Counting Polyominoes: A Parallel Implementation for Cluster Computing

Iwan Jensen

Department of Mathematics and Statistics
The University of Melbourne, Victoria 3010, Australia
I.Jensen@ms.unimelb.edu.au
http://www.ms.unimelb.edu.au/~iwan/

Abstract. The exact enumeration of most interesting combinatorial problems has exponential computational complexity. The finite-lattice method reduces this complexity for most two-dimensional problems. The basic idea is to enumerate the problem on small finite lattices using a transfer-matrix formalism. Systematically grow the size of the lattices and combine the results in order to obtain the desired series for the infinite lattice limit. We have developed a parallel algorithm for the enumeration of polyominoes, which are connected sets of lattice cells joined at an edge. The algorithm implements the finite-lattice method and associated transfer-matrix calculations in a very efficient parallel setup. Test runs of the algorithm on a HP server cluster indicates that in this environment the algorithm scales perfectly from 2 to 64 processors.

1 Introduction

The enumeration of polyominoes is a classical combinatorial problem [1]. A polyomino is a connected set of lattice cells joined at their edges. The fundamental problem is the calculation (up to translation) of the number of *fixed* polyominoes, a_n, with n cells. If we also take into account rotation and reflection symmetries we arrive at *free* polyominoes. Thus $\square\square$ and $\begin{smallmatrix}\square\\\square\end{smallmatrix}$ count as different fixed polyominoes, but they are the same free polyomino, while $\begin{smallmatrix}\square\\\square\square\end{smallmatrix}$ isn't a polyomino because the two cells don't share an edge. The enumeration of polyominoes has traditionally served as a benchmark for computer performance and algorithm design [2,3,4,5,6,7,8,9].

Polyominoes are closely related to lattice animals. A *site* animal can be viewed as a finite set of lattice sites connected by a network of nearest neighbor bonds. Polyominoes are thus identical to site animals on the dual lattice. In the physics literature lattice animals are often called *clusters* due to their close relationship to percolation problems [10]. Series expansions for various percolation properties, such as the percolation probability or the average cluster size, can be obtained as weighted sums over the number of lattice animals, $g_{n,m}$, enumerated according to the number of sites n and perimeter m [11].

P.M.A. Sloot et al. (Eds.): ICCS 2003, LNCS 2659, pp. 203–212, 2003.

It has been proven [12] that the number of polyominoes grows exponentially

$$\lim_{n\to\infty} n^{-1} \ln a_n = \sup_{n>0} n^{-1} \ln a_n = \lambda \tag{1}$$

where λ is the *growth constant*. The best numerical estimate, based on an analysis of the series for the generating function up to order 46, is $\lambda = 4.062570(8)$ [13], while rigorous lower and upper bounds shows that $3.903184\ldots \leq \lambda \leq 4.649551\ldots$, where the lower bound was derived in [13] from the aforementioned series using a method developed by Rands and Welsh [14] and the upper bound is due to Klarner and Riverst [15]. It is generally believed, though not rigorously proven, that there is a power-law correction to the exponential growth

$$a_n \simeq A\lambda^n n^{-\theta}. \tag{2}$$

where θ is called the *entropic exponent*. It is generally believed [16] from theoretical arguments that $\theta = 1$, a prediction, which is overwhelmingly supported by the available numerical evidence [13].

As with most interesting unsolved combinatorial problems the enumeration of polyominoes is of exponential computational complexity. That is the time $T(n)$ required to calculate the first n terms grows like $T(n) \propto \kappa^n$. The finite-lattice method (FLM) reduces the value of κ for two-dimensional problems. The basic idea is to enumerate the problem on small finite lattices using a transfer-matrix (TM) formalism. Systematically grow the size of the lattices and combine the results in order to obtain the desired series for the infinite lattice limit.

Sykes and Glen [11] calculated $g_{n,m}$ up to $n = 19$ on the square lattice, and thus obtained the number of polyominoes, $a_n = \sum_m g_{n,m}$, to the same order. Redelmeier [4] presented an improved algorithm for the enumeration of polyominoes and extended the results to $n = 24$. This algorithm was later used by Mertens [6] to devise an improved algorithm for the calculation of $g_{n,m}$ and a parallel version of the algorithm appeared a few years later [7]. All of the above algorithms were variations on direct counting and their computational complexity thus grows as $T(n) = \lambda^n$. A major advance was obtained by Conway [8] who used the FLM to calculate a_n and numerous other series up to $n = 25$ [17]. For this algorithm the computational complexity was $T(n) = 3^{n/2}$. In unpublished work Oliveira e Silva [18] used the parallel version of the Redelmeier algorithm to extend the enumeration to $n = 28$. In [9] we used an improved version of Conway's algorithm to extend the enumeration to $n = 46$. This improved algorithm has complexity $T(n) = \kappa^{n/2}$ with $\kappa \simeq 2$. Knuth [19] improved the algorithm somewhat further and managed to obtain one further term.

