November 24, 2020 at 13:23

1. Intro. This program is part of a series of "exact cover solvers" that I'm putting together for my own education as I prepare to write Section 7.2.2.1 of *The Art of Computer Programming*. My intent is to have a variety of compatible programs on which I can run experiments, in order to learn how different approaches work in practice.

The basic input format for all of these solvers is described at the beginning of program DLX1, and you should read that description now if you are unfamiliar with it. You should in fact read the beginning of DLX2, too, because it adds "color controls" to the repertoire of DLX1.

DLX5 extends DLX2 by allowing options to have nonnegative *costs*. The goal is to find a minimum-cost solution (or, more generally, to find the k best solutions, in the sense that the sum of their costs is minimized).

The input format is extended so that entries such as `|`n can be appended to any option, to specify its cost. If several such entries appear in the same option, the cost is their sum.

Whenever a solution is found whose cost is less than kth best seen so far, that solution is output. For example, suppose the given problem has only ten solutions, whose costs happen to be (0, 0, 1, 1, 2, 2, 3, 3, 4, 4). We might discover them in any order, perhaps (3, 1, 4, 1, 2, 3, 2, 4, 0, 0). If $k = 1$ (the default), we'll output solutions of cost 3, 1, 0. If $k = 3$, we'll output solutions of cost 3, 1, 4, 1, 2, 0, 0. If $k = 5$, we'll output solutions of cost 3, 1, 4, 1, 2, 3, 2, 0, 0. If $k \geq 8$, we'll output all ten solutions. Different values of k might, however, affect the order of discovery.

This program internally assigns a "tax" to each item, and changes the cost of each option to its *net cost*, which is the original cost minus the taxes on each of its items. For example, the net cost of option '`a b c |7`' will be not \$7 but \$1, if the tax on each of `a`, `b`, and `c` is \$2. This modification doesn't change the problem in any essential way, because the net cost of each solution is equal to the original cost of that solution minus the total tax on all items (and that total tax is constant). Taxes are assessed in such a way that each item belongs to at least one net-zero-cost option, yet all options have a nonnegative net cost. The point is that options whose net cost is large cannot be used in solutions whose net cost is small.

If the input contains no cost specifications, the behavior of DLX5 will almost exactly match that of DLX2, except for needing more time and space.

[*Historical note:* The simple cutoff rule in this program was used in one of the first computer codes for min-cost exact cover; see Garfinkel and Nemhauser, *Operations Research* **17** (1969), 848–856.]

2. After this program finds its solutions, it normally prints their total number on *stderr*, together with statistics about how many nodes were in the search tree, and how many "updates" and "cleansings" were made. The running time in "mems" is also reported, together with the approximate number of bytes needed for data storage. (An "update" is the removal of an option from its item list. A "cleansing" is the removal of a satisfied color constraint from its option. One "mem" essentially means a memory access to a 64-bit word. The reported totals don't include the time or space needed to parse the input or to format the output.)

Here is the overall structure:

```
#define o mems++   /* count one mem */
#define oo mems += 2   /* count two mems */
#define ooo mems += 3   /* count three mems */
#define O "%"   /* used for percent signs in format strings */
#define mod %   /* used for percent signs denoting remainder in C */
#define max_level 5000   /* at most this many options in a solution */
#define max_cols 100000   /* at most this many items */
#define max_nodes 10000000   /* at most this many nonzero elements in the matrix */
#define bufsize (9 * max_cols + 3)   /* a buffer big enough to hold all item names */
#define sortbufsize 32   /* for the z lookahead heuristic */

#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 11⟩;
  ⟨Global variables 6⟩;
  ⟨Subroutines 15⟩;
  main(int argc, char *argv[])
  {
    register int cc, i, j, k, p, pp, q, r, s, t, cur_node, best_itm;
    register ullng tmpcost, curcost, mincost, nextcost;
    ⟨Process the command line 7⟩;
    ⟨Do the input phase 3⟩;
    ⟨Solve the problem 39⟩;
  done: if (sanity_checking) sanity();
    ⟨Bid farewell 4⟩;
  }
```

3. ⟨Do the input phase 3⟩ ≡

```
  ⟨Input the item names 19⟩;
  ⟨Input the options 22⟩;
  ⟨Assign taxes 31⟩;
  ⟨Sort the item lists 32⟩;
  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;
```

This code is used in section 2.

4. ⟨ Bid farewell 4 ⟩ ≡

```
if (vbose & show_tots) ⟨ Report the item totals 38 ⟩;
if (vbose & show_profile) ⟨ Print the profile 61 ⟩;
if (vbose & show_basics) {
  fprintf(stderr, "Altogether␣"O"llu␣solution"O"s,␣"O"llu+"O"llu␣mems,", count,
      count ≡ 1 ? "" : "s", imems, mems);
  bytes = last_itm * sizeof(item) + last_node * sizeof(node) + maxl * sizeof(int);
  fprintf(stderr, "␣"O"llu␣updates,␣"O"llu␣cleansings,", updates, cleansings);
  fprintf(stderr, "␣"O"llu␣bytes,␣"O"llu␣nodes.\n", bytes, nodes);
}
if ((vbose & show_opt_costs) ∧ count) ⟨ Print the kthresh best costs found 9 ⟩;
⟨ Close the files 8 ⟩;
```

This code is used in section 2.

5. You can control the amount of output, as well as certain properties of the algorithm, by specifying options on the command line:

- '`v`⟨ integer ⟩' enables or disables various kinds of verbose output on *stderr*, given by binary codes such as *show_choices*;
- '`m`⟨ integer ⟩' causes every mth solution to be output (the default is `m0`, which merely counts them);
- '`k`⟨ positive integer ⟩' causes the algorithm to cut off solutions that don't improve costwise on the k best seen so far (the default is 1, and k must not exceed *maxk*);
- '`Z`⟨ string ⟩' causes a warning to be printed if there's an option that doesn't have exactly one primary item beginning with c, for each character c of the string (thereby allowing a special heuristic to be used for cutting off false starts);
- '`z`⟨ positive integer ⟩' causes a warning to be printed if there's an option that doesn't have exactly this many primary items in addition to those specified by `Z` (thereby allowing a special heuristic to be used for cutting off false starts);
- '`h`⟨ positive integer ⟩' sets *lenthresh*, a heuristic that limits the amount of lookahead when we're trying to identify the best item for branching (default 10);
- '`d`⟨ integer ⟩' sets *delta*, which causes periodic state reports on *stderr* after the algorithm has performed approximately *delta* mems since the previous report (default 10000000000);
- '`c`⟨ positive integer ⟩' limits the levels on which choices are shown during verbose tracing;
- '`C`⟨ positive integer ⟩' limits the levels on which choices are shown in the periodic state reports;
- '`l`⟨ nonnegative integer ⟩' gives a *lower* limit, relative to the maximum level so far achieved, to the levels on which choices are shown during verbose tracing;
- '`t`⟨ positive integer ⟩' causes the program to stop after this many solutions have been found;
- '`T`⟨ integer ⟩' sets *timeout* (which causes abrupt termination if $mems > timeout$ at the beginning of a level);
- '`S`⟨ filename ⟩' to output a "shape file" that encodes the search tree.

```
#define show_basics 1   /* vbose code for basic stats; this is the default */
#define show_choices 2   /* vbose code for backtrack logging */
#define show_details 4   /* vbose code for further commentary */
#define show_taxes 8   /* vbose code to print all nonzero item taxes */
#define show_opt_costs 16   /* vbose code to show the best k costs at 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 and end */
#define show_warnings 1024   /* vbose code for reporting options without primaries */
#define maxk 15000   /* upper limit on parameter k */
```

