7.2.2–(4) */
  hits++;     /* Algorithm 7.2.2.1Z treats this as a hit at level + 1 */
  if (vbose & show_hits) fprintf(stderr, "Solution\n");
  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);
  }
  zsol = 1;     /* the goal node of a ZDD */
  count++;
  if (count ≥ maxcount) timeout = 0;     /* exit as soon as possible */
  goto recover;
}
```

This code is used in section 36.

47. ⟨ Make a new ZDD node 47 ⟩ ≡

```
{
  if (spacing) printf(""O"x:␣(~"O"d?"O"x:"O"x)\n", zddnodes, optionno, znode, zsol);
  znode = zddnodes++;
  if (¬zddnodes) {     /* wow */
    fprintf(stderr, "Too␣many␣ZDD␣nodes␣(4294967296)!\n");
    exit(−232);
  }
  if (zddnodes > maxzdd) timeout = 0;     /* exit as soon as possible */
}
```

This code is used in section 36.

48. ⟨ 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␣hits,␣"O"lld␣mems,", count, hits, mems);
  fprintf(stderr, "␣and␣max␣level␣"O"d␣so␣far.\n", maxl);
}
```

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

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

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

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

⟨ Subroutines 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, "O"lld hits,", mems, count, hits);
  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);
}
```

50. ⟨ Print the profile 50 ⟩ ≡

```
{
  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.

51. Index.

DLX6