1. Intro. This program is an experimental XCC solver, which often looks ahead considerably further than DLX2 does. More precisely, it maintains "domain consistency": An option O is eliminated when its use would cause some primary item $I \notin O$ to have no options remaining. In a sense, I'm performing the work of DLX-PRE repeatedly as the search proceeds. With luck, the total time will decrease, although the time per node is potentially much larger.

Furthermore, I'm continuing to experiment with sparse-set data structures, as I did in the similar program SSXCC1, which was inspired by Christine Solnon's XCC-WITH-DANCING-CELLS.

This program was in fact derived directly from SSXCC1, by adding further data structures and algorithms. I confess in advance that the concepts below might not be easy to grasp, because some of them are rather subtle, and they're just beginning to make sense to me as I put the pieces together. Hopefully all will become clear by the time I finish! I've retained the documentation of SSXCC1's features, but they too are admittedly intricate. So let's take a deep breath together. We can handle this.

The DLX input format used in previous solvers is adopted here, without change, so that fair comparisons can be made. (See the program DLX2 for definitions. Much of the code from that program is used to parse the input for this one.)

2. After this program finds all solutions, it normally prints their total number on *stderr*, together with statistics about how many nodes were in the search tree, and how many "updates" 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.)

Empirical tests show that this program takes more elapsed time per mem than most other programs that I've written. I don't know why. Perhaps it's because the number of "global registers" is unusually large.

Here is the overall structure:

```
#define o mems++     /* count one mem */
#define oo mems += 2     /* count two mems */
#define ooo mems += 3     /* count three mems */
#define 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 5000     /* at most this many levels in the search tree */
#define max_cols 10000     /* at most this many items */
#define max_nodes 1000000     /* at most this many nonzero elements in the matrix */
#define poolsize 10000000     /* at most this many entries in pool */
#define savesize 1000000     /* at most this many entries on savestack */
#define bufsize (9 * max_cols + 3)     /* a buffer big enough to hold all item names */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
  typedef unsigned int uint;     /* a convenient abbreviation */
  typedef unsigned long long ullng;     /* ditto */
  ⟨Type definitions 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, best_itm;
    ⟨Process the command line 4⟩;
    ⟨Input the item names 28⟩;
    ⟨Input the options 30⟩;
    if (vbose & show_basics) ⟨Report the successful completion of the input phase 37⟩;
    if (vbose & show_tots) ⟨Report the item totals 38⟩;
    imems = mems, mems = 0;
    if (baditem) ⟨Report an uncoverable item 36⟩
    else ⟨Solve the problem 46⟩;
  done: ⟨Say adieu 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 *m*th solution to be output (the default is `m0`, which merely counts them);
- '`d`⟨integer⟩' sets *delta*, which causes periodic state reports on *stderr* after the algorithm has performed approximately *delta* mems since the previous report (default 10000000000);
- '`c`⟨positive integer⟩' limits the levels on which choices are shown during verbose tracing;
- '`C`⟨positive integer⟩' limits the levels on which choices are shown in the periodic state reports;
- '`l`⟨nonnegative integer⟩' gives a *lower* limit, relative to the maximum level so far achieved, to the levels on which choices are shown during verbose tracing;
- '`t`⟨positive integer⟩' causes the program to stop after this many solutions have been found;
- '`T`⟨integer⟩' sets *timeout* (which causes abrupt termination if *mems* > *timeout* at the beginning of a level);
- '`S`⟨filename⟩' to output a "shape file" that encodes the search tree; '`x`⟨positive integer⟩' causes partial solutions of this many stages to be written to files, not actually explored; '`X`⟨filename⟩' to input and resume a partial solution.

```
#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 stats about choices */
#define show_purges 8      /* vbose code to show inconsistent options deleted */
#define show_supports 16      /* vbose code to show new supports */
#define show_option_counts 32      /* vbose code to count active options */
#define show_mstats 64      /* vbose code to show memory usage in key arrays */
#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 */
```

```
⟨ Global variables 3 ⟩ ≡
  int vbose = show_basics + show_warnings;      /* level of verbosity */
  int spacing;      /* solution k is output if k is a multiple of spacing */
  int show_choices_max = 1000000;      /* above this level, show_choices is ignored */
  int show_choices_gap = 1000000;      /* below level maxl − show_choices_gap, show_details is ignored */
  int show_levels_max = 1000000;      /* above this level, state reports stop */
  int maxl, maxs;      /* maximum level and stage actually reached */
  int xcutoff, xcount;      /* stage when partial solutions output, and their number */
  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;      /* 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 */
  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, 18, 62, and 82.

This code is used in section 2.

4. If an option appears more than once on the command line, the first appearance takes precedence.

```
⟨ Process the command line 4 ⟩ ≡
  for (j = argc − 1, k = 0; j; j−−)
    switch (argv[j][0]) {
    case 'v': k |= (sscanf(argv[j] + 1, ""O"d", &vbose) − 1); break;
    case 'm': k |= (sscanf(argv[j] + 1, ""O"d", &spacing) − 1); break;
    case 'd': k |= (sscanf(argv[j] + 1, ""O"lld", &delta) − 1), thresh = delta; break;
    case 'c': k |= (sscanf(argv[j] + 1, ""O"d", &show_choices_max) − 1); break;
    case 'C': k |= (sscanf(argv[j] + 1, ""O"d", &show_levels_max) − 1); break;
    case 'l': k |= (sscanf(argv[j] + 1, ""O"d", &show_choices_gap) − 1); break;
    case 't': k |= (sscanf(argv[j] + 1, ""O"lld", &maxcount) − 1); break;
    case 'T': k |= (sscanf(argv[j] + 1, ""O"lld", &timeout) − 1); break;
    case 'S': shape_name = argv[j] + 1, shape_file = fopen(shape_name, "w");
      if (¬shape_file)
        fprintf(stderr, "Sorry,␣I␣can't␣open␣file␣`"O"s'␣for␣writing!\n", shape_name);
      break;
    case 'x': k |= (sscanf(argv[j] + 1, ""O"d", &xcutoff) − 1); break;
    case 'X': ⟨ Open xcutoff_file for reading, and break 79 ⟩;
    default: k = 1;     /* unrecognized command-line option */
    }
  if (k) {
    fprintf(stderr, "Usage:␣"O"s␣[v<n>]␣[m<n>]␣[d<n>]""␣[c<n>]␣[C<n>]␣[l<n\
        >]␣[t<n>]␣[T<n>]␣[S<bar>]␣[x<n>]␣[X<bar>]␣foo.dlx\n", argv[0]);
    exit(−1);
  }
```

This code is used in section 2.

5. I don't report the memory used for *deg*, *levelstage*, and *profile*, because they are only for documentation, not part of the search process.

```
⟨Say adieu 5⟩ ≡
  if (vbose & show_profile) ⟨Print the profile 76⟩;
  if (vbose & show_max_deg) fprintf(stderr, "The␣maximum␣branching␣degree␣was␣"O"d.\n", maxdeg);
  if (vbose & show_basics) {
    fprintf(stderr, "Altogether␣"O"llu␣solution"O"s,␣"O"llu+"O"llu␣mems,", count,
        count ≡ 1 ? "" : "s", imems, mems);
    bytes = (itemlength + setlength) ∗ sizeof(int) + last_node ∗ sizeof
        (node) + (4 ∗ maxs + maxl) ∗ sizeof(int) + maxsaveptr ∗ sizeof (twoints) + poolptr ∗ sizeof
        (twoints);
    fprintf(stderr, "␣"O"llu␣updates,␣"O"llu␣bytes,␣"O"llu␣nodes.\n", updates, bytes, nodes);
  }
  if (vbose & show_mstats) {
    fprintf(stderr, "␣itemlength="O"d,␣setlength="O"d,␣last_node="O"d;\n", itemlength, setlength,
        last_node);
    fprintf(stderr, "␣maxsaveptr="O"d,␣poolptr="O"d,␣maxstage="O"d,␣maxlevel="O"d.\n",
        maxsaveptr, poolptr, maxs, maxl);
  }
  if (sanity_checking) fprintf(stderr, "sanity_checking␣was␣on!\n");
  if (leak_checking) fprintf(stderr, "leak_checking␣was␣on!\n");
  if (shape_file) fclose(shape_file);
  if (xcount) ⟨Report the number of partial solutions output 81⟩;
```

This code is used in section 2.

