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 17-ominoes and partially to even larger polyominoes. [vixra:1905.0474]

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: June 26, 2020.
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 number of cells in 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 to connect all cells of the n-omino:

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

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 [9]. 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 [5], the actual discrepancy with his data is for $P_{5\times 5}(10) = 529$ 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.] This is similar to printing an object layer-by-layer in stereolithography [4, 11].

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][7], which shows the integrity of the program.

For $n \leq 14$ agree with the published table [8]; 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 7 5 : 3766042	17 8 5 : 5039358	17 9 5 : 3630653	17 10 5 : 1547398
17 11 5 : 395198	17 12 5 : 56623	17 13 5 : 3630	17 6 6 : 2567828
17 7 6 : 7630113	17 8 6 : 5799584	17 9 6 : 2633896	17 10 6 : 737198
17 11 6 : 119428	17 12 6 : 8689	17 7 7 : 3334120	17 8 7 : 3356103
17 9 7 : 1047667	17 10 7 : 191974	17 11 7 : 16025	17 8 8 : 587349
17 9 8 : 241977	17 10 8 : 22827	17 9 9 : 13006	17 : 50107909

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 9 4 : 4421955	18 10 4 : 4597056
18 11 4 : 2784608	18 12 4 : 1015049	18 13 4 : 218608	18 14 4 : 25758
18 15 4 : 1331	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

20 20 1 : 1         20 10 2 : 1         20 11 2 : 55        20 12 2 : 1141
20 13 2 : 6796      20 14 2 : 15041     20 15 2 : 14733     20 16 2 : 7195
20 17 2 : 1862      20 18 2 : 265       20 19 2 : 19        20 7 3 : 8
20 8 3 : 2207       20 9 3 : 75982      20 10 3 : 598863    20 11 3 : 1756745

21 21 1 : 1         21 11 2 : 6         21 12 2 : 335       21 13 2 : 4201
21 14 2 : 17116     21 15 2 : 28964     21 16 2 : 23265     21 17 2 : 9845
21 18 2 : 2270      21 19 2 : 298       21 20 2 : 19

22 22 1 : 1         22 11 2 : 1         22 12 2 : 66        22 13 2 : 1725
22 14 2 : 13353     22 15 2 : 39392     22 16 2 : 52556     22 17 2 : 35573
22 18 2 : 13189     22 19 2 : 2768      22 20 2 : 334       22 21 2 : 21
```

For fixed n and w, the sums $\sum_{h=w}^{\lfloor n/w\rfloor} P_{h\times w}(n)$ 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][6]. An equivalent triangle for fixed n-ominoes is also in the OEIS [2, A308359].

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

For fixed w and h, the sums $\sum_{n=w+h-1}^{wh} P_{h\times w}(n)$ are collected in [2, A268371].

Appendix A. The Java Program

A.1. **Compilation and Use.**

The Java classes that follow are compiled as usual with

```
javac de/mpg/mpia/rjm/{Composit.java,FreePoly.java,FreePolySet.java,FreePolySetThrd.java}
jar cfe FreePolySet.jar de.mpg.mpia.rjm.FreePolySet de/mpg/mpia/rjm/*
```

The compiled source code is also available in https://www.mpia.de/~mathar/progs/FreePolySet.jar. The main program is then called with

```
java -cp . de.mpg.mpia.rjm.FreePolySet [-v] [-j #] [-f] [-w #] [-h #] [-s {N,R,H,V,HVR}] n
```

respectively