6. ⟨ Global variables 6 ⟩ ≡

```
int vbose = show_basics + show_opt_costs + show_warnings;   /* level of verbosity */
int spacing;   /* solution t is output if t is a multiple of spacing */
int show_choices_max = 1000000;   /* above this level, show_choices is ignored */
int show_choices_gap = 1000000;   /* below level maxl − show_choices_gap, show_details is ignored */
int show_levels_max = 1000000;   /* above this level, state reports stop */
int maxl = 0;   /* maximum level actually reached */
char buf[bufsize];   /* input buffer */
ullng sortbuf[sortbufsize];   /* short buffer for sorting */
ullng count;   /* solutions found so far */
ullng options;   /* options seen so far */
ullng imems, mems;   /* mem counts */
ullng updates;   /* update counts */
ullng cleansings;   /* cleansing counts */
ullng bytes;   /* memory used by main data structures */
ullng nodes;   /* total number of branch nodes initiated */
ullng thresh = 10000000000;   /* report when mems exceeds this, if delta ≠ 0 */
ullng delta = 10000000000;   /* report every delta or so mems */
ullng maxcount = #ffffffffffffffff;   /* stop after finding this many solutions */
ullng timeout = #1fffffffffffffff;   /* give up after this many mems */
FILE *shape_file;   /* file for optional output of search tree shape */
char *shape_name;   /* its name */
int kthresh = 1;   /* this many mincost solutions will be found, if possible */
int lenthresh = 10;   /* at most this many options checked per item */
int zgiven;   /* this many primary items per option, if specified */
char Zchars[8];   /* prefix characters specified by parameter Z */
int ppgiven;   /* desired footprint of primary items in every option */
```

See also sections 13 and 41.

This code is used in section 2.

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

```
⟨Process the command line 7⟩ ≡
  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 'k': k |= (sscanf(argv[j] + 1, ""O"d", &kthresh) − 1);
      if (kthresh < 1 ∨ kthresh > maxk) {
        fprintf(stderr, "Sorry,␣parameter␣k␣must␣be␣between␣1␣and␣"O"d!\n", maxk);
        exit(−1);
      }
      break;
    case 'Z':
      if (strlen(argv[j]) > 8) {
        fprintf(stderr, "Sorry,␣parameter␣Z␣must␣specify␣at␣most␣7␣prefix␣characters!\n");
        k |= 1;
      } else sprintf(Zchars, "%s", argv[j] + 1);
      break;
    case 'z': k |= (sscanf(argv[j] + 1, ""O"d", &zgiven) − 1); break;
    case 'h': k |= (sscanf(argv[j] + 1, ""O"d", &lenthresh) − 1); break;
    case 'd': k |= (sscanf(argv[j] + 1, ""O"lld", &delta) − 1), thresh = delta; break;
    case 'c': k |= (sscanf(argv[j] + 1, ""O"d", &show_choices_max) − 1); break;
    case 'C': k |= (sscanf(argv[j] + 1, ""O"d", &show_levels_max) − 1); break;
    case 'l': k |= (sscanf(argv[j] + 1, ""O"d", &show_choices_gap) − 1); break;
    case 't': k |= (sscanf(argv[j] + 1, ""O"lld", &maxcount) − 1); break;
    case 'T': k |= (sscanf(argv[j] + 1, ""O"lld", &timeout) − 1); break;
    case 'S': shape_name = argv[j] + 1, shape_file = fopen(shape_name, "w");
      if (¬shape_file)
        fprintf(stderr, "Sorry,␣I␣can't␣open␣file␣`"O"s'␣for␣writing!\n", shape_name);
      break;
    default: k = 1;    /* unrecognized command-line option */
    }
  if (k) {
    fprintf(stderr,
        "Usage:␣"O"s␣[v<n>]␣[m<n>]␣[k<n>]␣[Z<ABC>]␣[z<n>]␣[h<n>]""␣[d<n>]␣[c<n>]␣[C<n\
        >]␣[l<n>]␣[t<n>]␣[T<n>]␣[S<bar>]␣<␣foo.dlx\n", argv[0]);
    exit(−1);
  }
```

This code is used in section 2.

8. ⟨Close the files 8⟩ ≡

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

This code is used in section 4.

9. ⟨Print the *kthresh* best costs found 9⟩ ≡

```
{
  fprintf(stderr, "The␣optimum␣cost"O"s", kthresh ≡ 1 ? "␣is" : "s␣are:\n");
  ⟨Sort the bestcost heap in preparation for final printing 57⟩;
  for (k = 1, tmpcost = infcost; k ≤ kthresh ∧ bestcost[k] < infcost; k++) {
    if (tmpcost ≡ totaltax + bestcost[k]) r++;
    else {
      ⟨Print a line (except the first time) 10⟩;
      tmpcost = totaltax + bestcost[k], r = 0;
    }
  }
  ⟨Print a line (except the first time) 10⟩;
}
```

This code is used in section 4.

10. ⟨Print a line (except the first time) 10⟩ ≡

```
if (tmpcost ≠ infcost) {
  if (r) fprintf(stderr, "␣$"O"llu␣(repeated␣"O"d␣times)\n", tmpcost, r + 1);
  else fprintf(stderr, "␣$"O"llu\n", tmpcost);
}
```

This code is used in section 9.

11. Data structures. Each item of the input matrix is represented by an **item** struct, and each option is represented as a list of **node** structs. There's one node for each nonzero entry in the matrix.

More precisely, the nodes of individual options appear sequentially, with "spacer" nodes between them. The nodes are also linked circularly with respect to each item, in doubly linked lists. The item lists each include a header node, but the option lists do not. Item header nodes are aligned with an **item** struct, which contains further info about the item.

Each node contains five important fields, and one other that's unused but might be important in extensions of this program. Two are the pointers *up* and *down* of doubly linked lists, already mentioned. A third points directly to the item containing the node. A fourth specifies a color, or zero if no color is specified. A fifth specifies the cost of the option in which this node occurs. A sixth points to the spacer at the end of the option; that one is currently set, but not looked at.

A "pointer" is an array index, not a C reference (because the latter would occupy 64 bits and waste cache space). The *cl* array is for **item** structs, and the *nd* array is for **node**s. I assume that both of those arrays are small enough to be allocated statically. (Modifications of this program could do dynamic allocation if needed.) The header node corresponding to $cl[c]$ is $nd[c]$.

Notice that each **node** occupies three octabytes. We count one mem for a simultaneous access to the *up* and *down* fields, or for a simultaneous access to the *itm* and *color* fields.

