Counting Polyominoes on Twisted Cylinders

Gill Barequet Micha Moffie Ares Ribó* Günter Rote

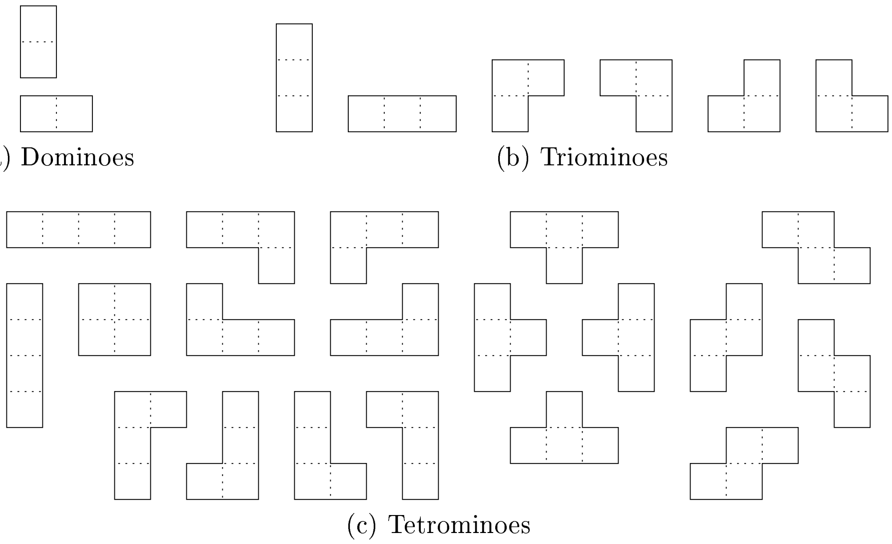
December 22, 2004

Abstract

Using numerical methods, we analyze the growth in the number of polyominoes on a twisted cylinder as the number of cells increases. These polyominoes are related to classical polyominoes (connected subsets of a square grid) that lie in the plane. We thus obtain improved lower bounds on the growth rate of the number of these polyominoes, which is also known as Klarner's constant.

1 Introduction

Polyominoes. A polyomino of size n, also called an n-omino, is a connected set of n adjacent squares on a regular square lattice (connectivity is through edges only). *Fixed* polyominoes are considered distinct if they have different shapes *or* orientations. The symbol $A(n)$ denotes the number of fixed polyominoes of size n on the plane. Figure 1(a) shows the only two fixed *dominoes* (adjacent pairs of squares). Similarly, Figures 1(b) and 1(c) show the six fixed *triominoes* and the 19 fixed *tetrominoes*—polyominoes of size 3 and 4, respectively. Thus, $A(2) = 2$, $A(3) = 6$, $A(4) = 19$, and so on.



(a) Dominoes (b) Triominoes (c) Tetrominoes

Figure 1: Fixed dominoes, triominoes, and tetrominoes

No analytic formula for $A(n)$ is known. The only methods for computing $A(n)$ are based on explicitly or implicitly enumerating all polyominoes.

*Supported by the Deutsche Forschungs-gemeinschaft within the European Graduate Program *Combinatorics, Geometry and Computation* (No. **GRK** 588/2).

Counting polyominoes has received a lot of attention in the literature. In Barequet et al. [1], we can find an overview of the development of counting fixed polyominoes, beginning in 1962 with R. C. Read.

We base our present study on Jensen's transfer-matrix algorithm [6], which is an improved version of the algorithm developed by Conway and Guttmann [3]. Using his algorithm, which was also optimized by Knuth [10], Jensen obtained the values up to $A(48)$. More recently, he used a parallel version of the algorithm and computed $A(n)$ for $n \leq 56$ [7].

It is known that $A(n)$ is exponential in n. Klarner [8] showed that $A(n) \sim C\lambda^n n^\theta$ for some constants $C > 0$ and $\theta \approx -1$, so that the limit $\lambda = \lim_{n\to\infty} A(n+1)/A(n)$ exists. Golomb [4] labelled λ as *Klarner's constant*. There have been several attempts to lower and upper bound λ, as well as to estimate it, based on knowing $A(n)$ up to certain values of n. The best-known published lower and upper bounds are 3.927378 [7] and 4.649551. However, this lower bound is based on an incorrect assumption, which goes back to the paper of Rands and Welsh [13]. As we point out in Section 7, the lower bound should have been corrected to 3.87565. Regardless of this matter, not even a single significant digit of λ is known for sure. The constant λ is estimated to be around 4.06 [3, 7]; see [9] for more background information on polyominoes.

In this paper, we improve the lower bound on Klarner's constant to 3.980137 by counting polyominoes on a different grid structure, a twisted cylinder.

The Twisted Cylinder. A *twisted cylinder* of width W is obtained from the integer grid $\mathbb{N} \times \mathbb{N}$ by identifying point (i, j) with $(i + 1, j + W)$, for all i, j. Geometrically, it can be imagined to be an infinite tube; see Figure 2.

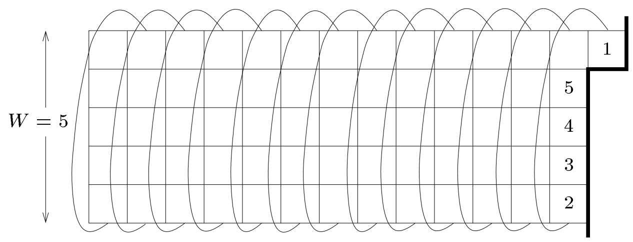


Figure 2: A twisted cylinder of width 5. The wrap-around connections are indicated; for example, cells 1 and 2 are adjacent.

The usual cylinder would be obtained by identifying (i, j) with $(i, j+W)$, without also moving one step in the horizontal direction. The reason for introducing the twist is that it allows us to build up the cylinder incrementally, one cell at a time, through a *uniform* process. This leads to a simpler recursion and algorithm. To build up the usual "untwisted" cylinder cell by cell, one has to go through a number of different cases until a complete column is built up.

We implemented an algorithm in C that iterates the transfer equations, thereby obtaining a lower bound on the growth rate of the number of polyominoes on the twisted cylinder. This is also an improved lower bound on the number of polyominoes in the plane.

The algorithm has to maintain a large vector of numbers whose entries are indexed by certain combinatorial objects that are called *states*. The states have a very clean combinatorial structure, and they are in bijection with so-called Motzkin paths. We use this bijection as a space-efficient scheme for addressing the entries of the state vector. Previous algorithms for counting polyominoes treated only a small fraction of all possible states. This so-called *pruning* of states was crucial for reaching larger values of n, but required the algorithms to encode and store states explicitly, using a hash-table, and could not have used our scheme.

Contents of the Paper. The paper is organized as follows. In Section 2 we present the idea of the transfer-matrix algorithm and define the notion of states, and how they are represented. In Section 3 we describe the recursive operations for enumerating polyominoes on our twisted cylinder grid and present the transfer equations, as well as the iteration process. We also provide an algebraic analysis of the growing rate of the number of polyominoes on the twisted cylinder. In Section 4 we prove a bijection between the states and Motzkin paths. In Section 5 we describe explicitly how Motzkin paths are generated, ranked and unranked, and updated. In Section 6 we report the results and the obtained lower bounds. In Section 7, we correct the previous lower bound given in [13]. Finally, in the concluding Section 8, we mention a few open questions. An appendix describes how the results of the computer calculations were checked by independent computer calculations.

Acknowledgements. We thank Stefan Felsner for discussions about the bijection between states and Motzkin paths. We thank Cordula Nimz for pointing out that the decomposition of polyominoes into prime factors by the $*$ operation is not unique, and hence the recursion (12) would lead to inconsistent results.

2 The Transfer-Matrix Algorithm

In this section we briefly describe the idea behind the transfer-matrix method for counting fixed polyominoes. In computer science terms, this algorithm would be classified as a dynamic programming algorithm.

The strategy is as follows. The polyominoes are built from left to right, adding one cell of the twisted cylinder at a time. Conceptually, the twisted cylinder is cut by a *boundary line* through the W rows. The *boundary cells* are the W cells adjacent to the left of the boundary line. In fact, the boundary cells are the W last added cells at a given moment of this building process (see Figure 3).

Instead of keeping track of all polyominoes, the procedure keeps track of the numbers of polyominoes with identical right boundaries. During the process, the configurations of the right boundaries of the (yet incomplete) polyominoes are called *states*, as will be described. A polyomino is expanded in the current column, cell by cell, from top to bottom. The new cell is either occupied (i.e., belongs to the new polyomino) or empty (i.e., does not belong to it). By "expanding" we mean updating both the states and their respective numbers of polyominoes.

A *partial polyomino* is the part of the polyomino lying on the left of the boundary line at some moment. A partial polyomino is not necessarily connected, but each component must contain a boundary cell.

2.1 Motzkin Paths

A *Motzkin path* [12] of length n is a path from $(0,0)$ to $(n,0)$ in a $n \times n$ grid, consisting of up-steps $(1,1)$, down-steps $(1,-1)$, and horizontal steps $(1,0)$, that never goes below the x-axis.

The number M_n of Motzkin paths of length n is known as the nth Motzkin number.

Motzkin numbers satisfy the recurrence

$$M_n = M_{n-1} + \sum_{i=0}^{n-2} M_i \cdot M_{n-i-2}, \tag{1}$$

for $n \geq 2$ and $M_0 = M_1 = 1$. The first few Motzkin numbers are $(M_n)_{n=0}^{\infty} = \{1, 1, 2, 4, 9, 21, 51, 127, 323, 835, 2188, \ldots\}$.

It is obvious that $M_n \leq 3^n$. A more precise asymptotic expression for Motzkin numbers,

$$M_n = \frac{3^n}{n^{3/2}} \cdot \sqrt{\frac{27}{4\pi}} \cdot (1 + O(1/n)),$$

can be deduced from the generating function

$$\sum_{n=0}^{\infty} M_n x^n = \frac{1 - x - \sqrt{(1-3x)(1+x)}}{2x^2}.$$

We represent the steps $(1,0)$, $(1,1)$, $(1,-1)$ of a Motzkin path by the vertical moves 0,1,-1, respectively. (We omit the horizontal moves since they are always 1.) Thus, a Motzkin path of length n is represented as a string of n symbols of the alphabet $\{1, -1, 0\}$.

Motzkin numbers have many different interpretations [14]. For example, there is a correspondence between Motzkin paths and drawing chords in an outerplanar graph.

2.2 Representation of States

A *state* represents the information about a partial polyomino at a given moment, as far as it is necessary to determine which cells can be added to make the partial polyomino a full polyomino.

We encode a state by its *signature* by first labelling the boundary cells as indicated in Figure 2. The signature of a partial polyomino is given as a collection of sets of occupied boundary cells. Each set represents the boundary cells of one connected component; see Figure 3 for an example.

A signature is not an arbitrary collection of disjoint subsets of $\{1, \ldots, W\}$. First of all, if two adjacent cells i and $i+1$ are occupied, they must belong to the same component. Moreover, the different components must be *noncrossing*: For $i < j < k < l$, it is impossible that i and k belong to one component and j and l belong to a different component; these two components would have to cross on the twisted cylinder.

A *valid* signature (or state) is a signature obeying these two rules. (This includes the "empty" state in which no boundary cell is occupied.)

The states can also be encoded by Motzkin paths of length $W+1$. In Section 4 we prove a bijection between the set of valid signatures and the set of Motzkin paths.

Figure 3 gives an example of both encodings of the same state, as a signature in the form of a set of sets and as a Motzkin path. In the notation of states as sets of sets, we use angle brackets to avoid an excessive accumulation of braces.

We prefer the term *state* when we regard a state as an abstract concept, without regard to its representation.

