Theor Chim Acta (1993) 86: 269–278

Theoretica Chimica Acta

On the number of square-cell configurations

Wolfgang R. Müller[1], Klaus Szymanski[1], Jan V. Knop[1], and Nenad Trinajstić[2]

[1] Computer Centre, The Heinrich Heine University, 4000 Düsseldorf, Germany

[2] The Rugjer Bošković Institute, P.O.B 1016, 41001 Zagreb, Croatia

Received December 27, 1991/Accepted December 7, 1992

Summary. The numbers of simply and multiply connected square-cell configurations are computed. The computation is based on the original algorithm for constructive enumeration of animals which is founded on the DAST (dualist angle-restricted spanning tree) code.

Key words: Square animals – Animal code – Constructive enumeration

1. Introduction

Recently in this journal Harary and Mezey [1] reported a study on similarity and complexity of shapes of square-cell configurations. Another name for these configurations is square animals [e.g., 2]. A square animal is made up of squares which are simply or multiply connected [3, 4]. It starts with a single square and grows by adding squares one at a time in such a way that the new square has at least one side in contact with a side of a square already present in the animal. Square animals are simply connected if they have no holes, whilst multiply connected square animals are configurations with holes (Harary and Palmer [2] called them holey animals). The smallest hole is of the size of the square. In Fig. 1 we give as examples a simply connected square animal with 8 squares and a multiply connected square animal with 10 squares.

Statistical properties of square animals and their embeddings in square lattices are important in modelling a variety of physical problems such as the thermodynamic properties of polymers in dilute solution [5–9] and for characterizing shapes of two-dimensional solids and molecular aggregates on the surfaces of catalysts [e.g. 10].

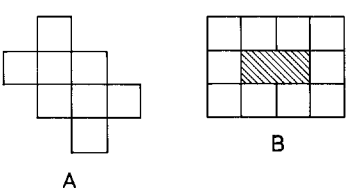


Fig. 1. A simply connected square animal with 8 cells (*A*) and multiply connected square animal with 10 cells (*B*)

The problem is to find the number of different square animals with n-squares [2, 11]. Two or more square-cell configurations which can be transformed into each other by translations, reflections or rotations in the plane are regarded as the same animal. This problem of counting square animals belongs to the celebrated, but difficult, cell-growth problem [2, 3, 11, 12] or polyomino problem [13–15].

The question concerning the exact number of square animals of a given finite cell number n is an "old" problem. It was listed by Harary in 1960 in his article on unsolved problems in the enumeration of graphs [11]. In this article he has also given the counting series for the simply connected square animals with up to 7 cells: $x + x^2 + 2x^3 + 5x^4 + 12x^5 + 35x^6 + 107x^7$. A few years later (1964) Harary has published in his book on applied combinatorics the counting series for the lower simply and multiply connected square animals [16]. Since we are in the position to provide the numbers and shapes of animals with a given number of cells [17], we will give here the numbers of square animals for larger values of n.

We have recently developed an algorithm for the constructive enumeration of hexagonal animals [18], which is based on the DAST (dualist angle-restricted spanning tree) code [19, 20]. It appears that the DAST code can be used with a slight modification for representing square animals. At this point we mention that the constructive enumerations for each finite n can also be accomplished by the Harary–Mezey code [1], but we decided to use our code because the preliminary results based on it were already available.

2 Definition of the DAST code for square animals

The name "polyomino" (generalization of domino for square counts other than 2) shall in the following stand for planar square animal, i.e. a square-cell configuration which describes a finite edge-connected subset on an unbounded chessboard. Here edge-connectivity shall be the transitive closure of the "having a common edge" neighborship relation for squares.

We will use – where this distinction is vital – the name "polyomino graph" for a polyomino as a system of vertices and edges, which therefore cannot have holes of the size of a square and "polyomino area" for a polyomino as a system of filled squares, which can have a single empty space in the center of a ring of eight squares. Obviously the set of polyomino graphs can be embedded in the set of polyomino areas as a subset by simply filling all possible squares.

