CORRIGENDUM TO "POLYOMINO ENUMERATION RESULTS. (PARKIN ET AL., SIAM FALL MEETING 1967)"

RICHARD J. MATHAR

Abstract. This work provides a Java program which constructs free polyominoes of size n sorted by width and height of the convex hull (i.e., its rectangular bounding box). The results correct counts for 15-ominoes published in the 1967 proceedings of the SIAM Fall Meeting, and extend them to 16-ominoes and partially to even larger polyominoes.

1. Free polyominoes

1.1. **Nomenclature.** Polyominoes are sets of n edge-connected squares, which means each of the squares can be reached from any other square of the set by a path that connects nearest neighbours (adjacent squares to the North, East, South and West) which all are members of the set.

Fixed polyominoes are polyominoes which can be mapped onto each other by translating them rigidly along the horizontal and/or vertical axes of the underlying square grid. They can be enumerated for example with the aid of a transfer matrix method [3][2, A001168][1].

A set of fixed polyominoes that can be mapped onto each other by further symmetry operations of rotations by multiples of 90 degrees and/or flips along the horizontal or vertical axes is a *free* polyomino. The convex hull (or bounding box) of a polyomino is the largest connected rectangular section of the underlying square grid such that each row and column of the hull contains at least one square of the polyomino. We are interested in classifying all n-ominoes by the height h and width w of their convex hull.

Definition 1. *(Free Polyominoes Classified by Bounding Box) $P_{h\times w}(n)$ denotes the number of free polyominoes of size n that fit into a tight $h \times w$ bounding box.*

The number of free polyominoes does not change if the bounding rectangle is rotated by multiples of 90 degrees (such that the roles of width and height are swapped):

$$P_{h\times w}(n) = P_{w\times h}(n). \tag{1}$$

There is only one free block polyomino where all cells within the bounding box of area hw are occupied:

$$P_{h\times w}(hw) = 1. \tag{2}$$

Date: May 23, 2019.
2010 *Mathematics Subject Classification.* Primary 05B50; Secondary 05A18, 51-04.
Key words and phrases. Free Polyomino, Enumeration, Convex Hull.

There is no polyomino if the number of cells in the bounding box is smaller than the size of the polyomino:

$$P_{h\times w}(n) = 0, \text{ if } n > hw. \tag{3}$$

There is no polyomino if the width and height of the bounding box are too large:

$$P_{h\times w}(n) = 0, \text{ if } h + w - 1 > n. \tag{4}$$

Definition 2. *(Free Polyominoes) The total number of free n-ominoes is*

$$P(n) \equiv \sum_{h=1}^{n} \sum_{w=1}^{h} P_{h\times w}(n). \tag{5}$$

1.2. **Read's Results.** For $n \le 10$ and $w \ne h$ these numbers have been tabulated by Read [8]. Note that his tables $b(q, n)$ for width 2, $c(q, n)$ for width 3 and $d(q, n)$ for width 4 are not reducing the fixed polyominoes to free polyominoes if width and height are the same [2, A259088], because he only applied a symmetry group of order 4 in all cases. One needs to compare our results with his $z_w(n)$ in cases where $w = h$.

As observed by Klarner [4], the actual discrepancy with his data is for $P_{5\times5}(10) = 529$ where Read reports only $z_5(10) = 340$, such that our total is $P(10) = 4655$, not his 4466.

2. Algorithm

The construction of free n-ominoes by the program in the Appendix includes the following steps:

2.1. **Compositions Look-up Tables.** Each polyomino is represented by a $h \times w$ matrix of zeros and ones, where zeros represent unoccupied and ones occupied squares, respectively. For a given n and a given height $h \le n$ of the bounding box, the matrix row sums are a rough classification of the n-ominoes. A list of the compositions of n into h parts of minimum size 1 and maximum size w is compiled, where the i-th part is the sum of the ith row of the binary matrix. The lower limit is 1 because the connectivity of the polyominoes requires at least one occupied square in each row, and the upper limit equals the width. Because free polyominoes are unchanged by flipping the rows along the middle axis, compositions are discarded which are lexicographically larger than their reversed associates. For each of these row sums (from 1 up to w) we also keep a list of the $2^w - 1$ possible bitsets (compositions into w parts that are either 0 or 1), representing a single row of at least 1 and at most w 1's in the binary matrix.

2.2. **Row-by-row Stacking.** In a outer loop over the compositions, the first row of the binary matrix is any of the bitsets (inner loop) compatible with the first part of the composition. The other rows are recursively filled with bitsets with a sum equal to the associated part of the composition, and requiring that at least one of the squares of the row has a common edge with a square of the previous row to ensure that adjacent rows are edge-connected. [The bit-wise AND-operation between adjacent rows must not be zero.]

2.3. **Reduction by Symmetry.** As the last row of the matrix has been filled with a bitset, we have essentially constructed a candidate of a fixed n-omino. The program checks first that the set of squares is connected, because that is not guaranteed by the stacking method. [The growth may have lead to separated columns.] In a loop over the 4 (or 8) symmetry operations of flips and rotations, a lexicographically smallest representative of the polyomino is selected, a free polyomino. This is compared with the members in the set of free $h \times w$ n-polyominoes constructed so far, and added to the set if it is "new."

