1. Intro. This program is an "XCC solver" that I'm writing as an experiment in the use of so-called sparse-set data structures instead of the dancing links structures that I've played with for thirty years. I plan to write it as if I live on a planet where the sparse-set ideas are well known, but doubly linked links are almost unheard-of. As I begin, I know that the similar program SSXCC1 works fine.

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

My original attempt, SSXC0, kept the basic structure of DLX1 and changed only the data structure link conventions. The present version incorporates new ideas from Christine Solnon's program XCC-WITH-DANCING-CELLS, which she wrote in October 2020. In particular, she proposed saving all the active set sizes on a stack; program SSXCC0 recomputed them by undoing the forward calculations in reverse.

Here I extend SSXCC1 by incorporating also the weighting scheme by which she improved the MRV heuristic. Motivated by the paper of Boussemart, Hemery, Lecoutre, and Sais in *Proc. 16 European Conference on Artificial Intelligence* (2004), 146–150, she increases the weight of a primary item when its current set of options becomes null. Items are chosen for branching based on the size of their set divided by their current weight.

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.)

Here is the overall structure:

```
#define o mems++   /* count one mem */
#define oo mems += 2   /* count two mems */
#define ooo mems += 3   /* count three mems */
#define O "%"   /* used for percent signs in format strings */
#define mod %   /* used for percent signs denoting remainder in C */
#define max_level 5000   /* at most this many options in a solution */
#define max_cols 100000   /* at most this many items */
#define max_nodes 10000000   /* at most this many nonzero elements in the matrix */
#define savesize 10000000   /* 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 7⟩;
  ⟨Global variables 3⟩;
  ⟨Subroutines 10⟩;
  main(int argc, char *argv[])
  {
    register int c, cc, i, j, k, p, pp, q, r, s, t, cur_choice, cur_node, best_itm;
    ⟨Process the command line 4⟩;
    ⟨Input the item names 14⟩; ⟨Input the options 16⟩;
    if (vbose & show_basics) ⟨Report the successful completion of the input phase 23⟩;
    if (vbose & show_tots) ⟨Report the item totals 24⟩;
    imems = mems, mems = 0;
    if (baditem) ⟨Report an uncoverable item 22⟩ else ⟨Solve the problem 25⟩;
  done: if (vbose & show_profile) ⟨Print the profile 42⟩;
    if (vbose & show_final_weights) {
      fprintf(stderr, "Final␣weights:\n");
      print_weights();
    }
    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) + 2 * maxl * sizeof(int) + maxsaveptr * sizeof (twoints);
      fprintf(stderr, "␣"O"llu␣updates,␣"O"llu␣bytes,␣"O"llu␣nodes.\n", updates, bytes, nodes);
    }
    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 *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);
- '`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.

```
#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_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 */
```

⟨ 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;    /* maximum level 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;    /* 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 */
  FILE *shape_file;    /* file for optional output of search tree shape */
  char *shape_name;    /* its name */
  int maxdeg;    /* the largest branching degree seen so far */
```

See also sections 8 and 26.

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 '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>]␣[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);
}
```

This code is used in section 2.

5. ⟨ Close the files 5 ⟩ ≡

```
if (shape_file) fclose(shape_file);
```

This code is used in section 2.

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

The basic idea is amazingly simple, when specialized to the situations that we need to deal with: We can represent a subset S of the universe $U = \{x_0, x_1, \ldots, x_{n-1}\}$ by maintaining two n-element arrays p and q, each of which is a permutation of $\{0, 1, \ldots, n-1\}$, together with an integer s in the range $0 \le s \le n$. In fact, p is the *inverse* of q; and s is the number of elements of S. The current value of the set S is then simply $\{x_{p_0}, \ldots, x_{p_{s-1}}\}$. (Notice that every s-element 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 permutation of $\{0, 1, \ldots, n-1\}$. The arrays that play the role of q in the following routines don't have indices that are consecutive; they live inside of other structures.

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

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.

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 */
#define lname(x) set[(x) − 4].i    /* the first four bytes of x's name */
#define rname(x) set[(x) − 3].i    /* the last four bytes of x's name */
#define wt(x) set[(x) − 5].f
```

⟨ Type definitions 7 ⟩ ≡

```
  typedef struct node_struct {
    int itm;    /* the item x corresponding to this node */
    int loc;    /* where this node resides in x's active set */
    int clr;    /* color associated with item x in this option, if any */
    int spr;    /* a spare field inserted only to maintain 16-byte alignment */
  } node;
  typedef union {
    int i;    /* an integer (32 bits) */
    float f;    /* a floating point value (fits in 4 bytes) */
  } tetrabyte;
```

See also section 9.