For a systematic approach we begin by defining an "entered polyomino" (P, a, b) as a polyomino P together with an ordered pair (a, b) of vertices adjacent on its boundary. We name this pair the "entrance" of the entered polyomino and the square to which it belongs the "entrance square". We further name (P, a, b) a "corner-entered polyomino", if – with the obvious coordinate system – vertex a has the maximal/minimal value in one coordinate among those vertices having the maximal/minimal value in the other coordinate, i.e., if – after a suitable rotation or reflection of the polyomino – a lies exactly north of b, no other vertex lies so, and no vertex lies further to the west ("a being corner" is a necessary but not a sufficient condition for this). Evidently there are at most 8 (non-isomorphic) corner-entered polyominoes for every polyomino (which coordinate first * minimal or maximal first coordinate * minimal or maximal second) and less for symmetric cases. Corner-entered polyominoes are essentially what

other authors name fixed polyominoes, and those authors name polyominoes in our sense free polyominoes.

Let (P, a, b) be an entered polyomino and let the vertex sequence (a, b, c, d, a) describe a path around the entrance square. According to the definition there are only three neighboring squares – N_1, N_2, N_3 – possible beyond the edges different from the entrance edge. This gives rise to a decomposition of the polyomino area into at most four disjoint regions:

P_0: the entrance square itself;

P_1: the edge-connectivity component of N_1 (if present) after elimination of P_0, N_2 and N_3;

P_2: the edge-connectivity component of N_2 (if present) after elimination of P_0, P_1, and N_3; and

P_3: the edge-connectivity component of N_3 (if present) after elimination of P_0, P_1, and P_2;

This decomposition clearly cannot be independent of the ordering of the neighbors. So there must be an arbitrary but then fixed convention about this ordering. In order to encode the presence or absence of any of the components in the three bits of an octal digit there must also be a convention about the mapping. We found it most useful to look first between a and d with weight 4, then between c and d with weight 1 and lastly between b and c with weight 2 (see Fig. 2). This gives at most three smaller entered polyominoes (P_1, a, d), (P_2, d, c), and (P_3, c, b).

Now we can define the "DAST tuple" of an entered polyomino (with respect to the above convention) by induction: The only entered polyomino with one square gets 0, the tuple consisting of a single zero, as its DAST tuple (this could also be deduced from the induction rule which follows).

Let the DAST tuples of all entered polyominoes with at most k squares be defined, and (P, a, b) be an entered polyomino with $k + 1$ squares. Then the decomposition described above gives rise to at most three smaller entered polyominoes which by induction have DAST tuples. We add the weights of the neighbors to get a digit from 0 to 7 and append to it the DAST tuples of the components (if present) ordered according to the convention. This gives the DAST tuple of (P, a, b).

For a given polyomino we define the "DAST code" as the lexicographic minimum of the DAST tuples corresponding to its 8 corner-entered polyominoes. This definition applies equally to polyomino graphs and to polyomino areas, and different polyominoes lead to different codes. A polyomino is completely reconstructable from its DAST code.

Like the n-tuple representation of trees [21] the DAST code is selfterminating, i.e., if a well-formed DAST code is hidden by appending digits to the end,

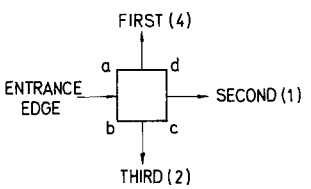


Fig. 2. Ordering of directions and their weights (in parentheses)

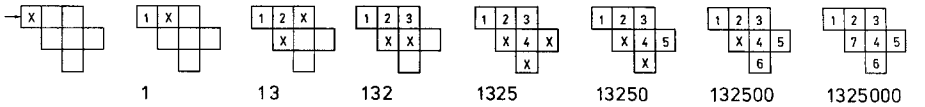


Fig. 3. The step-by-step development of the DAST code for a given square animal. The label in a square denotes the position in the DAST code of the octal digit corresponding to this square. *X* indicates a square already reserved for the later processing

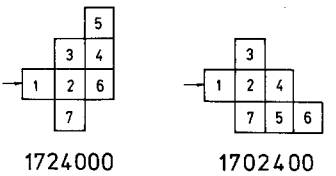


Fig. 4. The DAST tuple for another two orientations (90° and 180°) of the animal with 7 squares given in Fig. 3

