Orientation on mathematical research

Self-avoiding walks on the honeycomb lattice

By
Jannes Vleming (6054501)
Paul Fahner (6138551)
Daniël Brus (5802938)

Supervisor
Prof.Dr. R.H. Bisseling

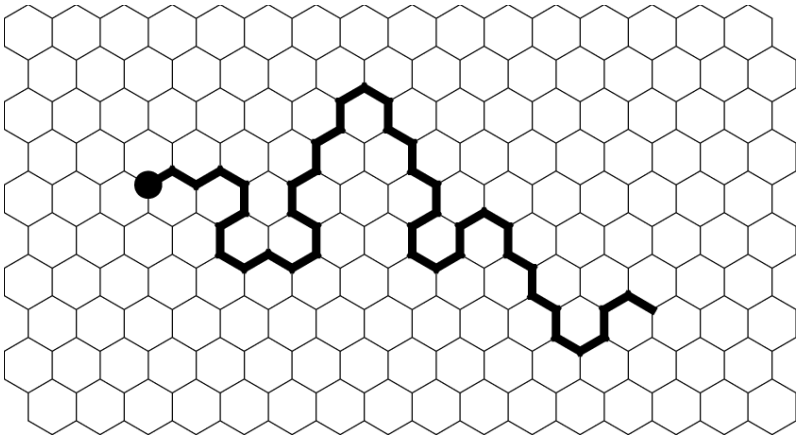


Figure 1: A self-avoiding walk on a honeycomb lattice[1].

April 29, 2023

Universiteit Utrecht

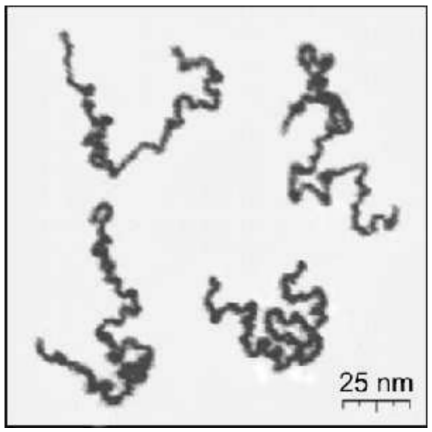
Abstract

In this report, we implement Zbarsky's method for enumerating self-avoiding walks (walks on a lattice that do not revisit sites) on the honeycomb lattice. We compare Zbarsky's method with our implementation of Jensen's method, the record-holding transfer matrix method, on which it is based. Zbarsky's method works by using the inclusion-exclusion formula on rows with more than a certain number of crossings, restricting the number of signatures.
In theory, Zbarsky's method has subexponential computation time, and indeed our implementation appears to confirm this. For the walk lengths that we reached, Zbarsky's method is still significantly slower than Jensen's method. We also used function fitting to predict from which length onward Zbarsky's method will be faster, which we think is very roughly $n = 158$ for the honeycomb lattice. This is a lot further than the current world record of 105, but not completely unreachable in the (near) future.

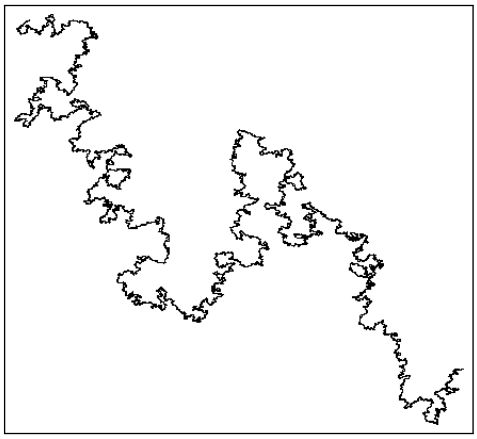
Contents

1 Introduction

A self-avoiding walk (SAW) is a path on a lattice that does not revisit lattice points. Chemist Paul Flory introduced them as a simple method to study polymers [2]. Polymers are very long molecules built from repeating subunits. Given the chemical complexity of a single polymer, it might seem surprising that such a simple model would give any usable results, but this is actually quite usual in statistical mechanics. Here a common theme is that a small system is scaled up, the details of this system do not matter anymore and can be summarized in a few important constants. Compare this to the central limit theorem, where we sum up infinitely many independent identically distributed random variables. While the exact distribution of such a random variable can have a very detailed and complicated shape, only its mean and variance matter in the limit.



(a) Picture of 4 linear polymer chains, obtained using an atomic force microscope.



(b) Example of a long self-avoiding walk.

Figure 2: Self-avoiding walks were introduced as a model for linear polymer chains because the microscopic behaviour of these long behaviour resembles the structure of a self-avoiding walk.[3]

It might seem like a great theoretical improvement that we can model polymers using such a simple model as the SAW, but it turns out that the mathematics are again very complicated. Basic questions, like how many SAWs there are of a given length and how the average distance to its endpoint grows with its length, seem completely unanswerable in general. Even asymptotic behaviour is very challenging. As a model, the SAW is still easy to simulate. Counting the number of SAWs, c_n, of a given length n is one of the main problems, and can be used to estimate parameters of asymptotic growth. Over the years, this counting problem has become a benchmark for algorithm design and computing power. In this project, we will mostly look at the 2-D case.

It is proven that c_n grows exponentially. The most elementary counting algorithm, called backtracking, will also have an exponential time complexity. While initially fast because of simplicity, this method thus quickly becomes infeasible for larger n. A breakthrough was the advent of the transfer matrix method for finite lattices due to Enting in 1980 [4]. These ideas were then subsequently improved upon.

We split a rectangle into two parts and note how the SAW crosses from one half to another in a signature, which is a set of labelled crossings. Then we can move this boundary using simple rules ignoring the precise details of the path on either side. For the 2-D square grid, the record is, to our knowledge, held by Jensen [5] who in 2013 counted c_n for n up to 79. These transfer ma-

trix methods still have exponential computation time, but simply with a (much) lower growth rate.

In 2019, Zbarsky came up with a theoretical improvement [6], modifying previous algorithms to give them subexponential computation time. When counting SAWs in a rectangle this method uses the inclusion-exclusion formula on columns that have at most a certain number of horizontal edges. Effectively this bounds the number of signatures that are calculated, at the cost of having to count every rectangle many times. It remains to see whether this method will ever be faster than existing methods in practice. To help answer this question, we implemented this method and compared it with our implementation of Jensen's current best method. We do this on the honeycomb lattice, since this is more sparse and hopefully allows us to penetrate regions of higher n where the subexponential behaviour of the algorithm is more prominent. Our implementation also works for the square grid, and the related problem of counting self-avoiding polygons, which are SAWs that end where they start.

2 Theory

In this section, we will define what we mean by a self-avoiding walk on a lattice. We will also introduce the square and honeycomb lattice. We make an observation about the exponential growth rate of the number of SAWs. We will introduce a simple algorithm to count self-avoiding walks on a lattice, but due to this exponential rate, it is hard to count long self-avoiding walks directly using this method.

2.1 Counting self-avoiding walks on a lattice

A graph $G = (V, E)$ consists of a set of vertices V (sometimes called sites) and edges $E \subseteq V^2$. We say two vertices v_1, v_2 are connected, we write $v_1 \sim v_2$, if $(v_1, v_2) \in E$ is an edge. A lattice is a certain kind of very regular infinite graph. A self-avoiding walk is an ordered finite set $(v_0, \ldots, v_n)$ such that subsequent sites are connected $v_i \sim v_{i+1}$, and each site is visited at most once. This particular SAW starts in v_0, ends in v_n and has length n. Once we fix an origin O, the number of SAWs of length n called c_n is the number of distinct SAWs starting in O on this particular lattice. Lattices are regular in the sense that the choice of O does not affect c_n, so we don't have to think about the choice of O in the remainder of this report. This leaves us with the main question of this report: What is c_n for a given n, and what is the fastest way to compute c_n?

Before we actually start with the problem of counting, we introduce the lattices that we will work with in this report: An important example of a lattice is the $d-$dimensional square lattice. Here, let $V = \mathbb{Z}^d$, the $d-$tuples of integers, and for the edges we have

$$E = \left\{ (v, w) \in V^2 \,\middle|\, ||v - w|| = 1 \right\}$$

with $||\cdot||$ the standard Euclidean distance. In other words vertices (v, w) in V^2 are connected if they are nearest neighbours. For $d = 2$ this is called the square lattice. Another lattice in two dimensions is the honeycomb lattice, which is seen in figure 3 on the left. The sites are the points where three edges intersect. As can be seen in the middle figure, this lattice can be transformed into a sublattice of the square lattice, with the same vertex set but fewer edges. Since a lot of counting techniques are known for the square lattice, we can also apply these to the honeycomb lattice with minor modifications.

2.2 The connective constant

Next, we make an observation about the rate of growth of c_n, adapted from [7][section 1.2]. Let W_n denote the set of all SAW of length n starting in $v_0 = O$. Note that if

$$(v_0, \dots, v_{n+m})$$

is a SAW of length $n+m$, then

$$(v_0, \dots, v_n), (v_n, \dots, v_{n+m})$$

are SAWs of length n and m respectively. We can shift the second walk such that it starts in v_0 because of the symmetries of the lattice. This gives us a natural injective function from W_{n+m} to $W_n \times W_m$. As a consequence, we have the inequality

$$c_{n+m} = |W_{n+m}| \leq |W_n \times W_m| = c_n \cdot c_m.$$

If we take the logarithm on both sides we get

$$\log c_{n+m} \leq \log c_n + \log c_m.$$

This is known as the subadditive property (that is, the sequence $(\log c_n)_{n\geq 1}$ is called subadditive). We have the following nice result for subadditive sequences, sometimes known as Fekete's lemma.

