1. Intro. Given the specification of a masyu puzzle in *stdin*, this program outputs DLX data for the problem of finding all solutions. (I first hacked it from SLITHERLINK-DLX; but a day later, I realized that a far more efficient solution was possible, because masyu clues force many of the secondary items to be identical or complementary.)

No attempt is made to enforce the "single loop" condition. Solutions found by DLX2 will consist of disjoint loops. But DLX2-LOOP will weed out any disconnected solutions; in fact it will nip most of them in the bud.

The specification begins with m lines of n characters each; those characters should be either '0' or '1' or '.', representing either a white circle, a black circle, or no clue.

```
#define maxn 31   /* m and n must be at most this */
#define bufsize 80
#define debug #0   /* verbose printing? (each bit turns on a different case) */
#define panic(message)
        { fprintf(stderr, "%s:␣%s", message, buf); exit(−1); }
#include <stdio.h>
#include <stdlib.h>
  ⟨Type definitions 3⟩;
  char buf[bufsize];
  int board[maxn][maxn];   /* the given clues */
  int m, n;   /* the given board dimensions */
  edge ejj[((maxn + maxn) ≪ 8) + (maxn + maxn)];   /* the union-find table */
  char code[ ] = {#c, #a, #5, #3, #9, #6, #0};
  int opt[4], optval[4];
  int optcount;
  ⟨Subroutines 5⟩;
  main( )
  {
    register int i, j, k, ii, jj, v, t, e, p;
    ⟨Read the input into board 2⟩;
    ⟨Reduce the secondary items to independent variables 9⟩;
    ⟨Print the item-name line 11⟩;
    ⟨Print options to make DLX2-LOOP happy 17⟩;
    for (i = 0; i < m; i++)
      for (j = 0; j < n; j++)
        if (board[i][j] ≡ '.') ⟨Print the options for tile (i, j) 19⟩
        else ⟨Print the options for circle (i, j) 20⟩;
  }
```

2. ⟨Read the input into *board* 2⟩ ≡

```
printf("|␣masyu-dlx:\n");
for (i = 0; i < maxn; i++) {
  if (¬fgets(buf, bufsize, stdin)) break;
  printf("|␣%s", buf);
  for (j = 0; j < maxn ∧ buf[j] ≠ '\n'; j++) {
    k = buf[j];
    if (k ≠ '.' ∧ (k < '0' ∨ k > '4')) panic("illegal␣clue");
    board[i][j] = k;
  }
  if (i ≡ 0) n = j;
  else if (n ≠ j) panic("row␣has␣wrong␣number␣of␣clues");
}
m = i;
if (m < 2 ∨ n < 2) panic("the␣board␣dimensions␣must␣be␣2␣or␣more");
fprintf(stderr, "OK,␣I've␣read␣a␣%dx%d␣array␣of␣clues.\n", m, n);
```

This code is used in section 1.

3. Big reductions. The original version of this program had one secondary item for each edge of the grid. These secondary items essentially acted as boolean variables, with their "colors" `0` and `1`.

Let the edges surrounding a clue be called N, S, E, and W. A black clue tells us, among other things, that $N = \bar{S}$ and $E = \bar{W}$. So it reduces the number of independent variables by two. A white clue does even more: It tells us, among other things, that $N = S$ and $E = W$ and $N = \bar{E}$.

Internally, we represent the edge between cell (i, j) and cell $(i', j)'$ by the packed value $256(i+i')+(j+j')$. A standard union-find algorithm is used to determine which edges are dependent, and to provide a canonical form for any edge in terms of an independent basis.

Edges off the board have the constant value `0`. We often can deduce a constant value for other edges. The "constant" edge is denoted internally by zero.

```
#define N(v) ((v) − (1 ≪ 8))
#define S(v) ((v) + (1 ≪ 8))
#define E(v) ((v) + 1)
#define W(v) ((v) − 1)
```

```
⟨Type definitions 3⟩ ≡
  typedef struct {
    int ldr;   /* the representative of this equivalence class */
    int cmp;   /* are we the same as the leader, or the same as its complement? */
    int nxt;   /* next member of this class, in a cyclic list */
    int siz;   /* the size of this class (if we're the leader) */
  } edge;
```

This code is used in section 1.

