$\lambda > 4$ *

Gill Barequet[1], Günter Rote[2], and Mira Shalah[1]

[1] Dept. of Computer Science
Technion—Israel Institute of Technology
Haifa 32000, Israel
{barequet,mshalah}@cs.technion.ac.il

[2] Institut für Informatik
Freie Universität Berlin
Takustraße 9, D-14195 Berlin, Germany
rote@inf.fu-berlin.de

Abstract. A *polyomino* ("lattice animal") is an edge-connected set of squares on the two-dimensional square lattice. Counting polyominoes is an extremely hard problem in enumerative combinatorics, with important applications in statistical physics for modeling processes of percolation and collapse of branched polymers. We investigated a fundamental question related to polyominoes, namely, what is their growth constant, the asymptotic ratio between $A(n+1)$ and $A(n)$ when $n \to \infty$, where $A(n)$ is the number of polyominoes of size n. This value is also known as "Klarner's constant" and denoted by λ. So far, the best lower and upper bounds on λ were roughly 3.98 and 4.65, respectively, and so not even a single decimal digit of λ was known. Using extremely high computing resources, we have shown (still rigorously) that $\lambda > 4.00253$, thereby settled a long-standing problem: proving that the leading digit of λ is 4.

Keywords: Polyominoes, lattice animals, growth constant.

1 Introduction

1.1 What is λ?

The universal constant λ appears in the study of three seemingly completely unrelated fields: combinatorics, percolation, and branched polymers. In combinatorics, the analysis of *self-avoiding walks* (SAWs, non-self-intersecting lattice paths starting at the origin, counted by lattice units), *simple polygons* or *self-avoiding polygons* (SAPs, closed SAWs, counted by either perimeter or area), and *polyominoes* (SAPs possibly with holes, edge-connected sets of lattice squares, counted by area), are all related. In statistical physics, SAWs and SAPs play a significant role in percolation processes and in the collapse transition which branched polymers undergo when being heated. A recent collection edited by

* We acknowledge the support of the facilities and staff of the HPI Future SOC Lab in Potsdam, where this project has been carried out.

A. J. Guttmann [11] provides an excellent review of all these topics and the connections between them. In this paper we describe our effort to prove that the growth constant of polyominoes is strictly greater than 4. To this aim we exploited to the maximum possible computer resources which were available to us, designing and implementing carefully the algorithm and the required data structures. Eventually we obtained a computer-generated proof which was verified by other programs implemented independently. Let us start with a brief description of the history of λ and the three research areas, then describe the method and computation of the lower bound on λ.

1.2 Brief History

Determining the exact value of λ (or even setting good bounds on it) is a hard problem in enumerative combinatorics. In 1967, Klarner [13] showed that the limit $\lim_{n\to\infty} \sqrt[n]{A(n)}$ exists and denoted it by λ. Since then, λ has been called "Klarner's constant." Only in 1999, Madras [16] proved the stronger statement that the asymptotic growth rate in the sense of the limit $\lambda = \lim_{n\to\infty} A(n+1)/A(n)$ exists.

By using interpolation methods, Sykes and Glen [21] *estimated* in 1976 that $\lambda = 4.06 \pm 0.02$. This estimate was sharpened several times, the most accurate (4.0625696 ± 0.0000005) given by Jensen [12] in 2003. Before carrying out this project, the best *proven* bounds on λ were roughly 3.9801 from below [4] and 4.6496 from above [14]. Thus, λ has always been an elusive constant, of which not even a single significant digit was (rigorously) known. Our goal was to raise the lower bound on λ over the barrier of 4, and thus reveal its first decimal digit and proving that $\lambda \neq 4$. The current improvement of the lower bound on λ to 4.0025 also cuts the difference between the known lower bound and the estimated value of λ by about 25% (from 0.0825 to 0.0600).