In this paper we describe a parallel implementation of the TM algorithm for the enumeration of polyominoes. The algorithm implements the finite-lattice method and associated transfer-matrix calculations in a very efficient parallel setup. Test runs of the algorithm on a HP server cluster indicates that in this environment the algorithm scales perfectly from 2 to 64 processors.

2 Enumeration of Polyominoes

The method we use to enumerate polyominoes on the square lattice is based on the method used by Conway [8] for the calculation of series expansions for percolation problems, and is similar to methods devised by Enting for enumeration of self-avoiding polygons [20]. In the following we give a brief description of the algorithm used to count polyominoes. We then give some details of the parallel version of the algorithm.

2.1 Transfer Matrix Algorithm

The number of fixed polyominoes that span rectangles of width W and length L are counted using a transfer matrix algorithm. By combining the results for all $W \times L$ rectangles with $W \leq W_{\max}$ and $W + L \leq 2W_{\max} + 1$ we can count all polyominoes up to $N_{\max} = 2W_{\max}$. Due to symmetry we only consider rectangles with $L \geq W$ while counting the contributions for rectangles with $L > W$ twice. The maximal size $N_{\max}$ up to which one can count the number of polyominoes is limited by the available computational resources.

The transfer matrix technique involves drawing an intersection through the rectangle cutting through a set of W cells. Polyominoes in rectangles of a given width are counted by moving the intersection so as to add one cell at a time, as shown in Fig. 1. For each configuration of occupied or empty cells along the intersection we maintain a truncated generating function for partially completed polyominoes. Each configuration can be represented by a set of states $S = \{\sigma_i\}$, where the value of the state σ_i at position i must indicate first of all if the cell is occupied or empty. An empty cell is simply indicated by $\sigma_i = 0$. Since we have to ensure that we count only connected graphs more information is required if a cell is occupied. We need a way of describing which occupied cells along the intersection are connected to one another via a set of occupied cells to the left of the intersection. The most compact encoding of this connectivity is [8]

$$\sigma_i = \begin{cases} 0 & \text{empty cell,} \\ 1 & \text{occupied cell not connected to others,} \\ 2 & \text{first among a set of connected cells,} \\ 3 & \text{intermediate among a set of connected cells,} \\ 4 & \text{last among a set of connected cells.} \end{cases} \tag{3}$$

Configurations are read from the bottom to the top. As an example the configuration along the intersection of the partially completed polyomino in Fig. 1 is $S = \{102023404\}$.

Pruning. In the original approach [8] polyominoes were required to span the rectangle only length-wise and polyominoes of width $\leq W$ and length L were counted several times. It is however easy to obtain polyominoes of width exactly W and length L from such data [20]. In our algorithm we directly enumerate

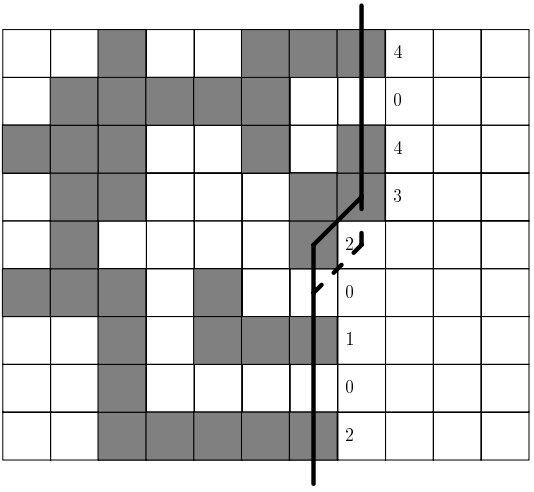

Fig. 1. A snapshot of the intersection (solid line) during the transfer matrix calculation on the square lattice. Polyominoes are enumerated by successive moves of the kink in the intersection, as exemplified by the position given by the dashed line, so that one cell at a time is added to the rectangle. To the left of the intersection we have drawn using shaded squares an example of a partially completed polyomino. Numbers along the intersection indicate the encoding of this particular configuration

