1. Intro. This program tries heuristically to construct a small circuit to decide when it is safe to move in the center in tictactoe. More precisely, we design about 1/8th of what is needed; we assume that eight copies of this circuit will be ORed together appropriately, taking account of the symmetries of the board.

In contrast to the circuit in TICTACTOE3, this program works entirely by statistical and formal methods rather than using any sort of logical intuition that makes sense.

There are 18 inputs, as in TICTACTOE3. I partition the truth table function into 2^6 rows indexed by $o_5x_5o_2o_6o_8o_4$ and 2^{12} columns indexed by $x_2x_6x_8x_4o_1o_3o_9o_7x_1x_3x_9x_7$, as suggested by the output of TICTACTOE5. Each column has a score (c,m) when it contains c cares and m ones among those cares. Low-scoring columns are the easiest to handle.

The computation proceeds in three phases.

Phase 1: For each equivalence class of positions $\{\alpha_1,\ldots,\alpha_k\}$ for which a center move is legitimate, let them correspond to columns $\{\beta_1,\ldots,\beta_k\}$ (which may not be distinct), and assume that β_1 has the lexicographically smallest score. Place a 1 in that position, and put don't-cares in the other $k-1$ positions. Repeating that process will yield a set of 2^{12} columns in which I hope there will be few columns that have many 1s.

Phase 2: Find a smallish set of vectors that matches every column. This is a covering problem, where a vector covers (or matches) a column if and only if it agrees with all of the care-bits in that column. When a column matches a vector, we essentially use that vector to fill in all the don't cares.

Phase 3: Build a circuit with Lupanov's method.

The standard input file should be the output of TICTACTOE4.

```
#define nn (1 << 12)
#define nnn (1 << 18)
#define cases 4520
#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. An important simplification occurs here: If $x_5 = 1$ or $o_5 = 1$, the output y_5 is guaranteed to be zero, so I can treat all such cases as don't-cares. (It follows that 48 of the 64 bits in every column are automatically don't-cares. Still, I've written the program to work with all 64 bits, because TICTACTOE7 will need that generality.)

```
#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 = #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);
    yval[k] = buf[11] − '0';   /* this is y5 */
    if (yval[k] < 0 ∨ yval[k] > 1) {
      fprintf(stderr, "invalid␣input␣line!␣%s", buf);
      exit(−1);
    }
    if (care[k] < #10000) ⟨ 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] >> 12;
  t = care[k] & #fff;
  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 $(leader[t] \neq t)$ **continue**;
      for $(j = t, t = link[t];\ t \neq care[k];\ t = link[t])$
        if $(col[t \mathbin{\&} \text{\#fff}].c \leq col[j \mathbin{\&} \text{\#fff}].c \wedge (col[t \mathbin{\&} \text{\#fff}].c < col[j \mathbin{\&} \text{\#fff}].c \vee col[t \mathbin{\&} \text{\#fff}].m < col[j \mathbin{\&} \text{\#fff}].m))$ $j = t$;
      ⟨ Set care bit j to 1 10 ⟩;
      for $(t = j, j = link[j];\ j \neq t;\ j = link[j])$ ⟨ 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 ⟩ ≡
  {
    $t = care[k]$;
    $j = t \oplus ((t \oplus (t \gg 1)) \mathbin{\&} \text{\#7777}) \oplus ((t \oplus (t \ll 3)) \mathbin{\&} \text{\#8888})$;
    if $(yval[dir[j]] \neq yval[k])$ {
      fprintf (*stderr*, `"Error:␣y[%05x]=%d,␣y[%05x]=%d!\n"`, $t, yval[k], j, yval[dir[j]]$);
      exit (-3);
    }
    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.

