November 24, 2020 at 13:24

1. Intro. This program is an "XCC solver" that I'm writing as an experiment in the use of so-called sparse-set data structures instead of the dancing links structures I've played with for thirty years. I plan to write it as if I live on a planet where the sparse-set ideas are well known, but doubly linked links are almost unheard-of. As I begin, I know that the similar program SSXC (which solves the special case of ordinary exact cover problems) works fine.

I shall accept the DLX input format used in the previous solvers, without change, so that a fair comparison can be made. (See the program DLX2 for definitions. Much of the code from that program is used to parse the input for this one.)

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>
  typedef unsigned int uint;   /* a convenient abbreviation */
  typedef unsigned long long ullng;   /* ditto */
  ⟨Type definitions 7⟩;
  ⟨Global variables 3⟩;
  ⟨Subroutines 10⟩;
  main(int argc, char *argv[])
  {
    register int cc, i, j, k, p, pp, q, r, s, t, cur_choice, cur_node, best_itm;
    ⟨Process the command line 4⟩;
    ⟨Input the item names 14⟩;
    ⟨Input the options 16⟩;
    if (vbose & show_basics) ⟨Report the successful completion of the input phase 22⟩;
    if (vbose & show_tots) ⟨Report the item totals 23⟩;
    imems = mems, mems = 0;
    ⟨Solve the problem 24⟩;
  done: if (sanity_checking) sanity();
    if (vbose & show_tots) ⟨Report the item totals 23⟩;
    if (vbose & show_profile) ⟨Print the profile 40⟩;
    if (vbose & show_max_deg)
      fprintf(stderr, "The␣maximum␣branching␣degree␣was␣"O"d.\n", maxdeg);
    if (vbose & show_basics) {
      fprintf(stderr, "Altogether␣"O"llu␣solution"O"s,␣"O"llu+"O"llu␣mems,", count,
          count ≡ 1 ? "" : "s", imems, mems);
      bytes = (itemlength + setlength) * sizeof(int) + 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);
- '`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 */
#define show_max_deg 2048   /* vbose code for reporting maximum branching degree */
```

⟨Global variables 3⟩ ≡

```
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;   /* 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 */
int maxdeg;   /* the largest branching degree seen so far */
```

See also sections 8 and 25.

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 '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);
}
```

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. Sparse-set data structures were introduced by Preston Briggs and Linda Torczon [*ACM Letters on Programming Languages and Systems* **2** (1993), 59–69], who realized that exercise 2.12 in Aho, Hopcroft, and Ullman's classic text *The Design and Analysis of Computer Algorithms* (Addison–Wesley, 1974) was much more than just a slick trick to avoid initializing an array. (Indeed, *TAOCP* exercise 2.2.6–24 calls it the "sparse array trick.")

The basic idea is amazingly simple, when specialized to the situations that we need to deal with: We can represent a subset S of the universe $U = \{x_0, x_1, \ldots, x_{n-1}\}$ by maintaining two n-element arrays p and q, each of which is a permutation of $\{0, 1, \ldots, n-1\}$, together with an integer s in the range $0 \le s \le n$. In fact, p is the *inverse* of q; and s is the number of elements of S. The current value of the set S is then simply $\{x_{p_0}, \ldots, x_{p_{s-1}}\}$. (Notice that every s-element can be represented in $s!\,(n-s)!$ ways.)

It's easy to test if $x_k \in S$, because that's true if and only if $q_k < s$. It's easy to insert a new element x_k into S: Swap indices so that $p_s = k$, $q_k = s$, then increase s by 1. It's easy to delete an element x_k that belongs to S: Decrease s by 1, then swap indices so that $p_s = k$ and $q_k = s$. And so on.

Briggs and Torczon were interested in applications where s begins at zero and tends to remain small. In such cases, p and q need not be permutations: The values of p_s, p_{s+1}, $\ldots$, p_{n-1} can be garbage, and the values of q_k need be defined only when $x_k \in S$. (Such situations correspond to Aho, Hopcroft, and Ullman, who started with an array full of garbage and used a sparse-set structure to remember the set of nongarbage cells.) Our applications are different: Each set begins equal to its intended universe, and gradually shrinks. In such cases, we might as well maintain inverse permutations. The basic operations go faster when we know in advance that we aren't inserting an element that's already present (nor deleting an element that isn't).

