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

1. Intro. This program tests my proposed Boolean chain for tictactoe moves.

```
#define rank z.I /* number of moves made */
#define link y.V /* next vertex of same rank */
#define head x.V /* first vertex of given rank */
#define winner w.I /* is this a winning position? */
#define score w.I /* minimax value of position */
#define bitcode v.I /* binary representation of this position */
#define goodmoves u.I /* union of all desirable successors */
#define gates 1000
#include "gb_graph.h"
#include "gb_save.h"
  typedef enum {
    inp, and, or, xor, butnot, nor
  } opcode;
  char *opcode_name[] = {"input","&","|","^",">","$"};
  int pref[] = {5,1,3,9,7,2,6,8,4}; /* preference order for moves */
  opcode op[gates];
  char val[gates];
  int jx[gates], kx[gates], p[gates];
  char name[gates][8];
  int x[10], o[10], m[10], w[10], b[10], f[10], c[10], y[10], a[10], e[10], z[10], xax[10][10], xox[10][10], oao[10][10],
      ooo[10][10], alf[10][10], bet[10][10]; /* addresses of named gates */
  int g; /* address of the most recently generated gate */
  char code[10];
  char tracing[1 << 18]; /* selective verbose printouts */
  int count;
  main()
  {
    register int j, k, l, q, qq, s;
    register Graph *gg = restore_graph("/tmp/tictactoe.gb");
    register Vertex *u, *v;
    register Arc *aa;
    ⟨ Construct the gates 2 ⟩;
    printf("(OK, I've set up %d gates.)\n", g);
    ⟨ Compute the optimum moves 8 ⟩;
    ⟨ Check for errors 10 ⟩;
    printf("%d cases checked.\n", count);
  }
```

2. Here's the design of the circuit, with gates built up one by one.

#**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*$(name[g],s,j,k), v=g$

⟨ Construct the gates 2 ⟩ ≡
for $(j=1;\ j\le 9;\ j{+}{+})$ *make1*(`"x%d"`$, j, 0, inp, 0, x[j]$);
for $(j=1;\ j\le 9;\ j{+}{+})$ *make1*(`"o%d"`$, j, 0, inp, 0, o[j]$);
for $(j=1;\ j\le 9;\ j{+}{+})$ *make1*(`"m%d"`$, j, x[j], nor, o[j], m[j]$);
⟨ Make pairs for each of the eight winning lines 3 ⟩;
⟨ Make the w, b, f, a, and c gates 4 ⟩;
⟨ Make the priority selections 6 ⟩;
⟨ Make the final equations 7 ⟩;

This code is used in section 1.

3. #**define** *makepair*(i,j)
make2(`"xax%d%d"`$, i, j, x[i], and, x[j], xax[i][j]$);
make2(`"xox%d%d"`$, i, j, x[i], xor, x[j], xox[i][j]$);
make2(`"oao%d%d"`$, i, j, o[i], and, o[j], oao[i][j]$);
make2(`"ooo%d%d"`$, i, j, o[i], xor, o[j], ooo[i][j]$); $ooo[j][i]=g$;
make2(`"alf%d%d"`$, i, j, xox[i][j], butnot, ooo[i][j], alf[i][j]$); $alf[j][i]=g$;
make2(`"bet%d%d"`$, i, j, ooo[i][j], butnot, xox[i][j], bet[i][j]$); $bet[j][i]=g$;
#**define** *makeline*(i,j,k)
makepair(i,j); *makepair*(i,k); *makepair*(j,k)

⟨ Make pairs for each of the eight winning lines 3 ⟩ ≡
makeline$(1,2,3)$;
makeline$(4,5,6)$;
makeline$(7,8,9)$;
makeline$(1,4,7)$;
makeline$(2,5,8)$;
makeline$(3,6,9)$;
makeline$(1,5,9)$;
makeline$(3,5,7)$;

This code is used in section 2.

