May 19, 2018 at 02:31

1*   Intro.   This program generates clauses for the transition relation from time t to time $t+1$ in Conway's Game of Life, for various values of t, as I'm trying to prove or disprove a certain conjecture.

Namely, it may be possible to set cells (x, y) for $x \leq 0$ and $y \leq 0$ (i.e., in the lower left quadrant) in such a way that cell (x, y) is reachable in $x + 2y$ steps when $0 \leq -x \leq y$, and in $2y + 2x$ steps when $x \geq 0$ and $y \geq 0$. Hopefully by seeing examples for small x and y I will have a handle on that conjecture.

The conjectured bounds agree with lower bounds that are readily proved. Hence the problem is to find matching upper bounds, if possible.

The command line should contain the coordinates x_0 and y_0 being tested.

When the conjectured bound is r, this program uses nested boards of sizes $(2r + 1) \times (2r + 1)$, $(2r - 1) \times (2r - 1)$, ..., 3×3, 1×1, centered on the cell (x, y) that we're trying to turn on. Many of the cells are known to be zero, because of the lower bounds; therefore we don't include them in the computation.

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 `A`k stands for the condition "at least k of the eight neighbors are alive." Similarly, `B`k stands for "at least k of the first four neighbors are alive," and `C`k 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

$$\bar{d}_1 b_1, \quad \bar{e}_1 b_1, \quad \bar{d}_2 b_2, \quad \bar{d}_1 \bar{e}_1 b_2, \quad \bar{e}_2 b_2, \quad \bar{d}_2 \bar{e}_1 b_3, \quad \bar{d}_1 \bar{e}_2 b_3, \quad \bar{d}_2 \bar{e}_2 b_4,$$
$$d_1 e_1 \bar{b}_1, \quad d_1 e_2 \bar{b}_2, \quad d_2 e_1 \bar{b}_2, \quad d_1 \bar{b}_3, \quad d_2 e_2 \bar{b}_3, \quad e_1 \bar{b}_3, \quad d_2 \bar{b}_4, \quad e_2 \bar{b}_4$$

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

$$\bar{a}_4 \bar{z}', \quad a_2 \bar{z}', \quad a_3 z \bar{z}', \quad \bar{a}_3 a_4 z', \quad \bar{a}_2 a_4 \bar{z} z'.$$

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* 100   /∗ maximum number of lines in the pattern supplied by *stdin* ∗/
#define *maxy* 100   /∗ 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 $f(x, y)$? ∗/
  int *tt*;   /∗ the time being considered ∗/
  int *x0*, *y0*;   /∗ the command-line parameters ∗/
  int *r*;   /∗ the conjectured bound ∗/
  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 $0 \le t \le 83$ ∗/
  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 ⟩
  $main(\textbf{int } argc, \textbf{char } *argv[\,])$
  {
    register int *j*, *k*, *x*, *y*;
    ⟨ Process the command line 3* ⟩;
    for ($tt = 0$; $tt < r$; $tt{+}{+}$) {
      $xmin = ymin = 1 + tt, xmax = ymax = r + r + 1 - tt$;
      for ($x = xmin + 1$; $x < xmax$; $x{+}{+}$)
        for ($y = ymin + 1$; $y < ymax$; $y{+}{+}$) {
          if ($bound(x, y) > tt + 1$) **continue**;
          $a(x, y)$;
          $zprime(x, y)$;
        }
    }
    $printf(\texttt{"\%d\%c\%d\textbackslash n"}, r + 1, timecode[tt], r + 1)$;   /∗ middle variable must be alive ∗/
  }

3* ⟨ Process the command line 3* ⟩ ≡

   if $(argc \neq 3 \vee sscanf(argv[1], \texttt{"\%d"}, \&x0) \neq 1 \vee sscanf(argv[2], \texttt{"\%d"}, \&y0) \neq 1)$ {

      $fprintf(stderr, \texttt{"Usage:\_\%s\_x0\_y0\textbackslash n"}, argv[0]);$

      $exit(-1);$

   }

   if $(y0 \leq 0)$ {

      $fprintf(stderr, \texttt{"The\_value\_of\_y0\_should\_be\_positive!\textbackslash n"});$

      $exit(-2);$

   }

   if $(x0 < -y0)$ {

      $fprintf(stderr, \texttt{"The\_value\_of\_x0\_should\_be\_at\_least\_-y0!\textbackslash n"});$

      $exit(-3);$

   }

   $r = (x0 > 0 \mathrel{?} 2 * (x0 + y0) : x0 + 2 * y0);$

   if $(r + r + 1 > maxx \vee r + r + 1 > maxy)$ {

      $fprintf(stderr, \texttt{"Recompile\_me:\_maxx\_and\_maxy\_must\_be\_at\_least\_\%d!\textbackslash n"}, r + r + 1);$

      $exit(-666);$

   }

   $printf(\texttt{"\textasciitilde\_sat-life-upper\_\%d\_\%d\textbackslash n"}, x0, y0);$

This code is used in section 2*.

