(See https://cs.stanford.edu/~knuth/programs.html for date.)

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 that 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.

The difference between this program and SSXCC, on which it's based, is that I use binary branching '$i = o$' versus '$i \neq o$' at each step, where i is an item and o is an option, while SSXCC does d-way branching on all d options that currently cover item i. The reason for binary branching is that I plan to extend this program in various ways, in order to experiment with several dynamic branching heuristics that I've seen in the literature; those heuristics were designed with binary branching in mind.

I suggest that you read SSXCC first.

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" 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, or 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.)

```
#define o mems++ /* count one mem */
#define oo mems += 2 /* count two mems */
#define ooo mems += 3 /* count three mems */
#define subroutine_overhead mems += 4
#define O "%" /* used for percent signs in format strings */
#define mod % /* used for percent signs denoting remainder in C */

#define max_stage 500 /* at most this many options in a solution */
#define max_level 32000 /* at most this many levels in the search tree */
#define max_cols 100000 /* at most this many items */
#define max_nodes 10000000 /* at most this many nonzero elements in the matrix */
#define savesize 10000000 /* at most this many entries on savestack */
#define bufsize (9 * max_cols + 3) /* a buffer big enough to hold all item names */

#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_record_weights 16 /* vbose code for first time a weight appears */
#define show_weight_bumps 32 /* vbose code to show new weights */
#define show_final_weights 64 /* vbose code to display weights at the end */
#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 */
#define show_warnings 1024 /* vbose code for reporting options without primaries */
#define show_max_deg 2048 /* vbose code for reporting maximum branching degree */
```

2. Here is the overall structure:

```
#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 8⟩;
  ⟨Global variables 3⟩;
  ⟨Subroutines 6⟩;
  main(int argc, char *argv[])
  {
    register int c, 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 15⟩;
    ⟨Input the options 17⟩;
    if (vbose & show_basics) ⟨Report the successful completion of the input phase 24⟩;
    if (vbose & show_tots) ⟨Report the item totals 25⟩;
    imems = mems, mems = 0;
    if (baditem) ⟨Report an uncoverable item 23⟩
    else {
      if (randomizing) ⟨Randomize the item list 26⟩;
      ⟨Solve the problem 29⟩;
    }
  done: if (vbose & show_profile) ⟨Print the profile 49⟩;
    if (vbose & show_final_weights) {
      fprintf(stderr, "Final␣weights:\n");
      print_weights();
    }
    if (vbose & show_max_deg) fprintf(stderr, "The␣maximum␣best_itm␣size␣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) + 2 * maxl * sizeof(int) + maxsaveptr * sizeof (twoints);
      fprintf(stderr, "␣"O"llu␣updates,␣"O"llu␣bytes,␣"O"llu␣nodes,", updates, bytes, nodes);
      fprintf(stderr, "␣ccost␣"O"lld%%.\n", mems ? (200 * cmems + mems)/(2 * mems) : 0);
    }
    if (sanity_checking) fprintf(stderr, "sanity_checking␣was␣on!\n");
    ⟨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 randomize the initial list of items (thus providing some variety, although the solutions are by no means uniformly random);
- '`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 (default 10);
- '`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);
- '`w`⟨ float ⟩' is the initial increment *dw* added to an item's weight (default 1.0);
- '`W`⟨ float ⟩' is the factor by which *dw* changes dynamically (default 1.0);
- '`S`⟨ filename ⟩' to output a "shape file" that encodes the search tree.

⟨ 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 = 10;   /* above this level, state reports stop */
int maxl;   /* maximum level actually reached */
int maxs;   /* maximum stage actually reached */
int maxsaveptr;   /* maximum size of savestack */
char buf[bufsize];   /* input buffer */
ullng count;   /* solutions found so far */
ullng options;   /* options seen so far */
ullng imems, mems, tmems, cmems;   /* mem counts */
ullng updates;   /* update 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 */
float w0 = 1.0, dw = 1.0, dwfactor = 1.0;   /* initial weight, increment, and growth */
float maxwt = 1.0;   /* largest weight seen so far */
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 9, 28, and 30.

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 'w': k |= (sscanf(argv[j] + 1, ""O"f", &dw) - 1); break;
  case 'W': k |= (sscanf(argv[j] + 1, ""O"f", &dwfactor) - 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>] [w<f>] [W<f>] [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. Here's a subroutine that I hope is never invoked (except maybe when I'm debugging).

⟨ Subroutines 6 ⟩ ≡