9. ⟨ Make $care[k]$ equivalent to its reflection 9 ⟩ ≡

```
{
  t = care[k];
  j = t ⊕ ((t ⊕ (t ≫ 1)) & #55) ⊕ ((t ⊕ (t ≪ 1)) & #aa) ⊕
      ((t ⊕ (t ≫ 2)) & #1100) ⊕ ((t ⊕ (t ≪ 2)) & #4400);
  if (yval[dir[j]] ≠ yval[k]) {
    fprintf(stderr, "Error:␣y[%05x]=%d,␣y[%05x]=%d!\n", t, yval[k], j, yval[dir[j]]);
    exit(−4);
  }
  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.

10. ⟨ Set care bit j to 1 10 ⟩ ≡

```
l = j ≫ 12, i = j & #fff;
if (l & #20) col[i].bh += 1 ≪ (l − #20);
else col[i].bl += 1 ≪ l;
col[i].m++;
p = col[i].prev, q = col[i].next, p→next = q, q→prev = p;
for (l = col[i].c; l ≡ q→c ∧ col[i].m > q→m; q = q→next) ;
p = q→prev;
p→next = q→prev = &col[i], col[i].prev = p, col[i].next = q;
```

This code is used in section 5.

11. ⟨ Change bit j to don't-care 11 ⟩ ≡

```
{
  l = j ≫ 12, i = j & #fff;
  if (l & #20) col[i].ah += 1 ≪ (l − #20);
  else col[i].al += 1 ≪ l;
  col[i].c−−;
  p = col[i].prev, q = col[i].next, p→next = q, q→prev = p;
  for (l = col[i].c; l < p→c ∨ (l ≡ p→c ∧ col[i].m < p→m); p = p→prev) ;
  q = p→next;
  p→next = q→prev = &col[i], col[i].prev = p, col[i].next = q;
}
```

This code is used in section 5.

12. ⟨ Print the results of Phase 1 12 ⟩ ≡

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

This code is used in section 5.

13. Phase 2. It turns out that Phase 1 does such a good job, there's really no need for a fancy Phase 2; all nonzero columns are simply covered by unit vectors or the all-1s vector. But I'll implement a greedy covering algorithm anyway, because it will be needed in TICTACTOE7.

The idea of this algorithm is to cover columns one by one, in decreasing order of (c, m). At each step we try to augment the current set of covering vectors in such a way as to minimize the number of 1s added, and (for a fixed number of 1s) the number of asterisks subtracted.

```
⟨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. We're using only 16 of the 64 rows, as remarked earlier. But we'll want to share logic with the output of TICTACTOE7, so all 64 minterms are made at this step. I will figure out later which are them aren't really needed.

```
⟨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⟩ ≡

```
vv[0] = 19;    /* kludge: this is 'ō₅ ∧ x̄₅' */
sprintf(name[19], "v0");
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++)
  if (col[j].m) {
    l = col[j].next − vec;
    if (¬colgate[j]) {
      make0(colbase0 + (j >> 6), and, colbase1 + (j & #3f));
      sprintf(name[g], "c%03x", j);
      colgate[j] = g;
    }
    ;
    if (¬uu[l]) uu[l] = colgate[j];
    else {
      make0(uu[l], or, colgate[j]);
      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 = k = 0; j < v; j++) {
  make0(uu[j], and, vv[j]);
  if (k) make0(k, or, g − 1);
  k = g;
}
sprintf(name[g], "y5");
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 eight copies of our circuit. So we evaluate eight times, and OR the results together.

28. ⟨ Check the results 28 ⟩ ≡

```
count = 0;
for (l = 0; l < cases; l++) {
  grandval = 0;
  for (i = 0, t = care[l]; i < 4; i++) {
    ⟨ Set x_j and o_j from t 29 ⟩;
    ⟨ Evaluate the chain 30 ⟩;
    grandval |= val[g];
    t = t ⊕ ((t ⊕ (t ≫ 1)) & #7777) ⊕ ((t ⊕ (t ≪ 3)) & #8888);
  }
  t = t ⊕ ((t ⊕ (t ≫ 1)) & #55) ⊕ ((t ⊕ (t ≪ 1)) & #aa) ⊕
      ((t ⊕ (t ≫ 2)) & #1100) ⊕ ((t ⊕ (t ≪ 2)) & #4400);
  for (i = 0; i < 4; i++) {
    ⟨ Set x_j and o_j from t 29 ⟩;
    ⟨ Evaluate the chain 30 ⟩;
    grandval |= val[g];
    t = t ⊕ ((t ⊕ (t ≫ 1)) & #7777) ⊕ ((t ⊕ (t ≪ 3)) & #8888);
  }
  if (grandval ≠ yval[l]) printf("Failure␣at␣%05x␣(should␣be␣%d)!\n", care[l], 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.

TICTACTOE6