(Downloaded from https://cs.stanford.edu/~knuth/programs.html and typeset on September 17, 2017)

1. Intro. Read TICTACTOE6 first. This program is similar to that one, but its job is more complicated because it handles the corner outputs $y_1y_3y_9y_7$ instead of the center output y_5.

We now keep four sets of 2^{12} columns, indexed by $x_2x_6x_8x_4o_1o_3o_9o_7x_1x_3x_9x_7d_1d_0$, where d_1d_0 are direction codes. These directions 00, 01, 10, and 11 essentially mean that the final value of y_1 will come from the current value of y_1, y_3, y_9, or y_7, respectively, after eight units have been hooked together appropriately.

The basic algorithms remain the same, but there is now a 2^{20}-bit address space instead of 2^{18}.

As before, the standard input file should be the output of TICTACTOE4.

```
#define nn (1 ≪ 14)
#define nnn (1 ≪ 20)
#define cases (4520 * 4)
#define head (&col[nn])

#include <stdio.h>
  typedef struct col_struct {
    int ah, al;  /* 64 asterisk specs, for don't cares */
    int bh, bl;  /* 64 bit specs, for cares */
    int c, m;  /* the number of cares and the number of 1s */
    struct col_struct *prev, *next;  /* links for priority list */
  } column;
  ⟨Global variables 3⟩
  int count;  /* counter for miscellaneous purposes */
  ⟨Subroutines 20⟩
  main()
  {
    register int i, j, k, l, t;
    register column *p, *q;
    ⟨Set up the initial columns 2⟩;
    ⟨Do phase 1 5⟩;
    ⟨Do phase 2 13⟩;
    ⟨Do phase 3 19⟩;
    ⟨Check the results 28⟩;
  }
```

2. **#define** *bit*(*j*, *k*) $(((bits \gg j) \,\&\, 1) \ll k)$

⟨ Set up the initial columns 2 ⟩ ≡

```
for (j = 0; j < nn; j++) col[j].ah = col[j].al = #ffffffff,
      col[j].next = &col[j + 1], col[j + 1].prev = &col[j];
head→next = &col[0], col[0].prev = head;
head→c = 99;    /* infinity */
k = 0;
while (1) {
  if (¬fgets(buf, 100, stdin)) break;
  if (buf[5] ≠ ':' ∨ sscanf(buf, "%x", &bits) ≠ 1) break;
  care[k] = (bit(16, 17) + bit(17, 16) + bit(6, 15) + bit(4, 14) + bit(2, 13) + bit(0, 12) +
      bit(7, 11) + bit(5, 10) + bit(3, 9) + bit(1, 8) +
      bit(14, 7) + bit(12, 6) + bit(10, 5) + bit(8, 4) +
      bit(15, 3) + bit(13, 2) + bit(11, 1) + bit(9, 0)) ≪ 2;
  yval[k] = buf[7] − '0';    /* this is y1 */
  if (yval[k] < 0 ∨ yval[k] > 1) {
    fprintf(stderr, "invalid input line! %s", buf);
    exit(−1);
  }
  ⟨ Make care[k] a "care" bit 4 ⟩;
  k++;
  care[k] = care[k − 1] + 1;
  yval[k] = buf[9] − '0';    /* this is y3 */
  if (yval[k] < 0 ∨ yval[k] > 1) {
    fprintf(stderr, "invalid input line! %s", buf);
    exit(−1);
  }
  ⟨ Make care[k] a "care" bit 4 ⟩;
  k++;
  care[k] = care[k − 1] + 1;
  yval[k] = buf[15] − '0';    /* this is y9 */
  if (yval[k] < 0 ∨ yval[k] > 1) {
    fprintf(stderr, "invalid input line! %s", buf);
    exit(−1);
  }
  ⟨ Make care[k] a "care" bit 4 ⟩;
  k++;
  care[k] = care[k − 1] + 1;
  yval[k] = buf[13] − '0';    /* this is y7 */
  if (yval[k] < 0 ∨ yval[k] > 1) {
    fprintf(stderr, "invalid input line! %s", buf);
    exit(−1);
  }
  ⟨ Make care[k] a "care" bit 4 ⟩;
  k++;
}
if (k ≠ cases) {
  fprintf(stderr, "There were %d cases, not %d!\n", k, cases);
  exit(−2);
}
```

This code is used in section 1.