1.3 Enumerative Combinatorics

A *polyomino* is a connected set of cells on the planar square lattice, where connectivity is along sides but not through corners of the cells. Polyominoes were made popular by the pioneering book of Golomb [10] and by Martin Gardner's columns in Scientific American, and counting polyominoes by size became a popular fascinating combinatorial problem. The size of a polyomino is the number of its cells. In this article we consider "fixed" polyominoes; two such polyominoes are considered identical if one can be obtained from the other by a translation, while rotations and flipping are not allowed. In the mathematical literature, the number of polyominoes of size n is usually denoted as $A(n)$, but no formula is known yet for it. Researchers have suggested efficient back-tracking [19,20] and transfer-matrix [6,12] algorithms for computing $A(n)$ for a given value of n. The latter algorithm was adapted in [4] and also in this work for twisted cylinders. To-date, the sequence $A(n)$ has been determined up to $n = 56$ by a parallel computation on an HP server cluster using 64 processors [12]. The exact value of the growth constant of this sequence, $\lambda = \lim_{n\to\infty} A(n+1)/A(n)$, has also

been elusive for many years. It has been interesting to know whether this value is smaller or greater than the *connective constant* of this lattice. This latter constant is simply the number of neighbors each cell of the lattice has, which is, in this case, 4. In this work we reveal the leading decimal digit of λ: It is 4.

1.4 Percolation Processes

In physics, chemistry, and materials science, percolation theory deals with the movement and filtering of fluids through porous materials. Giving it a mathematical model, the theory describes the behavior of connected clusters in random graphs. Suppose that a unit of liquid L is poured on top of some porous material M. What is the chance that L makes its way through M and reach the bottom? An idealized mathematical model of this process is a two- or three-dimensional grid of vertices ("sites") connected with edges ("bonds"), where each bond is independently open (or closed) for liquid flow with some probability p. Broadbent and Hammersley [5] asked in 1957, for a fixed value of p and for the size of the grid tending to infinity, what is the probability that a path consisting of open bonds exists from the top to the bottom. They essentially investigated solute diffusing through solvent and molecules penetrating a porous solid, representing space as a lattice with two distinct types of cells.

In the literature of statistical physics, fixed polyominoes are usually called "strongly-embedded lattice animals," and there, the analogue of the growth rate of polyominoes is the growth constant of lattice animals. The terms high and low *temperature* mean high and low *density* of clusters, respectively, and the term *free energy* corresponds to the natural logarithm of the growth constant. Lattice animals were used for computing the mean cluster density in percolation processes (Gaunt et al. [9]), in particular those of fluid flow in random media. Sykes and Glen [21] were the first to observe that $A(n)$, the total number of connected clusters of size n, grows asymptotically like $C\lambda^n n^\theta$, where λ is Klarner's constant and C, θ are two other fixed values.

1.5 Collapse of Branched Polymers

Another important topic in statistical physics is the existence of a collapse transition of branched polymers in dilute solution at a high temperature. In physics, a *field* is an entity each of whose points has a value which depends on location and time. Lubensky and Isaacson [15] developed a field theory of branched polymers in the dilute limit, using statistics of (bond) lattice animals (which are important in the theory of percolation) to imply when a solvent is good or bad for a polymer. Derrida and Herrmann [7] investigated two-dimensional branched polymers by looking at lattice animals on a square lattice and studying their free energy. Flesia et al. [8] made the connection between collapse processes to percolation theory, relating the growth constant of strongly-embedded site animals to the free energy in the processes. Madras et al. [17] considered several models of branched polymers in dilute solution, proving bounds on the growth constants for each such model.

2 Twisted Cylinders

A "twisted cylinder" is a half-infinite wrap-around spiral-like square lattice, as is shown in Fig. 1. We denote the perimeter (or "width") of the twisted cylinder by

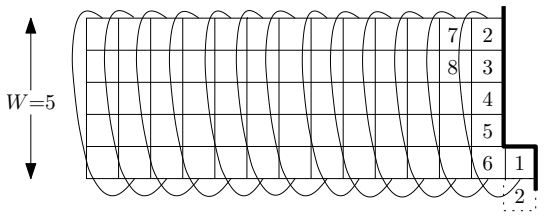


Fig. 1. A twisted cylinder of perimeter $W = 5$.

the symbol W. Like in the plane, one can count polyominoes on a twisted cylinder of width W and study their asymptotic growth constant, λ_W. It was proven that the sequence $(\lambda_W)_{W=1}^{\infty}$ is monotone increasing [4] and that it converges to λ [2]. Thus, the bigger W is, the better (higher) the lower bound λ_W on λ is.

