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

1. Intro. This is a modification of the program SSXCC-BINARY, extending it to take into account multiplicities for the items. For the multiplicities, the input format is the same as the one defined in the program DLX3. The extensions were introduced by Filip Stappers in 2023.

This program is another experiment in the use of so-called sparse-set data structures instead of the dancing links. It is written as if living on a planet where the sparse-set ideas are well known, but doubly linked links are almost unheard-of.

I suggest that you read SSXCC, SSXCC-BINARY and DLX3 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 the list of one its items.) 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⟩;
  int main(int argc, char *argv[])
  {
    register int c, cc, i, j, k, p, pp, q, r, s, t, cur_choice, cur_node, best_itm, istage, score, best_s, best_l;
    ⟨Process the command line 4⟩;
    ⟨Input the item names 15⟩;
    ⟨Input the options 19⟩;
    if (vbose & show_basics) ⟨Report the successful completion of the input phase 27⟩;
    if (vbose & show_tots) ⟨Report the item totals 28⟩;
    imems = mems, mems = 0;
    if (baditem) ⟨Report an uncoverable item 25⟩
    else {
      if (randomizing) ⟨Randomize the item list 29⟩;
      ⟨Solve the problem 32⟩;
    }
  done: if (vbose & show_profile) ⟨Print the profile 52⟩;
    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 (threeints);
      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, 31, and 33.

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, 37, 43, 49, 50, and 51.

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}, $\ldots$, 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 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.)

And finally, we have the *bound* and *slack* fields, $set[x-6]$ and $set[x-7]$. These are used in the same way as in DLX3 to keep track of the item's multiplicities.

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 slack(x) set[(x) − 5].i    /* if multiplicity [u..v], slack is equal to v-u and does not change */
#define bound(x) set[(x) − 6].i    /* residual capacity of this item */
#define wt(x) set[(x) − 7].f    /* the current floating-point "weight" of x */
#define primextra 7    /* this many extra entries of set for each primary item */
#define secondextra 4    /* and this many for each secondary item */
#define maxextra 7    /* maximum of primextra and secondextra */
#define ipropcount 6    /* the number of bytes used for each item in the input phase */
```

⟨ 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 oactive;    /* value of active before swapping out current-choice 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 = bound − slack */
  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 struct {
    int l, s, b;
  } threeints;
  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 %d!\n", p);
    return;
  }
  for (q = p; ; ) {
    print_item_name(x, stream);
    if (nd[q].clr) fprintf(stream, ":%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, " (%d of %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␣("O"d)", c);
  print_item_name(c, stderr);
  if (c < second) {
    if (slack(c) ∨ bound(c) ≠ 1) fprintf(stderr, "␣("O"d,"O"d)", bound(c) − slack(c), bound(c));
    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);
}
void print_items()
{
  register int i;
  for (i = 0; i < itemlength; i++) print_itm(item[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()
  {
    register int k, x, i, l, r, q, qq;
    int ok = 1;
    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,␣"O"d)!\n", k, pos(x), x);
        ok = 0;
      }
    }
    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);
          ok = 0;
        }
        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!,␣set[r].i="O"d\n", i, set[r].i);
            ok = 0;
          }
          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!,␣q,qq="O"d,"O"d\n", i, q, qq);
              ok = 0;
            }
            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 *ipropcount* entries of *set* per item, while initially 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;
    ⟨ Scan an item name, possibly prefixed by bounds 16 ⟩;
    oo, lname(last_itm ∗ ipropcount) = namebuf.lr.l, rname(last_itm ∗ ipropcount) = namebuf.lr.r;
    o, slack(last_itm ∗ ipropcount) = q − r, bound(last_itm ∗ ipropcount) = q;
    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. ⟨ Scan an item name, possibly prefixed by bounds 16 ⟩ ≡