Many variations are possible. For example, p could be a permutation of $\{x_0, x_1, \ldots, x_{n-1}\}$ instead of permutation of $\{0, 1, \ldots, n-1\}$. The arrays that play the role of q in the following routines don't have indices that are consecutive; they live inside of other structures.

7. This program has an array called *item*, with one entry for each item. The value of $item[k]$ is an index x into a much larger array called *set*. The set of all options that involve the kth item appears in that array beginning at $set[x]$; and it continues for s consecutive entries, where $s = size(x)$ is an abbreviation for $set[x-1]$. If $item[k] = x$, we maintain the relation $pos(x) = k$, where $pos(x)$ is an abbreviation for $set[x-2]$. Thus *item* plays the role of array p, in a sparse-set data structure for the set of all currently active items; and *pos* plays the role of q.

Suppose the kth item x currently appears in s options. Those options are indices into *nd*, which is an array of "nodes." Each node has three fields: *itm*, *loc*, and *clr*. If $x \le q < x + s$, let $y = set[q]$. This is essentially a pointer to a node, and we have $nd[y].itm = x$, $nd[y].loc = q$. In other words, the sequential list of s elements that begins at $x = item[k]$ in the *set* array is the sparse-set representation of the currently active options that contain the kth item. The *clr* field contains x's color for this option, or -1 if x has been purified to be compatible with this option. The *itm* fields remain constant, once we've initialized everything, but the *loc* and *clr* fields will change.

The given options are stored sequentially in the *nd* array, with one node per item, separated by "spacer" nodes. If y is the spacer node following an option with t items, we have $nd[y].itm = -t$. If y is the spacer node *preceding* an option with t items, we have $nd[y].loc = t$.

This probably sounds confusing, until you can see some code. Meanwhile, let's take note of the invariant relations that hold whenever k, q, x, and y have appropriate values:

$$pos(item[k]) = k; \quad nd[set[q]].loc = q; \quad item[pos(x)] = x; \quad set[nd[y].loc] = y.$$

(These are the analogs of the invariant relations $p[q[k]] = q[p[k]] = k$ in the simple sparse-set scheme that we started with.)

The *set* array contains also the item names, as well as "purified colors."

We count one mem for a simultaneous access to the *itm* and *loc* fields of a node. Each actually has a "spare" fourth field, *spr*, inserted solely to enforce alignment to 16-byte boundaries. (Some modification of this program might perhaps have a use for *spr*?)

```
#define size(x) set[(x) − 1]    /* number of active options of the kth item, x */
#define pos(x) set[(x) − 2]    /* where that item is found in the item array */
#define lname(x) set[(x) − 4]    /* the first four bytes of x's name */
#define rname(x) set[(x) − 3]    /* (the last four bytes of x's name */
#define color(x) set[(x) − 5]    /* the color of x, if purified (secondary x only) */
```

⟨ Type definitions 7 ⟩ ≡

```
typedef struct node_struct {
  int itm;    /* the item x corresponding to this node */
  int loc;    /* where this node resides in x's active set */
  int clr;    /* color associated with item x in this option, if any */
  int spr;    /* a spare field inserted only to maintain 16-byte alignment */
} node;
```

See also section 9.

This code is used in section 2.

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 */
int item[max_cols];    /* 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 */
int set[max_nodes + 4 * max_cols];    /* the sets of active options for active items */
int itemlength;    /* number of elements used in item */
int setlength;    /* number of elements used in set */
int active;    /* current number of active items */
```

9. We're going to store string data (an item name) in the midst of the integer array *set*. So we've got to do some type coercion using low-level C-ness.

⟨ Type definitions 7 ⟩ +≡

```
typedef struct {
  int l, r;
} twoints;
typedef union {
  char str[8];    /* eight one-byte characters */
  twoints lr;     /* two four-byte integers */
} stringbuf;
stringbuf namebuf;
```

10. ⟨ Subroutines 10 ⟩ ≡

```
void print_item_name(int k, FILE *stream)
{
  namebuf.lr.l = lname(k), namebuf.lr.r = rname(k);
  fprintf(stream, "␣%.8s", namebuf.str);
}
```

See also sections 11, 12, 13, 27, 29, 33, 34, 38, and 39.

This code is used in section 2.

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 list.

⟨ Subroutines 10 ⟩ +≡