Although the item-list pointers are called *up* and *down*, they need not correspond to actual positions of matrix entries. The elements of each item list can appear in any order, so that one option needn't be consistently "above" or "below" another. Indeed, we will sort each option list of a primary item from top to bottom in order of nondecreasing cost.

This program doesn't change the *itm* fields after they've first been set up. But the *up* and *down* fields will be changed frequently, although preserving relative order.

Exception: In the node $nd[c]$ that is the header for the list of item c, we use the *cost* field to hold the "tax" on that item—for diagnostic purposes only, not as part of the algorithm's decision-making. We also might use its *color* field for special purposes. The alternative names *len* for *itm*, *aux* for *color*, and *tax* for *cost* are used in the code so that this nonstandard semantics will be more clear.

A *spacer* node has $itm \le 0$. Its *up* field points to the start of the preceding option; its *down* field points to the end of the following option. Thus it's easy to traverse an option circularly, in either direction.

The *color* field of a node is set to -1 when that node has been cleansed. In such cases its original color appears in the item header. (The program uses this fact only for diagnostic outputs.)

```
#define len itm    /* item list length (used in header nodes only) */
#define aux color    /* an auxiliary quantity (used in header nodes only) */
#define tax cost    /* item tax (used in header nodes only) */
```

```
⟨Type definitions 11⟩ ≡
  typedef struct node_struct {
    int up, down;    /* predecessor and successor in item list */
    int itm;    /* the item containing this node */
    int color;    /* the color specified by this node, if any */
    ullng cost;    /* the cost of the option containing this node */
  } node;
```

See also section 12.

This code is used in section 2.

12. Each **item** struct contains three fields: The *name* is the user-specified identifier; *next* and *prev* point to adjacent items, when this item is part of a doubly linked list.

As backtracking proceeds, nodes will be deleted from item lists when their option has been hidden by other options in the partial solution. But when backtracking is complete, the data structures will be restored to their original state.

We count one mem for a simultaneous access to the *prev* and *next* fields.

```
⟨ Type definitions 11 ⟩ +≡
  typedef struct itm_struct {
    char name[8];   /* symbolic identification of the item, for printing */
    int prev, next;   /* neighbors of this item */
  } item;
```

13.
```
⟨ Global variables 6 ⟩ +≡
  node *nd;   /* the master list of nodes */
  int last_node;   /* the first node in nd that's not yet used */
  item cl[max_cols + 2];   /* the master list of items */
  int second = max_cols;   /* boundary between primary and secondary items */
  int last_itm;   /* the first item in cl that's not yet used */
  ullng totaltax;   /* the sum of all taxes assessed */
```

14. One **item** struct is called the root. It serves as the head of the list of items that need to be covered, and is identifiable by the fact that its *name* is empty.

```
#define root 0   /* cl[root] is the gateway to the unsettled items */
```

15. 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, given a cost threshold to measure the length of that list.

```
⟨ Subroutines 15 ⟩ ≡
  void print_option(int p, FILE *stream, ullng thresh)
  {
    register int c, j, k, q;
    register ullng s;
    c = nd[p].itm;
    if (p < last_itm ∨ p ≥ last_node ∨ c ≤ 0) {
      fprintf(stderr, "Illegal␣option␣"O"d!\n", p);
      return;
    }
    for (q = p, s = 0; ; ) {
      fprintf(stream, "␣"O".8s", cl[nd[q].itm].name);
      if (nd[q].color) fprintf(stream, ":"O"c", nd[q].color > 0 ? nd[q].color : nd[nd[q].itm].color);
      s += nd[nd[q].itm].tax;
      q++;
      if (nd[q].itm ≤ 0) q = nd[q].up;      /* −nd[q].itm is actually the option number */
      if (q ≡ p) break;
    }
    for (q = nd[c].down, k = 1; q ≠ p; k++) {
      if (q ≡ c) {
        fprintf(stream, "␣(?)"); goto finish;      /* option not in its item list! */
      } else q = nd[q].down;
    }
    for (q = nd[c].down, j = 0; q ≥ last_itm; q = nd[q].down, j++)
      if (nd[q].cost ≥ thresh) break;
    fprintf(stream, "␣("O"d␣of␣"O"d)", k, j);
  finish: if (s + nd[p].cost) fprintf(stream, "␣$"O"llu␣["O"llu]\n", s + nd[p].cost, nd[p].cost);
    else fprintf(stream, "\n");
  }
  void prow(int p)
  {
    print_option(p, stderr, infcost);
  }
```

See also sections 16, 17, 43, 44, 47, 48, 59, 60, and 62.

This code is used in section 2.

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

```
⟨Subroutines 15⟩ +≡
  void print_itm(int c)
  {
    register int p;
    if (c < root ∨ c ≥ last_itm) {
      fprintf(stderr, "Illegal␣item␣"O"d!\n", c);
      return;
    }
    if (c < second) fprintf(stderr, "Item␣"O".8s,␣neighbors␣"O".8s␣and␣"O".8s:\n", cl[c].name,
          cl[cl[c].prev].name, cl[cl[c].next].name);
    else fprintf(stderr, "Item␣"O".8s:\n", cl[c].name);
    for (p = nd[c].down; p ≥ last_itm; p = nd[p].down) prow(p);
  }
```

17. 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 15⟩ +≡
  void sanity(void)
  {
    register int k, p, q, pp, qq, t;
    for (q = root, p = cl[q].next; ; q = p, p = cl[p].next) {
      if (cl[p].prev ≠ q) fprintf(stderr, "Bad␣prev␣field␣at␣itm␣"O".8s!\n", cl[p].name);
      if (p ≡ root) break;
      ⟨Check item p 18⟩;
    }
  }
```

18. ⟨Check item *p* 18⟩ ≡

```
  for (qq = p, pp = nd[qq].down, k = 0; ; qq = pp, pp = nd[pp].down, k++) {
    if (nd[pp].up ≠ qq) fprintf(stderr, "Bad␣up␣field␣at␣node␣"O"d!\n", pp);
    if (pp ≡ p) break;
    if (nd[pp].itm ≠ p) fprintf(stderr, "Bad␣itm␣field␣at␣node␣"O"d!\n", pp);
    if (qq > p ∧ nd[pp].cost < nd[qq].cost)
      fprintf(stderr, "Costs␣out␣of␣order␣at␣node␣"O"d!\n", pp);
  }
  if (p < second ∧ nd[p].len ≠ k) fprintf(stderr, "Bad␣len␣field␣in␣item␣"O".8s!\n", cl[p].name);
```

This code is used in section 17.

19. 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.