```
java -jar FreePolySet.jar [-v] [-j #] [-f] [-w #] [-h #] [-s {N,R,H,V,HVR}] n
```

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

The option `-v` lets the program print the (0,1)-matrix for each polyomino that is constructed. This is done in two formats: (i) a list of row and column indices of the occupied celles in parenthesis (r,c) where rows and columns count from 0 upwards. (ii) a sequence of zeros and ones in the shape of the bounding box, where the ones are the occupied and the zeros the empty cells in the square lattice. Note that the lines with the counts of Section 3 (recognized by containing colons) are printed anyway.

The option `-f` lets the program handle *fixed* polyominoes without reduction for the symmetry groups [2, A001168,A292357,A308359].

$n\backslash w$	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		3426576
16	1	4168	256851	1928220	4784867	4740152	1292409	72587		13079255
17	1	7570	619642	5523946	15648966	19496736	7945889	852153	13006	50107909
18	1	13987	1493272	15745682						
19	1	25549	3593527							
20	1	47108								
21	1	86319								
22	1	158978								

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

The option `-j` followed by a positive integer number uses parallel threads (as many as given by the followup integer) to construct the n-ominoes of a given width and height. If the option is not used, only a single thread is run.

The option `-w` followed by a positive integer number lets the program consider only a bounding box of a specific width; this will only generate any output if the width is in the range from 1 to n. If the option is not used, the program will execute a sequential loop over all widths which are commensurate with n.

The option `-h` followed by a positive integer number lets the program consider only a bounding box of a specific height. If the option is not used, the program will execute a sequential loop over all heights in the range w to $n-w+1$.

The options `-w` and `-h` support parallelization of the computations on the operating system level by calling it more than once at the same time for different widths and/or heights.

The option `-s` followed by a string with a subset of Redelmeier's capital letters [10] filters the polyominoes with respect to symmetry. The letter N constructs only polyominoes without symmetry, the letter R constructs only those with a symmetry of a 180° rotation about the center of the bounding box, the letter H constructs only those with a symmetry of flipping along the short axis, the letter V constructs only those with a symmetry of flipping along the long axis. [The short axis of the bounding box has length w, the long axis length h.] The string HVR selects polyominoes that have all of the symmetries H, V and R.

The call

`java -jar FreePolySet.jar -v 5`

for example would show the 12 free pentominoes.

A.2. **Classes.** 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 program 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 de/mpg/mpia/rjm/Composit.java

```
/*
* $Header: de/mpg/mpia/rjm/Composit.java$
*/

/** @file
* A class which generates the compositions of an integer into a fixed number of parts.
* @author R. J. Mathar
* @see <a href="http://vixra.org/abs/1905.0474">vixra:1905.0474</a>
*/

package de.mpg.mpia.rjm ;

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

/**
* @brief The set of compositions of some fixed positive integer.
* @since 2019-05-11
* @author Richard J. Matar
*/
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.
* This is not just defining the task at hand but actually generating
* all compositions within the ctor.
* @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 different 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 */

    /** List all compositions
    * @return The compositions [c00,c01...],[c10,c11...]
    */
    public String toString()
    {
        String str = new String();
        for (int[] p : comps)
        {
            str += "[" ;
            for (int pi = 0 ; pi < p.length ; pi++)
                str += p[pi] +"," ;
            str += "]" ;
        }
        return str ;
    } /* toString */

    /** Test program
    */
    public static void main(String[] args)
    {
        for(int n = 0 ; n < 6 ; n++)
        for(int k = 0 ; k < 6 ; k++)
        {
            Composit c = new Composit(n,k,1,n) ;
            System.out.println("n "+n + " k " +k + " : " + c.comps.size() + " " + c.toString()) ;
        }
    } /* main */
} /* class Composit */
```

Appendix C. Source code of de/mpg/mpia/rjm/FreePoly.java

```
/*
* $Header: de/mpg/mpia/rjm/FreePoly.java$
*/

/** @file
* A n-omino with a r times c bounding box.
* @author R. J. Mathar
* @see <a href="http://vixra.org/abs/1905.0474">vixra:1905.0474</a>
*/

package de.mpg.mpia.rjm ;

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

/**
* @brief A free n-omino with a tight bound box (convex rectangular hull) of r rows and c columns.
* @since 2019-05-11
* @author Richard J. Mathar
*/
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 considered the same if rotated/flipped.
* @param Read is a flag: if zero or positive, exclude 90degree rotation symmetries.
*  That means if negative, the equivalence operations of reducing
*  the set of occupied cells to a single representative are all tested,
*  of which there are 4 operations (group of identity, two flips and 180 deg rotation)
*  for rectangular shapes, and 8 operations (those by adding 90 degree rotations)
*  for the square shapes. If the Read parameter is zero or positive, that additional
*  set of 4 operations (90 degree rotations) is *not* added to the square shapes,
*  and all results will be blind to those symmetries, i.e., this is not what
*  the usual meaning for free polyominoes would do.
* @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 */

/**
* 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 counted as one if rotated/flipped.
* @param Read is a flag: if zero or positive, exclude 90degree rotation symmetries.
*  That means if negative, the equivalence operations of reducing
*  the set of occupied cells to a single representative are all tested,
*  of which there are 4 operations (group of identity, two flips and 180 deg rotation)
*  for rectangular shapes, and 8 operations (those by adding 90 degree rotations)
*  for the square shapes. If the Read parameter is zero or positive, that additional
*  set of 4 operations (90 degree rotations) is *not* added to the square shapes,
*  and all results will be blind to those symmetries, i.e., this is not what
*  the usual meaning for free polyominoes would do.
* @since 2019-05-11
*/
public FreePoly(final int[][] zeroone, boolean freep, int Read)
{
    this(zeroone, zeroone.length, freep, Read) ;
} /* ctor */