```
void confusion(char *m)
{
  fprintf(stderr, ""O"s!\n", m);
}
```

See also sections 11, 12, 13, 14, 34, 40, 46, 47, and 48.

This code is used in section 2.

7. 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 subset 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}, …, 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 the treatment by 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 a 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.

8. 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.

A primary item x also has a *wt* field, $set[x-5]$, initially 1. The weight is increased by *dw* whenever we backtrack because x cannot be covered. (Weights aren't actually *used* in the present program; that will come in extensions to be written later. But it will be convenient to have space ready for them in our data structures, so that those extensions will be easy to write.)

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 $nd[y].clr$ contains x's color for this option. The *itm* and *clr* fields remain constant, once we've initialized everything, but the *loc* 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.

We count one mem for a simultaneous access to the *itm* and *loc* fields of a node. Each node 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].i   /* number of active options of the kth item, x */
#define pos(x) set[(x) − 2].i   /* where that item is found in the item array, k */
#define lname(x) set[(x) − 4].i   /* the first four bytes of x's name */
#define rname(x) set[(x) − 3].i   /* the last four bytes of x's name */
#define wt(x) set[(x) − 5].f   /* the current floating-point "weight" of x */
#define primextra 5   /* this many extra entries of set for each primary item */
#define secondextra 4   /* and this many for each secondary item */
#define maxextra 5   /* maximum of primextra and secondextra */
```

⟨Type definitions 8⟩ ≡

```
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;
typedef union {
  int i;   /* an integer (32 bits) */
  float f;   /* a floating point value (fits in 4 bytes) */
} tetrabyte;
```

See also section 10.

This code is used in section 2.

9. ⟨ 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;   /* items seen so far during input, plus 1 */
tetrabyte set[max_nodes + maxextra * max_cols];   /* 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 */
int baditem;   /* an item with no options, plus 1 */
int osecond;   /* setting of second just after initial input */
int force[max_cols];   /* stack of items known to have size 1 */
int forced;   /* the number of items on that stack */
```

10. We're going to store string data (an item's 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 8 ⟩ +≡

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

11. ⟨ Subroutines 6 ⟩ +≡

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

12. 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 6 ⟩ +≡

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

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

⟨ Subroutines 6 ⟩ +≡

```
void print_itm(int c)
{
  register int p;
  if (c < primextra ∨ 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 (c < second) fprintf(stderr, "␣("O"d␣of␣"O"d),␣length␣"O"d,␣weight␣"O".1f:\n",
        pos(c) + 1, active, size(c), wt(c));
  else if (pos(c) ≥ active)
    fprintf(stderr, "␣(secondary␣"O"d,␣purified),␣length␣"O"d:\n", pos(c) + 1, 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].i);
}
```

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

```
#define sanity_checking 0   /* set this to 1 if you suspect a bug */

⟨Subroutines 6⟩ +≡
  void sanity(void)
  {
    register int k, x, i, l, r, q, qq;
    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);
        qq = 0;
      } else {
        if (l > r) fprintf(stderr, "itm>loc␣in␣node␣"O"d!\n", i);
        else {
          if (set[r].i ≠ 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);
          }
          if (pos(l) < active) {
            if (r < l + size(l)) q = +1; else q = −1;   /* in or out? */
            if (q ∗ qq < 0) {
              fprintf(stderr, "Flipped␣status␣at␣option␣"O"d␣of␣item", r − l + 1);
              print_item_name(l, stderr);
              fprintf(stderr, "␣in␣node␣"O"d!\n", i);
            }
            qq = q;
          }
        }
      }
    }
  }
```

15. 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 15⟩ ≡
  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 16⟩;
    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++) ;
    }
  }
```

This code is used in section 2.

16. ⟨Check for duplicate item name 16⟩ ≡

```
  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 15.

17. 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 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]; ) {
      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] 18⟩;
      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 19⟩;
        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 20⟩;
  ⟨Expand set 21⟩;
  ⟨Adjust nd 22⟩;
```

This code is used in section 2.