The encoding by signatures is a very natural representation of the states, but it is expensive. In our program we use the representation as Motzkin paths, which is much more efficient. Indeed, we rank the Motzkin paths, i.e., we represent the states by an integer from 2 to M, where $M = M_{W+1}$ is the number of Motzkin

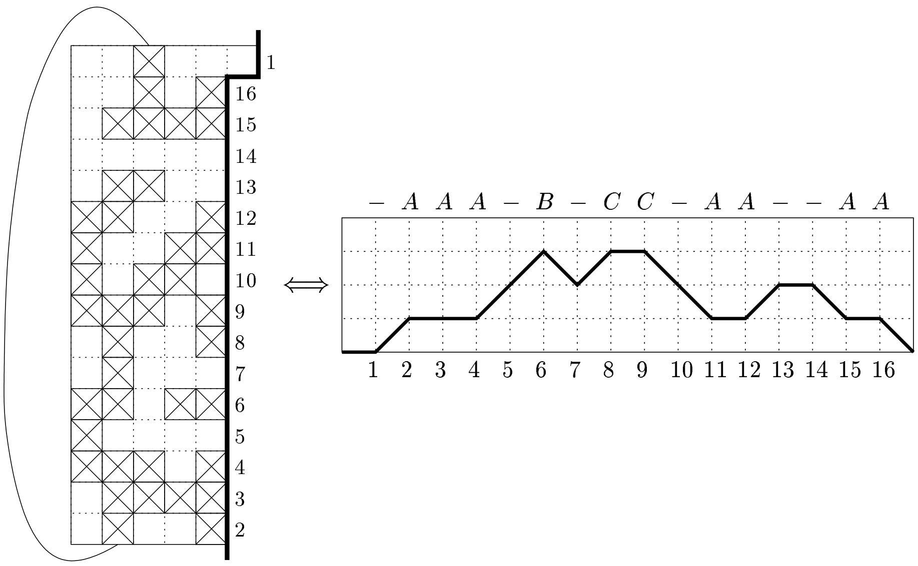


Figure 3: Left: A snapshot of the boundary line (solid line) during the transfer-matrix calculation. This state is encoded by the signature $\langle\{2,3,4,11,12,15,16\},\ \{6\},\ \{8,9\}\rangle$. Note that the bottom cell of the second column is adjacent to the top cell of the third column. The numbers are the labels of the boundary cells. Right: the same state encoded as the Motzkin path $(0,1,0,0,1,1,-1,1,0,-1,-1,0,1,0,-1,0,-1)$. For a better visualization of the state, we assign a symbol to each connected component and the symbol "−" to empty cells.

paths of length $W+1$. The ranking and unranking operations are described later in Section 5.1.

There also many other possible ways to encode the states. In his algorithm, Jensen [6] used signature strings of length W containing the five digits 0–4, which Knuth [10] replaced by the more intuitive five-character alphabet {`0`, `1`, `(`, `)`, `-`}. Conway [2] used strings of length W with eight digits 0–7.

3 Counting Polyominoes on a Twisted Cylinder

Let $Z_W(n)$ be the number of polyominoes of size n on our twisted cylinder of width W. It is related to the number $A(n)$ of polyominoes in the plane as follows:

Lemma 1. *For any W, we have $Z_W(n) \le A(n)$.*

Proof. We construct an injective function from n-ominoes on the cylinder to n-ominoes on the plane. First, it is clear that an n-omino X in the plane can be mapped to a polyomino $\alpha(X)$ on the cylinder, by simply wrapping it on the cylinder. This may cause different cells of X to overlap, which, therefore, $\alpha(X)$ may have fewer than n cells.

On the other hand, an n-omino Y on the cylinder can always be unfolded into an n-omino in the plane, usually in many different ways: Refer to the subgraph $G(Y)$ of the grid Z^2 that is generated by the vertex set Y. (The squares of the grid can be represented as the vertices of the infinite grid graph Z^2.) Select any spanning tree T in $G(Y)$. This spanning tree can be uniquely unfolded from Z^2 into the plane, and it defines an n-omino $\beta(Y)$ on the plane. $\beta(Y)$ will have all adjacencies between cells that were preserved in T, but some adjacencies of Y may be lost. When rolling $\beta(Y)$ back onto Z^2, there will be no overlapping cells and we retrieve the original polyomino Y:

$$\alpha(\beta(Y)) = Y$$

It follows that β is an injective mapping. □

The mapping β is, in general, far from unique. As soon as $G(Y)$ contains a cycle that "wraps around" the cylinder (i.e., that is not contractible), there are many different ways to unroll Y into the plane.

Klarner's constant λ, which is the growth rate of $A(n)$, is lower bounded by the growth rate λ_W of $Z_W(n)$, that is:

$$\lambda \geq \lambda_W = \lim_{n\to\infty} \frac{Z_W(n+1)}{Z_W(n)}$$

We enumerate *partial polyominoes* with n cells in a given state. The point of the twisted cylinder grid is that when adding a cell, we always repeat the same 2-step operation:

1. *Add new cell*: Update the state. If the cell is empty, the size of the polyomino remains the same. If the cell is occupied, the size grows by one unit.

2. *Rotate one position*: Shift the state, i.e., rotate the cylinder one position so that cell W becomes invisible, the labels $1 \ldots W-1$ are shifted by $+1$, and the new added cell is labelled as 1.

See the illustration for $W = 3$ in Figure 4.

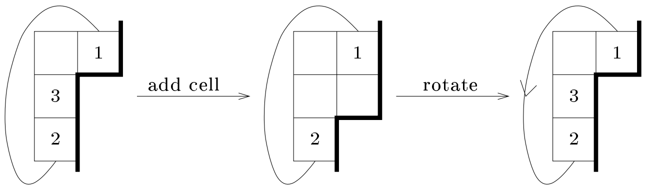


Figure 4: Addition of new cell and rotation.

In Section 5.2 we describe how the states, encoded by Motzkin paths, are updated when adding a cell and rotating.

3.1 System of Equations

3.1.1 Successor states

Let $\mathcal{S}$ be the set of all non-empty valid states. For each state $s \in \mathcal{S}$, there are two possible *successor states* each time a new cell is added and the grid is rotated, depending on whether the new cell is empty or occupied. Given s, let $succ_0(s)$ ($succ_1(s)$) be the successor state reached after adding a new empty (occupied) cell and rotating.

Example. For $W = 4$, the four boundary cells are labelled as in Figure 5. Consider the initial state $s = \langle\{1, 2\}, \{4\}\rangle$. After adding a new cell and rotating, we get $succ_0(s) = \langle\{2, 3\}\rangle$ and $succ_1(s) = \langle\{1, 2, 3\}\rangle$.

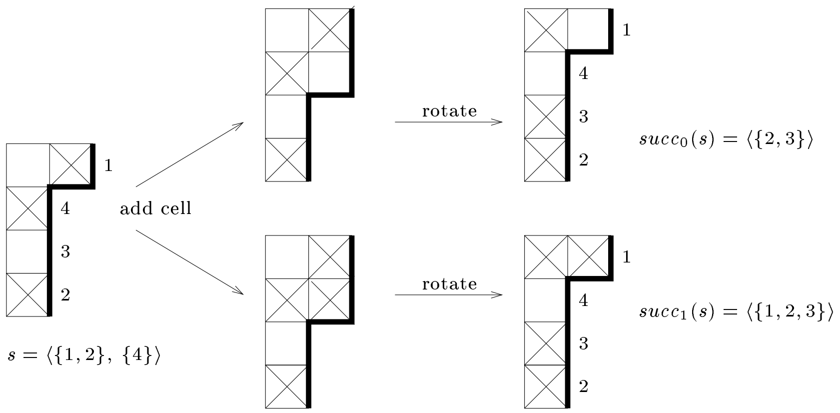


Figure 5: Example of successor states for $W = 4$. In the upper row the new cell is empty; in the lower row it is occupied.

Note that $succ_0(s)$ does not exist if, when adding an empty cell from an initial state s, some connected component becomes isolated from the boundary. (In this case a connected polyomino could never be completed.) This happens exactly when the component $\{W\}$ appears in s. For example, for $W = 3$, $succ_0(\langle\{3\}\rangle)$ and $succ_0(\langle\{1\},\{3\}\rangle)$ are not valid states, since in both cases the component containing 3 is forever isolated after the addition of an empty cell.

3.1.2 Transfer equations for counting polyominoes of a given size

Define the vector $\mathbf{x}^{(i)}$ of length $|\mathcal{S}|$ with components:

$$\mathbf{x}_s^{(i)} := \sharp\{\text{partial polyominoes with } i \text{ occupied cells in state } s\} \tag{2}$$

Lemma 2. *For each $n \in \mathbb{N}$, we have $\mathbf{x}^{(n)}_{\langle\{W\}\rangle} = Z_W(n)$.*

Proof. Given a polyomino of size n, the last added cell is the lower cell of its last column. This is the last cell that we see, the nth cell. If we add $W-1$ empty cells once the polyomino is completed, due to the rotation process of the twisted cylinder, the last occupied cell is labelled W and we always reach the state $\langle\{W\}\rangle$. Hence $x^{(n)}_{\langle\{W\}\rangle}$ equals the number of polyominoes of size n on the twisted cylinder. □

The following recursion keeps track of all operations:

$$\mathbf{x}_s^{(i+1)} = \sum_{s':s=succ_0(s')} \mathbf{x}_{s'}^{(i+1)} + \sum_{s':s=succ_1(s')} \mathbf{x}_{s'}^{(i)} \qquad \forall s \in \mathcal{S} \tag{3}$$

Note that the vector $\mathbf{x}^{(i+1)}$ depends on itself. There is, however, no cyclic dependency since we can order the states so that $succ_0(s)$ appears before s. This is done by grouping the states into sets $G_1, G_2, \ldots, G_W$ such that

$$G_k = \{\, s \in \mathcal{S} : k \text{ is the smallest label of an ocuppied cell}\,\}$$

For example, for $W = 3$ we have $G_1 = \{\langle\{1\}\rangle, \langle\{1,2\}\rangle, \langle\{1,3\}\rangle, \langle\{1,2,3\}\rangle, \langle\{1\},\{3\}\rangle\}$, $G_2 = \{\langle\{2\}\rangle, \langle\{2,3\}\rangle\}$ and $G_3 = \{\langle\{3\}\rangle\}$.

Proposition 1. *For each state $s \in G_k$, $k = 1\ldots W$, $succ_0(s)$ (if valid) belongs to G_{k+1} and $succ_1(s)$ belongs to G_1.*

Proof. For computing $succ_0(s)$ we first remove W from s, and second we shift each label l to $l+1$ ($l = 1 \mathrel{..} W-1$). As the smallest label is then incremented by one, the resulting state belongs to G_{k+1}. Note that the unique state belonging to G_W is $\langle\{W\}\rangle$, and there is no $succ_0(\langle\{W\}\rangle)$ in this case.

For computing $succ_1(s)$ we always add an occupied cell with label 1, so 1 always appears in $succ_1(s)$, hence the resulting state belongs to G_1. □

We can, therefore, use (2) to compute $\mathbf{x}^{(i+1)}$ from $\mathbf{x}^{(i)}$ if we process the states in the order of the groups to which they belong, as $G_W, G_{W-1}, \ldots, G_1$.

Corollary 1. *The states ordered as described above ensure that $succ_0(s)$ appears before s.* □

We draw a layered digraph (layers from 1 to n), with nodes $\mathbf{x}_s^{(i)}$ at layer i, for all $s \in \mathcal{S}$ and $i = 1 \mathrel{..} n$, and arcs from each node $\mathbf{x}_s^{(i)}$ to nodes $\mathbf{x}_{succ_0(s)}^{(i)}$ and $\mathbf{x}_{succ_1(s)}^{(i+1)}$, $i = 1 \mathrel{..} n-1$. We call this digraph the *recursion graph*. For simplicity, we denote at the same time, by $\mathbf{x}_s^{(i)}$, the node and its label, the number of partial polyominoes with i cells in state s.

