1. Intro. This is an experimental program in which I try to cut a square into a given number of pieces, in such a way that the pieces can be reassembled to fill another given shape. (Everything is done pixelwise, without "diagonal cuts.") The pieces can be rotated but not flipped over.

I don't insist that the pieces be internally connected. With change files I can add further restrictions.

The input on *stdin* is a sequence of lines containing periods and asterisks, where the asterisks mark usable positions. The number of asterisks should be a perfect square.

The desired number of pieces is a command-line parameter.

```
#define maxn 32   /* maximum number of input lines and characters per line */
#define maxd 7   /* maximum number of pieces */
#define bufsize maxn + 5   /* size of the input buffer */

#include <stdio.h>
#include <stdlib.h>
  ⟨ Type definitions 25 ⟩
  int d;   /* command-line parameter: the number of colors */
  char buf[bufsize];
  int maxrow;   /* largest row number used in the shape */
  int maxcol;   /* largest column number used in the shape */
  char aname[maxn * maxn][8];   /* symbolic names of the cells in the square */
  char bname[maxn * maxn][8];   /* symbolic names of the cells in the shape */
  int site[maxn * maxn];   /* where the cells are in the shape */
  int vbose;   /* level of verbosity */
  ⟨ Global variables 11 ⟩;
  ⟨ Subroutines 33 ⟩;
  main(int argc, char *argv[])
  {
    register int a, b, dd, i, j, k, l, ll, lll, m, n, nn, slack;
    ⟨ Process the command line 2 ⟩;
    ⟨ Input the shape 3 ⟩;
    ⟨ Find all solutions 6 ⟩;
    ⟨ Print statistics about the run 41 ⟩;
  }
```

2. ⟨ Process the command line 2 ⟩ ≡

```
  if (argc < 2 ∨ sscanf(argv[1], "%d", &d) ≠ 1) {
    fprintf(stderr, "Usage:␣%s␣d␣[verbose]␣[extra␣verbose]␣<␣foo.dots\n", argv[0]);
    exit(−1);
  }
  if (d < 2 ∨ d > maxd) {
    fprintf(stderr, "The␣number␣of␣pieces␣should␣be␣between␣2␣and␣%d,␣not␣%d!\n", maxd, d);
    exit(−2);
  }
  vbose = argc − 2;
```

This code is used in section 1.

3. **#define** $place(i,j)$ $((i)*maxn+(j))$

⟨ Input the shape 3 ⟩ ≡

```
for (i = nn = 0; ; i++) {
  if (¬fgets(buf, bufsize, stdin)) break;
  if (i ≥ maxn) {
    fprintf(stderr, "Recompile␣me:␣I␣allow␣at␣most␣%d␣lines␣of␣input!\n", maxn);
    exit(−3);
  }
  ⟨ Input row i of the shape 4 ⟩;
}
maxrow = i − 1;
if (maxrow < 0) {
  fprintf(stderr, "There␣was␣no␣input!\n");
  exit(−666);
}
fprintf(stderr, "OK,␣I've␣got␣a␣shape␣with␣%d␣lines␣and␣%d␣cells.\n", i, nn);
for (n = 1; n * n < nn; n++) ;    /* the shape has nn asterisks */
if (n * n ≠ nn) {
  fprintf(stderr, "The␣number␣of␣cells␣should␣be␣a␣positive␣perfect␣square!\n");
  exit(−4);
}
for (i = 0; i < n; i++)
  for (j = 0; j < n; j++) sprintf(aname[place(i, j)], "%02da%02d", i, j);
complement = place(n − 1, n − 1);
```

This code is used in section 1.

4. ⟨ Input row i of the shape 4 ⟩ ≡

```
for (j = 0; buf[j] ∧ buf[j] ≠ '\n'; j++) {
  if (buf[j] ≡ '*') {
    if (j > maxcol) {
      maxcol = j;
      if (j ≥ maxn) {
        fprintf(stderr, "Recompile␣me:␣I␣allow␣at␣most␣%d␣columns␣of␣input!\n", maxn);
        exit(−5);
      }
    }
    site[nn++] = place(i, j);
    sprintf(bname[place(i, j)], "%02db%02d", i, j);
  }
}
```

This code is used in section 3.

5. The algorithm. Let's consider a special case of the problem, in order to build some intuition and clarify the concepts. Suppose the input is

```
****
*..*
.***
```

and we want to cut the eight cells specified by these asterisks into $d = 2$ pieces that can be assembled into a 3×3 square. You can probably see one way to do the job: Break off the two cells in the leftmost column, rotate them 90°, and stick them into the "jaws" of the remaining seven. How can we get a computer to discover this?