3. ⟨ Global variables 3 ⟩ ≡

```
column col[nn + 1];
int care[cases], yval[cases];
char buf[100];
int bits;
```

See also sections 7, 14, 18, and 31.

This code is used in section 1.

4. A brute-force sorting method will keep the columns in lexicographic order by their (c, m) fields. Initially the m fields are all zero, so we don't worry about them here.

⟨ Make $care[k]$ a "care" bit 4 ⟩ ≡

```
{
  j = care[k] >> 14;
  t = care[k] & #3fff;
  if (j & #20) col[t].ah -= 1 << (j - #20);
  else col[t].al -= 1 << j;
  j = ++col[t].c;
  p = col[t].prev, q = col[t].next;
  p->next = q, q->prev = p;
  for ( ; j > q->c; q = q->next) ;
  p = q->prev;
  p->next = q->prev = &col[t], col[t].prev = p, col[t].next = q;
}
```

This code is used in section 2.

5. Phase 1.

⟨ Do phase 1 5 ⟩ ≡
```
⟨ Find the equivalence classes 6 ⟩;
for (k = 0; k < cases; k++)
  if (yval[k]) {
    t = care[k];
    if (leader[t] ≠ t) continue;
    while (t & 3) t = link[t];
    for (j = t, t = link[t]; t ≠ care[k]; t = link[t])
      if ((t & 3) ≡ 0 ∧ (col[t & #3fff].c ≤ col[j & #3fff].c ∧ (col[t & #3fff].c <
            col[j & #3fff].c ∨ col[t & #3fff].m < col[j & #3fff].m))) j = t;
    ⟨ Set care bit j to 1 10 ⟩;
    for (t = j, j = link[j]; j ≠ t; j = link[j]) ⟨ Change bit j to don't-care 11 ⟩;
  }
⟨ Print the results of Phase 1 12 ⟩;
```
This code is used in section 1.

6. Here I use the simplest union-find algorithm, without bells or whistles. (Years ago I never would have imagined being so shamelessly wasteful of computer time and space as I am today.)

⟨ Find the equivalence classes 6 ⟩ ≡
```
for (k = 0; k < cases; k++) dir[care[k]] = k, leader[care[k]] = link[care[k]] = care[k];
for (k = 0; k < cases; k++) ⟨ Make care[k] equivalent to its rotation 8 ⟩;
for (k = 0; k < cases; k++) ⟨ Make care[k] equivalent to its reflection 9 ⟩;
```
This code is used in section 5.

7. ⟨ Global variables 3 ⟩ +≡
```
int dir[nnn], link[nnn], leader[nnn];   /* equivalence class structures */
```

8. ⟨ Make care[k] equivalent to its rotation 8 ⟩ ≡
```
{
  t = care[k];
  j = t ⊕ ((t ⊕ (t ≫ 1)) & #1dddc) ⊕ ((t ⊕ (t ≪ 3)) & #22220) ⊕ 1 ⊕ ((t & 1) ≪ 1);
  if (yval[dir[j]] ≠ yval[k]) {
    fprintf(stderr, "Error:␣y[%05x]=%d,␣y[%05x]=%d!\n", t, yval[k], j, yval[dir[j]]);
    exit(−3);
  }
  if (leader[j] ≠ leader[t]) {
    do leader[j] = leader[t], j = link[j]; while (leader[j] ≠ leader[t]);
    l = link[j], link[j] = link[t], link[t] = l;
  }
}
```
This code is used in section 6.