Consider two layers i and $i+1$. The system of equations (3) is represented by drawing arcs from each node $\mathbf{x}_s^{(i)}$ to its successor nodes, $\mathbf{x}_{succ_0(s)}^{(i)}$ and $\mathbf{x}_{succ_1(s)}^{(i+1)}$. Figure 6 shows two successive layers for $W = 3$. Figure 8 shows the recursion graph for $W = 3$. It follows from Corollary 1 that the recursion graph is acyclic.

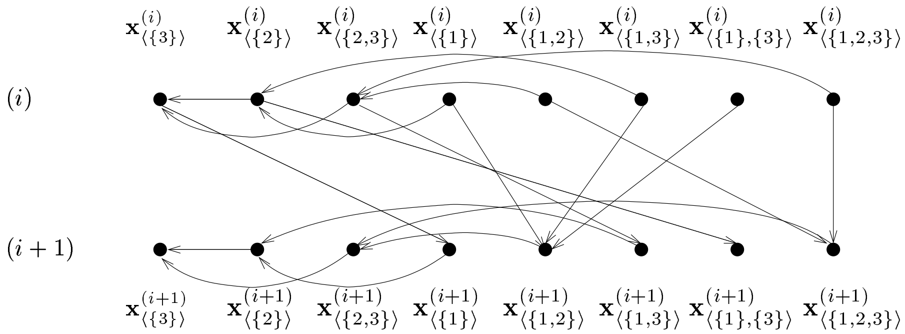


Figure 6: Schematic representation of the system (3) for $W = 3$.

In Figure 7 we show a schematic representation of the general graph, where the nodes are grouped together according to their corresponding set G_k, and instead of arcs between the original nodes we draw arcs between groups, using Proposition 1.

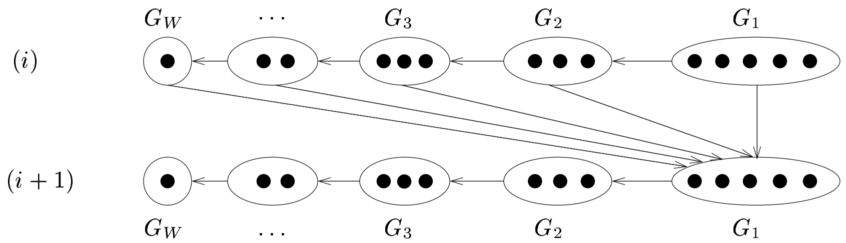


Figure 7: Representation of system (3) by groups.

3.1.3 Matrix notation

The system of equations (3) can also be written in a matrix form. We store the set of operations in two transfer matrices A and B, where rows correspond to the initial states, and columns correspond to the successor states. In A, for each row s there is a 1 at column $succ_0(s)$ (if $succ_0(s)$ is a valid state). In B, for each row s there is a 1 at column $succ_1(s)$. All the other entries are zeros. Our ordering of the states implies that A is strictly lower triangular.

Then system (3) translates into a matrix form as

$$\mathbf{x}^{(i+1)} = \mathbf{x}^{(i+1)}A + \mathbf{x}^{(i)}B, \tag{4}$$

where $\mathbf{x}^{(i)}$ is regarded as a row vector. We call this the *forward* iteration. The equation (4) can be written as

$$\mathbf{x}^{(i+1)} = \mathbf{x}^{(i)}T_{\text{for}} \tag{5}$$

with $T_{\text{for}} = B(1-A)^{-1}$.

3.2 Iterating the Equations

We start the iteration with the initial vector $\mathbf{x}^{(0)} := \mathbf{0}$ and set $\mathbf{x}^{(1)}_{\langle\{1\}\rangle} := 1$. We can then use (3) to obtain the remaining states of $\mathbf{x}^{(1)}$. We can imagine an initial column of empty cells, with the first occupied cell being the one with label 1, on the second column. Equivalently, we can begin directly with the initial conditions:

$$\begin{aligned} \mathbf{x}^{(1)}_{\langle\{w\}\rangle} &= 1 \qquad \text{for } w = 1, \dots, W \\ \mathbf{x}^{(1)}_{s} &= 0 \qquad \text{for all other states } s \in \mathcal{S} \end{aligned} \tag{6}$$

We iterate the system (3) so that at each step $\mathbf{x}^{(i)}$ becomes the old vector $\mathbf{x}^{\text{old}}$, and $\mathbf{x}^{(i+1)}$ is the newly-computed vector $\mathbf{x}^{\text{new}}$. This produces a recursion that, as we see later, gives the number of polyominoes of a given size.

We can prove the following lemma:

Lemma 3. *$Z_W(n)$ equals the number of paths from node $\mathbf{x}^{(1)}_{\langle\{1\}\rangle}$ to node $\mathbf{x}^{(n)}_{\langle\{W\}\rangle}$ in the recursion graph.*

Proof. Without loss of generality, we assume that all polyominoes begin at the same cell of the infinite twisted cylinder grid. This is a kind of normalization: we set the first appearing cell (the upper cell of the first column) of each polyomino to the same position.

Starting with the initial conditions (6), we proceed with the recursion adding one cell at each step and rotating. At each layer (i), the number of distinct paths starting at $\mathbf{x}^{(1)}_{\langle\{1\}\rangle}$ and arriving to a node $\mathbf{x}^{(i)}_{s}$ is the accumulation of all the arcs of the forward iteration. It represents the number of partial polyominoes with i cells in state s.

By Lemma 2, the process for completing a polyomino of size n ends at node $\mathbf{x}^{(n)}_{\langle\{W\}\rangle}$. Since each addition of a cell, empty or occupied, is represented by an arc on the recursion graph, the number of distinct paths from $\mathbf{x}^{(1)}_{\langle\{1\}\rangle}$ to $\mathbf{x}^{(n)}_{\langle\{W\}\rangle}$ represents the $Z_W(n)$ different ways of constructing a polyomino of size n.

Thus, we enumerate all polyominoes iterating the equations, which amounts to following all paths starting at $\mathbf{x}^{(1)}_{\langle\{1\}\rangle}$ and ending at $\mathbf{x}^{(n)}_{\langle\{W\}\rangle}$. □

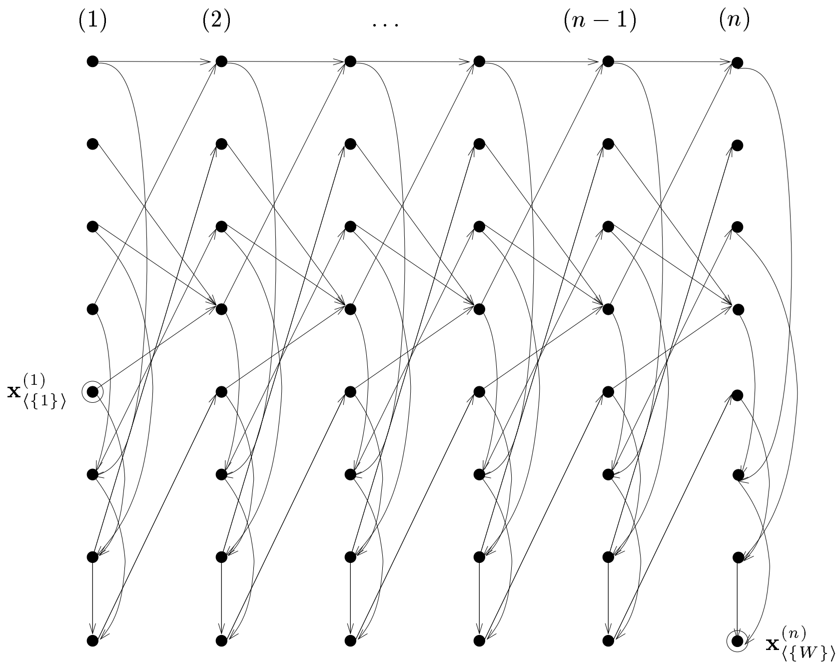


Figure 8: Recursion graph, $W = 3$

3.2.1 Backward recursion

In our program we use an alternative recursion that is, as we discuss later, preferable from a practical point of view. We iterate the following system of equations:

$$\mathbf{y}_s^{(i-1)} = \mathbf{y}^{(i-1)}_{succ_0(s)} + \mathbf{y}^{(i)}_{succ_1(s)} \qquad \forall s \in \mathcal{S} \tag{7}$$

If $succ_0(s)$ does not exist, the corresponding value is simply omitted. In other words, we walk backwards on the recursion graph. This translates into a matrix form as

$$\mathbf{y}^{(i-1)} = A\mathbf{y}^{(i-1)} + B\mathbf{y}^{(i)},$$

which can be written as

$$\mathbf{y}^{(i-1)} = T_{\text{back}}\mathbf{y}^{(i)}$$

with $T_{\text{back}} = (I - A)^{-1}B$.

As before, the vector $\mathbf{y}^{(i-1)}$ depends on itself, but there is no cyclic dependency, due to Corollary 1.

Consider the initial vector $\mathbf{y}^{(0)}$. We set

$$\mathbf{y}^{(0)} := \mathbf{0} \qquad \text{and} \qquad \mathbf{y}^{(-1)}_{\langle\{W\}\rangle} := 1, \tag{8}$$

and use (7) to obtain the remaining states of $\mathbf{y}^{(-1)}$.

Starting with the initial conditions (8), we iterate the system (7) so that at each step $\mathbf{y}^{(-i)}$ becomes the old vector $\mathbf{y}^{\text{old}}$ and $\mathbf{y}^{(-i-1)}$ is the newly-computed vector $\mathbf{y}^{\text{new}}$. We thus obtain the vectors $\mathbf{y}^{(0)}, \mathbf{y}^{(-1)}, \mathbf{y}^{(-2)}, \ldots, \mathbf{y}^{(-i)}, \mathbf{y}^{(-i-1)}, \ldots$.

As can be seen from the recursion graph,

$$\mathbf{y}_s^{(-i)} = \sharp\{\text{ paths from node } \mathbf{x}_s^{(n-i+1)} \text{ to node } \mathbf{x}^{(n)}_{\langle\{W\}\rangle}\} \tag{9}$$

Lemma 4. *Starting with initial conditions* (6) *and* (8), *we have*

$$\mathbf{y}^{(-n)}_{\langle\{1\}\rangle} = Z_W(n) = \mathbf{x}^{(n)}_{\langle\{W\}\rangle}.$$

Proof. By (9), $\mathbf{y}^{(-n)}_{\langle\{1\}\rangle}$ corresponds to the number of paths from node $\mathbf{x}^{(1)}_{\langle\{1\}\rangle}$ to node $\mathbf{x}^{(n)}_{\langle\{W\}\rangle}$, which by Lemma 3 is the number of polyominoes of size n on a twisted cylinder of width W. The second equation is given by Lemma 2. □

Lemma 5. *The matrices defining the forward and backward iterations have the same eigenvalues.*

Proof. The matrices $T_{\text{for}} = B(1-A)^{-1}$ and $T_{\text{back}} = (I-A)^{-1}B$ have the same eigenvalues because they are similar: $T_{\text{back}} = (I-A)^{-1}T_{\text{for}}(I-A)$. □

Independently of this proof, we see that the forward and the backward iterations must have the same dominant eigenvalues since both iterations define the number of polyominoes:

$$\lambda_W = \lim_{n\to\infty} \frac{Z_W(n+1)}{Z_W(n)} = \lim_{n\to\infty} \frac{\mathbf{x}^{(n+1)}_{\langle\{W\}\rangle}}{\mathbf{x}^{(n)}_{\langle\{W\}\rangle}} = \lim_{n\to\infty} \frac{\mathbf{y}^{(-n-1)}_{\langle\{1\}\rangle}}{\mathbf{y}^{(-n)}_{\langle\{1\}\rangle}}$$

3.3 The Growth Rate λ_W

In this section we explain how we bound the growth rate λ_W. First, we need to prove that there is a unique eigenvalue of largest absolute value. For this, we apply the Perron-Frobenius Theorem to our transfer matrix.