It turns out that analyzing the growth rate of polyominoes is more convenient on a twisted cylinder than in the plane. The reason is that we want to build up polyominoes incrementally by considering one square at a time. On a twisted cylinder, this can be done in a uniform way, without having to jump to a new row from time to time. Imagine that we walk along the spiral order of squares, and at each square decide whether or not to add it to the polyomino. Naturally, the size of a polyomino is the number of positive decisions we make on the way. The crucial observation is that no matter how big polyominoes are, they can be characterized in a *finite* number of ways that depends only on W. This is because all one needs to remember is the structure of the last W squares of the twisted cylinder (the "boundary"), and how they are inter-connected through cells that were considered before the boundary. This provides enough information for the continuation of the process: whenever a new square is considered, and a decision is taken about whether or not to add it to the polyomino, the boundary is updated accordingly. Thus, the growth of polyominoes on a twisted cylinder can be modeled by a finite-state automaton whose states are all possible boundaries. Every state in this automaton has two outgoing edges that correspond to whether or not the next square is added to the polyomino.

The number of states in the automaton that models the growth of polyominoes on a twisted cylinder of perimeter W is large [3,4]: it is the $(W+1)$st Motzkin number M_{W+1}. The nth Motzkin number, M_n, counts the number of Motzkin paths of length n (see Fig. 2 for an illustration): Such a path connects the integer grid points $(0, 0)$ and $(n, 0)$ with n steps, consisting only of steps taken from $\{(1, 1), (1, 0), (1, -1)\}$, and not going under the x axis. Motzkin numbers can also be defined in a variety of other ways [1]. Asymptotically, $M_n \sim 3^n n^{-3/2}$, thus, M_W increases roughly by a factor of 3 when W is incremented by 1.

The number of polyominoes with n cells that have state s as the boundary configuration is equal to the number of paths that the automaton can take from the starting state to the state s, paths which involve n transitions in which a cell is added to the polyomino. We compute these numbers by using a dynamic-programming recursion.

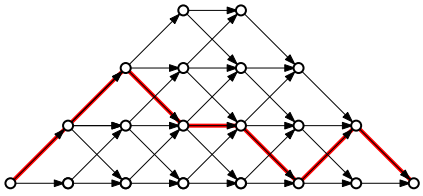


Fig. 2. A Motzkin path of length 7.

See Appendix A for more details about Motzkin paths, and Appendix B for a small example automaton and more explanations of this topic.

3 Method

In 2004, a sequential program that computes λ_W for any perimeter was developed by Ms. Ares Ribó as part of her Ph.D. thesis under the supervision of G. Rote. The program first computes the endpoints of the outgoing edges from all states of the automaton and saves them in two long arrays *succ0* and *succ1*, which correspond to adding an empty or an occupied cell. Both arrays are of length $M := M_{W+1}$. Two successive iteration vectors (which contain the number of polyominoes corresponding to each boundary) are stored as two arrays y^{old} and y^{new} of floating point numbers, also of length M. The four arrays are indexed from 0 to $M-1$. After initializing $y^{\text{old}} := (1, 0, 0, \dots)$, each iteration computes the new version of y by performing the following simple loop.

$y^{\text{new}}[0] := 0;$
for $s := 1, \dots, M-1$:
$(*)$ $\quad y^{\text{new}}[s] := y^{\text{new}}[\mathit{succ0}[s]] + y^{\text{old}}[\mathit{succ1}[s]];$

As mentioned, the pointer arrays *succ0*[ ] and *succ1*[ ] are computed beforehand. The pointer *succ0*[s] may be null, in which case the corresponding zero entry ($y^{\text{new}}[0]$) is used.

As explained above, each index s represents a state. The states are encoded by Motzkin paths, and these paths can be bijectively mapped to numbers s between 0 and $M-1$. In the iteration $(*)$, the vector y^{new} depends on itself, but this does not cause any problem because *succ0*[s], if it is non-null, is always less than s. Therefore, there are no circular references and each entry is set before it is used. In fact, the states can be partitioned into groups $G_1, G_2, \dots, G_W$: The group G_i contains the states corresponding to boundaries in which i is the smallest occupied cell, or, in other words, boundaries that start with $i-1$ empty cells. The dependence between the entries of the groups is schematically shown in Fig. 3: *succ0*[s] of an entry $s \in G_i$ (for $1 \le i \le W-1$), if it is non-null, belongs to G_{i+1}. Naturally, *succ0* of the single state in G_W is null since a boundary with all cells empty is invalid.

