

(Downloaded from <https://cs.stanford.edu/~knuth/programs.html> and typeset on May 28, 2023)

1* Intro. This program generates clauses for the transition relation from time 0 to time r in Conway’s Game of Life (thus simulating r steps), on an $m \times n$ grid, given m , n , and r . The live cells are constrained to remain in this grid, and the configuration at time r is constrained to be exactly the same as it was at time 0.

Moreover, this version assumes that the configurations all have symmetry under both horizontal and vertical reflection. (The situation at (x, y) is identical to the situation at $(m + 1 - x, y)$ and $(x, n + 1 - y)$.) Thus the number of independent variables is essentially reduced by a factor of 4.

The Boolean variable for cell (x, y) at time t is named by its so-called “xty code,” namely by the decimal value of x , followed by a code letter for t , followed by the decimal value of y . For example, if $x = 10$ and $y = 11$ and $t = 0$, the variable that indicates liveness of the cell is **10a11**; and the corresponding variable for $t = 1$ is **10b11**.

Up to 19 auxiliary variables are used together with each xty code, in order to construct clauses that define the successor state. The names of these variables are obtained by appending one of the following two-character combinations to the xty code: **A2**, **A3**, **A4**, **B1**, **B2**, **B3**, **B4**, **C1**, **C2**, **C3**, **C4**, **D1**, **D2**, **E1**, **E2**, **F1**, **F2**, **G1**, **G2**. These variables are derived from the Bailleux–Boufkhad method of encoding cardinality constraints: The auxiliary variable **Ak** stands for the condition “at least k of the eight neighbors are alive.” Similarly, **Bk** stands for “at least k of the first four neighbors are alive,” and **Ck** accounts for the other four neighbors. Codes **D**, **E**, **F**, and **G** refer to pairs of neighbors. Thus, for instance, **10a11C2** means that at least two of the last four neighbors of cell $(10, 11)$ are alive.

Those auxiliary variables receive values by means of up to 77 clauses per cell. For example, if u and v are the neighbors of cell z that correspond to a pairing of type **D**, there are six clauses

$$\bar{u}d_1, \quad \bar{v}d_1, \quad \bar{u}\bar{v}d_2, \quad uv\bar{d}_1, \quad u\bar{d}_2, \quad v\bar{d}_2.$$

The sixteen clauses

$$\begin{aligned} &\bar{d}_1b_1, \quad \bar{e}_1b_1, \quad \bar{d}_2b_2, \quad \bar{d}_1\bar{e}_1b_2, \quad \bar{e}_2b_2, \quad \bar{d}_2\bar{e}_1b_3, \quad \bar{d}_1\bar{e}_2b_3, \quad \bar{d}_2\bar{e}_2b_4, \\ &d_1e_1\bar{b}_1, \quad d_1e_2\bar{b}_2, \quad d_2e_1\bar{b}_2, \quad d_1\bar{b}_3, \quad d_2e_2\bar{b}_3, \quad e_1\bar{b}_3, \quad d_2\bar{b}_4, \quad e_2\bar{b}_4 \end{aligned}$$

define b variables from d ’s and e ’s; and another sixteen define c ’s from f ’s and g ’s in the same fashion. A similar set of 21 clauses will define the a ’s from the b ’s and c ’s.

Once the a ’s are defined, thus essentially counting the live neighbors of cell z , the next state z' is defined by five further clauses

$$\bar{a}_4\bar{z}', \quad a_2\bar{z}', \quad a_3z\bar{z}', \quad \bar{a}_3a_4z', \quad \bar{a}_2a_4\bar{z}z'.$$

For example, the last of these states that z' will be true (i.e., that cell z will be alive at time $t + 1$) if z is alive at time t and has ≥ 2 live neighbors but not ≥ 4 .

Nearby cells can share auxiliary variables, according to a tricky scheme that is worked out below. In consequence, the actual number of auxiliary variables and clauses per cell is reduced from 19 and $77 + 5$ to 13 and $57 + 5$, respectively, except at the boundaries.