```
void print_option(int p, FILE *stream)
{
  register int k, q, x;
  x = nd[p].itm;
  if (p ≥ last_node ∨ x ≤ 0) {
    fprintf(stderr, "Illegal␣option␣%d!\n", p);
    return;
  }
  for (q = p; ; ) {
    print_item_name(x, stream);
    if (nd[q].clr) fprintf(stream, ":%c", nd[q].clr > 0 ? nd[q].clr : color(x));
    q++;
    x = nd[q].itm;
    if (x < 0) q += x, x = nd[q].itm;
    if (q ≡ p) break;
  }
  k = nd[q].loc;
  fprintf(stream, "␣(%d␣of␣%d)\n", k − x + 1, size(x));
}
void prow(int p)
{
  print_option(p, stderr);
}
```

12. 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 < 4 ∨ c ≥ setlength ∨ pos(c) < 0 ∨ pos(c) ≥ itemlength ∨ item[pos(c)] ≠ c) {
    fprintf(stderr, "Illegal item "O"d!\n", c);
    return;
  }
  fprintf(stderr, "Item");
  print_item_name(c, stderr);
  if (pos(c) < second)
    fprintf(stderr, " ("O"d of "O"d), length "O"d:\n", pos(c) + 1, active, size(c));
  else if (color(c))
    fprintf(stderr, " (secondary "O"d:"O"c), length "O"d:\n", pos(c) + 1, color(c), size(c));
  else fprintf(stderr, " (secondary "O"d), length "O"d:\n", pos(c) + 1, size(c));
  for (p = c; p < c + size(c); p++) prow(set[p]);
}
```

13. 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, x, i, l, r;
  for (k = 0; k < itemlength; k++) {
    x = item[k];
    if (pos(x) ≠ k) {
      fprintf(stderr, "Bad pos field of item");
      print_item_name(x, stderr);
      fprintf(stderr, " ("O"d,"O"d)!\n", k, x);
    }
  }
  for (i = 0; i < last_node; i++) {
    l = nd[i].itm, r = nd[i].loc;
    if (l ≤ 0) {
      if (nd[i + r + 1].itm ≠ −r) fprintf(stderr, "Bad spacer in nodes "O"d, "O"d!\n", i, i + r + 1);
    } else {
      if (l > r) fprintf(stderr, "itm>loc in node "O"d!\n", i);
      else if (set[r] ≠ i) {
        fprintf(stderr, "Bad loc field for option "O"d of item", r − l + 1);
        print_item_name(l, stderr);
        fprintf(stderr, " in node "O"d!\n", i);
      }
    }
  }
}
```

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.

We use only four entries of *set* per item while reading the item-name line.

```
#define panic(m)
          { fprintf(stderr, ""O"s!\n"O"d:␣"O".99s\n", m, p, buf); exit(−666); }
```

⟨ Input the item names 14 ⟩ ≡

```
  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]; ) {
    o, namebuf.lr.l = namebuf.lr.r = 0;
    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, namebuf.str[j] = buf[p + j];
    }
    if (j ≡ 8 ∧ ¬isspace(buf[p + j])) panic("Item␣name␣too␣long");
    oo, lname(last_itm ≪ 2) = namebuf.lr.l, rname(last_itm ≪ 2) = namebuf.lr.r;
    ⟨ Check for duplicate item name 15 ⟩;
    last_itm++;
    if (last_itm > max_cols) panic("Too␣many␣items");
    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;
```

This code is used in section 2.

15. ⟨ Check for duplicate item name 15 ⟩ ≡

```
  for (k = last_itm − 1; k; k−−) {
    if (o, lname(k ≪ 2) ≠ namebuf.lr.l) continue;
    if (rname(k ≪ 2) ≡ namebuf.lr.r) break;
  }
  if (k) panic("Duplicate␣item␣name");
```

This code is used in section 14.

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

⟨ Input the options 16 ⟩ ≡