A solution to the general problem can be regarded as a way to color the square with d colors. The cells of the square are (i, j) for $0 \le i, j < n$, and each of these cells is supposed to be mapped into a distinct position (i', j') of the other shape, by rotation and shifting. The amount of rotation and shift must be the same for all cells of the same color. In the example, we can color the cells

```
111
221
111
```

and map those of color `1` by shifting one space right; those of color `2` are mapped by, say, rotating 90° clockwise about the square's center, then shifting one space left. The result is

```
2111
2..1
.111
```

as desired. These two color labelings are written to *stdout*.

There's always a set of allowable shift amounts, (a_0, b_0), (a_1, b_1), …, (a_{m-1}, b_{m-1}); these are the ways to shift the square so that it overlaps the other shape in at least one cell. Our example problem has $m = 29$ such shifts, namely $\bar2\bar2$, $\bar2\bar1$, $\bar20$, $\bar21$, $\bar22$, $\bar23$, $\bar1\bar2$, $\bar1\bar1$, $\bar10$, $\bar11$, $\bar12$, $\bar13$, $0\bar2$, $0\bar1$, 00, 01, 02, 03, $1\bar2$, $1\bar1$, 10, 11, 12, 13, $2\bar1$, 20, 21, 22, 23. (Here $\bar2$ stands for -2, and so on; our program uses row-and-column coordinates (i, j), so the left coordinate of a shift refers to shifting downward and the right coordinate refers to shifting rightward. This list of acceptable shifts includes all values (a, b) with $-2 \le a \le 2$ and $-2 \le b \le 3$ *except* for $2\bar2$. The latter is omitted, because shifting the square down 2 and left 2 does not intersect with the other shape.)

Each mapping can be specified by a pair (s, t) where $0 \le s < m$ and $0 \le t < 4$, meaning "rotate $90t$ degrees clockwise, then shift by (a_s, b_s)." The outer loop of the algorithm below runs through all possible mappings (s_1, t_1), …, (s_d, t_d), and tries to solve the corresponding bipartite matching problem that involves those maps. For example, if (s_1, t_1) is the map "shift right 1" and (s_2, t_2) is the map "rotate 90 and shift left 1," the bipartite graph for which a perfect matching describes a solution to the example problem has the edges

`0a0` — `0b1`, `0a1` — `0b2`, `0a2` — `0b3`, `1a2` — `1b3`, `2a0` — `2b1`, `2a1` — `2b2`, `2a2` — `2b3`

for color `1` and

`0a0` — `0b1`, `0a2` — `2b1`, `1a0` — `0b0`, `1a1` — `1b0`

for color `2`. (Here `0a0` stands for the cell in row 0 and column 0 of the square, while `0b0` stands for the cell in row 0 and column 0 of the other shape.) Notice that the edge `0a0` — `0b1` occurs *twice*, once for each color; this leads to *another* solution:

```
211   2211
221   2..1
111   .111
```

6. We save a factor of roughly $d!$ by assuming that

$$(s_1,t_1) \le (s_2,t_2) \le \cdots \le (s_d,t_d), \qquad \text{lexicographically,}$$

because a permutation of the colors doesn't change the solution. Furthermore we gain another factor of 4 by assuming that $t_1 = 0$, because rotation is a symmetry of the square.

(I could actually have written $(s_1,t_1) < \cdots < (s_d,t_d)$, with '$<$' instead of '$\le$'; the case $(s_k,t_k) = (s_{k+1},t_{k+1})$ won't occur in a solution for minimum d, because colors k and $k+1$ could be merged in such a case. However, equality might arise in extensions of this problem that involve further constraints. For example, we might require color classes to be connected, or to have a bounded size.)

Most of the matching problems that arise are obviously unsolvable, because they have isolated vertices. And most of those that remain are quite easy to solve, because many vertices have degree 1 and their partner is forced. The algorithm looks at all $\binom{m+d-1}{d}$ sets of shifts with $s_1 \le \cdots \le s_d$, and explores further only if those shifts cover all cells of the given shape. In the latter case, 4^{d-1} choices of $t_2, \ldots, t_d$ are considered, and the matching process is inaugurated only if those rotations cover all cells of the square.

For example, the only shifts that cover more than four cells of the shape in our toy problem are 00, 01, and 02. At least one of these is needed, because we need to cover nine cells with two shifts. Thus $\binom{m+d-1}{d}$ is not a scary number of subproblems to consider.

⟨Find all solutions 6⟩ ≡
  ⟨Generate the table of legal shifts 8⟩;
  while (1) {
    ⟨If the shape isn't covered by $\{s_1,\ldots,s_d\}$, **goto** *shapenot* 9⟩;
    counta++;
    ⟨Run through all sequences of shifts, $(t_2,\ldots,t_d)$ 7⟩;
  shapenot: **for** $(k = d;\ s[k] \equiv m-1;\ k--)$;
    if $(k \equiv 0)$ **break**;
    for $(j = s[k]+1;\ k \le d;\ k{+}{+})\ s[k] = j$;
  }

This code is used in section 1.