2* So here's the overall outline of the program.

```
#define maxx 50    /* maximum number of lines in the pattern supplied by stdin */
#define maxy 50     /* maximum number of columns per line in stdin */
#include <stdio.h>
#include <stdlib.h>
char p[maxx + 2][maxy + 2]; /* is cell (x,y) potentially alive? */
char have_b[maxx + 2][maxy + 2]; /* did we already generate b(x,y)? */
char have_d[maxx + 2][maxy + 2]; /* did we already generate d(x,y)? */
char have_e[maxx + 2][maxy + 4]; /* did we already generate e(x,y)? */
char have_f[maxx + 4][maxy + 2]; /* did we already generate f(x-2,y)? */
int tt; /* the time being considered */
int mm, nn, r; /* the command-line parameters */
int xmax, ymax; /* the number of rows and columns in the input pattern */
int xmin = maxx, ymin = maxy; /* limits in the other direction */
char timecode[] = "abcdefghijklmnopqrstuvwxyz"
                  "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
                  "!\"#$%&'()*+,-./:;<=>?@[\\]^_`{|}~"; /* codes for 0 ≤ t ≤ 83 */
char buf[maxy + 2]; /* input buffer */
unsigned int clause[4]; /* clauses are assembled here */
int clauseptr; /* this many literals are in the current clause */
⟨Subroutines 6*⟩
main(int argc, char *argv[])
{
    register int j, k, x, y;
    ⟨Process the command line 3*⟩;
    for (tt = 0; tt < r; tt++) {
        ymax = nn, ymin = 1;
        xmax = mm, xmin = 1;
        for (x = 0; x + x ≤ mm + 1; x++)
            for (y = 0; y + y ≤ nn + 1; y++) {
                ⟨If cell (x,y) is obviously dead at time t + 1, continue 5*⟩;
                a(x,y);
                zprime(x,y);
                if (pp(x,y) ≡ 0) applit(x,y,1,1), outclause(); /* keep the configuration caged */
            }
    }
    ⟨Rule out smaller periods 16*⟩;
}

3* ⟨Process the command line 3*⟩ ≡
if (argc ≠ 4 ∨ sscanf(argv[1], "%d", &mm) ≠ 1 ∨ sscanf(argv[2], "%d", &nn) ≠ 1 ∨ sscanf(argv[3], "%d",
    &r) ≠ 1) {
    fprintf(stderr, "Usage: _s_m_n_r\n", argv[0]);
    exit(-1);
}
printf("~sat-life-gridquad-cycle_d_d_d\n", mm, nn, r);
```

This code is used in section 2*.

4. $\langle \text{Input the pattern 4} \rangle \equiv$

```

for ( $x = 1$ ; ;  $x++$ ) {
  if ( $\neg fgets(buf, maxy + 2, stdin)$ ) break;
  if ( $x > maxx$ ) {
     $fprintf(stderr, "Sorry, \_the\_pattern\_should\_have\_at\_most\_ \%d\_rows! \n", maxx)$ ;
     $exit(-3)$ ;
  }
  for ( $y = 1$ ;  $buf[y - 1] \neq '\n'$ ;  $y++$ ) {
    if ( $y > maxy$ ) {
       $fprintf(stderr, "Sorry, \_the\_pattern\_should\_have\_at\_most\_ \%d\_columns! \n", maxy)$ ;
       $exit(-4)$ ;
    }
    if ( $buf[y - 1] \equiv '*'$ ) {
       $p[x][y] = 1$ ;
      if ( $y > ymax$ )  $ymax = y$ ;
      if ( $y < ymin$ )  $ymin = y$ ;
      if ( $x > xmax$ )  $xmax = x$ ;
      if ( $x < xmin$ )  $xmin = x$ ;
    } else if ( $buf[y - 1] \neq '.'$ ) {
       $fprintf(stderr, "Unexpected\_character\_ '%c' \_found\_in\_the\_pattern! \n", buf[y - 1])$ ;
       $exit(-5)$ ;
    }
  }
}