6. Here's a subroutine for use in debugging, but I hope it's never invoked.

```
⟨Subroutines 6⟩ ≡
  void confusion(char ∗id)
  {     /∗ an assertion has failed ∗/
    fprintf(stderr, "trouble␣after␣"O"lld␣mems,␣"O"lld␣nodes:␣%s!\n", mems, nodes, id);
  }
```

See also sections 11, 12, 13, 14, 15, 19, 20, 21, 22, 23, 24, 39, 40, 44, 45, 66, 73, 74, and 75.

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 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.

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 four fields: *itm*, *loc*, *clr*, and *xtra*. 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 *xtra* field has special uses as we maintain domain consistency, as explained later.

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

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

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

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

The *set* array contains also the item names, as well as two fields $mark(x)$ and $match(x)$ that are used for compatibility checking. (The *match* field is present only in secondary items.)

We count one mem for a simultaneous access to the *itm* and *loc* fields of a node, also one for simultaneous access to *clr* and *xtra*.

```
#define size(x) set[(x) − 1]    /* number of active options of the kth item, x */
#define pos(x) set[(x) − 2]    /* where that item is found in the item array */
#define lname(x) set[(x) − 4]    /* the first four bytes of x's name */
#define rname(x) set[(x) − 3]    /* the last four bytes of x's name */
#define mark(x) set[(x) − 5]    /* a stamp for incompatible items */
#define match(x) set[(x) − 6]    /* a required color in compatibility tests */
```

⟨ 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 xtra;    /* used for special purposes (see below) */
  } node;
```

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 */
int set[max_nodes + 6 * max_cols];   /* the sets of active options for active items */
int itemlength;   /* number of elements used in item */
int setlength;   /* number of elements used in set */
int active;   /* current number of active items */
int oactive;   /* value of active before swapping out current-choice items */
int totopts;   /* current number of active options */
int baditem;   /* an item with no options, plus 1 */
int osecond;   /* setting of second just after initial input */
```

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. If *showid* $= 1$, it also prints the value of $opt - 1$, which should be the location of the spacer just preceding *opt*. Otherwise it optionally prints the position of the option in its item list.

```
⟨ Subroutines 6 ⟩ +≡
  void print_option(int opt, FILE *stream, int showpos, int showid)
  {
    register int k, q, x;
    x = nd[opt].itm;
    if (opt ≥ last_node ∨ x ≤ 0) {
      fprintf(stderr, "Illegal␣option␣"O"d!\n", opt);
      return;
    }
    if (showid) fprintf(stream, ""O"d␣‘", opt − 1);
    for (q = opt; ; ) {
      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 ≡ opt) break;
    }
    k = nd[q].loc;
    if (showid) fprintf(stream, "␣’");
    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, 0);
  }
  void propt(int opt)
  {      /* opt should be the spacer just before an option */
    if (nd[opt].itm ≥ 0) fprintf(stderr, ""O"d␣isn’t␣an␣option␣id!\n", opt);
    else print_option(opt + 1, stderr, 0, 1);
  }
```

13. The *print_option* routine has a sort of inverse, which reads from *buf* what purports to be the description of an option and verifies it.

```
⟨Subroutines 6⟩ +≡
  int read_option(void)
  {
    register int k, q, x, j, opt;
    for (opt = 0, k = 1; o, buf[k] ≥ '0' ∧ buf[k] ≤ '9'; k++) opt = 10 ∗ opt + buf[k] − '0';
    if ((o, buf[k] ≠ '␣') ∨ (o, buf[k + 1] ≠ '`') ∨ (o, buf[k + 2] ≠ '␣')) return −1;
    for (k += 3, q = opt + 1; o, (x = nd[q].itm) > 0; q++) {
      oo, namebuf.lr.l = lname(x), namebuf.lr.r = rname(x);
      for (j = 0; j < 8; j++) {
        if (¬namebuf.str[j]) break;
        if (o, namebuf.str[j] ≠ buf[k + j]) return −1;
      }
      k += j;    /∗ we've verified the item name ∗/
      if (o, nd[q].clr) {
        if ((o, buf[k] ≠ ':') ∨ (o, (unsigned char) buf[k + 1] ≠ nd[q].clr)) return −1;
        k += 2;
      }
      if (o, buf[k++] ≠ '␣') return −1;
    }
    if (buf[k] ≠ '\'') return −1;
    return opt + 1;
  }
```

14. 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 < 5 ∨ 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:\n", pos(c) + 1, active, size(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]);
  }
```

15. 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) {
            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;
          }
        }
      }
    }
  }
```

16. Domain consistency. The data structures above were fine for SSXCC1, but this program aims to prune its search tree by maintaining "domain consistency." Several more things are therefore needed.

We regard the given XCC problem as a special case of the general binary CSP, in which the variables are the primary items. The domain of primary item p is the set of options that contain p. And there's a constraint between each pair of primary items p and p': Option o for p is allowed together with option o' for p' if and only if o and o' are *compatible*, in the sense that they're either equal or they have no items in common, except for secondary items with identical nonzero colors.

What does domain consistency mean in this context? "For every $p \neq p'$ and every o in the domain of p, there's a compatible option o' in the domain of p'." Stating this another way, suppose o is an option. Then the action of choosing o, in order to "cover" its primary items, must not "wipe out" the domain of any primary item that's not in o.

When an option doesn't meet this criterion, we remove it from consideration, thus simplifying the problem. The removal of an option also makes other options potentially removable. Eventually we either remove the last option from some item's domain, in which case there's no solution, or we reach a stable situation where all domains are nonempty and consistent. In the latter case, we'll choose an option, for an item that has comparatively few of them, and we'll recursively explore the consequences of either using that option or not.

To maintain domain consistency we shall combine the ideas of Christian Bessière's algorithm AC-6 [*Artificial Intelligence* **65** (1994), 179–190] with Christophe Lecoutre and Fred Hemery's algorithm AC3rm [*IJCAI Proceedings* **20** (2007), 125–130], by maintaining a table of *supports*: This program will essentially construct an array $S[o,p]$, with an element for every option o and every primary item p, such that $S[o,p]$ is equal to o' for some compatible option o' such that $p \in o'$, whenever $p \notin o$; and $S[o,p] = \#$ when $p \in o$. This array provides witnesses to the fact that the current domains are indeed consistent.

17. We don't, however, actually represent the support array S directly. Instead, we represent the inverse function: For each option o', we maintain a list of all the pairs (o,p) such that $S[o,p] = o'$. This list is called the *trigger list* of o', because we use it to maintain the support conditions: If option o' is removed for any reason, thereby leaving one or more holes in the S array, the removal will trigger a series of events that will refill those holes, one by one.

Each option o also has a *fixit list*, containing all pairs (o',p) for which the event (o,p) has been triggered by o' but the corresponding hole hasn't yet been refilled.

There's also a queue Q, containing all the options o for which at least one hole currently exists.

All of these lists — the triggers, the fixits, and the queue — are singly linked, in a array called *pool*, whose elements have two fields called *info* and *link* in familiar fashion. The trigger lists and fixit lists are stacks (last-in-first-out); the queue is first-in-first-out.

18. Internally, an option o is represented by the index of the spacer just preceding that option in *nd*. An item i, whether primary or secondary, is represented by the index where the main data for I appears in *set*. A link is represented by its index in *pool*.

Variables *qfront* and *qrear* are the indices of the front and rear of Q. More precisely, *qfront* points to the front element, the node that will be removed first; *qrear* points to a "blank" node that follows the element that will be removed last. The queue is empty if and only if *qfront* = *qrear*. The contents of *info*(*qrear*) and *link*(*qrear*) are both irrelevant; they will be filled in when a new element is enqueued and a new blank element is appended.