4. ⟨Initialize all edges to independent 4⟩ ≡

```
  for (ii = 0; ii ≤ m + m − 2; ii++)
    for (jj = 0; jj ≤ n + n − 2; jj++)
      if ((ii + jj) & 1) {
        e = (ii ≪ 8) + jj;
        ejj[e].ldr = ejj[e].nxt = e, ejj[e].siz = 1;
      }
```

This code is used in section 9.

5. ⟨Subroutines 5⟩ ≡

```
  int normalize(int e)
  {
    if (e < 0) return 0;   /* i negative */
    if ((e & #ff) > n + n − 2) return 0;   /* j negative or too large */
    if ((e ≫ 8) > m + m − 2) return 0;   /* i too large */
    return e;   /* this edge not obviously constant */
  }
```

See also sections 6, 12, 13, and 16.

This code is used in section 1.

6. ⟨ Subroutines 5 ⟩ +≡

```
int yewnion(int e, int ee, int comp)
{
  register int p, q, pp, qq, s, t;
  e = normalize(e), ee = normalize(ee);
  if (debug & 1) fprintf(stderr, "␣%c%c␣is␣%s%c%c\n", encode(e ≫ 8), encode(e & #ff), comp ? "~" : "",
        encode(ee ≫ 8), encode(ee & #ff));
  p = ejj[e].ldr, s = comp ⊕ ejj[e].cmp;
  pp = ejj[ee].ldr, s ⊕= ejj[ee].cmp;    /* now we want to set p to pp ⊕ s */
  if (p ≡ pp) ⟨ Check for consistency and exit 8 ⟩;
  if (p ≡ 0 ∨ (pp ≠ 0 ∧ ejj[p].siz > ejj[pp].siz)) t = p, p = pp, pp = t, t = e, e = ee, ee = t;
  ⟨ Merge classes p and pp 7 ⟩;
  return 0;    /* "no problem" */
}
```

7. ⟨ Merge classes p and pp 7 ⟩ ≡

```
ejj[pp].siz += ejj[p].siz;
if (debug & 2)
  fprintf(stderr, "␣(size␣of␣%c%c␣now␣%d)\n", encode(pp ≫ 8), encode(pp & #ff), ejj[pp].siz);
for (q = ejj[p].nxt; ; q = ejj[q].nxt) {
  ejj[q].ldr = pp, ejj[q].cmp ⊕= s;
  if (q ≡ p) break;
}
t = ejj[p].nxt, ejj[p].nxt = ejj[pp].nxt, ejj[pp].nxt = t;
```

This code is used in section 6.

8. ⟨ Check for consistency and exit 8 ⟩ ≡

```
{
  if (s ≡ 0) return 0;    /* the new relation is consistent (and redundant) */
  fprintf(stderr, "Inconsistency␣found␣when␣equating␣%c%c␣to␣%s%c%c!\n", encode(e ≫ 8),
      encode(e & #ff), comp ? "~" : "", encode(ee ≫ 8), encode(ee & #ff));
  return 1;    /* "one problem" */
}
```

This code is used in section 6.

9. ⟨ Reduce the secondary items to independent variables 9 ⟩ ≡

```
⟨ Initialize all edges to independent 4 ⟩;
for (i = 0; i < m; i++)
  for (j = 0; j < n; j++)
    if (board[i][j] ≠ '.') {
      v = ((i + i) ≪ 8) + (j + j);
      if (board[i][j] ≡ '1') t = yewnion(N(v), S(v), 1) + yewnion(E(v), W(v), 1);
      else t = yewnion(N(v), S(v), 0) + yewnion(E(v), W(v), 0) + yewnion(S(v), W(v), 1);
      if (t) {
        printf("abort\n");    /* abort with an unsolvable problem */
        exit(0);
      }
    }
```

This code is used in section 1.

10.

11. Items. The primary items are "tiles" and "circles." Tiles control the loop path; the name of tile (i,j) is $(2i,2j)$. Circles represent the clues; the name of circle (i,j) is $(2i+1,2j+1)$. A circle item is present only if a clue has been given for the corresponding tile of the board.

The secondary items are "edges" of the path. Their names are the midpoints of the tiles they connect.

We don't really need a tile item when a clue has been given; the tile constraints have been guaranteed by our reduction process. However, DLX2-LOOP requires a tile item for every vertex, as an essential part of its data structures! So we create "dummy" tile items, and put them into a special option, to make that program happy.