```
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]; ) {
    o, namebuf.lr.l = namebuf.lr.r = 0;
    for (j = 0; j < 8 ∧ (o, ¬isspace(buf[p + j])) ∧ buf[p + j] ≠ ':'; j++) o, namebuf.str[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");
    ⟨ Create a node for the item named in buf[p] 17 ⟩;
    if (buf[p + j] ≠ ':') o, nd[last_node].clr = 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].clr = (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 18 ⟩;
      last_node−−;
    }
  } else {
    o, nd[i].loc = last_node − i;      /* complete the previous spacer */
    last_node++;      /* create the next spacer */
    if (last_node ≡ max_nodes) panic("Too␣many␣nodes");
    options++;
    o, nd[last_node].itm = i + 1 − last_node;
    nd[last_node].spr = options;      /* option number, for debugging only */
  }
}
⟨ Initialize item 19 ⟩;
⟨ Expand set 20 ⟩;
⟨ Adjust nb 21 ⟩;
```

This code is used in section 2.

17. We temporarily use *pos* to recognize duplicate items in an option.

⟨ Create a node for the item named in *buf* [*p*] 17 ⟩ ≡

```
for (k = (last_itm − 1) ≪ 2; k; k −= 4) {
   if (o, lname(k) ≠ namebuf.lr.l) continue;
   if (rname(k) ≡ namebuf.lr.r) break;
}
if (¬k) panic("Unknown␣item␣name");
if (o, pos(k) > i) panic("Duplicate␣item␣name␣in␣this␣option");
last_node++;
if (last_node ≡ max_nodes) panic("Too␣many␣nodes");
o, t = size(k);      /* how many previous options have used this item? */
o, nd[last_node].itm = k ≫ 2, nd[last_node].loc = t;
if ((k ≫ 2) < second) pp = 1;
o, size(k) = t + 1, pos(k) = last_node;
```

This code is used in section 16.

18. ⟨ Remove *last_node* from its item list 18 ⟩ ≡

```
o, k = nd[last_node].itm ≪ 2;
oo, size(k)−−, pos(k) = i − 1;
```

This code is used in section 16.

19. Each primary item occupies four special positions in *set* (namely for *lname*, *rname*, *pos*, and *size*). Each secondary item occupies five (because it also has *color*).

⟨ Initialize *item* 19 ⟩ ≡

```
itemlength = last_itm − 1;
active = second = second − 1;
for (k = 0, j = 4; k < itemlength; k++) oo, item[k] = j, j += 4 + size((k + 1) ≪ 2) + (k + 1 ≥ second);
setlength = j − 4 − (k ≥ second);
```

This code is used in section 16.

20. Going from high to low, we now move the item names and sizes to their final positions (leaving room for the pointers into *nb*).

⟨ Expand *set* 20 ⟩ ≡

```
for ( ; k; k−−) {
   o, j = item[k − 1];
   oo, size(j) = size(k ≪ 2);
   o, pos(j) = k − 1;
   oo, rname(j) = rname(k ≪ 2), lname(j) = lname(k ≪ 2);
   if (k ≥ second) o, color(j) = 0;
}
```

This code is used in section 16.

21. ⟨ Adjust *nb* 21 ⟩ ≡

```
for (k = 1; k < last_node; k++) {
   if (o, nd[k].itm < 0) continue;      /* skip over a spacer */
   o, j = item[nd[k].itm − 1];
   i = j + nd[k].loc;      /* no mem charged because we just read nd[k].itm */
   o, nd[k].itm = j, nd[k].loc = i;
   o, set[i] = k;
}
```

This code is used in section 16.

22. The "number of entries" includes spacers (because DLX2 includes spacers in its reports). If you want to know the sum of the option lengths, just subtract the number of options.

⟨ Report the successful completion of the input phase 22 ⟩ ≡

```
fprintf(stderr, "("O"lld␣options,␣"O"d+"O"d␣items,␣"O"d␣entries␣successfully␣read)\n",
      options, second, last_itm − second − 1, last_node);
```

This code is used in section 2.

23. 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.

[Caution: They will probably appear in a different order than before!]

⟨ Report the item totals 23 ⟩ ≡

```
{
  fprintf(stderr, "Item␣totals:");
  for (k = 0; k < itemlength; k++) {
    if (k ≡ second) fprintf(stderr, "␣|");
    fprintf(stderr, "␣"O"d", size(item[k]));
  }
  fprintf(stderr, "\n");
}
```

This code is used in section 2.

24. The dancing. Our strategy for generating all exact covers will be to repeatedly choose always an item that appears to be hardest to cover, namely the item with smallest set, 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 sparse-set representations remember enough of what was done so that we can undo it later.

The basic operation is "covering an item." This means removing it from the set of items needing to be covered, and "hiding" its options: removing them from the sets of the other items they contain.