```

5* $\#define$ $pp(xx, yy) (((xx) < xmin \vee (yy) < ymin \vee (xx) > xmax \vee (yy) > ymax) ? 0 : 1)$

$\langle \text{If cell } (x, y) \text{ is obviously dead at time } t + 1, \text{ continue 5*} \rangle \equiv$

```

if ( $pp(x - 1, y - 1) + pp(x - 1, y) + pp(x - 1, y + 1) + pp(x, y - 1) + pp(x, y) + pp(x, y + 1) + pp(x + 1, y - 1) + pp(x + 1, y) + pp(x + 1, y + 1) < 3$ ) continue;

```

This code is used in section 2*.

6* Clauses are assembled in the *clause* array (surprise), where we put encoded literals.

The code for a literal is an unsigned 32-bit quantity, where the leading bit is 1 if the literal should be complemented. The next three bits specify the type of the literal (0 thru 7 for plain and A–G); the next three bits specify an integer k ; and the next bit is zero. That leaves room for two 12-bit fields, which specify x and y .

Type 0 literals have $k = 0$ for the ordinary xty code. However, the value $k = 1$ indicates that the time code should be for $t + 1$ instead of t . And $k = 2$ denotes a special “tautology” literal, which is always true. If the tautology literal is complemented, we omit it from the clause; otherwise we omit the entire clause. Finally, $k = 7$ denotes an auxiliary literal, used to avoid clauses of length 4.

Here’s a subroutine that outputs the current clause and resets the *clause* array.

```
#define taut (2 << 25)
#define sign (1<sub>U</sub> << 31)
⟨Subroutines 6*⟩ ≡
void outclause(void)
{
    register int c, k, x, y, p, t;
    for (p = 0; p < clauseptr; p++)
        if (clause[p] ≡ taut) goto done;
    for (p = 0; p < clauseptr; p++)
        if (clause[p] ≠ taut + sign) {
            if (clause[p] >> 31) printf("␣~"); else printf("␣");
            c = (clause[p] >> 28) & #7;
            k = (clause[p] >> 25) & #7;
            x = (clause[p] >> 12) & #fff;
            y = clause[p] & #fff;
            if (c ≡ 0) {
                if (x + x > mm + 1) x = mm + 1 - x;
                if (y + y > nn + 1) y = nn + 1 - y;
            }
            if (c) printf("%d%c%d%c%d", x, timecode[tt], y, c + '@', k);
            else if (k ≡ 7) printf("%d%c%dx", x, timecode[tt], y);
            else printf("%d%c%d", x, timecode[(tt + k) % r], y);
        }
    printf("\n");
done: clauseptr = 0;
}
```

See also sections 7*, 8, 9, 10, 11, 12, 14, and 15.

This code is used in section 2*.

7* And here’s another, which puts a type-0 literal into *clause*.

```
⟨Subroutines 6*⟩ +=
void applit(int x, int y, int bar, int k)
{
    if (k ≡ 0 ∧ pp(x, y) ≡ 0) clause[clauseptr++] = (bar ? 0 : sign) + taut;
    else clause[clauseptr++] = (bar ? sign : 0) + (k << 25) + (x << 12) + y;
}
```

8. The d and e subroutines are called for only one-fourth of all cell addresses (x, y) . Indeed, one can show that x is always odd, and that $y \bmod 4 < 2$.

Therefore we remember if we've seen (x, y) before.

Slight trick: If yy is not in range, we avoid generating the clause \bar{d}_k twice.

