November 24, 2020 at 13:23

1.   Generalized exact cover.   This program implements the algorithm discussed in my paper about "dancing links." I hacked it together from the XCOVER program that I wrote in 1994; I apologize for not having time to apply spit and polish.

Given a matrix whose elements are 0 or 1, the problem is to find all subsets of its rows whose sum is at most 1 in all columns and *exactly* 1 in all "primary" columns. The matrix is specified in the standard input file as follows: Each column has a symbolic name, from one to seven characters long. The first line of input contains the names of all primary columns, followed by '|', followed by the names of all other columns. (If all columns are primary, the '|' may be omitted.) The remaining lines represent the rows, by listing the columns where 1 appears.

The program prints the number of solutions and the total number of link updates. It also prints every nth solution, if the integer command line argument n is given. A second command-line argument causes the full search tree to be printed, and a third argument makes the output even more verbose.

#define *max_level* 150   /* at most this many rows in a solution */
#define *max_degree* 10000   /* at most this many branches per search tree node */
#define *max_cols* 10000   /* at most this many columns */
#define *max_nodes* 1000000   /* at most this many nonzero elements in the matrix */
#define *verbose* *Verbose*   /* kludge because of 64-bit madness in SGB library */

#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <string.h>
  ⟨ Type definitions 3 ⟩
  ⟨ Global variables 2 ⟩
  ⟨ Subroutines 6 ⟩;

  main (*argc*, *argv*)
      int *argc*;
      char *argv [ ];
  {
    ⟨ Local variables 10 ⟩;
    verbose = *argc* − 1;
    if (*verbose*) *sscanf* (*argv* [1], "%d", &*spacing*);
    ⟨ Initialize the data structures 7 ⟩;
    ⟨ Backtrack through all solutions 12 ⟩;
    printf ("Altogether␣%lld␣solutions,␣after␣%.15g␣updates.\n", *count*, *updates*);
    if (*verbose*) ⟨ Print a profile of the search tree 23 ⟩;
    exit (0);
  }

2.   ⟨ Global variables 2 ⟩ ≡
  int *verbose*;   /* > 0 to show solutions, > 1 to show partial ones too */
  long long *count* = 0;   /* number of solutions found so far */
  double *updates*;   /* number of times we deleted a list element */
  int *spacing* = 1;   /* if *verbose*, we output solutions when *count* % *spacing* ≡ 0 */
  double *profile* [*max_level*][*max_degree*];   /* tree nodes of given level and degree */
  double *upd_prof* [*max_level*];   /* updates at a given level */
  int *maxb* = 0;   /* maximum branching factor actually needed */
  int *maxl* = 0;   /* maximum level actually reached */

See also sections 8 and 14.

This code is used in section 1.

3.   Data structures.   Each column of the input matrix is represented by a **column** struct, and each row is represented as a linked list of **node** structs. There's one node for each nonzero entry in the matrix.

More precisely, the nodes are linked circularly within each row, in both directions. The nodes are also linked circularly within each column; the column lists each include a header node, but the row lists do not. Column header nodes are part of a **column** struct, which contains further info about the column.

Each node contains five fields. Four are the pointers of doubly linked lists, already mentioned; the fifth points to the column containing the node.

⟨ Type definitions 3 ⟩ ≡
    typedef struct node_struct {
      struct node_struct *∗left*, *∗right*;   /∗ predecessor and successor in row ∗/
      struct node_struct *∗up*, *∗down*;   /∗ predecessor and successor in column ∗/
      struct col_struct *∗col*;   /∗ the column containing this node ∗/
    } **node**;

See also section 4.

This code is used in section 1.

4.   Each **column** struct contains five fields: The *head* is a node that stands at the head of its list of nodes; the *len* tells the length of that list of nodes, not counting the header; the *name* is a user-specified identifier; *next* and *prev* point to adjacent columns, when this column is part of a doubly linked list.

As backtracking proceeds, nodes will be deleted from column lists when their row has been blocked by other rows in the partial solution. But when backtracking is complete, the data structures will be restored to their original state.

⟨ Type definitions 3 ⟩ +≡
    typedef struct col_struct {
      node *head*;   /∗ the list header ∗/
      int *len*;   /∗ the number of non-header items currently in this column's list ∗/
      char *name*[8];   /∗ symbolic identification of the column, for printing ∗/
      struct col_struct *∗prev*, *∗next*;   /∗ neighbors of this column ∗/
    } **column**;

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

#define *root*   *col_array*[0]   /∗ gateway to the unsettled columns ∗/

6.   A row is identified not by name but by the names of the columns it contains. Here is a routine that prints a row, given a pointer to any of its columns. It also prints the position of the row in its column.