7. ⟨Run through all sequences of shifts, $(t_2,\ldots,t_d)$ 7⟩ ≡
  for $(k = 2;\ k \le d;\ k{+}{+})\ t[k] = 0$;
  while (1) {
    for $(k = 2;\ k \le d;\ k{+}{+})$
      if $(s[k] \equiv s[k-1] \land t[k] \equiv t[k-1])$ **goto** *squarenot*;
    ⟨If the square isn't covered by $\{(s_1,t_1),\ldots,(s_d,t_d)\}$, **goto** *squarenot* 10⟩;
    countb++;
    ⟨Check for a perfect matching 12⟩;
  squarenot: **for** $(k = d;\ t[k] \equiv 3;\ k--)\ t[k] = 0$;
    if $(k \equiv 1)$ **break**;
    $t[k]{+}{+}$;
  }

This code is used in section 6.

8. ⟨ Generate the table of legal shifts 8 ⟩ ≡

```
for (m = 0, a = 1 − n; a ≤ maxrow; a++)
  for (b = 1 − n; b ≤ maxcol; b++) {
    for (k = 0, i = (a < 0 ? −a : 0); i < n ∧ a + i ≤ maxrow; i++)
      for (j = (b < 0 ? −b : 0); j < n ∧ b + j ≤ maxcol; j++)
        if (bname[place(a + i, b + j)][0]) bcover[m][k++] = place(a + i, b + j);
    if (k) {
      if (vbose > 1) fprintf(stderr, "␣S[%d]=(%d,%d)\n", m, a, b);
      shift[m] = place(a, b), bcovered[m++] = k;
    }
  }
if (vbose) fprintf(stderr, "There␣are␣%d␣legal␣shifts.\n", m);
```

This code is used in section 6.

9. ⟨ If the shape isn't covered by $\{s_1, \ldots, s_d\}$, **goto** *shapenot* 9 ⟩ ≡

```
for (slack = −nn, k = 1; k ≤ d; k++) slack += bcovered[s[k]];
if (slack < 0) goto shapenot;
for (k = 0; k < nn; k++) blen[site[k]] = 0;
for (k = 1; k ≤ d; k++) {
  for (i = 0, j = s[k]; i < bcovered[j]; i++) {
    l = bcover[j][i];
    if (¬blen[l]) blen[l] = 1;
    else {
      if (¬slack) goto shapenot;
      slack−−;
      blen[l]++;
    }
  }
}
```

This code is used in section 6.

10. While we make the second check for coverage, we also build the table of edges. Each edge is represented by its color and the position of the neighbor.

#define *pack*(c,p) $(((c) \ll 16) + (p))$

⟨ If the square isn't covered by $\{(s_1,t_1),\ldots,(s_d,t_d)\}$, **goto** *squarenot* 10 ⟩ ≡

```
for (k = 0; k < nn; k++) blen[site[k]] = 0;
for (i = 0; i < n; i++)
  for (j = 0; j < n; j++) alen[place(i, j)] = 0;
for (slack = −nn, k = 1; k ≤ d; k++) slack += bcovered[s[k]];
for (k = 1; k ≤ d; k++) {
  for (i = 0, j = s[k]; i < bcovered[j]; i++) {
    l = bcover[j][i];
    ll = l − shift[j];
    if (t[k] & 1) {
      register int q = ll/maxn, r = ll % maxn;
      ll = place(r, n − 1 − q);    /* rotate clockwise */
    }
    if (t[k] & 2) ll = complement − ll;
    if (alen[ll]) {
      if (¬slack) goto squarenot;
      slack −−;
    }
    aa[ll][alen[ll]++] = pack(k, l);
    bb[l][blen[l]++] = pack(k, ll);
  }
}
```

This code is used in section 7.

11. ⟨ Global variables 11 ⟩ ≡

```
char alen[maxn * maxn];    /* how many moves remain at this cell in the square */
char blen[maxn * maxn];    /* how many moves remain at this cell in the shape */
int aa[maxn * maxn][maxd];    /* moves for the square */
int bb[maxn * maxn][maxd];    /* moves for the shape */
int shift[4 * maxn * maxn];    /* offsets in the shifts */
int complement;    /* offset used for 180-degree rotation */
int bcover[4 * maxn * maxn][maxn * maxn];    /* cells covered by the shifts */
int bcovered[4 * maxn * maxn];    /* how many cells are covered */
int s[maxd + 1];    /* the current sequence of shifts */
int t[maxd + 1];    /* the current sequence of rotations */
```

See also sections 22 and 28.