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

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

```
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 17.

19. ⟨ Remove *last_node* from its item list 19 ⟩ ≡

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

This code is used in section 17.

20. ⟨ Initialize *item* 20 ⟩ ≡

```
active = itemlength = last_itm − 1;
for (k = 0, j = primextra; k < itemlength; k++)
  oo, item[k] = j, j += (k + 2 < second ? primextra : secondextra) + size((k + 1) ≪ 2);
setlength = j − 4;    /* a decent upper bound */
if (second ≡ max_cols) osecond = active, second = j;
else osecond = second − 1;
```

This code is used in section 17.

21. 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* 21 ⟩ ≡

```
for ( ; k; k−−) {
  o, j = item[k − 1];
  if (k ≡ second) second = j;    /* second is now an index into set */
  oo, size(j) = size(k ≪ 2);
  if (size(j) ≡ 0 ∧ k ≤ osecond) baditem = k;
  o, pos(j) = k − 1;
  oo, rname(j) = rname(k ≪ 2), lname(j) = lname(k ≪ 2);
  if (k ≤ osecond) o, wt(j) = w0;
}
```

This code is used in section 17.

22. ⟨ Adjust *nd* 22 ⟩ ≡

```
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].i = k;
}
```

This code is used in section 17.

23. ⟨ Report an uncoverable item 23 ⟩ ≡

```
{
  if (vbose & show_choices) {
    fprintf(stderr, "Item");
    print_item_name(item[baditem - 1], stderr);
    fprintf(stderr, "␣has␣no␣options!\n");
  }
}
```

This code is used in section 2.

24. 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 24 ⟩ ≡

```
fprintf(stderr, "("O"lld␣options,␣"O"d+"O"d␣items,␣"O"d␣entries␣successfully␣read)\n",
    options, osecond, itemlength - osecond, last_node);
```

This code is used in section 2.

25. The item lengths after input are shown (on request). But there's little use trying to show them after the process is done, since they are restored somewhat blindly. (Failures of the linked-list implementation in DLX2 could sometimes be detected by showing the final lengths; but that reasoning no longer applies.)

⟨ Report the item totals 25 ⟩ ≡

```
{
  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.

26. ⟨ Randomize the *item* list 26 ⟩ ≡

```
for (k = active; k > 1; ) {
  mems += 4, j = gb_unif_rand(k);
  k--;
  oo, oo, t = item[j], item[j] = item[k], item[k] = t;
  oo, pos(t) = k, pos(item[j]) = j;
}
```

This code is used in section 2.

27. Binary branching versus d-way branching. Nodes of the search tree in the previous program SSXCC, on which this one is based, are characterized by the name of a primary item i that hasn't yet been covered. If that item currently appears in d options $\{o_1, \ldots, o_d\}$, node i has d children, one for each choice of the option that will cover i.

The present program, however, makes 2-way branches, and its nodes are labeled with both an item i and an option o. The left child of node (i, o) represents the subproblem in which i is covered by o, as before. But the right child represents the subproblem for which option o is removed but item i is still uncovered (unless $d = 1$, in which case there's no right child). Thus our search tree is now rather like the binary tree that represents a general tree. (See *The Art of Computer Programming*, Section 2.3.2.)

There usually is no good reason to do binary branching when we choose i so as to minimize d. On the right branch, i will have $d - 1$ remaining options; and no item i' will have fewer than $d - 1$.

But this program is intended to provide the basis for *other* programs, which extend the branching heuristic by taking dynamic characteristics of the solution process into account. While exploring the left branch in such extensions, we might discover that a certain item i' is difficult to cover; hence we might prefer to branch on an option o' that covers i', after rejecting o for item i.

28. We shall say that we're in stage s when we've taken s left branches. We'll also say, as usual, that we're at level l when we've taken l branches altogether.

Suppose, for instance, that we're at level 5, having rejected o_1 for i_1, accepted o_2 for i_2, accepted o_3 for i_3, rejected o_4 for i_4, and rejected o_5 for i_5. Then we will have $stage = 2$, and $choice[k] = o_k$ for $0 \le k < 5$; here each o_k is a node whose itm field is i_k. Also

$$\begin{aligned} stagelevel[0] &= 0,\\ stagelevel[1] &= 0,\\ stagelevel[2] &= 1,\\ stagelevel[3] &= 2,\\ stagelevel[4] &= 2,\\ stagelevel[5] &= 2; \end{aligned} \qquad \begin{aligned} levelstage[0] &= 1,\\ levelstage[1] &= 2,\\ levelstage[2] &= 5. \end{aligned}$$

The option $choice[k]$ has been accepted if and only if $levelstage[stagelevel[k]] = k$.

