

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

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 >> j) & 1) << k)
⟨ Set up the initial columns 2 ⟩ ≡
    for (j = 0; j < nn; j++) col[j].ah = col[j].al = #fffffffff,
        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. \langle Global variables 3 $\rangle \equiv$
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.

\langle Make *care*[*k*] a “care” bit 4 $\rangle \equiv$
{
 j = *care*[*k*] \gg 14;
 t = *care*[*k*] & #3fff;
 if (*j* & #20) *col*[*t*].*ah* -= 1 \ll (*j* - #20);
 else *col*[*t*].*al* -= 1 \ll *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: %05x = %d, %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 $\rangle \equiv$

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

This code is used in section 6.

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

```
l = j  $\gg$  14, i = j & #3fff;
if (l & #20) col[i].bh += 1  $\ll$  (l - #20);
else col[i].bl += 1  $\ll$  l;
col[i].m++;
p = col[i].prev, q = col[i].next, p $\neg$ next = q, q $\neg$ prev = p;
for (l = col[i].c; l  $\equiv$  q $\neg$ c  $\wedge$  col[i].m > q $\neg$ m; q = q $\neg$ next) ;
p = q $\neg$ prev;
p $\neg$ next = q $\neg$ prev = &col[i], col[i].prev = p, col[i].next = q;
```

This code is used in section 5.

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

```
{
    l = j  $\gg$  14, i = j & #3fff;
    if (l & #20) col[i].ah += 1  $\ll$  (l - #20);
    else col[i].al += 1  $\ll$  l;
    col[i].c--;
    p = col[i].prev, q = col[i].next, p $\neg$ next = q, q $\neg$ prev = p;
    for (l = col[i].c; l < p $\neg$ c  $\vee$  (l  $\equiv$  p $\neg$ c  $\wedge$  col[i].m < p $\neg$ m); p = p $\neg$ prev) ;
    q = p $\neg$ next;
    p $\neg$ next = q $\neg$ prev = &col[i], col[i].prev = p, col[i].next = q;
}
```

This code is used in section 5.

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

```
for (p = head $\neg$ prev; p $\neg$ c > 1; p = p $\neg$ prev)
    if (p $\neg$ m) printf("%03x: %08x%08x, %08x%08x (%d, %d)\n", p - col, p $\neg$ ah, p $\neg$ al, p $\neg$ bh, p $\neg$ bl, p $\neg$ c, p $\neg$ m);
for (count = 0, p = head $\neg$ prev; p  $\neq$  head; p = p $\neg$ prev)
    if (p $\neg$ m) count++;
printf("%d columns contain 1s\n", count);
```

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 = #ffffff; /* all 1s vector */
  p = head-prev; /* the next column to be covered */
newvec: v++;
  vec[v].ah = vec[v].al = #ffffff, 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 & #fff]; /* count new 1s */
    t = p-bl & ~vec[k].bl, j += pop[((unsigned int) t) >> 16] + pop[t & #fff], j <= 16;
    t = vec[k].ah & ~p-ah, j += pop[((unsigned int) t) >> 16] + pop[t & #fff]; /* count lost *s */
    t = vec[k].al & ~p-al, j += pop[((unsigned int) t) >> 16] + pop[t & #fff];
    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 $v[0], \dots, v[n-1]$. The minterms appear in gates $o, o+1, \dots, o+2^n-1$, where o is the output.

\langle Subroutines 20 $\rangle \equiv$

```
int makemins(int *v,int n)
{
    register int j,k,g0,g1,fn,cn;
    if (n < 4)  $\langle$  Handle the base cases 21  $\rangle$ 
    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. \langle Handle the base cases 21 $\rangle \equiv$

```
{
    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. \langle Make minterms for the rows 22 $\rangle \equiv$

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

\langle Make minterms for the columns 23 $\rangle \equiv$

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

This code is used in section 19.