This code is used in section 1.

12. Prematching. When we've managed to jump through all those hoops, we're left with a perfect matching problem. And most of the time that matching problem is quite trivial; so we might as well throw out the easy cases before trying to do anything fancy.

In most cases some of the moves turn out to be forced, because a cell of the square has only one possible shape-mate or vice versa. We start by making all of those no-brainer moves.

```
⟨Check for a perfect matching 12⟩ ≡
  if (vbose > 1) ⟨Display the matching problem on stderr 13⟩;
  ⟨Make forced moves from the square, or goto done 14⟩;
  countc++;
  ⟨Make forced moves from the shape, or goto done 16⟩;
  ⟨Make all remaining forced moves 18⟩;
  countd++;
  ⟨Find all perfect matchings in the remaining bigraph 23⟩;
  done:
```

This code is used in section 7.

13. ⟨Display the matching problem on *stderr* 13⟩ ≡

```
  {
    fprintf(stderr, "␣Trying␣to␣match");
    for (k = 1; k ≤ d; k++) fprintf(stderr, "␣%d^%d", s[k], t[k]);
    fprintf(stderr, ":\n");
    for (i = 0; i < n; i++)
      for (j = 0; j < n; j++) {
        fprintf(stderr, "␣␣%s␣--", aname[place(i, j)]);
        for (k = 0; k < alen[place(i, j)]; k++)
          fprintf(stderr, "␣%s.%d", bname[aa[place(i, j)][k] & #ffff], aa[place(i, j)][k] ≫ 16);
        fprintf(stderr, "\n");
      }
  }
```

This code is used in section 12.

14. ⟨Make forced moves from the square, or **goto** *done* 14⟩ ≡

```
  for (acount = i = 0; i < n; i++)
    for (j = 0; j < n; j++) {
      if (alen[place(i, j)] > 1) apos[place(i, j)] = acount, alist[acount++] = place(i, j);
      else {
        l = aa[place(i, j)][0] & #ffff;
        if (¬blen[l]) goto done;    /* that position of the shape is already taken */
        acolor[place(i, j)] = bcolor[l] = aa[place(i, j)][0] ≫ 16;
        if (blen[l] ≡ 1) blen[l] = 0;
        else ⟨Remove all other edges that go to shape position l 15⟩;
      }
    }
```

This code is used in section 12.

15. Premature optimization is the root of all evil in programming. Yet I couldn't resist trying to make this program efficient in special cases.

The removal of edges might reduce *alen* to 1 for square cells that are in the *alist*, thus forcing further moves. I won't worry about that until later.

⟨Remove all other edges that go to shape position *l* 15⟩ ≡

```
{
  for (k = 0; k < blen[l]; k++) {
    ll = bb[l][k] & #ffff;
    if (ll ≠ place(i, j)) {
      register int opp = (bb[l][k] & #ffff0000) + l;    /* the opposite version of this edge */
      dd = alen[ll] − 1, alen[ll] = dd;
      if (¬dd) goto done;
      for (a = 0; aa[ll][a] ≠ opp; a++) ;
      if (a > dd) debug("ahi");
      if (a ≠ dd) aa[ll][a] = aa[ll][dd];
    }
  }
  blen[l] = 0;
}
```

This code is used in section 14.

16. ⟨Make forced moves from the shape, or **goto** *done* 16⟩ ≡

```
if (acount) {
  for (bcount = i = 0; i < nn; i++) {
    l = site[i];
    if (¬blen[l]) continue;    /* we've been forced to match this cell already */
    if (blen[l] > 1) bpos[l] = bcount, blist[bcount++] = l;
    else {
      ll = bb[l][0] & #ffff;
      if (¬alen[ll]) goto done;    /* that position of the square is already taken */
      acolor[ll] = bcolor[l] = bb[l][0] ≫ 16;
      acount−−;
      ⟨Make square cell ll inactive 17⟩;
    }
  }
  if (acount ≠ bcount) debug("count␣mismatch");
}
```

This code is used in section 12.

17. ⟨ Make square cell ll inactive 17 ⟩ ≡

```
j = apos[ll];
if (j ≠ acount) lll = alist[acount], alist[j] = lll, apos[lll] = j;
if (alen[ll] ≠ 1) {
  for (k = 0; k < alen[ll]; k++) {
    lll = aa[ll][k] & #ffff;
    if (lll ≠ l) {
      register int opp = (aa[ll][k] & #ffff0000) + ll;    /* the opposite version of this edge */
      dd = blen[lll] − 1, blen[lll] = dd;
      if (¬dd) goto done;
      for (b = 0; bb[lll][b] ≠ opp; b++) ;
      if (b > dd) debug("bhi");
      if (b ≠ dd) bb[lll][b] = bb[lll][dd];
    }
  }
  alen[ll] = 0;
}
```

This code is used in sections 16 and 21.