⟨ Global variables 3 ⟩ +≡

```
int stage;   /* number of choices in current partial solution */
int level;   /* current depth in the search tree (which is binary) */
int choice[max_level];   /* the option and item chosen on each level */
int deg[max_level];   /* the number of options the item had at that time */
int levelstage[max_stage];   /* the most recent level at each stage */
int stagelevel[max_level];   /* the stage that corresponds to each level */
ullng profile[max_stage];   /* number of search tree nodes on each stage */
```

29. The dancing. Our strategy for generating all exact covers will be to repeatedly choose an item that appears to be hardest to cover, namely an item whose set is currently smallest, among 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 sets are maintained. Depth-first search means last-in-first-out maintenance of data structures; and the sparse-set representations make it particularly easy to undo what we've done at deeper levels.

The basic operation is "including an option." That means (i) removing from the current subproblem all of the other options with which it conflicts, and (ii) considering all of its primary items to be covered, by making them inactive.

```
⟨ Solve the problem 29 ⟩ ≡
  {
    level = stage = 0;
  forward: nodes++;
    if (vbose & show_profile) profile[stage]++;
    if (sanity_checking) sanity();
    ⟨ Maybe do a forced move 39 ⟩;
    ⟨ Do special things if enough mems have accumulated 31 ⟩;
    ⟨ Set best_itm to the best item for branching and t to its size 41 ⟩;
    if (forced) {
      o, best_itm = force[--forced];
      ⟨ Do a forced move and goto advance 45 ⟩;
    }
    if (t ≡ inf_size) ⟨ Visit a solution and goto backup 42 ⟩;
    ⟨ Save the currently active items and their sizes 43 ⟩;
  advance: oo, choice[level] = cur_choice = set[best_itm].i;
    o, deg[level] = t;
    if (¬include_option(cur_choice)) goto tryagain;
    ⟨ Increase stage 32 ⟩; ⟨ Increase level 33 ⟩;
    goto forward;
  tryagain: if (t ≡ 1) goto prebackup;
    if (vbose & show_choices) fprintf(stderr, "Backtracking␣in␣stage␣"O"d\n", stage);
    goto purgeit;
  prebackup: o, saveptr = saved[stage];
  backup: if (--stage < 0) goto done;
    if (vbose & show_choices) fprintf(stderr, "Backtracking␣to␣stage␣"O"d\n", stage);
    o, level = levelstage[stage];
  purgeit: if (o, deg[level] ≡ 1) goto prebackup;
    ⟨ Restore the currently active items and their sizes 44 ⟩;
    o, cur_choice = choice[level];
    ⟨ Remove the option cur_choice 37 ⟩;
    ⟨ Increase level 33 ⟩;
    goto forward;
  }
```

This code is used in section 2.

30. We save the sizes of active items on *savestack*, whose entries have two fields l and r, for an item and its size. This stack makes it easy to undo all deletions, by simply restoring the former sizes.

⟨ Global variables 3 ⟩ +≡
```
int level;    /* number of choices in current partial solution */
int choice[max_level];    /* the node chosen on each level */
int saved[max_level + 1];    /* size of savestack on each level */
twoints savestack[savesize];
int saveptr;    /* current size of savestack */
int tough_itm;    /* item whose set of options has just become empty */
```

31. ⟨ Do special things if enough *mems* have accumulated 31 ⟩ ≡
```
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 29.

32. ⟨ Increase *stage* 32 ⟩ ≡
```
if (++stage > maxs) {
  if (stage ≥ max_stage) {
    fprintf(stderr, "Too␣many␣stages!\n");
    exit(−40);
  }
  maxs = stage;
}
```
This code is used in section 29.

33. ⟨ Increase *level* 33 ⟩ ≡
```
if (++level > maxl) {
  if (level ≥ max_level) {
    fprintf(stderr, "Too␣many␣levels!\n");
    exit(−4);
  }
  maxl = level;
}
oo, stagelevel[level] = stage, levelstage[stage] = level;
```
This code is used in section 29.

34. The *include_option* routine extends the current partial solution, by using option *opt* to cover one or more of the presently uncovered primary items. It returns 0, however, if that would make some other primary item uncoverable. (In the latter case, *tough_itm* is set to the item that was problematic.)

⟨ Subroutines 6 ⟩ +≡