polyominoes of width exactly W and length L. At first glance this would appear to be inefficient since for many intersection configurations we now have to keep 4 distinct generating functions depending on which borders have been touched. However, as demonstrated in practice [9] it actually leads to an algorithm which is both exponentially faster and whose memory requirement is exponentially smaller. Realizing the full savings in time and memory usage require enhancements to the original algorithm. The most important is what we call *pruning*. This procedure allows us to discard most of the possible configurations for large W because they only contribute to polyominoes of size greater than $N_{\max}$. Briefly this works as follows. Firstly, for each configuration we keep track of the current minimum number of cells N_{cur} already inserted to the left of the intersection. Secondly, we calculate the minimum number of additional cells N_{add} required to produce a valid polyomino. There are three contributions, namely the number of cells required to connect all sections of the partially completed polyomino, the number of cells needed (if any) to ensure that the polyomino touches both the lower and upper border, and finally the number of steps needed (if any) to extend at least W cells in the length-wise direction (remember we only need rectangles with $L \geq W$). If the sum $N_{\mathrm{cur}} + N_{\mathrm{add}} > N_{\max}$ we can discard the partial generating function for that configuration, and of course the configuration itself, because it won't make a contribution to the polyomino count up to the size we are trying to obtain.

The Updating Rules. In Table 1 we have listed the possible local 'input' states and the 'output' states which arise as the kink in the intersection is propagated by one step. The most important cell on the intersection is the 'lower' one situated at the bottom of the kink (the cell marked with the second '2' (counting from the bottom) in Fig. 1). This is the position in which the lattice is being extended. Obviously the new cell can be either empty or occupied. The state of the upper cell (the cell marked '3' in Fig. 1) is likely to be changed as a result of the move. In addition the state of a cell further afield may have to be changed if a branch of a partially completed polyomino terminates at the new cell or if two independent sections of a partially completed polyomino join at the new cell.

Details of how these updating rules are derived can be found in [9]. Here a few comments will have to suffice.

10: The lower cell is an isolated occupied cell and the new cell can be empty only if there are no other occupied cells on the intersection (otherwise we generate graphs with separate components) and if both the lower and upper borders have been touched. The result are valid polyominoes and the partial generating function is added to the total polyomino generating function.

14: This situation never occurs. The upper cell is last among a set of occupied cells, so the cell immediately to its left is also occupied, this in turn is connected to the lower cell, which therefore cannot be an isolated cell.

20: The lower cell is first among a set of occupied cells, so if the new cell is empty, another cell in the set changes its state. Either the *first* intermediate cell becomes the new first cell, and its state is changed from 3 to 2, or, if there are no intermediate cells, the last cell becomes an isolated cell, and its state is changed from 4 to 1. This relabeling of a matching cell is indicated in Table 1 by over-lining.

22: When the new cell is occupied two separate pieces of the polyomino are joined. The new cell remains the first cell in the joined piece while the upper cell becomes an intermediate cell. The last cell in the innermost set of connected cells also becomes an intermediate cell in the joined piece. We indicate this type of transformation by putting a hat over the string.

Table 1. The various 'input' states and the 'output' states which arise as the intersection is moved in order to include one more cell of the lattice. Each panel contains two 'output' states where the left (right) most is the configuration in which the new cell is empty (occupied)

Lower \ Upper	0	1	2	3	4
0	00 10	01 24	02 23	03 33	04 34
1	add 10	− 24	− 23	− 33	
2	$\overline{00}$ 20	$\overline{01}$ 23	$\overline{02}$ $\widehat{23}$	02 23	01 24
3	00 30	01 33	02 $\widehat{33}$	03 33	04 34
4	$\overline{00}$ 40	$\overline{01}$ 34	$\overline{02}$ 33	$\overline{03}$ $\widehat{33}$	

Computational Complexity. The algorithm has exponential complexity, that is the time required to obtain the polyominoes up to size n grows exponentially with n. Time and memory requirements are basically proportional to the maximum number of distinct configuration generated during a calculation. In [9] we s showed that the maximal number of configurations, N_{Conf}, grows with $W_{\max}$ as $N_{\mathrm{Conf}} \propto \kappa^{W_{\max}}$, and from the numerical data we estimated that κ is a little larger than 2. Note that this is much better than a direct enumeration in which time requirements are proportional to the number of polyominoes and therefore has the growth constant, $\lambda \simeq 4.06\ldots$, of polyominoes. The price we have to pay is that the memory requirement also grows exponentially like N_{Conf}, whereas in direct enumerations the memory requirement grows like a polynomial in n.