#define newlit(x, y, c, k) clause[clauseptr++] = (($c \ll 28$) + (($k \ll 25$) + (($x \ll 12$) + (y))

#define newcomplit(x, y, c, k) clause[clauseptr++] = sign + (($c \ll 28$) + (($k \ll 25$) + (($x \ll 12$) + (y))

⟨Subroutines 6*⟩ +≡

void d(int x , int y)

```
{
    register  $x1 = x - 1$ ,  $x2 = x$ ,  $yy = y + 1$ ;
    if (have_d[ $x$ ][ $y$ ]  $\neq$   $tt + 1$ ) {
        applit( $x1$ ,  $yy$ , 1, 0), newlit( $x$ ,  $y$ , 4, 1), outclause();
        applit( $x2$ ,  $yy$ , 1, 0), newlit( $x$ ,  $y$ , 4, 1), outclause();
        applit( $x1$ ,  $yy$ , 1, 0), applit( $x2$ ,  $yy$ , 1, 0), newlit( $x$ ,  $y$ , 4, 2), outclause();
        applit( $x1$ ,  $yy$ , 0, 0), applit( $x2$ ,  $yy$ , 0, 0), newcomplit( $x$ ,  $y$ , 4, 1), outclause();
        applit( $x1$ ,  $yy$ , 0, 0), newcomplit( $x$ ,  $y$ , 4, 2), outclause();
        if ( $yy \geq y_{min} \wedge yy \leq y_{max}$ ) applit( $x2$ ,  $yy$ , 0, 0), newcomplit( $x$ ,  $y$ , 4, 2), outclause();
        have_d[ $x$ ][ $y$ ] =  $tt + 1$ ;
    }
}
```

void e(int x , int y)

```
{
    register  $x1 = x - 1$ ,  $x2 = x$ ,  $yy = y - 1$ ;
    if (have_e[ $x$ ][ $y$ ]  $\neq$   $tt + 1$ ) {
        applit( $x1$ ,  $yy$ , 1, 0), newlit( $x$ ,  $y$ , 5, 1), outclause();
        applit( $x2$ ,  $yy$ , 1, 0), newlit( $x$ ,  $y$ , 5, 1), outclause();
        applit( $x1$ ,  $yy$ , 1, 0), applit( $x2$ ,  $yy$ , 1, 0), newlit( $x$ ,  $y$ , 5, 2), outclause();
        applit( $x1$ ,  $yy$ , 0, 0), applit( $x2$ ,  $yy$ , 0, 0), newcomplit( $x$ ,  $y$ , 5, 1), outclause();
        applit( $x1$ ,  $yy$ , 0, 0), newcomplit( $x$ ,  $y$ , 5, 2), outclause();
        if ( $yy \geq y_{min} \wedge yy \leq y_{max}$ ) applit( $x2$ ,  $yy$ , 0, 0), newcomplit( $x$ ,  $y$ , 5, 2), outclause();
        have_e[ $x$ ][ $y$ ] =  $tt + 1$ ;
    }
}
```

9. The f subroutine can't be shared quite so often. But we do save a factor of 2, because $x + y$ is always even.

⟨Subroutines 6*⟩ +≡

void f(int x , int y)

```
{
    register  $xx = x - 1$ ,  $y1 = y$ ,  $y2 = y + 1$ ;
    if (have_f[ $x$ ][ $y$ ]  $\neq$   $tt + 1$ ) {
        applit( $xx$ ,  $y1$ , 1, 0), newlit( $x$ ,  $y$ , 6, 1), outclause();
        applit( $xx$ ,  $y2$ , 1, 0), newlit( $x$ ,  $y$ , 6, 1), outclause();
        applit( $xx$ ,  $y1$ , 1, 0), applit( $xx$ ,  $y2$ , 1, 0), newlit( $x$ ,  $y$ , 6, 2), outclause();
        applit( $xx$ ,  $y1$ , 0, 0), applit( $xx$ ,  $y2$ , 0, 0), newcomplit( $x$ ,  $y$ , 6, 1), outclause();
        applit( $xx$ ,  $y1$ , 0, 0), newcomplit( $x$ ,  $y$ , 6, 2), outclause();
        if ( $xx \geq x_{min} \wedge xx \leq x_{max}$ ) applit( $xx$ ,  $y2$ , 0, 0), newcomplit( $x$ ,  $y$ , 6, 2), outclause();
        have_f[ $x$ ][ $y$ ] =  $tt + 1$ ;
    }
}
```