4. Here we can use the nice fact that, when i is a corner, the matrix

$$\begin{matrix} i & 2i & 3i \\ 4i & 5i & 6i \\ 7i & 8i & 9i \end{matrix}$$

gives us another way to look at the board, modulo 10.

```
#define mmod(i,x)  ((i*x)%10)
#define makecorner(i,j1,k1,j2,k2,j3,k3)
        make0(xax[j1][k1],or,xax[j2][k2]);
        make0(xax[j3][k3],or,g-1);
        make1("w%d",i,m[i],and,g-1,w[i]);
        make0(oao[j1][k1],or,oao[j2][k2]);
        make0(oao[j3][k3],or,g-1);
        make1("b%d",i,m[i],and,g-1,b[i]);
        make0(alf[j1][k1],and,alf[j2][k2]);
        make0(alf[j1][k1],or,alf[j2][k2]);
        make0(alf[j3][k3],and,g-1);
        make0(g-3,xor,g-1);
        make1("f%d",i,m[i],and,g-1,f[i]);
        make0(bet[j1][k1],and,bet[j2][k2]);
        make1("e%d",i,m[i],and,g-1,e[i]);
        make0(x[mmod(i,3)],and,bet[mmod(i,8)][mmod(i,9)]);
        make0(g-1,and,ooo[mmod(i,4)][mmod(i,6)]);
        make0(x[mmod(i,7)],and,bet[mmod(i,6)][mmod(i,9)]);
        make0(g-1,and,ooo[mmod(i,2)][mmod(i,8)]);
        make0(g-3,or,g-1);
        make0(x[mmod(i,2)],butnot,ooo[mmod(i,3)][mmod(i,6)]);
        make0(m[mmod(i,8)],and,g-1);
        make0(x[mmod(i,4)],butnot,ooo[mmod(i,7)][mmod(i,8)]);
        make0(m[mmod(i,6)],and,g-1);
        make0(g-3,or,g-1);
        make0(g-1,and,m[10-i]);
        make0(g-7,or,g-1);
        make1("a%d",i,m[i],and,g-1,a[i])
#define makemid(i,j1,k1,j2,k2)
        make0(xax[j1][k1],or,xax[j2][k2]);
        make1("w%d",i,m[i],and,g-1,w[i]);
        make0(oao[j1][k1],or,oao[j2][k2]);
        make1("b%d",i,m[i],and,g-1,b[i]);
        make0(alf[j1][k1],and,alf[j2][k2]);
        make1("f%d",i,m[i],and,g-1,f[i]);
        make0(bet[j1][k1],and,bet[j2][k2]);
        make1("e%d",i,m[i],and,g-1,e[i]);
        z[i]=m[i]
#define makecz(i)
        make1("z%d",i,m[i],butnot,e[10-i],z[i]);
        make0(x[mmod(i,6)],and,o[mmod(i,7)]);
        make0(e[i],butnot,g-1);
        make0(x[mmod(i,8)],and,o[mmod(i,3)]);
        make0(g-2,butnot,g-1);
        make1("c%d",i,g-1,butnot,e[10-i],c[i])
```

⟨ Make the w, b, f, a, and c gates 4 ⟩ ≡
```
makecorner(1,2,3,4,7,5,9);
makemid(2,1,3,5,8);
makecorner(3,1,2,6,9,5,7);
makemid(4,1,7,5,6);
⟨ Make the middle gates 5 ⟩;
makemid(6,3,9,4,5);
makecorner(7,1,4,8,9,3,5);
makemid(8,7,9,2,5);
makecorner(9,3,6,7,8,1,5);
makecz(1);
makecz(3);
makecz(7);
makecz(9);
```
This code is used in section 2.

5. ⟨ Make the middle gates 5 ⟩ ≡
```
make0(xax[1][9], or, xax[2][8]);
make0(xax[3][7], or, xax[4][6]);
make0(g − 2, or, g − 1);
make1("w%d", 5, m[5], and, g − 1, w[5]);
make0(oao[1][9], or, oao[2][8]);
make0(oao[3][7], or, oao[4][6]);
make0(g − 2, or, g − 1);
make1("b%d", 5, m[5], and, g − 1, b[5]);
make0(alf[1][9], xor, alf[3][7]);
make0(alf[1][9], xor, alf[2][8]);
make0(alf[3][7], xor, alf[4][6]);
make0(g − 3, or, g − 2);
make0(g − 3, xor, g − 2);
make0(g − 2, butnot, g − 1);
make1("f%d", 5, m[5], and, g − 1, f[5]);
make0(bet[1][9], xor, bet[3][7]);
make0(bet[1][9], xor, bet[2][8]);
make0(bet[3][7], xor, bet[4][6]);
make0(g − 3, or, g − 2);
make0(g − 3, xor, g − 2);
make1("e%d", 5, g − 2, butnot, g − 1, e[5]);
  /* make0(xox[1][9],and,ooo[3][7]); make0(xox[3][7],and,ooo[1][9]); */
make0(xox[1][3], xor, xox[7][9]);
make0(ooo[1][3], xor, ooo[7][9]);
make0(g − 1, butnot, g − 2);
make0(m[2], butnot, g − 1);
make0(m[4], and, g − 1);
make0(m[6], and, g − 1);
make0(m[8], and, g − 1);
make1("z%d", 5, m[5], butnot, g − 1, z[5]);
```
This code is used in section 4.

