1. Antisliding blocks. This program illustrates techniques of finding all nonequivalent solutions to an exact cover problem. I wrote it after returning from Japan in November, 1996, because Nob was particularly interested in the answers. (Two years ago I had written a similar program, which however did not remove inequivalent solutions; in 1994 I removed them by hand after generating all possible solutions.)

The general question is to pack $2 \times 2 \times 1$ blocks into an $l \times m \times n$ array in such a way that the blocks cannot slide. This means that there should be at least one occupied cell touching each of the six faces of the block; cells outside the array are always considered to be occupied. For example, one such solution when $l = m = n = 3$ is

```
1 1 .    5 4 4    5 4 4
1 1 2    5 . 2    5 6 6
3 3 2    3 3 2    . 6 6.
```

But

```
1 1 2    4 4 2    4 4 .
1 1 2    . . 2    . 5 5
3 3 .    3 3 .    . 5 5
```

is not a solution, because blocks `2`, `3`, and `5` can slide.

Two solutions are considered to be the same if they are isomorphic—that is, if there's a symmetry that takes one into the other. In this sense the solution

```
1 1 .    1 1 4    6 6 4
2 3 3    2 . 4    6 6 4
2 3 3    2 5 5    . 5 5
```

is no different from the first solution given above. Up to 48 symmetries are possible, obtained by permuting and complementing the coordinates in three-dimensional space. It turns out that the $3 \times 3 \times 3$ case has only one solution, besides the trivial case in which no blocks at all are present.

Before writing this program I tried to find highly symmetric solutions to the $4 \times 4 \times 4$ problem without using a computer. I found a beautiful 12-block solution

```
. 1 1 .    5 5 6 6    5 5 6 6    . B B .
2 1 1 3    2 . . 3    9 . . A    9 B B A
2 4 4 3    2 . . 3    9 . . A    9 C C A
. 4 4 .    7 7 8 8    7 7 8 8    . C C .
```

which has 24 symmetries and leaves the center cells and corner cells empty. But I saw no easy way to prove that an antisliding arrangement with fewer than 12 blocks is possible. This experience whetted my curiosity and got me "hooked" on the problem, so I couldn't resist writing this program even though I have many other urgent things to do. I'm considering it the final phase of my exciting visit to Japan. (I apologize for not having time to refine it further.)

Note: The program assumes that $l = m$ if any two of the dimensions are equal. Then the number of symmetries is 8 if $l \neq m$, or 16 if $l = m \neq n$, or 48 if $l = m = n$.

```
#define ll 4    /* the first dimension */
#define mm 4    /* the second dimension */
#define nn 4    /* the third */
#define ss 48    /* the number of symmetries */
#include <stdio.h>
  ⟨ Type definitions 3 ⟩
  ⟨ Global variables 2 ⟩
  ⟨ Subroutines 19 ⟩
  main(argc, argv)
```

```
    int argc;
    char *argv[];
{
  ⟨ Local variables 24 ⟩;
  if (argc > 1) {
    verbose = argc − 1;      /* set verbose to the number of command-line arguments */
    sscanf(argv[1], "%d", &spacing);
  }
  ⟨ Set up data structures for antisliding blocks 8 ⟩;
  ⟨ Backtrack through all solutions 25 ⟩;
  ⟨ Make redundancy checks to see if the backtracking was consistent 47 ⟩;
  printf("Altogether␣%d␣solutions.\n", count);
  if (verbose) ⟨ Print a profile of the search tree 45 ⟩;
}
```

2. ⟨ Global variables 2 ⟩ ≡

```
int verbose = 0;      /* > 0 to show solutions, > 1 to show partial ones too */
int count = 0;      /* number of antisliding solutions found so far */
int spacing = 1;      /* if verbose, we output solutions when count % spacing ≡ 0 */
int profile[ll * mm * nn + 1], prof_syms[ll * mm * nn + 1], prof_cons[ll * mm * nn + 1],
    prof_frcs[ll * mm * nn + 1];      /* statistics */
```

See also sections 5, 7, and 23.

This code is used in section 1.

3. Data structures. An exact cover problem is defined by a matrix M of 0s and 1s. The goal is to find a set of rows containing exactly one 1 in each column.

In our case the rows stand for possible placements of blocks; the columns stand for cells of the $l \times m \times n$ array. There are $l(m-1)(n-1)+(l-1)m(n-1)+(l-1)(m-1)n$ rows for placements of $2\times 2\times 1$ blocks and an additional lmn rows for $1\times 1\times 1$ blocks that correspond to unoccupied cells.