Further Details. The integer coefficients occurring in the calculation become very large so we used modular arithmetic [21]. This involves performing the calculation modulo various integers p_i and then reconstructing the full integer coefficients at the end. The p_i are called moduli and must be chosen so they are mutually prime. The Chinese remainder theorem ensures that any integer has a unique representation in terms of residues. Since we are using a heavily loaded shared facility CPU time was more of a immediate limitation than memory and secondly more memory was used for the data required to specify the configuration and manage the storage than for storing the actual terms of the generating functions. So we used the moduli $p_0 = 2^{62}$ and $p_1 = 2^{62} - 1$, which allowed us to represent a_n correctly using just these two moduli.

2.2 Parallelization

In the past decade or so parallel computing has become the paradigm for high performance computing. The early machines were largely dedicated MPP machines which more recently have been superceded by clusters. The transfer-matrix algorithms used in the calculations of the finite lattice contributions are eminently suited for parallel computations.

The most basic concerns in any efficient parallel algorithm is to minimise the communication between processors and ensure that each processor does the same amount of work and use the same amount of memory. In practice one naturally has to strike some compromise and accept a certain degree of variation across the processors.

One of the main ways of achieving a good parallel algorithm using data decomposition is to try to find an invariant under the updating rules. That is we seek to find some property about the configurations along the intersection which does not alter in a single iteration. The algorithm for the enumeration of polyominoes is quite complicated since not all possible configurations occur due to pruning and an update at a given set of cells might change the state of a cell far removed as explained above. However, there still is an invariant since any cell not in the kink itself cannot change from being empty to being occupied and vice versa. Only the kink cell can change its occupation status. This invariant allows

us to parallelise the algorithm in such a way that we can do the calculation completely independently on each processor with just two redistributions of the data set each time an extra column is added to the lattice. It should be noted that each redistribution results in a global synchronization of the processors.

The main points of the algorithm are summarized below:

1. With the intersection in an upright position distribute the data across processors so that configurations with the same occupation pattern along the *lower* half of the intersection are placed on the same processor.
2. Do the TM update inserting the top-half of a new column. This can be done *independently* by each processor because the occupation pattern in the lower half remains unchanged.
3. Upon reaching the half-way mark redistribute the data so that configurations with the the same occupation pattern along the *upper* half of intersection are placed on the same processor.
4. Do the TM update inserting the bottom-half of a new column.
5. Go back to 1.

The redistribution among processors is done as follows:

1. On each processor run through the configurations to establish the configuration pattern c of each configuration and calculate, $n(c)$, the number of configurations with a given pattern.
2. Calculate the global sum of $n(c)$ on say processor 0.
3. Sort $n(c)$ on processor 0.
4. On processor 0 assign each pattern to a processor $p(c)$ such that:
   a) Set $p_{id} = 0$.
   b) Assign the *most* frequent unassigned pattern c to processor p_{id}.
   c) If the number of configurations assigned to p_{id} is less than the number of configurations assigned to p_0 then assign the *least* frequent unassigned patterns to p_{id} until the desired inequality is achieved.
   d) set $p_{id} = (p_{id} + 1) \mod N_p$, where N_p is the number of processors.
   e) Repeat from (b) until all patterns have been assigned.
5. Send $p(c)$ to all processors.
6. On each processor run through the configurations sending each configuration to its assigned processor.

The calculations were performed on the facilities of the Australian Partnership for Advanced Computing (APAC) which is an HP Server Cluster with 125 ES45's each with 4 1 Ghz Alpha chips for a total of 500 processors in the compute partition. The cluster has a total peak speed of 1Tflop. Each server node has at least 4 Gb of memory. Nodes are interconnected by a fat-tree low latency (MPI $<$ 5 usecs), high bandwidth (250 Mb/sec bidirectional) Quadrics network.