Lemma 2.1. *Let $(a_n)_{n\geq 1}$ be a subadditive sequence, then*

$$\lim_{n\to\infty} \frac{a_n}{n} = \inf_{n\geq 1} \frac{a_n}{n} \in [-\infty, \infty).$$

Proof. Let $k \geq 1$ and define $A_k := \max_{1\leq r\leq k} a_r$. For $n \geq 1$, we write $n = jk + r$ such that $j \leq n/k$ and $r < k$. Using subadditivity we have

$$a_n = a_{jk+r} \leq ja_k + a_r \leq \frac{n}{k} a_k + A_k.$$

Dividing both sides by n and taking the $\limsup_{n\to\infty}$ gives us

$$\limsup_{n\to\infty} \frac{a_n}{n} \leq \frac{a_k}{k} + \limsup_{n\to\infty} \frac{A_k}{n} = \frac{a_k}{k},$$

which holds for every k. Taking $\liminf_{k\to\infty}$ in this equality yields

$$\limsup_{n\to\infty} \frac{a_n}{n} \leq \liminf_{k\to\infty} \frac{a_k}{k},$$

which proves the existence of the limit. Taking an $\inf_{k\to\infty}$ instead of $\liminf_{k\to\infty}$ gives

$$\lim_{n\to\infty} \frac{a_n}{n} = \limsup_{n\to\infty} \frac{a_n}{n} \leq \inf_{k\geq 1} \frac{a_k}{k}.$$

Since the other inequality is obvious we conclude

$$\lim_{n\to\infty} \frac{a_n}{n} = \inf_{k\geq 1} \frac{a_k}{k}.$$

□

Since $\log c_n$ is subadditive and clearly $\inf_{n\geq 1}(\log c_n)/n \geq 0$, the sequence $(\log c_n)/n$ must converge, we will call its limit $\log(\mu)$ for some $\mu > 0$, thus we have

$$\lim_{n\to\infty} \frac{\log c_n}{n} = \log \mu \qquad \text{and therefore} \qquad \lim_{n\to\infty} c_n^{\frac{1}{n}} = \mu. \tag{1}$$

We write

$$c_n \sim \mu^n.$$

With this notation we do not mean that the ratio of the left and right hand side converges to 1, we mean exactly 1. This constant μ is called the connective constant and depends on the lattice. For the relevant lattices, we have $\mu > 1$, which implies that there is an exponential growth of the number of SAWs as a function of n.

We can give a lower bound for μ of the honeycomb lattice as follows: The first step and all subsequent odd-numbered steps can be chosen in the same direction. For the even-numbered steps, we always have two choices. This procedure is guaranteed to give us a set of SAWs that grows with $\sqrt{2}$ every step on average. In this way, we see that $\mu \geq \sqrt{2}$.

We can also give an upper bound for μ of the honeycomb lattice. At each site in the lattice, there are exactly 3 edges connected to this site. If a path has reached this site, not on the first step, one of these edges is already occupied with the path. Since we look for SAWs, the walk cannot go back, and thus at each step the number of SAWs at most doubles, meaning $\mu \leq 2$.

For the honeycomb lattice recently the exact value for μ has been proven [8], namely

$$\mu_{honey} = \sqrt{2 + \sqrt{2}} \approx 1.85.$$

However, μ is still unknown for most lattices. The best we can do usually is give lower and upper bounds, basically more advanced versions of what we did above, or estimate μ with simulations. For the square lattice, it has been found that approximately

$$\mu_{square} \approx 2.64.$$

2.3 Backtracking algorithm

The next best thing after counting c_n by hand is letting a computer do it for you. This is essentially what a backtracking algorithm does. In literature, it is sometimes referred to as the naive algorithm. A backtracking algorithm works recursively: We start at the origin. Every time we arrive at a site, we go to each of its unoccupied neighbours consecutively. If we arrive at a site and the length of the path up until then equals n, the length of the SAW we want to count, we count one walk and do not visit the site's neighbours. Effectively we try out every direction at every point.

To calculate the time complexity of this algorithm, we count how many times the recursive function is called, that is, how many times we arrive at a site. Let n be the length of SAWs we are counting. We group these site arrivals according to how long the walk is up until then, call these lengths k where k ranges between 0 and n. When the walk arrives at a site after avoiding itself for k steps, it is a SAW of length k, thus there are c_k function calls at current length k. Adding up the calls for all k gives

$$\sum_{k=0}^{n} c_k \approx \sum_{k=0}^{n} \mu^k = \frac{\mu^{n+1} - 1}{\mu - 1} = \frac{\mu^{n+1} - 1}{\mu^n(\mu - 1)} \mu^n$$

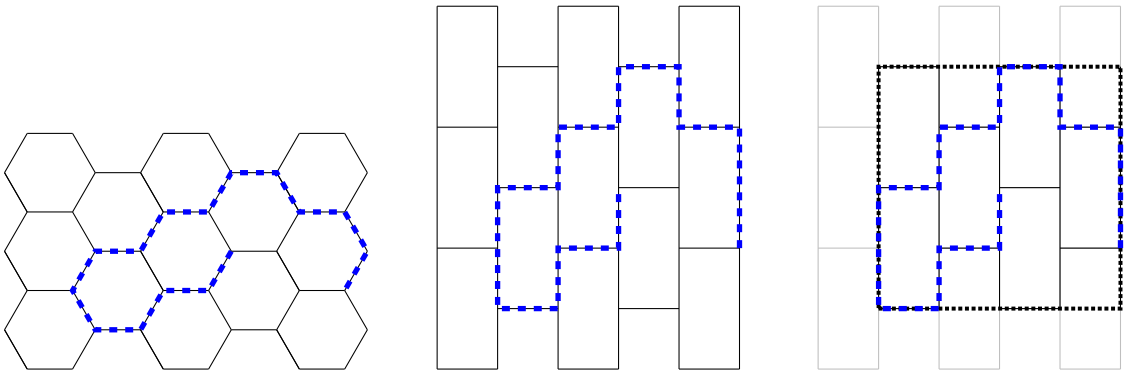
where in the last expression we have μ^n and something bounded, we say the time complexity is $\mathcal{O}(\mu^n)$. This immediately shows that backtracking will not get us very far, especially if μ is big, since every next step will take μ times as long as the previous. Fortunately, there exist more efficient methods of counting SAWs, which we will explore in the next section.

3 Method

Although the backtracking algorithm is easy to understand and implement, its computational complexity of $\mathcal{O}(\mu^n)$ prevents the algorithm from computing c_n for large n within a reasonable amount of time. A more efficient way is to use a so-called *transfer matrix method* as described by Jensen [5], who used it to count SAWs on a regular square grid. To use a transfer matrix method on our honeycomb lattice, we need to transform the grid into a subgraph of a square grid. The transfer matrix method divides the problem of counting SAWs on the infinite grid into subproblems of counting SAWs in finite rectangles. We will also describe and implement the modification of this transfer matrix method that Zbarsky suggested in 2019[6] to make the transfer matrix algorithm even faster. We will describe the important concepts of this subproblem and how to solve it in this section, and we will briefly describe how pruning can be implemented in the transfer matrix method.

3.1 Infinite lattice to rectangles

Transfer matrix methods are generally used to count SAWs on square grids, so we have to transform the hexagonal honeycomb grid into (a subgraph of) the square grid: Given a SAW on the honeycomb grid (figure 3a), we can transform the SAW together with the transformation from the honeycomb to the *brick grid*, and this infinite vertical brick grid in figure 3b is equivalent to the honeycomb grid, which gives an equivalence between SAWs on the honeycomb grid and the vertical brick grid.



(a) A SAW of length $n = 15$ on the honeycomb grid.

(b) The same SAW after transforming the honeycomb grid into the brick grid.

(c) There exists a unique smallest rectangle that contains the transformed SAW, in this case with dimensions $(W, H) = (5, 4)$

Figure 3: Illustration of the process of identifying a SAW on the honeycomb grid with a unique smallest rectangle that contains the transformed SAW.

Transfer matrix methods count SAWs on such an infinite square grid by dividing the counting problem into smaller subproblems of counting SAWs on a finite rectangular grid of width W and height H: As illustrated in figure 3c, the transformed SAW can be associated with a unique smallest rectangle of width W and height H that *exactly contains* the SAW (meaning that they touch the top, bottom, left side and right side of the rectangle). Finite lattice methods count all the SAWs of length n that are exactly contained in the rectangles with dimensions (W, H) for $W = 0, 1, 2, \ldots$ and $H = 0, 1, 2, \ldots$. The idea is that by doing so, all SAWs of length n on the original honeycomb grid will have their transformed counterpart counted in the unique smallest rectangle in which it is exactly contained, meaning that the total number of counted SAWs of length n over *all possible rectangles* will be equal to c_n, the number of SAWs on the original

honeycomb grid.