3. Results

The numerical results are summarized in the following table. Each entry has one of two formats:

- Three positive integer numbers n, h and w, a colon and $P_{h\times w}(n)$. These are the size of the free n-omino, the height and width of the bounding box, and the number of free n-ominoes fitting in that bounding box.
- One positive integer number n, a colon and $P(n)$. This is the size of the n-omino and the number of free n-ominoes of that size. This entry is absent for $n \geq 17$ because some of the $P_{h\times w}(n)$ have not yet been computed. For all cases computed, $P(n)$ matches the results in the literature [2, A000105][6], which shows the integrity of the program.

For $n \leq 14$ agree with the published table [7]; for $n = 15$ they correct their numbers, and for $n > 15$ they seem to be new.

```
1 1 1 : 1          1 : 1

2 2 1 : 1          2 : 1

3 3 1 : 1          3 2 2 : 1          3 : 2

4 4 1 : 1          4 2 2 : 1          4 3 2 : 3          4 : 5

5 5 1 : 1          5 3 2 : 2          5 4 2 : 3          5 3 3 : 6
5 : 12

6 6 1 : 1          6 3 2 : 1          6 4 2 : 6          6 5 2 : 5
6 3 3 : 7          6 4 3 : 15         6 : 35

7 7 1 : 1          7 4 2 : 2          7 5 2 : 11         7 6 2 : 5
7 3 3 : 7          7 4 3 : 39         7 5 3 : 25         7 4 4 : 18
7 : 108

8 8 1 : 1          8 4 2 : 1          8 5 2 : 10         8 6 2 : 19
8 7 2 : 7          8 3 3 : 3          8 4 3 : 59         8 5 3 : 96
8 6 3 : 35         8 4 4 : 77         8 5 4 : 61         8 : 369

9 9 1 : 1          9 5 2 : 3          9 6 2 : 22         9 7 2 : 28
9 8 2 : 7          9 3 3 : 1          9 4 3 : 42         9 5 3 : 210
9 6 3 : 188        9 7 3 : 49         9 4 4 : 181        9 5 4 : 383
9 6 4 : 97         9 5 5 : 73         9 : 1285
```

```
10 10 1 : 1       10 5 2 : 1        10 6 2 : 15       10 7 2 : 52
10 8 2 : 40       10 9 2 : 9        10 4 3 : 21       10 5 3 : 255
10 6 3 : 550      10 7 3 : 332      10 8 3 : 63       10 4 4 : 266
10 5 4 : 1304     10 6 4 : 822      10 7 4 : 155      10 5 5 : 529
10 6 5 : 240      10 : 4655

11 11 1 : 1       11 6 2 : 3        11 7 2 : 45       11 8 2 : 90
11 9 2 : 53       11 10 2 : 9       11 4 3 : 4        11 5 3 : 212
11 6 3 : 954      11 7 3 : 1231     11 8 3 : 529      11 9 3 : 81
11 4 4 : 251      11 5 4 : 2847     11 6 4 : 3548     11 7 4 : 1551
11 8 4 : 220      11 5 5 : 2413     11 6 5 : 2366     11 7 5 : 410
11 6 6 : 255      11 : 17073

12 12 1 : 1       12 6 2 : 1        12 7 2 : 21       12 8 2 : 119
12 9 2 : 158      12 10 2 : 69      12 11 2 : 11      12 4 3 : 1
12 5 3 : 103      12 6 3 : 1184     12 7 3 : 2800     12 8 3 : 2406
12 9 3 : 800      12 10 3 : 99      12 4 4 : 168      12 5 4 : 4441
12 6 4 : 10323    12 7 4 : 8239     12 8 4 : 2680     12 9 4 : 313
12 5 5 : 7375     12 6 5 : 13161    12 7 5 : 4738     12 8 5 : 646
12 6 6 : 2835     12 7 6 : 908      12 : 63600

13 13 1 : 1       13 7 2 : 4        13 8 2 : 73       13 9 2 : 257
13 10 2 : 238     13 11 2 : 86      13 12 2 : 11      13 5 3 : 33
13 6 3 : 964      13 7 3 : 4634     13 8 3 : 6818     13 9 3 : 4313
13 10 3 : 1142    13 11 3 : 121     13 4 4 : 66       13 5 4 : 5008
13 6 4 : 21995    13 7 4 : 29442    13 8 4 : 16821    13 9 4 : 4327
13 10 4 : 415     13 5 5 : 17041    13 6 5 : 51133    13 7 5 : 30998
13 8 5 : 8683     13 9 5 : 979      13 6 6 : 18533    13 7 6 : 11952
13 8 6 : 1553     13 7 7 : 950      13 : 238591

14 14 1 : 1       14 7 2 : 1        14 8 2 : 28       14 9 2 : 237
14 10 2 : 505     14 11 2 : 360     14 12 2 : 106     14 13 2 : 13
14 5 3 : 6        14 6 3 : 546      14 7 3 : 5497     14 8 3 : 14182
14 9 3 : 14722    14 10 3 : 7171    14 11 3 : 1580    14 12 3 : 143
14 4 4 : 20       14 5 4 : 4168     14 6 4 : 36035    14 7 4 : 79155