9. $\langle$ Make *care*[k] equivalent to its reflection 9 $\rangle \equiv$

$$
\begin{array}{l}
\{\\
\quad t = care[k];\\
\quad j = t \oplus ((t \oplus (t \gg 1))\ \&\ \#\mathtt{154}) \oplus ((t \oplus (t \ll 1))\ \&\ \#\mathtt{2a8}) \oplus\\
\qquad ((t \oplus (t \gg 2))\ \&\ \#\mathtt{4400}) \oplus ((t \oplus (t \ll 2))\ \&\ \#\mathtt{11000}) \oplus 1;\\
\quad \mathbf{if}\ (yval[dir[j]] \neq yval[k])\ \{\\
\qquad fprintf(stderr, \texttt{"Error:␣y[\%05x]=\%d,␣y[\%05x]=\%d!\textbackslash n"}, t, yval[k], j, yval[dir[j]]);\\
\qquad exit(-4);\\
\quad \}\\
\quad \mathbf{if}\ (leader[j] \neq leader[t])\ \{\\
\qquad \mathbf{do}\ \ leader[j] = leader[t], j = link[j];\ \ \mathbf{while}\ (leader[j] \neq leader[t]);\\
\qquad l = link[j], link[j] = link[t], link[t] = l;\\
\quad \}\\
\}
\end{array}
$$

This code is used in section 6.

10. $\langle$ Set care bit j to 1 10 $\rangle \equiv$

$$
\begin{array}{l}
l = j \gg 14, i = j\ \&\ \#\mathtt{3fff};\\
\mathbf{if}\ (l\ \&\ \#\mathtt{20})\ \ col[i].bh \mathrel{+}= 1 \ll (l - \#\mathtt{20});\\
\mathbf{else}\ \ col[i].bl \mathrel{+}= 1 \ll l;\\
col[i].m{+}{+};\\
p = col[i].prev, q = col[i].next, p\to next = q, q\to prev = p;\\
\mathbf{for}\ (l = col[i].c;\ l \equiv q\to c \land col[i].m > q\to m;\ q = q\to next)\ ;\\
p = q\to prev;\\
p\to next = q\to prev = \&col[i], col[i].prev = p, col[i].next = q;
\end{array}
$$

This code is used in section 5.

11. $\langle$ Change bit j to don't-care 11 $\rangle \equiv$

$$
\begin{array}{l}
\{\\
\quad l = j \gg 14, i = j\ \&\ \#\mathtt{3fff};\\
\quad \mathbf{if}\ (l\ \&\ \#\mathtt{20})\ \ col[i].ah \mathrel{+}= 1 \ll (l - \#\mathtt{20});\\
\quad \mathbf{else}\ \ col[i].al \mathrel{+}= 1 \ll l;\\
\quad col[i].c{-}{-};\\
\quad p = col[i].prev, q = col[i].next, p\to next = q, q\to prev = p;\\
\quad \mathbf{for}\ (l = col[i].c;\ l < p\to c \lor (l \equiv p\to c \land col[i].m < p\to m);\ p = p\to prev)\ ;\\
\quad q = p\to next;\\
\quad p\to next = q\to prev = \&col[i], col[i].prev = p, col[i].next = q;\\
\}
\end{array}
$$

This code is used in section 5.

12. $\langle$ Print the results of Phase 1 12 $\rangle \equiv$

$$
\begin{array}{l}
\mathbf{for}\ (p = head\to prev;\ p\to c > 1;\ p = p\to prev)\\
\quad \mathbf{if}\ (p\to m)\ \ printf(\texttt{"\%03x:␣\%08x\%08x,\%08x\%08x␣(\%d,\%d)\textbackslash n"}, p - col, p\to ah, p\to al, p\to bh, p\to bl, p\to c, p\to m);\\
\mathbf{for}\ (count = 0, p = head\to prev;\ p \neq head;\ p = p\to prev)\\
\quad \mathbf{if}\ (p\to m)\ \ count{+}{+};\\
printf(\texttt{"\%d␣columns␣contain␣1s\textbackslash n"}, count);
\end{array}
$$

This code is used in section 5.