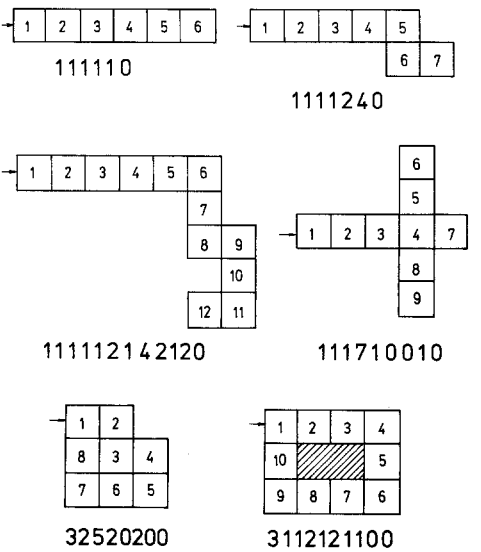


Fig. 5. Examples of several square-cell configurations with their DAST codes

then without further information the original end of the DAST code can be determined. This allows an encoding of sets of independent polyominoes by simply concatenating their DAST codes. As an example we give in Fig. 3 the step-by-step development, obeying the convention from the above, of the DAST code for a square animal with 7 cells.

The DAST code for the square animal in Fig. 3 is lexicographically the smallest of all possible codes for this particular animal. For example, if we rotate clockwise this animal for 90° and for 180°, then the corresponding DAST tuples are given by 1724000 and 1702400, respectively (see Fig. 4).

In Fig. 5 we give several additional examples of square-cell configurations with their DAST code.

3 The computer program

The DAST code is the basis of a computer program for generating and thereby enumerating all polyominoes with up to a given number of squares. To generate

all polyominoes with up to n squares the program virtually counts all tuples with up to n octal digits (i.e., values from 0 to 7) in lexicographically ascending order. During the counting the program tries to interpret the actual tuple as a DAST tuple of a corner-entered polyomino and eliminates it, if it

(a) self-terminates before its end (only $k-1$ one-bits in the first k digits of the tuple) or

(b) does not self-terminate at its end or

(c) describes the same square twice or more or

(d) describes a square as a neighbor from another square but the first one from which this would be possible or

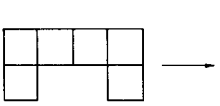
(e) starts from a wrong square (not a corner entrance).

Actually all these tests and countings can be combined so efficiently that the real counting through these tuples leads from one case passing all these tests to the next such case in one step of nearly constant CPU time expense. The result is sequential delivery of the DAST tuple for every corner-entered polyomino. Clearly the DAST codes of all polyominoes are among them. To eliminate all others every DAST tuple is simply compared to the DAST code of its polyomino. Thus, the program now delivers sequentially for every polyomino exactly once its DAST code at an average rate of about 8 counting steps per hit (8 orientations, symmetric cases are negligibly rare) but varying extremely from 1 step to million steps per hit. The overall time expense of the algorithm is directly proportional to the number of all squares in all generated polyominoes.

The program uses the obvious coordinate system (i.e., the axes originate at the center of a square and run parallel to the square edges) in the plane filled with squares, such that the square centers are in 1-1 correspondence with the pairs of integers representing coordinate values. This allows coding of each square in a finite section of the plane by a two-dimensional array (see Fig. 6 where '1' stands for "belongs to polyomino" and '.' for "does not").

The generating process uses 4 different states of squares (and thus four values in the array): "free", "blocked", "member" and "reached" (the choice of these values is merely a matter of taste). The four possible directions of propagation to adjacent squares can be seen as index differences $(1, 0)$, $(0, 1)$, $(-1, 0)$ and $(0, -1)$.

We assume the starting (entrance) square (denoted by "S" in Fig. 7) at position $(0, 0)$ in the center of the array (there are also allowed positions of squares with negative indices) and a starting direction to be $(1, 0)$. We enforce the starting point to be one of the 8 corner entrances by marking "blocked" (denoted by "B" in Fig. 7) all squares at coordinates (k, l) with $k = 0$, $l > 0$ (exactly northern) or $k = -1$, $l \leqslant 0$ (southwestern). This rails off half of the plane (see Fig. 7).



```
. . . . . .
. 1 1 1 1 .
. 1 . . 1 .
. . . . . .
```