18. Beware: I'm using *acount* and *bcount* in a somewhat tricky way here: The old *acount* is kept in *bcount* so that a change can be detected. (Again I apologize for weak resistance.)

⟨ Make all remaining forced moves 18 ⟩ ≡

```
while (acount) {
  for (i = 0; i < acount; i++)
    if (alen[ll = alist[i]] ≡ 1) ⟨ Force a move from ll 19 ⟩;
  for (i = 0; i < acount; i++)
    if (blen[l = blist[i]] ≡ 1) ⟨ Force a move from l 21 ⟩;
  if (acount ≡ bcount) break;
  bcount = acount;
}
```

This code is used in section 12.

19. ⟨ Force a move from ll 19 ⟩ ≡

```
{
  acount−−;
  if (i < acount) lll = alist[acount], alist[i] = lll, apos[lll] = i−−;
  l = aa[ll][0] & #ffff;
  acolor[ll] = bcolor[l] = aa[ll][0] ≫ 16;
  ⟨ Make shape cell l inactive 20 ⟩;
}
```

This code is used in section 18.

20. ⟨ Make shape cell l inactive 20 ⟩ ≡

```
j = bpos[l];
if (j < acount) lll = blist[acount], blist[j] = lll, bpos[lll] = j;
if (blen[l] ≠ 1) {
  for (k = 0; k < blen[l]; k++) {
    lll = bb[l][k] & #ffff;
    if (lll ≠ ll) {
      register int opp = (bb[l][k] & #ffff0000) + l;    /* the opposite version of this edge */
      dd = alen[lll] − 1, alen[lll] = dd;
      if (¬dd) goto done;
      for (a = 0; aa[lll][a] ≠ opp; a++) ;
      if (a > dd) debug("chi");
      if (a ≠ dd) aa[lll][a] = aa[lll][dd];
    }
  }
}
```

This code is used in section 19.

21. ⟨ Force a move from l 21 ⟩ ≡

```
{
  acount−−;
  if (i < acount) lll = blist[acount], blist[i] = lll, bpos[lll] = i−−;
  ll = bb[l][0] & #ffff;
  acolor[ll] = bcolor[l] = bb[l][0] ≫ 16;
  ⟨ Make square cell ll inactive 17 ⟩;
}
```

This code is used in section 18.

22. ⟨ Global variables 11 ⟩ +≡

```
int alist[maxn ∗ maxn], blist[maxn ∗ maxn];    /* list of cells not yet matched */
int apos[maxn ∗ maxn], bpos[maxn ∗ maxn];    /* inverses of those lists */
int acount, bcount;    /* the lengths of those lists */
int acolor[maxn ∗ maxn], bcolor[maxn ∗ maxn];    /* color patterns in a solution */
unsigned long long count;    /* the number of solutions */
unsigned long long counta, countb, countc, countd, counte;
  /* the number of times we reached key points */
```

23. Matching. Sometimes we actually have real work to do.

At first I didn't think the problem would often be challenging. So I just used brute-force backtracking, à la Algorithm 7.2.2B.

But a surprising number of large subproblems arose. So I'm now implementing a version of the original dancing links algorithm, hacked from DANCE.

```
⟨ Find all perfect matchings in the remaining bigraph 23 ⟩ ≡
  if (acount ≡ 0) ⟨ Print a solution 40 ⟩
  else {
    ⟨ Local variables 27 ⟩;
    counte++;
    if (vbose > 1) ⟨ Display the remaining matching problem on stderr 24 ⟩;
    ⟨ Initialize for dancing 29 ⟩;
    ⟨ Dance 31 ⟩;
  }
```

This code is used in section 12.

24.
```
⟨ Display the remaining matching problem on stderr 24 ⟩ ≡
  {
    fprintf(stderr, "␣which␣reduces␣to:\n");
    for (i = 0; i < acount; i++) {
      fprintf(stderr, "␣␣%s␣--", aname[alist[i]]);
      for (k = 0; k < alen[alist[i]]; k++)
        fprintf(stderr, "␣%s.%d", bname[aa[alist[i]][k] & #ffff], aa[alist[i]][k] ≫ 16);
      fprintf(stderr, "\n");
    }
  }
```

This code is used in section 23.

25. The DANCE program was developed to solve exact cover problems, and bipartite matching is a particularly easy case of that problem: Every column to be covered is a primary column, and every row specifies exactly two primary columns.

