2* This version of the program keeps removing rows at the bottom, thereby finding all solutions that have minimax row number. (And it usually also finds a few more, before it has found the best cutoff point.)

We assume that the columns contain rows in their original order; the *up* and *down* links actually point upwards and downwards. Therefore we disable the "randomizing" feature by which columns could be linked randomly. (Randomization still does apply, however, when choosing a column for branching.)

After this program finds all the desired 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 a row from its column. A "cleansing" is the removal of a satisfied color constraint from its row. 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  500      /* at most this many rows in a solution */
#define max_cols  100000      /* at most this many columns */
#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 column names */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include "gb_flip.h"
  typedef unsigned int uint;      /* a convenient abbreviation */
  typedef unsigned long long ullng;      /* ditto */
  ⟨Type definitions 5⟩;
  ⟨Global variables 3*⟩;
  ⟨Subroutines 9⟩;
  main(int argc, char *argv[])
  {
    register int cc, i, j, k, p, pp, q, r, t, cur_node, best_col;
    ⟨Process the command line 4⟩;
    ⟨Input the column names 13⟩;
    ⟨Input the rows 16⟩;
    if (vbose & show_basics) ⟨Report the successful completion of the input phase 20⟩;
    if (vbose & show_tots) ⟨Report the column totals 21⟩;
    imems = mems, mems = 0;
    ⟨Solve the problem 22⟩;
  done: if (sanity_checking) sanity();
    if (vbose & show_tots) ⟨Report the column totals 21⟩;
    if (vbose & show_profile) ⟨Print the profile 35⟩;
    if (vbose & show_basics) {
      fprintf(stderr, "Altogether␣"O"llu␣solution"O"s,␣"O"llu+"O"llu-"O"llu␣mems,", count,
          count ≡ 1 ? "" : "s", imems, mems, lmems);
      bytes = last_col * sizeof(column) + 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);
    }
  }
```

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);
- '`s`⟨ integer ⟩' causes the algorithm to make random choices in key places (thus providing some variety, although the solutions are by no means uniformly random), and it also defines the seed for any random numbers that are used;
- '`d`⟨ integer ⟩' to sets *delta*, which causes periodic state reports on *stderr* after the algorithm has performed approximately *delta* mems since the previous report;
- '`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).

```
#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_cutoffs 8  /* vbose code to report improvements in the cutoff point */
#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 column totals at start and end */
#define show_warnings 1024  /* vbose code for reporting rows without primaries */
```

```
⟨ Global variables 3* ⟩ ≡
  int random_seed = 0;  /* seed for the random words of gb_rand */
  int randomizing;  /* has 's' been specified? */
  int vbose = show_basics + show_warnings;  /* level of verbosity */
  int spacing;  /* solution k is output if k is a multiple of spacing */
  int show_choices_max = 1000000;  /* above this level, show_choices is ignored */
  int show_choices_gap = 1000000;  /* below level maxl − show_choices_gap, show_details is ignored */
  int show_levels_max = 1000000;  /* above this level, state reports stop */
  int maxl = 0;  /* maximum level actually reached */
  char buf[bufsize];  /* input buffer */
  ullng count;  /* solutions found so far */
  ullng rows;  /* rows seen so far */
  ullng imems, mems, lmems;  /* 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 = 0;  /* report when mems exceeds this, if delta ≠ 0 */
  ullng delta = 0;  /* report every delta or so mems */
  ullng maxcount = #ffffffffffffffff;  /* stop after finding this many solutions */
  ullng timeout = #1fffffffffffffff;  /* give up after this many mems */
```

See also sections 7* and 23.

This code is used in section 2*.

7* ⟨ 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 */
column cl[max_cols + 2];   /* the master list of columns */
int second = max_cols;   /* boundary between primary and secondary columns */
int last_col;   /* the first column in cl that's not yet used */
int cutoff = max_nodes;   /* nodes after this point have essentially disappeared */
```

11* 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 9 ⟩ +≡
  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␣col␣"O".8s!\n", cl[p].name);
      if (p ≡ root) break;
      ⟨ Check column p 12 ⟩;
    }
  }
```

26.* A subtle point should be noted: As we uncover column *c*, and run across a row ‘*c x* . . . ’ that should be restored to column *x*, the original successors ‘*x a* . . . ’, ‘*x b* . . . ’, etc., of that row in column *x* may now be cut off. In such a case we can be sure that those successor rows have disappeared from column *x*, and they have *not* been restored.

The reason is that each of those rows must have a primary column; and every primary column was covered before we changed the cutoff. The rows were therefore not restored to column *x* when we uncovered those primary columns.

