(See https://cs.stanford.edu/~knuth/programs.html for date.)

1. Intro. Given the specification of a fillomino puzzle in *stdin*, this program outputs DLX data for the problem of finding all solutions.

The specification consists of m lines of n entries each. An entry is either '`.`' or a digit from `1` to `9` or `a` to `f`.

A solution means that all '`.`' entries are replaced by digits. Every maximal rookwise connected set of cells labeled d must be a d-omino.

The maximum digit in the solution will be the maximum digit specified. (For example, the program will make no attempt to fit pentominoes into the blank cells, if all of the specified digits are less than `5`.)

N.B.: The assumption in the previous paragraph can be a serious deviation from the standard rules for fillomino. Use the change file FILLOMINO-DLX-LIMITS if you want fine control over which labels d are considered to be allowed in solutions.

The main interest in this program is its method for finding all feasible d-ominoes P that cover a given cell, when that cell has lexicographically smallest coordinates in that d-omino; P is infeasible if it includes a non-d label, or if it's adjacent to a d in a cell $\notin P$. The algorithm used here is an instructive generalization of Algorithm R in exercise 7.2.2–75 of *The Art of Computer Programming*.

```
#define maxn 16   /* at most 16 (or I'll have to go beyond hex) */
#define maxd 16   /* digits of the solution must be less than this */
#define bufsize 80
#define pack(i, j) ((((i) + 1) << 8) + (j) + 1)
#define unpack(ij) icoord = ((ij) >> 8) − 1, jcoord = ((ij) & #ff) − 1
#define board(i, j) brd[pack(i, j)]
#define panic(message)
        { fprintf(stderr, "%s:␣%s", message, buf); exit(−1); }
#include <stdio.h>
#include <stdlib.h>
  char buf[bufsize];
  int brd[pack(maxn, maxn)];   /* the given pattern */
  int dmax;   /* the maximum digit seen */
  ⟨Global data structures for Algorithm R 18⟩;
  ⟨Subroutines 19⟩;
  main()
  {
    register int a, d, i, j, k, l, m, n, p, q, s, t, u, v, di, dj, icoord, jcoord;
    printf("|␣fillomino-dlx:\n");
    ⟨Read the input into board 2⟩;
    ⟨Print the item-name line 3⟩;
    for (d = 1; d ≤ dmax; d++) ⟨Print all the options for d-ominoes 4⟩;
  }
```

2. ⟨ Read the input into *board* 2 ⟩ ≡

```
for (i = n = t = 0; i ≤ maxn; i++) {
  if (¬fgets(buf, bufsize, stdin)) break;
  printf("|␣%s", buf);
  for (j = k = 0; ; j++, k++) {
    if (buf[k] ≡ '\n') break;
    if (buf[k] ≡ '.') continue;
    if (buf[k] ≥ '1' ∧ buf[k] ≤ '9') board(i, j) = buf[k] − '0', t++;
    else if (buf[k] ≥ 'a' ∧ buf[k] ≤ 'f') board(i, j) = buf[k] − 'a' + 10, t++;
    else panic("illegal␣entry");
    if (board(i, j) > dmax) {
      if (board(i, j) ≥ maxd) {
        fprintf(stderr, "Sorry,␣all␣digits␣in␣the␣spec␣must␣be␣less␣than␣%d!\n", maxd);
        exit(−5);
      }
      dmax = board(i, j);
    }
  }
  if (j > n) n = j;    /* short rows are extended with '.'s */
}
if (i > maxn) panic("too␣many␣rows");
m = i;
for (i = 0; i < m; i++) board(i, −1) = board(i, n) = −1;    /* frame the board */
for (j = 0; j < n; j++) board(−1, j) = board(m, j) = −1;
fprintf(stderr, "OK,␣I've␣read␣%d␣clues␣<=␣%d,␣for␣a␣%dx%d␣board.\n", t, dmax, m, n);
mm = m, nn = n;
```

This code is used in section 1.

3. There are primary items ij for $0 \le i < m$ and $0 \le j < n$. They represent the cells to be filled.

There are secondary items vijd for each vertical boundary edge of a d-omino between $(i, j-1)$ and (i, j), for $0 \le i < m$ and $1 \le j < n$. Similarly, secondary items hijd for $1 \le i < m$ and $0 \le j < n$ are for horizontal boundary edges between $(i-1, j)$ and (i, j). These are needed only for edges between blank cells.

⟨ Print the item-name line 3 ⟩ ≡

```
for (i = 0; i < m; i++)
  for (j = 0; j < n; j++) printf("%x%x␣", i, j);
printf("|");
for (i = 0; i < m; i++)
  for (j = 1; j < n; j++)
    if (¬board(i, j) ∧ ¬board(i, j − 1))
      for (d = 1; d ≤ dmax; d++) printf("␣v%x%x%x", i, j, d);
for (i = 1; i < m; i++)
  for (j = 0; j < n; j++)
    if (¬board(i, j) ∧ ¬board(i − 1, j))
      for (d = 1; d ≤ dmax; d++) printf("␣h%x%x%x", i, j, d);
printf("\n");
```

This code is used in section 1.