⟨ Subroutines 6 ⟩ ≡
```
void print_row(p)
      node *p;
{ register node *q = p;
  register int k;

  do {
    printf("␣%s", q⃗col⃗name);
    q = q⃗right;
  } while (q ≠ p);
  for (q = p⃗col⃗head.down, k = 1; q ≠ p; k++)
    if (q ≡ &(p⃗col⃗head)) {
      printf("\n"); return;      /* row not in its column! */
    } else  q = q⃗down;
  printf("␣(%d␣of␣%d)\n", k, p⃗col⃗len);
}

void print_state(int lev)
{
  register int l;

  for (l = 0; l ≤ lev; l++)  print_row(choice[l]);
}
```
See also sections 15 and 16.

This code is used in section 1.

7.   Inputting the matrix.   Brute force is the rule in this part of the program.

⟨ Initialize the data structures 7 ⟩ ≡
  ⟨ Read the column names 9 ⟩;
  ⟨ Read the rows 11 ⟩;

This code is used in section 1.

8.   #define buf_size $8 * max\_cols + 3$ /∗ upper bound on input line length ∗/
⟨ Global variables 2 ⟩ +≡
  column $col\_array[max\_cols + 2]$;   /∗ place for column records ∗/
  node $node\_array[max\_nodes]$;   /∗ place for nodes ∗/
  char $buf[buf\_size]$;

9.   #define $panic(m)$
        { $fprintf(stderr, \texttt{"\%s!\textbackslash n\%s"}, m, buf)$; $exit(-1)$; }
⟨ Read the column names 9 ⟩ ≡
  $cur\_col = col\_array + 1$;
  $fgets(buf, buf\_size, stdin)$;
  if $(buf[strlen(buf) - 1] \neq \texttt{'\textbackslash n'})$ $panic(\texttt{"Input␣line␣too␣long"})$;
  for $(p = buf, primary = 1; *p; p{+}{+})$ {
    while $(isspace(*p))$ $p{+}{+}$;
    if $(\neg *p)$ **break**;
    if $(*p \equiv \texttt{'|'})$ {
      $primary = 0$;
      if $(cur\_col \equiv col\_array + 1)$ $panic(\texttt{"No␣primary␣columns"})$;
      $(cur\_col - 1){\rightarrow}next = \&root, root.prev = cur\_col - 1$;
      continue;
    }
    for $(q = p + 1; \neg isspace(*q); q{+}{+})$;
    if $(q > p + 7)$ $panic(\texttt{"Column␣name␣too␣long"})$;
    if $(cur\_col \geq \&col\_array[max\_cols])$ $panic(\texttt{"Too␣many␣columns"})$;
    for $(q = cur\_col{\rightarrow}name; \neg isspace(*p); q{+}{+}, p{+}{+})$ $*q = *p$;
    $cur\_col{\rightarrow}head.up = cur\_col{\rightarrow}head.down = \&cur\_col{\rightarrow}head$;
    $cur\_col{\rightarrow}len = 0$;
    if $(primary)$ $cur\_col{\rightarrow}prev = cur\_col - 1, (cur\_col - 1){\rightarrow}next = cur\_col$;
    else $cur\_col{\rightarrow}prev = cur\_col{\rightarrow}next = cur\_col$;
    $cur\_col{+}{+}$;
  }
  if $(primary)$ {
    if $(cur\_col \equiv col\_array + 1)$ $panic(\texttt{"No␣primary␣columns"})$;
    $(cur\_col - 1){\rightarrow}next = \&root, root.prev = cur\_col - 1$;
  }

This code is used in section 7.

10.   ⟨ Local variables 10 ⟩ ≡
  register column $*cur\_col$;
  register char $*p, *q$;
  register node $*cur\_node$;
  int $primary$;

See also sections 13 and 20.

This code is used in section 1.