Each column of the exact cover 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 six fields. Four are the pointers of doubly linked lists, already mentioned; the fifth points to the column containing the node; the sixth ties this node to the dissection problem we're solving.

```
⟨ Type definitions 25 ⟩ ≡
  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 info;   /* square position, shape position, and color of this edge */
  } node;
```

See also section 26.

This code is used in section 1.

26. 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 25 ⟩ +≡
```
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;   /* symbolic identification of the column, for printing */
  struct col_struct *prev, *next;   /* neighbors of this column */
} column;
```

27. 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 */
```

⟨ Local variables 27 ⟩ ≡
```
register column *cur_col;
register node *cur_node;
```

See also sections 32 and 38.

This code is used in section 23.

28.
```
#define max_cols (2 * maxn * maxn)
#define max_nodes (maxn * maxn * maxn * maxn * maxd)
```

⟨ Global variables 11 ⟩ +≡
```
column col_array[max_cols + 2];   /* place for column records */
node node_array[max_nodes];   /* place for nodes */
column *acol[maxn * maxn], *bcol[maxn * maxn];
node *choice[maxn * maxn];   /* the row and column chosen on each level */
```

29. ⟨ Initialize for dancing 29 ⟩ ≡
```
for (i = 0; i < acount; i++) {
  ll = alist[i], l = blist[i];
  acol[ll] = &col_array[i + i + 1], col_array[i + i + 1].name = aname[ll];
  bcol[l] = &col_array[i + i + 2], col_array[i + i + 2].name = bname[l];
}
root.prev = &col_array[acount + acount];
root.prev->next = &root;
for (cur_col = col_array + 1; cur_col <= root.prev; cur_col++) {
  cur_col->head.up = cur_col->head.down = &cur_col->head;
  cur_col->len = 0;
  cur_col->prev = cur_col - 1, (cur_col - 1)->next = cur_col;
}
for (cur_node = node_array, i = 0; i < acount; i++) {
  ll = alist[i];
  for (k = 0; k < alen[ll]; k++) ⟨ Create the node for the kth edge from ll 30 ⟩;
}
```

This code is used in section 23.

30. ⟨ Create the node for the kth edge from *ll* 30 ⟩ ≡

```
{
  register column *ccol;
  l = aa[ll][k] & #ffff;
  j = ((aa[ll][k] >> 16) << 24) + (l << 12) + ll;
  ccol = acol[ll];
  cur_node->left = cur_node->right = cur_node + 1;
  cur_node->col = ccol, cur_node->info = j;
  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++;
  ccol = bcol[l];
  cur_node->left = cur_node->right = cur_node - 1;
  cur_node->col = ccol, cur_node->info = j;
  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++;
}
```

This code is used in section 29.

31. 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.

⟨ Dance 31 ⟩ ≡

```
  level = 0;
forward: ⟨ Set best_col to the best column for branching 37 ⟩;
  cover(best_col);
  cur_node = choice[level] = best_col->head.down;
advance:
  if (cur_node == &(best_col->head)) goto backup;
  if (vbose > 1) fprintf(stderr, "L%d: %s %s\n", level, cur_node->col->name, cur_node->right->col->name);
  ⟨ Cover all other columns of cur_node 35 ⟩;
  if (root.next == &root) ⟨ Record solution and goto recover 39 ⟩;
  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 36 ⟩;
  cur_node = choice[level] = cur_node->down; goto advance;
```

This code is used in section 23.

32. ⟨ Local variables 27 ⟩ +≡

```
register int level;
register column *best_col;   /* column chosen for branching */
```

33. 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 33 ⟩ ≡

```
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 −−;
    }
}
```

See also sections 34 and 42.

This code is used in section 1.

34. 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 33 ⟩ +≡

```
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;
}
```

35. ⟨ Cover all other columns of *cur_node* 35 ⟩ ≡

```
cover(cur_node→right→col);
```

This code is used in section 31.

36. 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.)

(Thus the present implementation is overkill, for the special case of bipartite matching.)

⟨ Uncover all other columns of *cur_node* 36 ⟩ ≡
  uncover(*cur_node*→*left*→*col*);

This code is used in section 31.

37. ⟨ Set *best_col* to the best column for branching 37 ⟩ ≡
  $minlen = max\_nodes$;
  if ($vbose > 2$) *fprintf*(*stderr*, "Level␣%d:", *level*);
  for ($cur\_col = root.next$; $cur\_col \neq \&root$; $cur\_col = cur\_col\rightarrow next$) {
    if ($vbose > 2$) *fprintf*(*stderr*, "␣%s(%d)", *cur_col*→*name*, *cur_col*→*len*);
    if ($cur\_col\rightarrow len < minlen$) $best\_col = cur\_col$, $minlen = cur\_col\rightarrow len$;
  }
  if ($vbose > 2$) *fprintf*(*stderr*, "␣branching␣on␣%s(%d)\n", *best_col*→*name*, *minlen*);

This code is used in section 31.

38. ⟨ Local variables 27 ⟩ +≡
  register int *minlen*;
  register int *j*, *k*, *x*;

39. ⟨ Record solution and **goto** *recover* 39 ⟩ ≡
  {
    if ($vbose > 1$) *fprintf*(*stderr*, "(a␣good␣dance)\n");
    for ($k = 0$; $k \le level$; $k{+}{+}$) {
      $j = choice[k]\rightarrow info$;
      $acolor[j \mathbin{\&} \#\mathtt{fff}] = bcolor[(j \gg 12) \mathbin{\&} \#\mathtt{fff}] = j \gg 24$;
    }
    ⟨ Print a solution 40 ⟩;
    goto *recover*;
  }

This code is used in section 31.

40. ⟨ Print a solution 40 ⟩ ≡