4. ⟨Print all the options for d-ominoes 4⟩ ≡

```
{
  for (di = 0; di < m; di++)
    for (dj = 0; dj < n; dj++) ⟨Print the options for d-ominoes starting at (di, dj) 5⟩;
}
```

This code is used in section 1.

5. Now comes the interesting part. I assume the reader is familiar with Algorithm R in the solution to exercise 7.2.2–75. But we add a new twist: A *forced move* is made to a d-cell if we've chosen a vertex adjacent to it. The first vertex (v_0) is also considered to be forced.

Since I'm not operating with a general graph, the `ARCS` and `NEXT` aspects of Algorithm R are replaced with a simple scheme: Codes 1, 2, 3, 4 are used respectively for north, west, east, and south. In other words, the operation '$a \leftarrow$ `ARCS(`v`)`' is changed to '$a \leftarrow 1$'; '$a \leftarrow$ `NEXT(`a`)`' is changed to '$a \leftarrow a + 1$'; '$a = \Lambda$?' becomes '$a = 5$?'. The vertex `TIP(`a`)` is the cell north, west, east, or south of v, depending on a.

A forced move at level l is indicated by $a_l = 0$.

If cell (di, dj) is not already filled, we fill it with a d-omino that uses only unfilled cells and doesn't come next to a d-cell.

⟨Print the options for d-ominoes starting at (di, dj) 5⟩ ≡

```
{
  u = pack(di, dj);
  if (¬board(di, dj)) {
    for (q = 1; q ≤ 4; q++)
      if (brd[u + dir[q]] ≡ d) break;      /* next to d */
    if (q ≤ 4) continue;
    forcing = 0;
  } else if (board(di, dj) ≠ d) continue;
  else forcing = 1;
  ⟨Do step R1 6⟩;
  ⟨Do step R2 7⟩;
  ⟨Do step R3 11⟩;
  ⟨Do step R4 12⟩;
  ⟨Do step R5 13⟩;
  ⟨Do step R6 15⟩;
  ⟨Do step R7 17⟩;
done: checktags();
}
```

This code is used in section 4.

6. ⟨Do step R1 6⟩ ≡

```
r1:      /* initialize */
  for (i = 0; i < m; i++)
    for (j = 0; j < n; j++) tag[pack(i, j)] = 0;
  v = vv[0] = u, tag[v] = 1;
  i = ii[0] = 0, a = aa[0] = 0, l = 1;
```

This code is used in section 5.

7. At the beginning of step R2, we've just chosen the vertex u, which is $vv[l-1]$. If $l>1$, it's a vertex adjacent to $v=vv[i]$ in direction a, where $i=ii[l-1]$ and $a=aa[l-1]$.

⟨ Do step R2 7 ⟩ ≡
$r2$: /* enter level l */
  if (*forcing*) ⟨ Make forced choices of all d-cells adjacent to u; but **goto** $r7$ if there's a problem 8 ⟩;
  if ($l \equiv d$) {
    ⟨ Print an option for the current d-omino 9 ⟩;
    ⟨ Undo the latest forced moves 10 ⟩;
  }

This code is used in section 5.

8. Ye olde depth-first search.

If forcing, we backtrack if the d-omino gets too big, or if we're forced to choose a d-cell whose options have already been considered.

If not forcing, we backtrack if we're next to a d-cell, or if solutions for this cell have already been considered.

⟨ Make forced choices of all d-cells adjacent to u; but **goto** $r7$ if there's a problem 8 ⟩ ≡
  for ($stack[0]=u, s=1;\ s;\ $) {
    $u=stack[--s]$;
    for ($q=1;\ q\le 4;\ q$++) {
      $t=u+dir[q]$;
      if ($brd[t]\ne d$) **continue**; /* not a d-cell */
      if ($tag[t]$) **continue**; /* we've already chosen this d-cell */
      if ($t<vv[0]$) **goto** $r7$; /* it came earlier than (di,dj) */
      if ($l\equiv d$) **goto** $r7$; /* we've already got d vertices */
      $aa[l]=0, vv[l{+}{+}]=t, tag[t]=1, stack[s{+}{+}]=t$; /* forced move to t */
    }
  }

This code is used in section 7.

9. OK, we've got a viable d-omino to pass to the output.

⟨ Print an option for the current d-omino 9 ⟩ ≡

```
{
  curstamp++;
  for (p = 0; p < d; p++) {
    unpack(vv[p]);
    printf("␣%x%x", icoord, jcoord);
    stamp[vv[p]] = curstamp;
  }
  for (p = 0; p < d; p++) {
    unpack(vv[p]);
    if (¬board(icoord, jcoord))
      for (q = 1; q ≤ 4; q++)
        if (stamp[vv[p] + dir[q]] ≠ curstamp) {      /* boundary edge detected */
          switch (q) {
          case 1: if (icoord ∧ ¬board(icoord − 1, jcoord)) printf("␣h%x%x%x", icoord, jcoord, d); break;
          case 2: if (jcoord ∧ ¬board(icoord, jcoord − 1)) printf("␣v%x%x%x", icoord, jcoord, d); break;
          case 3: if (jcoord < n − 1 ∧ ¬board(icoord, jcoord + 1))
                printf("␣v%x%x%x", icoord, jcoord + 1, d); break;
          case 4: if (icoord < m − 1 ∧ ¬board(icoord + 1, jcoord))
                printf("␣h%x%x%x", icoord + 1, jcoord, d); break;
          }
        }
  }
  printf("\n");
}
```

This code is used in section 7.