(a) A rectangle with even parity ($p = 0$)

(b) A rectangle with odd parity ($p = 1$)

Figure 4: Two SAWs of length $n = 16$ that fit in different types of rectangles with $(W, H) = (5, 4)$ on the brick grid. This is distinction is described with the parity p of the rectangles, which depends on the parity of the vertex in its upper left corner.

We have to be careful: In our case, there are two possible rectangles of a given dimension (W, H), which is illustrated in figure 4. We use the following convention to distinguish the two ways that the brick grid can be placed in such a rectangle, which we refer to as the *parity* $p \in \{0, 1\}$: A vertex v of the brick grid is called *even* ($p_v = 0$) if it has an edge to its neighbour on the right (and therefore not to the left), and it is called *odd* ($p_v = 1$) if it has an edge to its neighbour on the left. The parity p of a rectangle corresponds to the parity of the upper left vertex in the rectangle. In general, these two versions are not mirrored copies of each other, and we have to count both separately.

Every transformed SAW that is exactly contained in a rectangle with dimensions (W, H) will either fit in the rectangle with parity $p = 0$ or $p = 1$. This means that we have to consider rectangles of all possible $W, H \geq 0$ and $p \in \{0, 1\}$. However, we do not have to consider infinitely-sized rectangles when counting the SAWs of finite length n, because we impose that the SAWs are exactly contained in the rectangle and therefore they have to touch all four sides of the rectangle. The largest horizontal distance that a SAW of length n can cover on the brick grid is $\lceil n/2 \rceil$, and the largest possible vertical distance that is left to cover after W horizontal steps is $n - W$.

We define $C^n_{W,H,p}$ to be the number of SAWs in a rectangle on the brick grid with width, height and parity (W, H, p), and take the sum of this quantity over all possible W, H and p to obtain the total number of possible SAWs of length n on the infinite brick grid (i.e. the transformed honeycomb lattice, which is equivalent to the original honeycomb lattice):

$$c_n = \sum_{W=0}^{\lceil n/2 \rceil} \sum_{H=0}^{n-W} \sum_{p \in \{0,1\}} C^n_{W,H,p} \tag{2}$$

This equation relates the solutions $C^n_{W,H,p}$ to our subproblems to the solution c_n of the original problem, and in the remainder of this section we will explore the methods we use to solve the subproblem of counting SAWs in these rectangles.

There is one remark to be made about equation (2): It will turn out to be more efficient to work with wide rectangles than with high rectangles from a computational perspective. This is because the cutline, which we will soon introduce, is shorter thus reducing the number of signatures. Motivated by this, we will rotate rectangles with $H > W$ by 90°, meaning that we create a horizontal brick grid instead of a vertical one, giving two more ways to place a brick grid in a rectangle. These two new ways can be described by two 'new parities' $p = 2, 3$ for rectangles with $H < W$ that we sum over in equation (2) instead of summing over rectangles with $H > W$. These

rectangles can be treated quite similarly to those with a vertical pattern, thus for simplicity in this report we will only mention the latter ones from now on.

3.2 Cutline, signatures and generating functions

In this section, we will introduce three new concepts. These may seem to come out of the blue, but the motivation is as follows: Instead of keeping track of every individual SAW (as the backtracking algorithm does), we group SAWs based on a *signature* and we try to count these groups of SAWs all at once, which is done using *generating functions*. In the end, this should result in an algorithm with lower time complexity than the backtracking algorithm for large numbers of SAWs.

To perform the transfer matrix method on a rectangle, we introduce the *cutline* that separates the rectangle into two subsets A and A^c of the finite rectangular lattice such that every grid point is either in A or A^c, as illustrated in figure 5. We want to stress that A is a set of vertices. This cutline is also referred to as the *boundary* between the left region A and the right region A^c. The SAWs in the rectangle can cross this cutline, and the transfer matrix method's idea is to move this cutline from left to right and keep track of the crossings to count the SAWs in the meantime. To explain this in detail, we need to introduce how we keep track of the crossings of SAWs with the cutline. After this, we can describe how we move the cutline from the left to the right while keeping track of how SAWs cross the cutline.

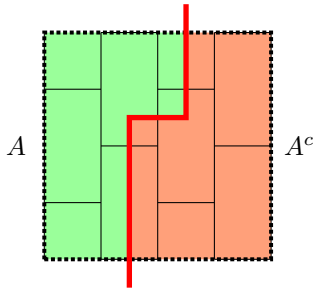


Figure 5: Dividing a rectangle into regions A and A^c with the red cutline.

A valid SAW in a rectangle touches both the left and right side of the rectangle, meaning that it crosses the cutline at least once (unless the boundary is completely to the left or right of the rectangle as we will see later). To describe how the SAWs cross the cutline, we introduce a *signature* S that contains labels that describe every possible edge of the graph (the brick grid in our case) that the cutline crosses. The labels of these *possible crossings* of the cutline are ordered from the edge at the bottom of the rectangle to the top. The possible crossings are indicated in figure 6 as horizontal red edges. Notice that we also consider the non-existing edges of our subgraph of the square lattice.

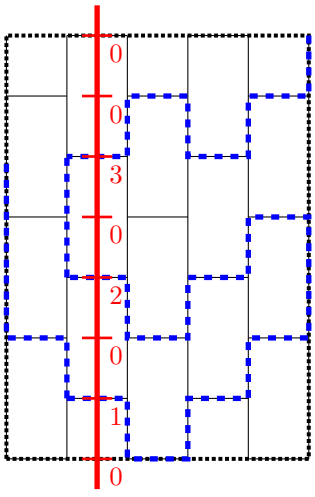


Figure 6: Example of a SAW crossing the cutline. The signature of the cutline for this SAW is $S = \{01020300, 1000\}$.

The possible crossings of a cutline receive labels σ_i based on how the SAW is connected in the right region, meaning that the length of the cutline corresponds to the number of labels σ_i in a signature S. If the SAW does not cross the cutline in a possible crossing, that edge is called an *empty edge*. If the SAW crosses the cutline, we make a distinction between three options based on how the SAW evolves on the right side of the crossing with the cutline, i.e. in the region A_c: If the SAW continues on the right until it reaches an endpoint of that 'free' part of the SAW, it is called a *free edge*. This is the case for the highest non-empty crossing of the cutline in figure 6. On the other hand, if the SAW comes back to the cutline from the right region, to connect with another crossing, these two crossings create a connected segment of the SAW together (which we refer to as a *loop*), and they are called a *lower edge* and *upper edge*, depending on which of the crossings of the cutline lies lower and which lies higher.

The lowest two crossings of SAW with the cutline in figure 6 form such a pair of a lower edge and upper edge.

With these four types of edges (possible crossings) of the cutline defined, the labels σ_i of the edges are defined as follows:

$$\sigma_i = \begin{cases} 0 & \text{if it is an empty edge} \\ 1 & \text{if it is a lower edge} \\ 2 & \text{if it is an upper edge} \\ 3 & \text{if it is a free edge} \end{cases} \tag{3}$$

Definition 3.1. *A signature $S = \{\sigma, ltbr\}$ is a tuple that describes how a SAW behaves in a rectangle with respect to the cutline. It contains labels σ_i of the edges that the cutline crosses, and it contains four truth values $l, t, b, r \in \{0, 1\}$ that describe whether the SAW touches a side of the rectangle (1) or not (0) to the left of the cutline.*

Because we only want to count the SAWs that touch every side of the rectangle, we also have to keep track of whether or not the SAWs of a signature S have touched each side of the rectangle in region A (to the left of the cutline). This adds four extra binary labels l, t, b, r (1/True or 0/False) to a signature for the left side, top, bottom and right side of the rectangle respectively. In the beginning (figure 7a), these will all be 0, and at the end (figure 7b), only the signatures with all labels l, t, b, r being 1 describe valid SAWs, so only this signature contributes to the number of valid SAWs of length n in the rectangle.

With this definition of the labels, we conclude that the signature corresponding to the cutline in figure 6 is $S = \{01020300, 1000\}$ for example. Notice that we only consider whether or not a SAW has touched the four sides of the rectangle in region A. We also immediately find some restrictions on valid signatures:

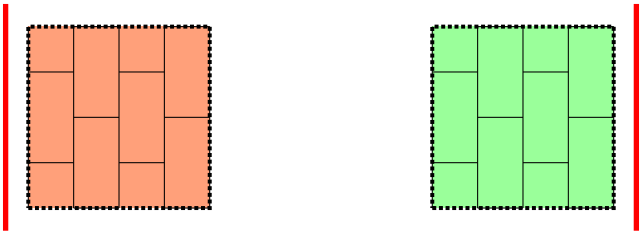
- A 3 can appear at most twice in a signature S because a SAW cannot have more than two starting points/endpoints.
- Every 1 in a signature is associated with a specific 2 by definition (because parts of a SAW cannot intersect), and vice versa. This means that the number of 1's and 2's has to be equal.
- More precisely: For every 1 in a signature, the associated 2 has to come later in the signature because the labels are given to the possible crossings from the bottom of the cutline to the top.

Before we continue, we should make a remark on how we determine the labels of the edges of the cutline: We choose to consider how the SAW connects in the right region of the cutline (A^c), but the transfer matrix method also works if you consider how the SAW connects in the left region A. Although updating the signatures when moving the boundary from the left to the right is less complicated when you consider A to determine the labels, we choose to consider A^c because it will allow for more pruning opportunities, as we will see later in this section.