14 8 4 : 71742    14 9 4 : 31576    14 10 4 : 6634    14 11 4 : 551
14 5 5 : 30320    14 6 5 : 153122   14 7 5 : 143230   14 8 5 : 65236
14 9 5 : 14894    14 10 5 : 1415    14 6 6 : 86974    14 7 6 : 89212
14 8 6 : 23215    14 9 6 : 2555     14 7 7 : 13402    14 8 7 : 3417
14 : 901971

15 15 1 : 1       15 8 2 : 4        15 9 2 : 119      15 10 2 : 591
15 11 2 : 895     15 12 2 : 498     15 13 2 : 127     15 14 2 : 13
15 5 3 : 1        15 6 3 : 187      15 7 3 : 4745     15 8 3 : 22011
15 9 3 : 36920    15 10 3 : 28762   15 11 3 : 11304   15 12 3 : 2107
15 13 3 : 169     15 4 4 : 3        15 5 4 : 2439     15 6 4 : 45748
15 7 4 : 167354   15 8 4 : 230998   15 9 4 : 156007   15 10 4 : 55084
```

```
15 11 4 : 9751      15 12 4 : 698       15 5 5 : 42670      15 6 5 : 366832
15 7 5 : 517302     15 8 5 : 348090     15 9 5 : 126496     15 10 5 : 24214
15 11 5 : 1991      15 6 6 : 323331     15 7 6 : 483329     15 8 6 : 193911
15 9 6 : 42154      15 10 6 : 3965      15 7 7 : 111296     15 8 7 : 54983
15 9 7 : 6003       15 8 8 : 3473       15 : 3426576

16 16 1 : 1         16 8 2 : 1          16 9 2 : 36         16 10 2 : 429
16 11 2 : 1353      16 12 2 : 1493      16 13 2 : 690       16 14 2 : 151
16 15 2 : 15        16 6 3 : 47         16 7 3 : 2833       16 8 3 : 26097
16 9 3 : 70760      16 10 3 : 84925     16 11 3 : 52245     16 12 3 : 16997
16 13 3 : 2752      16 14 3 : 195       16 4 4 : 1          16 5 4 : 1008
16 6 4 : 45289      16 7 4 : 285375     16 8 4 : 594488     16 9 4 : 585305
16 10 4 : 310890    16 11 4 : 91127     16 12 4 : 13852     16 13 4 : 885
16 5 5 : 47344      16 6 5 : 720244     16 7 5 : 1524442    16 8 5 : 1459163
16 9 5 : 763678     16 10 5 : 229573    16 11 5 : 37708     16 12 5 : 2715
16 6 6 : 991660     16 7 6 : 2097534    16 8 6 : 1182179    16 9 6 : 390229
16 10 6 : 72565     16 11 6 : 5985      16 7 7 : 675981     16 8 7 : 500827
16 9 7 : 105600     16 10 7 : 10001     16 8 8 : 59728      16 9 8 : 12859
16 : 13079255

17 17 1 : 1         17 9 2 : 5          17 10 2 : 172       17 11 2 : 1248
17 12 2 : 2709      17 13 2 : 2343      17 14 2 : 902       17 15 2 : 176
17 16 2 : 15        17 6 3 : 6          17 7 3 : 1173       17 8 3 : 22883
17 9 3 : 106490     17 10 3 : 193669    17 11 3 : 177886    17 12 3 : 89146
17 13 3 : 24660     17 14 3 : 3504      17 15 3 : 225       17 4 4 : 0
17 5 4 : 271        17 6 4 : 34112      17 7 4 : 395338     17 8 4 : 1256623
17 9 4 : 1764700    17 10 4 : 1331013   17 11 4 : 577936    17 12 4 : 143749
17 13 4 : 19119     17 14 4 : 1085      17 5 5 : 41330      17 6 5 : 1168734
17 6 6 : 2567828    17 8 8 : 587349     17 9 8 : 241977     17 10 8 : 22827
17 9 9 : 13006

18 18 1 : 1         18 9 2 : 1          18 10 2 : 45        18 11 2 : 720
18 12 2 : 3192      18 13 2 : 5097      18 14 2 : 3531      18 15 2 : 1180
18 16 2 : 204       18 17 2 : 17        18 6 3 : 1          18 7 3 : 324
18 8 3 : 14761      18 9 3 : 124513     18 10 3 : 353037    18 11 3 : 471268
18 12 3 : 345168    18 13 3 : 144905    18 14 3 : 34644     18 15 3 : 4396
18 16 3 : 255       18 4 4 : 0          18 5 4 : 55         18 6 4 : 19282
18 7 4 : 444276     18 8 4 : 2217704    18 5 5 : 27764      18 6 5 : 1574307

19 19 1 : 1         19 9 2 : 0          19 10 2 : 5         19 11 2 : 249
19 12 2 : 2356      19 13 2 : 7235      19 14 2 : 8859      19 15 2 : 5113
19 16 2 : 1482      19 17 2 : 233       19 18 2 : 17        19 6 3 : 0
19 7 3 : 64         19 8 3 : 6730       19 9 3 : 112111     19 10 3 : 514669
19 11 3 : 1007794   19 12 3 : 1043651   19 13 3 : 629639    19 14 3 : 225722
19 15 3 : 47445     19 16 3 : 5413      19 17 3 : 289
```

The partial sums $\sum_{h \geq w} P_{h \times w}(n)$ are summarized in Table 1. A closed-form formula for the values 1, 1, 6, 18, 73, 255,... on the diagonal is known [2, A057051][5].