For stating the Perron-Frobenius Theorem, we need a couple of definitions. A nonnegative matrix T is an *irreducible matrix* if for each entry i, j, there exists a $k \geq 1$ such that the (i,j) entry of T^k is strictly positive. A matrix T is irreducible if and only if its underlying graph is *strongly connected.*

A nonnegative matrix T is a *primitive matrix* if there exists a $k \geq 1$ such that all entries of T^k are strictly positive. A sufficient condition for a nonnegative matrix to be primitive is to be an irreducible matrix with at least one positive main diagonal entry.

The Perron-Frobenius Theorem is stated as follows, see [5].

Theorem 1. *Let T be a primitive nonnegative matrix. Then there is a unique eigenvalue with largest absolute value. This eigenvalue λ is positive, and its has an associated eigenvector which is positive. This vector is the only nonnegative eigenvector.*

Moreover, if we start the iteration $\mathbf{x}^{(i+1)} = \mathbf{x}^{(i)}T$ with any nonnegative nonzero vector $\mathbf{x}^{(0)}$, the iterated vectors, after normalizing their length to $\mathbf{x}^{(i)}/\|\mathbf{x}^{(i)}\|$, converge to this eigenvector. □

The matrix can be written as $T_{\text{for}} = B(1-A)^{-1} = B(1+A+A^2+A^3+\cdots)$. A is a triangular matrix with zeros on the diagonal, hence it is nilpotent and the above series expansion is finite. Since all the entries of A and B are nonnegative, it follows that this is also true for the entries of T_{for}.

Note that $succ_1(\{1,\ldots,W\}) = \{1,\ldots,W\}$. Hence, the diagonal entry of B corresponding to the state $\{1,\ldots,W\}$ is 1 (all other diagonal entries of B are zero). Since both A and B are nonnegative, the diagonal entry of T_{for} corresponding to the state $\{1,\ldots,W\}$ is positive.

However, in our case the graph is not strongly connected because some valid states cannot be reached. It will turn out that these states have no predecessor states, and the remaining states, with we will call *reachable*, form a strongly connected graph. Hence, we can apply the Perron-Frobenius Theorem to this subset of states. The result will then carry over to the original iteration: in the forward recursion, the unreachable states will always have value 0; and in the backward recursion, their value has no influence on successive iterations.

Let us now analyze the states in detail. Consider the state $\langle\{1,3\},\{5\}\rangle$ for $W=5$. Some cell, which is adjacent to boundary cell 1 has to be occupied since 1 is connected to 3. This occupied cell, cannot be cell 2 since it is not present in the signature. There is one remaining cell, which is adjacent to 1, but this cell is also adjacent to 5. It follows that 1 and 5 must belong to the same component. Therefore, this state corresponds to no partial polyomino, and it is not the successor of any other state. In fact, this type of example is the only case where a state is not reachable. We call a signature (or state) *unreachable* if

1. cell 1 is occupied, but it does not form a singleton component of its own;
2. cell 2 is not occupied; and
3. cell W is occupied, but it does not lie in the same component as cell 1.

Otherwise, we call a state *reachable*.

Lemma 6.

1. *Every non-empty reachable state can be reached from every other state, by a path that starts with a $succ_1$ operation.*
2. *No successor of any state is an unreachable state.*

Proof. We prove that from every state we can reach the state $\{1,\dots,W\}$ by a sequence of successor operations, and vice versa. If we start from any state and apply a sequence of W $succ_1$-operations, we arrive at state $\{1,\dots,W\}$.

To see that some reachable state s can be reached from $\{1,\dots,W\}$, we construct a partial polyomino corresponding to this state. We start with the boundary cells that are specified by the given signature. The problem is to extend these cells to the left to a partial polyomino with the given connected components. From the definition of reachable states it follows that adding an arbitrary number of cells to the left of existing cells does not change the connectivity between existing connected components.

The process is now similar to that for polyominoes in the plane; we do not need the wrap-around connections between row 1 and row W. We leave cells that are singleton components unconnected. We add cells to the left of all occupied cells that are not singleton components. After growing three layers of new cells, pieces that should form components and that have no other components nested inside (except singleton components) can be connected together. The remaining pieces can be further grown to the left and connected one by one. Finally, we grow one of the outermost components by adding a large block of occupied cells, such that several columns are completely occupied. See Figure 9.

In constructing this partial polyomino cell by cell, we pass from state $\{1,\dots,W\}$ to the current state s. Since the succeeding operations correctly model the growth of partial polyominoes, we get a path from $\{1,\dots,W\}$ to state s in the recursion graph.

The second part of the lemma is easy to see. Since an unreachable state s contains cell 1, it can only be a $succ_1$-successor. Since 1 is not a singleton component, the previous state $\hat{s}$ must have contained cell W (before the cyclic renumbering), as well as cell $W-1$ (which is renumbered to W in s). W and $W-1$ must belong to the same component in $\hat{s}$; hence, they will be in the same component as the new cell 1 in s, which is a contradiction. ◻

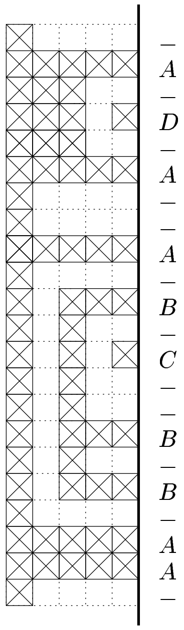


Figure 9: Construction for reaching state $-AA-B-B--C-B-A--A-D-A-$ from state $\{1\dots W\}$, or $AA\dots A$.

Let us clarify the relation between the recursion graph, which consists of successive layers, and the graph G_{for} that represents the structure of T_{for}, which has just one vertex for every state.

We have $T_{\text{for}} = B(1-A)^{-1} = B(1 + A + A^2 + A^3 + \cdots)$. An entry in the matrix $(1 + A + A^2 + A^3 + \cdots)$ corresponds to a sequence of 0 or more edges that are represented by the adjacency matrix A, i.e., a sequence of $succ_0$ operations. Therefore, T_{for} has a positive entry in the row corresponding to state s and the column corresponding to state t (and G_{for} has an edge from s to t) if and only if t can be reached from s by a single $succ_1$ operation followed by 0 or more $succ_0$ operations.

Thus a path P from state s to state t in G_{for} corresponds to a path P' from s on some layer of the recursion graph of a vertex t on some other layer of the recursion graph. This path starts with a $succ_1$ edge, but is otherwise completely arbitrary.

Conversely, each path in the recursion graph that starts with a $succ_1$ edge is reflected by some path in G_{for}. This leads to the following statement.

Lemma 7. *T_{for} has a unique eigenvalue λ_W of largest absolute value. This eigenvalue is positive, and has multiplicity one and a nonnegative corresponding eigenvector.*

Starting the forward recursion with any nonnegative nonzero vector will yield a sequence of vectors that, after normalization, converges to this eigenvector.

Proof. We first look at the submatrix $\bar{T}_{\text{for}}$ of T_{for} that consists only of rows and columns for reachable states.

By Lemma 6 and the above considerations, the graph of this matrix is strongly connected, and the matrix is irreducible. The matrix T_{for} has at least one positive diagonal entry. This is also true for the submatrix $\bar{T}_{\text{for}}$, since this mentioned entry corresponds to a reachable state. Hence, $\bar{T}_{\text{for}}$ is primitive.

By the Perron-Frobenius Theorem, the statement of the lemma holds for this reduced matrix. The sequence of iterated vectors $\bar{\mathbf{x}}$ converges (after normalization) to the Perron-Frobenius eigenvector, the unique nonnegative eigenvector, which corresponds to the largest eigenvalue.

If we extend the matrix to the full matrix, the second part of Lemma 6 implies that the components that correspond to unreachable states in $\mathbf{x}$ will be zero after the first iteration, no matter what the starting vector is. It means that the additional, unreachable components have no further influence on the iteration. Moreover, if the initial vector is nonzero, the next version of it will have a nonzero component for a reachable state, which ensures that the Perron-Frobenius theorem can be applied from this point on, and convergence happens as for the reduced vector $\bar{\mathbf{x}}$.

Compared to the reduced matrix $\bar{T}_{\text{for}}$, the additional columns of T_{for}, which correspond to unreachable states, are all zero. It follows that T_{for} has all eigenvalues of $\bar{T}_{\text{for}}$, plus an additional set of zero eigenvalues. Thus the statement about the unique eigenvector of largest absolute value holds for T_{for}, just as for $\bar{T}_{\text{for}}$. □

By Lemma 5, λ_W is the unique eigenvalue of largest absolute value of T_{back} as well.

Lemma 8. *λ_W is the unique eigenvalue of T_{back} of largest absolute value. This eigenvalue is positive, and has multiplicity one and a positive corresponding eigenvector.*

Starting the backward recursion with any nonnegative vector with at least one nonzero entry on a reachable state will yield a sequence of vectors which, after normalization, converges to this eigenvector.

Proof. The first sentence follows from Lemma 7 by Lemma 5. We consider the iterations with the reduced matrix $\bar{T}_{\text{back}}$ and a reduced vector $\bar{\mathbf{y}}$ for the reachable states, as in Lemma 7. If follows that this iteration converges to the Perron-Frobenius eigenvector, which is positive.

If we compare this iteration with the original recursion (7), we see that the components of $\mathbf{y}^{(i)}$ and $\mathbf{y}^{(i-1)}$ that correspond to unreachable states have no influence on the recursion because they do not appear on the right-hand side. It follows that the subvector of reachable states in $\mathbf{y}^{(i)}$ has exactly the same sequence of iterated versions as $\bar{\mathbf{y}}^{(i)}$.

The unreachable states in $\mathbf{y}^{(i-1)}$ can be calculated directly from $\bar{\mathbf{y}}^{(i-1)}$ and $\bar{\mathbf{y}}^{(i)}$ by (7). It follows, by taking the limit, that the the unreachable states in the eigenvector can be calculated from the eigenvector of $\bar{T}_{\text{back}}$ using (7), and the fact that the whole eigenvector is positive. □

Lemmas 7 and 8 imply that

$$Z_W(n) \leq c(\lambda_W)^n,$$

for some constant c. This is another way to express that λ_W is the growth rate of $Z_W(n)$. The following lemma (which is actually a key lemma in one possible way to prove the Perron-Frobenius Theorem) shows how to compute bounds for λ_W.

Lemma 9. *Let $\mathbf{y}^{\text{old}}$ be any vector with positive entries and let $\mathbf{y}^{\text{new}} = T_{\text{back}}\mathbf{y}^{\text{old}}$. Let λ_{low} and λ_{high} be, respectively, the minimum and maximum values of $\mathbf{y}^{\text{new}}_s/\mathbf{y}^{\text{old}}_s$ over all components s. Then, $\lambda_{\text{low}} \leq \lambda_W \leq \lambda_{\text{high}}$.*

Proof. From the definition of λ_{low} and λ_{high}, we have

$$\lambda_{\text{low}}\mathbf{y}^{\text{old}} \leq \mathbf{y}^{\text{new}} \leq \lambda_{\text{high}}\mathbf{y}^{\text{old}}.$$

Let $\mathbf{y}^*$ be the eigenvalue corresponding to the eigenvalue λ_W:

$$T_{\text{back}}\mathbf{y}^* = \lambda_W\mathbf{y}^*$$

By scaling $\mathbf{y}^*$, we can achieve $\mathbf{y}^* \le \mathbf{y}^{\text{old}}$ and $\mathbf{y}^*_s = \mathbf{y}^{\text{old}}_s$ for some state $s \in \mathcal{S}$. Then we have

$$\lambda_W \mathbf{y}^* = T_{\text{back}} \mathbf{y}^* \le T_{\text{back}} \mathbf{y}^{\text{old}} = \mathbf{y}^{\text{new}} \le \lambda_{\text{high}} \mathbf{y}^{\text{old}}.$$

This is true in particular for the s component: $\lambda_W \mathbf{y}^*_s \le \lambda_{\text{high}} \mathbf{y}^{\text{old}}_s$. Moreover, since we assumed $\mathbf{y}^*_s = \mathbf{y}^{\text{old}}_s$, this implies $\lambda_W \le \lambda_{\text{high}}$.