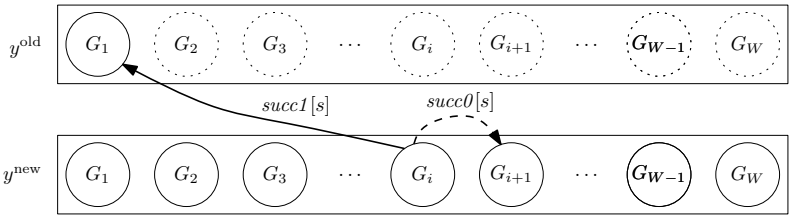


Fig. 3. The dependence between the different groups of y^{new} and y^{old}.

At the end, y^{new} is moved to y^{old} to start the new iteration. It was proven [4] that after every iteration, we have the following interval for bounding λ_W:

$$\min_s \frac{y^{\text{new}}[s]}{y^{\text{old}}[s]} \le \lambda_W \le \max_s \frac{y^{\text{new}}[s]}{y^{\text{old}}[s]} \tag{1}$$

In this procedure, the two bounds converge (by the Perron-Frobenius Theorem) to λ_W, and y^{old} converges to a corresponding eigenvector. The vector y^{old} is normalized after every few iterations in order to prevent overflow. The scale of the vector is irrelevant to the process. The program terminates when the two bounds are close enough. The left-hand side of (1) is a lower bound on λ_W, which in turn is a lower bound on λ, and this is our real goal.

4 Sequential Runs

In 2004 we obtained good approximations of λ_W up to $W = 22$. The program required extremely high resources in terms of main memory (RAM) by the standards of that time. The computation of $\lambda_{22} \approx 3.9801$ (with a single processor) took about 6 hours on a machine with 32 GB of RAM. (Today, the same program runs in 20 minutes on a regular workstation.) We extrapolated the first 22 values of (λ_W) (see Fig. 4) and estimated that only when we reach $W = 27$ we would break the mythical barrier of 4.0. However, as mentioned above, the required storage is proportional to M_W, which increases roughly by a factor of 3 when W is incremented by 1. With this exponential growth of both memory and running time, the goal of breaking the barrier seemed then to be out of reach.

5 Computing λ_{27}

Environment. The computation of λ_{27} was performed on a Hewlett Packard ProLiant DL980 G7 server of HPI (Hasso Plattner Institute) Future SOC Lab in Potsdam, Germany. It consists of 8 Intel Xeon X7560 nodes (Intel64 architecture), each having eight physical 2.26 GHz processors (16 virtual cores), for a total of 64 processors (128 virtual cores). Hyperthreading was used to allow processes to run on the physical cores of a node while sharing certain resources, thus yielding twice as many virtual processor cores as physical processors. Each

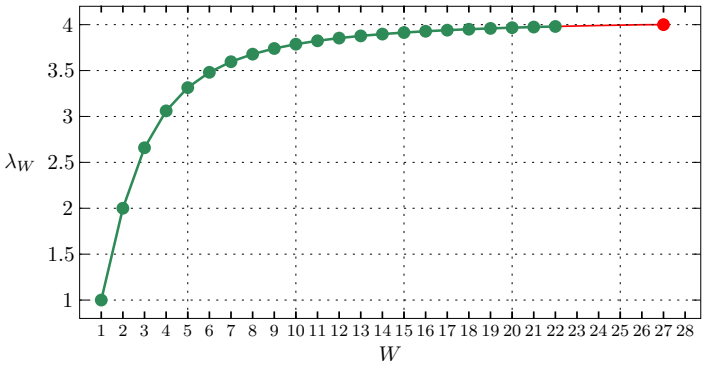


Fig. 4. Extrapolating the sequence λ_W.

node was equipped with 256 GiB of RAM (and 24 MiB of cache memory), for a total of **2 TiB of RAM**. Simultaneous access by all processors to the shared main memory was crucial to the success of the project. Distributed memory would incur a severe penalty in running time. The machine was run with the Ubuntu version of Gnu/Linux. Compilation was done using the `gcc` C compiler with OpenMP 2.0 directives for parallel processing.