We don't really need an edge item unless it's the root of its union-find tree. But once again we must pander to the whims of DLX2-LOOP. So we create special primary items #e, for each equivalence class of size 2 or more.

```
#define encode(x) ((x) < 10 ? (x) + '0' : (x) < 36 ? (x) − 10 + 'a' : (x) < 62 ? (x) − 36 + 'A' : '?')

⟨ Print the item-name line 11 ⟩ ≡
  for (i = 0; i < m; i++)
    for (j = 0; j < n; j++) printf("%c%c␣", encode(i + i), encode(j + j));
  for (i = 0; i < m; i++)
    for (j = 0; j < n; j++)
      if (board[i][j] ≠ '.') printf("%c%c␣", encode(i + i + 1), encode(j + j + 1));
  for (ii = 0; ii ≤ m + m − 2; ii++)
    for (jj = 0; jj ≤ n + n − 2; jj++)
      if ((ii + jj) & 1) {
        e = (ii ≪ 8) + jj;
        if (ejj[e].ldr ≡ e ∧ ejj[e].siz > 1) printf("#%c%c␣", encode(ii), encode(jj));
      }
  printf("|");
  for (i = 0; i < m + m − 1; i++)
    for (j = 0; j < n + n − 1; j++)
      if ((i + j) & 1) printf("␣%c%c", encode(i), encode(j));
  printf("\n");
```

This code is used in section 1.

12. Options. Each option to be output is either of special one (designed to make DLX2-LOOP happy) or consists of a primary item together with constraints on four edges. Those edges might be dependent, or off the board, in which case fewer than four constraints are involved.

In fact, all four edges might turn out to be constant, in which case the option is always true or always false. An option might also turn out to be just plain foolish, if it tries to make some boolean variable both true and false. Such options, and the always-false ones, should not be output.

Therefore we gather the constraints one by one, before outputting any option. Pending constraints are accumulated in a sorted array called *opt*.

⟨Subroutines 5⟩ +≡

```
void begin_opt(int ii, int jj)
{
  if (debug & 4) fprintf(stderr, "␣beginning␣an␣option␣for␣%c%c:\n", encode(ii), encode(jj));
  optcount = 0;
}
```

13. ⟨Subroutines 5⟩ +≡

```
void append_opt(int e, int val)
{
  register int q, p, s, t;
  if (optcount < 0) return;     /* option has already been cancelled */
  e = normalize(e), val = val & 1;
  if (debug & 4) fprintf(stderr, "␣␣%c%c:%d\n", encode(e >> 8), encode(e & #ff), val);
  p = ejj[e].ldr, val ⊕= ejj[e].cmp;
  if (p ≡ 0) ⟨Handle a constant constraint and exit 14⟩;
  for (t = 0; t < optcount; t++)
    if (opt[t] ≥ p) break;
  if (t < optcount ∧ opt[t] ≡ p) ⟨Handle a matching constraint and exit 15⟩;
  for (s = optcount++; s > t; s--) opt[s] = opt[s - 1], optval[s] = optval[s - 1];
  opt[s] = p, optval[s] = val;
  if (debug & 8) fprintf(stderr, "␣␣(%c%c:%d)\n", encode(p >> 8), encode(p & #ff), val);
  return;
}
```

14. The constant 0 is false.

⟨Handle a constant constraint and exit 14⟩ ≡

```
{
  if (val ≡ 1) {
    if (debug & 8) fprintf(stderr, "␣␣(false)\n");
    optcount = −1;
  } else if (debug & 8) fprintf(stderr, "␣␣(true)\n");
  return;
}
```

This code is used in section 13.

15. ⟨ Handle a matching constraint and exit 15 ⟩ ≡

```
{
  if (val ≠ optval[t]) {
    if (debug & 8) fprintf(stderr, "␣␣(%c%c:%d!)\n", encode(p ≫ 8), encode(p & #ff), val);
    optcount = −1;
  }
  if (debug & 8) fprintf(stderr, "␣␣(%c%c:%d)\n", encode(p ≫ 8), encode(p & #ff), val);
  return;
}
```

This code is used in section 13.

16. ⟨ Subroutines 5 ⟩ +≡

```
void finish_opt(int ii, int jj)
{
  register int t;
  if (optcount ≥ 0) {
    printf("%c%c", encode(ii), encode(jj));
    for (t = 0; t < optcount; t++)
      printf("␣%c%c:%d", encode(opt[t] ≫ 8), encode(opt[t] & #ff), optval[t]);
    printf("\n");
  }
}
```

17. Generating the special options. Just after making the item-name line, we generate a catchall option that names all tiles for which a clue has been given, as well as all edges whose boolean value is known in advance. This option will get DLX2-LOOP off to a good start.

