1. Generalized exact cover. This program implements an extension of 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 in that paper was 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, up 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.

Here I extend the idea so that nonprimary columns can have a different sort of restriction: If a row specifies a "color" in a nonprimary column, it rules out rows of all other colors in that column, but any number of rows with the same color are allowed. (The previous situation was the special case in which all rows had a different color.) If `xx` is a column name, a specification like `xx:a` as part of a row stands for color `a` in column `xx`. Each color is specified by a single character.

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 1000    /* at most this many rows in a solution */
#define max_degree 6000    /* at most this many branches per search tree node */
#define max_cols 6000    /* at most this many columns */
#define max_nodes 1000000    /* at most this many nonzero elements in the matrix */
#define verbose Verbose

#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,", count);
    printf("␣after␣%lld␣updates", updates);
    printf("␣and␣%lld␣cleansings.\n", purifs);
    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 */
unsigned long long count;   /* number of solutions found so far */
unsigned long long updates;   /* number of times we deleted a list element */
unsigned long long purifs;   /* number of times we purified a list element */
int spacing = 1;   /* if verbose, we output solutions when count % spacing ≡ 0 */
unsigned long long profile[max_level][max_degree];   /* tree nodes of given level and degree */
unsigned long long upd_prof[max_level];   /* updates at a given level */
unsigned long long pur_prof[max_level];   /* purifications 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 */
  int color;  /* color, if specified */
} 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 one-, two-, or ... or seven-letter 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 nodes. It also prints the position of the given node in its column.

```
⟨ Subroutines 6 ⟩ ≡
  print_row(p)
      node *p;
  { register node *q = p;
    register int k;
    do {
      printf("␣%s", q→col→name);
      if (q→color) ⟨ Print the color of node q 27 ⟩;
      q = q→right;
    } while (q ≠ p);
    for (q = p→col→head.down, k = 1; q ≠ p; k++)
      if (q ≡ &(p→col→head)) {
        printf("\n"); return 0;    /* row not in its column! */
      } else q = q→down;
    printf("␣(%d␣of␣%d)\n", k, p→col→len);
  }
```

See also sections 15, 16*, 25, 26*, and 28.

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 */
  node *cutoff = &node_array[max_nodes];   /* upper bound on active rows */
  char buf[buf_size];
  column *first_nonprim_col;   /* the first nonprimary column, if any */
  column *last_nonprim_col;   /* the first unused column */
```

9.

```
#define panic(m)
        { fprintf(stderr, "%s!\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] ≠ '\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;
      first_nonprim_col = cur_col;
      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→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;
    first_nonprim_col = cur_col;
  }
  last_nonprim_col = cur_col;
```

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, *x;
  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 ≠ ':'; q++) ;
    if (q > p + 7) panic("Column␣name␣too␣long");
    for (q = cur_col→name; ¬isspace(*p) ∧ *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;
    for (x = row_start; x ≠ cur_node; x++)
      if (x→col ≡ ccol) panic("A␣row␣can't␣use␣a␣column␣twice");
    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++;
    if (*p ≡ ':') ⟨Read a color restriction 24⟩;
    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 int 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 ≥ cutoff; rr = rr→up) {
    if (rr ≡ &(c→head)) break;
    c→len−−;
  }
  c→head.up = rr, rr→down = &(c→head);
  for ( ; rr ≠ &(c→head); rr = rr→up)
    for (nn = rr→left; nn ≠ rr; nn = nn→left) {
      uu = nn→up; dd = nn→down;
      if (dd ≥ cutoff) nn→down = dd = &(nn→col→head);
      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)
  if (¬pp→color) cover(pp→col);
  else if (pp→color > 0) purify(pp);
```

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)
  if (¬pp→color) uncover(pp→col);
  else if (pp→color > 0) unpurify(pp);
```
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;
long long xx, tt;
```

21*. ⟨ Record solution and **goto** *recover* 21* ⟩ ≡
```
{
  count++;
  ⟨ Recompute the row-cutoff threshold 29* ⟩;
  ⟨ Prune the columns currently in use 30* ⟩;
  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 ⟩ ≡