```
  if (second ≡ max_cols) istage = 0; else istage = 2;
start_name: for (j = 0; j < 8 ∧ (o, ¬isspace(buf[p + j])); j++) {
    if (buf[p + j] ≡ ':') {
      if (istage) panic("Illegal␣':'␣in␣item␣name");
      ⟨ Convert the prefix to an integer, q 17 ⟩;
      r = q, istage = 1;
      goto start_name;
    } else if (buf[p + j] ≡ '|') {
      if (istage > 1) panic("Illegal␣'|'␣in␣item␣name");
      ⟨ Convert the prefix to an integer, q 17 ⟩;
      if (q ≡ 0) panic("Upper␣bound␣is␣zero");
      if (istage ≡ 0) r = q;
      else if (r > q) panic("Lower␣bound␣exceeds␣upper␣bound");
      istage = 2;
      goto start_name;
    }
    o, namebuf.str[j] = buf[p + j];
  }
  switch (istage) {
  case 1: panic("Lower␣bound␣without␣upper␣bound");
  case 0: q = r = 1;
  case 2: break;
  }
  if (j ≡ 0) panic("Item␣name␣empty");
  if (j ≡ 8 ∧ ¬isspace(buf[p + j])) panic("Item␣name␣too␣long");
  ⟨ Check for duplicate item name 18 ⟩;
```

This code is used in section 15.

17. ⟨ Convert the prefix to an integer, q 17 ⟩ ≡

```
  for (q = 0, pp = p; pp < p + j; pp++) {
    if (buf[pp] < '0' ∨ buf[pp] > '9') panic("Illegal␣digit␣in␣bound␣spec");
    q = 10 * q + buf[pp] − '0';
  }
  p = pp + 1;
  while (j) namebuf.str[−−j] = 0;
```

This code is used in section 16.

18. ⟨ Check for duplicate item name 18 ⟩ ≡

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

This code is used in section 16.

19. 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 19⟩ ≡
  while (1) {
    if (¬fgets(buf, bufsize, stdin)) break;
    if (o, buf[p = strlen(buf) − 1] ≠ '\n') panic("Option␣line␣too␣long");
    for (p = 0; o, isspace(buf[p]); p++) ;
    if (buf[p] ≡ '|' ∨ ¬buf[p]) continue;      /* bypass comment or blank line */
    i = last_node;      /* remember the spacer at the left of this option */
    for (pp = 0; buf[p]; ) {
      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] 20⟩;
      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 21⟩;
        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 22⟩;
  ⟨Expand set 23⟩;
  ⟨Adjust nd 24⟩;
  ⟨Make optionless items invisible 26⟩;
```

This code is used in section 2.

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

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

```
for (k = (last_itm − 1) ∗ ipropcount; k ≥ 0; k −= ipropcount) {
  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/ipropcount, nd[last_node].loc = t;
if ((k/ipropcount) < second) pp = 1;
o, size(k) = t + 1, pos(k) = last_node;
```

This code is used in section 19.

21. ⟨ Remove *last_node* from its item list 21 ⟩ ≡

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

This code is used in section 19.

22. ⟨ Initialize *item* 22 ⟩ ≡

```
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) ∗ ipropcount);
  if (j < item[k] + ipropcount) j = item[k] + ipropcount;
}
setlength = j − ipropcount;    /∗ a decent upper bound ∗/
if (second ≡ max_cols) osecond = active, second = j;
else osecond = second − 1;
```

This code is used in section 19.

23. 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* 23 ⟩ ≡

```
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 ∗ ipropcount);
  o, pos(j) = k − 1;
  oo, rname(j) = rname(k ∗ ipropcount), lname(j) = lname(k ∗ ipropcount);
  oo, slack(j) = slack(k ∗ ipropcount), bound(j) = bound(k ∗ ipropcount);
  if (k ≤ osecond) {
    o, wt(j) = w0;
    if (size(j) < bound(j) − slack(j)) baditem = k;
    else if (size(j) ≡ 0) force[forced++] = j;
  } else if (size(j) ≡ 0) force[forced++] = j;
}
```

This code is used in section 19.