The heart of this program is its data structure for the matrix M. There is one node for each 1 in M, and the 1s of each row are cyclically linked via *left* and *right* fields. Each node also contains an array of pointers *sym*[0], *sym*[1], ..., which point to the nodes that are equivalent under each symmetry of the problem. Furthermore, the nodes for 1s in each column are doubly linked together by *up* and *down* fields.

Although the pointers are called *left*, *right*, *up*, and *down*, the row lists and column lists need not actually be linked together in any particular order. The row lists remain unchanged, but the column lists will change dynamically because we will implicitly remove rows from M that contain 1s in columns that are already covered as we are constructing a solution.

```
⟨ 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 node_struct *sym[ss];           /* symmetric equivalents */
    struct row_struct *row;                /* the row containing this node */
    struct col_struct *col;                /* the column containing this node */
  } node;
```

See also sections 4 and 6.

This code is used in section 1.

4. Each column corresponds to a cell of the array. Special information for each cell is stored in an appropriate record, which points to the $1\times 1\times 1$ block node for that cell (also called the cell head). We maintain a doubly linked list of the cells that still need to be covered, using *next* and *prev* fields; also a count of the number of ways that remain to cover a given cell. A few other items are maintained to facilitate the bookkeeping.

```
⟨ Type definitions 3 ⟩ +≡
  typedef struct col_struct {
    node head;      /* the empty option for this cell */
    int len;        /* the number of options for covering it */
    int init_len;   /* initial value of len, for redundancy check */
    struct col_struct *prev, *next;      /* still-to-be-covered neighbors */
    node *filled;   /* node by which this column was filled */
    int empty;      /* is this cell covered by the empty (1 × 1 × 1) option? */
    int nonempty;   /* is this cell known to be nonempty? */
    char name[4];   /* coordinates of this cell, as a string for printing */
    struct col_struct *invsym[ss];       /* reverse pointers to sym in head */
  } cell;
```

5. One **cell** 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.

```
⟨ Global variables 2 ⟩ +≡
  cell root;      /* gateway to the unsettled columns */
```

6. The rows of M also have special data structures: We need to know which sets of two or four cells are neighbors of the faces of a block. These are listed in the option records, followed by null pointers.

```
⟨ Type definitions 3 ⟩ +≡
  typedef struct row_struct {
    cell *neighbor[22];      /* sets of cells that shouldn't all be empty */
    int neighbor_ptr;        /* size of the neighbor info */
  } option;
```

7. Initialization. Like most table-driven programs, this one needs to construct its tables, using a rather long and boring routine. In compensation, we will be able to avoid tedious details in the rest of the code.

```
⟨ Global variables 2 ⟩ +≡
  cell cells[ll][mm][nn];      /* columns of the matrix */
  option opt[ll][mm][nn], optx[ll][mm − 1][nn − 1], opty[ll − 1][mm][nn − 1], optz[ll − 1][mm − 1][nn];
    /* rows */
  node blockx[ll][mm − 1][nn − 1][4], blocky[ll − 1][mm][nn − 1][4], blockz[ll − 1][mm − 1][nn][4];
    /* nodes */
```

8.
```
⟨ Set up data structures for antisliding blocks 8 ⟩ ≡
  ⟨ Set up the cells 9 ⟩;
  ⟨ Set up the options 11 ⟩;
  ⟨ Set up the nodes 15 ⟩;
```
This code is used in section 1.

9.
```
⟨ Set up the cells 9 ⟩ ≡
  q = &root;
  for (i = 0; i < ll; i++)
    for (j = 0; j < mm; j++)
      for (k = 0; k < nn; k++) {
        c = &cells[i][j][k];
        q→next = c;
        c→prev = q;
        q = c;
        p = &(c→head);
        p→left = p→right = p→up = p→down = p;
        p→row = &opt[i][j][k];
        p→col = c;
        ⟨ Fill in the symmetry pointers of c 10 ⟩;
        c→name[0] = i + '0';
        c→name[1] = j + '0';
        c→name[2] = k + '0';
        c→len = 1;
      }
  q→next = &root;
  root.prev = q;
```
This code is used in section 8.