⟨ Print options to make DLX2-LOOP happy 17 ⟩ ≡

```
for (i = 0; i < m; i++)
  for (j = 0; j < n; j++)
    if (board[i][j] ≠ '.') printf("%c%c␣", encode(i + i), encode(j + j));
for (p = ejj[0].nxt; p; p = ejj[p].nxt) printf("%c%c:%d␣", encode(p ≫ 8), encode(p & #ff), ejj[p].cmp);
printf("\n");
```

See also section 18.

This code is used in section 1.

18. ⟨ Print options to make DLX2-LOOP happy 17 ⟩ +≡

```
for (ii = 0; ii ≤ m + m − 2; ii++)
  for (jj = 0; jj ≤ n + n − 2; jj++)
    if ((ii + jj) & 1) {
      e = (ii ≪ 8) + jj;
      if (ejj[e].ldr ≡ e ∧ ejj[e].siz > 1) {
        printf("#%c%c", encode(ii), encode(jj));
        for (p = ejj[e].nxt; ; p = ejj[p].nxt) {
          printf("␣%c%c:%d", encode(p ≫ 8), encode(p & #ff), ejj[p].cmp);
          if (p ≡ e) break;
        }
        printf("\n");
        printf("#%c%c", encode(ii), encode(jj));
        for (p = ejj[e].nxt; ; p = ejj[p].nxt) {
          printf("␣%c%c:%d", encode(p ≫ 8), encode(p & #ff), ejj[p].cmp ⊕ 1);
          if (p ≡ e) break;
        }
        printf("\n");
      }
    }
```

19. Generating the normal options. The four constraints for a tile say that the N, S, E, W neighbors of (i,j) include exactly 0 or 2 true edges.

⟨ Print the options for tile (i,j) 19 ⟩ ≡

```
{
  e = ((i + i) ≪ 8) + (j + j);
  for (k = 0; k < 7; k++) {
    begin_opt(i + i, j + j);
    append_opt(N(e), code[k] ≫ 3);
    append_opt(W(e), code[k] ≫ 2);
    append_opt(E(e), code[k] ≫ 1);
    append_opt(S(e), code[k]);
    finish_opt(i + i, j + j);
  }
}
```

This code is used in section 1.

20. ⟨ Print the options for circle (i,j) 20 ⟩ ≡

```
{
  e = ((i + i) ≪ 8) + (j + j);
  if (board[i][j] ≡ '1') ⟨ Print the options for a black circle at (i, j) 21 ⟩
  else ⟨ Print the options for a white circle at (i, j) 22 ⟩;
}
```

This code is used in section 1.

21. The four constraints for circles look further, at neighbors that are two steps away.

```
#define NN(v) ((v) − (3 ≪ 8))
#define SS(v) ((v) + (3 ≪ 8))
#define EE(v) ((v) + 3)
#define WW(v) ((v) − 3)
```

⟨ Print the options for a black circle at (i,j) 21 ⟩ ≡

```
{
  for (k = 0; k < 4; k++) {
    begin_opt(i + i + 1, j + j + 1);
    if (code[k] & 8) append_opt(N(e), 1), append_opt(NN(e), 1);
    if (code[k] & 4) append_opt(W(e), 1), append_opt(WW(e), 1);
    if (code[k] & 2) append_opt(E(e), 1), append_opt(EE(e), 1);
    if (code[k] & 1) append_opt(S(e), 1), append_opt(SS(e), 1);
    finish_opt(i + i + 1, j + j + 1);
  }
}
```

This code is used in section 20.

22. ⟨Print the options for a white circle at (i, j) 22⟩ ≡

```
{
  for (k = 4; k < 6; k++)
    for (ii = 0; ii < 2; ii++)
      for (jj = 0; jj < 2; jj++)
        if (ii * jj ≡ 0) {
          begin_opt(i + i + 1, j + j + 1);
          if (code[k] & 8) {
            append_opt(N(e), 1);
            append_opt(NN(e), ii), append_opt(SS(e), jj);
          } else {
            append_opt(W(e), 1);
            append_opt(WW(e), ii), append_opt(EE(e), jj);
          }
          finish_opt(i + i + 1, j + j + 1);
        }
}
```

This code is used in section 20.

23. Index.

MASYU-DLX