```
#define panic(m)
            { fprintf(stderr, ""O"s!\n"O"d:␣"O".99s\n", m, p, buf); exit(−666); }
⟨Input the item names 19⟩ ≡
  nd = (node *) calloc(max_nodes, sizeof(node));
  if (¬nd) {
    fprintf(stderr, "I␣couldn’t␣allocate␣space␣for␣"O"d␣nodes!\n", max_nodes);
    exit(−666);
  }
  if (max_nodes ≤ 2 * max_cols) {
    fprintf(stderr, "Recompile␣me:␣max_nodes␣must␣exceed␣twice␣max_cols!\n");
    exit(−999);
  }     /* every item will want a header node and at least one other node */
  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]; ) {
    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, cl[last_itm].name[j] = buf[p + j];
    }
    if (j ≡ 8 ∧ ¬isspace(buf[p + j])) panic("Item␣name␣too␣long");
    ⟨Check for duplicate item name 20⟩;
    ⟨Initialize last_itm to a new item with an empty list 21⟩;
    for (p += j + 1; o, isspace(buf[p]); p++) ;
    if (buf[p] ≡ '|') {
      if (second ≠ max_cols) panic("Item␣name␣line␣contains␣|␣twice");
      second = last_itm;
      for (p++; o, isspace(buf[p]); p++) ;
    }
  }
  if (second ≡ max_cols) second = last_itm;
  oo, cl[last_itm].prev = last_itm − 1, cl[last_itm − 1].next = last_itm;
  oo, cl[second].prev = last_itm, cl[last_itm].next = second;
    /* this sequence works properly whether or not second = last_itm */
  oo, cl[root].prev = second − 1, cl[second − 1].next = root;
  last_node = last_itm;     /* reserve all the header nodes and the first spacer */
    /* we have nd[last_node].itm = 0 in the first spacer */
```

This code is used in section 3.

20. ⟨Check for duplicate item name 20⟩ ≡

```
  for (k = 1; o, strncmp(cl[k].name, cl[last_itm].name, 8); k++) ;
  if (k < last_itm) panic("Duplicate␣item␣name");
```

This code is used in section 19.

21. ⟨Initialize *last_itm* to a new item with an empty list 21⟩ ≡

```
if (last_itm > max_cols) panic("Too␣many␣items");
oo, cl[last_itm − 1].next = last_itm, cl[last_itm].prev = last_itm − 1;     /* nd[last_itm].len = 0 */
o, nd[last_itm].up = nd[last_itm].down = last_itm;
last_itm++;
```

This code is used in section 19.

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

⟨Input the options 22⟩ ≡

```
⟨Set ppgiven from parameters Z and z 28⟩;
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 */
  tmpcost = 0;
  for (pp = 0; buf[p]; ) {
    for (j = 0; j < 8 ∧ (o, ¬isspace(buf[p + j])) ∧ buf[p + j] ≠ ':'; j++)
      o, cl[last_itm].name[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");
    if (j < 8) o, cl[last_itm].name[j] = '\0';
    ⟨Create a node for the item named in buf[p] 25⟩;
    if (buf[p + j] ≠ ':') o, nd[last_node].color = 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].color = buf[p + j + 1];
      p += 2;
    } else panic("Primary␣item␣must␣be␣uncolored");
    ⟨Skip to next item, accruing cost information if any 23⟩;
  }
  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 27⟩;
      last_node−−;
    }
  } else {
    ⟨Check for consistency with parameters Z and z 29⟩;
    ⟨Insert the cost into each item of this option 24⟩;
    o, nd[i].down = last_node;
    last_node++;     /* create the next spacer */
    if (last_node ≡ max_nodes) panic("Too␣many␣nodes");
    options++;
    o, nd[last_node].up = i + 1;
    o, nd[last_node].itm = −options;
  }
}
```

This code is used in section 3.

23. ⟨ Skip to next item, accruing cost information if any 23 ⟩ ≡

```
while (1) {
  register ullng d;
  for (p += j + 1; o, isspace(buf[p]); p++) ;
  if (buf[p] ≠ '|') break;
  if (buf[p + 1] < '0' ∨ buf[p + 1] > '9') panic("Option␣cost␣should␣be␣a␣decimal␣number");
  for (j = 1, d = 0; o, ¬isspace(buf[p + j]); j++) {
    if (buf[p + j] < '0' ∨ buf[p + j] > '9') panic("Illegal␣digit␣in␣option␣cost");
    d = 10 ∗ d + buf[p + j] − '0';
  }
  tmpcost += d;
}
```

This code is used in section 22.

24. ⟨ Insert the cost into each item of this option 24 ⟩ ≡

```
for (j = i + 1; j ≤ last_node; j++) o, nd[j].cost = tmpcost;
```

This code is used in section 22.

25. ⟨ Create a node for the item named in *buf*[*p*] 25 ⟩ ≡

```
for (k = 0; o, strncmp(cl[k].name, cl[last_itm].name, 8); k++) ;
if (k ≡ last_itm) panic("Unknown␣item␣name");
if (o, nd[k].aux ≥ i) panic("Duplicate␣item␣name␣in␣this␣option");
last_node++;
if (last_node ≡ max_nodes) panic("Too␣many␣nodes");
o, nd[last_node].itm = k;
if (k < second) ⟨ Adjust pp for parameters Z and z 30 ⟩;
o, t = nd[k].len + 1;
⟨ Insert node last_node into the list for item k 26 ⟩;
```

This code is used in section 22.

26. Insertion of a new node is simple. Before taxes have been computed, we set only the *up* links of each item list.

We store the position of the new node into *nd*[*k*].*aux*, so that the test for duplicate items above will be correct.

⟨ Insert node *last_node* into the list for item *k* 26 ⟩ ≡

```
o, nd[k].len = t;    /∗ store the new length of the list ∗/
nd[k].aux = last_node;    /∗ no mem charge for aux after len ∗/
o, r = nd[k].up;    /∗ the "bottom" node of the item list ∗/
oo, nd[k].up = last_node, nd[last_node].up = r;
```

This code is used in section 25.

27. ⟨ Remove *last_node* from its item list 27 ⟩ ≡

```
o, k = nd[last_node].itm;
oo, nd[k].len−−, nd[k].aux = i − 1;
oo, nd[k].up = nd[last_node].up;
```

This code is used in section 22.

28. When the user has used the `Z` parameter to specify special prefix characters, we want to check that each option conforms to that specification.

The rightmost bits of variable *pp* will indicate which prefixes have been seen so far. The other bits of *pp* will count active items that don't have a `Z`-specified prefix.

```
⟨Set ppgiven from parameters Z and z 28⟩ ≡
  if (o, Zchars[0]) {
    for (r = 1; Zchars[r]; r++) ;
    ppgiven = (1 ≪ r) − 1 + (zgiven ≪ 8);
  } else ppgiven = zgiven ≪ 8;
```

This code is used in section 22.

29.
```
⟨Check for consistency with parameters Z and z 29⟩ ≡
  if (ppgiven) {
    if (zgiven ∧ ((pp ≫ 8) ≠ zgiven)) fprintf(stderr,
          "Option␣has␣"O"d␣non-Z␣primary␣items,␣not␣"O"d:␣"O"s", pp ≫ 8, zgiven, buf);
    if ((pp ⊕ ppgiven) & #ff) {
      for (r = 0; Zchars[r]; r++)
        if ((pp & (1 ≪ r)) ≡ 0) fprintf(stderr, "Option␣lacks␣a␣"O"c␣item:␣"O"s", Zchars[r], buf);
    }
  }
```

This code is used in section 22.

30.
```
⟨Adjust pp for parameters Z and z 30⟩ ≡
  {
    for (r = 0; Zchars[r]; r++)
      if (Zchars[r] ≡ cl[last_itm].name[0]) break;
    if (Zchars[r]) {
      if (pp & (1 ≪ r)) fprintf(stderr, "Option␣has␣two␣"O"c␣items:␣"O"s", Zchars[r], buf);
      else pp += 1 ≪ r;
    } else pp += 1 ≪ 8;
  }
```

This code is used in section 25.

31. We look at the option list for every primary item, in turn, to find an option with smallest cost. If that cost *minc* is positive, we "tax" the item by *minc*, and subtract *minc* from the cost of all options that contain this item.

If an item has no options, its tax is infinite. (But nobody ever gets to collect it.)