24. ⟨ Adjust *nd* 24 ⟩ ≡

```
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 19.

25. ⟨ Report an uncoverable item 25 ⟩ ≡

```
{
  if (vbose & show_choices) {
    k = item[baditem - 1];
    fprintf(stderr, "Item");
    print_item_name(k, stderr);
    fprintf(stderr, " has fewer than "O"d options!\n", bound(k) - slack(k));
  }
}
```

This code is used in section 2.

26. ⟨ Make optionless items invisible 26 ⟩ ≡

```
while (forced) {
  o, j = force[--forced];
  if (vbose & show_details) {
    fprintf(stderr, "Deactivating optionless item");
    print_item_name(j, stderr);
    fprintf(stderr, "\n");
  }
  oo, i = item[--active], pp = pos(j);
  oo, item[active] = j, item[pp] = i;
  oo, pos(j) = active, pos(i) = pp;
}
```

This code is used in section 19.

27. 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 27 ⟩ ≡

```
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.

28. 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 28 ⟩ ≡
```
{
  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.

29. ⟨ Randomize the *item* list 29 ⟩ ≡
```
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.

30. 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.

31. 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 ∗/

32. The dancing. Our strategy for generating all exact covers will be to repeatedly choose an active primary item and to branch on the ways to reduce the possibilities for covering that item. 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 have their bounds decreased by 1. If this would make the bound of an item 0, we can make that item inactive.

⟨ Solve the problem 32 ⟩ ≡
```
  {
    level = stage = 0;
  forward: nodes++;
    if (vbose & show_profile) profile[stage]++;
    if (sanity_checking) sanity();
    ⟨ Maybe do a forced move 42 ⟩;
    ⟨ Do special things if enough mems have accumulated 34 ⟩;
    ⟨ Set best_itm to the best item for branching, and let score be its branching degree 44 ⟩;
    if (forced) {
      o, best_itm = force[--forced];
      ⟨ Do a forced move and goto advance 48 ⟩;
    }
    if (score ≡ inf_size) ⟨ Visit a solution and goto backup 45 ⟩;
    ⟨ Save the currently active items and their sizes and bounds 46 ⟩;
  advance: oo, choice[level] = cur_choice = set[best_itm].i;
    o, deg[level] = score;
    if (¬include_option(cur_choice)) goto tryagain;
    ⟨ Increase stage 35 ⟩; ⟨ Increase level 36 ⟩;
    goto forward;
  tryagain: if (score ≡ 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 and bounds 47 ⟩;
    o, cur_choice = choice[level];
    ⟨ Remove the option cur_choice 40 ⟩;
    ⟨ Increase level 36 ⟩;
    goto forward;
  }
```
This code is used in section 2.

33. 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 and bounds.

⟨ 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 */
threeints savestack[savesize];
int saveptr;  /* current size of savestack */
int tough_itm;  /* item whose set of options has just become empty */
```

34. ⟨ Do special things if enough *mems* have accumulated 34 ⟩ ≡
```
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 32.

35. ⟨ Increase *stage* 35 ⟩ ≡
```
if (++stage > maxs) {
  if (stage ≥ max_stage) {
    fprintf(stderr, "Too␣many␣stages!\n");
    exit(−40);
  }
  maxs = stage;
}
```
This code is used in section 32.

36. ⟨ Increase *level* 36 ⟩ ≡
```
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 32.

37. The *include_option* routine extends the current partial solution, by hiding option *opt*. In addition, it will cover any primary items in *opt* if their bound after hiding *opt* becomes 0. The routine 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) {
        if (ii ≥ second) continue;    /* secondary item has been purified */
        confusion("active");    /* primary item of an active option must be active */
      }
      ⟨Cover or commit item ii, decrease bound of item ii if primary, potentially deactivate it, or return
          0 38⟩;
    }
    return 1;
  }
```