A signature does not contain all the information of a SAW, but it only describes how it crosses the cutline. This means that multiple SAWs might share the same signature S, and this is the motivation for defining a *generating function* $G_S(x)$ for a signature, which is more efficient than keeping track of individual SAWs. The generating function of a signature S is defined as a polynomial:

$$G_S(x) = \sum_{k=0}^{\infty} s_k x^k \tag{4}$$

The coefficients $s_k \in \mathbb{Z}_{\geq 0}$ describe how many ways there are for a SAW with signature S to have k edges to the left of the cutline (in A), excluding the crossings with the cutline. For example, the SAW in figure 6 contributes to coefficient s_7 because the SAW has 7 edges to the left of the cutline. When moving the cutline from the left of the rectangle to the right, we will update the signatures and update the corresponding generating functions accordingly. We will go into the details of the process of moving the cutline in the next subsection, but the following logic already explains how this process is used to count SAWs in the rectangle:



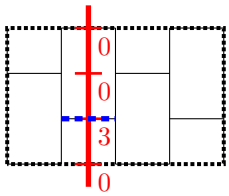
(a) Starting position (A (the green part) is empty)

(b) Final position (A (the green part) covers the entire rectangle)

Figure 7: The position of the cutline before and after the process of moving the cutline from the left of the rectangle to the right. Where we keep track of the random walks of a certain length in the green area.

At the beginning of the process (figure 7a), the region A is empty and the only signature will be $S = \{0 \dots 0,\ 0000\}$, with a corresponding generating function $G_S(x) = 1$ because there is only one option: 0 steps of the SAW in region A. At the end of the process (figure 7b), the region A covers the entire rectangle and the only valid signature $S = \{0 \dots 0,\ 1111\}$, meaning that the coefficients s_k of the generating functions $G_S(x)$ of this signature give the number of SAWs of length k in the *entire rectangle*, provided that the signatures and corresponding generating functions were updated correctly.

This already gives us a simple pruning method before we have described how the transfer matrix method moves the cutline: When counting the SAWs of length n in a rectangle in this way, you only have to consider the coefficients $s_0, \dots, s_n$ of the generating polynomials at every step, because all the coefficients s_{n+1} and higher count SAWs that are already too long.



(a) Illustration of the cutline and the signature. The only crossing of the SAW is a free edge.

Truth values *ltbr*	Generating function $G_S(x)$
1000	$x^3 + x^2$
1100	x^3
1010	$2x^5 + 2x^4 + x^3 + x^2$
1110	$x^6 + x^5$

(b) Generating functions of the signature. These functions describe how many steps of how many SAWs lie on the left of the cutline such that the given edges of the rectangle are touched.

Figure 8: Example of signatures $S = \{0300,\ ltbr\}$ for different values of t and b in a rectangle, with the corresponding generating functions. At this point, the evolution of these SAWs to the right of the cutline is undetermined, and the evolution to the left is described by $G_S(x)$.

To illustrate how generating functions of signatures work, we consider the example in figure 8a. The generating functions of the variants of this signature can be determined by counting the possible SAWs manually that touch the desired sides without self-intersecting. In the next section, we will describe how these generating functions can be obtained algorithmically instead of computing them manually.

3.3 Moving the cutline

Now that signatures and their generating functions are defined using the cutline, we have to describe how we update the signatures and generating functions during the process of moving the cutline from the left of the rectangle to the right (see figure 7). We introduce the transfer matrix method that Jensen described[5] in this section. Jensen's method moves the cutline to the right column by column, until the entire rectangle is to the left of the cutline. We will also refer to this as *updating* the columns one by one. To update a column, the vertices of the underlying grid are updated (i.e. added to the region A) one by one, starting at the top of the rectangle as illustrated in figure 9.

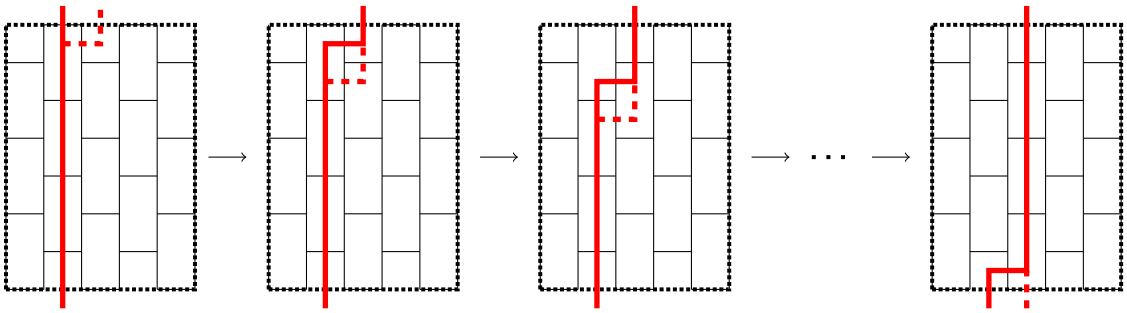


Figure 9: Example of how we move the cutline one column to the right. With Jensen's method, the columns are updated one by one by this procedure. At every step, one lattice point is added to the region on the left of the cutline, as indicated by the dashed line that shows what the cutline will be after updating the next vertex.

The process of correctly updating the signatures S and the corresponding generating functions G_S when updating one vertex can be formulated as a set of *update rules* that describe what signatures can originate from the current signature after the cutline is moved one square. These update rules depend mainly on the edges of the cutline around the particular vertex that is being updated, which are indicated as b and d in figure 10. Most often, there are several possible signatures that can originate from a signature, these are called *target signatures*. The goal of these update rules is to find all target signatures S' that can originate from the original signature S, and then to adjust the original generating function G_S to contribute to the new generating function $G'_{S'}$ of the target signature.

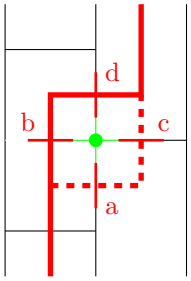


Figure 10: The update rules of the green vertex mainly depend on the labels of edges b and d of the original signature, and describe what the possible labels of the new edges a and c are for the target signatures.

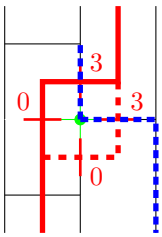
There are a few exceptions, but these target signatures of the moved cutline are in general identical to the original signature,

apart from the labels corresponding to the 'new edges' of the cutline, which are indicated as a and c in figure 10.

It should be noted that the update rules for vertices depend on the parity p_v of the vertex v that is being updated, because the parity of a vertex gives restrictions on the labels of both the original and target signatures: If v is even (as in figure 10), edge b is empty by definition. Similarly, edge c should be an empty edge for the target signatures if v is an odd vertex. Restrictions on the labels of edges a,b,c and d arise in a similar manner if the updated vertex v lies at the boundary of a rectangle. For example, if v lies at the bottom of the rectangle, the label of edge a in the target signatures has to be empty.

The most straightforward part of the update rules is the procedure of updating the truth values l, t, b and r of a signature $S = \{\sigma, ltbr\}$. If edges b and/or d, are nonempty and touch a side of the rectangle, the corresponding truth value should be updated to 1 if it was not 1 before to indicate that the SAWs of this signature have touched this side of the rectangle on the left of the moving cutline. The update rules for the labels σ of a signature S are more complicated, although they often only depend on edges b and d, which change into edges a and c (see figure 10). This implies that an update rule can be formulated as a mapping (b,d)↦(a,c), with often some strings attached and multiple possible outcomes that all have to be considered. The update rules are described in appendix A for completeness, but we will describe the underlying principles in this section. Two examples of possible target signatures are shown in figure 11.

If d is a free edge (3) and b is empty (0), this SAW can evolve in three ways to the right of the cutline: The SAW can end in the vertex, move down or move to the right. If it moves to the right, the resulting labels are (a,c)=(0,3), as shown here. If the SAW ends at this vertex, the resulting labels are (a,c)=(0,0) and if the SAW moves down from this vertex, the labels are (a,c)=(3,0).



If both b and d are empty edges (0), there are many possibilities for the labels in the target signature for a and c. One possibility results in the labels being (a,c)=(2,1), which occurs if the moving cutline encounters a part of the SAW that is part of a loop that already exists. Although this loop changes topologically (because the resulting signature describes two loops), the associated lower and upper edge elsewhere on the cutline keep their labels 1 and 2.

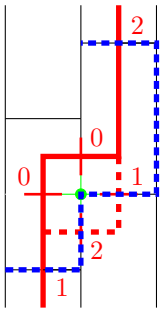


Figure 11: Examples of what target signatures can emerge when the cutline is moved by one vertex. There are many more possible labels for edges b and d of the signatures, that can all contribute to target signatures with several possibilities for the labels of a and c.

Using the notation from figure 10, the update rules for a vertex v to find possible target signature labels (a,c) from the original signature labels (b,d) are based on a few principles:

- If both b and d are non-empty, a and c are empty because a SAW can only visit this vertex once. This only contributes to a valid target signature if (b,d)=(1,2), otherwise the labels of b and/or d are incorrect.
- If either b or d is an upper/lower edge of a loop and the other edge is empty, the loop has to continue through edge a or c, so one of these has to become an upper/lower edge, and the other edge becomes an empty edge.