$n\backslash w$	1	2	3	4	5	6	7	8	9	$P(n)$
1	1									1
2	1									1
3	1	1								2
4	1	4								5
5	1	5	6							12
6	1	12	22							35
7	1	18	71	18						108
8	1	37	193	138						369
9	1	60	490	661	73					1285
10	1	117	1221	2547	769					4655
11	1	200	3011	8417	5189	255				17073
12	1	379	7393	26164	25920	3743				63600
13	1	669	18025	78074	108834	32038	950			238591
14	1	1250	43847	229881	408217	201956	16819			901971
15	1	2247	106206	668082	1427595	1046690	172282	3473		3426567
16	1	4168	256851	1928220	4784867	4740152	1292409	72587		13079255
17	1	7570	619642	5523946				852153	13006	50107909
18	1	13987	1493272							192622052
19	1	25549	3593527							

TABLE 1. The number of free n-ominoes with a bounding box of short edge w.

The sums $\sum_{w\ge 1}^{n} P_{w\times w}(n)$ for the free polynomials with a square bounding box have their their own OEIS entry [2, A259088].

APPENDIX A. OVERVIEW OF THE JAVA IMPLEMENTATION

The three Java classes that follow are compiled as usual with

```
javac -cp . Composit.java FreePoly.java FreePolySet.java
```

The main program is called with

```
java -cp . FreePolySet [-v] [-f] [-w #] n
```

where the last argument is the size (number of cells) of the polyomino, a positive integer.

The option `-v` lets the program print one (0,1)-matrix for each free polyomino that is constructed.

The option `-f` lets the program handle *fixed* polyominoes without reduction for the symmetry groups [2, A001168,A308359].

The option `-w` followed by a positive integer number lets the program consider only a bounding box of a specific width; by default the program will execute a sequential loop over all widths. This option supports parallelization of the computations on the operating system level by calling it more than once at the same time for different widths.

The class `Composit` constructs compositions of some number n into k parts given lower and upper bounds for the size of each part. This is done by standard recursion and filling the vector of parts left-to-right.

The class `FreePoly` represents a binary matrix of zeros and ones with given height (number of rows) and width (number of columns). It has member functions that rotate and/or flip the cells and a member function to select from these variants one normalized view of the free n-omino.

The class `FreePolySet` is the main callable function which first selects the size n of the polyominoes and the height and width of the bounding box, runs the double loop over compositions of n into h parts and bitsets with w parts (which amounts essentially to explicit construction of roughly a quarter of all fixed n-ominoes), and copies the free polyomino representatives into a set of eventually $P(n)$ binary matrices.

Appendix B. Source code of Composit.java

```
/** @file
* A class which generates the compositions of an integer into a fixed number of parts.
* @author R. J. Mathar
*/

import java.util.* ;
import java.lang.* ;

/**
* @brief The set of compositions of some fixed positive integer.
* @since 2019-05-11
*/
public class Composit
{
    /** the sum of the parts
    */
    int n ;

    /** the number of parts
    */
    int k ;

    /** the lowest size a part may have
    */
    int minPart ;

    /** the largest size a part may have
    */
    int maxPart ;

    /** The compositions to be generated.
    * Each composition is represented as a 1-dimensionsal array
    * of k numbers in the range minPart..maxPart and sum n.
    */
    Vector<int[]> comps;

    /**
    * Constructor defining the integer to be partitioned
    * @param n The sum of the parts
    * @param k The number of the parts
    * @param minP The smallest size any part may have.
    * @param maxP The largest size any part may have.
    * @since 2019-05-11
    */
    public Composit(int n, int k, int minP, int maxP)
    {
        this.n = n ;
        this.k = k ;
        minPart = minP ;
        maxPart = maxP ;
        /* the initially empty set of compositions.
        */
        comps= new Vector<int[]>() ;
```

```
    /* Generate the vector comps[] if basic requirements are met.
    *  Each part is >= minPart, so the total is >=k*minPart.
    *  Each part is <= maxPart, so the total is <=k*maxPart.
    */
    if ( k*minPart <= n && k*maxPart >= n)
        comps = generate( new int[0],n) ;
} /* ctor */

/**
* @return The number of compositions.
* Because the compositions are all generated with the ctor,
*  this number is available right away.
*/
public int size()
{
    return comps.size() ;
} /* size */

/** generated recursively the compositions of n
* @param given The initial sublist of parts already fixed.
* @param nResid The sum over the elements not yet in given[].
* @return The partitions represented as vectors of length k.
*/
private Vector<int[]> generate(int[] given, int nResid)
{
    /* the compositions that can be generated;
    * The result of this subroutine
    */
    Vector<int[]> subcomp = new Vector<int[]>() ;

    if ( nResid < 0 || given.length > k)
    {
        /* prefixed parts not compatible with requirements;
        * return with the empty set.
        */
        return subcomp;
    }

    if ( given.length == k)
    {
        if ( nResid  ==0 )
            subcomp.add(given.clone()) ;
        /* Return a vector of 0 or 1 elements composing n.
        */
        return subcomp;
    }

    /* here given.length < k and nResid >=0
    */
    if ( given.length == k-1)
    {
        /* one final part to  be appended to the given[].
        * need a part of the size nResid to fill up to
        * to n
        */
        if (nResid >= minPart && nResid <= maxPart )
        {
            int[] c = new int[k] ;
            for(int pi =0 ; pi < c.length ; pi++)
                c[pi] = (pi < given.length) ? given[pi] : nResid ;
            subcomp.add(c) ;
        }
    }
    else
    {
        int[] c = new int[given.length+1] ;
        for(int pi=0 ; pi < given.length ; pi++)
            c[pi] = given[pi] ;

        /* 2 or more parts to  be appended; number of missing
        * parts is k-given.length, each part >=minPart. Let p be the
        * part to be appended next. The minimum total of the unassigned
```

```
            * parts  is p+minPart*(k-given.length-1). This value must stay <= nResid.
            * p <= nResid -minPart*(k-given.length-1).
            * The maximum total of the unassigned
            * parts  is p+maxPart*(k-given.length-1); this value must stay >=nResid
            * p >= nResid-maxPart*(k-given.length-1).
            */
            final int nextmin = Math.max(minPart, nResid-maxPart*(k-given.length-1)) ;
            final int nextmax = Math.min(maxPart, nResid-minPart*(k-given.length-1)) ;
            for(int p = nextmin ; p <= nextmax ; p++)
            {
                /* fill in the last integer into the parts list */
                c[given.length] = p ;
                final Vector<int[]> iters = generate(c,nResid-p) ;
                subcomp.addAll(iters) ;
            }
        }

        return subcomp ;
    } /* generate */

    /** Compare two integer vectors element-wise left to right.
    * If the two vectors have differnt length, the longer one is considered larger.
    * If the two vectors have the same length, the lexicographic comparison
    *  (comparing elements at index 0, 1, 2..) is executed. The vector
    *  which first has a larger element than the other is the larger vector.
    * @return -1, 0 or +1 if left is considered smaller than, equal to or larger than right.
    */
    public static int compareTo(final int[] left, final int[] right)
    {
        if ( left.length > right.length)
            return 1;
        else if ( left.length < right.length)
            return -1;
        else
        {
            for(int i=0 ; i < left.length ; i++)
            {
                if ( left[i] > right[i])
                    return 1;
                else if ( left[i] < right[i])
                    return -1 ;
            }
            return 0 ;
        }

    } /* compareTo */


    /** Reverse the integers in a vector
    * @param arg The initial vector.
    * @param return The initial vector where arg[i] has been swapped with arg[length-1-i].
    */
    public static int[] reverse(final int[] arg)
    {
        int[] rev =new int[arg.length] ;
        for(int i=0 ; i < arg.length ; i++)
            rev[i] = arg[arg.length-1-i] ;
        return rev ;
    } /* reverse */


} /* class Composit */
```