10. ⟨ Fill in the symmetry pointers of c 10 ⟩ ≡

```
for (s = 0; s < ss; s++) {
  switch (s ≫ 3) {
  case 0: ii = i;
    jj = j;
    kk = k;
    break;
  case 1: ii = j;
    jj = i;
    kk = k;
    break;
  case 2: ii = k;
    jj = j;
    kk = i;
    break;
  case 3: ii = i;
    jj = k;
    kk = j;
    break;
  case 4: ii = j;
    jj = k;
    kk = i;
    break;
  case 5: ii = k;
    jj = i;
    kk = j;
    break;
  }
  if (s & 4) ii = ll − 1 − ii;
  if (s & 2) jj = mm − 1 − jj;
  if (s & 1) kk = nn − 1 − kk;
  p→sym[s] = &(cells[ii][jj][kk].head);
  cells[ii][jj][kk].invsym[s] = c;
}
```

This code is used in section 9.

11. ⟨ Set up the options 11 ⟩ ≡

```
⟨ Set up the optx options 12 ⟩;
⟨ Set up the opty options 13 ⟩;
⟨ Set up the optz options 14 ⟩;
```

This code is used in section 8.

12. `#define ox(j1,k1,j2,k2)`
```
{
  optx[i][j][k].neighbor[kk++] = &cells[i][j1][k1];
  optx[i][j][k].neighbor[kk++] = &cells[i][j2][k2];
  optx[i][j][k].neighbor[kk++] = Λ;
}
```
`#define oxx(i1)`
```
{
  optx[i][j][k].neighbor[kk++] = &cells[i1][j][k];
  optx[i][j][k].neighbor[kk++] = &cells[i1][j][k+1];
  optx[i][j][k].neighbor[kk++] = &cells[i1][j+1][k];
  optx[i][j][k].neighbor[kk++] = &cells[i1][j+1][k+1];
  optx[i][j][k].neighbor[kk++] = Λ;
}
```
⟨ Set up the *optx* options 12 ⟩ ≡
```
for (i = 0; i < ll; i++)
  for (j = 0; j < mm − 1; j++)
    for (k = 0; k < nn − 1; k++) {
      kk = 0;
      if (j) ox(j − 1, k, j − 1, k + 1);
      if (j < mm − 2) ox(j + 2, k, j + 2, k + 1);
      if (k) ox(j, k − 1, j + 1, k − 1);
      if (k < nn − 2) ox(j, k + 2, j + 1, k + 2);
      if (i) oxx(i − 1);
      if (i < ll − 1) oxx(i + 1);
      optx[i][j][k].neighbor_ptr = kk;
    }
```
This code is used in section 11.

13. #**define** *oy*(*i1*, *k1*, *i2*, *k2*)
```
{
  opty[i][j][k].neighbor[kk++] = &cells[i1][j][k1];
  opty[i][j][k].neighbor[kk++] = &cells[i2][j][k2];
  opty[i][j][k].neighbor[kk++] = Λ;
}
```
#**define** *oyy*(*j1*)
```
{
  opty[i][j][k].neighbor[kk++] = &cells[i][j1][k];
  opty[i][j][k].neighbor[kk++] = &cells[i][j1][k + 1];
  opty[i][j][k].neighbor[kk++] = &cells[i + 1][j1][k];
  opty[i][j][k].neighbor[kk++] = &cells[i + 1][j1][k + 1];
  opty[i][j][k].neighbor[kk++] = Λ;
}
```
⟨ Set up the *opty* options 13 ⟩ ≡
```
for (i = 0; i < ll − 1; i++)
  for (j = 0; j < mm; j++)
    for (k = 0; k < nn − 1; k++) {
      kk = 0;
      if (i) oy(i − 1, k, i − 1, k + 1);
      if (i < ll − 2) oy(i + 2, k, i + 2, k + 1);
      if (k) oy(i, k − 1, i + 1, k − 1);
      if (k < nn − 2) oy(i, k + 2, i + 1, k + 2);
      if (j) oyy(j − 1);
      if (j < mm − 1) oyy(j + 1);
      opty[i][j][k].neighbor_ptr = kk;
    }
```
This code is used in section 11.