Fig. 6. An example of a square animal and the corresponding two-dimensional array

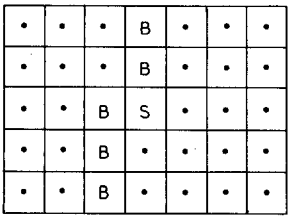


Fig. 7. An array of squares with the starting square (*S*) and blocked squares (*B*)

The starting square is marked as "member", the remaining squares being "free". We push the starting position and direction onto a stack and enter the recursive process, which operates as follows: If the stack is not empty, we take position and direction of a member from the stack and make them the current ones. We reserve the state of the three neighboring squares of the current one in the currently allowed direction (current straightforward direction and 90° to the left or to the right). Each subset of free squares among these neighbors (taking into account the maximum number of squares required) is considered, one case at a time. For each case we build the corresponding 3-bit binary number and insert it as a digit in the code in generation. We mark the neighbors in the subset as "members" and the former "free" ones as "blocked" so that they cannot be made "members" from another side (which would lead to a code different from the DAST code). We push the position and direction of the new "members" (if any) onto the stack such that they can be fitted in the chosen order of directions. Then for every case we reenter the recursion. After all cases have been inspected, we reset the three neighbors to their reserved state and trace back one recursion level. If the stack is empty, we have completed the generation of the DAST tuple of a corner-entered polyomino.

For all other 7 orientations we select the right-corner entrance and compute the DAST tuple (using suitable index transformations and the two-dimensional array in a recursive process like that above but simpler since the only case to consider is the maximum subset of members among the three neighboring squres, and, using the fourth state "reached" to mark temporarily members which were entered or reserved). If the original tuple is the smallest among the eight, we use it further in the generating process as the DAST code of a new polyomino area.

In order to detect any holes (especially to distinguish polyomino graphs from polyomino areas) we transform the DAST code into a boundary description [22]. We walk clockwise around every square making up the polyomino. The walk begins at the entrance edge. For this $4n$ (n = the number of squares in the polyomino) walk we eliminate all self-returning subwalks of length 2 (this elimination can already be done during the transformation process, i.e., referring to Fig. 2 we replace in the walk the part *a-b* directly by *a-d-c-b* and not by *a-b-a-d-c-b*). If the remaining walk contains closed subwalks (or the same vertex twice, which is easily determined using a two-dimensional array storing for every vertex position in the plane the last edge sequence number ending at this vertex), we have a hole (of size 1, if the sequence numbers differ by 4). If there are no holes, we also obtain a boundary code of the polyomino (not necessarily a canonical one).

We give a block-diagram of the computer program for generation and enumeration of square-cell configurations in Fig. 8.

Though not very suitable for vector processors the program matches ideally the possibilities of parallel processing hardware, as one may use as many