Programming Improvements. Since for $W = 27$ the finite automaton has $M_{28} \approx 2.1 \cdot 10^{11}$ states, we initially estimated that we would need memory for two 8-byte arrays (for storing *succ0* and *succ1*) and two 4-byte arrays (for storing y^{old} and y^{new}), all of length M_{28}, for a total of $24 \cdot 2.1 \cdot 10^{11} \approx 4.6$ TiB of RAM, which was certainly out of reach, even with the available supercomputer. Apparently, a combination of parallelization, storage-compression techniques, and a few other enhancements and tricks allowed us to push the lower bound on λ above 4.0.

1. **Parallelization.** Since the set of states G of the automaton could be partitioned into groups $G_1, \ldots, G_W$, such that *succ0*$[s]$ for an element $s \in G_i$ belongs to G_{i+1}, the groups $G_W, \ldots, G_1$ had to be processed sequentially (in this order) but all elements in one group could be computed in parallel. The size of the groups is exponentially decreasing; in fact, G_1 comprises more than half of all states, and G_W contains only a single state. Therefore, for the bulk of the work (the iterative computation of y^{new}), we easily achieved coarse-grained parallelization and a distribution of the work on all the 128 available cores, requiring concurrent read but no concurrent write operations. We also parallelized the preprocessing phase (computing the *succ* arrays) and various house-keeping tasks (e.g., rescaling the y vectors after every 10th iteration). Tests with different numbers of processors revealed indeed a speed-up close to linear.
2. **Elimination of unreachable states.** A considerable portion of the states of the automaton (about 11% asymptotically) are unreachable, i.e., there is no binary string leading to these states. This happens because not all

seemingly legal states can be realized by a valid boundary. These states do not affect the correctness of the computation, and there was no harm in leaving them, apart from the effect on the performance of the iteration. We were able to characterize the unreachable states fairly easily in terms of their Motzkin paths. After eliminating these states, we had to modify the bijection between the Motzkin paths representing the remaining states and the successive integers.

3. **Bit-streaming of the *succ0/1* arrays.** Instead of storing each entry of the *succ0/1* arrays in a full word (8 bytes, once the number of states exceeded 2^{32}), we allocated to each entry exactly the number of required bits and stored all entries consecutively in a packed manner. Since the *succ0/1* entries were only accessed sequentially, there was only a small overhead in running time for unpacking the resulting bit sequence. In addition, since we knew *a priori* to which group G_i each pointer belonged, we needed only $\lceil \log_2 |G_i| \rceil$ bits per pointer, for all entries in G_i (plus a negligible amount of bits required to delimit between the different sets G_i). On top of that, the *succ0*-pointer was often null because the choice of not adding the next cell to the polyomino caused a connected component of the polyomino to lose contact with the boundary. By spending one extra indicator bit per pointer, we eliminated altogether these illegal pointers, which comprised about 11% of all *succ0* entries.

4. **Storing higher groups only once.** For states s not in the group G_1, $y^{\text{old}}[s]$ is not needed in the recursion $(*)$. Thus, we did not need to keep two separate arrays in memory. The quotient $y^{\text{new}}[s]/y^{\text{old}}[s]$ could still be computed before overwriting $y^{\text{old}}[s]$ by $y^{\text{new}}[s]$, and thus the minimum and maximum of these quotients, which give the bounds (1) on λ, could be accumulated as we scanned the states.

5. **Recomputing *succ0*.** Instead of storing the *succ0* array, we computed its entries on-the-fly whenever they were needed, and thus saved completely the memory needed to store these pointers. Naturally, this required more running time. Streamlined computation of the pointers accelerated the successor computation (see below). This variation has also benefited from parallelization since each processor could do the pointer computations independently. Since the elimination of the *succ0* pointers was sufficient to get the program running with $W = 27$, we did not pursue the option of eliminating the *succ1* array in an analogous way.

6. **Streamlining the conversion from Motzkin paths to integers.** Originally, we represented a Motzkin path by a sequence of $W+1$ integer numbers taking values from $\{-1, 0, +1\}$. However, we compressed the representation into a sequence of $(W+1)$ 2-bit items, each one encoding one step of the path, which we could store in one 8-byte word (since $W \leq 31$). This compact storage opened up the possibility of word-level operations. For converting paths to numbers, we could process several symbols at a time, using look-up tables.