/**
* Constructor with a predefined n-omino.
* @param zeroone The array of the zeros and ones.
* @param rowsToKeep The number of valid rows in zeroone.
*   By default this should be the same as zeroone.length, but the
*   number can be chosen to be less such that the exceeding rows in zeroone will
```

```
*   be ignored.
* @param freep If true, construct free polynomios.
*  That means store a representation counted as one if rotated/flipped.
* @param Read is a flag: if zero or positive, exclude 90degree rotation symmetries.
*  That means if negative, the equivalence operations of reducing
*  the set of occupied cells to a single representative are all tested,
*  of which there are 4 operations (group of identity, two flips and 180 deg rotation)
*  for rectangular shapes, and 8 operations (those by adding 90 degree rotations)
*  for the square shapes. If the Read parameter is zero or positive, that additional
*  set of 4 operations (90 degree rotations) is *not* added to the square shapes,
*  and all results will be blind to those symmetries, i.e., this is not what
*  the usual meaning for free polyominoes would do.
* @since 2020-06-16
*/
public FreePoly(final int[][] zeroone, int rowsToKeep, boolean freep, int Read)
{
    if ( rowsToKeep < 0 || rowsToKeep > zeroone.length)
        throw new IndexOutOfBoundsException("row index " + rows) ;
    rows = rowsToKeep ;
    if ( zeroone.length > 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] = (byte) zeroone[r][c] ;
            n += bits[r][c] ;
        }

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



/** Check whether the bit set has all 1 connected
* This is a static variant because creating a object of FreePoly-type
* would (usually) consider reduce() which is an expensive operation and not needed here.
* @param zeroone The binary array.
* @param validRows the array of zeroone has zeroone.length rows, but
*    only the rows 0..validRows are valid/known.
* @return True if the zeroone[0..validRows] first elements are connected.
* @since 2020-06-16
*/
public static boolean isConnected(final int [][] bits, int validRows)
{

    /* Rearrange the information of the occupied cells by putting
    * their 2D coordinates into a vector.
    */
    Vector<byte[]> freeSet = new Vector<byte[]>() ;
    for(int r=0 ; r < validRows ; r++)
        for(int c=0 ; c < bits[0].length ; c++)
            if ( bits[r][c] > 0 )
            {
                byte[] coo= new byte[2] ;
                coo[0] = (byte) r ;
                coo[1] = (byte) c ;
                freeSet.add(coo) ;
            }

    /* the number of 1's in the zeroone[0..validRows] rows
    */
    final int n = freeSet.size() ;

    /* this set contains [r][c] lists of 2d cordinates
```

```
     * of set bits (squares of the n-ominoe) connected
     * with the first square. If all connections are checked,
     * the size of this vector must be n if the n-ominoe is connected.
     * We tackle the problem that some meandering forms of disconnected
     * cell clusters do not define a n-ominoe. The algorithm is to
     * put initially one of the freeSet[] cells into the cluster,
     * and moving recursively the remaining freeSet[] cells into the
     * cluster once it is verified that they share an edge with any of
     * the cells in the cluster.
     */
    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. Keep track
        * with the enlarged variable whether that succeeded for at
        * least one in the freeSet.
        */
        boolean enlarged = false ;
        for( byte[] cand: freeSet)
        {
            /* is this candidate neighbour of any in the cluster?
            */
            boolean isne = false ;
            for( byte[] inclus : coneCluster)
            {
                /* neighbour to the N, S, E or W: coordinate differences in the square lattice
                */
                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. It would be sufficient to use add()
                * here, but because the comparison in the isne loop starts at
                * the beginning of the coneCluster, and the initial freeSet was scanned
                * row-wise, it's faster to add these at the start...
                coneCluster.add(0,cand) ;
                */
                coneCluster.add(cand) ;
                freeSet.remove(cand) ;
                enlarged = true ;
                break ; /* needed to avoid scanning the moved elements */
            }
        }

        if ( ! enlarged)
            /* did not move any of the freeSet elements into the coneCluster
            */
            break ;
    }
    /* connected, if all coordinate pairs have been moved from the freeSet
    * to the conCluster in the loop..
    */
    return ( coneCluster.size() == n ) ;
} /* isConnectedOld */