Analogously, we can achieve $\mathbf{y}^* \ge \mathbf{y}^{\text{old}}$ and $\mathbf{y}^*_s = \mathbf{y}^{\text{old}}_s$ for some state $s \in \mathcal{S}$. In this case we have $\lambda_W \mathbf{y}^* \ge \lambda_{\text{low}} \mathbf{y}^{\text{old}}$, which implies $\lambda_W \ge \lambda_{\text{low}}$. □

Thus, $\lambda_{\text{low}} \le \lambda_W \le \lambda$ is a lower bound on Klarner's constant as well.

Our program iterates the equations until λ_{low} and λ_{high} are close enough.

4 Bijection between Signatures and Motzkin Paths

Consider a (partial) polyomino on a twisted cylinder of width W and unrestricted length. Figure 10 shows all the different possible boundaries and their states for $1 \le W \le 4$.

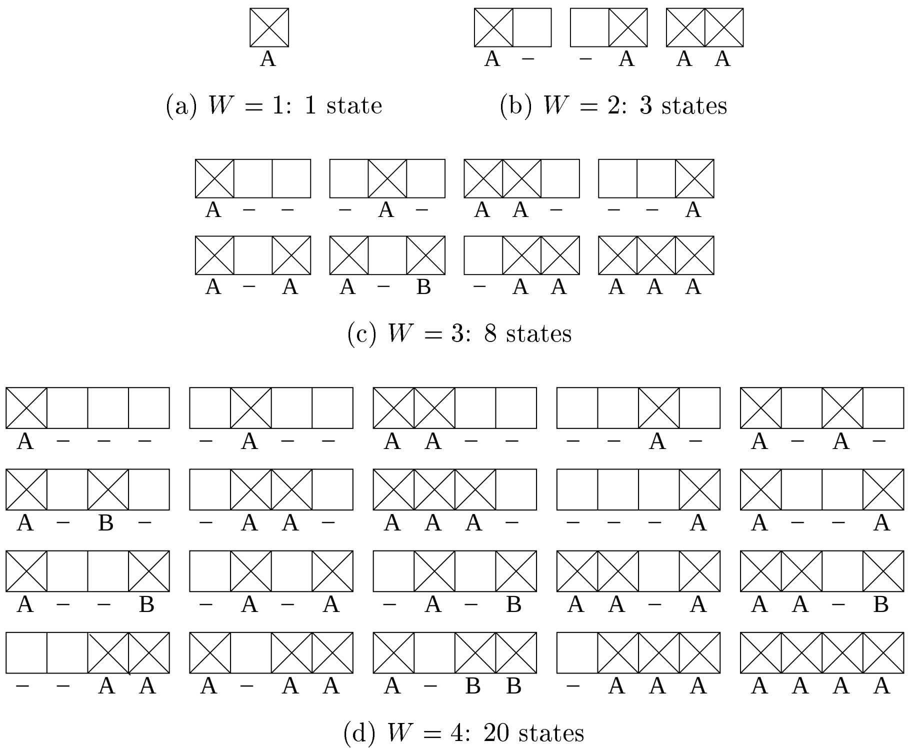


Figure 10: All states for $1 \le W \le 4$. For a better visualization, boundaries are drawn horizontally and states indicated using a slightly different notation. We assign a symbol to each connected component and the symbol '–' to empty cells.

A few points about Motzkin paths are in order here. A Motzkin path of length $W+1$ is an array $\mathbf{p} = (p[0], \ldots, p[W])$, where each component $\mathbf{p}[i]$ is one step $0, 1, -1$. The *levels* are, as the name suggests, the different y-coordinates of the paths. We can also assign a level to each node of the path, by

$$\mathit{level}[i+1] := \mathit{level}[i] + p[i] \qquad i = 0 \,..\, W$$

with $\mathit{level}[0] = 0$.

In the following theorem we give the relation between the number of signature strings and Motzkin numbers.

Theorem 2. *There is a bijection between valid states of length W and Motzkin paths of length $W+1$, Hence, the number of nonempty valid states is $M_{W+1}-1$.*

Proof. We explicitly describe the conversions (in both directions) between Motzkin paths and signatures as sets of sets. This is a bijective correspondence between both encodings of the states.

We convert a signature string to a Motzkin path as follows: Consider the edges between boundary cells. Edges between occupied cells of the same connected component or between empty cells are mapped to the horizontal step 0. For a block of consecutive, occupied cells of the same connected component, we distinguish between four cases:

- The left edge of the first block is mapped to 1.
- The left edge of all remaining blocks is mapped to -1.
- The right edge of the last block is mapped to -1.
- The right edge of all remaining blocks is mapped to 1.

When a new component starts, the path will rise to a new level $(+1)$. All cells of the component will lie on this level. When the component is interrupted (i.e., a block, which is not the last block of the component, ends), the path rises to a higher level $(+1)$, essentially pushing the current block on a stack, to be continued later. When the component resumes, the path will come down to the correct level (-1). At the very end of the component, the path will be lowered to the level that it had before the component was started (-1). For empty cells, the path lies on an even level, whereas occupied cells lie on odd levels. A connected component, which is nested within k other components, lies on level $1+2k$; see Figure 11 for an example.

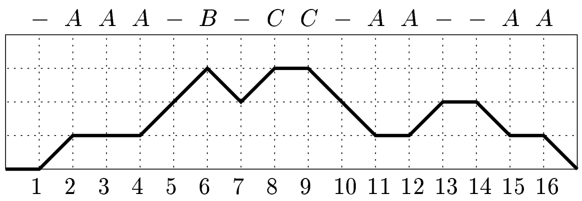


Figure 11: The Motzkin path $(0,1,0,0,1,1,-1,1,0,-1,-1,0,1,0,-1,0,-1)$, with $W=16$. The connected components $\{6\}$ and $\{8,9\}$ lie on level 3, nested within the component $\{2,3,4,11,12,15,16\}$ (on level 1). Cells $1,5,7,10,13,14$ are empty since they lie on levels 0 and 2.

The process can be better understood by the following simple algorithm that converts a Motzkin path to a signature. It maintains a stack of partially completed components in an array *current_set*[1], *current_set*[3], *current_set*[5], ... (The even entries of this array are not used.) The current element is always added to the topmost set on the stack.

Algorithm ConvertPathToSetofSets($\mathbf{p}$)
Input. A Motzkin path $\mathbf{p}[0\,..\,W]$
Output. The corresponding signature
level := 0

$signature := \{\}$
for $i = 0$ **to** W **do**
  if $\mathbf{p}[i] = 1$
    $level := level + 1$
    if *level* is odd (* Start a new set *)
      $current\_set[level] := \{\}$
  else if $\mathbf{p}[i] = -1$
    if *level* is odd (* Current set is complete. Store it *)
      $signature := signature \cup \{current\_set[level]\}$
    $level := level - 1$
  if *level* is odd
    Add i to $current\_set[level]$
return *signature*

Figure 3 shows an example of this bijection.

Finally, we explain the index shift between the number of signatures and Motzkin numbers. That is, why $S(W) = M_{W+1} - 1$. The shift occurs simply because a signature of length k is mapped to a path of length $k+1$, while the $(k+1)$st Motzkin number is the number of paths of length $k+1$. The missing state is the "empty" signature $\{\}$ that corresponds to the straight x-parallel path $(0, 0, \ldots, 0)$. □

5 Encoding States as Motzkin Paths

As defined before, $M = M_{W+1}$ denotes the number of Motzkin paths of length $W+1$. After discarding the empty signature, we have $M-1$ states.

5.1 Motzkin Path Generation

In order to store $\mathbf{x}^{(i)}$ in an array indexed by the states, we use an efficient data structure that allows us to generate all Motzkin paths of a given length $m = W+1$ in lexicographic order, as well to rank and unrank Motzkin paths.

Listing all Motzkin paths in lexicographic order establishes a bijection between all Motzkin paths and the integers between 1 and M. The operation of *ranking* refers to the direct computation of the corresponding integer for a given Motzkin path, and *unranking* means the inverse operation. Both processes can be carried out in $O(m)$ time. The data structure and the algorithms for ranking and unranking are quite standard; see for example [11, Section 3.4.1].

Construct a diagram such as the one in Figure 12, with $m+1$ nodes on the base and $\lceil \frac{m}{2} \rceil$ rows, representing the levels. Assign ones to all nodes of the upper-right diagonal. Running over all nodes from right to left, each node is assigned the sum of the values of its adjacent nodes to the right. This number is the number of paths that start in this node. At the end, the leftmost node will receive the value M, the number of Motzkin paths of length m. This preprocessing takes $O(m^2)$ time.

The bijection from the set of integers from 1 to M to the set of Motzkin paths of length m is established in the following way. Refer again to Figure 12, for an example with Motzkin paths of length $m = 5$. We proceed from left to right. Begin at the leftmost node. There are 21 Motzkin paths of length 5; the first nine start with the horizontal step (paths 1st till 9th), and the other 12 start with the up-step (paths 10th till 21st). If the first step is 0, the second node of the path is the one assigned the number 9 on the diagram. One can see that of these nine paths, four go straight (paths 1st till 4th) and five go up (paths 5th till 9th). If the first step is 1, the second node of the path is the one assigned the number 12 on the diagram. Of these 12 paths, four go down (paths 10th till 13th), five go straight (paths 14th till 18th) and the last three go up (paths 19th till 21th). This procedure continues

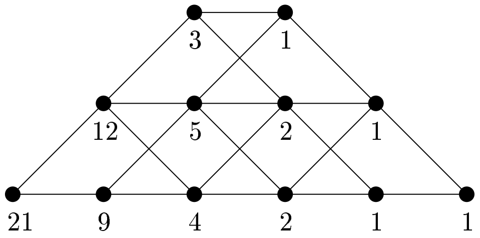


Figure 12: Generating Motzkin paths of length 5, $M_5 = 21$.

until the upper-right diagonal is reached. At this point the rest of the path goes always down ending at the rightmost node.

The following proposition shows that we obtain the paths in the desired order, hence no extra sorting is needed.

Proposition 2. *The Motzkin paths are generated in lexicographic order, satisfying the conditions of Corollary 1.*

Proof. Given a Motzkin path, the first nonzero step indicates the smallest label of an occupied cell—the group in which the state belongs. Thus a path $\mathbf{p}$ precedes a path $\mathbf{p}'$ in our ordering if the first nonzero step in $\mathbf{p}$ appears later than the first nonzero step in $\mathbf{p}'$.

Our numbering of paths also satisfies the following property: at each node, we assign the paths continuing with step -1, the lower numbers, those continuing with step 1, the upper numbers, and those continuing with step 0, the middle numbers. In particular, at each node on the base, we assign the paths continuing with step 1 the upper numbers and those continuing with step 0 the lower numbers. Accordingly, a path is assigned a lower number as long as the first nonzero step appears later. □

5.2 Motzkin Path Updating

Here we explain how the Motzkin paths are updated after the operations of adding a new cell and rotating.