```
⟨Solve the problem 24⟩ ≡
  level = 0;
forward: nodes++;
  if (vbose & show_profile) profile[level]++;
  if (sanity_checking) sanity();
  ⟨Do special things if enough mems have accumulated 26⟩;
  ⟨Set best_itm to the best item for branching 35⟩;
  if (t ≡ 0) goto donewithlevel;
  cover(best_itm);
  cur_choice = best_itm;
  oo, cur_node = choice[level] = set[best_itm];
  goto tryit;
advance: if (o, cur_choice ≥ best_itm + size(best_itm)) goto backup;
  oo, cur_node = choice[level] = set[cur_choice];
tryit: 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 31⟩;
  if (active ≡ 0) ⟨Visit a solution and goto recover 36⟩;
  if (++level > maxl) {
    if (level ≥ max_level) {
      fprintf(stderr, "Too␣many␣levels!\n");
      exit(−4);
    }
    maxl = level;
  }
  goto forward;
backup: uncover(best_itm);
donewithlevel: if (level ≡ 0) goto done;
  level−−;
  oo, cur_node = choice[level], best_itm = nd[cur_node].itm, cur_choice = nd[cur_node].loc;
recover: ⟨Uncover all other items of cur_node 32⟩;
  cur_choice++; goto advance;
```

This code is used in section 2.

25.
```
⟨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 */
```

26. ⟨Do special things if enough *mems* have accumulated 26⟩ ≡

```
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 24.

27. 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.

⟨Subroutines 10⟩ +≡

```
void cover(int c)
{
  register int k, a, cc, s, rr, ss, nn, tt, uu, vv, nnp;
  o, k = pos(c);
  if (k < second) {   /* update the active list, if c is primary */
    a = active − 1, active = a;
    o, cc = item[a];
    oo, item[a] = c, item[k] = cc;
    oo, pos(cc) = k, pos(c) = a;
    updates++;
  }
  for (o, rr = c, s = c + size(c); rr < s; rr++) {
    o, tt = set[rr];
    ⟨Remove the option tt from the other sets it's in 28⟩;
  }
}
```

28. ⟨Remove the option *tt* from the other sets it's in 28⟩ ≡

```
{
  for (nn = tt + 1; nn ≠ tt; )
    if (o, nd[nn].clr ≥ 0) {
      o, uu = nd[nn].itm, vv = nd[nn].loc;
      if (uu < 0) { nn += uu; continue; }
      o, ss = size(uu) − 1;
      o, nnp = set[uu + ss];
      o, size(uu) = ss;
      oo, set[uu + ss] = nn, set[vv] = nnp;
      oo, nd[nn].loc = uu + ss, nd[nnp].loc = vv;
      nn++;
      updates++;
    } else nn++;
}
```

This code is used in sections 27 and 33.

29. To undo the *cover* operation, we need only increase the set size, because the previously deleted element is in position to be seamlessly reinstated. (Inactive elements are never moved.) We need not swap that element back to its former position.

```
⟨ Subroutines 10 ⟩ +≡
  void uncover(int c)
  {
    register int k, cc, s, rr, ss, nn, tt, uu;
    for (o, rr = c, s = c + size(c); rr < s; rr++) {
      o, tt = set[rr];
      ⟨ Unremove the option tt from the other sets it was in 30 ⟩;
    }
    o, k = pos(c);
    if (k < second) active++;
  }
```

30.
```
⟨ Unremove the option tt from the other sets it was in 30 ⟩ ≡
  {
    for (nn = tt + 1; nn ≠ tt; )
      if (o, nd[nn].clr ≥ 0) {
        o, uu = nd[nn].itm;
        if (uu < 0) { nn += uu; continue; }
        o, ss = size(uu) + 1;
        o, size(uu) = ss;
        nn++;
      } else nn++;
  }
```

This code is used in sections 29 and 34.