38. We need to remove the options that conflict with *opt* from the sets of their items.

⟨ Cover or commit item *ii*, decrease bound of item *ii* if primary, potentially deactivate it, or return 0 38 ⟩ ≡

```
{
  if (ii < second) oo, bound(ii)--;
  if (ii ≥ second ∨ bound(ii) ≡ 0) {
    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 39 ⟩;
      }
    o, p = pos(ii);    /* note that pos(ii) might have changed */
    o, iii = item[--active];
    oo, item[active] = ii, item[p] = iii;
    oo, pos(ii) = active, pos(iii) = p;
  } else {
    o, ss = size(ii) - 1;
    if (oo, ss < bound(ii) - slack(ii)) {
      if ((vbose & show_details) ∧ level < show_choices_max ∧ level ≥ maxl - show_choices_gap) {
        fprintf(stderr, "␣can't␣cover");
        print_item_name(item[ii], stderr);
        fprintf(stderr, "\n");
      }
      tough_itm = ii;
      forced = 0;
      return 0;    /* abort the deletion, lest ii be wiped out */
    }
    if (ss ≡ 0) {    /* Just deactivate item ii, no hiding needed */
      o, iii = item[--active];
      oo, item[active] = ii, item[p] = iii;
      oo, pos(ii) = active, pos(iii) = p;
    } else {
      oo, nnp = set[ii + ss].i, size(ii) = ss;
      oo, set[ii + ss].i = opt, set[pp].i = nnp;
      oo, nd[opt].loc = ii + ss, nd[nnp].loc = pp;
      updates++;
    }
  }
}
```

This code is used in section 37.

39. 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 the sets of inactive secondary items. (These have been purified, and we shouldn't mess with their sets.)

⟨ Remove *optp* from its other sets, or return 0 39 ⟩ ≡

```
{
  register int nn, ii, iii, p, pp, 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++) {
    p = nd[nn].loc;
    if (p ≥ second ∧ (o, pos(ii)) ≥ active) continue;   /* ii already purified */
    o, ss = size(ii) − 1;
    if (p < second) {
      if (oo, ss < bound(ii) − slack(ii)) {
        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, ",␣size="O"d,␣bound="O"d,␣slack="O"d,␣u="O"d\n", ss, bound(ii),
              slack(ii), bound(ii) − slack(ii));
        }
        tough_itm = ii;
        forced = 0;
        return 0;   /* abort the deletion, lest ii be wiped out */
      }
      if (ss ≡ 0) {
        o, iii = item[−− active];
        o, pp = pos(ii);
        if (vbose & show_details) {
          fprintf(stderr, "Empty␣option␣list,␣deactivating");
          print_item_name(ii, stderr);
          fprintf(stderr, "\n");
        }
        oo, item[active] = ii, item[pp] = iii;
        oo, pos(ii) = active, pos(iii) = pp;
      }
    }
    if (ss > 0) {
      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 38.

40. ⟨Remove the option *cur_choice* 40⟩ ≡

```
{
  register int ii, iii, 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;
    if (p < second) {
      if (oo, ss < bound(ii) − slack(ii)) {
        if ((vbose & show_details) ∧ level < show_choices_max ∧ level ≥ maxl − show_choices_gap) {
          fprintf(stderr, "␣can't␣cover");
          print_item_name(item[ii], stderr);
          fprintf(stderr, "\n");
        }
        goto prebackup;
      }
      if (ss ≡ 0) {
        o, iii = item[−− active];
        o, pp = pos(ii);
        if (vbose & show_details) {
          fprintf(stderr, "Null␣move,␣deactivating");
          print_item_name(ii, stderr);
          fprintf(stderr, "\n");
        }
        oo, item[active] = ii, item[pp] = iii;
        oo, pos(ii) = active, pos(iii) = pp;
      }
    }
    if (ss > 0) {
      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 32.

41. 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 41⟩ ≡
  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;
  }
```

42. At level 0, the *force* stack contains primary items that had no options. Their lower bound was 0, so they should simply not appear.

```
⟨Maybe do a forced move 42⟩ ≡
  while (forced) {
    o, best_itm = force[−−forced];
    if (o, pos(best_itm) ≥ active) continue;
    ⟨Do a forced move and goto advance 48⟩;
  }
```

This code is used in section 32.

43. ⟨ 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]));
    }
}
```

44. The "best item" is considered to be an item that minimizes the branching degree. If there are several candidates, we choose the leftmost — unless we're randomizing, in which case we select one of them at random.

Consider an item that has four options $\{w, x, y, z\}$, and suppose its *bound* is 3. If the *slack* is zero, we've got to choose either w or x, so the branching degree is 2. But if $slack = 1$, we have three choices, w or x or y; if $slack = 2$, there are four choices; and if $slack \ge 3$, there are five, including the "null" choice.