/**
* @brief construct the r-rooted free polyominoes
* @param rMult The number of marked cells. currently always r=1, the standard definition of rooted polyominoes.
* @return The free polyomioes where rooted cells are marked with a 2 in the matrix cell.
```

```
* @since 2019-05-25
*/
public Vector<FreePoly> rooted(int rMult)
{
    /* loop over all 1bits in the matrix, flip this to 2, reduce again, and
    * collect the distinct results.
    */
    Vector<FreePoly> rtd = new Vector<FreePoly>() ;
    for (int r=0 ; r < rows ; r++)
    for (int c=0 ; c < cols ; c++)
    {
        if ( bits[r][c] > 0 )
        {
            byte[][] bitsR = new byte[rows][cols] ;
            for (int rr=0 ; rr < rows ; rr++)
            for (int cc=0 ; cc < cols ; cc++)
                if ( rr == r && cc == c)
                    bitsR[rr][cc] = (byte)2 ;
                else
                    bitsR[rr][cc] = bits[rr][cc] ;

            /* construct a normalized rooted free polyomino
            */
            FreePoly mrkd = new FreePoly(bitsR,true,-1) ;
            boolean kown = false ;
            for( FreePoly k : rtd)
            {
                if ( compareTo(mrkd,k) == 0 )
                {
                    kown = true ;
                    break ;
                }
            }
            if ( ! kown )
                rtd.add(mrkd) ;
        }
    }
    return rtd ;
}

/** Construct the rotated and flipped versions. Retain only one.
* @param Read is a flag: if zero or positive, exclude 90degree rotation symmetries.
*  That means if negative, the equivalence operations of reducing
*  the set of occupied cells to a single representative are all tested,
*  of which there are 4 operations (group of identity, two flips and 180 deg rotation)
*  for rectangular shapes, and 8 operations (those by adding 90 degree rotations)
*  for the square shapes. If the Read parameter is zero or positive, that additional
*  set of 4 operations (90 degree rotations) is *not* added to the square shapes,
*  and all results will be blind to those symmetries, i.e., this is not what
*  the usual meaning for free polyominoes would do.
*/
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[][] r180 = rot180(bits) ;
    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
        */
        byte[][] r90 = rot90(bits) ;
        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 */


/** Check whether the polynomino has a symmetry of order 2 etc.
* @param symm String N, R, H, V or empty.
*  The empty string means that we don't care aboutsymmetry and the result is always 'true.'
* @return true if it has a N, R, H, V symmetry.
*/
public boolean isSymm(String symm)
{
    if ( symm.isEmpty() )
        return true ;

    if ( symm.indexOf("R") >=0 )
    {
        byte[][] r180 = rot180(bits) ;
        if ( compareTo(bits, r180) != 0 )
            return false ;
    }

    if ( symm.indexOf("H") >=0 )
    {
        byte[][] fl = flipy(bits) ;
        if ( compareTo(bits, fl) != 0 )
            return false ;
    }

    if ( symm.indexOf("V") >=0 )
    {
        byte[][] fl = flipx(bits) ;
        if ( compareTo(bits, fl) != 0 )
            return false ;
    }

    if ( symm.indexOf("N") >=0 )
    {
        /* request that this must not have the R, H or V symmetry...
        */
        byte[][] img = rot180(bits) ;
        if ( compareTo(bits, img) == 0 )
            return false ;
        img = flipx(bits) ;
        if ( compareTo(bits, img) == 0 )
            return false ;
        img = flipy(bits) ;
        if ( compareTo(bits, img) == 0 )
            return false ;
    }

    return true ;
```

```
}

/** Test whehter another object is an polyominoe of the same arrangement of cells.
* @param oth The object to be compared to this.
* @return True if the object is a polyomino with the same shape and arrangement of cells.
*/
@Override
public boolean equals(Object oth)
{
    if ( oth instanceof FreePoly )
    {
        byte[][] othbits = ((FreePoly)(oth)).bits ;
        return ( compareTo(bits,othbits) ==0 ) ;
    }
    else
        return false;
}

/** 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 ;
}

/** Flip elements of array by swapping rows
* @return The byte array where row r is swapped with row in.length-1-r.
*/
static byte[][] flipy(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[rows-1-r][c] ;
    return out ;
}

/** Rotate array by 180 degrees
* @return The byte array flipped by y and then by x.
*/
static byte[][] rot180(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[rows-1-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 ;
}