```
{
  register int OK = 1;    /* this (declaration facilitates change files) */
  if (OK) {
    count++;
    printf("Solution␣%lld,␣from", count);
    for (k = 1; k ≤ d; k++) printf("␣%d^%d", s[k], t[k]);
    printf(":\n");
    for (i = 0; i < n ∨ i ≤ maxrow; i++) {
      for (j = 0; j < n; j++) printf("%c", i < n ? acolor[place(i, j)] + '0' : '␣');
      if (i ≤ maxrow) {
        printf("␣␣");
        for (j = 0; j ≤ maxcol; j++)
          printf("%c", bname[place(i, j)][0] ? bcolor[place(i, j)] + '0' : '.');
      }
      printf("\n");
    }
  }
}
```

This code is used in sections 23 and 39.

41. ⟨ Print statistics about the run 41 ⟩ ≡

```
fprintf(stderr, "%lld␣solutions;␣run␣stats␣%d,%lld,%lld,%lld,%lld,%lld.\n", count, m, counta,
    countb, countc, countd, counte);
```

This code is used in section 1.

42. ⟨ Subroutines 33 ⟩ +≡

```
void debug(char *s)
{
  fflush(stdout);
  fprintf(stderr, "***%s!\n", s);
}
```

43. Index.

a: 1.
aa: 10, 11, 13, 14, 15, 17, 19, 20, 24, 30.
acol: 28, 29, 30.
acolor: 14, 16, 19, 21, 22, 39, 40.
acount: 14, 16, 17, 18, 19, 20, 21, 22, 23, 24, 29.
advance: 31.
alen: 10, 11, 13, 14, 15, 16, 17, 18, 20, 24, 29.
alist: 14, 15, 17, 18, 19, 22, 24, 29.
aname: 1, 3, 13, 24, 29.
apos: 14, 17, 19, 22.
argc: 1, 2.
argv: 1, 2.
b: 1.
backup: 31.
bb: 10, 11, 15, 16, 17, 20, 21.
bcol: 28, 29, 30.
bcolor: 14, 16, 19, 21, 22, 39, 40.
bcount: 16, 18, 22.
bcover: 8, 9, 10, 11.
bcovered: 8, 9, 10, 11.
best_col: 31, 32, 37.
blen: 9, 10, 11, 14, 15, 16, 17, 18, 20.
blist: 16, 18, 20, 21, 22, 29.
bname: 1, 4, 8, 13, 24, 29, 40.
bpos: 16, 20, 21, 22.
buf: 1, 3, 4.
bufsize: 1, 3.
c: 33, 34.
ccol: 30.
choice: 28, 31, 39.
col: 25, 30, 31, 33, 34, 35, 36.
col_array: 27, 28, 29.
col_struct: 25, 26.
column: 26, 27, 28, 30, 32, 33, 34.
complement: 3, 10, 11.
count: 22, 40, 41.
counta: 6, 22, 41.
countb: 7, 22, 41.
countc: 12, 22, 41.
countd: 12, 22, 41.
counte: 22, 23, 41.
cover: 31, 33, 35.
cur_col: 27, 29, 37.
cur_node: 27, 29, 30, 31, 35, 36.
d: 1.
dd: 1, 15, 17, 20, 33, 34.
debug: 15, 16, 17, 20, 42.
done: 12, 14, 15, 16, 17, 20, 31.
down: 25, 29, 30, 31, 33, 34.
exit: 2, 3, 4.
fflush: 42.
fgets: 3.
forward: 31.
fprintf: 2, 3, 4, 8, 13, 24, 31, 37, 39, 41, 42.
head: 26, 29, 30, 31, 33, 34.
i: 1.
info: 25, 30, 39.
j: 1, 38.
k: 1, 33, 38.
l: 1, 33, 34.
left: 25, 30, 34, 36.
len: 26, 29, 30, 33, 34, 37.
level: 31, 32, 37, 39.
ll: 1, 10, 15, 16, 17, 18, 19, 20, 21, 29, 30.
lll: 1, 17, 19, 20, 21.
m: 1.
main: 1.
max_cols: 28.
max_nodes: 28, 37.
maxcol: 1, 4, 8, 40.
maxd: 1, 2, 11, 28.
maxn: 1, 3, 4, 10, 11, 22, 28.
maxrow: 1, 3, 8, 40.
minlen: 37, 38.
n: 1.
name: 26, 27, 29, 31, 37.
next: 26, 29, 31, 33, 34, 37.
nn: 1, 3, 4, 9, 10, 16, 33, 34.
node: 25, 26, 27, 28, 33, 34.