- If either b or d is a free edge and the other one is empty, the free part of the SAW can continue to through a or c (as with the previous case), but it can also end in this vertex, which would yield (a,c)=(0,0). One of these cases is illustrated in example 1 of figure 11.
- The most complex case is if the original signature has (b,d)=(0,0): The moving cutline could encounter a part of the SAW that used to lie to the right of the cutline, but has now created new non-empty crossings of the SAW with the cutline. We refer to this as the cutline *catching* a part of the SAW while being updated.
  - It is possible that the cutline does not catch a part of the SAW when being updated at all. In this case, the resulting labels are (a,c)=(0,0).
  - If the cutline catches a free part of the SAW, this free part changes into a loop, meaning that a or c becomes an upper/lower edge. The corresponding free edge elsewhere on the cutline becomes the corresponding lower/upper edge, i.e. the label 3 in the signature that corresponded to this free part of the SAW becomes a 1 or 2.
  - If the cutline catches a loop of the SAW, two separate loops are created, which is illustrated in example 2 of figure 11. The illustrated case yields (a,c)=(2,1), and the crossings of the old loop keep their label, although they now form separate loops. If both crossings of the caught loop lie above or below the vertex, the resulting labels (a,c) are (1,1) or (2,2) respectively. One of the original crossings of the loop elsewhere on the cutline changes into an upper/lower edge.
  - If there is no SAW in region A yet (meaning that the entire SAW lies to the right of the cutline), it is possible to catch the entire SAW. This results in either a, c or both a and c becoming free edges. This can not happen if there is already a part of the SAW on the left, because this part would not be connected to the newly encountered SAW.

An interesting remark is that this procedure can also be used to count self-avoiding polygons (SAP) on lattices using the transfer matrix method. The update rules for SAPs are less complicated because a SAP can only have loops (giving labels 1 and 2) and no free parts (with label 3), so every update rule with free edges can be discarded for SAPs. Practically, the only thing we have to change in our algorithm is to make sure it starts the signature with a connected lower and upper edge, instead of one or two free edges as is the case for SAWs.

After all the possible target signatures S' from an original signature S are determined, updating the generating function is relatively simple: Let $G_S(x)$ be the generating function of S before the vertex was updated, and let $m \in \{0, 1, 2\}$ be the number of non-empty edges in {b,d}. The generating function of the target signature is computed from the generating function of the original signature by multiplying it by x^m:

$$G'_{S'}(x) = x^m G_S(x) \tag{5}$$

This procedure has to be repeated for every existing signature S when updating a vertex. This means that two existing signatures S and T can contribute to the same target signature S'. If this happens, we add the resulting generating functions $x^{m_S} G_S(x)$ and $x^{m_T} G_T(x)$ to find the generating function $G'_{S'}(x)$ of the target signature. Notice that the number of non-empty incoming edges m can differ between existing signatures, hence the notation m_S and m_T. This logic extends to multiple existing signatures that contribute to the same target signature S' and is described in algorithm 1.

To find out if a generated target signature S' is really new, or that it was encountered already and we need to add the generating functions (see algorithm 1, where we need to check if $S' \in \Sigma'$), we would have to compare S' to every target signature that is already generated. If this is implemented carelessly, this would become extremely expensive from a computational point of view. Signatures can be described efficiently using a hash table, so that a signature only has to be compared to signatures with the same hash.

Algorithm 1 Updating a single vertex in the procedure of moving the cutline

Require: A set Σ of signatures corresponding to the current cutline
Require: A generating function $G_S(x)$ for all $S \in \Sigma$
  Initialize the set of target signatures $\Sigma' = \emptyset$
  for every $S \in \Sigma$ **do**
    m = number of non-empty incoming edges from signature S
    for every possible target signature S' that originates from S **do**
      if $S' \notin \Sigma'$ **then**
        Add S' to Σ' and set $G'_{S'}(x) = x^m G_S(x)$
      else
        $G'_{S'}(x) \leftarrow G'_{S'}(x) + x^m G_S(x)$
      end if
    end for
  end for
  return The set Σ' of new signatures, with corresponding generating functions $G'_{S'}(x)$ for $S' \in \Sigma'$

The hash of a signature $S = \{\sigma, ltbr\}$ can be constructed as follows: Every crossing of the cutline has a label $\sigma_i \in \{0, 1, 2, 3\}$ which can be represented by 2 bits. Each truth value l, t, b, r can be described by 1 bit. If we concatenate all those bits, we get one very large integer number N, of which the size depends on the length of the cutline. We can create a hash for the signature S by taking this integer N mod p for some large prime p (the size of the hash table). This operation can be done very quickly, reducing the time complexity of one update step approximately from the number of signatures squared to just the number of signatures.

3.4 Zbarsky's method

For a thorough treatment of Zbarsky's method, we refer to the original paper [6] and the nice exposition [9]. Zbarsky's method is a modification to the 'standard' transfer matrix method we described thus far, which we refer to as Jensen's method and is illustrated in figure 9. The difference between Zbarsky's method and Jensen's method is that Zbarsky's method restricts the number of nonempty horizontal crossings in certain columns, thus restricting the number of signatures. To optimally exploit this restriction of crossings, we move the cutline in a different way than before, where we skip some columns, as in figure 12. To still get the correct result, we apply the inclusion formula. This might seem counterproductive because we have to repeat the process of moving the cutline several times, but Zbarsky showed that ignoring some of the signatures in the process gives a strong bound on the computation time, making this method asymptotically faster (i.e. faster for large enough n) than Jensen's method.

We will describe how Zbarsky's method computes $C^n_{W,H,p}$, the number of SAWs in a certain rectangle. As always, n denotes the length of the SAWs that we want to count. First, we have to choose an integer $2 \leq k < n$ to perform Zbarsky's method. To minimize the computation time, we should take

$$k = \sqrt{n \log(n)}$$

rounded to the nearest integer. Furthermore, let $q = \lfloor n/k \rfloor$. For a SAW in the rectangle called w with length n, we define

$$C(w) = \{\beta \in \{0, 1, ..., k-1\} | \text{ every column with column number mod } k \text{ equal to } \beta \text{ ,}$$
$$\text{has at most } q \text{ horizontal edges of the self avoiding walk in that column}\}.$$

The value $q = \lfloor n/k \rfloor$ is chosen to make the set $C(w)$ nonempty for every SAW w. We give a short proof of this claim:

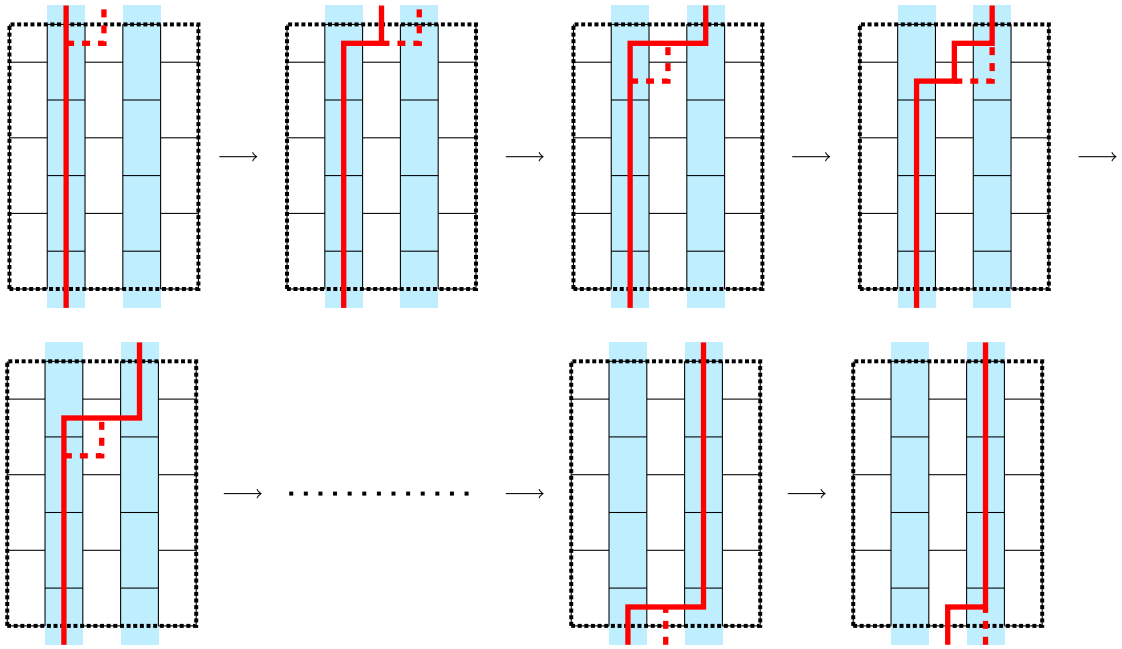


Figure 12: Example of how we 'skip' a column when moving the cutline from one blue column to the next one with Zbarsky's method. The non-blue columns are updated together with the next blue column instead of updating the columns one by one (as in figure 9)

Proof. Say that $C(w)$ is empty, then there are at least k columns with more than $q = \lfloor n/k \rfloor$ crossings (one for each $\beta \in \mathbb{Z}_k$). That is, there at least k columns that are crossed horizontally at least $q+1$ times by the SAW w. The total number of edges of this SAW is therefore greater or equal than

$$k(q+1) = k\left(\left\lfloor \frac{n}{k} \right\rfloor + 1\right) > k\frac{n}{k} = n.$$

But this is a contradiction since the SAW has exactly n edges. □

For each $K \subset \{0, 1, ..., k-1\}$, let N_K denote the number of self-avoiding walks w with $K \subseteq C(w)$, meaning that there are N_K SAWs that have at most q crossings in the set of columns $K + k\mathbb{Z}$. Since we saw that $C(w)$ is nonempty for every SAW w, we have that the set of all SAWs can be written as

$$\{w \mid w \text{ is a SAW}\} = \bigcup_{j=0}^{k-1} \{w \mid w \text{ is a SAW}, j \in C(w)\}.$$

These sets have significant overlap, we can use the inclusion-exclusion principle [10, theorem 10.1] to obtain the total number of SAWs in the rectangle

$$C^n_{W,H,p} = \sum_{K \subseteq \{0,\dots,k-1\}, K \neq \emptyset} (-1)^{|K|+1} N_K. \tag{6}$$

We can count N_K efficiently using the transfer matrix method. We do this using algorithm 1 again, but in the columns with column number in $K + k\mathbb{Z}$, we discard the target signatures S' with more than q crossings in that column. If we can minimize the number of crossings, we will also have fewer signatures which decreases the computation time of the method. Wherever the cutline overlaps with a column in K modulo k, that is the blue columns in figure 13, there are at most q crossings in those parts. At every step, we make sure that as much length of the cutline as possible lies in these columns, prompting us to use this specific way of moving the boundary.