In general, the branching degree turns out to be $l + s - b + 1$, where l is the length of the item, b is the current bound, and s is the minimum of b and the slack. This formula gives degree ≤ 0 if and only if l is too small to satisfy the item constraint; in such cases we will backtrack immediately. (It would have been possible to detect this condition early, before updating all the data structures and increasing *level*. But that would make the downdating process much more difficult and error-prone. Therefore I wait to discover such anomalies until item-choosing time.)

Let's assign the score $l + s - b + 1$ to each item. If two items have the same score, I prefer the one with smaller s, because slack items are less constrained. If two items with the same s have the same score, I (counterintuitively) prefer the one with larger b (hence larger l), because that tends to reduce the size of the final search tree.

Consider, for instance, the following example taken from MDANCE: If we want to choose 2 options from 4 in one item, and 3 options from 5 in another, where all slacks are zero, and if the items are otherwise independent, it turns out that the number of nodes per level if we choose the smaller item first is $(1, 3, 6, 6 \cdot 3, 6 \cdot 6, 6 \cdot 10)$. But if we choose the larger item first it is $(1, 3, 6, 10, 10 \cdot 3, 10 \cdot 6)$, which is smaller in the middle levels.

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).

#define *inf_size* `#7fffffff`

⟨ Set *best_itm* to the best item for branching, and let *score* be its branching degree 44 ⟩ ≡

```
{
  score = 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 = slack(item[k]);
      if (o, s > bound(item[k])) s = bound(item[k]);
      o, t = size(item[k]) + s − bound(item[k]) + 1;
      if ((vbose & show_details) ∧ level < show_choices_max ∧ level ≥ maxl − show_choices_gap) {
        print_item_name(item[k], stderr);
        if (bound(item[k]) ≠ 1 ∨ s ≠ 0) {
          fprintf(stderr, "("O"d:"O"d,"O"d)", bound(item[k]) − s, bound(item[k]), t);
        } else fprintf(stderr, "("O"d)", t);
      }
      if (t ≡ 1)
        for (i = bound(item[k]) − slack(item[k]); i > 0; i−−) o, force[forced++] = item[k];
      else if (t ≤ score ∧ (t < score ∨ (s ≤ best_s ∧ (s < best_s ∨ (size(item[k]) ≥ best_l ∧ (size(item[k]) >
            best_l ∨ (item[k] < best_itm)))))))
        score = t, best_itm = item[k], best_s = s, best_l = size(item[k]);
    }
  if ((vbose & show_details) ∧ level < show_choices_max ∧ level ≥ maxl − show_choices_gap) {
    if (score ≡ inf_size) fprintf(stderr, "␣solution\n");
    else if (forced) {
      fprintf(stderr, "␣found␣"O"d␣forced:", forced);
      for (i = 0; i < forced; i++) print_item_name(force[i], stderr);
      fprintf(stderr, "\n");
```

```
      } else {
        fprintf(stderr, "␣branching␣on");
        print_item_name(best_itm, stderr);
        fprintf(stderr, "("O"d)\n", score);
      }
    }
    if (score > maxdeg ∧ score < inf_size ∧ ¬forced) maxdeg = score;
    if (shape_file) {
      if (score ≡ inf_size) fprintf(shape_file, "sol\n");
      else {
        fprintf(shape_file, ""O"d", score);
        print_item_name(best_itm, shape_file);
        fprintf(shape_file, "\n");
      }
      fflush(shape_file);
    }
    cmems += mems − tmems;
  }
```

This code is used in section 32.

45. ⟨Visit a solution and **goto** *backup* 45⟩ ≡

```
  {
    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 32.

46. ⟨Save the currently active items and their sizes and bounds 46⟩ ≡

```
  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++) {
    mems += 4, savestack[saveptr + p].l = item[p], savestack[saveptr + p].s = size(item[p]);
    if (item[p] < second) o, savestack[saveptr + p].b = bound(item[p]);
  }
  saveptr += active;
```

This code is used in section 32.

47. ⟨ Restore the currently active items and their sizes and bounds 47 ⟩ ≡