This code is used in section 2.

8. ⟨ Global variables 3 ⟩ +≡

```
node nd[max_nodes];   /* the master list of nodes */
int last_node;   /* the first node in nd that's not yet used */
int item[max_cols];   /* the master list of items */
int second = max_cols;   /* boundary between primary and secondary items */
int last_itm;   /* items seen so far during input, plus 1 */
tetrabyte set[max_nodes + 5 * 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 */
```

9. 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 7 ⟩ +≡

```
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;
```

10. ⟨ Subroutines 10 ⟩ ≡

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

See also sections 11, 12, 13, 31, 34, 39, 40, and 41.

This code is used in section 2.

11. An option is identified not by name but by the names of the items it contains. Here is a routine that prints an option, given a pointer to any of its nodes. It also prints the position of the option in its item list.

⟨ Subroutines 10 ⟩ +≡

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

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

⟨ Subroutines 10 ⟩ +≡

```
void print_itm(int c)
{
  register int p;
  if (c ≤ 4 ∨ c ≥ setlength ∨ pos(c) < 0 ∨ pos(c) ≥ itemlength ∨ item[pos(c)] ≠ c) {
    fprintf(stderr, "Illegal␣item␣"O"d!\n", c);
    return;
  }
  fprintf(stderr, "Item");
  print_item_name(c, stderr);
  if (c < second) fprintf(stderr, "␣("O"d␣of␣"O"d),␣length␣"O"d,␣weight␣"O".1f:\n",
        pos(c) + 1, active, size(c), wt(c));
  else if (pos(c) ≥ active)
    fprintf(stderr, "␣(secondary␣"O"d,␣purified),␣length␣"O"d:\n", pos(c) + 1, size(c));
  else fprintf(stderr, "␣(secondary␣"O"d),␣length␣"O"d:\n", pos(c) + 1, size(c));
  for (p = c; p < c + size(c); p++) prow(set[p].i);
}
```