Fortunately there's room enough in the existing data structures of program SSXCC1 to store the two pointers that we need for each option: The top of o's trigger stack, called *trigger*(o), is kept in location *nd*[o].*clr*; and the top of o's fixit stack, called *fixit*(o), is kept in *nd*[o].*xtra*. We have *fixit*(o) = 0 if and only if o is not in the queue.

We'll see later than every inactive option has an *age*, indicating when it was purged from the current partial solution. This value, *age*(o) appears in *nd*[$o+1$].*xtra*.

(Kludge note: With these conventions, all of an option's dynamic data has been squeezed into the three otherwise unused fields *nd*[o].*clr*, *nd*[o].*xtra*, and *nd*[$o+1$].*xtra*. If another special datum had been needed, I could have put it into *nd*[$o+1$].*clr*, because this program ensures that every option begins with a primary item.)

```
#define info(p) pool[p].l
#define link(p) pool[p].r
#define trigger(opt) nd[opt].clr    /* beginning of the trigger stack */
#define fixit(opt) nd[opt].xtra    /* beginning of the fixit stack */
#define age(opt) nd[opt + 1].xtra    /* when was this option last purged? */
#define stamp(opt) nd[(opt) + 1].xtra
```

```
⟨ Global variables 3 ⟩ +≡
  twoints pool[poolsize];    /* where the linked lists live */
  int poolptr = 1;    /* the first unused cell of pool */
  int qfront, qrear;    /* the front and rear of Q */
  int curstamp;    /* the current "time stamp" */
  int biggeststamp;    /* the largest time stamp used so far */
  int compatstamp;    /* another stamp, used for compatibility tests */
```

19. A few basic primitive routines undergird all of our list processing.

(When counting mems here, we consider *avail* and *poolptr* to be in global registers. The compiler could inline this code, so I don't count any overhead for these subroutine calls.)

```
#define avail pool[0].r    /* head of the stack of available cells */
⟨Subroutines 6⟩ +≡
  int getavail(void)
  {    /* return a pointer to an unused cell */
    register int p;
    p = avail;
    if (p) {
      o, avail = link(p);
      return p;    /* info(p) might be anything */
    }
    if (poolptr++ ≥ poolsize) {
      fprintf(stderr, "Pool␣overflow␣(poolsize="O"d)!\n", poolsize);
      exit(−7);
    }
    return poolptr − 1;
  }
  void putavail(int p)
  {    /* free the single cell p */
    o, link(p) = avail;
    avail = p;
  }
```

20. Entries of a trigger list are pairs (o, p), with the cell that mentions option o linking to the cell that mentions primary item p.

```
⟨Subroutines 6⟩ +≡
  void print_trigger(int opt)
  {
    register int p, q;
    fprintf(stderr, "trigger␣stack␣for␣option␣", opt);
    print_option(opt + 1, stderr, 0, 1);
    for (p = trigger(opt); p; p = link(q)) {
      q = link(p);
      fprintf(stderr, "␣");
      if (info(p) ≥ 0) {
        print_option(info(p) + 1, stderr, −1, 1);
        fprintf(stderr, ",");
        p = link(p);
        print_item_name(info(p), stderr);
      } else ⟨Print a trigger hint 57⟩;
      fprintf(stderr, "\n");
    }
  }
```

21. ⟨Subroutines 6⟩ +≡

```
void print_triggers(int all)
{
  register int opt, jj, optp;
  for (opt = 0; opt < last_node; opt += nd[opt].loc + 1) {
    if (¬all) {
      jj = nd[opt + 1].itm;    /* jj is opt's first item */
      if (nd[opt + 1].loc ≥ jj + size(jj)) continue;    /* is opt in jj's set? */
    }
    print_trigger(opt);
  }
}
```

22. Entries of a fixit list are pairs (o, p), with the cell that mentions option o linking to the cell that mentions primary item p.

⟨Subroutines 6⟩ +≡

```
void print_fixit(int opt)
{
  register int p, q;
  fprintf(stderr, "fixit␣stack␣for␣option␣", opt);
  print_option(opt + 1, stderr, −1, 1);
  fprintf(stderr, ":");
  for (p = fixit(opt); p; p = link(q)) {
    q = link(p);
    fprintf(stderr, "␣");
    print_item_name(info(q), stderr);
    fprintf(stderr, "["O"d]", info(p));
  }
  fprintf(stderr, "\n");
}
```

23. ⟨Subroutines 6⟩ +≡

```
void print_queue(void)
{
  register int p;
  for (p = qfront; p ≠ qrear; p = link(p)) print_option(info(p) + 1, stderr, 0, 1);
}
```

24. Linked lists are wonderful; but a single weak link can cause a catastrophic error. Therefore, when debugging, I want to be extra sure that this program doesn't make any silly errors when it uses pool pointers.

Furthermore, since I'm doing my own garbage collection, I want to avoid any "memory leaks" that would occur when I've forgotten to recycle a no-longer-used entry of the *pool*.

The *list_check* routine laboriously goes through everything and makes sure that every cell less than *poolptr* currently has one and only one use.

```
#define leak_checking 0     /* set this nonzero if you suspect linked-list bugs */
#define signbit #8000000
#define vet_and_set(l)
        { if ((l) ≤ 0 ∨ (l) ≥ poolptr) {
            fprintf(stderr, "Bad␣link␣"O"d!\n", l);
            return;
          }
          if (link(l) & signbit) {
            fprintf(stderr, "Double␣link␣"O"d!\n", l);
            return;
          }
          link(l) ⊕= signbit;
        }
⟨Subroutines 6⟩ +≡
  void list_check(int count)
  {
    register int p, t, opt;

    for (t = 0, p = avail; p; t++, p = signbit ⊕ link(p)) vet_and_set(p);
    if (count) fprintf(stderr, "avail␣size␣"O"d\n", t);
    for (opt = 0; opt < last_node; opt += nd[opt].loc + 1) {
      for (p = trigger(opt); p; p = signbit ⊕ link(p)) vet_and_set(p);
      for (p = fixit(opt); p; p = signbit ⊕ link(p)) vet_and_set(p);
    }
    for (p = qfront; ; p = signbit ⊕ link(p)) {
      vet_and_set(p);
      if (p ≡ qrear) break;
    }
    for (p = 1; p < poolptr; p++) {
      if (link(p) & signbit) link(p) ⊕= signbit;
      else fprintf(stderr, "Lost␣cell␣"O"d!\n", p);
    }
  }
```

25. One of our main activities is to find options O' that are compatible with a given option O. We do this by marking each item I of O with *compatstamp*, and also recording I's color if I is secondary. Then, given a candidate O', we can easily spot incompatibility.

That idea works when $I \in O$ is equivalent to $mark(I) \equiv compatstamp$. So we start with all *mark* fields equal to zero; and we increase *compatstamp* by 1 whenever starting this process with a new O.

But there's a hitch: If this testing is done 2^{32} times, *compatstamp* will "wrap around" to zero, and our test might be invalid. In such a case we can still guarantee success if we take the trouble to zero out all the *mark* fields again.

```
#define badstamp 0    /* set this to 3, say, when initially debugging */
⟨Bump compatstamp 25⟩ ≡
  if (++compatstamp ≡ badstamp) {
    for (ii = 0; ii < itemlength; ii++) oo, mark(item[ii]) = 0;
    compatstamp = 1;
  }
```

This code is used in section 26.

26. ⟨Prepare the *mark* fields for testing compatibility with *opt* 26⟩ ≡

```
  ⟨Bump compatstamp 25⟩;
  for (nn = opt + 1; o, (ii = nd[nn].itm) > 0; nn++) {
    o, mark(ii) = compatstamp;
    if (ii ≥ second) {
      if (o, nd[nn].clr) o, match(ii) = nd[nn].clr;
      else o, match(ii) = −1;    /* this won't match any color */
    }
  }
```

This code is used in sections 44 and 45.