24. \langle Make the row selector functions 24 $\rangle \equiv$

```

for ( $j = 1; j < v; j++$ ) {
  for ( $k = 0; k < 32; k++$ )
    if ( $vec[j].bl \& (1 \ll k)$ ) {
      if ( $\neg vv[j]$ )  $vv[j] = rowbase + k;$ 
      else {
         $make0(vv[j], or, rowbase + k);$ 
         $vv[j] = g;$ 
      }
    }
  }
  for ( $k = 0; k < 32; k++$ )
    if ( $vec[j].bh \& (1 \ll k)$ ) {
      if ( $\neg vv[j]$ )  $vv[j] = rowbase + k + 32;$ 
      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. \langle Make the column selector functions 25 $\rangle \equiv$

```

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

```

This code is used in section 19.

26. \langle Combine the selector functions to make the output 26 $\rangle \equiv$

```

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. $\langle \text{Check the results 28} \rangle \equiv$

```

count = 0;
for (l = 0; l < cases; l += 4) {
  grandval = 0, t = care[l] >> 2;
   $\langle \text{Set } x_j \text{ and } o_j \text{ from } t \text{ 29} \rangle$ ;
   $\langle \text{Evaluate the chain 30} \rangle$ ;
  grandval |= val[g];
  t = t  $\oplus$  ((t  $\oplus$  (t >> 1)) & #2200)  $\oplus$  ((t  $\oplus$  (t << 1)) & #4400)  $\oplus$ 
    ((t  $\oplus$  (t >> 2)) & #11)  $\oplus$  ((t  $\oplus$  (t << 2)) & #44)  $\oplus$ 
    ((t  $\oplus$  (t >> 3)) & #1100)  $\oplus$  ((t  $\oplus$  (t << 3)) & #8800);
   $\langle \text{Set } x_j \text{ and } o_j \text{ from } t \text{ 29} \rangle$ ;
   $\langle \text{Evaluate the chain 30} \rangle$ ;
  grandval |= val[g];
  if (grandval  $\neq$  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 >> j) & 1`
`#define seto(i, j) val[o[i]] = (t >> j) & 1`
 $\langle \text{Set } x_j \text{ and } o_j \text{ from } t \text{ 29} \rangle \equiv$

```

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. $\langle \text{Evaluate the chain 30} \rangle \equiv$

```

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. $\langle \text{Global variables 3} \rangle + \equiv$