14.
```
#define oz(i1,j1,i2,j2)
        {
          optz[i][j][k].neighbor[kk++] = &cells[i1][j1][k];
          optz[i][j][k].neighbor[kk++] = &cells[i2][j2][k];
          optz[i][j][k].neighbor[kk++] = Λ;
        }
#define ozz(k1)
        {
          optz[i][j][k].neighbor[kk++] = &cells[i][j][k1];
          optz[i][j][k].neighbor[kk++] = &cells[i][j+1][k1];
          optz[i][j][k].neighbor[kk++] = &cells[i+1][j][k1];
          optz[i][j][k].neighbor[kk++] = &cells[i+1][j+1][k1];
          optz[i][j][k].neighbor[kk++] = Λ;
        }
⟨ Set up the optz options 14 ⟩ ≡
  for (i = 0; i < ll − 1; i++)
    for (j = 0; j < mm − 1; j++)
      for (k = 0; k < nn; k++) {
        kk = 0;
        if (i) oz(i − 1, j, i − 1, j + 1);
        if (i < ll − 2) oz(i + 2, j, i + 2, j + 1);
        if (j) oz(i, j − 1, i + 1, j − 1);
        if (j < mm − 2) oz(i, j + 2, i + 1, j + 2);
        if (k) ozz(k − 1);
        if (k < nn − 1) ozz(k + 1);
        optz[i][j][k].neighbor_ptr = kk;
      }
```
This code is used in section 11.

15.
```
⟨ Set up the nodes 15 ⟩ ≡
  ⟨ Set up the blockx nodes 16 ⟩;
  ⟨ Set up the blocky nodes 17 ⟩;
  ⟨ Set up the blockz nodes 18 ⟩;
```
This code is used in section 8.

16. ⟨ Set up the *blockx* nodes 16 ⟩ ≡

```
for (i = 0; i < ll; i++)
  for (j = 0; j < mm − 1; j++)
    for (k = 0; k < nn − 1; k++) {
      for (t = 0; t < 4; t++) {
        p = &blockx[i][j][k][t];
        p→right = &blockx[i][j][k][(t + 1) & 3];
        p→left = &blockx[i][j][k][(t + 3) & 3];
        c = &cells[i][j + ((t & 2) ≫ 1)][k + (t & 1)];
        pp = c→head.up;
        pp→down = c→head.up = p;
        p→up = pp;
        p→down = &(c→head);
        p→row = &optx[i][j][k];
        p→col = c;
        c→len++;
      }
      make_syms(blockx[i][j][k]);
    }
```

This code is used in section 15.

17. ⟨ Set up the *blocky* nodes 17 ⟩ ≡

```
for (i = 0; i < ll − 1; i++)
  for (j = 0; j < mm; j++)
    for (k = 0; k < nn − 1; k++) {
      for (t = 0; t < 4; t++) {
        p = &blocky[i][j][k][t];
        p→right = &blocky[i][j][k][(t + 1) & 3];
        p→left = &blocky[i][j][k][(t + 3) & 3];
        c = &cells[i + ((t & 2) ≫ 1)][j][k + (t & 1)];
        pp = c→head.up;
        pp→down = c→head.up = p;
        p→up = pp;
        p→down = &(c→head);
        p→row = &opty[i][j][k];
        p→col = c;
        c→len++;
      }
      make_syms(blocky[i][j][k]);
    }
```

This code is used in section 15.

18. ⟨ Set up the *blockz* nodes 18 ⟩ ≡

```
for (i = 0; i < ll − 1; i++)
  for (j = 0; j < mm − 1; j++)
    for (k = 0; k < nn; k++) {
      for (t = 0; t < 4; t++) {
        p = &blockz[i][j][k][t];
        p→right = &blockz[i][j][k][(t + 1) & 3];
        p→left = &blockz[i][j][k][(t + 3) & 3];
        c = &cells[i + ((t & 2) ≫ 1)][j + (t & 1)][k];
        pp = c→head.up;
        pp→down = c→head.up = p;
        p→up = pp;
        p→down = &(c→head);
        p→row = &optz[i][j][k];
        p→col = c;
        c→len++;
      }
      make_syms(blockz[i][j][k]);
    }
```

This code is used in section 15.

19. ⟨ Subroutines 19 ⟩ ≡

```
make_syms(pp)
    node pp[];
{
  register char *q;
  register int s, t, imax, imin, jmax, jmin, kmax, kmin, i, j, k;
  for (s = 0; s < ss; s++) {
    imax = jmax = kmax = −1;
    imin = jmin = kmin = 1000;
    for (t = 0; t < 4; t++) {
      q = pp[t].col→head.sym[s]→col→name;
      i = q[0] − '0';
      j = q[1] − '0';
      k = q[2] − '0';
      if (i < imin) imin = i;
      if (i > imax) imax = i;
      if (j < jmin) jmin = j;
      if (j > jmax) jmax = j;
      if (k < kmin) kmin = k;
      if (k > kmax) kmax = k;
    }
    if (imin ≡ imax) ⟨ Map to blockx nodes 20 ⟩
    else if (jmin ≡ jmax) ⟨ Map to blocky nodes 21 ⟩
    else ⟨ Map to blockz nodes 22 ⟩;
  }
}
```

See also sections 27, 28, and 43.

