TILING HEXAGONS WITH SMALLER HEXAGONS AND UNIT TRIANGLES

RICHARD J. MATHAR

Abstract. This is a numerical study of the combinatorial problem of packing hexagons of some equal size into a larger hexagon. The problem is well defined if all hexagon edges have integer length and if their centers and vertices share the common lattice points of a triangular grid with unit distances.

1. Definition of Shapes

A regular equilateral hexagon with integer edge length n (n-hexagon) can be tiled by hexagons of smaller edge length $1 \le s \le n$ (s-hexagons) and by equilateral unit triangles with each length 1 that fill the interstitial region between the s-hexagons. The smaller s-hexagons are centered at the vertices of the unit triangles and must not overlap. The n-hexagon consists of $6n^2$ unit triangles and the s-hexagon of $6s^2$ unit triangles, so there can be at most

$$0 \le k \le n^2/s^2 \tag{1}$$

s-hexagons inside the n-hexagon.

Definition 1. *$H_n(s,k)$ is the number of ways of tiling the regular hexagon with side length n with k hexagons each of side length s that share the same triangular unit lattice, and filling the residual space with $6(n^2 - ks^2)$ unit triangles.*

This type of filling is—for large n and $s = 1$—an approximation to the problem of how many circles of a given radius fit into a larger circle [2, 1][3, A023393].

There is

$$H_n(s,0) = 1 \tag{2}$$

way of not placing any hexagon. There are [3, A003215]

$$H_n(1,1) = 3(n-1)n + 1 \tag{3}$$

ways of placing a single 1-hexagon of unit side length into a bigger n-hexagon, because that is just the number of internal vertex points inside the n-hexagon at which the 1-hexagons can be anchored. For $2s > n$ no more than one s-hexagon fits into the n-hexagon; then (2) and

$$H_n(s,1) = 3(n-s)(n-s+1) + 1, \quad 2s > n, \tag{4}$$

cover all results.

Date: August 28, 2016.

2010 *Mathematics Subject Classification.* Primary 52C20; Secondary 05B45.

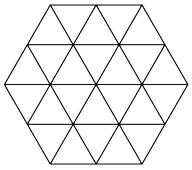

FIGURE 1. The $H_n(1,0) = 1$ variant with no hexagon inside a hexagon of any size.

2. Illustrations

Figures (1)–(4) illustrate tiling of the 2-hexagon by 1-hexagons. Figure 4 scales up to all cases where $s = n/2$:

$$H_{2s}(s,3) = 2. \tag{5}$$

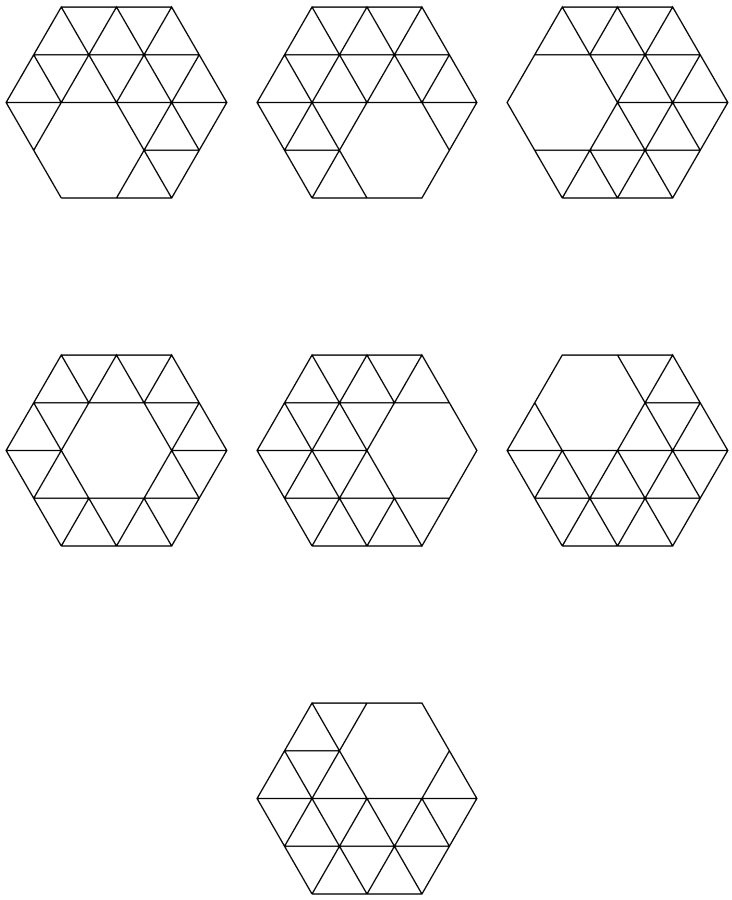

FIGURE 2. The $H_2(1,1) = 7$ variants of placing 1 hexagon of size $s = 1$ into a hexagon of size $n = 2$.