```
o, active = saveptr − saved[stage];
saveptr = saved[stage];
for (p = 0; p < active; p++) {
  ooo, size(savestack[saveptr + p].l) = savestack[saveptr + p].s;
  if (savestack[saveptr + p].l < second) o, bound(savestack[saveptr + p].l) = savestack[saveptr + p].b;
}
```

This code is used in section 32.

48. A forced move occurs when *best_itm* has *bound* − *slack* remaining options. In this case we can streamline the computation, because there's no need to save the current active sizes and bounds. (They won't be looked at.)

⟨ Do a forced move and **goto** *advance* 48 ⟩ ≡

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

This code is used in sections 32 and 42.

49. ⟨ Subroutines 6 ⟩ +≡

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

50. ⟨ Subroutines 6 ⟩ +≡

```
void print_state(void)
{
  register int l, s;
  fprintf(stderr, "Current␣state␣(level␣"O"d):\n", level);
  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);
}
```

51. 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.)

⟨ 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) {
      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);
}
```

52. ⟨ Print the profile 52 ⟩ ≡

```
{
  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.

53. Index.

active: 9, 13, 14, 22, 26, 29, 37, 38, 39, 40, 42, 44, 46, 47.
advance: 32, 48.
argc: 2, 4.
argv: 2, 4.
b: 10.
backup: 32, 45.
baditem: 2, 9, 23, 25.
best_itm: 2, 32, 42, 44, 48.
best_l: 2, 44.
best_s: 2, 44.
bound: 8, 9, 13, 15, 23, 25, 38, 39, 40, 44, 46, 47.
buf: 3, 15, 16, 17, 19.
bufsize: 1, 3, 15, 19.
bytes: 2, 3.
c: 2, 13, 37, 51.
cc: 2.
choice: 31, 32, 33, 45, 50.
clr: 8, 12, 19, 38.
cmems: 2, 3, 41, 44.
confusion: 6, 37.
count: 2, 3, 45, 50, 51.
cur_choice: 2, 32, 40.
cur_node: 2.
d: 51.
dangerous: 41.
deg: 31, 32, 50, 51.
delta: 3, 4, 34.
done: 2, 32, 34, 45.
ds: 51.
dw: 3, 4, 8, 41.
dwfactor: 3, 4, 41.
exit: 4, 15, 35, 36, 46.
f: 8, 51.
fclose: 5.
fd: 51.
fflush: 44, 45.
fgets: 15, 19.
fopen: 4.
force: 9, 23, 26, 32, 42, 44.
forced: 9, 23, 26, 32, 38, 39, 42, 44.
forward: 32.
fprintf: 2, 4, 6, 11, 12, 13, 14, 15, 19, 25, 26, 27, 28, 32, 34, 35, 36, 37, 38, 39, 40, 41, 43, 44, 46, 48, 49, 50, 51, 52.
gb_init_rand: 4.
gb_rand: 3.
gb_unif_rand: 29.
i: 2, 8, 13, 14.
ii: 37, 38, 39, 40.
iii: 37, 38, 39, 40.
imems: 2, 3.
include_option: 32, 37.
inf_size: 32, 44.
ipropcount: 8, 15, 18, 20, 21, 22, 23.
isspace: 15, 16, 19.
istage: 2, 16.
item: 8, 9, 13, 14, 22, 23, 24, 25, 26, 28, 29, 37, 38, 39, 40, 41, 43, 44, 46.
itemlength: 2, 9, 13, 14, 22, 27, 28, 41, 43.
itm: 8, 12, 14, 19, 20, 21, 24, 31, 37, 39, 40.
j: 2.
k: 2, 11, 12, 14, 41, 43, 49, 51.
l: 10, 14, 50, 51.
last_itm: 9, 15, 18, 20, 22.
last_node: 2, 9, 12, 14, 19, 20, 21, 24, 27.