10. ⟨ Undo the latest forced moves 10 ⟩ ≡

```
for (l−−; aa[l] ≡ 0; l−−) {
  if (l ≡ 0) goto done;
  tag[vv[l]] = 0;
}
```

This code is used in sections 7 and 17.

11. ⟨ Do step R3 11 ⟩ ≡

```
r3:     /* advance a */
  a++;
```

This code is used in section 5.

12. ⟨ Do step R4 12 ⟩ ≡

```
r4:     /* done with level? */
  if (a ≠ 5) goto r5;
  if (i ≡ l − 1) goto r6;
  v = vv[++i], a = 1;
```

This code is used in section 5.

13. ⟨ Do step R5 13 ⟩ ≡

```
r5:    /* try a */
  u = v + dir[a];
  if (brd[u]) goto r3;    /* not really a neighbor of v */
  tag[u]++;
  if (tag[u] > 1) goto r3;    /* already chosen */
  if (¬forcing) ⟨ If u was already handled, or if it's adjacent to a d-cell, goto r3 14 ⟩;
  ii[l] = i, aa[l] = a, vv[l] = u, l++;
  goto r2;
```

This code is used in section 5.

14. ⟨ If u was already handled, or if it's adjacent to a d-cell, **goto** $r3$ 14 ⟩ ≡

```
  {
    if (u < vv[0]) goto r3;    /* it's earlier than (di, dj) */
    for (q = 1; q ≤ 4; q++)
      if (brd[u + dir[q]] ≡ d) goto r3;
  }
```

This code is used in section 13.

15. ⟨ Do step R6 15 ⟩ ≡

```
r6:    /* backtrack */
  ⟨ Undo previous forced moves 16 ⟩;
  for (i = ii[l], k = i + 1; k ≤ l; k++) {
    t = vv[k];
    for (q = 1; q ≤ 4; q++)
      if (brd[t + dir[q]] ≡ 0) tag[t + dir[q]]−−;    /* untag the neighbors of vv[k] */
  }
  for (a = aa[l] + 1, v = vv[i]; a ≤ 4; a++)
    if (brd[v + dir[a]] ≡ 0) tag[v + dir[a]]−−;    /* untag late neighbors of vv[i] */
  a = aa[l];
  goto r3;
```

This code is used in section 5.

16. ⟨ Undo previous forced moves 16 ⟩ ≡

```
  for (l−−; aa[l] ≡ 0; l−−) {
    if (l ≡ 0) goto done;
    t = vv[l];
    for (q = 1; q ≤ 4; q++)
      if (brd[t + dir[q]] ≡ 0) tag[t + dir[q]]−−;    /* untag the neighbors of vv[l] */
    tag[t] = 0;
  }
```

This code is used in section 15.

17. ⟨ Do step R7 17 ⟩ ≡

```
r7:    /* recover from bad forcing */
  ⟨ Undo the latest forced moves 10 ⟩;
  i = ii[l], v = vv[i], a = aa[l];
  goto r3;
```

This code is used in section 5.

18. ⟨ Global data structures for Algorithm R 18 ⟩ ≡

```
int forcing;
int dir[5] = {0, -(1 << 8), -1, 1, 1 << 8};
int tag[pack(maxn, maxn)];
int vv[maxd], aa[maxd], ii[maxd], stack[maxd];    /* state variables */
int curstamp;
int stamp[pack(maxn, maxn)];
int mm, nn;
```

This code is used in section 1.

19. ⟨ Subroutines 19 ⟩ ≡

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

See also sections 20 and 21.

This code is used in section 1.

20. Here's a handy routine for debugging the tricky parts.

⟨ Subroutines 19 ⟩ +≡

```
void showtags(void)
{
  register int i, j;
  for (i = 0; i < mm; i++)
    for (j = 0; j < nn; j++)
      if (tag[pack(i, j)]) printf("%x%x:%d\n", i, j, tag[pack(i, j)]);
}
```

21. ⟨ Subroutines 19 ⟩ +≡

```
void checktags(void)
{
  register int i, j, q;
  for (i = 0; i < mm; i++)
    for (j = 0; j < nn; j++)
      if (tag[pack(i, j)]) {
        if (pack(i, j) == vv[0]) continue;
        for (q = 1; q <= 4; q++)
          if (pack(i, j) == vv[0] + dir[q]) break;
        if (q <= 4) continue;
        debug("bad tag");
      }
}
```

22. Index.

FILLOMINO-DLX