27. At the beginning of this section, $nd[optp].itm$ is an item ii in the middle of some option O'. If O' is compatible with *opt*, we want to reset *optp* so that $nd[optp]$ is the spacer preceding O'. We use the fact that ii isn't present in *opt*.

```
⟨If optp is compatible with opt, break 27⟩ ≡
  for (qq = optp, nn = qq + 1; nn ≠ qq; nn++) {
    if (o, (jj = nd[nn].itm) ≤ 0) optp = nn + jj − 1, nn = optp;    /* nn is a spacer */
    else if (o, mark(jj) ≡ compatstamp) {    /* watch out, jj is in opt */
      if (jj < second ∨ (o, nd[nn].clr ≡ 0) ∨ (o, nd[nn].clr ≠ match(jj))) break;    /* incompatible */
    }
  }
  if (nn ≡ qq) break;    /* not incompatible */
```

This code is used in sections 44 and 45.

28. 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 28⟩ ≡
  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 29⟩;
    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.

29. ⟨Check for duplicate item name 29⟩ ≡

```
  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 28.

30. ⟨ Input the options 30 ⟩ ≡

```
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] 31 ⟩;
    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 32 ⟩;
      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;
  }
}
⟨ Initialize item 33 ⟩;
⟨ Expand set 34 ⟩;
⟨ Adjust nd 35 ⟩;
```

This code is used in section 2.

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

This program shifts the items of an option, if necessary, so that the very first item is always primary. In other words, secondary items that precede the first primary item are actually stored in $nd[last\_node+1]$.

```
⟨Create a node for the item named in buf[p] 31⟩ ≡
  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 + 1 ≥ max_nodes) panic("Too␣many␣nodes");
  o, t = size(k);      /* how many previous options have used this item? */
  if (¬pp) {      /* no primary items seen yet */
    if ((k ≫ 2) < second) o, pp = 1, nd[i + 1].itm = k ≫ 2, nd[i + 1].loc = t;
    else o, nd[last_node + 1].itm = k ≫ 2, nd[last_node + 1].loc = t;
  } else o, nd[last_node].itm = k ≫ 2, nd[last_node].loc = t;
  o, size(k) = t + 1, pos(k) = last_node;
```

This code is used in section 30.

32. ⟨Remove *last_node* from its item list 32⟩ ≡

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

This code is used in section 30.

33. ⟨Initialize *item* 33⟩ ≡

```
  active = itemlength = last_itm − 1;
  if (second ≡ max_cols) osecond = active;
  else osecond = second − 1;
  for (k = 0, j = 5; k < itemlength; k++)
    oo, item[k] = j, j += (k + 1 < osecond ? 5 : 6) + size((k + 1) ≪ 2);
  setlength = j − 5;
  if (osecond ≡ active) second = setlength;      /* no secondary items */
```

This code is used in section 30.

34. 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 34⟩ ≡
  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);
    o, mark(j) = 0;
  }
```

This code is used in section 30.

35. ⟨ Adjust *nd* 35 ⟩ ≡

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

This code is used in section 30.

36. ⟨ Report an uncoverable item 36 ⟩ ≡

```
{
    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.

37. 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 37 ⟩ ≡

```
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.

38. 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 38 ⟩ ≡

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

This code is used in section 2.

39. Maintaining supports. It's time now to implement some of the mechanisms used for the "virtual support array S" described earlier.

First, let's see what happens when an option goes away. The value returned is 0 if this was the final option for some primary item. Otherwise the option's trigger list will enqueue fixits, to provide replacements for any supports that are no longer valid.

When *purge_option* purges an option, it sets that option's "age" to *cur_age*, which measures our progress to a complete solution. Options that are purged early, on the basis of fewer assumptions, are "younger" than options that are purged later.

We'll see later that a trigger list may contain hints about the ages of its entries. Such hints are signalled by negative entries.

The *purge_option* procedure rearranges the entries of a long trigger list by carrying out a "bucket sort," which puts the youngest remaining entries last. This sorting process uses auxiliary arrays *trig_head* and *trig_tail*; *trig_head* is assumed to be zero upon entry and exit.

```
#define infinite_age (2 * max_stage + 2)
⟨Subroutines 6⟩ +≡
  int purge_option(int opt, int act)
  {
    register int ii, jj, nn, nnp, p, q, qq, pp, ss, t, optp, cutoff, tmin = infinite_age;
    subroutine_overhead;
    if (vbose & show_purges) {
      fprintf(stderr, "␣"O"d."O"c␣purging␣option␣", cur_age >> 1, cur_age & 1 ? '5' : '0', opt);
      print_option(opt + 1, stderr, 0, 1);
    }
    ⟨Delete opt from the sets of all its unpurified items, possibly returning 0 41⟩;
    o, age(opt) = cur_age;
    for (o, p = trigger(opt), pp = 0; p; p = pp) {
      o, optp = info(p), q = link(p);
      o, ii = info(q), pp = link(q);
      if (optp < 0) ⟨If all remaining triggers are known to be inactive, set pp = p and break; otherwise
            discard this hint and continue 53⟩;
      ⟨If optp has been purged, set t to its age and goto inactive 54⟩;
      ⟨If ii isn't active, set t = cur_age and goto inactive 55⟩;
      ooo, info(p) = opt, link(q) = fixit(optp);     /* change trigger to fixit */
      if (¬fixit(optp)) {     /* we should enqueue optp */
        o, link(qrear) = getavail(), info(qrear) = optp, qrear = link(qrear);
        o, age(optp) = infinite_age;
      }
      o, fixit(optp) = p;
      continue;
    inactive: if (o, trig_head[t] ≡ 0) o, trig_tail[t] = q;
      oo, link(q) = trig_head[t], trig_head[t] = p;     /* move trigger to temp list t */
      if (t < tmin) tmin = t;
    }
    ⟨Replace trigger(opt) by its unused entries, reordered and hinted 56⟩;
    totopts−−;
    return 1;
  }
```

40. ⟨Subroutines 6⟩ +≡

```
int purge_the_option(register int opt, int act)
{    /* opt isn't at the left spacer */
  for (opt--; o, nd[opt].itm > 0; opt--) ;
  return purge_option(opt, act);
}
```

41. After a secondary item has been purified, we mustn't mess with its set. Secondary items that lie between *active* and the parameter *act* are in the process of being purified.

⟨Delete *opt* from the sets of all its unpurified items, possibly returning 0 41⟩ ≡

```
  for (nn = opt + 1; o, (ii = nd[nn].itm) > 0; nn++) {
    p = nd[nn].loc;
    if (p ≥ second ∧ (o, pos(ii) ≥ act)) continue;    /* ii already purified */
    o, ss = size(ii) − 1;
    if (ss ≡ 0 ∧ p < second) {    /* oops: opt was item ii's only surviving option */
      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");
      }
      ⟨Clear the queue and return 0 42⟩;
    }
    o, nnp = set[ii + ss];
    o, size(ii) = ss;
    oo, set[ii + ss] = nn, set[p] = nnp;
    oo, nd[nn].loc = ii + ss, nd[nnp].loc = p;
    updates++;
  }
```

This code is used in section 39.

42. We can't complete the current options to a viable set that's domain consistent. So all of the fixit lists remaining in the queue must go back into the trigger lists that triggered them.

⟨Clear the queue and **return** 0 42⟩ ≡

```
  while (qfront ≠ qrear) {
    o, p = qfront, opt = info(p), qfront = link(p), putavail(p);
    ⟨Change the entries of fixit(opt) back to triggers 43⟩;
  }
  return 0;
```

This code is used in section 41.

43. ⟨Change the entries of *fixit*(*opt*) back to triggers 43⟩ ≡

```
  {
    for (o, p = fixit(opt); p; p = pp) {
      oo, optp = info(p), q = link(p), info(p) = opt;
      o, pp = link(q);    /* info(q) is the same for triggers and fixits */
      ooo, info(p) = opt, link(q) = trigger(optp), trigger(optp) = p;
    }
    o, fixit(opt) = 0;
  }
```

This code is used in sections 42, 44, and 60.