6. #**define** *makeprior*(*s*)
{ *make2*("p%s%d", #*s*, *pref*[*k*], *qq*, *or*, *q*, *p*[*s*[*pref*[*k*]]]); *qq* = *g*, *q* = *s*[*pref*[*k*]]; }

⟨ Make the priority selections 6 ⟩ ≡
```
qq = w[5], q = w[1], p[q] = qq;
for (k = 2; k < 9; k++) makeprior(w);
for (k = 0; k < 9; k++) makeprior(b);
for (k = 0; k < 9; k++) makeprior(f);
for (k = 1; k < 5; k++) makeprior(a);
for (k = 1; k < 5; k++) makeprior(c);
for (k = 0; k < 9; k++) makeprior(z);
```
This code is used in section 2.

7. ⟨ Make the final equations 7 ⟩ ≡
```
for (k = 1; k ≤ 9; k++) {
  if (k ≡ 5) q = w[5];      /* w[5] has no predecessor */
  else {
    make0(w[k], butnot, p[w[k]]);
    q = g;
  }
  make0(b[k], butnot, p[b[k]]);
  make0(q, or, g − 1);
  make0(f[k], butnot, p[f[k]]);
  make0(g − 2, or, g − 1);
  if ((k & 1) ∧ (k ≠ 5)) {      /* attack and counterattack */
    make0(a[k], butnot, p[a[k]]);
    make0(g − 2, or, g − 1);
    make0(c[k], butnot, p[c[k]]);
    make0(g − 2, or, g − 1);
  }
  make0(z[k], butnot, p[z[k]]);
  make1("y%d", k, g − 2, or, g − 1, y[k]);
}
```
This code is used in section 2.

8. The *score* takes over from the *winner* field in the input graph.

```
⟨ Compute the optimum moves 8 ⟩ ≡
  ⟨ Complement the bit codes on even levels 9 ⟩;
  for (l = 9; l ≥ 0; l−−)
    for (v = (gg→vertices + l)→head; v; v = v→link) {
      if (v→winner) v→score = −1;
      else if (v→rank < 9) {
        for (s = 99, aa = v→arcs; aa; aa = aa→next) {
          u = aa→tip;
          if (s > u→score) s = u→score;
        }
        v→score = −s;
        for (aa = v→arcs; aa; aa = aa→next) {
          u = aa→tip;
          if (s ≡ u→score) v→goodmoves |= u→bitcode;
        }
      }
    }
```

This code is used in section 1.

9. In this program, I don't like the way TICTACTOE1 set up the bit codes in the graph. Instead, each `X` is now represented by 01, and each `O` by 10.

```
⟨ Complement the bit codes on even levels 9 ⟩ ≡
  for (l = 2; l < 9; l += 2)
    for (v = (gg→vertices + l)→head; v; v = v→link) {
      for (j = 0, k = 3; j < 9; j++, k ≪= 2)
        if (v→bitcode & k) v→bitcode ⊕= k;
    }
```

This code is used in section 8.

10.

```
⟨ Check for errors 10 ⟩ ≡
  for (l = 8; l ≥ 0; l−−)
    for (v = (gg→vertices + l)→head; v; v = v→link)
      if (v→arcs) {
        ⟨ Set x_j and o_j from v→bitcode 11 ⟩;
        ⟨ Evaluate the chain 12 ⟩;
        ⟨ Check that the computed move is present in v→goodmoves 13 ⟩;
      }
```

This code is used in section 1.

11.

```
⟨ Set x_j and o_j from v→bitcode 11 ⟩ ≡
  for (j = 8, k = v→bitcode; j ≥ 0; j−−, k ≫= 2) val[x[pref[j]]] = k & 1, val[o[pref[j]]] = (k ≫ 1) & 1;
```

This code is used in section 10.