```
#define infcost ((ullng) −1)   /* "infinite" cost */
⟨Assign taxes 31⟩ ≡
  for (k = 1; k < second; k++) {
    register ullng minc;
    for (p = nd[k].up, minc = infcost; p > k ∧ minc; o, p = nd[p].up)
      if (o, nd[p].cost < minc) minc = nd[p].cost;
    if (minc) {
      if (vbose & show_taxes) fprintf(stderr, "␣"O".8s␣tax=$"O"llu\n", cl[k].name, minc);
      totaltax += minc;
      for (p = nd[k].up; p > k; o, p = nd[p].up) {
        for (q = p + 1; ; ) {
          o, cc = nd[q].itm;
          if (cc ≤ 0) o, q = nd[q].up;
          else {
            oo, nd[q].cost −= minc;
            if (q ≡ p) break;
            q++;
          }
        }
      }
      nd[k].tax = minc;   /* for documentation only, so no mem charged */
    }
  }
  if (totaltax ∧ (vbose & show_taxes)) fprintf(stderr, "␣(total␣tax␣is␣$"O"llu)\n", totaltax);
```

This code is used in section 3.

32. We use the "natural list merge sort," namely Algorithm 5.2.4L as modified by exercise 5.2.4–12.

```
⟨Sort the item lists 32⟩ ≡
  for (k = 1; k < last_itm; k++) {
  l1: o, p = nd[k].up, q = nd[p].up;
    for (o, t = root; q > k; o, p = q, q = nd[p].up)   /* one mem charged for nd[p].cost */
      if (o, nd[p].cost < nd[q].cost) nd[t].up = −q, t = p;
    if (t ≠ root) ⟨Sort item list k 34⟩;
    ⟨Make the down links consistent with the up links 33⟩;
  }
```

This code is used in section 3.

33. ⟨Make the *down* links consistent with the *up* links 33⟩ ≡

```
  for (o, p = k, q = nd[p].up; q > k; o, p = q, q = nd[p].up) o, nd[q].down = p;
  oo, nd[p].up = k, nd[k].down = p;
```

This code is used in section 32.

34. The item list is now divided into sorted sublists, separated by links that have temporarily been negated.

The sorted sublists are merged, two by two. List t is "above" list s; hence the sorting is stable with respect to nodes of equal cost.

```
⟨Sort item list $k$ 34⟩ ≡
  {
    $oo, nd[t].up = nd[p].up = 0$;    /* terminate the last two sublists with a null link */
  $l2$: while ($o, nd[root].up$) {    /* begin new pass */
      $oo, s = k, t = root, p = nd[s].up, q = -nd[root].up$;    /* mem charged for $nd[p].cost$ */
    $l3$: if ($o, nd[p].cost < nd[q].cost$) goto $l6$;
    $l4$: ⟨Advance $p$ 35⟩;
    $l6$: ⟨Advance $q$ 36⟩;
    $l8$: $p = -p, q = -q$;
      if ($q$) goto $l3$;
      $oo, nd[s].up = -p, nd[t].up = 0$;    /* end of pass */
    }
  }
```

This code is used in section 32.

35. ⟨Advance p 35⟩ ≡

```
  $o, nd[s].up = (nd[s].up \le 0 \ ?\ -p : p)$;
  $o, s = p, p = nd[p].up$;
  if ($p > 0$) goto $l3$;
$l5$: $o, nd[s].up = q, s = t$;
  for ( ; $q > 0$; $o, q = nd[q].up$) $t = q$;    /* move $q$ to the end of its sublist */
  goto $l8$;    /* both sublists have now been merged */
```

This code is used in section 34.

36. ⟨Advance q 36⟩ ≡

```
  $o, nd[s].up = (nd[s].up \le 0 \ ?\ -q : q)$;
  $o, s = q, q = nd[q].up$;
  if ($q > 0$) goto $l3$;
$l7$: $o, nd[s].up = p, s = t$;
  for ( ; $p > 0$; $o, p = nd[p].up$) $t = p$;    /* move $p$ to the end of its sublist */
  goto $l8$;    /* both sublists have now been merged */
```

This code is used in section 34.

37. ⟨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, second − 1, last_itm − second, last_node − last_itm);
```

This code is used in section 3.

38. The item lengths after input should agree with the item lengths after this program has finished. I print them (on request), in order to provide some reassurance that the algorithm isn't badly screwed up.

⟨ Report the item totals 38 ⟩ ≡

```
{
  fprintf(stderr, "Item␣totals:");
  for (k = 1; k < last_itm; k++) {
    if (k ≡ second) fprintf(stderr, "␣|");
    fprintf(stderr, "␣"O"d", nd[k].len);
  }
  fprintf(stderr, "\n");
}
```

This code is used in sections 3 and 4.

39. The dancing. Our strategy for generating all exact covers will be to repeatedly choose always the item that appears to be hardest to cover, namely the item with shortest list, from all items that still need to be covered. And we explore all possibilities via depth-first search.

The neat part of this algorithm is the way the lists are maintained. Depth-first search means last-in-first-out maintenance of data structures; and it turns out that we need no auxiliary tables to undelete elements from lists when backing up. The nodes removed from doubly linked lists remember their former neighbors, because we do no garbage collection.

The basic operation is "covering an item." This means removing it from the list of items needing to be covered, and "hiding" its options: removing nodes from other lists whenever they belong to an option of a node in this item's list.

```
⟨Solve the problem 39⟩ ≡
  ⟨Initialize for level 0 40⟩;
forward: nodes++;
  if (vbose & show_profile) profile[level]++;
  if (sanity_checking) sanity();
  ⟨Do special things if enough mems have accumulated 42⟩;
  ⟨If the remaining cost is clearly too high, goto backdown 49⟩;
  ⟨Set best_itm to the best item for branching, or goto backdown 53⟩;
  o, partcost[level] = curcost;
  oo, cur_node = choice[level] = nd[best_itm].down;
  o, nextcost = curcost + nd[cur_node].cost;
  o, coverthresh0[level] = cutoffcost − nextcost;     /* known to be positive */
  cover(best_itm, coverthresh0[level]);
advance: if ((vbose & show_choices) ∧ level < show_choices_max) {
    fprintf(stderr, "L"O"d:", level);
    print_option(cur_node, stderr, cutoffcost − curcost);
  }
  ⟨Cover all other items of cur_node 45⟩;
  if (o, cl[root].next ≡ root) ⟨Visit a solution and goto recover 55⟩;
  if (++level > maxl) {
    if (level ≥ max_level) {
      fprintf(stderr, "Too␣many␣levels!\n");
      exit(−4);
    }
    maxl = level;
  }
  curcost = nextcost;
  goto forward;
backup: o, uncover(best_itm, coverthresh0[level]);
backdown: if (level ≡ 0) goto done;
  level−−;
  oo, cur_node = choice[level], best_itm = nd[cur_node].itm;
  o, curcost = partcost[level];
recover: ⟨Uncover all other items of cur_node 46⟩;
  oo, cur_node = choice[level] = nd[cur_node].down;
  if (cur_node ≡ best_itm) goto backup;
  o, nextcost = curcost + nd[cur_node].cost;
  if (nextcost ≥ cutoffcost) goto backup;
  goto advance;
```

This code is used in section 2.