44. ⟨ Subroutines 6 ⟩ +≡

```
int empty_the_queue(void)
{
  register int p, q, pp, qq, s, ss, ii, jj, nn, opt, optp;
  subroutine_overhead;
  while (qfront ≠ qrear) {
    o, p = qfront, opt = info(p), qfront = link(p), putavail(p);
    if (fixit(opt) ≡ 0) confusion("queue");
    if (leak_checking) list_check(0);
    ⟨ If opt is no longer active, revert its fixit list and continue 60 ⟩;
    ⟨ Prepare the mark fields for testing compatibility with opt 26 ⟩;
    for (o, p = fixit(opt); p; p = pp) {
      o, q = link(p);    /* ignore info(p), which is irrelevant for now */
      o, ii = info(q), pp = link(q);    /* ii is a primary item, not in opt */
      for (o, s = ii, ss = s + size(ii); s < ss; s++) {
        o, optp = set[s];
        if (optp ≡ opt) confusion("fixit");
        ⟨ If optp is compatible with opt, break 27 ⟩;
      }
      if (s ≡ ss) {    /* opt is inconsistent */
        if (vbose & show_supports) {
          print_option(opt + 1, stderr, −1, 1);
          fprintf(stderr, ",");
          print_item_name(ii, stderr);
          fprintf(stderr, "␣not␣supported\n");
        }
        fixit(opt) = p;
        ⟨ Change the entries of fixit(opt) back to triggers 43 ⟩;
        if (¬purge_option(opt, active)) return 0;
        break;    /* move to another opt */
      } else ⟨ Record optp as the support for opt and ii 47 ⟩;
    }
    o, fixit(opt) = 0;
  }
  return 1;
}
```

45. Here's how we get the ball rolling by making every domain consistent in the first place.
At the beginning, all *stamp* fields are zero, and so is *curstamp*.

```
⟨Subroutines 6⟩ +≡
  int establish_dc(void)
  {
    register int k, ii, jj, nn, opt, optp, p, q, qq, s, ss;
    cur_age = 1;    /* "age 1/2," see below */
    qfront = qrear = getavail();
    for (opt = 0; opt < last_node; o, opt += nd[opt].loc + 1) {
      if (leak_checking) list_check(0);
      ⟨Prepare the mark fields for testing compatibility with opt 26⟩;
      for (k = 0; k < osecond; k++) {
        o, ii = item[k];
        if (o, mark(ii) ≠ compatstamp) {    /* ii not in opt */
          for (o, s = ii, ss = s + size(ii); s < ss; s++) {
            o, optp = set[s];
            ⟨If optp is compatible with opt, break 27⟩;
          }
          if (s ≡ ss) {    /* opt is inconsistent */
            if (vbose & show_supports) {
              print_option(opt + 1, stderr, −1, 1);
              fprintf(stderr, ",");
              print_item_name(ii, stderr);
              fprintf(stderr, "␣not␣supported\n");
            }
            if (¬purge_option(opt, active)) return 0;
            break;    /* move to the next opt */
          } else {
            p = getavail(), q = getavail();
            o, link(p) = q;
            o, info(q) = ii;
            ⟨Record optp as the support for opt and ii 47⟩;
          }
        }
      }
    }
    return empty_the_queue();
  }
```

46.
```
⟨Solve the problem 46⟩ ≡
  totopts = options;
  if (¬establish_dc()) {
    if (vbose & show_choices) fprintf(stderr, "Inconsistent␣options!\n");
    goto done;
  }
  if (vbose & show_choices) fprintf(stderr, "Initial␣consistency␣after␣"O"lld␣mems.\n", mems);
  ⟨Do a backtrack search, maintaining domain consistency 61⟩;
```
This code is used in section 2.

47. ⟨ Record *optp* as the support for *opt* and *ii* 47 ⟩ ≡

```
{
  if (vbose & show_supports) {
    print_option(opt + 1, stderr, −1, 1);
    fprintf(stderr, ",");
    print_item_name(ii, stderr);
    fprintf(stderr, "␣supported␣by␣");
    print_option(optp + 1, stderr, 0, 1);
  }
  o, info(p) = opt;
  oo, link(q) = trigger(optp);
  o, trigger(optp) = p;
}
```

This code is used in sections 44 and 45.

48. A view from the top. 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, in the following way: First we try using the first option in that item's set; then we explore the consequences of *forbidding* that item.

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 less-deep levels.

The basic operation is "covering" each item of a chosen option. Covering means to make an item inactive. If it is primary, we remove it from the set of items needing to be covered, and we purge all other options that contain it. If the item is secondary and still active (not yet purified), we purge all options in which it has the wrong color.

The branching discipline that we follow is quite different from what we did in DLX2 or SSXCC1, however, because we're now maintaining domain consistency throughout the search. The old way was to choose a "best item" p, having say d options, and then to try option 1 of the d possibilities for p, then option 2 of those d, ..., option d of those d, before backtracking to the previous level.

The new way, given consistent domains, starts out the same as before. We choose a best item p_1, having d_1 options, and we try its first option. But after returning from that branch, we purge that option and restore domain consistency; then we choose a new best item p_2, having d_2 options, and try the first of those. Eventually, after trying and purging the first remaining options of p_1 through p_k, we'll reach a point where we can't make the remaining domains both consistent and nonempty. That's when we back up.

In this scenario, all of the subproblems for p_1, ..., p_k are trying to extend the same partial solution with s choices to a partial solution that has $s + 1$ choices. We call this "stage s" of the search. Stage s actually involves k different nodes of the (binary) search tree, each of which is on its own "level." (The level is the distance from the root; the stage is the number of options that have been chosen in the current partial solution.)

We might think of the search as a tree that makes a k-way branch at stage s, instead of as a tree that makes binary branches at each level. Such an interpretation is equivalent to the "natural correspondence" between ordinary trees and binary trees, discussed in *TAOCP* Section 2.3.2.

49. As search proceeds, the current subproblem gets easier and easier as the number of active items and options gets smaller and smaller. Let I_s be the set of all items that are active when s options $c_1, \ldots, c_s$ have been chosen to be in the partial solution-so-far. Thus I_0 is the set of all items initially given; and I_s for $s > 0$ is obtained from I_{s-1} by removing the primary items and the previously unpurified secondary items of c_s. We denote the primary items of I_s by P_s; these are the primary items not in $c_1, \ldots, c_s$.

Let O_0 be the set of all options actually given. Just before entering stage 1, we reduce O_0 to O_1^{init}, the largest subset of O_0 that is domain consistent, by purging options that have no support. In general, stage s for $s > 0$ begins with a domain-consistent set of options O_s^{init}. Later on in stage s we usually work with a smaller set of active options O_s, which is the largest domain-consistent set that's contained in O_s^{init} after we've removed the options whose consequences as potential choices were previously examined in this stage.

If every item in P_{s-1} still belongs to at least one option of O_s, we're ready to make a new c_s from among those remaining options. We get O_{s+1}^{init} from O_s by removing c_s and every option incompatible with it, and then by purging options that aren't domain-consistent.

Thus when we're in stage s, there's a sequence of sets of options

$$O_0 \supseteq O_1^{\text{init}} \supseteq O_1 \supseteq O_2^{\text{init}} \supseteq O_2 \supseteq \cdots \supseteq O_s^{\text{init}} \supseteq O_s,$$

all of which are domain consistent except possibly O_0. Notice that

$$\text{if } o \in O_s^{\text{init}} \text{ and } p \in O \text{ then } p \in P_{s-1}.$$

And there's good news: The support array $S[o,p]$ follows the nested structure of our search in a useful way. Recall that $S[o,p] = \#$ if $p \in o$; otherwise $S[o,p] = o'$, where $p \in o'$ and o' is compatible with o.

