May 19, 2018 at 02:30

1. Intro. Given an exact cover problem, presented on *stdin* in the format used by DANCE, we generate clauses for an equivalent satisfiability problem in the format used by my SAT routines.

I hacked this program by starting with DANCE; then I replaced the dancing links algorithm with a new back end. (A lot of the operations performed are therefore pointless leftovers from the earlier routine. Much of the commentary is also superfluous; I did this in a big hurry.)

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, either one or two or three 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.

```
#define max_level 150 /* at most this many rows in a solution */
#define max_degree 1000 /* 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 */
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <string.h>
  ⟨ Type definitions 2 ⟩
  ⟨ Global variables 6 ⟩
  main(argc, argv)
      int argc;
      char *argv[];
  {
    ⟨ Local variables 8 ⟩;
    ⟨ Initialize the data structures 5 ⟩;
    ⟨ Output the clauses 10 ⟩;
  }
```

2. 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.

Well, actually I've now included a sixth field. It specifies the row number.

⟨ Type definitions 2 ⟩ ≡

```
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 */
  int num;      /* the row in which this node appears */
} node;
```

See also section 3.

This code is used in section 1.

3. 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 one-, two-, or three-letter identifier; *next* and *prev* point to adjacent columns, when this column is part of a doubly linked list.

⟨ Type definitions 2 ⟩ +≡

```
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;
```

4. 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 */
```

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

```
⟨ Initialize the data structures 5 ⟩ ≡
  ⟨ Read the column names 7 ⟩;
  ⟨ Read the rows 9 ⟩;
```

This code is used in section 1.

6. **#define** *buf_size* $4 * max\_cols + 3$ /* upper bound on input line length */

```
⟨ Global variables 6 ⟩ ≡
  column col_array[max_cols + 2];  /* place for column records */
  node node_array[max_nodes];  /* place for nodes */
  char buf[buf_size];
  node *row[max_nodes];  /* the first node in each row */
  int rowptr;  /* this many rows have been seen */
```

This code is used in section 1.

7. **#define** *panic*(*m*)
{ *fprintf*(*stderr*, "%s!\n%s", *m*, *buf*); *exit*(−1); }

```
⟨ Read the column names 7 ⟩ ≡
  cur_col = col_array + 1;
  fgets(buf, buf_size, stdin);
  if (buf[strlen(buf) − 1] ≠ '\n') panic("Input line too long");
  for (p = buf, primary = 1; *p; p++) {
    while (isspace(*p)) p++;
    if (¬*p) break;
    if (*p ≡ '|') {
      primary = 0;
      if (cur_col ≡ col_array + 1) panic("No primary columns");
      (cur_col − 1)→next = &root, root.prev = cur_col − 1;
      continue;
    }
    for (q = p + 1; ¬isspace(*q); q++) ;
    if (q > p + 7) panic("Column name too long");
    if (cur_col ≥ &col_array[max_cols]) panic("Too many columns");
    for (q = cur_col→name; ¬isspace(*p); q++, p++) *q = *p;
    cur_col→head.up = cur_col→head.down = &cur_col→head;
    cur_col→head.num = −1;
    cur_col→len = 0;
    if (primary) cur_col→prev = cur_col − 1, (cur_col − 1)→next = cur_col;
    else cur_col→prev = cur_col→next = cur_col;
    cur_col++;
  }
  if (primary) {
    if (cur_col ≡ col_array + 1) panic("No primary columns");
    (cur_col − 1)→next = &root, root.prev = cur_col − 1;
  }
```

This code is used in section 5.

8. ⟨ Local variables 8 ⟩ ≡

```
register column *cur_col;
register char *p, *q;
register node *cur_node;
int primary;
int j, k;
```

This code is used in section 1.

9. ⟨ Read the rows 9 ⟩ ≡

```
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→num = rowptr;
    cur_node++;
  }
  if (¬row_start) panic("Empty␣row");
  row[rowptr++] = row_start;
  row_start→left = cur_node − 1, (cur_node − 1)→right = row_start;
}
```

This code is used in section 5.

10. Clausing. There's one variable for each row; its meaning is "this row is in the cover." There are two kinds of clauses: For each primary column, we must select one of its rows. For each pair of intersecting rows, we must not select them both.

```
⟨ Output the clauses 10 ⟩ ≡
  ⟨ Output the column clauses 11 ⟩;
  ⟨ Output the intersection clauses 12 ⟩;
```

This code is used in section 1.

11.
```
⟨ Output the column clauses 11 ⟩ ≡
  for (cur_col = root.next; cur_col ≠ &root; cur_col = cur_col→next) {
    for (cur_node = cur_col→head.down; cur_node ≠ &cur_col→head; cur_node = cur_node→down)
      printf("␣%d", cur_node→num + 1);
    printf("\n");
  }
```

This code is used in section 10.

12.
```
⟨ Output the intersection clauses 12 ⟩ ≡
  for (j = 0; j < rowptr; j++) {
    for (cur_node = row[j]→right; cur_node ≠ row[j]; cur_node = cur_node→right)
      cur_node→col→head.num = j;
    cur_node→col→head.num = j;
    for (k = j + 1; k < rowptr; k++) {
      for (cur_node = row[k]→right; cur_node ≠ row[k]; cur_node = cur_node→right)
        if (cur_node→col→head.num ≡ j) goto clash;
      if (cur_node→col→head.num ≡ j) goto clash;
      continue;
    clash: printf("~%d␣~%d\n", j + 1, k + 1);
    }
  }
```

This code is used in section 10.

13. Index.

⟨ Global variables 6 ⟩ Used in section 1.
⟨ Initialize the data structures 5 ⟩ Used in section 1.
⟨ Local variables 8 ⟩ Used in section 1.
⟨ Output the clauses 10 ⟩ Used in section 1.
⟨ Output the column clauses 11 ⟩ Used in section 10.
⟨ Output the intersection clauses 12 ⟩ Used in section 10.
⟨ Read the column names 7 ⟩ Used in section 5.
⟨ Read the rows 9 ⟩ Used in section 5.
⟨ Type definitions 2, 3 ⟩ Used in section 1.

SAT-DANCE