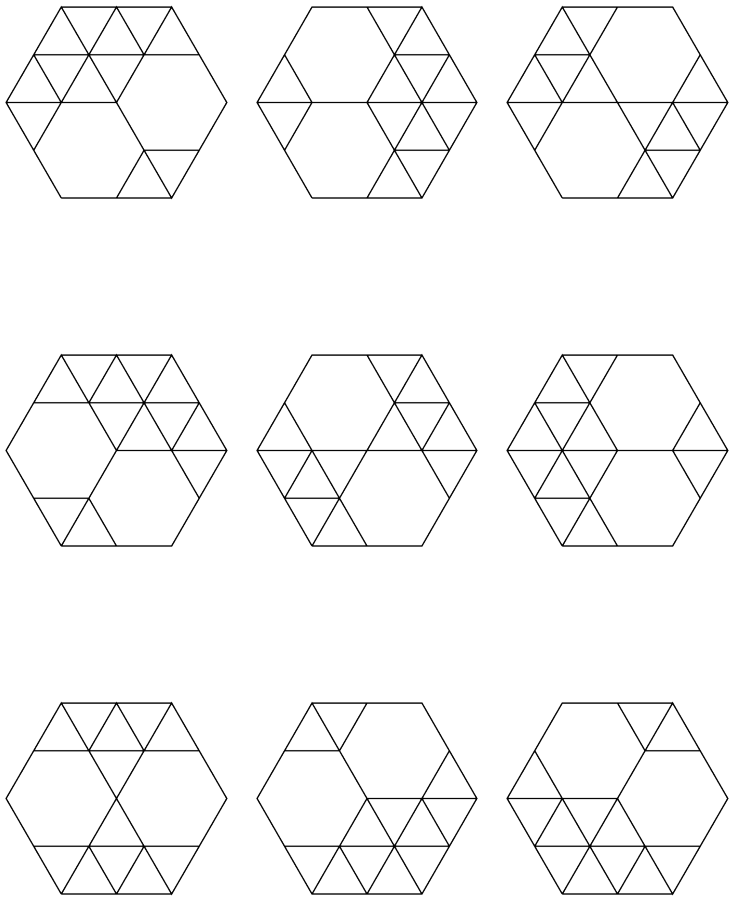

FIGURE 3. The $H_2(1,2) = 9$ variants of placing $k = 2$ hexagons of size $s = 1$ into a hexagon of size $n = 2$.

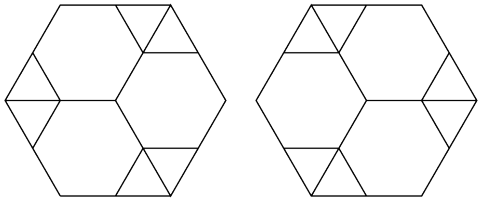

FIGURE 4. The $H_2(1,3) = 2$ variants of placing $k = 3$ hexagons of size $s = 1$ into a hexagon of size $n = 2$.

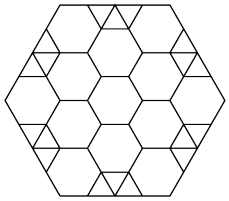

FIGURE 5. The $H_4(1,13) = 1$ variants of placing 13 1-hexagons in a 4-hexagon. After slicing the unit triangles into 4 triangles, this also illustrates the $H_8(2,13) = 1$ variants of placing 13 2-hexagons in a 8-hexagon.

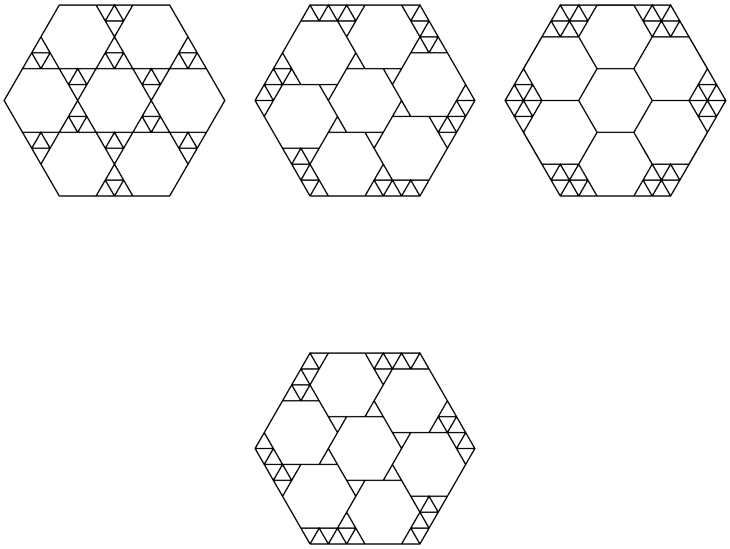

FIGURE 6. The $H_6(2,7) = 4$ variants of placing seven 2-hexagons in a 6-hexagon.

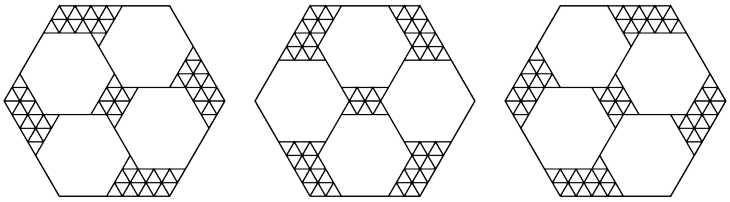

FIGURE 7. The $H_7(3,4) = 3$ variants of placing four 3-hexagons in a 7-hexagon.

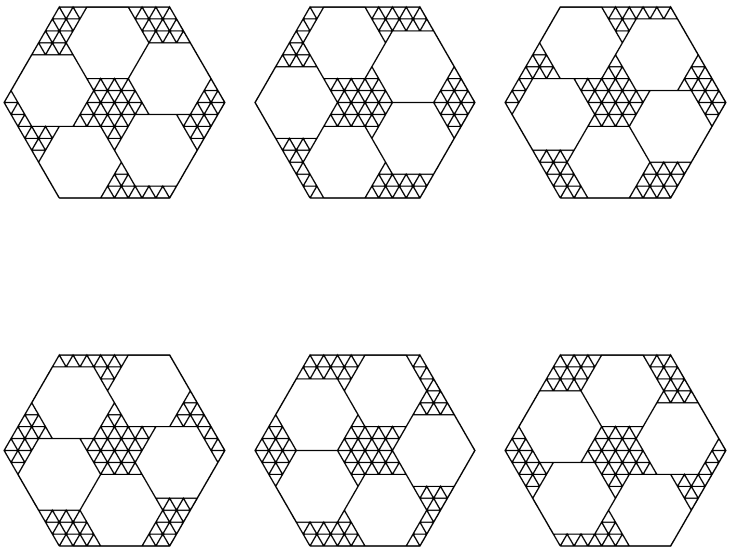

FIGURE 8. The $H_8(3,5) = 6$ variants of placing five 3-hexagons in a 8-hexagon.