Appendix C. Source code of FreePoly.java

```
/** @file
* A n-omino with a r times c bounding box.
* @author R. J. Mathar
*/

import java.util.* ;
import java.lang.* ;
```

```
/**
* @brief A free n-omino with a tight bound box of r rows and c columns.
* @since 2019-05-11
*/
public class FreePoly
{
    /** the sum of the parts
    */
    int n ;

    /** the number of rows
    */
    int rows ;

    /** the number of columns
    */
    int cols ;

    /** The array of zeros and ones for each cell indexed by row and column
    */
    byte[][] bits ;


    /**
    * Constructor with a predefined n-omino.
    * @param zeroone The array of the zeros and ones.
    * @param freep If true, construct free polynomios.
    *  That means store a representation that may be rotated/flipped.
    * @param Read
    * @since 2019-05-11
    */
    public FreePoly(final byte[][] zeroone, boolean freep, int Read)
    {
        rows = zeroone.length ;
        if ( rows > 0 )
            cols = zeroone[0].length ;
        else
            cols = 0 ;
        bits = new byte[rows][cols] ;
        n=0 ;
        /* clone the elements of zeroone (which may be modified later
        * by the calling program)
        */
        for (int r=0 ; r < rows ; r++)
        for (int c=0 ; c < cols ; c++)
        {
            bits[r][c] = zeroone[r][c] ;
            n += bits[r][c] ;
        }

        if (freep)
            reduce(Read) ;
    } /* ctor */

    /** Construct the rotated and flipped versions. Retain only one.
    * @param Read
    */
    private void reduce(int Read)
    {
        /* if rows <> cols, compare this byte arry with the
        * three varians of flipped x, flipped y and rotated by 180 (group of order4).
        * If rows = cols, compare with the full D_8 group of order 8 by
        * including rotations by 90 or 270 degrees. piv is the pivotal variant
        * which is "smallest" in all the rotated/flipped variants.
        */
        byte[][] piv = bits ;
        byte[][] r90 = rot90(bits) ;
        byte[][] r180 = rot90(r90) ;
        byte[][] fpiv = flipx(bits) ;
        byte[][] fpiv180 = flipx(r180) ;
        if ( compareTo(r180,piv) < 0 )
```

```
        piv = r180 ;
    if ( compareTo(fpiv,piv) < 0 )
        piv = fpiv ;
    if ( compareTo(fpiv180,piv) < 0 )
        piv = fpiv180 ;
    if ( rows == cols && Read < 0)
    {
        /* consider 4 more versions if the matrix is square and
        * Read's table is not to be reproduced
        */
        if ( compareTo(r90,piv) < 0 )
            piv = r90 ;

        byte[][] r270 = rot90(r180) ;
        if ( compareTo(r270,piv) < 0 )
            piv = r270 ;

        byte[][] fpiv90 = flipx(r90) ;
        if ( compareTo(fpiv90,piv) < 0 )
            piv = fpiv90 ;

        byte[][] fpiv270 = flipx(r270) ;
        if ( compareTo(fpiv270,piv) < 0 )
            piv = fpiv270 ;
    }
    /* replace the representation by the "smallest" one.
    * Sum of parts, row and col are not changed by this representation.
    */
    bits = piv ;
} /* reduce */


/** Define a lexicographic order of 2D byte arrays by comparing them row by row
* @param left The first array to be considered.
*   Must be rectangular (must have the same number of elements in each row).
* @param right The second array to be considered.
*   Must be rectangular (must have the same number of elements in each row).
* @return a value of -1, 0 or +1 if left is considere to  be smaller than, equal to or larger than right.
*/
private static int compareTo( final byte[][] left, final byte[][] right)
{
    if ( left.length > right.length)
        return 1 ;
    else if ( left.length < right.length)
        return -1 ;
    else if ( left.length == 0 )
        return 0 ;
    else
    {
        if ( left[0].length > right[0].length)
            return 1;
        else if ( left[0].length < right[0].length)
            return -1;
        else if ( left[0].length == 0)
            return 0;
        else
        {
            final int rows =left.length ;
            final int cols =left[0].length ;
            for(int r=0 ; r < rows ; r++)
            for(int c=0 ; c < cols ; c++)
            {
                if ( left[r][c] > right[r][c])
                    return 1 ;
                else if ( left[r][c] < right[r][c])
                    return -1 ;
            }
            return 0 ;
        }
    }
}
```

```
    /** Define a lexicograph order of fixed n-ominoes by comparing their binary matrix representations
    * @param left The first polyomino.
    * @param right The second polyomino.
    * @return a value of -1, 0 or +1 if left is regarded to  be smaller, equal to or larger than right.
    */
    static int compareTo( final FreePoly left, final FreePoly right)
    {
        return compareTo(left.bits, right.bits) ;
    }

    /** Flip elements of array by swapping columns
    * @return The clone of the byte array where within each row the order of elements is reversed
    */
    static byte[][] flipx(final byte[][] in)
    {
        final int rows = in.length ;
        final int cols = ( rows > 0 ) ? in[0].length : 0 ;
        byte[][] out = new byte[rows][cols] ;
        for(int r = 0 ; r < rows ; r++)
            for(int c=0 ; c < cols ; c++)
                out[r][c] = in[r][cols-1-c] ;
        return out ;
    }



    /** Rotate array by 90 degrees ccw.
    * @return The clone of the byte array where the number of columns and rows have been swapped.
    */
    static byte[][] rot90(final byte[][] in)
    {
        final int cols = in.length ;
        final int rows = ( cols > 0 ) ? in[0].length : 0 ;
        byte[][] out = new byte[rows][cols] ;
        for(int r = 0 ; r < rows ; r++)
            for(int c=0 ; c < cols ; c++)
                out[r][c] = in[c][rows-1-r] ;
        return out ;
    }

    /** List all compositions
    * @param in[][] A rectangular array of 1-digit numbers
    * @return A string representation of the vectors [c00,c01...],[c10,c11...] separated by line feeds.
    */
    public static String toString(final byte in[][])
    {
        final int rows = in.length ;
        final int cols = ( rows > 0 ) ? in[0].length : 0 ;
        String str = new String();
        for(int r=0 ; r < rows ; r++)
        {
            for(int c=0 ; c < cols ; c++)
                str += in[r][c] ;
            str += "\n" ; /* todo: use locale line feed */
        }
        return str ;
    } /* toString */

    /** Print a human-readable pattern of 0's and 1's that represent the polyomino.
    * @return The zeros and ones with one list per output line.
    */
    public String toString()
    {
        return toString(bits) ;
    } /* toString */

} /* FreePoly */
```