This code is used in section 1.

20. ⟨ Map to *blockx* nodes 20 ⟩ ≡

```
for (t = 0; t < 4; t++) {
  q = pp[t].col→head.sym[s]→col→name;
  i = q[0] - '0';
  j = q[1] - '0';
  k = q[2] - '0';
  pp[t].sym[s] = &blockx[i][jmin][kmin][(j - jmin) * 2 + k - kmin];
}
```

This code is used in section 19.

21. ⟨ Map to *blocky* nodes 21 ⟩ ≡

```
for (t = 0; t < 4; t++) {
  q = pp[t].col→head.sym[s]→col→name;
  i = q[0] - '0';
  j = q[1] - '0';
  k = q[2] - '0';
  pp[t].sym[s] = &blocky[imin][j][kmin][(i - imin) * 2 + k - kmin];
}
```

This code is used in section 19.

22. ⟨ Map to *blockz* nodes 22 ⟩ ≡

```
for (t = 0; t < 4; t++) {
  q = pp[t].col→head.sym[s]→col→name;
  i = q[0] - '0';
  j = q[1] - '0';
  k = q[2] - '0';
  pp[t].sym[s] = &blockz[imin][jmin][k][(i - imin) * 2 + j - jmin];
}
```

This code is used in section 19.

23. Backtracking and isomorph rejection. The basic operation of this program is a backtrack search, which repeatedly finds an uncovered cell and tries to cover it in all possible ways. We save lots of work if we always choose a cell that has the fewest remaining options. The program considers each of those options in turn; a given option covers certain cells and removes all other options that cover those cells. We must backtrack if we run out of options for any uncovered cell.

The solutions are sequences $a_1\, a_2 \ldots a_l$, where each a_k is a node. Node a_k belongs to column c_k, the cell chosen for covering at level k, and to row r_k, the option chosen for covering that cell.

With 48 symmetries we can reduce the number of cases considered by a factor of up to 48 if we spend a bit more time on each case, by being careful to weed out solutions that are isomorphic to others that have been or will be found. If $a_1\, a_2 \ldots a_l$ is a solution that defines a covering C, and if σ is a symmetry of the problem, the nodes $\sigma a_1, \sigma a_2, \ldots, \sigma a_l$ define a covering σC that is isomorphic to C. For each k in the range $1 \le k \le l$, let a'_k be the node for which $\sigma a'_k$ is the node that covers σc_k in σC. We will consider only solutions such that $a'_1\, a'_2 \ldots a'_l$ is lexicographically less than or equal to $a_1\, a_2 \ldots a_l$; this will guarantee that we obtain exactly one solution from every equivalence class of isomorphic coverings. (Notice that the number of symmetries of a given solution $a_1\, a_2 \ldots a_l$ is the number of σ for which we have $a'_1\, a'_2 \ldots a'_l = a_1\, a_2 \ldots a_l$.)

If $a_l\, a_2 \ldots a_l$ is a partial solution and σ is any symmetry, we can compute $a'_1\, a'_2 \ldots a'_j$ where j is the smallest subscript such that σc_{j+1} has not yet been covered. The partial solution $a_l\, a_2 \ldots a_l$ can be rejected if $a'_1\, a'_2 \ldots a'_j$ is lexicographically less than $a_1\, a_2 \ldots a_j$. The symmetry σ need not be monitored in extensions of $a_l\, a_2 \ldots a_l$ to higher levels if $a'_1\, a'_2 \ldots a'_j$ is lexicographically greater than $a_1\, a_2 \ldots a_j$. We keep a list at level l of all (σ, j) for which $a'_1\, a'_2 \ldots a'_j = a_1\, a_2 \ldots a_j$, where j is defined as above; this is called the *symcheck list*. The symcheck list is the key to isomorph rejection.