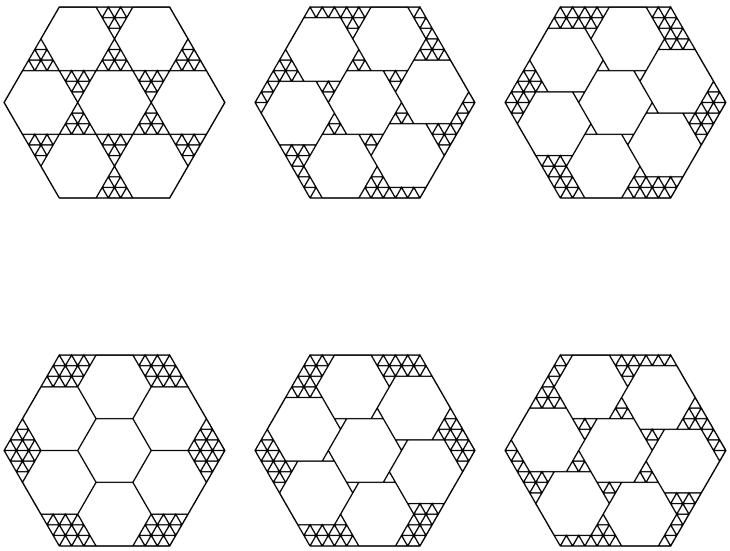

FIGURE 9. The $H_9(3,7) = 6$ variants of placing seven 3-hexagons in a 9-hexagon.

3. Numerical Results

The counts are summarized by showing in a single line the side length s of the smaller hexagons, the side length n of the larger circumscribing hexagon, a colon, the number of configurations with $k = 0, 1, \ldots$ embedded hexagons, another colon, and the total count (i.e., the row sum).

```
s n : H_N(s,0) H_n(s,1) H_n(s,2) ... : sum_k H_n(s,k)
```

The second line in that list was illustrated by Figures 1–4.

```
1 1:  1 1 : 2
1 2:  1 7 9 2 0 : 19
1 3:  1 19 129 390 532 297 55 2 0 0 : 1425
1 4:  1 37 576 4941 25711 84297 175446 228427 179532 80244 18579 1917 70 1  \
        : 799779
1 5:  1 61 1674 27371 297726 2278680 12660618 51953982 158735886 361525329  \
        610941281 758333301 680533543 431879221 188232405 54298606 9927462 \
        1104543 74263 3273 105 2 0 0 0 0 : 3322809333

2 2:  1 1 : 2
2 3:  1 7 0 : 8
2 4:  1 19 24 2 0 : 46
2 5:  1 37 279 470 78 0 0 : 865
2 6:  1 61 1095 6886 13433 5652 434 4 0 0 : 27566
2 7:  1 91 2904 40889 262701 736395 825129 304526 13434 0 0 0 0 : 2186070
2 8:  1 127 6246 153629 2042136 14885937 58151335 114431808 100893858 34380563 3756409 \
          92019 218 1 : 328794287

3 3:  1 1 : 2
3 4:  1 7 : 8
3 5:  1 19 0 : 20
3 6:  1 37 45 2 0 : 85
3 7:  1 61 465 454 3 0 : 984
3 8:  1 91 1710 7394 3786 6 0 0 : 12988
3 9:  1 127 4326 46154 123843 43641 1767 6 0 0 : 219865
3 10:  1 169 8961 182083 1364744 3230019 2167147 346656 234 0 0 0 : 7300014
3 11:  1 217 16371 540245 8064675 52211964 135627952 119305248 16143621 128670 0 0 0 0 : 332038964

4 4:  1 1 : 2
4 5:  1 7 : 8
4 6:  1 19 0 : 20
4 7:  1 37 0 0 : 38
4 8:  1 61 72 2 0 : 136
4 9:  1 91 687 432 0 0 : 1211
4 10:  1 127 2415 7350 696 0 0 : 10589
4 11:  1 169 5922 48188 48753 342 0 0 : 103375
4 12:  1 217 11970 194124 695779 224928 5266 8 0 0 : 1132293
4 13:  1 271 21423 591825 5146864 10553679 4655066 395938 0 0 0 : 21365067

5 5:  1 1 : 2
5 6:  1 7 : 8
5 7:  1 19 : 20
5 8:  1 37 0 : 38
5 9:  1 61 0 0 : 62
5 10:  1 91 105 2 0 : 199
5 11:  1 127 945 432 0 : 1505
5 12:  1 169 3210 7076 75 0 : 10531
```

```
5 13:  1 217 7686 48300 15507 0 0 : 71711
5 14:  1 271 15255 200202 333333 4908 0 0 : 553970
5 15:  1 331 26895 616982 2868479 903675 12911 10 0 0 : 4429284
5 16:  1 397 43686 1575993 15875049 29034351 8954967 445382 0 0 0 : 55929826

6 6:  1 1 : 2
6 7:  1 7 : 8
6 8:  1 19 : 20
6 9:  1 37 0 : 38
6 10:  1 61 0 : 62
6 11:  1 91 0 0 : 92
6 12:  1 127 144 2 0 : 274
6 13:  1 169 1239 432 0 : 1841
6 14:  1 217 4095 6844 3 0 : 11160
6 15:  1 271 9618 47236 3726 0 0 : 60852
6 16:  1 331 18810 201234 140085 6 0 0 : 360467
6 17:  1 397 32769 631906 1561989 38184 0 0 0 : 2265246
6 18:  1 469 52689 1624834 9545274 3045378 27642 12 0 0 : 14296299

7 7:  1 1 : 2
7 8:  1 7 : 8
7 9:  1 19 : 20
7 10:  1 37 0 : 38
7 11:  1 61 0 : 62
7 12:  1 91 0 : 92
7 13:  1 127 0 0 : 128
7 14:  1 169 189 2 0 : 361
7 15:  1 217 1569 432 0 : 2219
7 16:  1 271 5070 6750 0 0 : 12092
7 17:  1 331 11718 45872 675 0 : 58597
7 18:  1 397 22635 198630 49467 0 0 : 271130
7 19:  1 469 39039 635778 777348 390 0 0 : 1453025
7 20:  1 547 62244 1656746 5686629 205308 0 0 0 : 7611475
7 21:  1 631 93660 3738042 27131308 8982081 53551 14 0 0 : 39999288

8 8:  1 1 : 2
8 9:  1 7 : 8
8 10:  1 19 : 20
8 11:  1 37 : 38
8 12:  1 61 0 : 62
8 13:  1 91 0 : 92
8 14:  1 127 0 0 : 128
8 15:  1 169 0 0 : 170
8 16:  1 217 240 2 0 : 460
8 17:  1 271 1935 432 0 : 2639
8 18:  1 331 6135 6750 0 0 : 13217
8 19:  1 397 13986 44758 75 0 : 59217
8 20:  1 469 26730 194374 14793 0 0 : 236367
8 21:  1 547 45705 630960 343665 0 0 : 1020878
8 22:  1 631 72345 1667302 3170739 5724 0 0 : 4916742
8 23:  1 721 108180 3799672 17275638 858060 0 0 0 : 22042272

9 9:  1 1 : 2
9 10:  1 7 : 8
9 11:  1 19 : 20
9 12:  1 37 : 38
```

```
9 13:  1 61 0 : 62
9 14:  1 91 0 : 92
9 15:  1 127 0 : 128
9 16:  1 169 0 0 : 170
9 17:  1 217 0 0 : 218
9 18:  1 271 297 2 0 : 571
9 19:  1 331 2337 432 0 : 3101
9 20:  1 397 7290 6750 0 : 14438
9 21:  1 469 16422 44118 3 0 : 61013
9 22:  1 547 31095 189950 3675 0 : 225268
9 23:  1 631 52767 621192 135507 0 0 : 810098
9 24:  1 721 82992 1660152 1622676 6 0 0 : 3366548
9 25:  1 817 123420 3823640 10452858 44520 0 0 : 14445256
```