4. ⟨ Input the pattern 4 ⟩ ≡

   for $(x = 1; \ ; \ x\texttt{++})$ {

      if $(\neg fgets(buf, maxy + 2, stdin))$ **break**;

      if $(x > maxx)$ {

         $fprintf(stderr, \texttt{"Sorry,\_the\_pattern\_should\_have\_at\_most\_\%d\_rows!\textbackslash n"}, maxx);$

         $exit(-3);$

      }

      for $(y = 1; \ buf[y-1] \neq \texttt{'\textbackslash n'}; \ y\texttt{++})$ {

         if $(y > maxy)$ {

            $fprintf(stderr, \texttt{"Sorry,\_the\_pattern\_should\_have\_at\_most\_\%d\_columns!\textbackslash n"}, maxy);$

            $exit(-4);$

         }

         if $(buf[y-1] \equiv \texttt{'*'})$ {

            $p[x][y] = 1;$

            if $(y > ymax) \ ymax = y;$

            if $(y < ymin) \ ymin = y;$

            if $(x > xmax) \ xmax = x;$

            if $(x < xmin) \ xmin = x;$

         } **else if** $(buf[y-1] \neq \texttt{'.'})$ {

            $fprintf(stderr, \texttt{"Unexpected\_character\_`\%c'\_found\_in\_the\_pattern!\textbackslash n"}, buf[y-1]);$

            $exit(-5);$

         }

      }

   }

5. **#define** $pp(xx, yy)$ $((xx) \geq 0 \wedge (yy) \geq 0 \mathrel{?} p[xx][yy] : 0)$

⟨ If cell (x, y) is obviously dead at time $t + 1$, **continue** 5 ⟩ ≡

   if $(pp(x-1, y-1) + pp(x-1, y) + pp(x-1, y+1) + pp(x, y-1) + p[x][y] + p[x][y+1] + pp(x+1,$

      $y-1) + p[x+1][y] + p[x+1][y+1] < 3)$ **continue**;

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* $(2 \ll 25)$
#define *sign* $(1_{\mathrm{U}} \ll 31)$

⟨ Subroutines 6 ⟩ ≡
  void *outclause*(**void**)
  {
    register int c, k, x, y, p;

    for $(p = 0;\ p < clauseptr;\ p{+}{+})$
      if $(clause[p] \equiv taut)$ **goto** *done*;
    for $(p = 0;\ p < clauseptr;\ p{+}{+})$
      if $(clause[p] \neq taut + sign)$ {
        if $(clause[p] \gg 31)$ *printf*(`"␣~"`); **else** *printf*(`"␣"`);
        $c = (clause[p] \gg 28)\ \&\ {}^{\#}7$;
        $k = (clause[p] \gg 25)\ \&\ {}^{\#}7$;
        $x = (clause[p] \gg 12)\ \&\ {}^{\#}\texttt{fff}$;
        $y = clause[p]\ \&\ {}^{\#}\texttt{fff}$;
        if (c) *printf*(`"%d%c%d%c%d"`, x, *timecode*[tt], y, $c + $ `'@'`, k);
        else if $(k \equiv 7)$ *printf*(`"%d%c%dx"`, x, *timecode*[tt], y);
        else *printf*(`"%d%c%d"`, x, *timecode*[$tt + k$], y);
      }
    printf(`"\n"`);
  done: *clauseptr* = 0;
  }

See also sections 7*, 8, 9, 10, 11, 12, 14, 15, and 16*.

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 $(k \equiv 0 \wedge bound(x, y) > tt)$ $clause[clauseptr{+}{+}] = (bar\ ?\ 0 : sign) + taut$;
    else $clause[clauseptr{+}{+}] = (bar\ ?\ sign : 0) + (k \ll 25) + (x \ll 12) + y$;
  }

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

$\langle$ Subroutines 6 $\rangle$ $+\equiv$

  void $d(\textbf{int } x, \textbf{int } y)$
  {
    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 $(yy \geq ymin \wedge yy \leq ymax)$ $applit(x2, yy, 0, 0), newcomplit(x, y, 4, 2), outclause( );$
      $have\_d[x][y] = tt + 1;$
    }
  }

  void $e(\textbf{int } x, \textbf{int } y)$
  {
    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 $(yy \geq ymin \wedge yy \leq ymax)$ $applit(x2, yy, 0, 0), newcomplit(x, y, 5, 2), outclause( );$
      $have\_e[x][y] = tt + 1;$
    }
  }

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.