13. Phase 2.

```
⟨ Do phase 2 13 ⟩ ≡
  ⟨ Compute the pop table for population counts 15 ⟩;
  vec[0].ah = vec[0].al = 0, vec[0].bh = vec[0].bl = #ffffffff;   /* all 1s vector */
  p = head→prev;   /* the next column to be covered */
newvec: v++;
  vec[v].ah = vec[v].al = #ffffffff, vec[v].bh = vec[v].bl = 0;   /* all *s vector */
coverit: l = 0;
  if (p→m ≡ 0) goto advancep;
  for (k = 0, count = #990099; k ≤ v; k++) {
    ⟨ If p is incompatible with vec[k], continue 16 ⟩;
    t = p→bh & ~vec[k].bh, j = pop[((unsigned int) t) ≫ 16] + pop[t & #ffff];   /* count new 1s */
    t = p→bl & ~vec[k].bl, j += pop[((unsigned int) t) ≫ 16] + pop[t & #ffff], j ≪= 16;
    t = vec[k].ah & ~p→ah, j += pop[((unsigned int) t) ≫ 16] + pop[t & #ffff];   /* count lost *s */
    t = vec[k].al & ~p→al, j += pop[((unsigned int) t) ≫ 16] + pop[t & #ffff];
    if (j < count) count = j, l = k;
  }
  ⟨ Cover column p with vector l 17 ⟩;
advancep: p = p→prev;
  if (p→c) {
    if (l ≡ v) goto newvec;
    goto coverit;
  }
  printf("there␣are␣%d␣covering␣vectors\n", v);
```

This code is used in section 1.

14. **#define** *vecs* 500

```
⟨ Global variables 3 ⟩ +≡
  int pop[1 ≪ 16];   /* table of 16-bit population counts */
  column vec[vecs];   /* covering vectors */
  int v;   /* the current number of vectors */
```

15.

```
⟨ Compute the pop table for population counts 15 ⟩ ≡
  for (k = 1; k < #10000; k += k)
    for (j = 0; j < k; j++) pop[k + j] = 1 + pop[j];
```

This code is used in section 13.

16.

```
⟨ If p is incompatible with vec[k], continue 16 ⟩ ≡
  t = p→bl ⊕ vec[k].bl;
  if (t & (~vec[k].al) & (~p→al)) continue;
  t = p→bh ⊕ vec[k].bh;
  if (t & (~vec[k].ah) & (~p→ah)) continue;
```

This code is used in section 13.

17. The *next* field is now changed to point to the covering vector.

```
⟨ Cover column p with vector l 17 ⟩ ≡
  q = &vec[l];
  p→next = q;
  q→bl |= p→bl, q→bh |= p→bh;
  q→al &= p→al, q→ah &= p→ah;
  printf("cover␣%03x:%08x%08x,%08x%08x␣with␣%d:%08x%08x,%08x%08x\n", p − col, p→ah, p→al, p→bh,
      p→bl, l, q→ah, q→al, q→bh, q→bl);
```

This code is used in section 13.

18. Phase 3. Finally, we construct the Boolean chain by using the methods of TICTACTOE3.

#define *gates* 1000

⟨ Global variables 3 ⟩ +≡
```
typedef enum {
  inp, and, or, xor, butnot, nor
} opcode;
char *opcode_name[] = {"input","&","|","^",">","$"};
opcode op[gates];
char val[gates];
int jx[gates], kx[gates], p[gates];
char name[gates][8];
int x[10], o[10], vx[20], uu[vecs], vv[vecs], colgate[1 << 12];   /* addresses of named gates */
int colgate[1 << 12];   /* addresses of columns, if generated */
int g;   /* address of the most recently generated gate */
int rowbase, colbase0, colbase1;   /* addresses of key gates */
```

19. **#define** *makegate*(l,o,r) $op[{+}{+}g]=o, jx[g]=l, kx[g]=r$
#define *make0*(l,o,r) *makegate*(l,o,r), $name[g][0]=$ '\0'
#define *make1*(s,j,l,o,r,v) *makegate*(l,o,r), *sprintf*$(name[g],s,j), v=g$
#define *make2*(s,j,k,l,o,r,v) *makegate*(l,o,r), *sprintf*$(name[g],s,j,k), v=g$