```
int include_option(int opt)
{
  register int c, optp, nn, nnp, ss, ii, iii, p, pp, s;
  subroutine_overhead;
  if (vbose & show_choices) {
    fprintf(stderr, "S"O"d:", stage);
    print_option(opt, stderr, 1);
  }
  for ( ; o, nd[opt − 1].itm > 0; opt−−) ;    /* move to the beginning of the option */
  for ( ; o, (ii = nd[opt].itm) > 0; opt++) {
    pp = nd[opt].loc;    /* where opt appears in ii's set */
    o, p = pos(ii);    /* where ii appears in item */
    if (p < active) ⟨ Deactivate item ii, or return 0 35 ⟩;
  }
  return 1;
}
```

35. We don't need to remove options from the set of *ii*, because *ii* will soon be inactive. But of course we do need to remove the options that conflict with *opt* from the sets of their items.

⟨ Deactivate item *ii*, or return 0 35 ⟩ ≡

```
{
  o, ss = size(ii);
  if (ii < second) c = 0;
  else o, c = nd[opt].clr;
  for (s = ii + ss − 1; s ≥ ii; s−−)
    if (s ≠ pp) {
      o, optp = set[s].i;
      if (c ≡ 0 ∨ (o, nd[optp].clr ≠ c)) ⟨ Remove optp from its other sets, or return 0 36 ⟩;
    }
  o, iii = item[−−active];
  oo, item[active] = ii, item[p] = iii;
  oo, pos(ii) = active, pos(iii) = p;
}
```

This code is used in section 34.

36. At this point *optp* points to a node of an option that we want to remove from the current subproblem. We swap it out of the sets of all its items except for $nd[optp].itm$ itself, and except for the sets of inactive secondary items. (The latter have been purified, and we shouldn't mess with their sets.)

⟨ Remove *optp* from its other sets, or return 0 36 ⟩ ≡

```
{
  register int nn, ii, p, ss, nnp;
  for (nn = optp; o, nd[nn − 1].itm > 0; nn−−) ;    /* move to beginning of the option */
  for ( ; o, (ii = nd[nn].itm) > 0; nn++)
    if (nn ≠ optp) {
      p = nd[nn].loc;
      if (p ≥ second ∧ (o, pos(ii) ≥ active)) continue;    /* ii already purified */
      o, ss = size(ii) − 1;
      if (ss ≤ 1 ∧ p < second) {
        if (ss ≡ 0) {
          if ((vbose & show_details) ∧ level < show_choices_max ∧ level ≥ maxl − show_choices_gap) {
            fprintf(stderr, "␣can't␣cover");
            print_item_name(ii, stderr);
            fprintf(stderr, "\n");
          }
          tough_itm = ii;
          forced = 0;
          return 0;    /* abort the deletion, lest ii be wiped out */
        }
        else o, force[forced++] = ii;
      }
      o, nnp = set[ii + ss].i;
      o, size(ii) = ss;
      oo, set[ii + ss].i = nn, set[p].i = nnp;
      oo, nd[nn].loc = ii + ss, nd[nnp].loc = p;
      updates++;
    }
}
```

This code is used in section 35.

37. At this point every active primary item has at least two options in its set. Therefore, when we delete *cur_choice* from the sets of each of its active items, every set will still be nonempty.

⟨ Remove the option *cur_choice* 37 ⟩ ≡

```
{
  register int ii, ss, p, nnp;
  for ( ; o, nd[cur_choice − 1].itm > 0; cur_choice−−) ;    /* move to beginning */
  for ( ; o, (ii = nd[cur_choice].itm) > 0; cur_choice++) {
    p = nd[cur_choice].loc;
    if (p ≥ second ∧ (o, pos(ii) ≥ active)) continue;    /* ii inactive */
    o, ss = size(ii) − 1;
    oo, nnp = set[ii + ss].i, size(ii) = ss;
    oo, set[ii + ss].i = cur_choice, set[p].i = nnp;
    oo, nd[cur_choice].loc = ii + ss, nd[nnp].loc = p;
    updates++;
  }
}
```

This code is used in section 29.

38. If a weight becomes dangerously large, we rescale all the weights.

(That will happen only when *dwfactor* isn't 1.0. Adding a constant eventually "converges": For example, if the constant is 1, we have convergence to 2^{17} after $2^{17}-1 = 16777215$ steps. If the constant *dw* is .250001, convergence to `8.38861e+06` occurs after 25165819 steps!)

(Note: I threw in the parameters *dw* and *dwfactor* only to do experiments. My preliminary experiments didn't turn up any noteworthy results. But I didn't have time to do a careful study; hence there might be some settings that work unexpectedly well. The code for rescaling might be flaky, since it hasn't been tested very thoroughly at all.)