40. ⟨ Initialize for level 0 40 ⟩ ≡

```
if (zgiven) {
  for (r = 0; Zchars[r]; r++) ;
  if ((second − 1) mod (zgiven + r)) {
    fprintf(stderr, "There␣are␣"O"d␣primary␣items,␣but␣z="O"d␣and␣Z="O"s!\n", second − 1,
        zgiven, Zchars);
    goto done;
  }
}
level = 0;
for (k = 0; k < kthresh; k++) o, bestcost[k] = infcost;
cutoffcost = infcost;
curcost = 0;
```

This code is used in section 39.

41. ⟨ Global variables 6 ⟩ +≡

```
int level;    /* number of choices in current partial solution */
int choice[max_level];    /* the node chosen on each level */
ullng profile[max_level];    /* number of search tree nodes on each level */
ullng partcost[max_level];    /* the net cost so far, on each level */
ullng coverthresh0[max_level], coverthresh[max_level];    /* historic thresholds */
ullng bestcost[maxk + 1];    /* the best kthresh net costs known so far */
ullng cutoffcost;    /* bestcost[0], the cost we need to beat */
ullng cumcost[7];    /* accumulated costs for the Z prefix characters */
int solutionsize;    /* the number of options per solution, if fixed and known */
```

42. ⟨ Do special things if enough *mems* have accumulated 42 ⟩ ≡

```
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 39.

43. When an option is hidden, it leaves all lists except the list of the item that is being covered. Thus a node is never removed from a list twice.

We can save time by not removing nodes from secondary items that have been purified. (Such nodes have *color* < 0. Note that *color* and *itm* are stored in the same octabyte; hence we pay only one mem to look at them both.)

We save even more time by not updating the *len* fields of secondary items.

It's not necessary to hide all the options of the list being covered. Only the options whose cost is below a given threshold will ever be relevant, since we seek only minimum-cost solutions.

```
⟨ Subroutines 15 ⟩ +≡
  void cover(int c, ullng thresh)
  {
    register int cc, l, r, rr, nn, uu, dd, t;
    o, l = cl[c].prev, r = cl[c].next;
    oo, cl[l].next = r, cl[r].prev = l;
    updates++;
    for (o, rr = nd[c].down; rr ≥ last_itm; o, rr = nd[rr].down) {
      if (o, nd[rr].cost ≥ thresh) break;
      for (nn = rr + 1; nn ≠ rr; ) {
        if (o, nd[nn].color ≥ 0) {
          o, uu = nd[nn].up, dd = nd[nn].down;
          cc = nd[nn].itm;
          if (cc ≤ 0) {
            nn = uu; continue;
          }
          oo, nd[uu].down = dd, nd[dd].up = uu;
          updates++;
          if (cc < second) oo, nd[cc].len--;
        }
        nn++;
      }
    }
  }
```

44. I used to think that it was important to uncover an item by processing its options from bottom to top, since covering was done from top to bottom. But while writing this program I realized that, amazingly, no harm is done if the options are processed again in the same order. It's easier to go down than up, because of the cutoff threshold; hence that observation is good news. Whether we go up or down, the pointers execute an exquisitely choreographed dance that returns them almost magically to their former state.

Of course we must be careful to use exactly the same thresholds when uncovering as we did when covering, even though the *cutoffcost* in this program is a moving target.

```
⟨Subroutines 15⟩ +≡
  void uncover(int c, ullng thresh)
  {
    register int cc, l, r, rr, nn, uu, dd, t;
    for (o, rr = nd[c].down; rr ≥ last_itm; o, rr = nd[rr].down) {
      if (o, nd[rr].cost ≥ thresh) break;
      for (nn = rr + 1; nn ≠ rr; ) {
        if (o, nd[nn].color ≥ 0) {
          o, uu = nd[nn].up, dd = nd[nn].down;
          cc = nd[nn].itm;
          if (cc ≤ 0) {
            nn = uu; continue;
          }
          oo, nd[uu].down = nd[dd].up = nn;
          if (cc < second) oo, nd[cc].len++;
        }
        nn++;
      }
    }
    o, l = cl[c].prev, r = cl[c].next;
    oo, cl[l].next = cl[r].prev = c;
  }
```

45. ⟨Cover all other items of *cur_node* 45⟩ ≡

```
  o, coverthresh[level] = cutoffcost − nextcost;
  for (pp = cur_node + 1; pp ≠ cur_node; ) {
    o, cc = nd[pp].itm;
    if (cc ≤ 0) o, pp = nd[pp].up;
    else {
      if (¬nd[pp].color) cover(cc, coverthresh[level]);
      else if (nd[pp].color > 0) purify(pp, coverthresh[level]);
      pp++;
    }
  }
```

This code is used in section 39.

46. We must go leftward as we uncover the items, because we went rightward when covering them.

```
⟨Uncover all other items of cur_node 46⟩ ≡
  o;      /* charge one mem for putting coverthresh[level] in a register */
  for (pp = cur_node − 1; pp ≠ cur_node; ) {
    o, cc = nd[pp].itm;
    if (cc ≤ 0) o, pp = nd[pp].down;
    else {
      if (¬nd[pp].color) uncover(cc, coverthresh[level]);
      else if (nd[pp].color > 0) unpurify(pp, coverthresh[level]);
      pp−−;
    }
  }
```

This code is used in section 39.

47. When we choose an option that specifies colors in one or more items, we "purify" those items by removing all incompatible options. All options that want the chosen color in a purified item are temporarily given the color code −1 so that they won't be purified again.