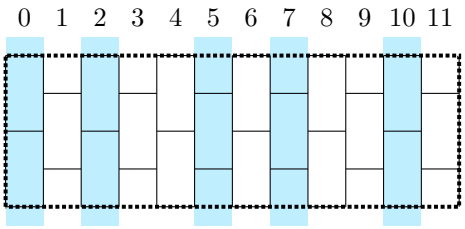


Figure 13: With Zbarsky's method, the cutline is updated from blue column to blue column, as illustrated in figure 12. In this figure, we have $k = 5$, and the cutline stops at columns mod k from the set $K = \{0, 2\}$. In other words, Zbarsky's method updates the cutline between the columns in $K + k\mathbb{Z}$, which are shown in blue. The column numbers are displayed for clarity.

One might question the computational efficiency of Zbarsky's method: For several W, H and p, we have to determine $C^n_{W,H,p}$ by moving the cutline through each of the rectangles $2^k - 1$ times, with the inclusion-exclusion principle (6). Indeed, this results in $2^k - 1$ as many iterations as Jensen's method, and is immeasurably more complex than the simple recursive backtracking algorithm. However, being able to ignore some of the target signatures turns out to outweigh these drawbacks for large enough n. Zbarsky showed that the time complexity of this method is subexponential:

$$t \sim \exp(C\sqrt{n \log n}) \tag{7}$$

where C is a constant. In the limit $n \to \infty$ this will be faster than any exponential computation time. However, without the constant C determined, it remains to be seen if it is also faster than Jensen's method in practice. If the constant C is quite big for example, the algorithm might be slower for all values n that can be reached within reasonable time.

3.5 Pruning

The transfer matrix method as we described it thus far is not yet as fast as it can be. This is because during the calculation a lot of signatures are considered that do not contribute to the count. Pruning is the act of deleting those signatures. While pruning is not necessary for the algorithm to give a correct result, it can help it to become much faster. The number of signatures is the most restricting factor in the computation time, thus the earlier we filter out irrelevant signatures the better.

Our method of pruning is not very advanced and likely can be improved upon significantly. Still, compared to not using pruning it reduces the growth rate of the computation time, and thus eventually the algorithm becomes orders of magnitude faster.

The main way in which we spot invalid signatures is by calculating how many edges are needed to complete a signature. We connect upper crossings to lower crossings in the most efficient way and also make sure the walk touches the right wall somehow. The connection between an upper and lower edge is called a loop. If the number of edges to do so plus the shortest way to get this signature on the left as stored in the generating function is more than n, we discard the signature. A complicating factor is that the cutline does not have to be straight and the fact that we are working on a honeycomb lattice, which prompted us to opt for a simple method.

For each pair of a lower and upper crossing, we calculate the minimum number of horizontal and vertical crossings, by considering both the difference in position of the two crossings and potential nested loops and free edges which have to be avoided on the way. To take into account the

honeycomb lattice, we notice that in one direction the walk cannot make a straight line. Thus for example in the vertical brick pattern, if the walk has to have 5 horizontal edges, this means the walk must also have at least 4 vertical edges in between. Updating the number of horizontal or vertical edges accordingly, we can add both numbers up to get the length of the loop. Additionally, for each loop or free edge that is not nested in another loop, we calculate how many additional edges would be needed to touch the right wall, and take the minimum of these numbers and add it to the number of edges needed to complete loops.

On the one hand, we want to prune as often as possible to reduce the number of signatures. On the other hand, pruning itself costs calculation time. For our method, it turned out to be optimal to prune every two steps, where a step is moving the boundary with one vertex. Pruning every step is almost as fast and reduces memory usage, which also has its applications.

4 Results

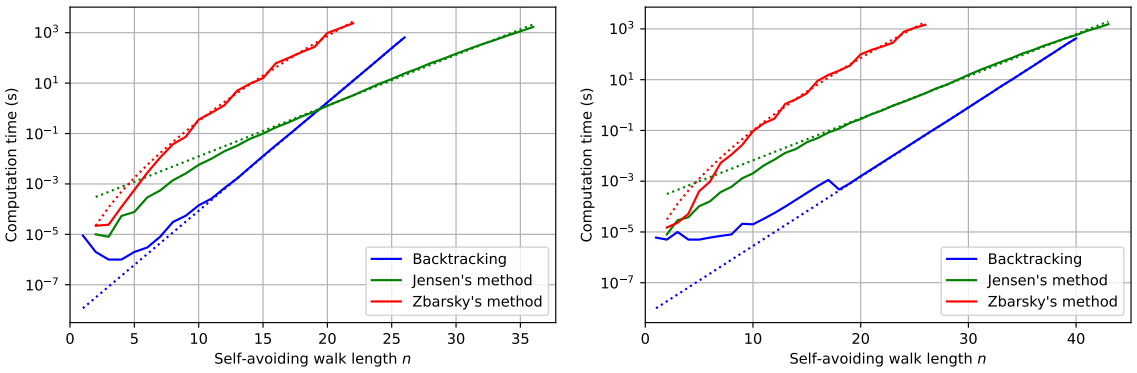
It is time to put the different algorithms to the test and compare them. We will run the backtracking algorithm, Jensen's method and Zbarsky's method, on both the honeycomb grid and the square grid. We will run the code first on a laptop, and then also on a supercomputer.

4.1 First results

We counted the number of SAWs for walk lengths $n = 2, 3, \ldots$, until the iterations take $\sim 10^3$ seconds to run, keeping track of the time it takes for each n to count c_n with each algorithm. We implemented the algorithms in C and we used a laptop with Intel(R) Core(TM) i7-9750H core at 2.60GHz.

We obtained results for c_n that are in accordance with the known sequences[11][12] for both the honeycomb grid and the square grid. The resulting values of c_n are given in appendix B.

Since the methods all produced the correct results for the values of c_n, we shift our focus to the computation time that the algorithms used to compute c_n for each n. We keep track of the computation time for each of the three methods applied to both the honeycomb and square grid, and we plot the results. These are shown in figure 14a for the square grid and in figure 14b. We plot the results in a logarithmic graph so that it is immediately visible whether or not the computation time is exponential in n (which would result in a straight line in the graph).



(a) Square lattice

(b) Honeycomb lattice

Figure 14: Logarithmic plots of the computation time of c_n for different n on the honeycomb grid and square grid. To make these plots, we used all three algorithms. The algorithms were implemented in C and ran on a laptop using one core.

In figure 14, we see that the naive backtracking algorithm is the quickest for the first few n (especially for the honeycomb lattice), but the computation times also increase quite steeply. Apart from the first few n, the computation time seems to increase exponentially, characterized by a linear increase in the logarithmic plot. We ignore the strange behaviour for very low n since this is the result of only very small differences in time. For the square lattice, it could be a result of our implementation of exploiting symmetry.

Jensen's method is slower than backtracking for low values of n, but around $n = 20$ for the square grid, and $n = 40$ for the honeycomb grid, Jensen's method overtakes backtracking. Jensen's method also seems to have exponential time complexity, but the growth rate seems to decrease

slightly for higher n. The growth rate is lower than that of backtracking, as we see by the less steep graph.

We note that Zbarsky's method produces some interesting behaviour: At first, the computation time seems to increase (much) faster than the other methods, but the growth rate of Zbarsky's method seems to decrease, which suggests that the time complexity of Zbarsky's method is indeed subexponential.

Backtracking and Jensen's method seem to have exponential time complexity, as we proved for backtracking and is expected for Jensen's according to the literature. To determine the rate of growth, we took the logarithms of the computation times, and took a linear approximation of the computation times as a function of n, minimizing the mean square error. Returning to non-logarithmic time, we have the approximations

$$t(n) = A \cdot C^n. \tag{8}$$

We call the constant C the growth rate, the values we found can be found in table 1. We put the function fits as dotted lines in 14.

	Backtracking	Jensen's method
Square lattice	2.69	1.59
Honeycomb lattice	1.87	1.46

Table 1: Growth rate C of the computation time for the different algorithms on different lattices. The computation time of the algorithm will be $\mathcal{O}(C^n)$.