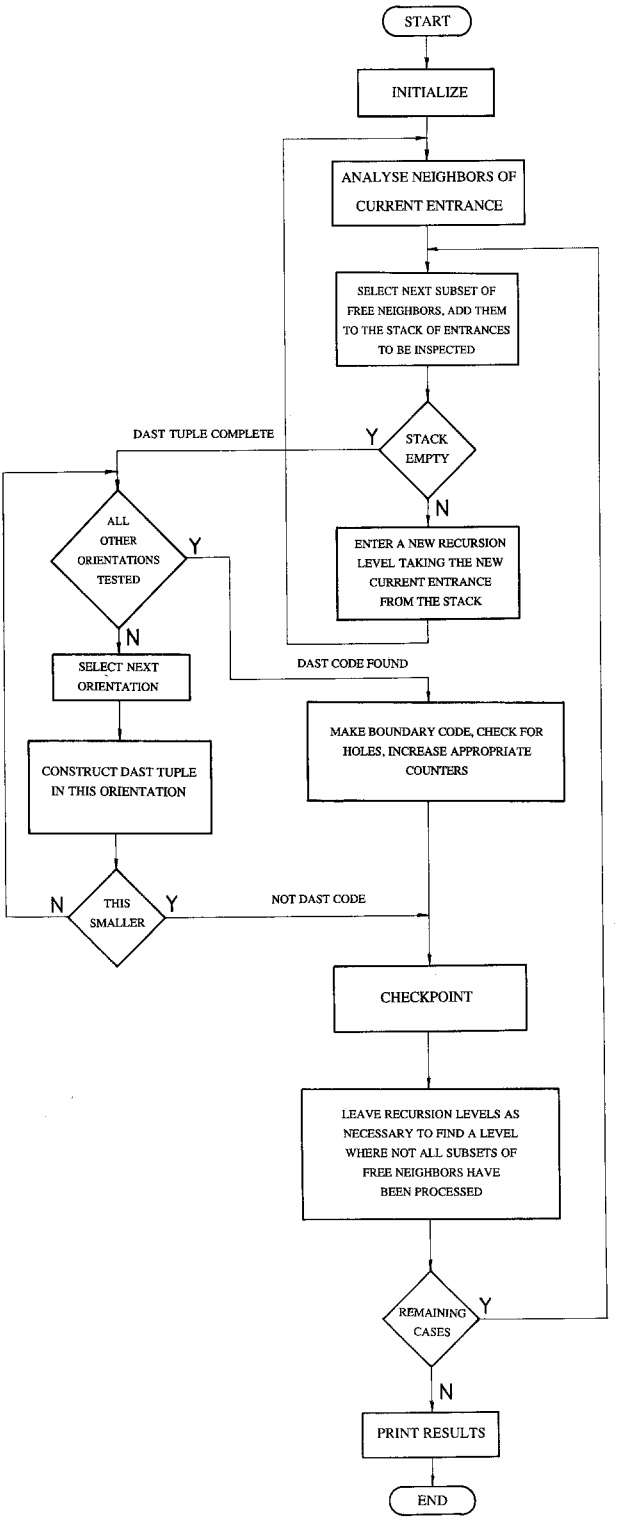


Fig. 8. A block-diagram of the computer program

computers as one wishes at the same time on disjoint intervals of tuple values. The only information needed to continue an interrupted generation process is the last known tuple produced before the interruption.

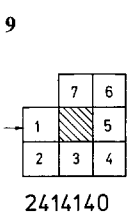
4 Results and discussion

In Table 1 we give the numbers of square-cell configurations with up to 16 squares.

Computations have been carried out on a PC (386-AT, 40 MHz). The CPU times needed to complete computations are also reported in the table. The smallest polyomino area with a hole (the smallest holey square animal [2]) is a configuration with 7 cells which is depicted in Fig. 9.

Table 1. The number of square animals with n cells

Square animal									
n	Simply connected	Multiply connected				Grand total	cpu time		
		Single square hole	Single large hole	Several large holes	Total		h	min	s
1	1					1			0.33
2	1					1			0.27
3	2					2			0.33
4	5					5			0.33
5	12					12			0.33
6	35					35			0.49
7	107	1			1	108			0.94
8	363	6			6	369			2.64
9	1248	36	1		37	1285			8.85
10	4460	182	13		195	4655			25.16
11	16094	884	95		979	17073		1	7.84
12	58973	4074	589		4663	63600		4	24.85
13	217117	18254	3220		21474	238591		17	18.42
14	805475	80008	16486	2	96496	901971	1	8	13.87
15	3001127	345415	79997	37	425449	3426576	4	29	49.84
16	11230003	1474145	374628	479	1849252	13079255	17	50	5.00



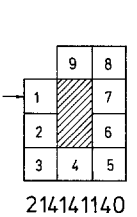


Fig. 9. The smallest holey square animal and its DAST code

Fig. 10. The smallest multiply connected square animal with a single hole of the size of two squares and its DAST code

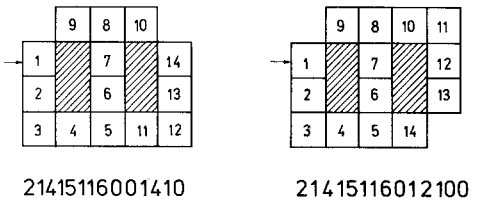


Fig. 11. Two smallest multiply connected square animals with two holes of the size of two squares and their DAST codes