```

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

```

32. Index.

advancep: [13](#).
ah: [1](#), [2](#), [4](#), [11](#), [12](#), [13](#), [16](#), [17](#).
al: [1](#), [2](#), [4](#), [11](#), [12](#), [13](#), [16](#), [17](#).
and: [18](#), [20](#), [21](#), [25](#), [26](#), [30](#).
bh: [1](#), [10](#), [12](#), [13](#), [16](#), [17](#), [24](#).
bit: [2](#).
bits: [2](#), [3](#).
bl: [1](#), [10](#), [12](#), [13](#), [16](#), [17](#), [24](#).
buf: [2](#), [3](#).
butnot: [18](#), [21](#), [30](#).
c: [1](#).
care: [2](#), [3](#), [4](#), [5](#), [6](#), [8](#), [9](#), [28](#).
cases: [1](#), [2](#), [3](#), [5](#), [6](#), [28](#).
cn: [20](#).
col: [1](#), [2](#), [3](#), [4](#), [5](#), [10](#), [11](#), [12](#), [17](#), [25](#).
col_struct: [1](#).
colbase0: [18](#), [23](#), [25](#).
colbase1: [18](#), [23](#), [25](#).
colgate: [18](#), [25](#).
column: [1](#), [3](#), [14](#).
count: [1](#), [12](#), [13](#), [28](#).
coverit: [13](#).
dir: [6](#), [7](#), [8](#), [9](#).
exit: [2](#), [8](#), [9](#).
fgets: [2](#).
fn: [20](#).
fprintf: [2](#), [8](#), [9](#).
g: [18](#).
gates: [18](#).
grandval: [28](#), [31](#).
g0: [20](#), [21](#).
g1: [20](#).
head: [1](#), [2](#), [12](#), [13](#).
i: [1](#).
inp: [18](#), [19](#).
j: [1](#), [20](#).
jx: [18](#), [19](#), [30](#).
k: [1](#), [20](#).
kx: [18](#), [19](#), [30](#).
l: [1](#).
leader: [5](#), [6](#), [7](#), [8](#), [9](#).
link: [5](#), [6](#), [7](#), [8](#), [9](#).
m: [1](#).
main: [1](#).
makegate: [19](#).
makemins: [20](#), [22](#), [23](#).
make0: [19](#), [20](#), [21](#), [24](#), [25](#), [26](#).
make1: [19](#).
make2: [19](#).
n: [20](#).
name: [18](#), [19](#), [22](#), [24](#), [25](#), [26](#), [30](#).

newvec: [13](#).
next: [1](#), [2](#), [4](#), [10](#), [11](#), [17](#), [25](#).
nn: [1](#), [2](#), [3](#), [25](#).
nnn: [1](#), [7](#).
nor: [18](#), [21](#), [30](#).
o: [18](#).
op: [18](#), [19](#), [30](#).
opcode: [18](#).
opcode_name: [18](#), [30](#).
or: [18](#), [24](#), [25](#), [26](#), [30](#).
p: [1](#), [18](#).
pop: [13](#), [14](#), [15](#).
prev: [1](#), [2](#), [4](#), [10](#), [11](#), [12](#), [13](#).
printf: [12](#), [13](#), [17](#), [26](#), [28](#), [30](#).
q: [1](#).
rowbase: [18](#), [22](#), [24](#).
seto: [29](#).
setx: [29](#).
sprintf: [19](#), [22](#), [24](#), [25](#), [26](#).
sscanf: [2](#).
stderr: [2](#), [8](#), [9](#).
stdin: [2](#).
t: [1](#).
tracing: [30](#), [31](#).
uu: [18](#), [25](#), [26](#).
v: [14](#), [20](#).
val: [18](#), [28](#), [29](#), [30](#).
vec: [13](#), [14](#), [16](#), [17](#), [24](#), [25](#).
vecs: [14](#), [18](#).
vv: [18](#), [24](#), [26](#).
vx: [18](#), [19](#), [22](#), [23](#).
x: [18](#).
xor: [18](#), [30](#).
yval: [2](#), [3](#), [5](#), [8](#), [9](#), [28](#).

〈Change bit j to don't-care 11〉 Used in section 5.
 〈Check the results 28〉 Used in section 1.
 〈Combine the selector functions to make the output 26〉 Used in section 19.
 〈Compute the *pop* table for population counts 15〉 Used in section 13.
 〈Cover column p with vector l 17〉 Used in section 13.
 〈Do phase 1 5〉 Used in section 1.
 〈Do phase 2 13〉 Used in section 1.
 〈Do phase 3 19〉 Used in section 1.
 〈Evaluate the chain 30〉 Used in section 28.
 〈Find the equivalence classes 6〉 Used in section 5.
 〈Global variables 3, 7, 14, 18, 31〉 Used in section 1.
 〈Handle the base cases 21〉 Used in section 20.
 〈If p is incompatible with $vec[k]$, **continue** 16〉 Used in section 13.
 〈Make minterms for the columns 23〉 Used in section 19.
 〈Make minterms for the rows 22〉 Used in section 19.
 〈Make the column selector functions 25〉 Used in section 19.
 〈Make the row selector functions 24〉 Used in section 19.
 〈Make $care[k]$ a “care” bit 4〉 Used in section 2.
 〈Make $care[k]$ equivalent to its reflection 9〉 Used in section 6.
 〈Make $care[k]$ equivalent to its rotation 8〉 Used in section 6.
 〈Print the results of Phase 1 12〉 Used in section 5.
 〈Set x_j and o_j from t 29〉 Used in section 28.
 〈Set care bit j to 1 10〉 Used in section 5.
 〈Set up the initial columns 2〉 Used in section 1.
 〈Subroutines 20〉 Used in section 1.

TICTACTOE7

	Section	Page
Intro	1	1
Phase 1	5	4
Phase 2	13	6
Phase 3	18	7
Checking	27	10
Index	32	12