For Zbarsky's method, we do not expect exponential computation time. However, it is difficult to say a lot about the time complexity of Zbarsky's method. This method is quite slow compared to the other methods for the values of n in this graph, meaning we have only a few values of n for which we can compare the computation times.

4.2 Supercomputer results

To get a better picture of how the time complexity of Zbarsky's method develops for higher values of n, it is needed to significantly increase our computation time. To do this, we used the Dutch National Supercomputer Snellius located in Amsterdam. Every core of the supercomputer is about as powerful as a core of an ordinary laptop. A supercomputer however has many more cores and more available working memory. We used one node with 1 TB of memory (a fat node), and used 44 of its cores, for about two days. Multithreading was done by allocating the different rectangles to be counted over the different cores. Afterwards, we added up the SAW counts and the computation times. We only considered the honeycomb lattice and used Zbarsky's algorithm. The results, alongside our earlier results for backtracking and Jensen's method, are shown in figure 15.

It requires some justification to show and compare the results of a laptop and a supercomputer in one figure. As said before, per core they are quite comparable, thus after adding up the time spent per core we have a reasonable comparison. The difference we do encounter between the two would be in the form of a scaling factor, which looks like a small vertical shift in a logarithmic graph. In practice the graphs of the preliminary and supercomputer results for Zbarsky's on the honeycomb more or less overlapped, except for very low n.

We see in figure 15 that Zbarsky's method is significantly slower for values of n we are able to reach. The growth rate seems to decrease slowly for larger n. We have not reached the point

Figure 15: Logarithmic plots of the computation time of c_n for different n on the honeycomb grid and square grid of the three algorithms. The algorithms were implemented in C, and backtracking and Jensen's method were run on a laptop using one core. Zbarsky's method was run on the Snellius supercomputer using 44 cores, and the computation times of each core were added together. Since the cores of Snellius are about as fast as a core of a laptop, this is a fair comparison.

where the growth rate of Zbarsky's is lower than Jensen's. Keeping in mind the theoretical time complexity of Zbarsky's, we approximate the logarithm of the computation time as a function of n by the function

$$\log t = a + c\sqrt{n \log(n)},$$

choosing a and c to minimize the mean squared error. This gave us the approximation

$$t(n) = 1.88 \cdot 10^{-6} \cdot 9.83^{\sqrt{n \log(n)}}.$$

This approximation is also shown as a dotted line in figure 15. The main thing we see is that this function fit approximates the observed data very well. If we extrapolate the function fits for Jensen's method and Zbarsky's method for large n and intersect them, this would happen at $n = 158$ with a computation time of order of magnitude 10^{22} seconds.

5 Discussion

Our first observation is that our backtracking algorithm has a growth rate of about μ, both for the square lattice and the honeycomb lattice. This is as predicted because the backtracking algorithm has to count every SAW individually. Consequently, we cannot improve on this much more beyond reducing the constant multiplicative factor. Our implementation of Jensen's method has an exponential growth rate which is lower than μ as a function of n. While initially somewhat slower than backtracking, Jensen's method soon overtakes it and for higher n it will be very significantly faster. Jensen's method overtakes backtracking much earlier on the square grid than on the honeycomb grid because the exponential growth rate μ of the computation time of the backtracking algorithm is lower on the honeycomb grid than on the square grid. Perhaps backtracking is in general a better method for such a sparse lattice. It could also be that our implementation of Jensen's method is not very well adapted to the honeycomb grid.

Our implementation of Jensen's method is not yet optimal and can be improved upon significantly. Jensen's original implementation [5] of his algorithm has a growth rate of about 1.2 for the square lattice. For the honeycomb lattice, this growth rate could even be lower. Thus our implementation of Jensen's method is better than backtracking, but not optimal yet.

Since Zbarsky's method is based on Jensen's method, our implementation of it must also not be optimal yet. For the values of n that we reached, it is significantly slower than both backtracking and Jensen's method. However, where other algorithms have exponential time complexity, it indeed seems to be the case that Zbarsky's method has subexponential time complexity. After function fitting the constants, the practical computation time matches the theoretical computation time very well.

On the honeycomb lattice, if we intersect the function fits of Jensen's and Zbarsky's method, this would happen at approximately $n = 158$. Since this is an extrapolation there is a very large margin of error in this prediction. Let us however assume for one moment this prediction is approximately correct, then $n = 158$ is completely unreachable with our current implementation. The world record for counting SAWs on the honeycomb lattice is $n = 105$ by Jensen in 2006 [13]. This is already a long time ago and likely can be improved upon. We think it is not currently possible to use Zbarsky's method to immediately improve current records. However, the method might be faster than previous methods in the near future, after optimizing the method further, and more computational resources become available.

Zbarsky showed that his new method has subexponential time complexity. He however did not determine the constants of this time complexity. We determined those constants experimentally for our method. These constants are quite high, but also not extraordinarily high. This method should therefore be taken seriously in the future as an alternative to the current popular transfer matrix methods.

5.1 Outlook

There are still a lot of improvements possible to our code. The most obvious starting point for this is our pruning algorithm. We used a quite simple bound to estimate how many edges are needed to connect the upper and lower edges, but a sharper bound will allow us to discard more signatures. We also did not include the touching of the lower and upper sides in our pruning. We found out it is desirable to do the pruning as often as possible, also to reduce memory usage. One possible improvement on our current implementation is that the pruning calculations should not be redone every time the cutline is updated, but the results should be stored temporarily. This way we could integrate the pruning and generating of signatures, not generating invalid signatures instead of removing them afterwards.

More fundamentally, the value q in Zbarsky's algorithm should be chosen as low as possible to reduce the most number of signatures. The value $q = \lfloor n/k \rfloor$ is specifically chosen to make Zbarsky's method work for the square grid. However, with a brick grid (i.e. the transformed honeycomb grid), it should be possible in some cases to choose a lower value of q because a certain number of horizontal crossings automatically implies the existence of a number of vertical edges.

Another fundamental observation on implementing Zbarsky's method in practice is that the value k was chosen to ensure the optimal asymptotic computation time for $n \to \infty$. However, it might be advisable to choose k differently to count SAWs of a finite length n. We note that for very low k, Zbarsky's algorithm more or less coincides with Jensen's algorithm. For low n Jensen's algorithm is faster than Zbarsky's, thus perhaps choosing k a bit lower is worth trying.

References

[1] László Lovász. *Graph Theory Over 45 Years*, pages 85–95. Springer Berlin Heidelberg, 2011. https://doi.org/10.1007/978-3-642-19533-4_7.

[2] Paul Flory. *Principles of Polymer Chemistry*. Cornell University Press, 1953.

[3] Gordon Slade. The self-avoiding walk: A brief survey. 2011. https://personal.math.ubc.ca/~slade/spa_proceedings.pdf.

[4] I. G. Enting. Generating functions for enumerating self-avoiding rings on the square lattice. *J. Phys. A: Math. Gen. 13 3713*, 1980. https://dx.doi.org/10.1088/0305-4470/13/12/021.

[5] Iwan Jensen. A new transfer-matrix algorithm for exact enumerations: self-avoiding walks on the square lattice. *preprint*, 2013. https://arxiv.org/pdf/1309.6709.pdf.

[6] Samuel Zbarsky. Asymptotically faster algorithm for counting self-avoiding walks and self-avoiding polygons. *Journal of Physics A: Mathematical and Theoretical*, 52(50), nov 2019. https://dx.doi.org/10.1088/1751-8121/ab52b0.

[7] Gordon Slade Neal Madras. *The Self-Avoiding Walk*. Birkhäuser Boston, MA, 1996. https://doi.org/10.1007/978-1-4612-4132-4.

[8] Stanislav Smirnov Hugo Duminil-Copin. The connective constant of the honeycomb lattice equals $\sqrt{2+\sqrt{2}}$. *Annals of mathematics*, 175, 2012. https://doi.org/10.4007/annals.2012.175.3.14.

[9] Nathan van den Berg, Robin van der Laag, Michiel van den Eshof, Myrthe van Leeuwen, and Bjorn Buitink. Zbarsky's asymptotically faster algorithm for counting self avoiding polygons. Orientation on mathematical research, Utrecht University, period 1, nov 2022.

[10] R. M. Wilson J. H. van Lint. *A Course in Combinatorics*. Cambridge University Press, 2 edition, 2001. https://doi.org/10.1017/CBO9780511987045.

[11] N. J. A. Sloane. A001668 - oeis. https://oeis.org/A001668. visited on 12/4/2023.

[12] A. J. Guttmann N. J. A. Sloane. A001668 - oeis. https://oeis.org/A001411. visited on 12/4/2023.

[13] Iwan Jensen. Honeycomb lattice polygons and walks as a test of series analysis techniques. *J. Phys.: Conf. Ser.*, 42(163), 2006. https://doi.org/10.1088/1742-6596/42/1/016.

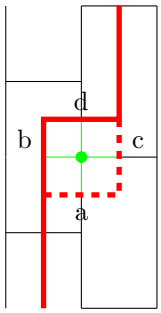
A Appendix: Update rules

In this appendix, we provided the detailed update rules for signatures when updating the cutline by one vertex. Specifically, we give all possible combinations of labels (a,c) in a target signature that originates from a signature with labels (b,d) on the edges around the updated vertex. These rules are illustrated with figures, where the red line indicates the original position of the cutline, and the dashed line is the shape the cutline will move to. The characters a, b, c and d denote the crossing of the cutline with a grid line that may or may not exist around the updated vertex, which is indicated in green in the figures.
We will look at the updated vertex in the middle to separate different cases. We will define this point as odd when there is no horizontal line segment to its right. We will define this point as even when there is no horizontal line segment to its left. We will look at six different situations.