First, we describe the updating for an initial state s encoded as sets of connected components. (This is also the representation used in the Maple program in Appendix B, at the beginning of the procedure *check*.) The rotation (shift) is always done by removing W, shifting each label l to $l+1$ for $l = 1 \,..\, W-1$, and labelling the new cell as 1. For every case we provide an example with $W = 3$.

1. **Add Empty Cell and Shift (Compute $succ_0(s)$)**
   - *If W does not appear in s*, just shift. Example: $succ_0(\langle\{2\}\rangle) = \langle\{3\}\rangle$.
   - *If W appears in s, but is not alone in its component*, delete the element W and shift. Example: $succ_0(\langle\{1,3\}\rangle) = \langle\{2\}\rangle$.
   - *If W appears in s, alone in its component as $\{W\}$*, then $succ_0(s)$ is not valid, since the cell W is always disconnected when an empty cell is added. Example: No $succ_0(\langle\{1\},\{3\}\rangle)$.
2. **Add Occupied Cell and Shift (Compute $succ_1(s)$)**
   - *If W and 1 are in the same component*, just shift. Example: $succ_1(\langle\{1,3\}\rangle) = \langle\{1,2\}\rangle$.
   - *If W and 1 appear in different components*, unite these two components and shift. Example: $succ_1(\langle\{1\},\{3\}\rangle) = \langle\{1,2\}\rangle$.

- *If 1 appears in s but W does not*, add the element W to the component containing 1 and shift. Example: $succ_1(\langle\{1,2\}\rangle) = \langle\{1,2,3\}\rangle$.
- *If W appears in s and 1 does not*, just shift. Example: $succ_1(\langle\{2,3\}\rangle) = \langle\{1,3\}\rangle$.
- *If 1 and W do not appear in s*, add a new component $\{W\}$ to s and shift. Example: $succ_1(\langle\{2\}\rangle) = \langle\{1\},\{3\}\rangle$.

Now we translate these operations into a Motzkin-path notation. Let $\mathbf{p} = (\mathbf{p}[0]\ldots\mathbf{p}[W])$ be a Motzkin path of length $W+1$ representing a state s. Below we describe the routines for computing the two possible successor states.

Given $\mathbf{p}$, the shifting is performed differently depending on whether the last step $\mathbf{p}[W]$ is 0 or -1. If $\mathbf{p}[W] = 0$, it means that the cell W is empty. The shifting is performed by cutting the last step and gluing it at the beginning of the path (i.e., shifting one position to the right). Consequently, the resulting path is $(0, \mathbf{p}[0\mathinner{..}W-1])$. On the other hand, if $\mathbf{p}[W] = -1$, the cell W is occupied. For a better exposition, the process in this case is described below in a separate routine ShiftW. See an example of the shifting of a path with $\mathbf{p}[W] = -1$ in Figure 13.

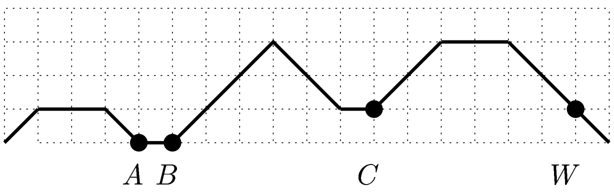


Figure 13: Example of the points A, B, and C.

The following algorithms for computing successors rearrange and change parts of the given Motzkin paths, depending on the first and last two steps on the input path. The algorithm refers to three positions A, B, and C in the input path. A is the leftmost position $A > 0$ such that $level[A] = 0$. If cell 1 is occupied, then $A-1$ is the largest element in the component containing 1. If $A = W+1$, it means that cells 1 and W lie in the same component. In a few cases, the algorithm makes a distinction depending on whether or not A equals $W+1$.

B is the rightmost position (for $B \le W$) such that $level[B] = 0$. If cell W is occupied, then $B+1$ is the smallest cell in the component containing W. If $A < W+1$, then $A \le B$.

Finally, C is defined as the rightmost position (for $C \le W-1$) such that $level[C] = 1$. C is used when cell W is occupied but does not form a singleton component. In this case C is the largest cell in the component containing W, and $C > B$.

In the interesting cases, we show how $\mathbf{p}$ is initially composed of different subsequences between the points A, B, or C, and how the output is composed of the same pieces, to make the similarities and the differences between the input and the output clearly visible. In most cases (except where the output contains only one "piece," and except in case $(0,\ldots,-1,-1)$ of AddEmptyCell), these pieces form Motzkin paths in their own right: the total sum of all entries is 0, and the paths never go below 0. (They may, however, be empty).

In the ordering of the cases, the rightmost element of $\mathbf{p}$ is considered to be the most important sorting criterion.

Algorithm AddEmptyCell
Input. A Motzkin path $\mathbf{p} = p[0\mathinner{..}W]$ representing a state s

Output. Updated Motzkin path representing $succ_0(s)$

Depending on the pattern of $\mathbf{p}$, perform one of the following operations:

$(\ldots, 0)$: (∗ W does not appear in s ∗)
return $(0, p[0\,..\,W-1])$

$(\ldots, 0, -1)$: (∗ W and $W-1$ appear in s ∗)
return $(0, p[0\,..\,W-2], -1)$

$(\ldots, -1, -1)$: (∗ $\mathbf{p} = (p[0\,..\,C-1], 1, p[C+1\,..\,W-2], -1, -1)$ ∗)
return $(0, p[0\,..\,B-1], -1, p[B+1\,..\,W-2], 0)$

$(\ldots, 1, -1)$: (∗ W forms a singleton component ∗)
return null

Algorithm AddOccupiedCell

Input. A Motzkin path $\mathbf{p} = p[0\,..\,W]$ representing s

Output. Updated Motzkin path representing $succ_1(s)$

Depending on the pattern of $\mathbf{p}$, perform one of the following operations:

$(0, \ldots, 0)$: (∗ 1 and W do not appear ∗)
return $(1, -1, p[1\,..\,W-1])$

$(1, \ldots, 0)$: (∗ 1 appears and W does not appear ∗)
return $(1, 0, p[1\,..\,W-1])$

$(0, \ldots, 1, -1)$: (∗ 1 does not appear and W is a singleton ∗)
return $(1, -1, p[1\,..\,W-2], 0)$

$(1, \ldots, 1, -1)$: (∗ 1 appears and W is a singleton ∗)
return $(1, 0, p[1\,..\,W-2], 0)$

$(0, \ldots, 0, -1)$: (∗ 1 does not appear and W is not a singleton ∗)
(∗ $\mathbf{p} = (0, p[1\,..\,B-1], 1, p[B+1\,..\,W-2], 0, -1)$ ∗)
return $(1, 1, p[1\,..\,B-1], -1, p[B+1\,..\,W-2], -1)$

$(1, \ldots, 0, -1)$: (∗ 1 and W appear, and W is not a singleton ∗)
if $A = W+1$ (∗ 1 and W are connected ∗)
then return $(1, 0, p[0\,..\,W-2], -1)$
else (∗ $\mathbf{p} = (1, p[1\,..\,A-2], -1, p[A\,..\,B-1], 1,$
$p[B+1\,..\,W-2], 0, -1)$ ∗)
return $(1, 0, p[1\,..\,A-2], 1, p[A\,..\,B-1], -1,$
$p[B+1\,..\,W-2], -1)$

$(0, \ldots, -1, -1)$: (∗ 1 does not appear and W is not a singleton ∗)
(∗ $\mathbf{p} = (0, p[1\,..\,B-1], 1, p[B+1\,..\,C-1], 1,$
$p[C+1\,..\,W-2], -1, -1)$ ∗)
return $(1, 1, p[1\,..\,B-1], -1, p[B+1\,..\,C-1], -1,$
$p[C+1\,..\,W-2], 0)$

$(1, \ldots, -1, -1)$: (∗ 1 and W appear and W is not a singleton ∗)
if $A = W+1$ (∗ 1 and W are connected ∗)
then (∗ $\mathbf{p} = (1, p[1\,..\,C-1], 1, p[C+1\,..\,W-2], -1, -1)$ ∗)
return $(1, 0, p[1\,..\,C-1], -1, p[C+1\,..\,W-2], 0)$
else (∗ 1 and W are not connected ∗)
(∗ $\mathbf{p} = (1, p[1\,..\,A-2], -1, p[A\,..\,B-1], 1,$
$p[B+1\,..\,C-1], 1, p[C+1\,..\,W-2], -1, -1)$ ∗)
return $(1, 0, p[1\,..\,A-2], 1, p[A\,..\,B-1], -1,$
$p[B+1\,..\,C-1], -1, p[C+1\,..\,W-2], 0)$

In our program, we precompute the successors $succ_0(s)$ and $succ_1(s)$ once for each state $s = 2 \ldots M$ (the first Motzkin path is the horizontal one, which is not valid) and store them in two arrays $succ_0$ and $succ_1$.

6 Results

We report our results in Table 1. We iterate the equations until $\lambda_{\text{high}} < 1.000001\,\lambda_{\text{low}}$ (we check it every ten iterations). Already for $W = 13$, we get a better lower bound on Klarner's constant than the best previous lower bound of 3.874623 (Section 7). At $W = 16$ we beat the best previously claimed (incorrect) lower bound of 3.927378. The values of λ_{low} are truncated after six digits and the values of λ_{high} are rounded up. Thus, the entries of the table are conservative bounds.

W	Number of iterations	λ_{low}	λ_{high}
3	20	2.658967	2.658968
4	20	3.060900	3.060902
5	30	3.314099	3.314101
6	40	3.480942	3.480944
7	40	3.596053	3.596056
8	50	3.678748	3.678750
9	60	3.740219	3.740222
10	70	3.787241	3.787244
11	80	3.824085	3.824089
12	90	3.853547	3.853551
13	110	3.877518	3.877521
14	120	3.897315	3.897319
15	130	3.913878	3.913883
16	140	3.927895	3.927899
17	160	3.939877	3.939882
18	170	3.950210	3.950215
19	190	3.959194	3.959198
20	200	3.967059	3.967064
21	220	3.973992	3.973996
22	240	3.980137	3.980142

Table 1: The bounds on λ_W

So the best lower bound that we obtained is $\lambda > 3.980137$, for $W = 22$. We independently checked the results of the computation using Maple; see Appendix A. This has been done for $W \leq 20$ and led to a "certified" bound of $\lambda \geq 348080/87743 > 3.96704$.

We performed the calculations on a PC with 32 gigabytes of memory. We could not compute λ_{low} for $W = 23$ and more, since the storage requirement is too large. The number M of Motzkin paths of length $W+1$ is roughly proportional to $3^{W+1}/(W+1)^{3/2}$. We store four arrays of size M: two vectors $succ_0$ and $succ_1$ of 32-bit unsigned integers, which are computed in an initialization phase, and the old and the new versions of the eigenvector, $\mathbf{y}^{\text{new}}$ and $\mathbf{y}^{\text{old}}$, which are single-precision floating-point vectors. For $W = 23$, the number of Motzkin paths of length 24 is $M = 3{,}192{,}727{,}797 \approx 2^{31.57}$. With our current code, this would require about 48 gigabytes (5.1×10^{10} bytes) of memory.

Some obvious optimizations are possible. We do not need to store all M components of $\mathbf{y}^{\text{old}}$—only those in the first group G_1. By Proposition 1, we only need the states belonging to the group G_1 for computing $\mathbf{y}^{\text{new}}$. This does not make a large difference since G_1 is quite big. Asymptotically, G_1 accounts for 2/3 of all states. (The states not in G_1 correspond to Motzkin paths of length W.)