level: 31, 32, 33, 36, 38, 39, 40, 44, 48, 50, 51.
levelstage: 31, 32, 36, 45, 50, 51.
ll: 51.
lname: 8, 11, 15, 18, 20, 23.
loc: 8, 12, 14, 19, 20, 24, 37, 38, 39, 40.
lr: 10, 11, 15, 18, 19, 20.
m: 6.
main: 2.
max_cols: 1, 9, 15, 16, 22.
max_level: 1, 31, 33, 36.
max_nodes: 1, 9, 19, 20.
max_stage: 1, 31, 35.
maxcount: 3, 4, 45.
maxdeg: 2, 3, 44.
maxextra: 8, 9.
maxl: 2, 3, 36, 38, 39, 40, 44, 50.
maxs: 3, 35, 52.
maxsaveptr: 2, 3, 46.
maxwt: 3, 41.
mems: 1, 2, 3, 29, 34, 41, 44, 46, 50, 51.
mod: 1, 45.
namebuf: 10, 11, 15, 16, 17, 18, 19, 20.
nb: 23.
nd: 8, 9, 12, 14, 19, 20, 21, 24, 37, 38, 39, 40.
nn: 37, 39.
nnp: 37, 38, 39, 40.
node: 2, 8, 9.
node_struct: 8.
nodes: 2, 3, 32, 41.
O: 1.
o: 1.
oactive: 9.
ok: 14.
oo: 1, 15, 21, 22, 23, 26, 29, 32, 36, 38, 39, 40, 41.
ooo: 1, 47.
opt: 37, 38.

options: 3, 19, 27.
optp: 37, 38, 39.
osecond: 9, 22, 23, 27.
p: 2, 12, 13, 37, 39, 40, 51.
panic: 15, 16, 17, 18, 19, 20.
pos: 8, 13, 14, 20, 21, 23, 26, 29, 37, 38, 39, 40, 42.
pp: 2, 17, 19, 20, 26, 37, 38, 39, 40.
prebackup: 32, 40.
primextra: 8, 13, 22.
print_item_name: 11, 12, 13, 14, 25, 26, 38, 39, 40, 41, 43, 44, 49.
print_items: 13.
print_itm: 13.
print_option: 12, 37, 45, 50.
print_progress: 34, 51.
print_savestack: 49.
print_state: 34, 50.
print_weights: 2, 43.
printf: 45.
profile: 31, 32, 52.
prow: 12, 13.
purgeit: 32.
q: 2, 12, 14.
qq: 14.
r: 2, 10, 14.
random_seed: 3, 4.
randomizing: 2, 3, 4.
rname: 8, 11, 15, 18, 20, 23.
s: 2, 10, 37, 50.
sanity: 14, 32.
sanity_checking: 2, 14, 32.
saved: 32, 33, 46, 47, 48.
saveptr: 32, 33, 46, 47, 48.
savesize: 1, 33, 46.
savestack: 1, 3, 33, 46, 47, 48, 49.
score: 2, 32, 44, 48.
second: 9, 13, 15, 16, 19, 20, 22, 23, 28, 37, 38, 39, 40, 41, 43, 44, 46, 47.
secondextra: 8, 22.
set: 8, 9, 10, 13, 14, 15, 23, 24, 32, 38, 39, 40.
setlength: 2, 9, 13, 22.
shape_file: 3, 4, 5, 44.
shape_name: 3, 4.
show_basics: 1, 2, 3.
show_choices: 1, 3, 25, 32, 37, 48.
show_choices_gap: 3, 4, 38, 39, 40, 44.
show_choices_max: 3, 4, 38, 39, 40, 44, 48.
show_details: 1, 3, 26, 38, 39, 40, 44.
show_final_weights: 1, 2.
show_full_state: 1, 34.
show_levels_max: 3, 4, 50, 51.
show_max_deg: 1, 2.
show_profile: 1, 2, 32.
show_record_weights: 1, 41.
show_tots: 1, 2.
show_warnings: 1, 3, 19.
show_weight_bumps: 1, 41.
showpos: 12.
size: 8, 9, 12, 13, 14, 20, 21, 22, 23, 28, 38, 39, 40, 44, 46, 47.
slack: 8, 9, 13, 15, 23, 25, 38, 39, 40, 44.
spacing: 3, 4, 45.
spr: 8, 19.
ss: 37, 38, 39, 40.
sscanf: 4.
stage: 31, 32, 35, 36, 37, 45, 46, 47, 48.
stagelevel: 31, 36, 50, 51.
start: 49.
start_name: 16.
stderr: 1, 2, 3, 4, 6, 12, 13, 14, 15, 19, 25, 26, 27, 28, 32, 34, 35, 36, 37, 38, 39, 40, 41, 43, 44, 46, 48, 49, 50, 51, 52.
stdin: 15, 19.
stdout: 45.
stop: 49.
str: 10, 11, 16, 17, 19.