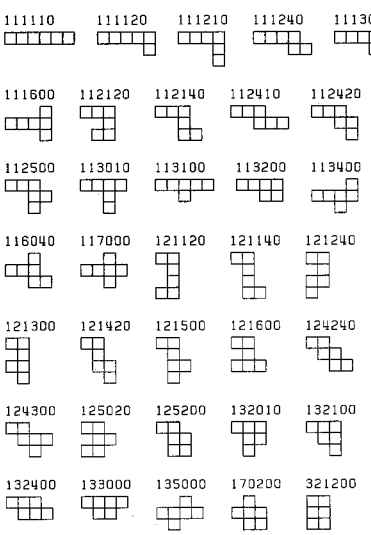


Fig. 12. A copy of the computer output containing square animals with 6 cells and their DAST codes. These animals are ordered according to the lexicographically increasing codes

The smallest polyomino graph with a hole (the smallest multiply connected animal with a single large hole, i.e., a hole of the size of two squares) appears in the class of square-cell configurations with 9 cells. This square animal is shown in Fig. 10.

There are two smallest polyomino graphs with at least two holes, i.e., two smallest multiply connected square animals with two large holes. They appear in the class of square-cell configurations with 14 cells. These two 14-square animals are given in Fig. 11.

Our results in Table 1 are in full agreement with several previous computations which produce the total number of square animals [2, 11, 16, 23–25]. The feature we emphasize in our algorithm is to get every free polyomino exactly once, and so we spend most of the CPU time in deciding whether the actual fixed polyomino is the right representative for the corresponding free one. This allows us to do further operations on all free polyominoes of a given size, e.g., to produce graphic output. In Fig. 12 we give as an example a copy of the computer output containing the diagrams of 35 square-cell configurations (all square animals with 6 cells).

Acknowledgements. One of us (NT) was supported by the Ministry of Science, Technology and Informatics of the Republic of Croatia via Grant No. 1-07-159. We are thankful to the referees for their constructive comments.

References

1. Harary F, Mezey (1991) Theoret Chim Acta 79:379
2. Harary F, Palmer EM (1973) Graphical enumeration. Academic Press, NY, p 234
3. Read RC (1962) Can J Math 14:1
4. Harary F (1971) Graph theory, 2nd printing. Addison-Wesley, Reading, MA, p 194
5. Whittington SG (1987) in: Lacher RC (ed) MATH/CHEM/COMP 1987. Elsevier, Amsterdam, p 285
6. Soteros CE, Whittington SG (1988) J Phys A 21:2187
7. Madras N, Soteros CE, Whittington SG (1988) J Phys A 21:4617
8. Whittington SG, Soteros CE, Madras N (1991) J Math Chem 7:87
9. Brak R, Guttman AJ, Whittington SG (1991) J Math Chem 8:255
10. Silverberg M, Ben-Shaul A (1987) J Chem Phys 87:3178
11. Harary F (1960) Publ Math Inst Hungarian Acad Sci 5:63
12. Palmer EM (1972) Lecture Notes in Mathematics 303:215
13. Golomb SW (1965) Polyominoes. Scribner, NY
14. Klarner DA (1965) Fibonacci Quart 3:9
15. Delest M (1991) J Math Chem 8:3
16. Harary F (1964) Applied Combinatorics. Wiley, NY, p 200
17. Trinajstić N, Nikolić S, Knop JV, Müller WR, Szymanski K (1991) Computational chemical graph theory: Characterization, enumeration and generation of chemical structures by computer methods. Simon & Schuster, NY
18. Müller WR, Szymanski K, Knop JV, Nikolić S, Trinajstić N (1990) J Comput Chem 11:223
19. Nikolić S, Trinajstić N, Knop JV, Müller WR, Szymanski K (1990) J Math Chem 4:357
20. Knop JV, Müller WR, Szymanski K, Nikolić S, Trinajstić N (1991) in: Rouvray DH (ed) Computational chemical graph theory. Nova Science Publ, NY, p 9
21. Knop JV, Müller WR, Jeričević Ž, Trinajstić N (1981) J Chem Inf Comput Sci 21:91
22. Knop JV, Szymanski K, Jeričević Ž, Trinajstić N (1983) J Comput Chem 4:23
23. Klarner DA (1967) Can J Math 19:851
24. Read RC (1978) in: Beineke LW, Wilson RJ (eds) Selected topics in graph theory. Academic Press, London, p 417
25. Redelmeier DH (1981) Discrete Math 36:191