12. ⟨ Evaluate the chain 12 ⟩ ≡

```
if (tracing[v→bitcode]) {
  printf("Tracing␣%s:\n", v→name);
  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[v→bitcode]) {
    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 10.

13. ⟨ Check that the computed move is present in $v\rightarrow goodmoves$ 13 ⟩ ≡

```
for (j = k = 0; j < 9; j++) k = (k ≪ 2) + (val[y[pref[j]]] ? 2 : 0);
count++;
if (k & (k − 1)) printf("It␣made␣more␣than␣one␣move␣from␣%s!\n", v→name);
if (¬(k & v→goodmoves)) {
  printf("%s␣(%d)␣moved␣to␣", v→name, v→score);
  for (j = 8, k = (k ≫ 1) + v→bitcode; j ≥ 0; j−−, k ≫= 2)
    switch (k & 3) {
    case 0: code[pref[j]] = '␣'; break;
    case 1: code[pref[j]] = 'X'; break;
    case 2: code[pref[j]] = 'O'; break;
    }
  printf("%c%c%c/%c%c%c/%c%c%c␣instead␣of␣",
      code[1], code[2], code[3], code[4], code[5], code[6], code[7], code[8], code[9]);
  for (j = 8, k = v→goodmoves; j ≥ 0; j−−, k ≫= 2)
    switch (k & 3) {
    case 0: code[pref[j]] = '␣'; break;
    case 1: code[pref[j]] = 'O'; break;
    case 2: code[pref[j]] = 'X'; break;
    }
  printf("%c%c%c/%c%c%c/%c%c%c\n",
      code[1], code[2], code[3], code[4], code[5], code[6], code[7], code[8], code[9]);
}
```

This code is used in section 10.

14. Index.

a: 1.
aa: 1, 8.
alf: 1, 3, 4, 5.
and: 1, 3, 4, 5, 12.
Arc: 1.
arcs: 8, 10.
b: 1.
bet: 1, 3, 4, 5.
bitcode: 1, 8, 9, 11, 12, 13.
butnot: 1, 3, 4, 5, 7, 12.
c: 1.
code: 1, 13.
count: 1, 13.
e: 1.
f: 1.
g: 1.
gates: 1.
gg: 1, 8, 9, 10.
goodmoves: 1, 8, 13.
Graph: 1.
head: 1, 8, 9, 10.
inp: 1, 2.
j: 1.
jx: 1, 2, 12.
j1: 4.
j2: 4.
j3: 4.
k: 1.
kx: 1, 2, 12.
k1: 4.
k2: 4.
k3: 4.
l: 1.
link: 1, 8, 9, 10.
m: 1.
main: 1.
makecorner: 4.
makecz: 4.
makegate: 2.
makeline: 3.
makemid: 4.
makepair: 3.
makeprior: 6.
make0: 2, 4, 5, 7.
make1: 2, 4, 5, 7.
make2: 2, 3, 6.
mmod: 4.
name: 1, 2, 12, 13.
next: 8.
nor: 1, 2, 12.
o: 1.
oao: 1, 3, 4, 5.
ooo: 1, 3, 4, 5.
op: 1, 2, 12.
opcode: 1.
opcode_name: 1, 12.
or: 1, 4, 5, 6, 7, 12.
p: 1.
pref: 1, 6, 11, 13.
printf: 1, 12, 13.
q: 1.
qq: 1, 6.
rank: 1, 8.
restore_graph: 1.
s: 1.
score: 1, 8, 13.
sprintf: 2.
tip: 8.
tracing: 1, 12.
u: 1.
v: 1.
val: 1, 11, 12, 13.
Vertex: 1.
vertices: 8, 9, 10.
w: 1.
winner: 1, 8.
x: 1.
xax: 1, 3, 4, 5.
xor: 1, 3, 4, 5, 12.
xox: 1, 3, 5.
y: 1.
z: 1.

⟨ Check for errors 10 ⟩ Used in section 1.
⟨ Check that the computed move is present in $v\neg goodmoves$ 13 ⟩ Used in section 10.
⟨ Complement the bit codes on even levels 9 ⟩ Used in section 8.
⟨ Compute the optimum moves 8 ⟩ Used in section 1.
⟨ Construct the gates 2 ⟩ Used in section 1.
⟨ Evaluate the chain 12 ⟩ Used in section 10.
⟨ Make pairs for each of the eight winning lines 3 ⟩ Used in section 2.
⟨ Make the final equations 7 ⟩ Used in section 2.
⟨ Make the middle gates 5 ⟩ Used in section 4.
⟨ Make the priority selections 6 ⟩ Used in section 2.
⟨ Make the w, b, f, a, and c gates 4 ⟩ Used in section 2.
⟨ Set x_j and o_j from $v\neg bitcode$ 11 ⟩ Used in section 10.

TICTACTOE3

	Section	Page
Intro	1	1
Index	14	8