⟨ Do phase 3 19 ⟩ ≡
```
for (j = 1; j <= 9; j++) make1("x%d", j, 0, inp, 0, x[j]);
for (j = 1; j <= 9; j++) make1("o%d", j, 0, inp, 0, o[j]);
vx[0] = o[5], vx[1] = x[5];
vx[2] = o[1], vx[3] = o[3], vx[4] = o[9], vx[5] = o[7];
vx[6] = x[1], vx[7] = x[3], vx[8] = x[9], vx[9] = x[7];
vx[10] = o[2], vx[11] = o[6], vx[12] = o[8], vx[13] = o[4];
vx[14] = x[2], vx[15] = x[6], vx[16] = x[8], vx[17] = x[4];
⟨ Make minterms for the rows 22 ⟩;
⟨ Make minterms for the columns 23 ⟩;
⟨ Make the row selector functions 24 ⟩;
⟨ Make the column selector functions 25 ⟩;
⟨ Combine the selector functions to make the output 26 ⟩;
```
This code is used in section 1.

20. Here's a simple recursive subroutine that makes minterms for $n > 1$ variables $v[0]$, ..., $v[n-1]$. The minterms appear in gates o, $o+1$, ..., $o+2^n-1$, where o is the output.

⟨Subroutines 20⟩ ≡

```
int makemins(int *v, int n)
{
  register int j, k, g0, g1, fn, cn;
  if (n < 4) ⟨Handle the base cases 21⟩
  else {
    fn = n/2, cn = n - fn;
    g0 = makemins(v, cn);
    g1 = makemins(v + cn, fn);
    for (j = 0; j < 1 << cn; j++)
      for (k = 0; k < 1 << fn; k++) make0(g0 + j, and, g1 + k);
  }
  return g - (1 << n) + 1;
}
```

This code is used in section 1.

21. ⟨Handle the base cases 21⟩ ≡

```
{
  make0(v[0], nor, v[1]);
  make0(v[1], butnot, v[0]);
  make0(v[0], butnot, v[1]);
  make0(v[0], and, v[1]);
  if (n > 2) {
    g0 = g - 3;
    for (j = 0; j < 4; j++) {
      make0(g0 + j, butnot, v[2]);
      make0(g0 + j, and, v[2]);
    }
  }
}
```

This code is used in section 20.

22. ⟨Make minterms for the rows 22⟩ ≡

```
rowbase = makemins(vx, 6);
for (j = 0; j < 64; j++) sprintf(name[rowbase + j], "r%x", j);
```

This code is used in section 19.

23. Most of the 2^{12} columns will not be used. So we only make base addresses from which full minterms can be generated as needed.

⟨Make minterms for the columns 23⟩ ≡

```
colbase0 = makemins(vx + 6, 6);
colbase1 = makemins(vx + 12, 6);
```

This code is used in section 19.

24. ⟨ Make the row selector functions 24 ⟩ ≡

```
for (j = 1; j < v; j++) {
  for (k = 0; k < 32; k++)
    if (vec[j].bl & (1 << k)) {
      if (!vv[j]) vv[j] = rowbase + k;
      else {
        make0(vv[j], or, rowbase + k);
        vv[j] = g;
      }
    }
  for (k = 0; k < 32; k++)
    if (vec[j].bh & (1 << k)) {
      if (!vv[j]) vv[j] = rowbase + k + 32;
      else {
        make0(vv[j], or, rowbase + k + 32);
        vv[j] = g;
      }
    }
  sprintf(name[vv[j]], "v%d", j);
}
```

This code is used in section 19.

25. ⟨ Make the column selector functions 25 ⟩ ≡

```
for (j = 0; j < nn; j += 4)
  if (col[j].m) {
    l = col[j].next - vec;
    if (!colgate[j >> 2]) {
      make0(colbase0 + (j >> 8), and, colbase1 + ((j >> 2) & #3f));
      sprintf(name[g], "c%03x", j >> 2);
      colgate[j >> 2] = g;
    }
    ;
    if (!uu[l]) uu[l] = colgate[j >> 2];
    else {
      make0(uu[l], or, colgate[j >> 2]);
      uu[l] = g;
    }
  }
for (j = 0; j < v; j++) sprintf(name[uu[j]], "u%d", j);
```

This code is used in section 19.