In Table 2 we list the time and memory use of the algorithm for $N_{\max} = 48$ at $W = 20$ using from 1 to 64 processors. The memory use of the single processor job was about 2Gb. As can be seen the algorithm scales perfectly from 1 to 64 processors since the total CPU time (column 2) stays almost constant

Table 2. Total CPU time, elapsed time, time cost of redistribution, and memory use for the parallel algorithm for enumerating polyominoes of maximal size 48 at width 20

Proc	CPU	Elapsed	Max Cost	Min Cost	Max Conf	Min Conf	Max Term	Min Term
1	15:06	15:08:04			47586716		121194174	
2	14:44	7:23:15	23:47	20:15	24773601	24248682	62848978	62119094
4	15:06	3:49:09	14:03	12:39	12438275	12247276	32034803	31180773
8	14:30	1:49:44	10:47	7:30	6291877	6070370	16270110	15592551
16	14:17	54:12	7:30	4:52	3298793	3024985	8472602	7541084
32	14:14	27:03	4:59	3:07	1753622	1570854	4513708	3834315
64	14:06	13:55	4:09	1:59	899144	851739	2464817	1923254

while the elapsed time is halved when the number of processors is doubled. We expect that the drop in CPU time at 32 or 64 processors is caused by better single processor optimization by the compiler. One would for example expect that the average time taken to fetch elements from memory drops as the problem size on individual processors drops from 2Gb for the computation using a single processor to just under 40Mb for the 64 processor computation. Another main issue in parallel computing is that of load balancing, that is, we wish to ensure that the workload is shared almost equally among the processors. This algorithm is quite well balanced. Even with 64 processors, where each processor uses only about 40Mb of memory, the difference between the processor handling the maximal and minimal number of configurations is less than 10%. For the total number of terms retained in the generating functions the spread is less than 20%. A simple timing of different sub-routines of the parallel algorithm shows that the typical time to do a redistribution is about the same as the average time taken in order to move the kink once. Further on this subject we have listed, in columns 4 and 5, the maximal and minimal 'redistribution cost' (total time spent in the redistribution sub-routine). Firstly we note that the typical overall cost of parallel execution is smaller than 10%, when the per processor problem size is large. As the number of processors is increased we are not surprised to see that the relative cost of parallel execution increases and as the problem becomes less well-balanced we also see an increase in the difference between the maximal and minimal cost of redistribution.

Another way of examining the efficiency of the parallel algorithm is to look at a fixed rectangle (in this case a 22×22 square) and grow the overall problem size by increasing $N_{\max}$. This means that more and more configurations and terms are retained. By increasing the number of processors we ensured that the problem size handled by individual processors remained relatively stable (we tried to make the number of configurations almost constant). In Table 3 we list the time and memory use of the algorithm as $N_{\max}$ is increased from 50 to 56 while using from 4 to 32 processors. Clearly we achieved the goal of keeping the number of configurations fairly constant. The elapsed time also stays fairly constant with changes largely reflecting the changes in workload as the number of configurations and terms increase or decrease. One thing we do note is that the difference between the maximum and minimum number of configurations

Table 3. Elapsed time, redistribution cost, and memory use for the parallel algorithm for enumerating polyominoes of maximal size $N_{\max}$ on a 22×22 square

$N_{\max}$	Proc	Elapsed	Max Cost	Min Cost	Max Conf	Min Conf	Max Term	Min Term
50	4	6:30:25	0:21:58	0:21:09	19148131	18767145	31440647	30611130
51	8	5:51:45	0:26:23	0:19:19	17250302	17004054	33497276	31924532
52	12	6:14:21	0:34:23	0:22:17	18141158	17262224	40396483	39119225
53	16	6:43:25	0:44:00	0:28:02	18808580	17856432	48789794	46255379
54	20	7:09:07	0:49:34	0:27:49	19899189	18821333	59065745	56473260
55	28	6:37:14	0:58:35	0:30:58	18402937	16789846	59699548	56015367
56	32	7:21:54	1:11:58	0:32:39	19525496	17883903	73484899	65934722

(and terms) increase significantly with problem size. In particular the difference in the maximal and minimal redistribution costs is markedly increased from the 4 to the 32 processor problem. Obviously this would indicate that there is some scope for further optimization of the parallel algorithm.

We have extended the series for square lattice polyominoes up to size 56. The calculations requiring most resources were at widths 22–24. These cases were done using 40 processors and took about 8-10 hours each. The total CPU time required was about 1500 hours per prime. Calculations for each width and prime are totally independent and several can be done simultaneously.