10. The g subroutine cleans up the dregs, by somewhat tediously locating the two neighbors that weren't handled by d , e , or f . No sharing is possible here.

```

⟨Subroutines 6*⟩ +≡
void g(int x, int y)
{
    register x1, x2, y1, y2;
    if (x & 1) x1 = x - 1, y1 = y, x2 = x + 1, y2 = y ⊕ 1;
    else x1 = x + 1, y1 = y, x2 = x - 1, y2 = y - 1 + ((y & 1) << 1);
    applit(x1, y1, 1, 0), newlit(x, y, 7, 1), outclause();
    applit(x2, y2, 1, 0), newlit(x, y, 7, 1), outclause();
    applit(x1, y1, 1, 0), applit(x2, y2, 1, 0), newlit(x, y, 7, 2), outclause();
    applit(x1, y1, 0, 0), applit(x2, y2, 0, 0), newcomplit(x, y, 7, 1), outclause();
    applit(x1, y1, 0, 0), newcomplit(x, y, 7, 2), outclause();
    applit(x2, y2, 0, 0), newcomplit(x, y, 7, 2), outclause();
}

```

11. Fortunately the b subroutine *can* be shared (since x is always odd), thus saving half of the sixteen clauses generated.

```

⟨Subroutines 6*⟩ +≡
void b(int x, int y)
{
    register j, k, xx = x, y1 = y - (y & 2), y2 = y + (y & 2);
    if (have_b[x][y] ≠ tt + 1) {
        d(xx, y1);
        e(xx, y2);
        for (j = 0; j < 3; j++)
            for (k = 0; k < 3; k++)
                if (j + k) {
                    if (j) newcomplit(xx, y1, 4, j); /*  $\bar{d}_j$  */
                    if (k) newcomplit(xx, y2, 5, k); /*  $\bar{e}_k$  */
                    newlit(x, y, 2, j + k); /*  $b_{j+k}$  */
                    outclause();
                    if (j) newlit(xx, y1, 4, 3 - j); /*  $d_{3-j}$  */
                    if (k) newlit(xx, y2, 5, 3 - k); /*  $e_{3-k}$  */
                    newcomplit(x, y, 2, 5 - j - k); /*  $\bar{b}_{5-j-k}$  */
                    outclause();
                }
        have_b[x][y] = tt + 1;
    }
}

```

12. The (unshared) c subroutine handles the other four neighbors, by working with f and g instead of d and e .

If $y = 0$, the overlap rules set $y1 = -1$, which can be problematic. I've decided to avoid this case by omitting f when it is guaranteed to be zero.

⟨Subroutines 6*⟩ +≡

```

void  $c(\text{int } x, \text{int } y)$ 
{
  register  $j, k, x1, y1$ ;
  if  $(x \& 1)$   $x1 = x + 2, y1 = (y - 1) \mid 1$ ;
  else  $x1 = x, y1 = y \& -2$ ;
   $g(x, y)$ ;
  if  $(x1 - 1 < xmin \vee x1 - 1 > xmax \vee y1 + 1 < ymin \vee y1 > ymax)$  ⟨Set  $c$  equal to  $g$  13⟩
  else {
     $f(x1, y1)$ ;
    for  $(j = 0; j < 3; j++)$ 
      for  $(k = 0; k < 3; k++)$ 
        if  $(j + k)$  {
          if  $(j)$   $newcomplit(x1, y1, 6, j)$ ; /*  $\bar{f}_j$  */
          if  $(k)$   $newcomplit(x, y, 7, k)$ ; /*  $\bar{g}_k$  */
           $newlit(x, y, 3, j + k)$ ; /*  $c_{j+k}$  */
           $outclause()$ ;
          if  $(j)$   $newlit(x1, y1, 6, 3 - j)$ ; /*  $f_{3-j}$  */
          if  $(k)$   $newlit(x, y, 7, 3 - k)$ ; /*  $g_{3-k}$  */
           $newcomplit(x, y, 3, 5 - j - k)$ ; /*  $\bar{c}_{5-j-k}$  */
           $outclause()$ ;
        }
      }
  }
}