We also maintain a list of constraints: Sets of uncovered cells that must not all be empty; these constraints ensure an antisliding solution.

```
⟨ Global variables 2 ⟩ +≡
  int symcheck_sig[(ll * mm * nn + 1) * (ss - 1)], symcheck_j[(ll * mm * nn + 1) * (ss - 1)];
    /* symcheck list elements */
  int symcheck_ptr[ll * mm * nn + 2];      /* beginning of symcheck list on each level */
  cell *constraint[ll * mm * nn * 22];     /* sets of cells that shouldn't all be empty */
  int constraint_ptr[ll * mm * nn + 2];    /* beginning of constraint list on each level */
  cell *force[ll * mm * nn];      /* list of cells forced to be nonempty */
  int force_ptr[ll * mm * nn + 1];      /* beginning of force records on each level */
  cell *best_cell[ll * mm * nn + 1];      /* cell chosen for covering on each level */
  node *move[ll * mm * nn + 1];      /* the nodes a_k on each level */
```

24. ⟨ Local variables 24 ⟩ ≡

```
  register int i, j, k, s;      /* miscellaneous indices */
  register int l;      /* the current level */
  register cell *c;      /* the cell being covered on the current level */
  register node *p;      /* the current node of interest */
  register cell *q;      /* the current cell of interest */
  register option *r;      /* the current option of interest */
  int ii, jj, kk, t;
  node *pp;
```

This code is used in section 1.

25. As usual, I'm using labels and **goto** statements as I backtrack, and making only a half-hearted apology for my outrageous style.

```
⟨ Backtrack through all solutions 25 ⟩ ≡
  ⟨ Initialize for level 0 46 ⟩;
  l = 1; goto choose;
advance: ⟨ Remove options that cover cells other than best_cell[l] 29 ⟩;
  if (verbose) ⟨ Handle diagnostic info 44 ⟩;
  l++;
choose: ⟨ Choose the moves at level l 26 ⟩;
backup: l--;
  if (l ≡ 0) goto done;
  ⟨ Unremove options that cover cells other than best_cell[l] 30 ⟩;
  goto unmark;      /* reconsider the move on level l */
solution: ⟨ Record a solution 42 ⟩;
  goto backup; done:
```

This code is used in section 1.

26. The usual trick in backtracking is to update the data structures in such a way that we can faithfully downdate them as we back up. The harder cases, namely the symcheck list and the constraint list, are explicitly recomputed on each level so that downdating is unnecessary. The *force_ptr* array is used to remember where forcing moves need to be downdating.

```
⟨ Choose the moves at level l 26 ⟩ ≡
  ⟨ Select c = best_cell[l], or goto solution if all cells are covered 31 ⟩;
  force_ptr[l] = force_ptr[l − 1];
  cover(c);      /* remove options that cover best_cell[l] */
  c¬empty = 1;
  ⟨ Set a_l to the empty option of c; goto try_again if that option isn't allowed 41 ⟩;
try: ⟨ Mark the newly covered elements 32 ⟩;
  ⟨ Compute the new constraint list; goto unmark if previous choices are disallowed 34 ⟩;
  ⟨ Compute the new symcheck list; goto unmark if a_1 a_2 . . . a_l is rejected 40 ⟩;
  goto advance;
unmark: ⟨ Unmark the newly covered elements 33 ⟩;
  ⟨ Delete the new forcing table entries 39 ⟩;
try_again: move[l] = move[l]¬up;
  best_cell[l]¬empty = 0;
  if (move[l]¬right ≠ move[l]) goto try;      /* a_l not the empty option */
  c = best_cell[l];
  uncover(c);
```

This code is used in section 25.

27. Here's a subroutine that removes all options that cover cell c from all cell lists except list c.

```
⟨ Subroutines 19 ⟩ +≡
  cover(c)
      cell *c;
  { register cell *l, *r;
    register node *rr, *pp, *uu, *dd;
    l = c→prev; r = c→next;
    l→next = r; r→prev = l;
    for (rr = c→head.down; rr ≠ &(c→head); rr = rr→down)
      for (pp = rr→right; pp ≠ rr; pp = pp→right) {
        uu = pp→up; dd = pp→down;
        uu→down = dd; dd→up = uu;
        pp→col→len−−;
      }
  }
```

28. 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—because the old pointers still exist! (I think this technique was invented in Japan.)

```
⟨ Subroutines 19 ⟩ +≡
  uncover(c)
      cell *c;
  { register cell *l, *r;
    register node *rr, *pp, *uu, *dd;
    for (rr = c→head.up; rr ≠ &(c→head); rr = rr→up)
      for (pp = rr→left; pp ≠ rr; pp = pp→left) {
        uu = pp→up; dd = pp→down;
        uu→down = dd→up = pp;
        pp→col→len++;
      }
    l = c→prev; r = c→next;
    l→next = r→prev = c;
  }
```

29. ⟨ Remove options that cover cells other than *best_cell*[*l*] 29 ⟩ ≡

```
for (p = move[l]→right; p ≠ move[l]; p = p→right) cover(p→col);
```

This code is used in section 25.