By glancing at these numbers we find heuristically for the class of compositions in Figure 3:

Conjecture 1.

$$H_{2s}(s,2) = 3s(s+2), \quad n = 2s. \tag{6}$$

If there is room of one more triangle to slide these $k = 2$ hexagons sideways:

Conjecture 2.

$$H_{2s+1}(s,2) = 3(6s^2 + 32s + 5), \quad n = 2s+1. \tag{7}$$

Attempts to pack a third hexagon in the barely oversized n-hexagon seem to be limited:

Conjecture 3.

$$H_{2s+1}(s,3) = 432, \quad s \geq 4, \quad n = 2s+1. \tag{8}$$

Definition 2. *$\hat{k}_n(s)$ is the maximum number of s-hexagons that can be placed in a n-hexagon.*

The rightmost nonzero entry in these results realize the maximum number k of s-hexagons that can be placed in n-hexagons, Table 1.

4. Lower bounds

An obvious strategy of maximizing the number of s-hexagons that fit into the n-hexagon is to place the first s-hexagon in one of the six corners of the n-hexagon, and all others such that the s-hexagons are a subset of the hexagonal close package. This yields covers like in Figure 4 or 5. Many of the other figures demonstrate that this strategy does often *not* lead to the largest k that is possible. The (lower) estimates of $\hat{k}_n(1)$ obtained with that closed-packing strategy are $1, 3, 6, 13, 21, 30, 43, 57, 72, 91, 111, \ldots$ for $n \geq 1$:

$$\hat{k}_n(s) \geq (n-s+1)(n-s) + t(n-s+1), \quad n \geq s, \tag{9}$$

where $t(n) = 1, 1, 0, 1, 1, 0 \ldots$ is a periodic sequence with period length 3.

A similar strategy of maximizing the number is to place the first s-hexagon at the center of the n-hexagon and attaching the others in the hexagonal closed-packed structure. This yields lower estimates $1, 1, 7, 13, 19, 31, 43, 55, 73, 91, 109, 133 \ldots$ for $n \geq 1$:

$$\hat{k}_n(1) \geq (n-s+1)(n-s) + g(n-s+1), \quad n \geq s, \tag{10}$$

$n\backslash s$	1	2	3	4	5	6	7	8	9
1	1	0	0	0	0	0	0	0	0
2	3	1	0	0	0	0	0	0	0
3	7	1	1	0	0	0	0	0	0
4	13	3	1	1	0	0	0	0	0
5	21	4	1	1	1	0	0	0	0
6		7	3	1	1	1	0	0	0
7		8	4	1	1	1	1	0	0
8		13	5	3	1	1	1	1	0
9			7	3	1	1	1	1	1
10			8	4	3	1	1	1	1
11			9	5	3	1	1	1	1
12				7	4	3	1	1	1
13				7	4	3	1	1	1
14					5	4	3	1	1
15					7	4	3	1	1
16					7	5	3	3	1
17						5	4	3	1
18						7	4	3	3
19							5	4	3
20							5	4	3
21							7	4	4
22								5	4
23								5	4
24									5
25									5

TABLE 1. $\hat{k}_n(s)$, the maximum number of s-hexagons that can be placed in n-hexagons.

where $g(n) = 1, -1, 1, 1, -1, 1, 1, \ldots$ is a periodic sequence with period length 3.