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

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

⟨ Subroutines 10 ⟩ +≡
  void sanity(void)
  {
    register int k, x, i, l, r, q, qq;
    for (k = 0; k < itemlength; k++) {
      x = item[k];
      if (pos(x) ≠ k) {
        fprintf(stderr, "Bad pos field of item");
        print_item_name(x, stderr);
        fprintf(stderr, " ("O"d,"O"d)!\n", k, x);
      }
    }
    for (i = 0; i < last_node; i++) {
      l = nd[i].itm, r = nd[i].loc;
      if (l ≤ 0) {
        if (nd[i + r + 1].itm ≠ −r) fprintf(stderr, "Bad spacer in nodes "O"d, "O"d!\n", i, i + r + 1);
        qq = 0;
      } else {
        if (l > r) fprintf(stderr, "itm>loc in node "O"d!\n", i);
        else {
          if (set[r].i ≠ i) {
            fprintf(stderr, "Bad loc field for option "O"d of item", r − l + 1);
            print_item_name(l, stderr);
            fprintf(stderr, " in node "O"d!\n", i);
          }
          if (pos(l) < active) {
            if (r < l + size(l)) q = +1; else q = −1;   /* in or out? */
            if (q ∗ qq < 0) {
              fprintf(stderr, "Flipped status at option "O"d of item", r − l + 1);
              print_item_name(l, stderr);
              fprintf(stderr, " in node "O"d!\n", i);
            }
            qq = q;
          }
        }
      }
    }
  }
```

14. Inputting the matrix. Brute force is the rule in this part of the code, whose goal is to parse and store the input data and to check its validity.

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

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

⟨Input the item names 14⟩ ≡
  while (1) {
    if (¬fgets(buf, bufsize, stdin)) break;
    if (o, buf[p = strlen(buf) − 1] ≠ '\n') panic("Input␣line␣way␣too␣long");
    for (p = 0; o, isspace(buf[p]); p++) ;
    if (buf[p] ≡ '|' ∨ ¬buf[p]) continue;      /* bypass comment or blank line */
    last_itm = 1;
    break;
  }
  if (¬last_itm) panic("No␣items");
  for ( ; o, buf[p]; ) {
    o, namebuf.lr.l = namebuf.lr.r = 0;
    for (j = 0; j < 8 ∧ (o, ¬isspace(buf[p + j])); j++) {
      if (buf[p + j] ≡ ':' ∨ buf[p + j] ≡ '|') panic("Illegal␣character␣in␣item␣name");
      o, namebuf.str[j] = buf[p + j];
    }
    if (j ≡ 8 ∧ ¬isspace(buf[p + j])) panic("Item␣name␣too␣long");
    oo, lname(last_itm ≪ 2) = namebuf.lr.l, rname(last_itm ≪ 2) = namebuf.lr.r;
    ⟨Check for duplicate item name 15⟩;
    last_itm++;
    if (last_itm > max_cols) panic("Too␣many␣items");
    for (p += j + 1; o, isspace(buf[p]); p++) ;
    if (buf[p] ≡ '|') {
      if (second ≠ max_cols) panic("Item␣name␣line␣contains␣|␣twice");
      second = last_itm;
      for (p++; o, isspace(buf[p]); p++) ;
    }
  }
```

This code is used in section 2.

15. ⟨Check for duplicate item name 15⟩ ≡

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

This code is used in section 14.

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

⟨ Input the options 16 ⟩ ≡
```
while (1) {
  if (¬fgets(buf, bufsize, stdin)) break;
  if (o, buf[p = strlen(buf) − 1] ≠ '\n') panic("Option␣line␣too␣long");
  for (p = 0; o, isspace(buf[p]); p++) ;
  if (buf[p] ≡ '|' ∨ ¬buf[p]) continue;    /* bypass comment or blank line */
  i = last_node;    /* remember the spacer at the left of this option */
  for (pp = 0; buf[p]; ) {
    o, namebuf.lr.l = namebuf.lr.r = 0;
    for (j = 0; j < 8 ∧ (o, ¬isspace(buf[p + j])) ∧ buf[p + j] ≠ ':'; j++) o, namebuf.str[j] = buf[p + j];
    if (¬j) panic("Empty␣item␣name");
    if (j ≡ 8 ∧ ¬isspace(buf[p + j]) ∧ buf[p + j] ≠ ':') panic("Item␣name␣too␣long");
    ⟨ Create a node for the item named in buf[p] 17 ⟩;
    if (buf[p + j] ≠ ':') o, nd[last_node].clr = 0;
    else if (k ≥ second) {
      if ((o, isspace(buf[p + j + 1])) ∨ (o, ¬isspace(buf[p + j + 2])))
        panic("Color␣must␣be␣a␣single␣character");
      o, nd[last_node].clr = (unsigned char) buf[p + j + 1];
      p += 2;
    } else panic("Primary␣item␣must␣be␣uncolored");
    for (p += j + 1; o, isspace(buf[p]); p++) ;
  }
  if (¬pp) {
    if (vbose & show_warnings) fprintf(stderr, "Option␣ignored␣(no␣primary␣items):␣"O"s", buf);
    while (last_node > i) {
      ⟨ Remove last_node from its item list 18 ⟩;
      last_node−−;
    }
  } else {
    o, nd[i].loc = last_node − i;    /* complete the previous spacer */
    last_node++;    /* create the next spacer */
    if (last_node ≡ max_nodes) panic("Too␣many␣nodes");
    options++;
    o, nd[last_node].itm = i + 1 − last_node;
    nd[last_node].spr = options;    /* option number, for debugging only */
  }
}
⟨ Initialize item 19 ⟩;
⟨ Expand set 20 ⟩;
⟨ Adjust nd 21 ⟩;
```
This code is used in section 2.