```

13. ⟨Set c equal to g 13⟩ ≡

```

{
  for  $(k = 1; k < 3; k++)$  {
     $newcomplit(x, y, 7, k), newlit(x, y, 3, k), outclause()$ ; /*  $\bar{g}_k \vee c_k$  */
     $newlit(x, y, 7, k), newcomplit(x, y, 3, k), outclause()$ ; /*  $g_k \vee \bar{c}_k$  */
  }
   $newcomplit(x, y, 3, 3), outclause()$ ; /*  $\bar{c}_3$  */
   $newcomplit(x, y, 3, 4), outclause()$ ; /*  $\bar{c}_4$  */
}

```

This code is used in section 12.

14. Totals over all eight neighbors are then deduced by the a subroutine.

```

⟨Subroutines 6*⟩ +≡
void  $a(\text{int } x, \text{int } y)$ 
{
  register  $j, k, xx = x \mid 1;$ 
   $b(xx, y);$ 
   $c(x, y);$ 
  for ( $j = 0; j < 5; j++$ )
    for ( $k = 0; k < 5; k++$ )
      if ( $j + k > 1 \wedge j + k < 5$ ) {
        if ( $j$ )  $\text{newcomplit}(xx, y, 2, j);$  /*  $\bar{b}_j$  */
        if ( $k$ )  $\text{newcomplit}(x, y, 3, k);$  /*  $\bar{c}_k$  */
         $\text{newlit}(x, y, 1, j + k);$  /*  $a_{j+k}$  */
         $\text{outclause}();$ 
      }
  for ( $j = 0; j < 5; j++$ )
    for ( $k = 0; k < 5; k++$ )
      if ( $j + k > 2 \wedge j + k < 6 \wedge j * k$ ) {
        if ( $j$ )  $\text{newlit}(xx, y, 2, j);$  /*  $b_j$  */
        if ( $k$ )  $\text{newlit}(x, y, 3, k);$  /*  $c_k$  */
         $\text{newcomplit}(x, y, 1, j + k - 1);$  /*  $\bar{a}_{j+k-1}$  */
         $\text{outclause}();$ 
      }
}

```

15. Finally, as mentioned at the beginning, z' is determined from z , a_2 , a_3 , and a_4 .

I actually generate six clauses, not five, in order to stick to 3SAT.

```

⟨Subroutines 6*⟩ +≡
void  $zprime(\text{int } x, \text{int } y)$ 
{
   $\text{newcomplit}(x, y, 1, 4), \text{applit}(x, y, 1, 1), \text{outclause}();$  /*  $\bar{a}_4 \bar{z}'$  */
   $\text{newlit}(x, y, 1, 2), \text{applit}(x, y, 1, 1), \text{outclause}();$  /*  $a_2 \bar{z}'$  */
   $\text{newlit}(x, y, 1, 3), \text{applit}(x, y, 0, 0), \text{applit}(x, y, 1, 1), \text{outclause}();$  /*  $a_3 z \bar{z}'$  */
   $\text{newcomplit}(x, y, 1, 3), \text{newlit}(x, y, 1, 4), \text{applit}(x, y, 0, 1), \text{outclause}();$  /*  $\bar{a}_3 a_4 z'$  */
   $\text{applit}(x, y, 0, 7), \text{newcomplit}(x, y, 1, 2), \text{newlit}(x, y, 1, 4), \text{outclause}();$  /*  $x \bar{a}_2 a_4$  */
   $\text{applit}(x, y, 1, 7), \text{applit}(x, y, 1, 0), \text{applit}(x, y, 0, 1), \text{outclause}();$  /*  $\bar{x} \bar{z} z'$  */
}