11.   ⟨ Read the rows 11 ⟩ ≡
 cur_node = *node_array*;
 while (*fgets*(*buf*, *buf_size*, *stdin*)) {
  register column *∗ccol*;
  register node *∗row_start*;
  if (*buf*[*strlen*(*buf*) − 1] ≠ '\n') *panic*("Input␣line␣too␣long");
  row_start = Λ;
  for (*p* = *buf*; *∗p*; *p*++) {
   while (*isspace*(*∗p*)) *p*++;
   if (¬*∗p*) **break**;
   for (*q* = *p* + 1; ¬*isspace*(*∗q*); *q*++) ;
   if (*q* > *p* + 7) *panic*("Column␣name␣too␣long");
   for (*q* = *cur_col*→*name*; ¬*isspace*(*∗p*); *q*++, *p*++) *∗q* = *∗p*;
   ∗q = '\0';
   for (*ccol* = *col_array*; *strcmp*(*ccol*→*name*, *cur_col*→*name*); *ccol*++) ;
   if (*ccol* ≡ *cur_col*) *panic*("Unknown␣column␣name");
   if (*cur_node* ≡ &*node_array*[*max_nodes*]) *panic*("Too␣many␣nodes");
   if (¬*row_start*) *row_start* = *cur_node*;
   else *cur_node*→*left* = *cur_node* − 1, (*cur_node* − 1)→*right* = *cur_node*;
   cur_node→*col* = *ccol*;
   cur_node→*up* = *ccol*→*head.up*, *ccol*→*head.up*→*down* = *cur_node*;
   ccol→*head.up* = *cur_node*, *cur_node*→*down* = &*ccol*→*head*;
   ccol→*len*++;
   cur_node++;
  }
  if (¬*row_start*) *panic*("Empty␣row");
  row_start→*left* = *cur_node* − 1, (*cur_node* − 1)→*right* = *row_start*;
 }
This code is used in section 7.

12.   Backtracking.   Our strategy for generating all exact covers will be to repeatedly choose always the column that appears to be hardest to cover, namely the column with shortest list, from all columns 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 a column." This means removing it from the list of columns needing to be covered, and "blocking" its rows: removing nodes from other lists whenever they belong to a row of a node in this column's list.

⟨ Backtrack through all solutions 12 ⟩ ≡
    level = 0;
forward: ⟨ Set *best_col* to the best column for branching 19 ⟩;
    cover(*best_col*);
    cur_node = *choice*[*level*] = *best_col*→*head*.*down*;
advance:
    if (*cur_node* ≡ &(*best_col*→*head*)) **goto** *backup*;
    if (*verbose* > 1) {
       printf("L%d:", *level*);
       print_row(*cur_node*);
    }
    ⟨ Cover all other columns of *cur_node* 17 ⟩;
    if (*root*.*next* ≡ &*root*) ⟨ Record solution and **goto** *recover* 21 ⟩;
    level++;
    goto *forward*;
backup: *uncover*(*best_col*);
    if (*level* ≡ 0) **goto** *done*;
    level−−;
    cur_node = *choice*[*level*]; *best_col* = *cur_node*→*col*;
recover: ⟨ Uncover all other columns of *cur_node* 18 ⟩;
    cur_node = *choice*[*level*] = *cur_node*→*down*; **goto** *advance*;
done:
    if (*verbose* > 3) ⟨ Print column lengths, to make sure everything has been restored 22 ⟩;
This code is used in section 1.

13.   ⟨ Local variables 10 ⟩ +≡
   register column *∗best_col*;   /∗ column chosen for branching ∗/
   register node *∗pp*;   /∗ traverses a row ∗/

14.   ⟨ Global variables 2 ⟩ +≡
   int *level*;   /∗ number of choices in current partial solution ∗/
   node *∗choice*[*max_level*];   /∗ the row and column chosen on each level ∗/

15. When a row is blocked, it leaves all lists except the list of the column that is being covered. Thus a node is never removed from a list twice.

⟨ Subroutines 6 ⟩ +≡

```
cover(c)
     column *c;
{ register column *l, *r;
  register node *rr, *nn, *uu, *dd;
  register k = 1;      /* updates */

  l = c→prev;  r = c→next;
  l→next = r;  r→prev = l;
  for (rr = c→head.down;  rr ≠ &(c→head);  rr = rr→down)
    for (nn = rr→right;  nn ≠ rr;  nn = nn→right) {
       uu = nn→up;  dd = nn→down;
       uu→down = dd;  dd→up = uu;
       k++;
       nn→col→len −−;
    }
  updates += k;
  upd_prof [level] += k;
}
```

16. Uncovering is done in precisely the reverse order. The pointers thereby execute an exquisitely choreographed dance which returns them almost magically to their former state.

⟨ Subroutines 6 ⟩ +≡

```
uncover(c)
     column *c;
{ register column *l, *r;
  register node *rr, *nn, *uu, *dd;

  for (rr = c→head.up;  rr ≠ &(c→head);  rr = rr→up)
    for (nn = rr→left;  nn ≠ rr;  nn = nn→left) {
       uu = nn→up;  dd = nn→down;
       uu→down = dd→up = nn;
       nn→col→len ++;
    }
  l = c→prev;  r = c→next;
  l→next = r→prev = c;
}
```

17.   ⟨ Cover all other columns of *cur_node* 17 ⟩ ≡
    for (*pp* = *cur_node*→*right*; *pp* ≠ *cur_node*; *pp* = *pp*→*right*) *cover*(*pp*→*col*);
This code is used in section 12.