The updated vertex is even

When the updated vertex is even we will have three possible situations. Namely, the updated vertex is in the middle of the grid, the updated vertex is on the right vertical border line of the grid or the updated vertex is on the left vertical border line of the grid. In the case that the updated vertex is even, b will always be 0 because there is no line segment in b.
For simplicity reasons, we will represent the possible values of the edges in a table.

The updated vertex is in the middle



$(\mathbf{b}, \mathbf{d})$	$(\mathbf{a}, \mathbf{c})$
$(0,0)$	$(0,0), (0,1)^e, (0,2)^f,$ $(1,0)^e, (1,1)^h, (1,3)^e,$ $(2,0)^f, (2,1)^g, (2,2)^i, (2,3)^f,$ $(3,1)^e, (3,2)^f$
$(0,1)^!$	$(0,1), (1,0)$
$(0,2)^?$	$(0,2), (2,0)$
$(0,3)^@$	$(0,0), (0,3), (3,0)$

Figure 16: The updated vertex is even and in the middle

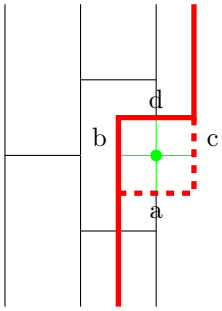
We will define a set of notes, which we will also use in the other cases.

Notes:
e: a free edge (3) above this point has to become an upper edge (2).
f: a free edge (3) below this point has to become a lower edge (1).
g: there has to be a lower edge (1) below and an upper edge (2) above.
h: a lower edge (1) above this point has to become an upper edge (2).
i: an upper edge (2) below this point has to become a lower edge (1).
!: There is a lower edge, so this still has to exist when the cutline moves. Otherwise, this would not be a lower edge.
?: There is an upper edge, so this still has to exist when the cutline moves. Otherwise, this would not be an upper edge.
@: There is a free edge, so now there are three possibilities: Or this is the end of the free edge, so after moving the cutline, it doesn't exist anymore. Or this free edge still exists in either a or c, if possible.

Note that in all situations where we have a crossing before the cutline moves, this crossing can not give two new paths, just one, after the cutline moved. Otherwise, the self-avoiding walk would split up. It is also not possible that the self-avoiding walk stops, because in that case it would not have touched all the borders and it would be invalid.
Of course a lower or an upper edge can never become a free edge after moving the cutline.

Not possible situations for (a, c) when $(b, d) = (0, 0)$:
$(1, 2)$: Because if a is the lower and c is its upper crossing, this would give a closed loop, which would not touch the left border. If c would not be the corresponding upper crossing for a, there would be a lower edge of some crossing above c, and c would be the upper edge of some crossing below a, these paths would cross each other somewhere on the right side of the cutline.
$(0, 3)$, $(3, 0)$ or $(3, 3)$: This is only possible if the rest of the signature is empty, but if it would be empty this would give an invalid walk because this would not touch all borders.

The updated vertex is on the right border

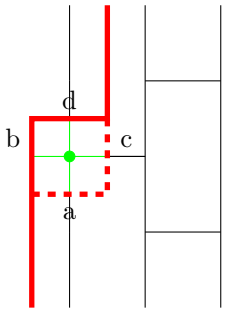


$(\mathbf{b}, \mathbf{d})$	$(\mathbf{a}, \mathbf{c})$
$(0, 0)$	$(0, 0), (2, 0)^f$
$(0, 2)^?$	$(2, 0)$
$(0, 3)$	$(0, 0), (3, 0)$

Figure 17: The updated vertex is even and on the right border

Due to the fact that this is on a border, many cases vanish. The green line to c doesn't exists, so c will always be 0.
Because the signature is empty above the cutline, there can never be a lower edge (1) in a or d.
Also there will never be a free edge in a because in that case the self-avoiding walk is not valid, due to the fact that it would not touch all the borders of the rectangle.

The updated vertex is on the left border



$(\mathbf{b}, \mathbf{d})$	$(\mathbf{a}, \mathbf{c})$
$(0, 0)$	$(0, 0), (1, 0)^e, (3, 0)$
	$(0, 1)^e, (0, 3), (1, 1)^e$
	$(1, 3)^e, (3, 1)^e, (3, 3)$
$(0, 1)^!$	$(0, 1), (1, 0)$
$(0, 3)^@$	$(0, 0), (3, 0), (0, 3)$

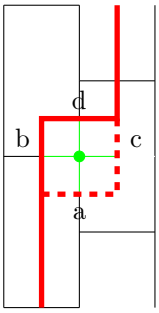
Figure 18: The updated vertex is even and on the left border

a, c or d can never be an upper edge (2) because the signature below this point is empty.

The updated vertex is odd

Again we will have three possible situations. The updated vertex is in the middle of the grid, the updated vertex is on the right vertical border line of the grid or the updated vertex is in the left vertical border line of the grid. In the case that the updated vertex is odd, c will always be 0.

The updated vertex is in the middle

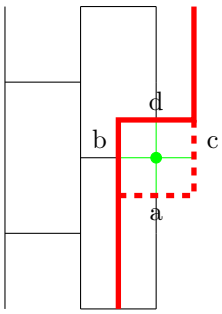


$(\mathbf{b},\mathbf{d})$	$(\mathbf{a},\mathbf{c})$
$(0,0)$	$(0,0),(1,0)^{e},(2,0)^{f}$
$(0,1),(1,0)^{!}$	$(1,0)$
$(0,2),(2,0)^{?}$	$(2,0)$
$(0,3),(3,0)^{@}$	$(0,0),(3,0)$
$(1,2)$	$(0,0)$

Figure 19: The updated vertex is odd and in the middle

Again after $(0,0)$ it is not possible to get a free edge in a because then this signature should be empty, but in that case, this walk would not be valid because not all borders would be touched. It cannot be that two paths come together and go on as one, so always, either b or d has to be 0. This is only possible if b is the lower edge and d is the upper edge, so when b is 1 and d is 2. Then we just cut a corner of the walk, which will be non-existent when the border is moved.

The updated vertex is on the right border

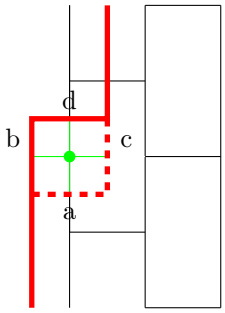


$(\mathbf{b},\mathbf{d})$	$(\mathbf{a},\mathbf{c})$
$(0,0)$	$(0,0),(2,0)^{f}$
$(0,2)^{?}$	$(2,0)$
$(0,3),(3,0)^{@}$	$(0,0),(3,0)$
$(1,2)$	$(0,0)$

Figure 20: The updated vertex is odd and on the right border

For b and d the same holds as in the last example, either one of them should be 0 or b should be 1 and d should be 2.
Again d can never be a lower edge (1) because the signature is empty 'above' d, the same holds for a.
After $(0,0)$ it is not possible to get a free edge in a because if there would appear an edge, this would be an upper edge because a free edge can not appear due to the fact that we are on the right border.

The updated vertex is on the left border



$(\mathbf{b},\mathbf{d})$	$(\mathbf{a},\mathbf{c})$
$(0,0)$	$(0,0),(1,0)^{e},(3,0)$
$(0,1)^{!}$	$(1,0)$
$(0,3)^{@}$	$(0,0),(3,0)$

Figure 21: The updated vertex is odd and on the left border

For sure b and c will always be 0 so this limits the cases.
Again d can never be an upper edge (2). And also a can never become an upper edge.
In this case, after $(0,0)$ one can also find a free edge (3) in a, in contrast to the case on the right border.

B Appendix: Results

n	c_n
2	12
3	36
4	100
5	284
6	780
7	2172
8	5916
9	16268
10	44100
11	120292
12	324932
13	881500
14	2374444
15	6416596
16	17245332
17	46466676
18	124658732
19	335116620
20	897697164
21	2408806028
22	6444560484
23	17266613812
24	46146397316
25	123481354908
26	329712786220
27	881317491628
28	2351378582244
29	6279396229332
30	16741957935348
31	44673816630956
32	119034997913020
33	317406598267076
34	845279074648708
35	2252534077759844
36	5995740499124412

Number of SAWs on the square lattice
(obtained using Jensen's method)

n	c_n
2	6
3	12
4	24
5	48
6	90
7	174
8	336
9	648
10	1218
11	2328
12	4416
13	8388
14	15780
15	29892
16	56268
17	106200
18	199350
19	375504
20	704304
21	1323996
22	2479692
23	4654464
24	8710212
25	16328220
26	30526374
27	57161568
28	106794084
29	199788408
30	372996450
31	697217994
32	1300954248
33	2430053136
34	4531816950
35	8459583678
36	15769091448
37	29419727280
38	54816035922
39	102216080286
40	190380602052
41	354843312276
42	660671299170
43	1230891734724

Number of SAWs on the honeycomb lattice
(obtained using Jensen's method)