30. ⟨ Unremove options that cover cells other than *best_cell*[*l*] 30 ⟩ ≡

```
for (p = move[l]→left; p ≠ move[l]; p = p→left) uncover(p→col);
```

This code is used in section 25.

31. ⟨ Select $c = best\_cell[l]$, or **goto** *solution* if all cells are covered 31 ⟩ ≡

```
q = root.next;
if (q ≡ &root) goto solution;
for (c = q, j = q→len, q = q→next; q ≠ &root; q = q→next)
  if (q→len < j) c = q, j = q→len;
best_cell[l] = c;
```

This code is used in section 26.

32. ⟨ Mark the newly covered elements 32 ⟩ ≡
```
for (p = move[l]→right; p ≠ move[l]; p = p→right) {
  p→col→filled = p;
  p→col→nonempty++;
}
p→col→filled = p;
if (p→right ≠ p) p→col→nonempty++;
```
This code is used in section 26.

33. ⟨ Unmark the newly covered elements 33 ⟩ ≡
```
for (p = move[l]→left; p ≠ move[l]; p = p→left) {
  p→col→filled = Λ;
  p→col→nonempty--;
}
p→col→filled = Λ;
if (p→right ≠ p) p→col→nonempty--;
```
This code is used in section 26.

34. ⟨ Compute the new constraint list; **goto** *unmark* if previous choices are disallowed 34 ⟩ ≡
```
j = constraint_ptr[l − 1];
k = constraint_ptr[l];
if (p→right ≡ p)
  ⟨ Delete current cell from the constraint list, possibly forcing other cells to be nonempty 35 ⟩
else {
  ⟨ Add new constraints; goto unmark if previous choices are disallowed 37 ⟩;
  ⟨ Copy former constraints that are still unsatisfied 38 ⟩;
}
constraint_ptr[l + 1] = k;
```
This code is used in section 26.

35. ⟨ Delete current cell from the constraint list, possibly forcing other cells to be nonempty 35 ⟩ ≡
```
{
  c = p→col;
  while (j < constraint_ptr[l]) {
    kk = k;
    while ((q = constraint[j])) {
      if (q ≠ c) constraint[k++] = q;
      j++;
    }
    j++;
    if (k ≡ kk + 1) ⟨ Force constraint[kk] to be nonempty 36 ⟩
    else constraint[k++] = Λ;
  }
}
```
This code is used in section 34.

36. ⟨ Force *constraint*[*kk*] to be nonempty 36 ⟩ ≡

```
{
  k = kk;
  q = constraint[k];
  if (¬q→nonempty) {
    q→nonempty = 1;
    q→len −−;
    force[force_ptr[l]++] = q;
  }
}
```

This code is used in section 35.

37. ⟨ Add new constraints; **goto** *unmark* if previous choices are disallowed 37 ⟩ ≡

```
r = p→row;
for (i = 0; i < r→neighbor_ptr; i++) {
  kk = k;
  while ((q = r→neighbor[i])) {
    if (q→nonempty) {      /* constraint is satisfied */
      do i++; while (r→neighbor[i]);
      goto no_problem;
    }
    else if (¬q→empty) constraint[k++] = q;
    i++;
  }
  if (k > kk + 1) {
    constraint[k++] = Λ;
    continue;
  }
  if (k ≡ kk) goto unmark;      /* all were covered by empty cells */
  q = constraint[kk];
  q→nonempty = 1;
  q→len −−;
  force[force_ptr[l]++] = q;
no_problem: k = kk;
}
```

This code is used in section 34.

38. ⟨ Copy former constraints that are still unsatisfied 38 ⟩ ≡

```
while (j < constraint_ptr[l]) {
  kk = k;
  while ((q = constraint[j])) {
    if (q→nonempty) goto flush;      /* constraint is satisfied */
    constraint[k++] = q;
    j++;
  }
  constraint[k++] = Λ;
  j++;
  continue;
flush: do j++; while (constraint[j]);
  k = kk;
  j++;
}
```

This code is used in section 34.

39. ⟨ Delete the new forcing table entries 39 ⟩ ≡

```
while (force_ptr[l] ≠ force_ptr[l − 1]) {
  q = force[−−force_ptr[l]];
  q→len ++;
  q→nonempty = 0;
}
```

This code is used in section 26.