Switching between the two strategies depending on the value of $(n-s) \mod 3$ yields [3, A002061]

$$\hat{k}_n(s) \geq (n-s+1)(n-s)+1, \quad n \geq s. \tag{11}$$

Appendix A. Brute Force Counting: Java Program

The next two sections contain the Java classes that produces these results. They are compiled with

`javac -cp . Hexa.java Hexboard.java`

and run with

`java -cp . Hexboard` s n

to produce a histogram statistics of in how many ways s-hexagons can be placed in n-hexagons.

Appendix B. Hexa.java

```
1  import java.math.* ;
2  import java.util.* ;
3
4  /****************************************
5  * An object of the class Hexa is an equilateral hexagon with integer side with specified center coordin
6  * @since 2016-08-24
7  * @author R. J. Mathar
8  */
9  class Hexa
10 {
11     /** side/edge length of the hexagon.
12     * There are 6*s^2 equilateral unit triangles inside.
13     */
14     int s ;
15
16     /** i and j coordinate of the center.
17     * These are coordinates in the triangular grid measured along
18     * two unit vectors pointing into direction (1,0) and (1/2,sqrt(3)/2) in Cartesian coordinates.
19     */
20     int[] coord ;
21
22     /** Square root of 3 halved.
23     * This is the height of a unit triangle from the base to the opposite vertex.
24     */
25     final static double SQRT3_2 = 0.86602540378443864676372317 ;
26
27     /** Fully specified constructor with size and place.
28     * @param edgel Positive edge length.
29     * @param ci Coordinate along the first unit vector of the triangular lattice.
30     * @param cj Coordinate along the second unit vector of the triangular lattice.
31     */
32     Hexa(final int edgel, final int ci, final int cj)
33     {
34         s=edgel ;
35         coord=new int[2] ;
36         coord[0]=ci ;
37         coord[1]=cj ;
38     } /* ctor */
39
40
41     /** Computer whether this hexagon overlaps with another one.
```

```
42      * @param oth The other hexagon.
43      * @return Yes if there is overlap. This means that at least
44      *   one point of the other hexagon is fully inside this hexagon.
45      *   Fully inside means not just on the rim but closer to its center.
46      */
47      boolean overlaps(Hexa oth)
48      {
49          /* We will check in a loop all triangular points of oth being inside this.
50          * For efficiency check the coordinates in a loop over points in the smaller of the two,
51          * so we swap the roles of the two hexagons if oth is larger than this.
52          */
53          if ( oth.s > s)
54              return oth.overlaps(this) ;
55
56
57
58          /* Start with coordinates in the lower left corner and the
59          * work counter-clockwise to check each point of the 6 edges.
60          * i and j are the unit-vector coordinates relative to the center.
61          */
62          int i = 0 ;
63          int j = -oth.s ;
64          while (i < oth.s)
65          {
66              /* bottom horizontal edge */
67              if ( contains(oth.coord[0]+i, oth.coord[1]+j,false) )
68                  return true;
69              i++ ;
70          }
71          while (j < 0)
72          {
73              /* lower right edge */
74              if ( contains(oth.coord[0]+i, oth.coord[1]+j,false) )
75                  return true;
76              j++ ;
77          }
78          while (j < oth.s)
79          {
80              /* upper right edge */
81              if ( contains(oth.coord[0]+i, oth.coord[1]+j,false) )
82                  return true;
83              j++ ;
84              i-- ;
85          }
86          while (i > -oth.s)
87          {
88              /* top horizontal edge */
89              if ( contains(oth.coord[0]+i, oth.coord[1]+j,false) )
90                  return true;
91              i-- ;
92          }
93          while (j > 0)
94          {
```

```
95              /* upper left edge */
96              if ( contains(oth.coord[0]+i, oth.coord[1]+j,false) )
97                  return true;
98              j-- ;
99          }
100         while (i <= 0)
101         {
102             /* lower left edge */
103             if ( contains(oth.coord[0]+i, oth.coord[1]+j,false) )
104                 return true;
105             j-- ;
106             i++ ;
107         }
108         return false ;
109     } /* overlaps */
110
111     /** Check whether this hexagon is inside another.
112     * @param oth the other hexagon.
113     * @return true if each point inside this hexagon is inside the other hexagon.
114     *   Here inside means fully inside or on the rim.
115     */
116     boolean inside(Hexa oth)
117     {
118         /* We check only the 6 corner points of this being inside oth.
119         * (i,j)=(0,-s) lower left, (i,j)=(s,-s) lower right, (i,j)=(s,0) right,
120         * (i,j)=(0,s) upper right, (i,j)=(-s,s) upper left, (i,j)=(-s,0) left.
121         */
122         return ( oth.contains(coord[0], coord[1]-s,true)
123             && oth.contains(coord[0]+s, coord[1]-s,true)
124             && oth.contains(coord[0]+s, coord[1],true)
125             && oth.contains(coord[0], coord[1]+s,true)
126             && oth.contains(coord[0]-s, coord[1]+s,true)
127             && oth.contains(coord[0]-s, coord[1],true) ) ;
128     } /* inside */
129
130     /** Test whether (i,j) is in the hexagon.
131     * @param i coordinate along first unit vector
132     * @param j coordinat along second unit vector
133     * @param rimIsInside if true consider a point on the edges/rim as inside.
134     * @return True if the coordinate pair describes a triangle point that is inside the hexagon.
135     */
136     boolean contains(int i, int j, boolean rimIsInside)
137     {
138         /* reduce coordinates of the point relative to the center of this hexagon.
139         */
140         i -= coord[0] ;
141         j -= coord[1] ;
142         /* the effective size of the hexagon (which implies whether
143         * the limits on i and j include or exclude the 6 sides.)
144         */
145         final int e = (rimIsInside  ? s : (s-1) );
146
147         if ( i * j >= 0)
```

```
148         {
149             /* upper right triangle is 0<=i, 0<=j and i+j<=e (on the rim)
150             * lower right triangle is i<=0 j<=0 and |i|+|j|<=e (on the rim)
151             * both cases mean the same sign on i and j.
152             */
153             return ( Math.abs(i)+Math.abs(j) <= e) ;
154         }
155         else
156         {
157             /* Left and upper left triangles are -e<=i<=0 and 0<=j<=e (on the rim).
158             * Right and lower right triangles are -e<=j<=0 and 0<=i<=e (on the rim)
159             */
160             return ( Math.abs(i) <= e && Math.abs(j) <= e) ;
161         }
162     } /* contains */
163
164     /** Test whether (i,j) is in the hexagon.
165     * @param i coordinate along first unit vector
166     * @param j coordinat along second unit vector
167     * @return If the two coordinates describe a point inside the hexagon.
168     *   A point on the rim is considered inside.
169     */
170     boolean contains(double i, double j)
171     {
172         /* reduce coordinates of the point relative to the center of this hexagon.
173         */
174         i -= coord[0] ;
175         j -= coord[1] ;
176
177         /* calculation exactly as in the other Hexa.contains() but less forgivin