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

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

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

This code is used in section 16.

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

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

This code is used in section 16.

19. ⟨ Initialize *item* 19 ⟩ ≡

```
active = itemlength = last_itm − 1;
for (k = 0, j = 5; k < itemlength; k++) oo, item[k] = j, j += (k < second − 2) + 4 + size((k + 1) ≪ 2);
setlength = j − 4;
if (second ≡ max_cols) osecond = active, second = j;
else osecond = second − 1;
```

This code is used in section 16.

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

⟨ Expand *set* 20 ⟩ ≡

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

This code is used in section 16.

21. ⟨ Adjust *nd* 21 ⟩ ≡

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

This code is used in section 16.

22. ⟨ Report an uncoverable item 22 ⟩ ≡

```
{
  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.

23. 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 23 ⟩ ≡

```
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.

24. 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 24 ⟩ ≡

```
{
  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.

25. The dancing. Our strategy for generating all exact covers will be to repeatedly choose an item that appears to be hardest to cover, namely an item whose set is currently smallest, among all items that still need to be covered. And we explore all possibilities via depth-first search.

The neat part of this algorithm is the way the sets are maintained. Depth-first search means last-in-first-out maintenance of data structures; and the sparse-set representations make it particularly easy to undo what we've done at less-deep levels.

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

```
⟨Solve the problem 25⟩ ≡
  {
    level = 0;
  forward: nodes++;
    if (vbose & show_profile) profile[level]++;
    if (sanity_checking) sanity();
    ⟨Do special things if enough mems have accumulated 27⟩;
    ⟨Set best_itm to the best item for branching 35⟩;
    if (t ≡ max_nodes) ⟨Visit a solution and goto backup 36⟩;
    cur_choice = best_itm;
    if (t ≡ 0) {      /* t = size(best_itm) */
      tough_itm = best_itm;
      goto abort;
    }
    ⟨Swap best_itm out of the active list 28⟩;
    oactive = active, hide(best_itm, 0, 0);      /* hide its options */
    ⟨Save the currently active sizes 37⟩;
  advance: oo, cur_node = choice[level] = set[cur_choice].i;
  tryit: if ((vbose & show_choices) ∧ level < show_choices_max) {
      fprintf(stderr, "L"O"d:", level);
      print_option(cur_node, stderr);
    }
    ⟨Swap out all other items of cur_node 29⟩;
    ⟨Hide the other options of those items, or goto abort 30⟩;
    if (++level > maxl) {
      if (level ≥ max_level) {
        fprintf(stderr, "Too␣many␣levels!\n");
        exit(−4);
      }
      maxl = level;
    }
    goto forward;
  backup: if (level ≡ 0) goto done;
    level−−;
    o, cur_node = choice[level];
    o, best_itm = nd[cur_node].itm, cur_choice = nd[cur_node].loc;
    goto reconsider;
  abort: ⟨Increase the weight of tough_itm 33⟩;
  reconsider: if (o, cur_choice + 1 ≥ best_itm + size(best_itm)) goto backup;
    ⟨Restore the currently active sizes 38⟩;
    cur_choice++; goto advance;
  }
```

This code is used in section 2.

26. We save the sizes of active items on *savestack*, whose entries have two fields *itm* and *siz*. This stack makes it easy to undo all deletions, by simply restoring the former sizes.

⟨ Global variables 3 ⟩ +≡
```
int level;   /* number of choices in current partial solution */
int choice[max_level];   /* the node chosen on each level */
int saved[max_level + 1];   /* size of savestack on each level */
ullng profile[max_level];   /* number of search tree nodes on each level */
twoints savestack[savesize];
int saveptr;   /* current size of savestack */
int tough_itm;   /* an item that led to a conflict */
```

27. ⟨ Do special things if enough *mems* have accumulated 27 ⟩ ≡
```
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 25.