This array is defined for all options $o \in O_1^{\text{init}}$, and for all primary items $p \in P_0$. However, when we're in stage s, we're interested only in the much smaller subarray that contains supports when $o \in O_s$ and $p \in P_{s-1}$. And when we're transitioning from stage s to stage $s+1$, we care only about a still-smaller array subarray, for $o \in O_s$ and $p \in P_s$. In particular, domain consistency implies that we have

$$\begin{aligned}&\text{if } o \in O_s^{\text{init}} \text{ and } p \notin o \text{ and } p \in P_{s-1} \text{ then } S[o,p] \in O_s^{\text{init}};\\ &\text{if } o \in O_s \text{ and } p \notin o \text{ and } p \in P_s \text{ then } S[o,p] \in O_s.\end{aligned}$$

50. Eventually a choice will fail, of course. Backtracking becomes necessary in two distinct ways: (1) If we've settled on a new c_s among the options of O_s, but we're unable to reduce the remaining compatible options to a domain-consistent O_{s+1}^{init} without emptying some domain, we "backtrack in stage s" and reject that choice. (Thus, we stay in stage s but move to a new level; the active items remain the same.) (2) If we've finished exploring a choice from O_s and are unable to reduce the other options to a smaller domain-consistent O_s, we "backtrack to stage $s-1$" and reject c_{s-1}. (Thus, we resume where we left off at the previous stage's deepest level; the active items revert back from P_s to the larger set P_{s-1}.)

I wish I could say that it was easy for me to discover the programming logic just described. I guess it was my baptism into what researchers have called "fine-grained" versus "coarse-grained" algorithms.

Notice that when we backtrack, we need not change the S array in any way. A support is always a support. Thus there's no point in trying to undo any of the changes we've made to the current support structure.

51. The triggering. Suppose there are 1000 options and 100 items. Then the S array has 100,000 entries, most of which are supports (that is, not #). Every support is an entry in a trigger list; hence the trigger lists are necessarily long. The task of maintaining domain consistency might therefore seem hopelessly inefficient.

On the other hand, after we've made some choices, there may be only 100 options left, and perhaps 30 items not yet covered. Then at most 3000 supports are relevant, and most of the information in trigger lists is of no interest to us. An efficient scheme might therefore still be possible, if we can figure out a way to avoid looking at useless triggers.

Ideally we'd like options from O_s to appear at the top of each trigger stack, with options from $O_s^{\rm init}$ just below them, and with O_{s-1}, $O_{s-1}^{\rm init}$, $\ldots$, $O_1^{\rm init}$, O_0 furthest down. The pairs (o,p) of interest would then appear only near the top.

Unfortunately such an arrangement cannot be guaranteed. Indeed, that's obvious: The trigger-list entries occur in essentially arbitrary order when we first form $O_1^{\rm init}$. If they happen to be supports that work for every subsequent stage, no changes to the trigger lists will be needed, and we won't even want to look at those lists.

We can, however, come sort of close to an ideal arrangement, by exploiting the fact that every option not in the current O_s has been purged at least once. We look at *trigger*(o) only after o has been purged; and at that time we can reorder its entries.

Therefore this program inserts markers into the trigger lists, saying that "all further entries of this list are young" (meaning purged early, hence uninteresting until we've backtracked to an early stage). Every such marker is accompanied by a time stamp, so that we can recognize later when its message is no longer true.

52. When purging an option from $O_s^{\rm init}$ that won't be in O_s, the "current age" *cur_age* is s. And when purging an option from O_s that won't be in $O_{s+1}^{\rm init}$ it is $s+1/2$. (Inside the computer, of course, we want to work strictly with integers. So we internally double the true values, using $2s$ and $2s+1$.)

It turns out that the value of *cur_age* starts at 1/2 and goes through an interesting sequence as this computation proceeds: From a value $s+1/2$ it can go either to $s+3/2$ (beginning stage $s+1$) or to $s-1/2$ (backtracking within stage s). From a value s it can either go to $s-1$ (backtracking to a previous stage) or go to $s+1/2$ (beginning a new level within stage s).

Incidentally, I've tried to avoid making bad puns based on *cur_age* versus courage, or *age* versus *stage*.

53. A negative entry $optp = -c$ in a trigger list is a hint that all future entries will have at most age c. The search tree may have changed since this hint was put into the list; so we must look at the relevant stage stamp, to ensure that the hint is still valid.

Suppose o has age s. Then o is in $O_s^{\rm init}$ but not in O_s. As computation proceeds, without backtracking to stage $s-1$, the set O_s might get smaller and smaller, but o will still not be in O_s. Therefore a trigger hint saying that o is inactive will be valid until $stagestamp[s]$ changes.

Suppose o has age $s{+}1/2$. Then o is in O_s but not in $O_{s+1}^{\rm init}$. As computation proceeds, without backtracking in or to stage s, the set $O_{s+1}^{\rm init}$ won't change. Therefore a trigger hint saying that o is inactive will be valid until $stagestamp[s+1]$ changes.

That's why the following code says '$(cutoff+1) \gg 1$' when selecting the relevant stage stamp.

⟨If all remaining triggers are known to be inactive, set $pp = p$ and **break**; otherwise discard this hint and **continue** 53⟩ ≡

```
{
  $cutoff = -optp$;
  if ($cutoff < cur\_age \land (o, ii \equiv stagestamp[(cutoff+1) \gg 1])$) {
    $pp = p$;
    break;
  }
  $putavail(p), putavail(q)$;      /* discard an obsolete hint */
  continue;      /* and ignore it */
}
```

This code is used in section 39.

54. If $optp$ is inactive, it has been purged and its recorded age is cur_age or less. Thus we can conclude that $optp$ is active whenever $age(optp) > cur\_age$.

In general, of course, that age test won't be conclusive and a slightly more expensive test needs to be made by looking further into the data structures. Option $optp$ is active if and only if it appears in the current set of its first item. (This is where we use the fact that the first item of $optp$ is primary.)

⟨If $optp$ has been purged, set t to its age and **goto** *inactive* 54⟩ ≡

```
$o, t = age(optp)$;
if ($t \le cur\_age$) {
  $o, jj = nd[optp+1].itm$;      /* $jj$ is $optp$'s first item */
  if ($o, nd[optp+1].loc \ge jj + size(jj)$) goto inactive;
}      /* branch if $optp$ was removed from $jj$'s set */
```

This code is used in section 39.

55. When the trigger list for opt refers to an item ii, that item is in opt. Suppose ii is currently inactive; then we wouldn't be purging opt unless ii has just become inactive (and we're calling $purge\_option$ from within $include\_option$).

⟨If ii isn't active, set $t = cur\_age$ and **goto** *inactive* 55⟩ ≡

```
if ($o, pos(ii) \ge active$) {
  if ($pos(ii) \ge act$) confusion("active");
  $t = cur\_age$;
  goto inactive;
}
```

This code is used in section 39.

56. When we get here, *pp* is either zero or the cell where we found *cutoff*. In the latter case, $pp = p$ and $link(p) = q$; thus the cutoff hint is in p and q.

All of the unused trigger entries have been redirected to the *trig_head* lists, sorted by their age.

```
⟨Replace trigger(opt) by its unused entries, reordered and hinted 56⟩ ≡
  if (pp ≡ 0) cutoff = 0;
  if (tmin ≤ cutoff) {
    if (tmin < cutoff) confusion("trig");
    o, pp = link(q), putavail(p), putavail(q);      /* avoid double hint */
  }
  for (t = tmin; t < cur_age; t++)
    if (o, trig_head[t]) {
      oo, link(trig_tail[t]) = pp;
      o, p = getavail(), q = getavail(), link(p) = q;      /* make new hint */
      o, info(p) = −t;
      oo, info(q) = stagestamp[(t + 1) ≫ 1], link(q) = trig_head[t];
      o, trig_head[t] = 0;
      pp = p;
    }
  if (trig_head[cur_age]) {
    oo, link(trig_tail[cur_age]) = pp;      /* give no hint for inactive options of the current age */
    o, pp = trig_head[cur_age], trig_head[cur_age] = 0;
  }
  o, trigger(opt) = pp;
```

This code is used in section 39.