18. We included *left* links, thereby making the rows doubly linked, so that columns would be uncovered in the correct LIFO order in this part of the program. (The *uncover* routine itself could have done its job with *right* links only.) (Think about it.)

⟨ Uncover all other columns of *cur_node* 18 ⟩ ≡
    for (*pp* = *cur_node*→*left*; *pp* ≠ *cur_node*; *pp* = *pp*→*left*) *uncover*(*pp*→*col*);
This code is used in section 12.

19.   ⟨Set *best_col* to the best column for branching 19⟩ ≡
 minlen = *max_nodes*;
 if (*verbose* > 2) *printf*("Level␣%d:", *level*);
 for (*cur_col* = *root.next*; *cur_col* ≠ &*root*; *cur_col* = *cur_col*→*next*) {
  if (*verbose* > 2) *printf*("␣%s(%d)", *cur_col*→*name*, *cur_col*→*len*);
  if (*cur_col*→*len* < *minlen*) *best_col* = *cur_col*, *minlen* = *cur_col*→*len*;
 }
 if (*verbose*) {
  if (*level* > *maxl*) {
   if (*level* ≥ *max_level*) *panic*("Too␣many␣levels");
   maxl = *level*;
  }
  if (*minlen* > *maxb*) {
   if (*minlen* ≥ *max_degree*) *panic*("Too␣many␣branches");
   maxb = *minlen*;
  }
  profile[*level*][*minlen*]++;
  if (*verbose* > 2) *printf*("␣branching␣on␣%s(%d)\n", *best_col*→*name*, *minlen*);
 }
This code is used in section 12.

20.   ⟨Local variables 10⟩ +≡
 register int *minlen*;
 register int *j*, *k*, *x*;

21.   ⟨Record solution and **goto** *recover* 21⟩ ≡
 {
  count++;
  if (*verbose*) {
   profile[*level* + 1][0]++;
   if (*count* % *spacing* ≡ 0) {
    printf("%lld:\n", *count*);
    for (*k* = 0; *k* ≤ *level*; *k*++) *print_row*(*choice*[*k*]);
   }
  }
  goto *recover*;
 }
This code is used in section 12.

22.   ⟨Print column lengths, to make sure everything has been restored 22⟩ ≡
 {
  printf("Final␣column␣lengths");
  for (*cur_col* = *root.next*; *cur_col* ≠ &*root*; *cur_col* = *cur_col*→*next*)
   printf("␣%s(%d)", *cur_col*→*name*, *cur_col*→*len*);
  printf("\n");
 }
This code is used in section 12.

23.   ⟨ Print a profile of the search tree 23 ⟩ ≡

  {

    double *tot*, *subtot*;

    tot = 1;   /∗ the root node doesn't show up in the profile ∗/
    for (*level* = 1; *level* ≤ *maxl* + 1; *level*++) {
      subtot = 0;
      for (*k* = 0; *k* ≤ *maxb*; *k*++) {
        printf ("␣%5.6g", *profile*[*level*][*k*]);
        subtot += *profile*[*level*][*k*];
      }
      printf ("␣%5.15g␣nodes,␣%.15g␣updates\n", *subtot*, *upd_prof* [*level* − 1]);
      tot += *subtot*;
    }
    printf ("Total␣%.15g␣nodes.\n", *tot*);

  }

This code is used in section 1.

24.   Index.

⟨ Backtrack through all solutions 12 ⟩   Used in section 1.
⟨ Cover all other columns of *cur_node* 17 ⟩   Used in section 12.
⟨ Global variables 2, 8, 14 ⟩   Used in section 1.
⟨ Initialize the data structures 7 ⟩   Used in section 1.
⟨ Local variables 10, 13, 20 ⟩   Used in section 1.
⟨ Print a profile of the search tree 23 ⟩   Used in section 1.
⟨ Print column lengths, to make sure everything has been restored 22 ⟩   Used in section 12.
⟨ Read the column names 9 ⟩   Used in section 7.
⟨ Read the rows 11 ⟩   Used in section 7.
⟨ Record solution and **goto** *recover* 21 ⟩   Used in section 12.
⟨ Set *best_col* to the best column for branching 19 ⟩   Used in section 12.
⟨ Subroutines 6, 15, 16 ⟩   Used in section 1.
⟨ Type definitions 3, 4 ⟩   Used in section 1.
⟨ Uncover all other columns of *cur_node* 18 ⟩   Used in section 12.

DANCE