Execution. After 120 iterations, the program announced the lower bound 4.00064 on λ_{27}, thus breaking the 4 barrier. We continued to run the program for a few more days. Then, after 290 iterations, the program reached the stable situation (observed in a few successive tens of iterations) $4.002537727 \leq \lambda_{27} \leq 4.002542973$, establishing the new record $\lambda > 4.00253$. The total running time for the computations leading to this result was about 36 hours. In total, we used a few dozens of hours of exclusive use of the server spread over several weeks.

6 Validity and Certification

Our proof depends heavily on computer calculations. This raises two issues about its validity: (a) Elaborate calculations on a large computer are hard to reproduce, and in particular when a complicated parallel computer program is involved, one should be skeptical. (b) We performed the computations with 32-digit floating-point numbers. We address these issues in turn.

(a) What our program tries to compute is an eigenvalue of a matrix. The amount and length of the computations are irrelevant to the fact that eventually we have a witness array of floating-point numbers (the "proof"), about 450 GB in size, which is a good approximation of the eigenvector corresponding to λ_{27}. This array provides rigorous bounds on the true eigenvalue λ_{27}, because the relation (1) holds for *any* vector y^{old} and its successor vector y^{new}. To check the proof and evaluate the bounds (1), one only has to read the approximate eigenvector y^{old} and carry out one iteration $(*)$. This approach of providing simple certificates for the result of complicated computations is the philosophy of *certifying algorithms* [18]. We ran two different programs for the checking task. The code for the only technically challenging part of the algorithm, the successor computation, was based on programs written independently by two people who used different state representations. Both programs ran in a purely sequential manner, and the running time was about 20 hours each.

(b) Regarding the accuracy of the calculations, one can look how the recurrence $(*)$ produces y^{new} from y^{old}. One finds that each term in the lower bound (1) results from the input data (the approximate eigenvector y^{old}) through at most 26 additions of positive numbers for computing $y^{\text{new}}[s]$, plus one division, all in single-precision `float`. The final minimization is error-free. Since we made sure that no denormalized floating-point numbers occurred, the magnitude of the numerical errors is comparable to the accuracy of floating-point numbers, and the accumulated error is much smaller than the gap that we opened above 4. By bounding the floating-point error, we obtain 4.00253176 as a certified lower bound on λ. Thus, in particular, we now know that the leading digit of λ is 4.

7 Conclusion

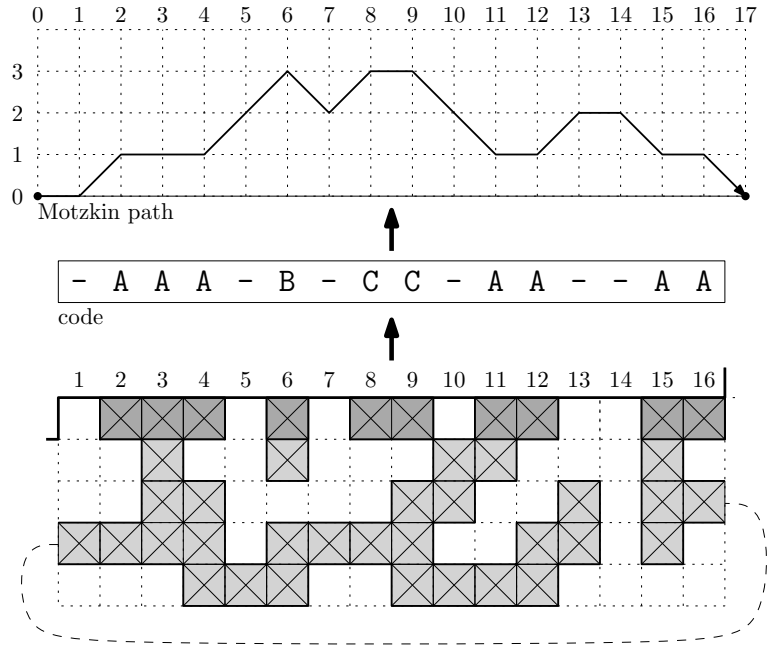
In this project we computed λ_{27} and set a new lower bound on λ, which is greater than 4. By this we also excluded the possibility that $\lambda = 4$. We believe that with some more effort, it will be feasible to run the program for $W = 28$.