57.

```
⟨Print a trigger hint 57⟩ ≡
  fprintf(stderr, "cutoff␣for␣age␣"O"d."O"c", (−info(p)) ≫ 1, info(p) & 1 ? '5' : '0');
  if (info(q) ≠ stagestamp[(1 − info(p)) ≫ 1]) fprintf(stderr, "␣(obsolete)");
```

This code is used in section 20.

58. At this point we want *curstamp* to have a value that's larger from anything found in a trigger list hint. Moreover, the values of $stagestamp[1], \ldots, stagestamp[stage-1]$ should all be distinct and less than *curstamp*, because they might be used in future hints.

We may not be able to satisfy those conditions when *badstamp* is a small positive constant! But we will have checked out the following code at least once before failing.

```
⟨Bump curstamp 58⟩ ≡
  if (++biggeststamp ≡ badstamp) {
    if (badstamp > 0 ∧ stage > badstamp) {
      fprintf(stderr, "Timestamp␣overflow␣(badstamp="O"d)!\n", badstamp);
      exit(−11);
    }
    ⟨Remove all hints from all trigger lists 59⟩;
    for (k = 1; k < stage; k++) o, stagestamp[k] = k − 1;
    biggeststamp = k − 1;
  }
  curstamp = biggeststamp;
```

This code is used in section 63.

59. Therefore, when *curstamp* "wraps around," we must abandon all of the hints that were to be validated by obsolete timestamps.

```
⟨Remove all hints from all trigger lists 59⟩ ≡
  for ($k = 0$; $k < last\_node$; $k$ += $nd[k].loc + 1$) {
    for ($p = trigger(k)$; $p$; $o, p = link(p)$)
      if ($o, info(p) < 0$) {
        $o, q = link(p), r = link(q)$;      /* we know that $link(q) \neq 0$ */
        $oo, info(p) = info(r), link(p) = link(r)$;
        $putavail(q), putavail(r)$;
      }
  }
```

This code is used in section 58.

60. When *opt* was put into the queue, we made its age infinite. So it will have been purged in the meantime if and only if its age is now *cur_age*.

```
⟨If opt is no longer active, revert its fixit list and continue 60⟩ ≡
  if ($o, age(opt) \neq infinite\_age$) {
    ⟨Change the entries of $fixit(opt)$ back to triggers 43⟩;
    continue;
  }
```

This code is used in section 44.

61. The dancing.

```
⟨ Do a backtrack search, maintaining domain consistency 61 ⟩ ≡
   level = stage = 0;
newstage: ⟨ Increase stage 63 ⟩;
newlevel: nodes ++;
   ⟨ Increase level 64 ⟩;
   ⟨ Save the currently active sizes 71 ⟩;
   if (vbose & show_profile) profile[stage]++;
   if (sanity_checking) sanity();
   if (leak_checking) list_check(0);
   ⟨ Do special things if enough mems have accumulated 65 ⟩;
   if (stage ≤ groundstage) ⟨ Read and act on an option from xcutoff_file 80 ⟩;
   ⟨ Set best_itm to the best item for branching and t to its size 69 ⟩;
   if (stage ≡ xcutoff) ⟨ Output a partial solution and goto reset 77 ⟩;
   if (t ≡ max_nodes) ⟨ Visit a solution and goto reset 70 ⟩;
   oo, choice[level] = cur_choice = set[best_itm];
got_choice: deg[level] = t;      /* no mem charge (printout only) */
   cur_age = stage + stage + 1;      /* age s + 1/2 */
   if (¬include_option(cur_choice)) goto tryagain;
   if (¬empty_the_queue()) goto tryagain;
   goto newstage;
tryagain: if (t ≡ 1) goto backup;
   if (vbose & show_choices) fprintf(stderr, "Backtracking␣in␣stage␣"O"d\n", stage);
   goto purgeit;
reset: o, saveptr = saved[stage];
backup: if (−−stage ≤ groundstage) goto done;
   if (vbose & show_choices) fprintf(stderr, "Backtracking␣to␣stage␣"O"d\n", stage);
   oo, level = stagelevel[stage], curstamp = stagestamp[stage];
purgeit: if (vbose & show_option_counts) totopts = levelopts[level];
   ⟨ Restore the currently active sizes 72 ⟩;
   cur_age = stage + stage;      /* age s */
   if (¬(o, purge_the_option(choice[level], active))) goto backup;
   if (¬empty_the_queue()) goto backup;
   goto newlevel;
```

This code is used in section 46.

62. 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 stage;  /* number of choices in current partial solution */
  int level;  /* current depth in the search tree (which is binary) */
  int cur_age;  /* current stage or stage + 1/2 (times 2) */
  int choice[max_level];  /* the option and item chosen on each level */
  int deg[max_level];  /* the number of options that item had at the time */
  int saved[max_stage];  /* size of savestack at each stage */
  int stagelevel[max_stage];  /* the most recent level at each stage */
  int levelstage[max_level];  /* the stage that corresponds to each level */
  int levelopts[max_level];  /* options remaining at each level */
  int stagestamp[max_stage];  /* timestamp that's current at each stage */
  ullng profile[max_stage] = {1};  /* number of search tree nodes on each stage */
  twoints savestack[savesize];
  int saveptr;  /* current size of savestack */
  int color[max_cols];  /* current color of inactivated items */
  int trig_head[infinite_age], trig_tail[infinite_age];  /* in purge_option */
```

63.
```
⟨Increase stage 63⟩ ≡
  if (++stage > maxs) {
    if (stage ≥ max_stage) {
      fprintf(stderr, "Too␣many␣stages!\n");
      exit(−40);
    }
    maxs = stage;
  }
  ⟨Bump curstamp 58⟩;
  o, stagestamp[stage] = curstamp;
```
This code is used in section 61.

64.
```
⟨Increase level 64⟩ ≡
  if (++level > maxl) {
    if (level ≥ max_level) {
      fprintf(stderr, "Too␣many␣levels!\n");
      exit(−4);
    }
    maxl = level;
  }
  oo, levelstage[level] = stage, stagelevel[stage] = level;
  if (vbose & show_option_counts) levelopts[level] = totopts;
```
This code is used in section 61.

65. ⟨ Do special things if enough *mems* have accumulated 65 ⟩ ≡

```
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 61.

66. This is where we extend the partial solution.

Notice a tricky point: We must go through the sets from right to left, because the options we purge move right as they leave the set.

⟨ Subroutines 6 ⟩ +≡

```
int include_option(int opt)
{
  register int c, cc, k, p, q, pp, s, ss, optp;
  subroutine_overhead;
  if (vbose & show_choices) {
    fprintf(stderr, "S"O"d:", stage);
    print_option(opt, stderr, 1, 0);
  }
  for (opt--; o, nd[opt].itm > 0; opt--) ;      /* move back to the spacer */
  ⟨ Inactivate all items of opt, and record their colors 67 ⟩;
  for (k = active; k < oactive; k++) {
    oo, s = item[k], ss = s + size(s) − 1;
    if (s ≥ second ∧ (o, c = color[k])) {      /* we must purify s */
      for ( ; ss ≥ s; ss--) {
        o, optp = set[ss];
        if ((o, nd[optp].clr ≠ c) ∧ ¬purge_the_option(optp, oactive)) return 0;
      }
    } else
      for ( ; ss ≥ s; ss--) {
        o, optp = set[ss] − 1;
        while (o, nd[optp].itm > 0) optp--;      /* move to the spacer */
        if (optp ≠ opt ∧ ¬purge_option(optp, oactive)) return 0;
      }
  }
  ⟨ Make opt itself inactive 68 ⟩;
  return 1;
}
```

67. An item becomes inactive when it becomes part of the solution-so-far (hence it leaves the problem-that-remains). Active primary items are those that haven't yet been covered. Active secondary items are those that haven't yet been purified.

The active items are the first *active* entries of the *item* list. At one time I thought it would be wise to keep primary and secondary items separate, using a sparse-set discipline independently on each sector. But I found that the time spent in maintaining and searching the active list was negligible in comparison with the overall running time; so I've kept the implementation simple.