```
{
  xx = 1;    /* the root node doesn't show up in the profile */
  for (level = 1; level ≤ maxl + 1; level++) {
    tt = 0;
    for (k = 0; k ≤ maxb; k++) {
      printf("%10lld", profile[level][k]);
      tt += profile[level][k];
    }
    printf("%16lld␣nodes,␣%llu␣updates,␣%llu␣cleansings\n", tt, upd_prof[level − 1],
        pur_prof[level − 1]);
    xx += tt;
  }
  printf("Total␣%lld␣nodes.\n", xx);
}
```

This code is used in section 1.

24. Color barriers. Finally, here's the new material related to coloring.

⟨ Read a color restriction 24 ⟩ ≡

```
  {
    if (ccol < first_nonprim_col) panic("Color␣isn't␣allowed␣in␣a␣primary␣column");
    if (isspace(*(p + 1)) ∨ ¬isspace(*(p + 2))) panic("Color␣should␣be␣a␣single␣character");
    cur_node→color = *(p + 1);
    p += 2;
  }
```

This code is used in section 11.

25. When we choose a row that specifies colors in one or more columns, we "purify" those columns by removing all incompatible rows. All rows that want the same color in a purified column will now be given the color code -1 so that we need not purify the column again.

⟨ Subroutines 6 ⟩ +≡

```
  purify(p)
      node *p;
  { register column *c = p→col;
    register int x = p→color;
    register node *rr, *nn, *uu, *dd;
    register int k = 0, kk = 1;      /* updates */
    c→head.color = x;      /* this is used only to help print_row */
    for (rr = c→head.down; rr ≠ &(c→head); rr = rr→down)
      if (rr→color ≠ x) {
        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−−;
        }
      } else if (rr ≠ p) kk++, rr→color = −1;
    updates += k, purifs += kk;
    upd_prof[level] += k, pur_prof[level] += kk;
  }
```

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

⟨ Subroutines 6 ⟩ +≡

```
unpurify(p)
    node *p;
{ register column *c = p→col;
  register int x = p→color;
  register node *rr, *nn, *uu, *dd;
  for (rr = c→head.up; rr ≥ cutoff; rr = rr→up)
    if (rr ≡ &(c→head)) break;
  c→head.up = rr, rr→down = &(c→head);
  for ( ; rr ≠ &(c→head); rr = rr→up)
    if (rr→color < 0) rr→color = x;
    else if (rr ≠ p) {
      for (nn = rr→left; nn ≠ rr; nn = nn→left) {
        uu = nn→up; dd = nn→down;
        if (dd ≥ cutoff) nn→down = dd = &(nn→col→head);
        uu→down = dd→up = nn;
        nn→col→len++;
      }
    }
  c→head.color = 0;
}
```

27. ⟨ Print the color of node *q* 27 ⟩ ≡

```
printf(":%c", q→color > 0 ? q→color : q→col→head.color);
```

This code is used in section 6.

28. Help for debugging. Here's a subroutine for when I'm doing a long run and want to check the current progress.

```
⟨ Subroutines 6 ⟩ +≡
  void show_state()
  {
    register int k;
    printf("Current␣state␣(level␣%d):\n", level);
    for (k = 0; k < level; k++) print_row(choice[k]);
    printf("Max␣level␣so␣far:␣%d\n", maxl);
    printf("Max␣branching␣so␣far:␣%d\n", maxb);
    printf("Solutions␣so␣far:␣%lld\n", count);
  }
```

29*. Added material. Here we use the fact that the nodes have been allocated sequentially.

⟨ Recompute the row-cutoff threshold 29* ⟩ ≡

```
for (k = 0, cutoff = Λ; k ≤ level; k++)
  if (choice[k] ≥ cutoff) {
    for (pp = choice[k]; pp→right > pp; pp = pp→right) ;
    cutoff = pp + 1;
  }
```

This code is used in section 21*.

30*. ⟨ Prune the columns currently in use 30* ⟩ ≡

```
for (k = 0; k ≤ level; k++) {
  cur_col = choice[k]→col;
  for (pp = cur_col→head.up, j = 0; pp ≥ cutoff; pp = pp→up) j++;
  if (j) {
    pp→down = &(cur_col→head);
    cur_col→head.up = pp;
    cur_col→len −= j;
  }
}
```

This code is used in section 21*.

31*. Index.

The following sections were changed by the change file: 8, 16, 21, 26, 29, 30, 31.

GDANCE-CUTOFF