```
⟨Subroutines 15⟩ +≡
  void purify(int p, ullng thresh)
  {
    register int cc, rr, nn, uu, dd, t, x;
    o, cc = nd[p].itm, x = nd[p].color;
    nd[cc].color = x;      /* no mem charged, because this is for print_option only */
    cleansings++;
    for (o, rr = nd[cc].down; rr ≥ last_itm; o, rr = nd[rr].down) {
      if (o, nd[rr].cost ≥ thresh) break;
      if (o, nd[rr].color ≠ x) {
        for (nn = rr + 1; nn ≠ rr; ) {
          if (o, nd[nn].color ≥ 0) {
            o, uu = nd[nn].up, dd = nd[nn].down;
            cc = nd[nn].itm;
            if (cc ≤ 0) {
              nn = uu; continue;
            }
            oo, nd[uu].down = dd, nd[dd].up = uu;
            updates++;
            if (cc < second) oo, nd[cc].len−−;
          }
          nn++;
        }
      } else if (rr ≠ p) cleansings++, o, nd[rr].color = −1;
    }
  }
```

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

```
⟨Subroutines 15⟩ +≡
  void unpurify(int p, ullng thresh)
  {
    register int cc, rr, nn, uu, dd, t, x;
    o, cc = nd[p].itm, x = nd[p].color;      /* there's no need to clear nd[cc].color */
    for (o, rr = nd[cc].down; rr ≥ last_itm; o, rr = nd[rr].down) {
      if (o, nd[rr].cost ≥ thresh) break;
      if (o, nd[rr].color < 0) o, nd[rr].color = x;
      else if (rr ≠ p) {
        for (nn = rr + 1; nn ≠ rr; ) {
          if (o, nd[nn].color ≥ 0) {
            o, uu = nd[nn].up, dd = nd[nn].down;
            cc = nd[nn].itm;
            if (cc ≤ 0) {
              nn = uu; continue;
            }
            oo, nd[uu].down = nd[dd].up = nn;
            if (cc < second) oo, nd[cc].len++;
          }
          nn++;
        }
      }
    }
  }
```

49. Here's where we use the `Z` and `z` heuristics to provide lower bounds that don't apply in general.

```
⟨If the remaining cost is clearly too high, goto backdown 49⟩ ≡
  if (ppgiven ∧ cutoffcost ≠ infcost) {
    if (zgiven > 1) {
      if (second − level ∗ zgiven ≤ sortbufsize + 1) pp = zgiven;
      else if (ppgiven & #ff) pp = 0;
      else pp = −1;
    } else pp = zgiven;
    if (pp ≥ 0) ⟨Go to backdown if the remaining min costs are too high 50⟩
  }
```

This code is used in section 39.

50. ⟨ Go to *backdown* if the remaining min costs are too high 50 ⟩ ≡

```
{
  register ullng newcost, oldcost, acccost;
  acccost = curcost;
  for (r = 0; Zchars[r]; r++) o, cumcost[r] = curcost;
  for (o, k = cl[root].next, t = 0; k ≠ root; o, k = cl[k].next) {
    o, p = nd[k].down;
    if (p < last_itm) {
      if (explaining)
        fprintf(stderr, "(Level␣"O"d,␣"O".8s's␣list␣is␣empty)\n", level, cl[k].name);
      goto backdown;
    }
    oo, cc = cl[k].name[0], tmpcost = nd[p].cost;
    for (r = 0; Zchars[r]; r++)
      if (Zchars[r] ≡ cc) break;
    if (Zchars[r]) ⟨ Include tmpcost in cumcost[r] 51 ⟩
    else if (pp) ⟨ Include tmpcost in acccost 52 ⟩;
  }
}
```

This code is used in section 49.

51. ⟨ Include *tmpcost* in *cumcost*[*r*] 51 ⟩ ≡

```
{
  if (o, cumcost[r] + tmpcost ≥ cutoffcost) {
    if (explaining)
      fprintf(stderr, "(Level␣"O"d,␣"O".8s's␣cost␣overflowed)\n", level, cl[k].name);
    goto backdown;
  }
  o, cumcost[r] += tmpcost;
}
```

This code is used in section 50.

52. At this point $pp = zgiven$ is a positive number z, and $cl[k]$ is one of the pp active items that doesn't begin with a `Z`-specified prefix. We also know that exactly $kz = second - 1 - level * z$ primary items are active, and that exactly k more levels must be completed before we have a solution.

The situation is simple when $z = 1$. But when $z > 1$, suppose the minimum net costs of active items are $c_1 \le c_2 \le \cdots \le c_{kz}$. Then we'll spend at least $c_z + c_{2z} + \cdots + c_{kz}$ while covering them. A cute little online algorithm computes this lower bound nicely.

⟨Include *tmpcost* in *acccost* 52⟩ ≡

```
{
  if (pp ≡ 1) {
    if (acccost + tmpcost ≥ cutoffcost) {
      if (explaining)
        fprintf(stderr, "(Level␣"O"d,␣"O".8s's␣cost␣overflowed)\n", level, cl[k].name);
      goto backdown;
    }
    acccost += tmpcost;
  } else {      /* we'll sort tmpcost into sortbuf, which has t costs already */
    for (p = t, oldcost = 0; p; p−−, oldcost = newcost) {
      o, newcost = sortbuf[sortbufsize − p];
      if (tmpcost ≤ newcost) break;
      if ((p mod pp) ≡ 0) {
        acccost += newcost − oldcost;
        if (acccost ≥ cutoffcost) {
          if (explaining)
            fprintf(stderr, "(Level␣"O"d,␣"O".8s's␣cost␣overflowed)\n", level, cl[k].name);
          goto backdown;
        }
      }
      o, sortbuf[sortbufsize − p − 1] = newcost;      /* it had been oldcost */
    }
    if ((p mod pp) ≡ 0) {
      acccost += tmpcost − oldcost;
      if (acccost ≥ cutoffcost) {
        if (explaining) fprintf(stderr, "("O".8s's␣cost␣caused␣overflow)\n", cl[k].name);
        goto backdown;
      }
    }
    o, sortbuf[sortbufsize − p − 1] = tmpcost;      /* it had been oldcost */
    t++;
  }
}
```

This code is used in section 50.

53. The “best item” is considered to be an item that minimizes the number of remaining choices. If there are several candidates, we choose the leftmost one that has maximum minimum net cost (because that cost must be paid somehow).

(This part of the program, whose logic is justified by the sorting that was done during the input phase, represents the most significant changes between DLX5 and DLX2. I imagine that the heuristics used here might be significantly improvable, especially for certain classes of problems. For example, it may be better to do a 5-way branch on expensive choices than a 2-way branch on cheap ones, because the expensive choices might quickly peter out. And more elaborate ways to derive lower bounds on the cost of covering the remaining primary items might be based on the minimum cost per item in the remaining options. For example, we could give each node a new field *optref*, which points to the spacer following its option. Then the length of this option would readily be obtained from that spacer, $nd[nd[p].optref]$. One could use the currently dormant *cost* and *optref* fields of each spacer to maintain a doubly linked list of options in order of their cost/item. But I don't have time to investigate such ideas myself.)