```
#define dangerous 1·10^32_F
#define wmin 1·10^−30_F

⟨Increase the weight of tough_itm 38⟩ ≡
  cmems += 2, oo, wt(tough_itm) += dw;
  if (vbose & show_record_weights ∧ wt(tough_itm) > maxwt) {
    maxwt = wt(tough_itm);
    fprintf(stderr, ""O"8.1f␣", maxwt);
    print_item_name(tough_itm, stderr);
    fprintf(stderr, "␣"O"lld\n", nodes);
  }
  if (vbose & show_weight_bumps) {
    print_item_name(tough_itm, stderr);
    fprintf(stderr, "␣wt␣"O".1f\n", wt(tough_itm));
  }
  dw *= dwfactor;
  if (wt(tough_itm) ≥ dangerous) {
    register int k;
    register float t;

    tmems = mems;
    for (k = 0; k < itemlength; k++)
      if (o, item[k] < second) {
        o, t = wt(item[k]) * 1·10^−20_F;
        o, wt(item[k]) = (t < wmin ? wmin : t);
      }
    dw *= 1·10^−20_F;
    if (dw < wmin) dw = wmin;
    w0 *= 1·10^−20_F;
    if (w0 < wmin) w0 = wmin;
    cmems += mems − tmems;
  }
```

39.
```
⟨Maybe do a forced move 39⟩ ≡
  while (forced) {
    o, best_itm = force[−−forced];
    if (o, pos(best_itm) < active) {
      ⟨Do a forced move and goto advance 45⟩;
    }
  }
```

This code is used in section 29.

40. ⟨ Subroutines 6 ⟩ +≡

```
void print_weights(void)
{
  register int k;
  for (k = 0; k < itemlength; k++)
    if (item[k] < second ∧ wt(item[k]) ≠ w0) {
      print_item_name(item[k], stderr);
      fprintf(stderr, "␣wt␣"O".1f\n", wt(item[k]));
    }
}
```

41. The “best item” is considered to be an active primary 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.)

We assume that t is set to *inf_size* if and only if all primary items have been covered. We also assume that t is set to 1 if and only if some uncovered primary item has size 1. (Every uncovered primary item must have size at least 1, because we've been careful to avoid any choices that could cause a size to become 0.)

Notice that a secondary item is active if and only if it has not been purified (that is, if and only if it hasn't yet appeared in a chosen option).

Important: The code below will usually be changed, via a change file, so that the best item is chosen by using another heuristic. Whatever heuristic is used, it *must* deliver a primary item whose option size s is 1, if such an item exists. In other words, it must somehow recognize a forced move, unless there are no forced moves.

```
#define inf_size #7fffffff
⟨Set best_itm to the best item for branching and t to its size 41⟩ ≡
  {
    t = inf_size, tmems = mems;
    if ((vbose & show_details) ∧ level < show_choices_max ∧ level ≥ maxl − show_choices_gap)
      fprintf(stderr, "Level␣"O"d:", level);
    for (k = 0; k < active; k++)
      if (o, item[k] < second) {
        o, 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 ≤ 1) {
          if (s ≡ 0) fprintf(stderr, "I'm␣confused.\n");    /* include_option missed this */
          else o, force[forced++] = item[k];
        } else if (s ≤ t) {
          if (s < t) best_itm = item[k], t = s;
          else if (item[k] < best_itm) best_itm = item[k];    /* suggested by P. Weigel */
        }
      }
    if ((vbose & show_details) ∧ level < show_choices_max ∧ level ≥ maxl − show_choices_gap) {
      if (forced) fprintf(stderr, "␣found␣"O"d␣forced\n", forced);
      else if (t ≡ inf_size) fprintf(stderr, "␣solution\n");
      else {
        fprintf(stderr, "␣branching␣on");
        print_item_name(best_itm, stderr);
        fprintf(stderr, "("O"d)\n", t);
      }
    }
    if (t > maxdeg ∧ t < inf_size ∧ ¬forced) maxdeg = t;
    if (shape_file) {
      if (t ≡ inf_size) fprintf(shape_file, "sol\n");
      else {
        fprintf(shape_file, ""O"d", t);
        print_item_name(best_itm, shape_file);
        fprintf(shape_file, "\n");
      }
      fflush(shape_file);
```