/** @brief List occupied cells.
* This prints the array in a parenthetical list of the form (c1r,c1c)(c2r,c2c)(c3r,c3c),..
* where (cir,cic) are the row and column indices of the non-zero entries in the array.
* Both cir and cic and 0-based, i.e., start at 0 at the first row and first column.
* The order of the occpuied cells is row by row.
* @param in[][] A rectangular array of numbers
* @return A string representation of the coordinates of occupied cells.
* @since 2020-06-25
*/
public static String toStringCells(final byte in[][])
{
    final int rows = in.length ;
    final int cols = ( rows > 0 ) ? in[0].length : 0 ;
    String str = new String();
    /* first a single line of cell coordinates
    */
    for(int r=0 ; r < rows ; r++)
    {
        for(int c=0 ; c < cols ; c++)
            if ( in[r][c] != 0 )
```

```
                str += "(" + r + "," + c + ")" ;
        }
        return str ;
    } /* toStringCells */

    /** Derive an ASCII representation of the polyomino.
    * @param in[][] A rectangular array of 1-digit numbers
    * @return A string representation of the occupied cells and a string representation of the binary matrix.
    * @since 2020-06-25 Add another  line with a parenthetical list of the occupied cells
    */
    public static String toString(final byte in[][])
    {
        final int rows = in.length ;
        final int cols = ( rows > 0 ) ? in[0].length : 0 ;
        /* start with a line of coordinate pairs of occupied cells
        */
        String str = toStringCells(in) ;
        str += System.getProperty("line.separator") ;

        /* Then the version of a binary matrix where a human can recognize the connectivity.
        * First row 0, then row 1 (to relate this to the info in the previous line)
        */
        for(int r=0 ; r < rows ; r++)
        {
            for(int c=0 ; c < cols ; c++)
                str += in[r][c] ;
            str += System.getProperty("line.separator") ;
        }
        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 de/mpg/mpia/rjm/FreePolySet.java

```
/*
* $Header: de/mpg/mpia/rjm/FreePolySet.java$
*/

/** @file
* The set of n-ominoes with a fixed rows X cols bounding box.
* @author Richard J. Mathar
* @see <a href="http://vixra.org/abs/1905.0474">vixra:1905.0474</a>
*/

package de.mpg.mpia.rjm ;

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

/**
* @brief compute the set of all free n-ominoes with given bounding rectangle.
* @since 2019-05-11
* @author Richard J. Mathar
*/
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 ;

/** Flag to introduce intermedia percolation checks
*/
boolean TEAROFF = false ;

/** Flag to impose only a set of 4 (not 8) symmetry operations on free polyonomoes of square shape.
* Use or not use C. Read's incomplete symmetries for polyominoes with square bounding rectangle.
*    If this parameter is zero or positive, the reduction of free polyominoes
*    to a single representative does not try reduction by 90 degree rotations of square shapes
*    as in C. Read, doi:10.4153/CJM-1962-001-2, A059682, A059683, A059684.
*/
static int CREAD = -1 ;

/**
* 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.
* @param freep If true create free polyominoes, else fixed.
* @param s A string of N, H, ,V, R symmetry requirements. Empty if none.
* @param jThrd Number of parallel threads generating the free n-ominoes
*/
public void create(boolean freep, final String symm, int jThrd) throws InterruptedException
{
    /* no solution if there are more cells n than r*c cells in the rectangle.
    */
    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. Create all compositions of n
    * into 'rows' parts, each part in the range 1..cols.
    */
    Composit rowComp = new Composit(n,rows,1,cols) ;

    if ( rowComp.size() <= 0 )
        return ;

    /* Compute only once all possible bit sets of the rows.
    * bitsets[i] contains the bitsets with i bits set, and each bitset with 'cols' bits.
    * To simplify the indexing, the sum is also performed for bweit=0, although
    * this gives actually only the trivial all-0 list which cannot occur if n>=1.
    */
    Vector<Composit> bitsets = new Vector<Composit>() ;
    for (int bweit =0 ; bweit <= cols ; bweit++)
    {
        Composit hamm = new Composit(bweit,cols,0,1) ;
        bitsets.add(hamm) ;
    }
```

```
    /* prepare threads to run. We will execute one or more row compositions
    * in a single thread, so it would be useless to generate more threads than
    * the current number of compositions...
    */
    jThrd = Math.min(jThrd,rowComp.size()) ;

    FreePolySetThrd[] thrds = new FreePolySetThrd[jThrd] ;
    Thread[] thrdsT = new Thread[jThrd] ;
    for (int m=0 ; m < jThrd ; m++)
    {
        thrds[m] = new FreePolySetThrd(n, rows, cols, freep, CREAD, symm, rowComp, bitsets, m, jThrd) ;
        thrdsT[m] = new Thread(thrds[m]) ;
        thrdsT[m].start() ;
    }

    /* wait for all threads to finish */
    for(int m=0 ; m < jThrd; m++)
    {
        thrdsT[m].join(0) ;
    }

    /* collect all the polyominioes of the threads into a single vector.
    */
    for(int m=0 ; m < jThrd; m++)
    {
        if ( rows != cols)
            polys.addAll( thrds[m].polys) ;
        else
        {
            /* Those of square shape may have
            * been generated by a row sum and also by another row sums (treated as a column sum).
            * These are to be reduced to generate them only once.
            */
            for( FreePoly p: thrds[m].polys)
            {
                if ( ! polys.contains(p) )
                    polys.add(p) ;
            }
        }
    }

} /* create */


/** List all polyominoes in the set
* @return The binary vectors [c00,c01...],[c10,c11...]
*/
public String toString()
{
    return toString(polys) ;
} /* toString */

/** List all polyominoes in the set
* @return The binary vectors [c00,c01...],[c10,c11...]
*/
public static String toString(Vector<FreePoly> polys)
{
    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] [-h #height] [-r R] [-w #width] [-s symm] [-j Nthreadds] #size
*/
public static void main(String[] args) throws InterruptedException
{
    /* if verb=true, print also the {0,1} matrices
    */
    boolean verb = false ;
```