```

16* In particular, this step rules out the case where everything is dead.

```

⟨Rule out smaller periods 16*⟩ ≡
for ( $k = 1; k < r; k++$ )
  if ( $(r \% k \equiv 0)$ ) { /* generation  $k$  shouldn't equal gen 0 */
    for ( $x = 1; x + x \leq mm + 1; x++$ )
      for ( $y = 1; y + y \leq nn + 1; y++$ ) {
         $\text{printf}(\text{"\%da\%d\%d\%c\%d\%da\%c\%d\n"}, x, y, x, \text{timecode}[k], y, x, \text{timecode}[k], y);$ 
         $\text{printf}(\text{"\%da\%d\%d\%c\%d\%da\%c\%d\n"}, x, y, x, \text{timecode}[k], y, x, \text{timecode}[k], y);$ 
      }
    for ( $x = 1; x + x \leq mm + 1; x++$ )
      for ( $y = 1; y + y \leq nn + 1; y++$ )  $\text{printf}(\text{"\%da\%c\%d"}, x, \text{timecode}[k], y);$ 
       $\text{printf}(\text{"\n"});$ 
  }
}

```

This code is used in section 2*.

17* Index.

The following sections were changed by the change file: 1, 2, 3, 5, 6, 7, 16, 17.

a: [14](#).
applit: [2*](#), [7*](#), [8](#), [9](#), [10](#), [15](#).
argc: [2*](#), [3*](#).
argv: [2*](#), [3*](#).
b: [11](#).
bar: [7*](#).
buf: [2*](#), [4](#).
c: [6*](#), [12](#).
clause: [2*](#), [6*](#), [7*](#), [8](#).
clauseptr: [2*](#), [6*](#), [7*](#), [8](#).
d: [8](#).
done: [6*](#).
e: [8](#).
exit: [3*](#), [4](#).
f: [9](#).
fgets: [4](#).
fprintf: [3*](#), [4](#).
g: [10](#).
have_b: [2*](#), [11](#).
have_d: [2*](#), [8](#).
have_e: [2*](#), [8](#).
have_f: [2*](#), [9](#).
j: [2*](#), [11](#), [12](#), [14](#).
k: [2*](#), [6*](#), [7*](#), [11](#), [12](#), [14](#).
main: [2*](#).
maxx: [2*](#), [4](#).
mxy: [2*](#), [4](#).
mm: [2*](#), [3*](#), [6*](#), [16*](#).
newcomplit: [8](#), [9](#), [10](#), [11](#), [12](#), [13](#), [14](#), [15](#).
newlit: [8](#), [9](#), [10](#), [11](#), [12](#), [13](#), [14](#), [15](#).
nn: [2*](#), [3*](#), [6*](#), [16*](#).
outclause: [2*](#), [6*](#), [8](#), [9](#), [10](#), [11](#), [12](#), [13](#), [14](#), [15](#).
p: [2*](#), [6*](#).
pp: [2*](#), [5*](#), [7*](#).
printf: [3*](#), [6*](#), [16*](#).
r: [2*](#).
sign: [6*](#), [7*](#), [8](#).
sscanf: [3*](#).
stderr: [3*](#), [4](#).
stdin: [2*](#), [4](#).
t: [6*](#).
taut: [6*](#), [7*](#).
timecode: [2*](#), [6*](#), [16*](#).
tt: [2*](#), [6*](#), [8](#), [9](#), [11](#).
x: [2*](#), [6*](#), [7*](#), [8](#), [9](#), [10](#), [11](#), [12](#), [14](#), [15](#).
xmax: [2*](#), [4](#), [5*](#), [9](#), [12](#).
xmin: [2*](#), [4](#), [5*](#), [9](#), [12](#).
xx: [5*](#), [9](#), [11](#), [14](#).
x1: [8](#), [10](#), [12](#).
x2: [8](#), [10](#).

⟨ If cell (x, y) is obviously dead at time $t + 1$, **continue** 5* ⟩ Used in section 2*.

⟨ Input the pattern 4 ⟩

⟨ Process the command line 3* ⟩ Used in section 2*.

⟨ Rule out smaller periods 16* ⟩ Used in section 2*.

⟨ Set c equal to g 13 ⟩ Used in section 12.

⟨ Subroutines 6*, 7*, 8, 9, 10, 11, 12, 14, 15 ⟩ Used in section 2*.

SAT-LIFE-GRIDQUAD-CYCLE

	Section	Page
Intro	1	1
Index	17	9