28. ⟨ Swap *best_itm* out of the active list 28 ⟩ ≡
```
p = active − 1, active = p;
o, pp = pos(best_itm);
o, cc = item[p];
oo, item[p] = best_itm, item[pp] = cc;
oo, pos(cc) = pp, pos(best_itm) = p;
updates++;
```
This code is used in section 25.

29. Note that a colored secondary item might have already been purified, in which case it has already been swapped out. We don't want to tamper with any of the inactive items.

⟨ Swap out all other items of *cur_node* 29 ⟩ ≡
```
p = oactive = active;
for (q = cur_node + 1; q ≠ cur_node; ) {
  o, c = nd[q].itm;
  if (c < 0) q += c;
  else {
    o, pp = pos(c);
    if (pp < p) {
      o, cc = item[−−p];
      oo, item[p] = c, item[pp] = cc;
      oo, pos(cc) = pp, pos(c) = p;
      updates++;
    }
    q++;
  }
}
active = p;
```
This code is used in section 25.

30. A secondary item was purified at lower levels if and only if its position is $\ge$ *oactive*.

$\langle$ Hide the other options of those items, or **goto** *abort* 30 $\rangle \equiv$

```
for (q = cur_node + 1; q ≠ cur_node; ) {
  o, cc = nd[q].itm;
  if (cc < 0) q += cc;
  else {
    if (cc < second) {
      if (hide(cc, 0, 1) ≡ 0) goto abort;
    } else {      /* do nothing if cc already purified */
      o, pp = pos(cc);
      if (pp < oactive ∧ (o, hide(cc, nd[q].clr, 1) ≡ 0)) goto abort;
    }
    q++;
  }
}
```

This code is used in section 25.

31. The *hide* routine hides all of the incompatible options remaining in the set of a given item. If *check* is nonzero, it returns zero if that would cause a primary item to be uncoverable.

If the *color* parameter is zero, all options are incompatible. Otherwise, however, the given item is secondary, and we retain options for which that item has a *color* match.

When an option is hidden, it leaves all sets except the set of that given item. And the given item is inactive. Thus a node is never removed from a set twice.

$\langle$ Subroutines 10 $\rangle$ +$\equiv$

```
int hide(int c, int color, int check)
{
  register int cc, s, rr, ss, nn, tt, uu, vv, nnp;
  for (o, rr = c, s = c + size(c); rr < s; rr++) {
    o, tt = set[rr].i;
    if (¬color ∨ (o, nd[tt].clr ≠ color)) ⟨Remove option tt from the other sets it's in 32⟩;
  }
  return 1;
}
```

32. ⟨Remove option tt from the other sets it's in 32⟩ ≡

```
{
  for (nn = tt + 1; nn ≠ tt; ) {
    o, uu = nd[nn].itm, vv = nd[nn].loc;
    if (uu < 0) { nn += uu; continue; }
    if (o, pos(uu) < oactive) {
      o, ss = size(uu) − 1;
      if (ss ≡ 0 ∧ check ∧ uu < second ∧ (o, pos(uu) < active)) {
        if ((vbose & show_choices) ∧ level < show_choices_max) {
          if (¬(vbose & show_weight_bumps)) {
            fprintf(stderr, "␣can't␣cover");
            print_item_name(uu, stderr);
            fprintf(stderr, "\n");
          }
        }
        tough_itm = uu;
        return 0;
      }
      o, nnp = set[uu + ss].i;
      o, size(uu) = ss;
      oo, set[uu + ss].i = nn, set[vv].i = nnp;
      oo, nd[nn].loc = uu + ss, nd[nnp].loc = vv;
      updates++;
    }
    nn++;
  }
}
```

This code is used in section 31.