We can also eliminate the unreachable states, at the expense of making the ranking and unranking procedures more complicated. For $W = 23$ this would save about 11 %; asymptotically, for larger and larger n, one can prove that the unreachable states make a fraction of $4/27 \approx 15\,\%$.

The largest and smallest entries of the iteration vector $\mathbf{y}$ differ by a factor of more than 10^{11}, for the largest width W. Thus, it is not straightforward to replace the floating-point representation of these numbers by a more compact representation.

One might also try to eliminate the storage of the *succ* arrays completely, computing the required values on-the-fly, as one iterates over all states.

With these improvements and some additional programming tricks, we could try to optimize the memory requirement. Nevertheless, we do not believe that we could go beyond $W = 24$. This would not allow us to push the lower bound above the barrier of 4, even with an optimistic extrapolation of the figures from Table 1. Probably one needs to go to $W = 27$ to reach a bound larger than 4 using our approach.

The running time grows approximately by a factor of 3 when increasing W by one unit. The running time for the largest example ($W = 22$) was about 6 hours.

The code of our C program can be found on the world-wide web at

`http://www.inf.fu-berlin.de/~rote/Software/polyominos/`.

Backward Iteration versus Forward Iteration. One reason for choosing the backward iteration (7) over the forward one (3) is that it is very simple to program it, as a loop with three lines. Another reason is that this scheme should lead to a faster program because it interacts beneficially with computer hardware, for the following reasons.

The elements of the vector $\mathbf{y}^{\mathrm{new}}$ are written in sequential order, and only once. Access to the arrays $succ_0$ and $succ_1$ is read-only and purely sequential. This has a beneficial effect on memory caches and virtual memory. Non-sequential access is restricted to the one or two successor positions in the array $\mathbf{y}^{\mathrm{old}}$. There is some locality of reference here, too: adjacent Motzkin paths tend to have close 0-successors and 1-successors, in the lexicographic order. At least the access pattern conforms to the group structure of Proposition 1.

Contrast this with a forward iteration. The simplest way to program it would require the array $\mathbf{x}^{\mathrm{new}}$ to be cleared at the beginning of every iteration. It would make a loop over all states s that would typically involve statements such as

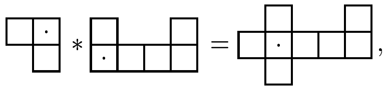
$$xnew[succ1[s]] \mathrel{+}= xold[s],$$

which involves reading an old value and rewriting a different value in a random-access pattern.

However, the above considerations are only speculations, which may depend on details of the computer architecture of the operating system and which are not substantiated by computer experiments. In fact, we tried to run our program for $W = 23$, using virtual memory, but it thrashed hopelessly, even though about half of the total memory requirement of 48 gigabytes was accessed read-only in a purely sequential manner and the other half would have fit comfortably into physical memory. If we had let the program run to completion, it would have taken about half a year.

7 Previous Lower Bounds on Klarner's Constant

The best previously published lower bounds on Klarner's constant were based on a technique of Rands and Welsh [13]. They defined an operation $a * b$ which takes two polyominoes a and b of m and n cells, respectively, and constructs a new polyomino with $m+n-1$ cells by identifying the bottommost cell in the leftmost column of b with the the topmost cell in the rightmost column of a. For example,



where we have marked the identified cells with a dot. Let us call polyomino c *$*$-indecomposable* if it cannot be written as a composition $c = a * b$ of two other polyominoes in a non-trivial way, i.e., with a and b each containing at least two cells. (In [13], this was called $*$-inconstructible.) It is clear that every polyomino c which is not $*$-indecomposable can be written as a non-trivial composition

$$c = \delta * b \tag{10}$$

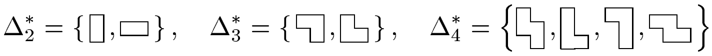
of a $*$-indecomposable polyomino δ with another polyomino b. Denoting the sets of all polyominoes and of all $*$-indecomposable polyominoes of size i by A_i and Δ_i^*, respectively, one obtains

$$A_n = (\Delta_2^* * A_{n-1}) \cup (\Delta_3^* * A_{n-2}) \cup \cdots \cup (\Delta_{n-1}^*) * A_2 \cup \Delta_n^*, \tag{11}$$

for $n \geq 2$, where we have extended the $*$ operation to *sets* of polyominoes. However, the union on the right side of (11) is not disjoint, because the decomposition in (10) is not unique: $= * = *$, and both and are $*$-indecomposable. Rands and Welsh [13] erroneously assumed that the union is disjoint and derived from this the recursion

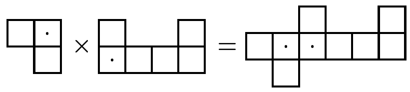
$$a_n = \delta_2^* a_{n-1} + \delta_3^* a_{n-2} + \cdots + \delta_{n-1}^* a_2 + \delta_n^* a_1 \tag{12}$$

for the respective numbers a_n and δ_n^* of polyominoes. The first few numbers are $\delta_2^* = 2$, $\delta_3^* = 2$, $\delta_4^* = 4$:


(13)

If (12) were true, this would lead to $a_1 = 1$, $a_2 = 2$, $a_3 = 6$, and $a_4 = 20$, which is too high because the true number of polyominoes with 4 cells is 19. (This is actually how the mistake was discovered in a class on algorithms for counting and enumeration taught by G. Rote, where the calculation of a_n with the help of (12) was posed as an exercise.) Even of the reader does not want to check that the list of $*$-indecomposable polyominoes in (13) is complete, one can still conclude that the value of a_4 is too high, and (12) cannot hold.

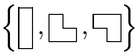
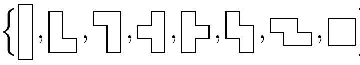
The paper [13] also mentions another composition of polyominoes, which goes back to Klarner [8]. The operation $a \times b$ for two polyominoes a and b is defined similarly as $a*b$, except that bottommost cell in the leftmost column of b is now put *adjacent* to the topmost cell in the rightmost column of a, separated by a vertical edge. The resulting polyomino has $m+n$ cells:



Now, for this operation, unique factorization holds: every polyomino c which is not $\times$-indecomposable can be written in a *unique* way as a non-trivial composition $c = \delta \times b$ of a $\times$-indecomposable polyomino δ with another polyomino b. (In this case, a *non-trivial* product $a \times b$ means that both a and b are non-empty.) Thus, one obtains the recursion

$$a_n = \delta_1 a_{n-1} + \delta_2 a_{n-2} + \cdots + \delta_{n-1} a_1 + \delta_n, \tag{14}$$

where δ_i denotes the number of $\times$-indecomposable polyominoes of size i. There are $\delta_1 = 1$, $\delta_2 = 1$, $\delta_3 = 3$, $\delta_4 = 8$, $\delta_5 = 24$ polyominoes with up to 5 cells which are indecomposable:

$\Delta_1 = \{$$\}$, $\Delta_2 = \{$$\}$, $\Delta_3 = \{$$\}$, $\Delta_4 = \{$$\}$

The idea of Rands and Welsh to derive a lower bound on the growth rate of a_n is as follows: If the values $a_1, \dots, a_N$ are known up to some size N, one can use (14) to compute $\delta_1, \dots, \delta_N$. If one replaces the unknown numbers $\delta_{N+1}, \delta_{N+2}, \dots$ by the trivial lower bound of 0, (14) turns into a recursion for numbers $\hat{a}_n$ which are a lower bound on a_n.

$$\hat{a}_n = \sum_{i=1}^{N} \delta_i \hat{a}_{n-i}, \text{ for } n > N$$

This is a linear recursion of order N with constant coefficients $\delta_1, \dots, \delta_N$, and hence its growth rate can be determined as the root of its characteristic equation

$$x^N - \delta_1 x^{N-1} - \delta_2 x^{N-2} - \cdots - \delta_{N-1} x - \delta_N = 0. \tag{15}$$

The unique positive root x is a lower bound on Klarner's constant. Applying this technique to the numbers a_i for i up to $N = 56$ [7] yields a lower bound of $\lambda \geq 3.87462343\dots$, which is however weaker than the bound $3.927379\dots$ published in [7] that would follow in an analogous way from (12).

We finally mention an easy way to strengthen this technique, although with the present knowledge about the values of a_n, it still gives much weaker bounds on Klarner's constant than our method of counting polyominoes on the twisted cylinder. One can check that any number of cells can be added above or below existing cells in a indecomposable polyomino without destroying the property of indecomposability. Thus, the number of indecomposable polyominoes increases with size. For example, a indecomposable polyomino a with n cells can be turned into a indecomposable polyomino a' with $n+1$ cells by adding the cell above the topmost cell in the rightmost column of a. Every polyomino a' can be obtained at most once in this way. It follows that $\delta_{i+1} \geq \delta_i$.

Now, if one replaces the unknown numbers $\delta_{N+1}, \delta_{N+2}, \dots$ in (14) by the lower bound δ_N instead of 0, one gets a better lower bound on a_n. The characteristic equation (15), after dividing by x^N, turns into

$$\begin{aligned} 1 &= \delta_1 x^{-1} + \delta_2 x^{-2} + \cdots + \delta_{N-1} x^{-N+1} + \delta_N x^{-N} + \delta_N x^{-N-1} + \delta_N x^{-N-2} + \cdots \\ &= \delta_1 x^{-1} + \delta_2 x^{-2} + \cdots + \delta_{N-1} x^{-N+1} + \delta_N x^{-N} \cdot \frac{1}{1 - 1/x}, \end{aligned}$$

whose root gives the stronger bound $\lambda \geq 3.87565527$.

8 Open Questions

The number of polyominoes with a fixed number of cells on a twisted cylinder increases as we enlarge the width W. This can be shown by an injective mapping as in Lemma 1. It seems obvious that the limiting growth factors behave similarly, i.e., $\lambda_{W+1} > \lambda_W$. This conjecture is substantiated by Table 1, but we do not have proof. We also do not know whether $\lim_{W \to \infty} \lambda_W \to \lambda$, although this looks like a natural assumption.

References

[1] G. Barequet and M. Moffie, The complexity of Jensen's algorithm for counting polyominoes, *Proc. 1st Workshop on Analytic Algorithmics and Combinatorics (ANALCO)*, New Orleans, ed. L. Arge, G. F. Italiano, and R. Sedgewick, SIAM, Philadelphia 2004, pp. 161–169.

[2] A. R. Conway, Enumerating 2D percolation series by the finite-lattice method: Theory *J. Physics, A: Mathematical and General*, **28** (1995), 335–349.

[3] A. R. Conway and A. J. Guttmann, On two-dimensional percolation, *J. Physics, A: Mathematical and General* **28** (1995), 891–904.

[4] S. W. Golomb, *Polyominoes*, 2nd ed., Princeton University Press, 1994.

[5] R. A. Horn and C. R. Johnson, *Matrix Analysis*, Cambridge University Press, 1985.

[6] I. Jensen, Enumerations of lattice animals and trees, *J. of Statistical Physics*, **102** (2001), 865–881.

[7] I. Jensen, Counting polyominoes: A parallel implementation for cluster computing, *Lecture Notes in Computer Science*, **2659** (2003), 203–212.

[8] D. A. Klarner, Cell growth problems, *Canad. J. Math.* **19** (1967), 851–863.

[9] D. A. Klarner, Polyominoes, *Handbook of Discrete and Computational Geometry* (J.E. Goodman and J. O'Rourke, eds.), CRC Press (1997), Ch. 12, 225–240.

[10] D. E. Knuth, Programs POLYNUM and POLYSLAVE, `http://sunburn.stanford.edu/~knuth/programs.html#polyominoes`

[11] D. L. Kreher and D. R. Stinson, *Combinatorial Algorithms, Generation, Enumeration and Search* (CAGES), CRC Press, 1998.