178         * with respect to floating point rounding. So therefore the application
179         * calls should consider moving slightly away from the full integer coordinates.
180         */
181         if ( i * j >= 0)
182         {
183             return ( Math.abs(i)+Math.abs(j) <= s) ;
184         }
185         else
186         {
187             return ( Math.abs(i) <= s && Math.abs(j) <= s) ;
188         }
189     } /* contains */
190
191     /** Calculate the Cartesian coordinates of two coordinates in the triangular lattice.
192     * @param i coordinate along the first unit vector of the triangular lattice.
193     * @param j coordinate along the second unit vector of the triangular lattice.
194     */
195     static double[] toCart(final int i, final int j)
196     {
197         /* the place is i*(1,0) + j*(1/2, sqrt(3)/2)
198         */
199         double[] c = {i+j/2.0, SQRT3_2*j} ;
200         return c ;
```

```
201     } /* toCart */
202
203     /** Consider hexagons the same if the have the same shape and center.
204     * @param oth The hexagon this is to be compared with.
205     * @return -1, 0 or +1 depending on whether this hexagon is considered smaller than,
206     *  equal to or larger than oth.
207     */
208     int compareTo(Hexa oth)
209     {
210         if ( s > oth.s)
211             return 1;
212         else if ( s < oth.s)
213             return -1;
214         else if ( coord[1] > oth.coord[1])
215             return 1 ;
216         else if ( coord[1] < oth.coord[1])
217             return -1;
218         else if ( coord[0] > oth.coord[0])
219             return 1;
220         else if ( coord[0] < oth.coord[0])
221             return -1;
222         else
223             return 0;
224     }
225
226     /*********************************
227     * Print Cartesian coordinates (one per line) of line segments in gnuplot style to stdout.
228     * @param avoid If not null, print also all triangles which are not
229     *  inside any of these hexagons.
230     */
231     void toGnuplot(Vector<Hexa> avoid)
232     {
233         /* Start with coordinates in the lower left corner and the
234         * work counter-clockwise to print each of the 6 edges.
235         * i and j are the unit-vector coordinates relative to the center.
236         */
237         int j = -s ;
238         int i = 0 ;
239         while (i < s)
240         {
241             /* bootom horizontal edge */
242             double[] c = toCart(coord[0]+i, coord[1]+j) ;
243             System.out.printf("%f %f\n",c[0], c[1]) ;
244             i++ ;
245         }
246         while (j < 0)
247         {
248             /* lower right edge */
249             double[] c = toCart(coord[0]+i, coord[1]+j) ;
250             System.out.printf("%f %f\n",c[0], c[1]) ;
251             j++ ;
252         }
253         while (j < s)
```

```
254         {
255             /* upper right edge */
256             double[] c = toCart(coord[0]+i, coord[1]+j) ;
257             System.out.printf("%f %f\n",c[0], c[1]) ;
258             j++ ;
259             i-- ;
260         }
261         while (i > -s)
262         {
263             /* top horizontal edge */
264             double[] c = toCart(coord[0]+i, coord[1]+j) ;
265             System.out.printf("%f %f\n",c[0], c[1]) ;
266             i-- ;
267         }
268         while (j > 0)
269         {
270             /* upper left edge */
271             double[] c = toCart(coord[0]+i, coord[1]+j) ;
272             System.out.printf("%f %f\n",c[0], c[1]) ;
273             j-- ;
274         }
275         while (i <= 0)
276         {
277             /* lower left edge */
278             double[] c = toCart(coord[0]+i, coord[1]+j) ;
279             System.out.printf("%f %f\n",c[0], c[1]) ;
280             j-- ;
281             i++ ;
282         }
283         System.out.printf("\n") ;
284
285         if ( avoid != null)
286         {
287             /* walk through all mid-points of the triangular edges inside
288             * this hexagon. Check whether they are inside any of the edges covered
289             * by the hexagons in the avoid set, and if not, print these triangular edges.
290             */
291             for(i= -s+coord[0] ; i <= s+coord[0] ; i++)
292             for(j= -s+coord[1] ; j <= s+coord[1] ; j++)
293             {
294                 if ( contains(i,j,true) )
295                 {
296                     /* check triangular edge from (i,j) to the down right direction */
297                     if ( contains(i+1,j-1,true) )
298                     {
299                         boolean inavoidset = false ;
300                         for( Hexa h: avoid)
301                         {
302                             if ( h.contains(i+0.5,j-0.5) )
303                             {
304                                 inavoidset = true;
305                                 break ;
306                             }
```

```
307                         }
308                         if ( !inavoidset)
309                         {
310                             double[] c = toCart(i,j) ;
311                             System.out.printf("%f %f\n",c[0], c[1]) ;
312                             c = toCart(i+1,j-1) ;
313                             System.out.printf("%f %f\n\n",c[0], c[1]) ;
314                         }
315                     }
316                     /* check triangular edge from (i,j) to the horizontal right direction */
317                     if ( contains(i+1,j,true) )
318                     {
319                         boolean inavoidset = false ;
320                         for( Hexa h: avoid)
321                         {
322                             if ( h.contains(i+0.5,j) )
323                             {
324                                 inavoidset = true;
325                                 break ;
326                             }
327                         }
328                         if ( !inavoidset)
329                         {
330                             double[] c = toCart(i,j) ;
331                             System.out.printf("%f %f\n",c[0], c[1]) ;
332                             c = toCart(i+1,j) ;
333                             System.out.printf("%f %f\n\n",c[0], c[1]) ;
334                         }
335                     }
336                     /* check triangular edge from (i,j) to the upwards right direction */
337                     if ( contains(i,j+1,true) )
338                     {
339                         boolean inavoidset = false ;
340                         for( Hexa h: avoid)
341                         {
342                             if ( h.contains(i,j+0.5) )
343                             {
344                                 inavoidset = true;
345                                 break ;
346                             }
347                         }
348                         if ( !inavoidset)
349                         {
350                             double[] c = toCart(i,j) ;
351                             System.out.printf("%f %f\n",c[0], c[1]) ;
352                             c = toCart(i,j+1) ;
353                             System.out.printf("%f %f\n\n",c[0], c[1]) ;
354                         }
355                     }
356                 }
357             }
358         }
```

```
359     } /* toGnuplot */
360 } /* Hexa */
```