26. ⟨ Combine the selector functions to make the output 26 ⟩ ≡

```
for (j = 1, k = uu[0]; j < v; j++) {
  make0(uu[j], and, vv[j]);
  make0(k, or, g - 1);
  k = g;
}
sprintf(name[g], "y1");
printf("Phase␣3␣created␣%d␣gates.\n", g);
```

This code is used in section 19.

27. Checking. Now comes the proof of the pudding: We run through all 4520 inputs, and make sure that we've produced the desired output.

The tricky thing is that we want to hook up two copies of our circuit. So we evaluate twice, and OR the results together.

28. ⟨ Check the results 28 ⟩ ≡

```
count = 0;
for (l = 0; l < cases; l += 4) {
  grandval = 0, t = care[l] >> 2;
  ⟨ Set x_j and o_j from t 29 ⟩;
  ⟨ Evaluate the chain 30 ⟩;
  grandval |= val[g];
  t = t ⊕ ((t ⊕ (t >> 1)) & #2200) ⊕ ((t ⊕ (t << 1)) & #4400) ⊕
      ((t ⊕ (t >> 2)) & #11) ⊕ ((t ⊕ (t << 2)) & #44) ⊕
      ((t ⊕ (t >> 3)) & #1100) ⊕ ((t ⊕ (t << 3)) & #8800);
  ⟨ Set x_j and o_j from t 29 ⟩;
  ⟨ Evaluate the chain 30 ⟩;
  grandval |= val[g];
  if (grandval ≠ yval[l]) printf("Failure␣at␣%05x␣(should␣be␣%d)!\n", care[l] >> 2, yval[l]);
  count++;
}
printf("%d␣cases␣checked.\n", count);
```

This code is used in section 1.

29. **#define** *setx*(i,j) $val[x[i]] = (t \gg j)\ \&\ 1$
#define *seto*(i,j) $val[o[i]] = (t \gg j)\ \&\ 1$

⟨ Set x_j and o_j from t 29 ⟩ ≡

```
seto(5, 17), setx(5, 16);
seto(1, 15), seto(3, 14), seto(9, 13), seto(7, 12);
setx(1, 11), setx(3, 10), setx(9, 9), setx(7, 8);
seto(2, 7), seto(6, 6), seto(8, 5), seto(4, 4);
setx(2, 3), setx(6, 2), setx(8, 1), setx(4, 0);
```

This code is used in section 28.

30. ⟨ Evaluate the chain 30 ⟩ ≡

```
if (tracing[t]) {
  printf("Tracing␣case␣%05x:\n", t);
  for (k = 1; k < 19; k++) printf("%d=%d␣(%s)\n", k, val[k], name[k]);
}
for (k = 19; k ≤ g; k++) {
  switch (op[k]) {
  case and: val[k] = val[jx[k]] & val[kx[k]]; break;
  case or: val[k] = val[jx[k]] | val[kx[k]]; break;
  case xor: val[k] = val[jx[k]] ⊕ val[kx[k]]; break;
  case butnot: val[k] = val[jx[k]] & ∼val[kx[k]]; break;
  case nor: val[k] = 1 − (val[jx[k]] | val[kx[k]]); break;
  }
  if (tracing[t]) {
    printf("%d=", k);
    if (name[jx[k]][0]) printf(name[jx[k]]);
    else printf("%d", jx[k]);
    printf(opcode_name[op[k]]);
    if (name[kx[k]][0]) printf(name[kx[k]]);
    else printf("%d", kx[k]);
    printf("=%d", val[k]);
    if (name[k][0]) printf("␣(%s)\n", name[k]);
    else printf("\n");
  }
}
```

This code is used in section 28.

31. ⟨ Global variables 3 ⟩ +≡

```
int grandval;    /* OR of the eight inputs */
char tracing[1 ≪ 18];    /* selective verbose printouts */
```

32. Index.

TICTACTOE7