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

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

⟨Increase the weight of tough_itm 33⟩ ≡
  oo, wt(tough_itm) += dw;
  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;

    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;
  }
```

This code is used in section 25.

34. ⟨Subroutines 10⟩ +≡

```
  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]));
      }
  }
```

35. The "best item" is considered to be an item that minimizes the number of remaining choices, divided by the item's weight. If there are several candidates, we choose the leftmost.

When an item has at most one option left, however, we consider it to be forced.

Sometimes an item has no remaining options. This couldn't happen in SSXCC1; but the present program might choose to branch on a heavyweight item whose options strictly include all of the remaining options of a lightweight item. (The heavyweight's options are removed when the *check* parameter to *hide* is 0.)

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

```
#define infty 2·10^32_F   /* twice dangerous */
⟨Set best_itm to the best item for branching 35⟩ ≡
{
  register float score, tscore, w;
  register int force;
  score = infty, force = 0, t = max_nodes;
  if ((vbose & show_details) ∧ level < show_choices_max ∧ level ≥ maxl − show_choices_gap)
    fprintf(stderr, "Level␣"O"d:", level);
  for (k = 0; k < active; k++)
    if (o, item[k] < second) {
      oo, s = size(item[k]), w = wt(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,"O".1f)", s, w);
      }
      if (s ≤ 1) {
        if (s < t) force = 1, t = s, best_itm = item[k];
      } else if (¬force) {
        tscore = s/w;
        if (tscore ≥ infty) tscore = dangerous;
        if (tscore < score) best_itm = item[k], score = tscore;
      }
    }
  if (¬force) t = (score ≡ infty ? max_nodes : size(best_itm));
  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); if (t ≤ 1) fprintf(stderr, "(forced)\n");
      else fprintf(stderr, "("O"d),␣score␣"O".4f\n", t, score);
    }
  }
  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 25.

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

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

This code is used in section 25.

37. ⟨ Save the currently active sizes 37 ⟩ ≡

```
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]);
o, saved[level + 1] = saveptr = saveptr + active;
```

This code is used in section 25.

38. ⟨ Restore the currently active sizes 38 ⟩ ≡

```
o, saveptr = saved[level + 1];
o, active = saveptr − saved[level];
for (p = −active; p < 0; p++) oo, size(savestack[saveptr + p].l) = savestack[saveptr + p].r;
```

This code is used in section 25.

39. ⟨ Subroutines 10 ⟩ +≡

```
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);
  }
}
```

40. ⟨ Subroutines 10 ⟩ +≡

```
void print_state(void)
{
  register int l;
  fprintf(stderr, "Current␣state␣(level␣"O"d):\n", level);
  for (l = 0; l < level; l++) {
    print_option(choice[l], stderr);
    if (l ≥ show_levels_max) {
      fprintf(stderr, "␣...\n");
      break;
    }
  }
  fprintf(stderr, "␣"O"lld␣solutions,␣"O"lld␣mems,␣and␣max␣level␣"O"d␣so␣far.\n", count,
      mems, maxl);
}
```

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

$$\texttt{0},\ \texttt{1},\ \ldots,\ \texttt{9},\ \texttt{a},\ \texttt{b},\ \ldots,\ \texttt{z},\ \texttt{A},\ \texttt{B},\ \ldots,\texttt{Z}.$$

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

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

⟨ Subroutines 10 ⟩ +≡

```
void print_progress(void)
{
  register int l, k, d, c, p;
  register double f, fd;
  fprintf(stderr, "␣after␣"O"lld␣mems:␣"O"lld␣sols,", mems, count);
  for (f = 0.0, fd = 1.0, l = 0; l < level; l++) {
    c = nd[choice[l]].itm, d = size(c), k = nd[choice[l]].loc − c + 1;
    fd *= d, f += (k − 1)/fd;    /* choice l is k of d */
    fprintf(stderr, "␣"O"c"O"c", k < 10 ? '0' + k : k < 36 ? 'a' + k − 10 : k < 62 ? 'A' + k − 36 : '*',
        d < 10 ? '0' + d : d < 36 ? 'a' + d − 10 : d < 62 ? 'A' + d − 36 : '*');
    if (l ≥ show_levels_max) {
      fprintf(stderr, "...");
      break;
    }
  }
  fprintf(stderr, "␣"O".5f\n", f + 0.5/fd);
}
```

42. ⟨ Print the profile 42 ⟩ ≡

```
{
  fprintf(stderr, "Profile:\n");
  for (level = 0; level <= maxl; level++) fprintf(stderr, ""O"3d: "O"lld\n", level, profile[level]);
}
```

This code is used in section 2.

43. Index.

SSXCC2