```
⟨ Subroutines 9 ⟩ +≡
  void uncover(int c)
  {
    register int cc, l, r, rr, nn, uu, dd, t;
    for (o, t = 0, rr = nd[c].up; rr ≥ cutoff; o, rr = nd[rr].up) t++;
    if (t) {      /* t rows that we covered have been cut off */
      oo, nd[c].len −= t;
      if (c ≥ second) lmems += 2;
      oo, nd[c].up = rr, nd[rr].down = c;
    }
    for ( ; rr ≥ last_col; o, rr = nd[rr].up)
      for (nn = rr + 1; nn ≠ rr; ) {
        if (o, nd[nn].color ≥ 0) {
          o, uu = nd[nn].up, dd = nd[nn].down;
          cc = nd[nn].col;
          if (cc ≤ 0) {
            nn = uu;
            continue;
          }
          if (dd ≥ cutoff) o, nd[nn].down = dd = cc;      /* see the “subtle point” above */
          oo, nd[uu].down = nd[dd].up = nn;
          o, t = nd[cc].len + 1;
          o, nd[cc].len = t;
          if (cc ≥ second) lmems += 2;
        }
        nn++;
      }
    o, l = cl[c].prev, r = cl[c].next;
    oo, cl[l].next = cl[r].prev = c;
  }
```

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

```
⟨ Subroutines 9 ⟩ +≡
  void unpurify(int p)
  {
    register int cc, rr, nn, uu, dd, t, x;
    o, cc = nd[p].col, x = nd[p].color;      /* there's no need to clear nd[cc].color */
    for (o, t = 0, rr = nd[cc].up; rr ≥ cutoff; o, rr = nd[rr].up) t++;
    if (t) {      /* t rows that we covered have been cut off */
      oo, nd[cc].len −= t;
      lmems += 2;
      oo, nd[cc].up = rr, nd[rr].down = cc;
    }
    for ( ; rr ≥ last_col; o, rr = nd[rr].up) {
      if (o, nd[rr].color < 0) o, nd[rr].color = x;
      else if (rr ≠ p) {
        for (nn = rr − 1; nn ≠ rr; ) {
          o, uu = nd[nn].up, dd = nd[nn].down;
          o, cc = nd[nn].col;
          if (cc ≤ 0) {
            nn = dd; continue;
          }
          if (nd[nn].color ≥ 0) {
            if (dd ≥ cutoff) o, nd[nn].down = dd = cc;      /* see the "subtle point" above */
            oo, nd[uu].down = nd[dd].up = nn;
            o, t = nd[cc].len + 1;
            o, nd[cc].len = t;
            if (cc ≥ second) lmems += 2;
          }
          nn−−;
        }
      }
    }
  }
```

32.* ⟨Record solution and **goto** *recover* 32*⟩ ≡

```
{
  count++;
  for (k = 0, pp = 0; k ≤ level; k++)
    if (choice[k] > pp) pp = choice[k];
  for (pp++; o, nd[pp].col > 0; pp++) ;      /* move to end of largest chosen row */
  if (pp ≠ cutoff) {
    cutoff = pp;
    if (vbose & show_cutoffs) {
      fprintf(stderr, "␣new␣cutoff␣after␣row␣"O"d:\n", −nd[pp].col);
      prow(nd[pp].up);
    }
    for (k = 0; k ≤ level; k++) {
      o, cc = nd[choice[k]].col;      /* cc will stay covered until we backtrack */
      for (o, t = 0, pp = nd[cc].up; pp ≥ cutoff; o, pp = nd[pp].up) t++;
      if (t) {      /* need to prune unneeded options from column cc */
        oo, nd[pp].down = cc, nd[cc].up = pp;
        oo, nd[cc].len −= t;
      }
    }
  }
  if (spacing ∧ (count mod spacing ≡ 0)) {
    printf(""O"lld:\n", count);
    for (k = 0; k ≤ level; k++) print_row(choice[k], stdout);
    fflush(stdout);
  }
  if (count ≥ maxcount) goto done;
  goto recover;
}
```

This code is used in section 22.

36.* Index.

The following sections were changed by the change file: 2, 3, 7, 11, 26, 30, 32, 36.

⟨ Check column p 12 ⟩ Used in section 11*.
⟨ Check for duplicate column name 14 ⟩ Used in section 13.
⟨ Cover all other columns of *cur_node* 27 ⟩ Used in section 22.
⟨ Create a node for the column named in $buf[p]$ 17 ⟩ Used in section 16.
⟨ Do special things if enough *mems* have accumulated 24 ⟩ Used in section 22.
⟨ Global variables 3*, 7*, 23 ⟩ Used in section 2*.
⟨ Initialize *last_col* to a new column with an empty list 15 ⟩ Used in section 13.
⟨ Input the column names 13 ⟩ Used in section 2*.
⟨ Input the rows 16 ⟩ Used in section 2*.
⟨ Insert node *last_node* into the list for column k 18 ⟩ Used in section 17.
⟨ Print the profile 35 ⟩ Used in section 2*.
⟨ Process the command line 4 ⟩ Used in section 2*.
⟨ Record solution and **goto** *recover* 32* ⟩ Used in section 22.
⟨ Remove *last_node* from its column 19 ⟩ Used in section 16.
⟨ Report the column totals 21 ⟩ Used in section 2*.
⟨ Report the successful completion of the input phase 20 ⟩ Used in section 2*.
⟨ Set *best_col* to the best column for branching 31 ⟩ Used in section 22.
⟨ Solve the problem 22 ⟩ Used in section 2*.
⟨ Subroutines 9, 10, 11*, 25, 26*, 29, 30*, 33, 34 ⟩ Used in section 2*.
⟨ Type definitions 5, 6 ⟩ Used in section 2*.
⟨ Uncover all other columns of *cur_node* 28 ⟩ Used in section 22.

DLX2-CUTOFF