Appendix D. Source code of FreePolySet.java

```
/** @file
* The set of n-ominoes with a r X c bounding box and fixed n.
```

```
* @author R. J. Mathar
*/

import java.util.* ;
import java.lang.* ;

/**
* @brief compute the set of all free n-ominoes with given bounding rectangle.
* @since 2019-05-11
*/
public class FreePolySet
{
    /** the sum of the parts
    */
    int n ;

    /** the number of rows
    */
    int rows ;

    /** the number of columns
    */
    int cols ;

    Vector<FreePoly> polys ;

    /**
    * Constructor with a predefined n-omino.
    * This just stores the main parameters and does not actually
    * compute anything.
    * @param n The number of squares in each n-onmino
    * @param r The number of rows in each n-omino
    * @param c The number of columns in each n-omino
    * @since 2019-05-11
    */
    public FreePolySet(int n, int r, int c)
    {
        this.n = n ;
        rows = r ;
        cols = c ;
        polys = new Vector<FreePoly>() ;
    } /* ctor */

    /** Main part of the solution: create all n-ominoes.
    */
    public void create(boolean freep, int Read)
    {
        /* no solution if there are more n than r*c
        */
        if ( n > rows*cols)
            return ;

        /* each row must contain at least one square to
        * keep the n-omino connected, and at most cols squares because
        * the columns are essentially bitsets
        */
        Composit rowComp = new Composit(n,rows,1,cols) ;

        /* accumulate only once all possible bit sets of the rows
        */
        Vector<Composit> bitsets = new Vector<Composit>() ;
        for (int bweit =0 ; bweit <= cols ; bweit++)
        {
            Composit hamm = new Composit(bweit,cols,0,1) ;
            bitsets.add(hamm) ;
        }


        /* loop over all compoistions of the row sums */
        for ( int[] rc : rowComp.comps)
        {
            /* skip those where the reverse of the composition would be smaller,
```

```
            * because we'll create them anyway by the 180 deg rotations...
            */
            int[] rcrev = Composit.reverse(rc) ;
            if ( Composit.compareTo(rcrev,rc) >= 0  || !freep)
            {
                /* now row sums are fixed ; distribute them over
                * rows: need binary vectors with rc[] bits set.
                */
                byte[][] bits = new byte[rows][cols] ;
                create(bits,0,rc,freep,Read,bitsets) ;
            }
        }
    } /* create */

    /** Main part of the calculation: create all of them
    * @param bits The polyomino with a bit[r][c] equal to one of the square is covered.
    * @param prow The pivotal row from 0 up to the number of rows (-1 in Java).
    *         This is the row in bits[][] which needs to be filled next.
    * @param rc The vector of row sums. rc[r] is the number of bits to be set in row r.
    * @param freep If true calculate free n-omonioes, else fixed.
    * @param Read Use C. Read incomplete symmetries for polyominoes with square bounding rectangle.
    * @param bitsets bitsets[b] contains all ways to distribute b bits over columns.
    *    The parameter may be null, then vector is inefficiently recomputed locally.
    */
    public void create(byte[][] bits, int prow, int [] rc, boolean freep, int Read,
        final Vector<Composit> bitsets)
    {
        /* impossible to create solutions if that row sum is larger than
        * the number of columns.
        */
        if ( rc[prow] > cols)
            return ;

        /* strategy is to find the bitsets that have as many
        * bits set as rc[prow] indicates. Check each of them
        * in turn if that has a common edge with the previous
        * row of bits (ie. bit-wise and is not zero), and preliminarily
        * add this as a new row.
        */
        final Composit thisrow = (bitsets == null) ?
            new Composit(rc[prow],cols,0,1) : bitsets.elementAt(rc[prow]) ;
        for( int[] brow : thisrow.comps)
        {
            /* is the connectivity (percolation reuirement) satisfied ?
            */
            boolean percol ;
            /* no contraint on bitset if this is the first row.
            */
            if ( prow == 0 )
            {
                percol = true;
                /* if this is for free polyominoes, we only need to start
                *with approximately the smaller "half" of the bitsets because
                * the other polyominoies can be created by flipping along the horiz. axis.
                * Skip dealing with this set brow[] of bits if the reversed
                * would be lexicographically smaller.
                */
                if ( freep)
                {
                    final int[] bitsRev = Composit.reverse(brow) ;
                    if ( Composit.compareTo(bitsRev, brow) < 0 )
                        continue ;
                }
            }
            else
            {
                percol = false;
            }

            /* run with a bit (column) wise and along the columns and
            * check that at least one of the squares is edge-connected with
            * a square of the previous row
```

```
        */
        for(int c =0 ; c < cols && !percol; c++)
            if ( brow[c] == 1 && bits[prow-1][c] == (byte) 1)
                percol = true ;

        if ( percol)
        {
            for(int c =0 ; c < cols ; c++)
                bits[prow][c] = (byte) brow[c] ;

            if ( prow == rows-1)
                /* reached a leave of the search scan
                */
                create(bits,freep,Read) ;
            else
            {
                /* recursively add another row */
                create(bits, prow+1,rc,freep,Read,bitsets) ;
            }
        }
    }
} /* create */

/** Check whether bits[][] is a valid n=ominoe and