This would probably require (a) To eliminate also the storage for the *succ1*-successors and compute them along with the *succ0*-successors; (b) To eliminate all groups $G_2, \ldots, G_W$ and keep only group G_1; and (c) To implement a customized floating-point storage format for the numbers y. (With a total of 2 TiB of RAM, we can only afford 27 bits per entry.) We anticipate that this would increase the lower bound on λ to about 4.0065.

References

1. M. Aigner, Motzkin Numbers, *European J. of Combinatorics*, **19**, 663–675, 1998.
2. G. Aleksandrowicz, A. Asinowski, G. Barequet, R. Barequet, Formulae for polyominoes on twisted cylinders, *Proc. 8th Int. Conf. on Lang. and Automata Theory and Appl.*, Madrid, Spain, *LLNCS*, 8370, Springer-Verlag, 76–87, March 2014.
3. G. Barequet, M. Moffie, On the complexity of Jensen's algorithm for counting fixed polyominoes, *J. of Discrete Algorithms*, **5**, 348–355 (2007).
4. G. Barequet, M. Moffie, A. Ribó, G. Rote, Counting polyominoes on twisted cylinders, *INTEGERS: Elec. J. of Comb. Number Theory*, **6**, #A22, 37 pp., 2006.
5. S.R. Broadbent, J.M. Hammersley, Percolation processes: I. Crystals and mazes, *Proc. Cambridge Philosophical Society*, **53**, 629–641, 1957.
6. A. Conway, Enumerating 2D percolation series by the finite-lattice method: Theory, *J. of Physics, A: Mathematical and General*, **28**, 335–349, 1995.
7. B. Derrida, H.J. Herrmann, Collapse of branched polymers, *J. de Physique*, **44**, 1365–1376, 1983.
8. S. Flesia, D.S. Gaunt, C.E. Soteros, S.G. Whittington, Statistics of collapsing lattice animals, *J. of Physics, A: Mathematical and General*, **27**, 5831–5846, 1994.
9. D.S. Gaunt, M.F. Sykes, H. Ruskin, Percolation processes in d-dimensions, *J. of Physics A: Mathematical and General*, **9**, 1899–1911, 1976.
10. S.W. Golomb, *Polyominoes*, Princeton Univ. Press, Princeton, NJ, 2nd ed., 1994.
11. A.J. Guttmann, Ed., *Polygons, Polyominoes and Polycubes*, *Lecture Notes in Physics*, 775, Springer, Heidelberg, 2009.
12. I. Jensen, Counting polyominoes: A parallel implementation for cluster computing, *Proc. Int. Conf. on Computational Science*, part III, Melbourne, Australia and St. Petersburg, Russia, *LNCS*, 2659, Springer, 203–212, June 2003.
13. D.A. Klarner, Cell growth problems, *Canad. J. of Mathematics*, **19**, 851–863, 1967.
14. D.A. Klarner, R.L. Rivest, A procedure for improving the upper bound for the number of n-ominoes, *Canadian J. of Mathematics*, **25**, 585–602, 1973.
15. T.C. Lubensky, J. Isaacson, Statistics of lattice animals and dilute branched polymers, *Physical Review A*, **20**, 2130–2146, 1979.
16. N. Madras, A pattern theorem for lattice clusters, *Annals of Combinatorics*, **3**, 357–384, 1999.
17. N. Madras, C.E. Soteros, S.G. Whittington, J.L. Martin, M.F. Sykes, S. Flesia, D.S. Gaunt, The free energy of a collapsing branched polymer, *J. of Physics, A: Mathematical and General*, **23**, 5327–5350, 1990.
18. R.M. McConnell, K. Mehlhorn, S. Näher, P. Schweitzer, Certifying algorithms, *Computer Science Review*, **5**, 119–161, 2011.
19. S. Mertens, M.E. Lautenbacher, Counting lattice animals: A parallel attack, *J. of Statistical Physics*, **66**, 669–678, 1992.
20. D.H. Redelmeier, Counting polyominoes: Yet another attack, *Discrete Mathematics*, **36**, 191–203, 1981.
21. M.F. Sykes, M. Glen, Percolation processes in two dimensions: I. Low-density series expansions, *J. of Physics, A: Mathematical and General*, **9**, 87–95, 1976.