stream: 11, 12.
stringbuf: 10.
strlen: 15, 19.
subroutine_overhead: 1, 37.
t: 2, 41.
tetrabyte: 8, 9.
threeints: 2, 10, 33.
thresh: 3, 4, 34.
timeout: 3, 4, 34.
tmems: 3, 41, 44.
tough_itm: 33, 37, 38, 39, 41.
tryagain: 32.
twoints: 10.
uint: 2.
ullng: 2, 3, 31.
updates: 2, 3, 38, 39, 40.
vbose: 1, 2, 3, 4, 19, 25, 26, 32, 34, 37, 38, 39, 40, 41, 44, 48.
wmin: 41.
wt: 8, 13, 23, 41, 43.
w0: 3, 23, 41, 43.
x: 12, 14.

⟨ Adjust *nd* 24 ⟩ Used in section 19.
⟨ Check for duplicate item name 18 ⟩ Used in section 16.
⟨ Close the files 5 ⟩ Used in section 2.
⟨ Convert the prefix to an integer, q 17 ⟩ Used in section 16.
⟨ Cover or commit item *ii*, decrease bound of item *ii* if primary, potentially deactivate it, or return 0 38 ⟩ Used in section 37.
⟨ Create a node for the item named in *buf*[p] 20 ⟩ Used in section 19.
⟨ Do a forced move and **goto** *advance* 48 ⟩ Used in sections 32 and 42.
⟨ Do special things if enough *mems* have accumulated 34 ⟩ Used in section 32.
⟨ Expand *set* 23 ⟩ Used in section 19.
⟨ Global variables 3, 9, 31, 33 ⟩ Used in section 2.
⟨ Increase the weight of *tough_itm* 41 ⟩
⟨ Increase *level* 36 ⟩ Used in section 32.
⟨ Increase *stage* 35 ⟩ Used in section 32.
⟨ Initialize *item* 22 ⟩ Used in section 19.
⟨ Input the item names 15 ⟩ Used in section 2.
⟨ Input the options 19 ⟩ Used in section 2.
⟨ Make optionless items invisible 26 ⟩ Used in section 19.
⟨ Maybe do a forced move 42 ⟩ Used in section 32.
⟨ Print the profile 52 ⟩ Used in section 2.
⟨ Process the command line 4 ⟩ Used in section 2.
⟨ Randomize the *item* list 29 ⟩ Used in section 2.
⟨ Remove the option *cur_choice* 40 ⟩ Used in section 32.
⟨ Remove *last_node* from its item list 21 ⟩ Used in section 19.
⟨ Remove *optp* from its other sets, or return 0 39 ⟩ Used in section 38.
⟨ Report an uncoverable item 25 ⟩ Used in section 2.
⟨ Report the item totals 28 ⟩ Used in section 2.
⟨ Report the successful completion of the input phase 27 ⟩ Used in section 2.
⟨ Restore the currently active items and their sizes and bounds 47 ⟩ Used in section 32.
⟨ Save the currently active items and their sizes and bounds 46 ⟩ Used in section 32.
⟨ Scan an item name, possibly prefixed by bounds 16 ⟩ Used in section 15.
⟨ Set *best_itm* to the best item for branching, and let *score* be its branching degree 44 ⟩ Used in section 32.
⟨ Solve the problem 32 ⟩ Used in section 2.
⟨ Subroutines 6, 11, 12, 13, 14, 37, 43, 49, 50, 51 ⟩ Used in section 2.
⟨ Type definitions 8, 10 ⟩ Used in section 2.
⟨ Visit a solution and **goto** *backup* 45 ⟩ Used in section 32.

SSMCC

	Section	Page
Intro	1	1
Data structures	7	5
Inputting the matrix	15	11
Binary branching versus d-way branching	30	17
The dancing	32	18
Index	53	30