$\langle$ Subroutines 6 $\rangle$ $+\equiv$

  void $f(\textbf{int } x, \textbf{int } y)$
  {
    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 $(xx \geq xmin \wedge xx \leq xmax)$ $applit(xx, y2, 0, 0), newcomplit(x, y, 6, 2), outclause( );$
      $have\_f[x][y] = tt + 1;$
    }
  }

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 $g(\textbf{int } x, \textbf{int } y)$

  {

    register $x1$, $x2$, $y1$, $y2$;

    if $(x \mathbin{\&} 1)$ $x1 = x - 1, y1 = y, x2 = x + 1, y2 = y \oplus 1$;

    else $x1 = x + 1, y1 = y, x2 = x - 1, y2 = y - 1 + ((y \mathbin{\&} 1) \ll 1)$;

    $applit(x1, y1, 1, 0), newlit(x, y, 7, 1), outclause(\,)$;

    $applit(x2, y2, 1, 0), newlit(x, y, 7, 1), outclause(\,)$;

    $applit(x1, y1, 1, 0), applit(x2, y2, 1, 0), newlit(x, y, 7, 2), outclause(\,)$;

    $applit(x1, y1, 0, 0), applit(x2, y2, 0, 0), newcomplit(x, y, 7, 1), outclause(\,)$;

    $applit(x1, y1, 0, 0), newcomplit(x, y, 7, 2), outclause(\,)$;

    $applit(x2, y2, 0, 0), newcomplit(x, y, 7, 2), outclause(\,)$;

  }

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

⟨ Subroutines 6 ⟩ +≡

  void $b(\textbf{int } x, \textbf{int } y)$

  {

    register j, k, $xx = x$, $y1 = y - (y \mathbin{\&} 2)$, $y2 = y + (y \mathbin{\&} 2)$;

    if $(have\_b[x][y] \ne tt + 1)$ {

      $d(xx, y1)$;

      $e(xx, y2)$;

      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(int x, int y)
{
  register j, k, x1, y1;
  if (x & 1) x1 = x + 2, y1 = (y − 1) | 1;
  else x1 = x, y1 = y & −2;
  g(x, y);
  if (x1 − 1 < xmin ∨ x1 − 1 > xmax ∨ y1 + 1 < ymin ∨ 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 $a(\textbf{int } x, \textbf{int } y)$

  {

    register j, k, $xx = x \mid 1$;

    $b(xx, y)$;

    $c(x, y)$;

    for $(j = 0;\ j < 5;\ j\mathord{+}\mathord{+})$

      for $(k = 0;\ k < 5;\ k\mathord{+}\mathord{+})$

        if $(j + k > 1 \wedge j + k < 5)$ {

          if (j) $newcomplit(xx, y, 2, j)$;   /* $\bar{b}_j$ */

          if (k) $newcomplit(x, y, 3, k)$;   /* $\bar{c}_k$ */

          $newlit(x, y, 1, j + k)$;   /* a_{j+k} */

          $outclause(\,)$;

        }

    for $(j = 0;\ j < 5;\ j\mathord{+}\mathord{+})$

      for $(k = 0;\ k < 5;\ k\mathord{+}\mathord{+})$

        if $(j + k > 2 \wedge j + k < 6 \wedge j * k)$ {

          if (j) $newlit(xx, y, 2, j)$;   /* b_j */

          if (k) $newlit(x, y, 3, k)$;   /* c_k */

          $newcomplit(x, y, 1, j + k - 1)$;   /* $\bar{a}_{j+k-1}$ */

          $outclause(\,)$;

        }

  }

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 $zprime(\textbf{int } x, \textbf{int } y)$

  {

    $newcomplit(x, y, 1, 4), applit(x, y, 1, 1), outclause(\,)$;   /* $\bar{a}_4 \bar{z}'$ */

    $newlit(x, y, 1, 2), applit(x, y, 1, 1), outclause(\,)$;   /* $a_2 \bar{z}'$ */

    $newlit(x, y, 1, 3), applit(x, y, 0, 0), applit(x, y, 1, 1), outclause(\,)$;   /* $a_3 z \bar{z}'$ */

    $newcomplit(x, y, 1, 3), newlit(x, y, 1, 4), applit(x, y, 0, 1), outclause(\,)$;   /* $\bar{a}_3 a_4 z'$ */

    $applit(x, y, 0, 7), newcomplit(x, y, 1, 2), newlit(x, y, 1, 4), outclause(\,)$;   /* $x \bar{a}_2 a_4$ */

    $applit(x, y, 1, 7), applit(x, y, 1, 0), applit(x, y, 0, 1), outclause(\,)$;   /* $\bar{x} \bar{z} z'$ */

  }

16.* In this variation of the program, I compute the known lower bounds. At time t, only the entries of p that are $\leq t$ are considered potentially alive.

I've been thinking "rows and columns" instead of Cartesian coordinates, so the notation is a bit schizophrenic■ here. An x value in the user interface corresponds to column $x + c$, where $c = 1 + r - x_0$; and a y value corresponds to row $d - y$, where $d = 1 + r + y_0$. (Hence in particular, cell $(0,0)$ corresponds to column c of row d. Since $r \geq 2y_0 - x_0$, we have $c \geq 3$.)

$\langle$ Subroutines 6 $\rangle$ $+\equiv$

```
int ff (int x, int y)
{
    if (x ≤ 0 ∧ y ≤ 0) return 0;
    if (y < 0) return ff (y, x);
    if (x ≤ −y) return y;
    if (x ≤ 0) return x + y + y;
    return x + x + y + y;
}
int bound (int xx, int yy)
{
    return ff (yy − (1 + r − x0), (1 + r + y0) − xx);
}
```

17*   Index.

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

SAT-LIFE-UPPER