```
    }
    cmems += mems − tmems;
  }
```

This code is used in section 29.

42. ⟨ Visit a solution and **goto** *backup* 42 ⟩ ≡

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

This code is used in section 29.

43. ⟨ Save the currently active items and their sizes 43 ⟩ ≡

```
  o, saved[stage] = saveptr;
  if (saveptr + active > maxsaveptr) {
    if (saveptr + active ≥ savesize) {
      fprintf(stderr, "Stack␣overflow␣(savesize="O"d)!\n", savesize);
      exit(−5);
    }
    maxsaveptr = saveptr + active;
  }
  for (p = 0; p < active; p++)
    ooo, savestack[saveptr + p].l = item[p], savestack[saveptr + p].r = size(item[p]);
  saveptr += active;
```

This code is used in section 29.

44. ⟨ Restore the currently active items and their sizes 44 ⟩ ≡

```
  o, active = saveptr − saved[stage];
  saveptr = saved[stage];
  for (p = 0; p < active; p++) oo, size(savestack[saveptr + p].l) = savestack[saveptr + p].r;
```

This code is used in section 29.

45. A forced move occurs when *best_itm* has only one remaining option. In this case we can streamline the computation, because there's no need to save the current active sizes. (They won't be looked at.)

⟨ Do a forced move and **goto** *advance* 45 ⟩ ≡

```
  {
    if ((vbose & show_choices) ∧ level < show_choices_max) fprintf(stderr, "(forcing)\n");
    o, saved[stage] = saveptr;      /* nothing placed on savestack */
    t = 1;
    goto advance;
  }
```

This code is used in sections 29 and 39.

46. ⟨ Subroutines 6 ⟩ +≡

```
void print_savestack(int start, int stop)
{
  register k;
  for (k = start; k < stop; k++) {
    print_item_name(savestack[k].l, stderr);
    fprintf(stderr, "("O"d),␣"O"d\n", savestack[k].l, savestack[k].r);
  }
}
```

47. ⟨ Subroutines 6 ⟩ +≡

```
void print_state(void)
{
  register int l, s;
  fprintf(stderr, "Current␣state␣(level␣"O"d,␣stage␣"O"d):\n", level, stage);
  for (l = 0; l < level; l++) {
    if (levelstage[stagelevel[l]] ≠ l) fprintf(stderr, "~");
    print_option(choice[l], stderr, −1);
    fprintf(stderr, "␣(of␣"O"d)\n", deg[l]);
    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);
}
```

48. 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 node degrees, preceded by '~' if the node wasn't the current node in its stage (that is, if the node represents an option that has already been fully explored — "we've been there done that").

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 the kth choice in stage 0 and has degree d, the estimate is $(k-1)/(d+k-1)$ plus $1/(d+k-1)$ times the recursively evaluated estimate for the kth subtree. (This estimate might obviously be very misleading, in some cases, but at least it tends to grow monotonically.)

Fine point: If we've just backtracked within stage *stage*, the string of node degrees with end with a '~' entry, and we haven't yet made *any* choice in the current stage. The test '$l \equiv level - 1$' below uses the fact that $levelstage[stage] = level$ to adjust the fractional estimate appropriately for the partial progress in the current stage.

⟨ Subroutines 6 ⟩ +≡

```
void print_progress(void)
{
  register int l, ll, k, d, c, p, ds = 0;
  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++) {
    if (l < show_levels_max)
      fprintf(stderr, "␣"O"s"O"d", levelstage[stagelevel[l]] ≡ l ? "" : "~", deg[l]);
    if (levelstage[stagelevel[l]] ≡ l ∨ l ≡ level − 1) {    /* see remark above */
      for (k = 1, d = deg[l], ll = l − 1; ll ≥ 0 ∧ stagelevel[ll] ≡ stagelevel[l]; k++, d++, ll−−) ;
      fd *= d, f += (k − 1)/fd;     /* choice l is treated like k of d */
    }
    if (l ≥ show_levels_max ∧ ¬ds) ds = 1, fprintf(stderr, "...");
  }
  fprintf(stderr, "␣"O".5f\n", f + 0.5/fd);
}
```

49. ⟨ Print the profile 49 ⟩ ≡

```
{
  fprintf(stderr, "Profile:\n");
  for (k = 0; k ≤ maxs; k++) fprintf(stderr, ""O"3d:␣"O"lld\n", k, profile[k]);
}
```

This code is used in section 2.

50. Index.

SSXCC-BINARY