Appendix C. Hexboard.java

```
1  import java.math.* ;
2  import java.util.* ;
3
4  /*****************************
5  * A Hexboard is a hexagon with function to count placements of smaller hexagons.
6  * @since 2016-08-24
7  * @author R. J. Mathar
8  */
9  class Hexboard extends Hexa
10 {
11     /* A value which triggers gnuplot support of the
12     * number of s-hexagons is that many in the n-hexagons.
13     * To disable the output, set it to some negative number so that criterion is not met.
14     */
15     int verb ;
16
17     /** The number of points of the triangular lattice which are fully inside
18     * the hexagon.
19     */
20     int insid ;
21
22     /** A counter for the placements found, starting at 0.
23     * This is only used to tag gnplot outputs such that
24     * gnuplot> plot "..." wi lines index ....
25     * commands are easier to construct.
26     */
27     int runidx ;
28
29     /** linij[idx][0..1] show where the idx'd inner point is
30     * in the i-direction [0] and in the j-direction [1] of the triangular unit vectors.
31     * This is calculated once as a look-up table.
32     */
33     int[][] linij ;
34
35     /************************
36     * @param e The edge length >=1 .
37     * @param v Verbosity. Gnuplot graph coordinates are included
38     *   in the standard output if the number of hexagons in solutions equals v.
39     */
40     Hexboard(int e, int v)
41     {
42         super(e,0,0) ;
43         verb=v ;
44         insid = pointsinsid() ;
45         runidx=0 ;
46         linij = new int[insid][];
47         for(int l =0 ; l< linij.length ; l++)
48             linij[l] =lin2ij(l) ;
```

```
49      } /* ctor */
50
51      /*********************************
52      * @param ssmall the edge length of the embedded hexagons
53      * @return A vector which shows in component k how many configurations with k small hexagons exist.
54      */
55      int[] count(int ssmall)
56      {
57          /* this starts the recursive call of building a tree-like
58          * attempt to add new hexagons at all possible inner points.
59          */
60          Vector<Hexa> given = new Vector<Hexa>() ;
61          return count(ssmall,0,given) ;
62      } /* count */
63
64      /*********************************
65      * Count placements of smaller hexagons with some space already occupied by other hexagons.
66      * @param small the edge length of the embedded hexagons
67      * @param linidx The lowest inner point to be tried next for placement.
68      * @param given The hexagons (of edge length small) already put into this hexagons.
69      * @return A vector which shows in component k how many configurations with k small hexagons exist.
70      */
71      int[] count(int small, int linidx,Vector<Hexa> given)
72      {
73          /* individual needs 6*small^2 triangles. board has 6*s^2,
74          * so count <= s^2/small^2. 1 more because 0 count is an entry in the array.
75          */
76          final int ctmax = s*s/(small*small) ;
77          int[] ct= new int[ctmax+1] ;
78
79          /* reached end of decision tree and we return a statistics
80          * which contains a 1 at the place in the ct[] vector that matches
81          * the number of hexagons actually placed
82          */
83          if ( linidx >= insid )
84          {
85              /* count number of given hexagons
86              */
87              int numsq = given.size() ;
88              ct[numsq]++ ;
89              if ( numsq == verb)
90              {
91                  System.out.println("# numsq " + numsq + " " + runidx++) ;
92                  /* plot the bare perimeter of this big hexagon
93                  * and the triangles not in any of the placed s-hexagons.
94                  */
95                  toGnuplot(given) ;
96                  /* plot the perimeters of all the small hexagons
97                  */
98                  for(int i=0 ; i < numsq ; i++)
99                      given.elementAt(i).toGnuplot(null) ;
100                 System.out.printf("\n") ;
101
```

```
102             }
103             return ct ;
104         }
105
106         /* try to place next at linidx. Get the hex-lattice integer coordinates of linidx.
107         */
108         int[] sqcoor = linij[linidx] ;
109         /* construct the potential candidate of the s-hexagon
110         */
111         Hexa sqnex = new Hexa(small, sqcoor[0],sqcoor[1]) ;
112         /* test whether the candidate is inside the n-hexagon, else skip
113         */
114         if ( sqnex.inside(this) )
115         {
116             /* test overlap with hexagons already placed.
117             */
118             boolean over = false;
119             for( Hexa g : given)
120                 if ( sqnex.overlaps(g) )
121                 {
122                     over = true ;
123                     break;
124                 }
125
126             if ( ! over)
127             {
128                 /* if there is overlap so the candidate can be placed,
129                 * continue with a tree-type recursive search for
130                 * configurations that contain this hexagon, too. First
131                 * add the candidate to the old hexagons to construct the