```
#define explaining
        ((vbose & show_details) ∧ level < show_choices_max ∧ level ≥ maxl − show_choices_gap)

⟨Set best_itm to the best item for branching, or goto backdown 53⟩ ≡
  t = max_nodes, tmpcost = 0;
  if (explaining) fprintf(stderr, "Level␣"O"d:", level);
  for (o, k = cl[root].next; k ≠ root; o, k = cl[k].next) {
    o, p = nd[k].down;
    if (p ≡ k) {      /* the item list is empty, we must backtrack */
      if (explaining) fprintf(stderr, "␣"O".8s(0)", cl[k].name);
      t = 0, best_itm = k;
      break;
    }
    o, mincost = nd[p].cost;
    if (mincost ≥ cutoffcost − curcost) {      /* no usable items, we must backtrack */
      if (explaining) fprintf(stderr, "␣"O".8s(0$"O"llu)", cl[k].name, mincost);
      t = 0, best_itm = k;
      break;
    }
    ⟨Look at the least-cost options for item k, possibly updating best_itm 54⟩;
  }
  if (explaining) fprintf(stderr, "␣branching␣on␣"O".8s("O"d)\n", cl[best_itm].name, t);
  if (shape_file) {
    fprintf(shape_file, ""O"d␣"O".8s\n", t, cl[best_itm].name);
    fflush(shape_file);
  }
  if (t ≡ 0) goto backdown;
```

This code is used in section 39.

54. At this point we know that $t \ge 1$, $p = nd[k].down \ne k$, and $mincost = nd[p].cost < cutoffcost - curcost$. Therefore k might turn out to be the new *best_itm*.

```
⟨Look at the least-cost options for item k, possibly updating best_itm 54⟩ ≡
  for (o, s = 1, p = nd[p].down; ; o, p = nd[p].down, s++) {
    if (p < last_itm ∨ (o, nd[p].cost ≥ cutoffcost − curcost)) {
      if (explaining) fprintf(stderr, "␣"O".8s("O"d$"O"llu)", cl[k].name, s, mincost);
      break;      /* there are s usable options in k's item list */
    }
    if (s ≡ t) {      /* there are more than t usable options */
      if (explaining) fprintf(stderr, "␣"O".8s(>"O"d)", cl[k].name, t);
      goto no_change;
    }
    if (s ≥ lenthresh) {      /* let's not search too far down the list */
      o, s = nd[k].len;      /* be content with an upper bound */
      if (explaining) fprintf(stderr, "␣"O".8s("O"d?$"O"llu)", cl[k].name, s, mincost);
      break;
    }
  }
  if (s < t ∨ (s ≡ t ∧ mincost > tmpcost)) t = s, best_itm = k, tmpcost = mincost;
  no_change:
```

This code is used in section 53.

55.

```
⟨Visit a solution and goto recover 55⟩ ≡
  {
    nodes++;      /* a solution is a special node, see 7.2.2–(4) */
    if (level + 1 > maxl) {
      if (level + 1 ≥ max_level) {
        fprintf(stderr, "Too␣many␣levels!\n");
        exit(−5);
      }
      maxl = level + 1;
    }
    if (vbose & show_profile) profile[level + 1]++;
    if (shape_file) {
      fprintf(shape_file, "sol\n"); fflush(shape_file);
    }
    ⟨Update cutoffcost 56⟩;
    ⟨Record solution and goto recover 58⟩;
  }
```

This code is used in section 39.

56. We remember the *kthresh* best costs found so far in a heap, with $bestcost[h] \ge bestcost[h+h+1]$ and $bestcost[h] \ge bestcost[h+h+2]$. In particular, $bestcost[0] = cutoffcost$ is the largest of these net costs, and we remove it from the heap when a new solution has been found.

When *kthresh* is even, this code uses the fact that $bestcost[kthresh] = 0$.

```
⟨Update cutoffcost 56⟩ ≡
  {
    register int h, hh;      /* a hole in the heap, and its larger successor */
    tmpcost = cutoffcost;
    for (h = 0, hh = 2; hh ≤ kthresh; hh = h + h + 2) {
      if (oo, bestcost[hh] > bestcost[hh − 1]) {
        if (nextcost < bestcost[hh]) o, bestcost[h] = bestcost[hh], h = hh;
        else break;
      } else if (nextcost < bestcost[hh − 1]) o, bestcost[h] = bestcost[hh − 1], h = hh − 1;
      else break;
    }
    o, bestcost[h] = nextcost;
    o, cutoffcost = bestcost[0];
  }
```

This code is used in section 55.

57. ⟨Sort the *bestcost* heap in preparation for final printing 57⟩ ≡

```
  for (p = kthresh; p > 2; p−−) {
    register int h, hh;      /* a hole in the heap, and its larger successor */
    nextcost = bestcost[p − 1], bestcost[p − 1] = 0, bestcost[p] = bestcost[0];
    for (h = 0, hh = 2; hh < p; hh = h + h + 2) {
      if (bestcost[hh] > bestcost[hh − 1]) {
        if (nextcost < bestcost[hh]) bestcost[h] = bestcost[hh], h = hh;
        else break;
      } else if (nextcost < bestcost[hh − 1]) bestcost[h] = bestcost[hh − 1], h = hh − 1;
      else break;
    }
    bestcost[h] = nextcost;
  }
  bestcost[p] = bestcost[0];      /* at this point p = 1 or p = 2 */
    /* now bestcost[1] ≤ bestcost[2] ≤ · · · ≤ bestcost[kthresh] */
```

This code is used in section 9.

58. ⟨Record solution and **goto** *recover* 58⟩ ≡

```
  {
    count++;
    if (spacing ∧ (count mod spacing ≡ 0)) {
      printf(""O"lld:␣(total␣cost␣$"O"llu)\n", count, totaltax + nextcost);
      for (k = 0; k ≤ level; k++) print_option(choice[k], stdout, tmpcost − partcost[k]);
      fflush(stdout);
    }
    if (count ≥ maxcount) goto done;
    goto recover;
  }
```

This code is used in section 55.

59. ⟨ Subroutines 15 ⟩ +≡

```
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, cutoffcost − partcost[l]);
    if (l ≥ show_levels_max) {
      fprintf(stderr, "␣...\n");
      break;
    }
  }
  if (cutoffcost < infcost) fprintf(stderr,
        "␣"O"lld␣solutions,␣$"O"llu,␣"O"lld␣mems,␣and␣max␣level␣"O"d␣so␣far.\n", count,
        cutoffcost + totaltax, mems, maxl);
  else fprintf(stderr, "␣"O"lld␣solutions,␣"O"lld␣mems,␣and␣max␣level␣"O"d␣so␣far.\n",
        count, mems, maxl);
}
```

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

`0, 1, ..., 9, a, b, ..., z, A, B, ..., Z.`

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

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

```
⟨Subroutines 15⟩ +≡
  void print_progress(void)
  {
    register int l, k, d, c, p;
    register double f, fd;
    if (cutoffcost < infcost) fprintf(stderr, "␣after␣"O"lld␣mems:␣"O"lld␣sols,␣$"O"llu,", mems,
          count, cutoffcost + totaltax);
    else 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;
      for (k = 1, p = nd[c].down; p ≠ choice[l]; k++, p = nd[p].down) ;
      for (d = k − 1; p ≥ last_itm; p = nd[p].down, d++)
        if (nd[p].cost ≥ cutoffcost − partcost[l]) break;
      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);
  }
```

61. ⟨Print the profile 61⟩ ≡

```
  {
    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 4.

62. ⟨Subroutines 15⟩ +≡

```
  int confusioncount;
  void confusion(char *id)
  {     /* an assertion has failed */
    if (confusioncount++ ≡ 0)     /* can fiddle with debugger */
      fprintf(stderr, "This␣can't␣happen␣(%s)!\n", id);
  }
```

63. Index.

DLX5