* add to the list if not yet present.
* @param bits
* @param freep
* @param Read If this is >=1, reduce free polyominoes only with a group symmetry of order 4,
*     even if the width equals the height.
*/
void create(byte[][] bits,boolean freep, int Read)
{
    /* this is the set of squares that are not yet associated
    * with the cluster.
    * Check that all parts of the composition of the column sums are >0
    */
    for(int c=0 ; c < cols ; c++)
    {
        int su = 0 ;
        for(int r=0 ; r < rows ;r++)
            su += bits[r][c] ;
        if ( su == 0 )
            return ;
    }
    Vector<byte[]> freeSet = new Vector<byte[]>() ;
    for(int r=0 ; r < rows ; r++)
    for(int c=0 ; c < cols ; c++)
        if ( bits[r][c] > 0 )
        {
            byte[] coo= new byte[2] ;
            coo[0] = (byte) r ;
            coo[1] = (byte) c ;
            freeSet.add(coo) ;
        }

    /* this set contains [r][c] lists of 2d cordinates
    * of set bits (squares of the n-ominoe) connected
    * with the first square. If all conncetions are checked,
    * the size of this vector must be n if the n-ominoe is connected
    */
    Vector<byte[]> coneCluster = new Vector<byte[]>() ;
    /* assume n>=1, so at least one element in freeSet()
    */
    coneCluster.add(freeSet.firstElement()) ;
    freeSet.removeElementAt(0) ;

    for(; ! freeSet.isEmpty() ;)
    {
        /* search through all freeSet squares and
        * try to add some to the connected cluster
        */
        boolean enlarged = false ;
```

```
            for( byte[] cand: freeSet)
            {
                /* is this candidate neighbout of any in the conecluster?
                */
                boolean isne = false ;
                for( byte[] inclus : coneCluster)
                {
                    if ( Math.abs(cand[0]-inclus[0]) == 1 && cand[1]==inclus[1]
                        ||  Math.abs(cand[1]-inclus[1]) == 1 && cand[0]==inclus[0])
                    {
                        isne =true ;
                        break ;
                    }
                }
                if ( isne)
                {
                    /* has  neibbour in the cluster: move from the
                    * freeSet to coneCluster */
                    coneCluster.add(cand) ;
                    freeSet.remove(cand) ;
                    enlarged = true ;
                    break ; /* needed to avoid scanning the mof */
                }
            }
            if ( ! enlarged)
                break ;
        }

        if ( coneCluster.size() == n)
        {
            FreePoly cand =new FreePoly(bits,freep,Read) ;
            /* check wheter this is a new n-omino */
            boolean known =false ;
            for( FreePoly pol :  polys)
                if ( FreePoly.compareTo(pol, cand) == 0 )
                {
                    known = true ; break;
                }

            /* append the new polyomino if it differs from all the known ones.
            */
            if ( ! known)
                polys.add(cand) ;
        }
    } /* create*/

    /** List all polyominoes in the set
    * @return The binary vectors [c00,c01...],[c10,c11...]
    */
    public String toString()
    {
        String str = new String() ;
        for (int i=0 ; i < polys.size() ; i++)
            str += polys.elementAt(i).toString() + "\n" ;
        return str ;
    } /* toString */

    /** Main program
    * usage: java -cp . FreePolySet [-v] [-f] [-R #height] [-w #width] #size
    */
    public static void main(String[] args)
    {
        /* if verb=true, print also the {0,1} matrices
        */
        boolean verb = false ;

        /* if freep=true, generate only free polynomios, else fixed
        */
        boolean freep = true ;

        /* if Read=true, use the reduced symmetry set of Read.
        */
```

```
        int Read = -1 ;

        /* If this is a nonnegative integer: consider only polyominoes with that specific
        * number of columns (=width)
        */
        int fixedCol = -1 ;

        for( int optind =0 ; optind < args.length ; optind++)
        {
            if ( args[optind].equals("-v") )
                verb =true ;
            if ( args[optind].equals("-f") )
                freep =false  ;
            if ( args[optind].equals("-R") )
                Read = Integer.parseInt(args[++optind]) ;
            if ( args[optind].equals("-w") )
                fixedCol = Integer.parseInt(args[++optind]) ;
        }

        /* last command line argument is the polyomino size
        */
        int n = Integer.parseInt(args[args.length-1]) ;

        /* counter for the number of polyominios in that class
        */
        int tot = 0 ;

        /* loop over all numbers of columns
        */
        for(int c= 1; c<=n; c++)
        {
            /* if a request for only a single width (column) is made,
            * skip all others.
            */
            if ( fixedCol >= 0 && c != fixedCol)
                continue ;

            /* need r*c >= n, so don't start at 1..
            */
            int rmin = (Read>0 || !freep) ? 1 : Math.max(n/c,c) ;
            for(int r= rmin ; r+c-1 <=n ; r++)
            {
                if ( Read > 0 && r != Read)
                    continue ;
                FreePolySet po = new FreePolySet(n,r,c) ;
                po.create(freep,Read) ;
                if ( verb)
                    System.out.println(po.toString()) ;

                if ( po.polys.size() > 0 )
                {
                    System.out.println(""+ n + " " + r + " " + c + " : " + po.polys.size()) ;
                    tot += po.polys.size() ;
                }
            }
        }
        if ( fixedCol < 0)
            System.out.println(""+ n + " : " + tot) ;
    } /* main */
} /* FreePolySet */
```