3 Results and Conclusion

We have presented a parallel algorithm for the enumeration of polyominoes on the square lattice. The computational complexity of the algorithm is exponential with time (and memory) growing as $\kappa^{n/2}$, where κ appears to be a little larger than 2. Implementation on the APAC server cluster has allowed us to count the number of polyominoes up size 56. In Table 4 we have listed the new terms obtained in this work for the number of polyominoes with perimeter 47–56. The number of polyominoes of length ≤ 46 can be found in [9]. Repeating the analysis of [13] we obtain an improved numerical estimate for the growth constant $\lambda = 4.0625696(5)$ and an improved lower bound $\lambda \geq 3.927378\ldots$.

Table 4. The number, a_n, of fixed n-cell polyominoes on the square lattice

n	a_n	n	a_n
47	272680844424943840614538634	52	2731266600165191432933200262 56
48	1085035285182087705685323738	53	10889336855593503008200959900 30
49	4319331509344565487555270660	54	43429974696239331559427538990 00
50	17201460881287871798942420736	55	17326987021737904384935434351490
51	68530413174845561618160604928	56	69150714562532896936574425480218

Acknowledgements. Financial support from the Australian Research Council is gratefully acknowledged. The calculations presented in this paper would not have been possible without a generous grant of computer time on the server cluster of the Australian Partnership for Advanced Computing (APAC).

References

1. Golomb, S.: Polyominoes: Puzzles, Patterns, Problems and Packings. Second edn. Princeton U P, Princetion, N. J. (1994)
2. Lunnon, W.F.: Counting polyominoes. In Atkin, A.O.L., Birch, B.J., eds.: Computers in Number Theory. Academic Press, London (1971) 347–372
3. Martin, J.L.: Computer enumerations. In Domb, C., Green, M.S., eds.: Phase Transitions and Critical Phenomna. Volume 3. Academic Press, London (1974)
4. Redelmeier, D.H.: Counting polyominoes: Yet another attack. Discrete Math. **36** (1981) 191–203
5. Martin, J.L.: The impact of large-scale computing on lattice statistics. J. Stat. Phys. **58** (1990) 749–774
6. Mertens, S.: Lattice animals: A fast enumeration algorithm and new perimeter polynomials. J. Stat. Phys. **58** (1990) 1095–1108
7. Mertens, S., Lautenbacher, M.E.: Counting lattice animals – a parallel attack. J. Stat. Phys. **66** (1992) 669–678
8. Conway, A.R.: Enumerating 2D percolation series by the finite lattice method. J. Phys. A: Math. Gen. **28** (1995) 335–349
9. Jensen, I.: Enumerations of lattice animals and trees. J. Stat. Phys. **102** (2001) 865–881
10. Stauffer, D., Aharony, A.: Introduction to Percolation Theory. 2 edn. Taylor & Francis, London (1992)
11. Sykes, M.F., Glen, M.: Percolation processes in two dimensions. I. Low-density series expansion. J. Phys. A: Math. Gen. **9** (1976) 87–95
12. Klarner, D.A.: Cell growth problems. Canad. J. Math. **19** (1967) 851–863
13. Jensen, I., Guttmann, A.J.: Statistics of lattice animals (polyominoes) and polygons. J. Phys. A: Math. Gen. **33** (2000) L257–L263
14. Rands, B.M.I., Welsh, D.J.A.: Animals, trees and renewal sequences. IAM J. Appl. Math. **27** (1981) 1–17
15. Klarner, D.A., Riverst, R.: A procedure for improving the upper bound for the number of n-ominoes. Canad. J. Math. **25** (1973) 565–602
16. Lubensky, T., Isaacson, J.: Statistics of lattice animals and dilute branched polymers. Phys. Rev. A **20** (1979) 2130–2146
17. Conway, A.R., Guttmann, A.J.: On two-dimensional percolation. J. Phys. A: Math. Gen. **28** (1995) 891–904
18. Oliveira e Silva, T.: See the web at http://www.ieeta.pt/~tos/animals.html
19. Knuth, D.E.: Polynum. Program available from Knuth's home-page at http://Sunburn.Stanford.EDU/~knuth/programs.html#polyominoes (2001)
20. Enting, I.G.: Generating functions for enumerating self-avoiding rings on the square lattice. J. Phys. A: Math. Gen. **13** (1980) 3713–3722
21. Knuth, D.E.: Seminumerical Algorithms. The Art of Computer Programming, Vol 2. Addison Wesley, Reading, Mass (1969)