node_array: 28, 29.
node_struct: 25.
`OK`: 40.
opp: 15, 17, 20.
pack: 10.
place: 3, 4, 8, 10, 13, 14, 15, 40.
prev: 26, 29, 33, 34.
printf: 40.
q: 10.
r: 10, 33, 34.
recover: 31, 39.
right: 25, 30, 31, 33, 35, 36.
root: 27, 29, 31, 37.
rr: 33, 34.
s: 11, 42.
shapenot: 6, 9.
shift: 8, 10, 11.
site: 1, 4, 9, 10, 16.
slack: 1, 9, 10.
sprintf: 3, 4.
squarenot: 7, 10.
sscanf: 2.
stderr: 2, 3, 4, 8, 13, 24, 31, 37, 39, 41, 42.

stdin: 1, 3.
stdout: 5, 42.
t: 11.
uncover: 31, 34, 36.
up: 25, 29, 30, 33, 34.
uu: 33, 34.
vbose: 1, 2, 8, 12, 23, 31, 37, 39.
x: 38.

⟨ Check for a perfect matching 12 ⟩ Used in section 7.
⟨ Cover all other columns of *cur_node* 35 ⟩ Used in section 31.
⟨ Create the node for the kth edge from *ll* 30 ⟩ Used in section 29.
⟨ Dance 31 ⟩ Used in section 23.
⟨ Display the matching problem on *stderr* 13 ⟩ Used in section 12.
⟨ Display the remaining matching problem on *stderr* 24 ⟩ Used in section 23.
⟨ Find all perfect matchings in the remaining bigraph 23 ⟩ Used in section 12.
⟨ Find all solutions 6 ⟩ Used in section 1.
⟨ Force a move from *ll* 19 ⟩ Used in section 18.
⟨ Force a move from l 21 ⟩ Used in section 18.
⟨ Generate the table of legal shifts 8 ⟩ Used in section 6.
⟨ Global variables 11, 22, 28 ⟩ Used in section 1.
⟨ If the shape isn't covered by $\{s_1, \ldots, s_d\}$, **goto** *shapenot* 9 ⟩ Used in section 6.
⟨ If the square isn't covered by $\{(s_1, t_1), \ldots, (s_d, t_d)\}$, **goto** *squarenot* 10 ⟩ Used in section 7.
⟨ Initialize for dancing 29 ⟩ Used in section 23.
⟨ Input row i of the shape 4 ⟩ Used in section 3.
⟨ Input the shape 3 ⟩ Used in section 1.
⟨ Local variables 27, 32, 38 ⟩ Used in section 23.
⟨ Make all remaining forced moves 18 ⟩ Used in section 12.
⟨ Make forced moves from the shape, or **goto** *done* 16 ⟩ Used in section 12.
⟨ Make forced moves from the square, or **goto** *done* 14 ⟩ Used in section 12.
⟨ Make shape cell l inactive 20 ⟩ Used in section 19.
⟨ Make square cell *ll* inactive 17 ⟩ Used in sections 16 and 21.
⟨ Print a solution 40 ⟩ Used in sections 23 and 39.
⟨ Print statistics about the run 41 ⟩ Used in section 1.
⟨ Process the command line 2 ⟩ Used in section 1.
⟨ Record solution and **goto** *recover* 39 ⟩ Used in section 31.
⟨ Remove all other edges that go to shape position l 15 ⟩ Used in section 14.
⟨ Run through all sequences of shifts, $(t_2, \ldots, t_d)$ 7 ⟩ Used in section 6.
⟨ Set *best_col* to the best column for branching 37 ⟩ Used in section 31.
⟨ Subroutines 33, 34, 42 ⟩ Used in section 1.
⟨ Type definitions 25, 26 ⟩ Used in section 1.
⟨ Uncover all other columns of *cur_node* 36 ⟩ Used in section 31.

BACK-DISSECT

	Section	Page
Intro	1	1
The algorithm	5	3
Prematching	12	7
Matching	23	11
Index	43	17