40. ⟨ Compute the new symcheck list; **goto** *unmark* if $a_1\, a_2 \ldots a_l$ is rejected 40 ⟩ ≡

```
for (k = symcheck_ptr[l − 1], kk = symcheck_ptr[l]; k < symcheck_ptr[l]; k++) {
  for (i = symcheck_sig[k], j = symcheck_j[k] + 1; j ≤ l; j++) {
    c = best_cell[j]→invsym[i];      /* σc_i */
    if (¬c→filled) break;
    p = c→filled→sym[i];      /* a'_i */
    if (p < move[j]) goto unmark;
    if (p > move[j]) goto okay;
  }
  symcheck_sig[kk] = i;
  symcheck_j[kk] = j − 1;
  kk++;
okay: ;
}
symcheck_ptr[l + 1] = kk;
```

This code is used in section 26.

41. ⟨ Set a_l to the empty option of c; **goto** *try_again* if that option isn't allowed 41 ⟩ ≡

```
move[l] = &(c→head);
if (c→nonempty) goto try_again;
```

This code is used in section 26.

42. ⟨ Record a solution 42 ⟩ ≡

```
count++;
if (verbose) {
  if (count % spacing ≡ 0) {
    printf("%d:␣", count);
    for (j = 1; j < l; j++) print_move(move[j]);
    if (symcheck_ptr[l] ≡ symcheck_ptr[l − 1]) printf("(1␣sym,␣%d␣blks)\n", (ll ∗ mm ∗ nn + 1 − l)/3);
    else
      printf("(%d␣syms,␣%d␣blks)\n", symcheck_ptr[l] − symcheck_ptr[l − 1] + 1, (ll ∗ mm ∗ nn + 1 − l)/3);
  }
}
```

This code is used in section 25.

43. ⟨ Subroutines 19 ⟩ +≡

```
print_move(p)
    node ∗p;
{
  register node ∗q;
  for (q = p→right; q ≠ p; q = q→right) printf("%s-", q→col→name);
  printf("%s␣", q→col→name);
}
```

44. ⟨ Handle diagnostic info 44 ⟩ ≡

```
{
  profile[l]++;
  prof_syms[l] += symcheck_ptr[l + 1] − symcheck_ptr[l] + 1;
  prof_cons[l] += constraint_ptr[l + 1] − constraint_ptr[l];
  prof_frcs[l] += force_ptr[l] − force_ptr[l − 1];
  if (verbose > 1) {
    printf("Level␣%d,␣", l);
    print_move(move[l]);
    printf("(%d,%d,%d)\n", symcheck_ptr[l + 1] − symcheck_ptr[l] + 1,
        constraint_ptr[l + 1] − constraint_ptr[l], force_ptr[l] − force_ptr[l − 1]);
  }
}
```

This code is used in section 25.

45. ⟨ Print a profile of the search tree 45 ⟩ ≡

```
{
  for (j = 1; j ≤ ll ∗ mm ∗ nn; j++)
    printf("␣Level␣%d:␣%d␣sols,␣%#.1f␣syms,␣%#.1f␣cons,␣%#.1f␣frcs\n", j, profile[j], (double)
        prof_syms[j]/(double) profile[j], (double) prof_cons[j]/(double) profile[j], (double)
        prof_frcs[j]/(double) profile[j]);
}
```

This code is used in section 1.

46. ⟨ Initialize for level 0 46 ⟩ ≡

```
for (i = 0; i < ll; i++)
  for (j = 0; j < mm; j++)
    for (k = 0; k < nn; k++) {
      c = &cells[i][j][k];
      c→init_len = c→len;
    }
for (k = 0; k < ss; k++) symcheck_sig[k] = k + 1;
symcheck_ptr[1] = ss − 1;
```

This code is used in section 25.

47. ⟨ Make redundancy checks to see if the backtracking was consistent 47 ⟩ ≡

```
q = &root;
for (i = 0; i < ll; i++)
  for (j = 0; j < mm; j++)
    for (k = 0; k < nn; k++) {
      c = &cells[i][j][k];
      if (c→nonempty ∨ c→len ≠ c→init_len ∨ c→prev ≠ q ∨ q→next ≠ c)
        printf("Trouble␣at␣cell␣%s!\n", c→name);
      q = c;
    }
```

This code is used in section 1.

48. Index.

ANTISLIDE3