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

/* The number of rooted cells. Initially 0 (=not rooted) for A000105.
* Using rMult=1 (with the option -r 1) gives OEIS A126202, the "pointed" polyominoes with n cells.
*/
int rMult = 0 ;

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

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

/* number of threads run in parallel
*/
int jThrd =1 ;

/* If not empty: count only polyominoes with a symmetry.
* N=none, R=rotation by 180 degrees, H=rotation along the short axis, V=rotation along long axis.
* See Redelmeister.
* Note that these are minium requirements. There is for example the full-block polyominos
* which has H+V+R.
* There is no implementation here to check for symmetry along the 2 diagonals for
* the polyominoes with a square hull.
*/
String symm = new String() ;

/* empty list of arguments: usage hint
*/
if ( args.length == 0 )
{
    final FreePolySet tmp = new FreePolySet(0,0,0) ;
    System.out.println("Usage: java -cp . " + tmp.getClass().getName()
        + " [-j Nthrd] [-w width] [-v] [-s Symm] ncell") ;
    return ;
}

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") )
        rMult = Integer.parseInt(args[++optind]) ;
    if ( args[optind].equals("-w") )
    {
        fixedCol = Integer.parseInt(args[++optind]) ;
        /* correct if erroneous non-positive input
        */
        fixedCol = Math.max(fixedCol,1) ;
    }
    if ( args[optind].equals("-h") )
    {
        fixedRow = Integer.parseInt(args[++optind]) ;
        /* correct if erroneous non-positive input
        */
        fixedRow = Math.max(fixedRow,1) ;
    }
    if ( args[optind].equals("-j") )
    {
        jThrd = Integer.parseInt(args[++optind]) ;
        /* correct if erroneous non-positive input
        */
        jThrd = Math.max(jThrd,1) ;
```

```
            }
            if ( args[optind].equals("-s") )
                symm = args[++optind] ;
        }

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

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

        /* loop over all numbers of columns (=widths)
        */
        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 ;

            /* Loop over all row lengths (=heights).
            * Need r*c >= n, so don't start at 1..
            */
            int rmin = (CREAD>0 || !freep) ? 1 : Math.max(n/c,c) ;
            for(int r= rmin ; r+c-1 <=n ; r++)
            {
                if ( fixedRow > 0 && r != fixedRow)
                    continue ;
                FreePolySet po = new FreePolySet(n,r,c) ;
                po.create(freep,symm,jThrd) ;
                if ( rMult > 0 && freep)
                {
                    /* generate the rMult-rooted free polyominoes
                    */
                    Vector<FreePoly> rtd = new Vector<FreePoly>() ;
                    for ( FreePoly nonrtd : po.polys)
                    {
                        rtd.addAll( nonrtd.rooted(rMult) ) ;
                    }

                    if ( rtd.size() > 0 )
                    {
                        if ( verb)
                            System.out.println( FreePolySet.toString(rtd) ) ;

                        System.out.println(""+ n + " " + r + " " + c + " : " + rtd.size()) ;
                        tot += rtd.size() ;
                    }
                }
                else
                {
                    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 */
```

Appendix E. Source code of de/mpg/mpia/rjm/FreePolySetThrd.java