132                 * fixed hexagons for the recursive call.
133                 */
134                 Vector<Hexa> nexgiv = new Vector<Hexa>() ;
135                 for( Hexa g : given)
136                     nexgiv.add ( g) ;
137                 nexgiv.add(sqnex) ;
138                 /* start the recursive call and get its frequency statistics of k-values
139                 */
140                 int[] nxct = count(small,linidx+1,nexgiv) ;
141                 /* add that statistics for each k to the statistics to be returned.
142                 */
143                 for(int i=0 ; i < ct.length; i++)
144                     ct[i] += nxct[i] ;
145             }
146         }
147
148         /* skip to place at linidx, which is always an option independent
149         * of the potential obstrusion by fixed hexagons.
150         */
151         int[] nxCt = count(small,linidx+1,given) ;
152         for(int i=0 ; i < ct.length; i++)
153             ct[i] += nxCt[i] ;
154
```

```
155         return ct;
156     } /* count */
157
158     /** Map a linear index for all points fully inside this hexagon to the 2D triangular coordinates.
159     * @param lidx The index in the range 0 up to (excluding) the number of points inside this.
160     * @return The two triangular coordinates relative to the hexagon center.
161     */
162     int[] lin2ij(int lidx)
163     {
164         if ( lidx < 0 || lidx >= insid)
165             return null ;
166
167         int scanlin =0 ;
168         int[] ij = new int[2] ;
169         /* lower half: lower limit on x given by |x|+|y|<=s-1, x,y<=0, -x-y<s.
170         */
171         for ( ij[1] = -(s-1) ; ij[1] < 0 ; ij[1]++)
172         for ( ij[0] = -(s-1)-ij[1] ; ij[0] < s ; ij[0]++)
173         {
174             if ( scanlin == lidx)
175                 return ij ;
176             scanlin++ ;
177         }
178
179         /* upper half: upper limit on x given by |x|+|y|<=s-1, x,y>=0
180         */
181         for ( ij[1] = 0 ; ij[1] < s ; ij[1]++)
182         for ( ij[0] = -(s-1) ; ij[0] < s-ij[1] ; ij[0]++)
183         {
184             if ( scanlin == lidx)
185                 return ij ;
186             scanlin++ ;
187         }
188         return null ;
189     }
190
191     /* map a linear index for all points fully inside s to a center
192     * coordinate for the ssmall hexagnos without yet paying attention to the size of them
193     * This gives 0 for s=0, 1 for s=1, 7 for s=2, 19 for s=3... 3*(s-1)*s+1 see A003215
194     */
195     int pointsinsid()
196     {
197         if ( s <= 0 )
198             return 0;
199         else
200             return 1+3*(s-1)*s ;
201     } /* pointsinsid */
202
203     /** Main counter program for placing s-hexagons in n-hexagons.
204     * The usage is (to compile)
205     *   javac -cp . Hexboard.java Hexa.java
206     * and to run
207     *   javac -cp . Hexboard s n
```

```
208     * as detailed below.
209     * @param argv The command line arguments.
210     *  This contains up to one option followed by two mandatory integer arguments.
211     *  the option is -v followed by the number of hexagons that should trigger gnuplot output.
212     *  The two integer arguments are s (edge length of the smaller hexagons) and n
213     *  (edge length of the larger hexagon).
214     */
215     public static void main(String[] argv)
216     {
217         /* disable gnuplot output for illustrations.
218         */
219         int verb = -1 ;
220         /* gather command line arguments
221         */
222         int optindx =0 ;
223         if ( argv[0].startsWith("-v") )
224         {
225             optindx++ ;
226             verb = Integer.parseInt(argv[optindx++]) ;
227         }
228         int s = Integer.parseInt(argv[optindx++]) ;
229         int n = Integer.parseInt(argv[optindx++]) ;
230 
231         /* construct the large hexagon
232         */
233         Hexboard b = new Hexboard(n,verb) ;
234 
235         /* fill the statistics of placing s-hexagons in it
236         */
237         int[] ct = b.count(s) ;
238         /* print the statistics
239         */
240         int kmax = 0 ;
241         BigInteger ctsum = BigInteger.ZERO ;
242         System.out.printf(" %d %d: ",s,n) ;
243         for(int i=0 ; i < ct.length; i++)
244         {
245             System.out.printf(" %d",ct[i]) ;
246             ctsum = ctsum.add(new BigInteger(""+ct[i])) ;
247             if ( ct[i] >0 )
248                 kmax = i ;
249         }
250         System.out.printf(" : %d %d\n",ctsum,kmax) ;
251 
252     } /* main */
253 } /* Hexboard */
```

References

1. A. Grosso, A. R. M. J. U. Jamali, M. Locatelli, and F. Schoen, *Solving the problem of packing equal and unequal circles in a circular container*, J. Glob. Optim. **47** (2010), 63–81.
2. Wein Qui Huang and Tao Ye, *Quasi-physical global optimization method for solving the equal circle packing problem*, Sci. Chin. Inf. Sci. **54** (2011), no. 7, 1333–1339.

3. Neil J. A. Sloane, *The On-Line Encyclopedia Of Integer Sequences*, Notices Am. Math. Soc. **50** (2003), no. 8, 912–915, http://oeis.org/. MR 1992789 (2004f:11151)

URL: `http://www.mpia.de/~mathar`

Hoeschstr. 7, 52372 Kreuzau, Germany