[12] T. Motzkin, Relations between hypersurface cross ratios, and a combinatorial formula for partitions of a polygon, for permanent preponderance, and for non-associative products, *Bull. American Mathematical Society*, **54** (1948), 352–360.

[13] B. M. I. Rands and D. J. A. Welsh, Animals, trees and renewal sequences, *IMA J. Appl. Math.* **27** (1981), 1–17; Corrigendum, **28** (1982), 107.

[14] R. P. Stanley, *Enumerative Combinatorics*, Vol. 2, Cambridge Studies in Advanced Mathematics, 1999.

A Certification of the Results

Since the bounds were calculated by a computer program requiring an unusually large memory model, and programmers and compilers cannot always be trusted, we tried to confirm the results independently, using Maple as a programming language. We did not rerun the whole computation, but we used the output of the C program described in Section 6 after the final iteration. The program writes the last iteration vector $\mathbf{y}^{(-i)}$ that it has computed into a file. The Maple program reads this vector and uses it as an estimate $\mathbf{y}^{\text{old}}$ for the Perron-Frobenius eigenvector with $\mathbf{y}^{\text{new}} = T_{\text{for}}\mathbf{y}^{\text{old}} \approx \lambda_W \mathbf{y}^{\text{old}}$. Instead of the standard backward iteration

$$\mathbf{y}_s^{\text{new}} = \mathbf{y}_{succ_0(s)}^{\text{new}} + \mathbf{y}_{succ_1(s)}^{\text{old}}$$

(see (7)), we write

$$\lambda_W \mathbf{y}_s^{\text{old}} \approx \lambda_W \mathbf{y}_{succ_0(s)}^{\text{old}} + \mathbf{y}_{succ_1(s)}^{\text{old}}.$$

Hereafter, as usual, an expression such as $\mathbf{y}_{succ_0(s)}^{\text{new}}$ or $\mathbf{y}_{succ_0(s)}^{\text{new}}$ is understood as being 0 if $succ_0(s)$ does not exist. We now find a value λ_{low} with

$$\lambda_{\text{low}} \mathbf{y}_s^{\text{old}} \le \lambda_{\text{low}} \mathbf{y}_{succ_0(s)}^{\text{old}} + \mathbf{y}_{succ_1(s)}^{\text{old}}, \tag{16}$$

for all states s. In matrix notation, this is written as $\lambda_{\text{low}}\mathbf{y}^{\text{old}} \le \lambda_{\text{low}} A\mathbf{y}^{\text{old}} + B\mathbf{y}^{\text{old}}$ or $\lambda_{\text{low}}(I-A)\mathbf{y}^{\text{old}} \le B\mathbf{y}^{\text{old}}$. We can multiply with the nonnegative matrix $(I-A)^{-1}$ and obtain

$$\lambda_{\text{low}}\mathbf{y}^{\text{old}} \le (I-A)^{-1}B\mathbf{y}^{\text{old}} = T_{\text{for}}\mathbf{y}^{\text{old}}.$$

By Lemma 9 we can now conclude that $\lambda_{\text{low}} \le \lambda_W$.

The maximum possible value of λ_{low} is simply determined by looking at every state s and solving (16) for λ_{low}. We have to take the minimum of

$$\frac{\mathbf{y}_{succ_1(s)}^{\text{old}}}{\mathbf{y}_s^{\text{old}} - \mathbf{y}_{succ_0(s)}^{\text{old}}} \tag{17}$$

over all states s. Similarly, by reversing the inequality in (16) and taking the maximum of the expressions (17), one can find an upper bound λ_{high} on λ_W. In general, these bounds turn out to be a little weaker than the bounds that are calculated from $\mathbf{y}^{\text{old}}$ and $\mathbf{y}^{\text{new}}$ by Lemma 9.

In implementing this, we tried to avoid the use of excessive memory. The C program writes the vector $\mathbf{y}$ in such a form that the Maple program simply has to scan the file sequentially. The input file for $W = 4$ is partially shown in Figure 14(a).

(a)	(b)
read "procfile.maple":	*finish*():
init(4):	*init*(4):
$setx(\{\{1\}\}, 17554423808, 1)$:	*read* "procfile.maple":
$setx(\{\{4\}\}, 5735051264, 0)$:	$setx\ := checkx$:
$setx(\{\{4\}\}, 5735051264, 1)$:	$setx(\{\{1,2,3,4\}\}, 28618452992, 0)$:
$setx(\{\{1\},\{4\}\}, 7791677952, 1)$:	$setx(\{\{1,2,3,4\}\}, 28618452992, 1)$:
$setx(\{\{3\}\}, 8280601600, 0)$:	$setx(\{\{1,2,3,4\}\}, 28618452992, 1)$:
$setx(\{\{4\}\}, 5735051264, 1)$:	$setx(\{\{1,2,3\}\}, 28618452992, 0)$:
$setx(\{\{1,4\}\}, 17554423808, 1)$:	$setx(\{\{1,2,3\}\}, 28618452992, 1)$:
$setx(\{\{3,4\}\}, 11470102528, 0)$:	$setx(\{\{1,2,3\}\}, 28618452992, 1)$:
...	$setx(\{\{1,2,3\}\}, 28618452992, 1)$:
finish():	$setx(\{\{1,2,4\}\}, 23849553920, 0)$:
terminate():	...
$setx\ := checkx$:	*terminate*():

Figure 14: (a) The check file for $W = 4$. (b) The sorted version of this file.

At the start the program reads function definitions from the file procfile.maple, which is listed in Appendix B, and performs some initializations in the procedure *init*. The main work is done in the procedure calls $setx(s, y, flag)$, which indicates that the component for state s in the vector $\mathbf{y}^{\text{old}}$ equals y. A state is represented by its signature, as a set of sets. A *flag* value of 1 indicates that the program should just remember this value. A *flag* of 0 in a procedure call with state s indicates that the program should use this value and the previously-stored values of $\mathbf{y}^{\text{old}}$ to evaluate (17) for this state, and update λ_{low} and λ_{high} if appropriate. At the end, the procedure *finish* prints out the final values of λ_{low} and λ_{high}.

The C program writes the values for $succ_0(s)$ (if it exists) and $succ_1(s)$, with a flag of 1, immediately before writing a line for s with a flag of 0. After processing a line with a flag of 0, Maple can, therefore, forget all values that it has stored. Accordingly, as can be seen in the example of Figure 14(a), some states occur several times.

The Maple program calculates $succ_0(s)$ and $succ_1(s)$ on its own, and it generates an error message if the required values were not stored in the preceding calls to *setx*.

We also performed a slightly more paranoid check to ensure that no state was omitted, and the program did not inadvertently use two different values for the same state. More precisely, we checked that all states that are present in the file with a flag of 1 are also present in the file with a flag of 0, with identical values y. The format of the input file is designed in such a way that one just has to sort the lines alphabetically and read the sorted file into Maple to carry out this check. Figure 14(b) shows the sorted version of the file of Figure 14(a)[1]. All calls to *setx* that refer to the same state are now grouped together. There must first be a call with flag 0, where the value is memorized, followed by an arbitrary number of calls with the same state and with flag 1, where the program just checks if the given values coincide. The meaning of the procedure *setx* is changed at the beginning by the assignment *setx* := *checkx*. Note that *finish* and *init* appear before procfile.maple is read; thus, the first two lines have no effect.

In the first phase we have already checked that all successors of all states, which are present in the file with a flag of 0, are also present in the file, with a flag of 1. As a consequence, it is ensured that at least all reachable states have been processed in the first phase (provided that at least one state was processed). This is enough to establish correctness of the result.[2]

For $W = 20$ the size of the check file was about 30 gigabytes. Each pass over the file with Maple took about 20 hours, and sorting took almost five hours. (We mention these running times only to give a rough indication. We ran our program on different computers of different speeds.) It would also be feasible to check larger values of W but we did not think it was worthwhile. The procedures *finish* and *terminate* printed the following output, after reading the unsorted check file.

```
348080  10836799
------, --------, [3.967040106, 3.967082162], 3967040105*10^-9,
87743   2731680

    3967082162*10^-9

  142547558 configurations were checked.
```

[1]We leave the question of why the first two states in alphabetic order, $\{1,2,3,4\}$ and $\{1,2,3\}$, have the same value, to the reader to ponder.

[2]We did not check if a state appears more than once with a flag of 0, possibly even with different values. It is not difficult to work out why this does not harm the reliability of the result.

B The Maple Program

This is the contents of the file procfile.maple, which contains the Maple procedure definitions and the initialization of two variables *nerrors* and *nchecks*.

```
check := proc(state)
local setcontaininglast, setcontainingfirst, rest,
        combinefirstandlast, succ0, succ1, ratio;
global n, x, maxratio, minratio, nerrors, nchecks;
  nchecks := nchecks + 1;
  setcontaininglast, rest := selectremove(has, state, n);
    # split state into the part containing n (if it exists) and the rest
  if setcontaininglast = {{n}} then succ0 := 0
  elif setcontaininglast = {} then succ0 := map(shift1, state)
  else succ0 := map(shift1, rest union {setcontaininglast[1] minus {n}})
  end if;
  setcontainingfirst, rest := selectremove(has, rest, 1);
  combinefirstandlast :=
    map(op, setcontaininglast) union map(op, setcontainingfirst);
  succ1 := map(shift1, rest union {(combinefirstandlast union {0}) minus {n}});
  if not [state] in [indices(x)] then
    nerrors := nerrors + 1; error "Value %1 not initialized.", state
  end if;
  if not [succ1] in [indices(x)] then
    nerrors := nerrors + 1;
    error "Value %1 (succ1) not  initialized for %2.", succ1, state
  end if;
  if succ0 <> 0 and not [succ0] in [indices(x)] then
    nerrors := nerrors + 1;
    error "Value %1 (succ0) not  initialized for %2.", succ0, state
  end if;
  if succ0 = 0 then   # xnew[state] := xold[succ1]
    ratio := x[succ1]/x[state]
  else      # xnew[state] := xold[succ1] + xnew[succ0]
    ratio := x[succ1]/(x[state] - x[succ0])
  end if;
  minratio := min(minratio, ratio);
  maxratio := max(maxratio, ratio)
end proc;

shift1 := proc(part) map(x -> x + 1, part) end proc;

setx := proc(state, value, flag)
global x;
  x[state] := value; if flag = 0 then check(state); x := table() end if
end proc;
```

```
checkx := proc(state, value, flag)
global rememberstate, remembervalue, nerrors, nchecks;
   if flag = 0 then
      nchecks := nchecks + 1 ; rememberstate := state ; remembervalue := value
   else
      if rememberstate ≠ state then
         nerrors := nerrors + 1 ;
         error "incorrect state %1,  should be %2", state, rememberstate
      end if;
      if remembervalue ≠ value then
         nerrors := nerrors + 1 ;
         error "incorrect value %2 for state %1, should  be %3",
               state, value, remembervalue
      end if
   end if
end proc ;

init := proc(w)
global n, x, maxratio, minratio, nerrors, nchecks;
   n := w ; x := table() ; minratio := ∞ ; maxratio := 0 ; nerrors := −1 ; nchecks := 0
end proc ;

finish := proc()
local scale;
global minratio, maxratio, min1, max1, nchecks;
   scale := 10^9 ;
   min1 := floor(minratio * scale) ;
   max1 := ceil(maxratio * scale) ;
   print(minratio, maxratio, evalf([minratio, maxratio]), cat(min1, "*10^-9"),
      cat(max1, "*10^-9"))
end proc ;

terminate := proc()
global nerrors, nchecks;
   printf("  %d configurations were  checked.\n", nchecks);
   if nerrors = 0 then printf("  OK.\n")
   elif 0 < nerrors  then printf("There were %d errors.\n", nerrors)
   end if
end proc ;

nerrors := 0;    # initialisation
nchecks := 0;
```