```
/*
* $Header: de/mpg/mpia/rjm/FreePolySetThrd.java$
*/

/** @file
* Construct the set of n-ominoes with a r X c bounding box and fixed n with a fixed number of threads.
* @author Richard J. Mathar
* @see <a href="http://vixra.org/abs/1905.0474">vixra:1905.0474</a>
*/

package de.mpg.mpia.rjm ;

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

/**
* @brief compute a subset of all free n-ominoes with given bounding rectangle.
* @since 2020-06-22
* @author Richard J. Mathar
*/
public class FreePolySetThrd implements Runnable
{
    /** The number of cells in the n-omino and also the partition of the row sums.
    */
    int n ;

    /** The number of rows in each n-omino.
    */
    int rows ;

    /** The number of columns in each n-omino.
    */
    int cols ;

    /** Number of parallel threads generating the free n-ominoes
    */
    int jThrd ;

    /** This thread index, from 0 to Thrd-1.
    */
    int mThrd ;

    /** The polyominoes created by this thread
    */
    Vector<FreePoly> polys ;

    /** Flag to indicate if free (true) or fixed (false) n-ominoes are created
    */
    boolean freep ;

    /** Use C. Read incomplete symmetries for polyominoes with square bounding rectangle.
    */
    int Read ;

    /** A string with letters N, H, V, R symmetry requirements.
    * If the string is empty, no such filtering by symmetry is done (but of course
    * the free polyominoes are reduced to a single representative always with
    * symmetry operations of flips taken into account).
    */
    String symm ;

    /** The compositions of the row sums of this thread. Each polyomino
    * is a rows x cols matrix of 0's and 1's. rc[i] is the predefined row sum of row i.
    * Thread number m works on the row sums m, m+jThrd, m+2*jThrd etc. Because
    * these are disttinc if the free polyominoes are distinct, the sets of free
    * polyominoes of the different threads are non-overlapping.
    */
    Composit rowComp ;

    /** the row sum number i is flushed out into a bitset to generate
```

```
* values of 0's and 1's for row number i. All these bitsets are
* initially computed only once in advance to save some time, supposing
* that row sums are not unique in the matrices of 0's and 1's.
*/
Vector<Composit> bitsets ;

/** Flag to introduce intermediate percolation checks.
* Heuristically it does not seem to speed up the computation, at least for n up to 13,
* so it is disabled here by default.
*/
boolean TEAROFF = false ;

/**
* Constructor that defines the task to be run.
* This just stores the main parameters and does not actually
* compute anything.
* @param n The number of squares/cells in each n-onmino
* @param r The number of rows in each n-omino
* @param c The number of columns in each n-omino
* @param freep If this is true, generate/count free polyomioes, else fixed polyominoes.
* @param CReadflag Use C. Read incomplete symmetries for polyominoes with square bounding rectangle.
* @param symmFilt A string with letters N, H, V and/or R to filter symmetric polyominoes.
*   If N is present, count/create only polys without symmetry (symmetry group has only
*   the identity). If H and/or V demand symmetry with respect to horizontal or
*   vertical flips, if R demand symmetry with respect to rotation by 180 degrees.
*   (Obviously taking H and V implies that R is also in the group, but not vice versa...)
*   If empty, count polymonoes indepdenent of symmetry.
* @param rsums The compositions of n into r parts. Each composition
*     defines a layout for row sums of the binary matrix of a polyomino.
* @param bits01 The bits01(s) contains the compositions of s in c parts,
*     each part either 0 or 1 (so these are bitsets).
* @param m Thread number in the range 0..j-1.
* @param j Total number of treads to be called.
* @since 2020-06-22
*/
public FreePolySetThrd(int n, int r, int c, boolean freep, int CReadflag,
        final String symmFilt, Composit rsums, Vector<Composit> bits01, int m , int j)
{
    this.n = n ;
    rows = r ;
    cols = c ;
    this.freep = freep ;
    Read = CReadflag ;
    symm = symmFilt ;
    polys = new Vector<FreePoly>() ;
    rowComp = rsums ;
    bitsets =bits01 ;
    mThrd = m ;
    jThrd = j ;
} /* ctor */

/** Main part of the solution: create n-ominoes where the rowComp's index is assigned to this thread.
*/
public void run()
{
    /* no solution if there are more cells n than r*c cells in the rectangle.
    */
    if ( n > rows*cols)
        return ;

    if ( rowComp.size() <= 0 )
        return ;