References

1. Edward A. Bender, L. Bruce Richmond, and S. G. Williamson, *Central and local limit theorems applied to asymptotic enumeration. iii. matrix recursions*, J. Combin. Theory A **35** (1983), no. 3, 263–278. MR 0721368
2. O. E. I. S. Foundation Inc., *The On-Line Encyclopedia Of Integer Sequences*, (2019), http://oeis.org/. MR 3822822
3. Iwan Jensen, *Counting polyominoes: a parallel implementation for cluster computing*, Lect. Notes Comp. Science **2659** (2003), 203–212.

4. David A. Klarner, *Cell growth problems*, Canad. J. Math. **19** (1967), 851–863. MR 0214489
5. Donald E. Knuth and Richard Stong, *Problem 10875, animals in a cage*, Am. Math. Monthly **110** (2003), no. 3, 243–245.
6. Wolfgang R. Müller, Klaus Szymanski, Jan V. Knop, and Nenad Trinajstic, *On the number of square-cell configurations*, Theor. Chim. Acta **86** (1993), 269–278.
7. T. R. Parkin, L. J. Lander, and D. R. Parkin, *Polyomino enumeration results*, SIAM review, vol. 10, SIAM Fall Meeting, no. 2, 1967, pp. 244–290.
8. Ronald C. Read, *Contributions to the cell growth problem*, Canad. J. Math. **14** (1962), no. 1, 1–20. MR 0131367

Email address: mathar@mpia-hd.mpg.de
URL: http://www.mpia.de/~mathar

Max-Planck Institute of Astronomy, Königstuhl 17, 69117 Heidelberg, Germany