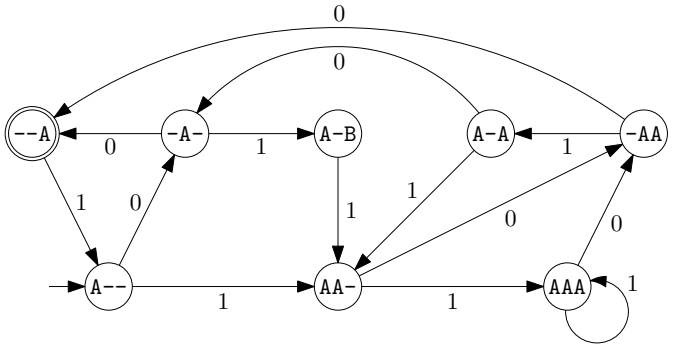
Appendix A: Representing Boundaries as Motzkin Paths



The figure above illustrates the representation of boundaries of polyominoes on twisted cylinders as Motzkin paths. The figure should be read from bottom to top. The bottom of the figure shows a partially constructed polyomino on a twisted cylinder of width 16. The dashed line indicates two adjacent cells which are connected "around the cylinder," where this is not immediately apparent. The boundary cells (top row) are shown darker. The light-gray cells away from the boundary need not be remembered individually; what matters is the connectivity among the boundary cells that they provide. This is indicated in a symbolic *code* `-AAA-B-CC-AA--AA`. Boundary cells in the same component are represented by the same letter, and the character '`-`' denotes an empty cell. However, this code was not used in our program. Instead, we represented a boundary as a Motzkin path, as shown in the top part of the figure, because this representation allows for a convenient bijection to successive integers and therefore for a compact storage of the boundary in a vector. Intuitively, the Motzkin path follows the movements of a stack when reading the code from left to right. Whenever a new component starts, like component `A` in position 2 or component `B` in position 6, the path moves *up*. Whenever a component is temporarily interrupted, such as component `A` in position 5, the path also moves *up*. The path moves *down* when an interrupted component is resumed (e.g., component `A` in positions 11 and 15) or when a component is completed (positions 7, 10, and 17). The crucial property is that components cannot cross, i.e., a pattern like $\ldots$`A`$\ldots$`B`$\ldots$`A`$\ldots$`B`$\ldots$ cannot occur. As a consequence of these rules, the occupied cells correspond to odd levels in the path, and the free cells correspond to even levels.

Appendix B: Automata for Modeling Polyominoes on Twisted Cylinders

A finite automaton is a very convenient tool for representing the growth of polyominoes. Below is the automaton for width $W = 3$. The starting state is `A--`. The states `A-B` and `--A` have no 0-successors.



We associate the labels '1' and '0' with the edges, corresponding to whether or not a cell was added to the polyomino in the corresponding step. The construction of a polyomino is modeled by tracing the path in the automaton while processing an input word of 1s and 0s, where the number of occurrences of '1' is the size of the polyomino (minus 1, since the starting state contains already one cell). We want to accept only legal polyominoes, that is, polyominoes that are composed of *connected* squares. It is sufficient to consider a single *accepting state* for the automaton (`--A` in the example) because this state can always be reached by adding enough empty cells. The number of polyominoes on a twisted cylinder is equal to the number of binary words recognized by the automaton, that is, whose processing by the automaton terminates in an accepting state.

Automata theory and linear algebra give us strong tools to analyze the behavior of a finite automaton. First, we can represent the automaton as an $M \times M$ 0/1 transfer matrix B, where M is the number of states of the automaton, and the (ij)th entry of B is 1 if an edge leads from the ith to the jth state of the automaton. One can derive from B (through its characteristic polynomial) the generating function of the sequence enumerating polyominoes on the twisted cylinder, and a linear recurrence formula satisfied by this sequence. This has been carried out [2] up to width $W = 10$.

Second, this matrix has a few interesting properties. It is proven [4] that the largest eigenvalue (in absolute value) of B is exactly the desired growth constant λ_W. Moreover, B is a primitive and irreducible matrix, and, hence, λ_W is the only positive eigenvalue of B. Under these conditions, the Perron-Frobenius theorem provides an effective method for computing this eigenvalue: Start from any positive vector (e.g., the vector in which all entries are 1) and repeatedly multiply it by B. In the limit, this process converges to the eigenvector, and the ratios between successive vectors in this process converge to the desired eigenvalue, which is the desired growth constant.