31.
```
⟨ Cover all other items of cur_node 31 ⟩ ≡
  for (pp = cur_node + 1; pp ≠ cur_node; ) {
    o, cc = nd[pp].itm;
    if (cc < 0) pp += cc;
    else {
      if (o, nd[pp].clr ≡ 0) cover(cc);
      else if (nd[pp].clr > 0) purify(pp);
      pp++;
    }
  }
```

This code is used in section 24.

32. Covering and uncovering both traverse options to the right. That's okay—although it takes a bit of thought to verify that all sets are restored correctly. (An item that has lost k options from its set will regain those k options, but not necessarily in the same order.)

But we do need to go left here, *not* right.

⟨ Uncover all other items of *cur_node* 32 ⟩ ≡

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

This code is used in section 24.

33. 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.

(At first I thought it would be a good idea to rearrange the set entries, putting first the correctly colored options. There's an appealing way to do that with a minimum number of swaps. However, I soon realized that there's no real reason to change the order. The size of this set doesn't change; and we won't be considering it again until it's unpurified.)

When *purify* is called, *nd*[*p*] is part of an option that has been deleted from all sets. The secondary item *nd*[*p*].*itm* is being purified to have color *nd*[*p*].*clr*.

⟨ Subroutines 10 ⟩ +≡

```
void purify(int p)
{
   register int c, x, tt, rr, s, ss, nn, uu, vv, nnp;
   o, c = nd[p].itm;
   o, x = nd[p].clr;
   color(c) = x;     /* no mem charged, because this is needed only in printout */
   cleansings++;
   for (o, rr = c, s = c + size(c); rr < s; rr++) {
      o, tt = set[rr];
      if (o, nd[tt].clr ≠ x) ⟨ Remove the option tt from the other sets it's in 28 ⟩
      else o, cleansings++, nd[tt].clr = −1;
   }
}
```

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

⟨ Subroutines 10 ⟩ +≡

```
void unpurify(int p)
{
  register int c, x, tt, rr, s, ss, nn, uu;
  o, c = nd[p].itm;
  o, x = nd[p].clr;
  color(c) = 0;      /* no mem charged, because this is needed only in printout */
  for (o, rr = c, s = c + size(c); rr < s; rr++) {      /* going to the right is okay again */
    o, tt = set[rr];
    if (o, nd[tt].clr ≥ 0) ⟨ Unremove the option tt from the other sets it was in 30 ⟩
    else o, nd[tt].clr = x;
  }
}
```

35. 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.

(This program explores the search space in a different order from DLX2, because the ordering of items in the active list is no longer fixed. Thus ties are broken in a different way.)

⟨ Set *best_itm* to the best item for branching 35 ⟩ ≡

```
t = max_nodes;
if ((vbose & show_details) ∧ level < show_choices_max ∧ level ≥ maxl − show_choices_gap)
  fprintf(stderr, "Level␣"O"d:", level);
for (k = 0; t ∧ (k < active); k++) {
  oo, s = size(item[k]);
  if ((vbose & show_details) ∧ level < show_choices_max ∧ level ≥ maxl − show_choices_gap) {
    print_item_name(item[k], stderr);
    fprintf(stderr, "("O"d)", s);
  }
  if (s < t) best_itm = item[k], t = s;
}
if ((vbose & show_details) ∧ level < show_choices_max ∧ level ≥ maxl − show_choices_gap) {
  fprintf(stderr, "␣branching␣on");
  print_item_name(best_itm, stderr);
  fprintf(stderr, "("O"d)\n", t);
}
if (t > maxdeg) maxdeg = t;
if (shape_file) {
  fprintf(shape_file, ""O"d", t);
  print_item_name(best_itm, shape_file);
  fprintf(shape_file, "\n");
  fflush(shape_file);
}
```

This code is used in section 24.

36. ⟨ Visit a solution and **goto** *recover* 36 ⟩ ≡

```
{
  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 37 ⟩;
}
```

This code is used in section 24.

37. ⟨ Record solution and **goto** *recover* 37 ⟩ ≡

```
{
  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 36.

38. ⟨ 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);
}
```

39. 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,", mems, count);
    for (f = 0.0, fd = 1.0, l = 0; l < level; l++) {
      c = nd[choice[l]].itm, d = size(c), k = nd[choice[l]].loc − c + 1;
      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);
  }
```

40. ⟨Print the profile 40⟩ ≡

```
  {
    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.

41. Index.

SSXCC