At this point in the computation, an item of *opt* will be inactive if and only if it is secondary and purified, because we are including *opt* in the partial solution.

```
⟨ Inactivate all items of opt, and record their colors 67 ⟩ ≡
  p = oactive = active;
  for (q = opt + 1; o, (c = nd[q].itm) > 0; q++) {
    o, pp = pos(c);
    if (pp < p) {      /* c is active */
      o, cc = item[--p];
      oo, item[p] = c, item[pp] = cc;
      oo, color[p] = nd[q].clr;
      oo, pos(cc) = pp, pos(c) = p;
      updates++;
    }
  }
  active = p;
```

This code is used in section 66.

68. This program differs from SSXCC1 in one significant way: It makes option *opt* inactive. In particular, it makes all of *opt*'s items have size 0, except for unpurified secondaries. (Thus, we essentially say that newly assigned variables—the inactivated primary items—should have empty domains when they leave the current subproblem, while SSXCC1 left them with domains of size 1.)

It would be a mistake to call *purge_option*(*opt*, *oactive*), of course, because that procedure doesn't want any primary items to become optionless. On the contrary, we have precisely the opposite goal: We *celebrate* the fact that all of the primaries in *opt* have become covered.

We don't have to change *trigger*(*opt*), because no active options involve inactive primary items.

```
⟨ Make opt itself inactive 68 ⟩ ≡
  for (k = active; k < oactive; k++) {
    o, s = item[k];
    if (s ≥ second) continue;
    if (size(s) ≠ 1) confusion("include");
    o, size(s) = 0;
  }
  if (vbose & show_purges) {
    fprintf(stderr, "␣"O"d."O"c␣removing␣option␣", cur_age ≫ 1, cur_age & 1 ? '5' : '0', opt);
    print_option(opt + 1, stderr, 0, 1);
  }
  o, age(opt) = cur_age;
  totopts--;
```

This code is used in section 66.

69. The "best item" is considered to be an item that minimizes the number of remaining choices. If there are several candidates, we choose the first one that we encounter.

Each primary item should have at least one valid choice, because of domain consistency.

```
⟨Set best_itm to the best item for branching and t to its size 69⟩ ≡
  t = max_nodes;
  if ((vbose & show_details) ∧ level < show_choices_max ∧ level ≥ maxl − show_choices_gap)
    fprintf(stderr, "Stage␣"O"d,", stage);
  for (k = 0; t > 1 ∧ 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 < t) {
        if (s ≡ 0) fprintf(stderr, "I'm␣confused.\n");      /* hide missed this */
        best_itm = item[k], t = s;
      }
    }
  if ((vbose & show_details) ∧ level < show_choices_max ∧ level ≥ maxl − show_choices_gap) {
    if (t ≡ max_nodes) 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 < max_nodes) maxdeg = t;
  if (shape_file) {
    if (t ≡ max_nodes) 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);
  }
```

This code is used in section 61.

70.

```
⟨Visit a solution and goto reset 70⟩ ≡
  {
    count++;
    if (spacing ∧ (count mod spacing ≡ 0)) {
      printf(""O"lld:\n", count);
      for (k = 1; k < stage; k++) print_option(choice[stagelevel[k]], stdout, 0, 0);
      fflush(stdout);
    }
    if (count ≥ maxcount) goto done;
    goto reset;
  }
```

This code is used in section 61.

71. ⟨ Save the currently active sizes 71 ⟩ ≡

```
oo, 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 61.

72. ⟨ Restore the currently active sizes 72 ⟩ ≡

```
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 61.

73. ⟨ 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);
  }
}
```

74. ⟨ Subroutines 6 ⟩ +≡

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

75. 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;
    register double f, fd;
    fprintf(stderr, "␣after␣"O"lld␣mems:␣"O"lld␣sols,", mems, count);
    for (f = 0.0, fd = 1.0, l = stagelevel[groundstage] + 1; l < level; l++) {
      fprintf(stderr, "␣"O"s"O"d", stagelevel[levelstage[l]] ≡ l ? "" : "~", deg[l]);
      if (stagelevel[levelstage[l]] ≡ l) {
        for (k = 1, d = deg[l], ll = l − 1; levelstage[ll] ≡ levelstage[l]; k++, d++, ll−−) ;
        fd ∗= d, f += (k − 1)/fd;      /∗ choice l is treated like k of d ∗/
      }
      if (l ≥ show_levels_max) {
        fprintf(stderr, "...");
        break;
      }
    }
    fprintf(stderr, "␣"O".5f\n", f + 0.5/fd);
  }
```

76. ⟨ Print the profile 76 ⟩ ≡

```
  {
    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 5.

77. I'm experimenting with a mechanism by which partial solutions of a large problem can be saved to temporary files and computed separately — for example, by a cluster of computers working in parallel. Each partial solution can be completed to full solutions when this program is run with one of the files output here, using `X`⟨ filename ⟩ on the command line.

```
⟨ Output a partial solution and goto reset 77 ⟩ ≡
  {
    ⟨ Open a new xcutoff_file 78 ⟩;
    fprintf(xcutoff_file, "Resume␣at␣stage␣"O"d\n", stage − 1);
    for (k = 1; k < level; k++) {
      for (o, j = choice[k]; o, nd[j − 1].itm > 0; j−−) ;
      putc(stagelevel[levelstage[k]] ≠ k ? '-' : '+', xcutoff_file);
      print_option(j, xcutoff_file, 0, 1);
    }
    fclose(xcutoff_file);
    xcount++;
    goto reset;
  }
```

This code is used in section 61.

78. #**define** *part_file_prefix* `"/tmp/part"` /∗ should be at most 10 or so characters ∗/
#**define** *part_file_name_size* 20

```
⟨ Open a new xcutoff_file 78 ⟩ ≡
  k = sprintf(xcutoff_name, part_file_prefix O"d", xcount);
  xcutoff_file = fopen(xcutoff_name, "w");
  if (¬xcutoff_file) {
    fprintf(stderr, "Sorry,␣I␣can't␣open␣file␣`"O"s'␣for␣writing!\n", xcutoff_name);
    exit(−1);
  }
```

This code is used in section 77.

79.
```
⟨ Open xcutoff_file for reading, and break 79 ⟩ ≡
  strncpy(xcutoff_name, argv[j] + 1, part_file_name_size − 1);
  xcutoff_file = fopen(xcutoff_name, "r");
  if (¬xcutoff_file)
    fprintf(stderr, "Sorry,␣I␣can't␣open␣file␣`"O"s'␣for␣reading!\n", xcutoff_name);
  if (fgets(buf, bufsize, xcutoff_file)) {
    if (strncmp(buf, "Resume␣at␣stage␣", 16) ≡ 0) {
      for (groundstage = 0, i = 16; o, buf[i] ≥ '0' ∧ buf[i] ≤ '9'; i++)
        groundstage = 10 ∗ groundstage + buf[i] − '0';
      if (vbose & show_basics) fprintf(stderr, "Resuming␣at␣stage␣"O"d\n", groundstage);
    }
  }
  break;
```

This code is used in section 4.

80. ⟨ Read and act on an option from *xcutoff_file* 80 ⟩ ≡

```
{
  if (¬fgets(buf, bufsize, xcutoff_file)) confusion("resuming");
  o, choice[level] = cur_choice = read_option();
  if (cur_choice < 0) {
    fprintf(stderr, "Misformatted␣option␣in␣file␣`"O"s':\n", xcutoff_name);
    fprintf(stderr, ""O"s", buf);
    exit(−1);
  }
  t = 1;
  if (o, buf[0] ≡ '+') goto got_choice;
  goto purgeit;
}
```

This code is used in section 61.

81. ⟨ Report the number of partial solutions output 81 ⟩ ≡

```
fprintf(stderr, "Partial␣solutions␣saved␣on␣"part_file_prefix"0.."O"s.\n", xcutoff_name);
```

This code is used in section 5.

82. ⟨ Global variables 3 ⟩ +≡

```
char xcutoff_name[part_file_name_size];
FILE *xcutoff_file;
int groundstage;    /* the stage where calculation begins or resumes */
```

83. Index.

XCCDC1