    /* Outer loop over all compositions of the row sums .
    * Select the ones to be handled by this thread. By using a simple modulo
    * distribution to disperse the threads, this should be like well-balanced,
    * because the row compositions have some regular distribution...
    */
    for ( int j = mThrd ; j < rowComp.comps.size() ; j += jThrd)
    {
        final int[] rc = rowComp.comps.elementAt(j) ;
        /* skip those where the reverse of the composition would be larger,
```

```
        * because we'll create them anyway by the 180 deg rotations...
        * Note that using the completementary set where compareTo(rc,rcrev)>=0
        * gives the same results, but is slower, because the bit sets of
        * larger weight in the initial rows then, and there is better efficiency
        * of using the restricition of connectivity.
        */
        final int[] rcrev = Composit.reverse(rc) ;
        if ( Composit.compareTo(rcrev,rc) >= 0  || !freep)
        {
            /* now row sums are fixed ; inner loop: distribute them over
            * rows: need binary vectors with rc[] bits set.
            */
            int[][] bits = new int[rows][cols] ;
            create(bits,0,rc) ;
        }
    }
} /* run */

/** 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.
*/
public void create(int[][] bits, int prow, int [] rc)
{
    /* 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. Connectivity: Check each of them
    * in turn if that has at least one 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 requirement) satisfied ?
        * percol=true if it is as defined for polyominoes.
        */
        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] == 1)
                    percol = true ;
            }

        }

        /* continue only if connectivity with adjacent row is verified
        */
        if ( percol)
        {
            /* copy selected bitset into the current pivotal row
            */
            bits[prow] = brow ;

            if ( prow == rows-1)
                /* reached a leave of the search scan: all bits[][] now defined.
                */
                add(bits) ;
            else
            {
                if ( TEAROFF && ( prow > 0 ) && ( prow % 5 == 0) )
                {
                    /* cover the array of bits with a top layer of all-1
                    * as if optimistic that there will be some arching cluster of
                    * 1's added latter to connect the pieces. Note that prow+1 cannot
                    * exceed rows here because that's caught in the if-clause above.
                    */
                    int roweff ;
                    if ( Arrays.equals(bits[prow], bitsets.elementAt(cols).comps.elementAt(0)) )
                    {
                        roweff = prow+1 ;
                    }
                    else
                    {
                        bits[prow+1] = bitsets.elementAt(cols).comps.elementAt(0) ;
                        roweff = prow+2 ;
                    }
                    if ( ! FreePoly.isConnected(bits, roweff) )
                    {
                        continue ;
                    }
                }
                /* recursively add next adjacent row */
                create(bits, prow+1,rc) ;
            }
        }
    }
} /* create */

/** Check whether bits[][] is a valid n-omino.
* Add to the list if not yet present.
* @param bits The 2D bit set with 1's for occupied and 0's for unoccpied celles.
*/
void add(int[][] bits)
{
    /* Check that all parts of the composition of the column sums are >0
    * (no shotgun solutions admitted..)
    */
    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 ;
        }
    }

    if ( FreePoly.isConnected(bits,rows) )
```

```
        {
            /* create a candidate polyomino for insertion, normalized representation
            */
            FreePoly cand =new FreePoly(bits,freep,Read) ;
            /* check whether this fails to be in any symmetry class that might be requested
            */
            if ( ! cand.isSymm(symm) )
                return ;
            /* check wheter this is a new n-omino.
            * Append the new polyomino if it differs from all the known ones.
            */
            if ( ! polys.contains(cand) )
                polys.add(cand) ;
        }
    } /* add */

} /* FreePolySetThrd */
```

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*, (2020), https://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, *Some results concerning polyominoes*, Fib. Quart. **3** (1965), no. 1, 9–20. MR 0186569
5. ———, *Cell growth problems*, Canad. J. Math. **19** (1967), 851–863. MR 0214489
6. Donald E. Knuth and Richard Stong, *Problem 10875, animals in a cage*, Am. Math. Monthly **110** (2003), no. 3, 243–245.
7. 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.
8. 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.
9. Ronald C. Read, *Contributions to the cell growth problem*, Canad. J. Math. **14** (1962), no. 1, 1–20. MR 0131367
10. D. Hugh Redelmeier, *Counting polyominoes: Yet another attack*, Disc. Math. **36** (1981), no. 2, 191–203. MR 0675352
11. Arne Schmidt, Sheryl Manzoor, Li Huang, Aaron T. Becker, and Sándor P. Fekete, *Efficient parallel self-assembly under uniform control inputs*, IEEE Robot. Autom. Lett. **3** (2018), no. 4, 3521.

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

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