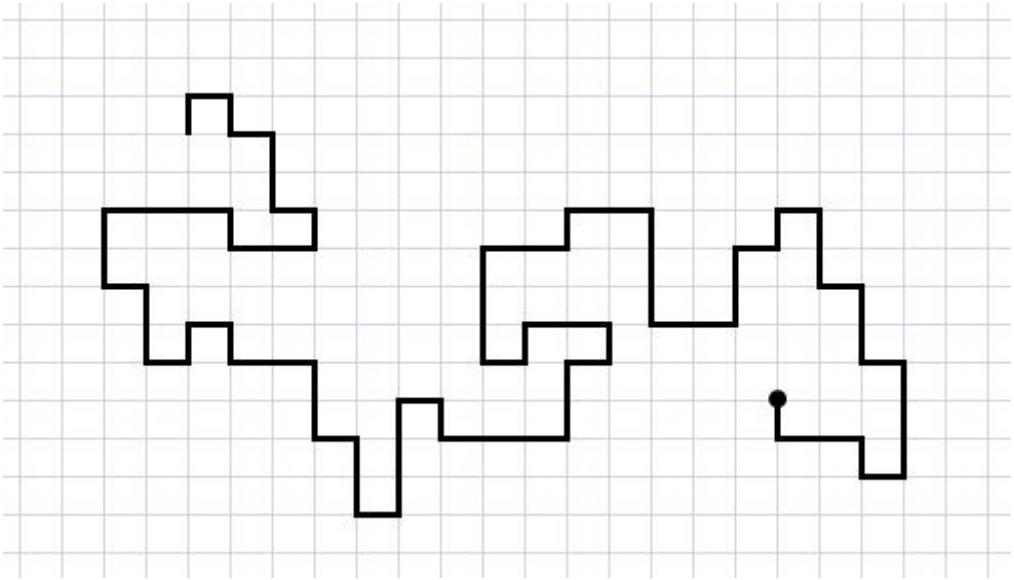
Counting Self-Avoiding Walks on the 2D Square Lattice

Winfried van den Dool
Thesis supervisor: Rob Bisseling

June 15, 2016



Contents

1 Introduction

In this thesis we describe multiple algorithms for counting self-avoiding walks on the square lattice. We begin by informally introducing this problem and its background. For a quick read only sections 1.2 and 1.4 of this introduction are relevant.
After having described a few known algorithms, one of which, the so-called Finite Lattice Method, forming a major part of this thesis, we present our own newly developed algorithm together with some promising results.

1.1 The Unexpected Difficulty of Seemingly Easy Questions

Let us start with a short anecdote, inspired by [1]. We go back to about a hundred years ago, when mathematicians Srinivasa Ramanujan and Godfrey Harold Hardy considered the following question: How many different ways are there to write a number as a sum of positive integers?
Now this doesn't seem to be such a hard question. It might take more than a little while to find the answer, but there must obviously be some simple and elegant solution to such a compact question, right? Didn't some Greek mathematicians examine such problems over 2000 years earlier? Let us first clarify this little puzzle some more, and look for example at the number 4. We find five distinct ways to divide it in smaller integers:

$$4$$

$$3 + 1$$

$$2 + 2$$

$$2 + 1 + 1$$

$$1 + 1 + 1 + 1$$

As can be seen above $1 + 3$ and $3 + 1$ are considered as identical options. We refer to $p(n)$, or the partition of n, as the number of distinct ways of writing n as a sum of integers. Using this notation, we found $p(4) = 5$. The first few terms of the series $p(n)$ are: 1, 2, 3, 5, 7, 11, 15 ...
Though the problem seems simple, finding an actual formula for $p(n)$ turned out to be quite hard. In fact at the beginning of the 20th century, most mathematicians had given up on the search. But not Ramanujan, the young clerk from India. Being completely auto-didactic led to his unique optimism that a solution had to be found. In collaboration with the more critical English mathematician Hardy, always checking his wild enthusiasm, Ramanujan succeeded in finally finding the

formula for $p(n)$. For all those loving the elegance of mathematics, how difficult questions can be solved by a simple trick or a short cunning proof, don't get too excited. This is what the formula looked like:

$$p(n) = \frac{1}{2\pi\sqrt{2}} \sum_{1\leq k<N} \sqrt{k}\Big(\sum_{\substack{1\leq h<k \\ (h,k)=1}} \omega_{h,k} e^{-2\pi i \frac{hn}{k}} \Big) \frac{\mathrm{d}}{\mathrm{d}n} \left(\frac{e^{\left(\frac{\pi\sqrt{n-\frac{1}{24}}}{k} \sqrt{\frac{2}{3}} \right)}}{\sqrt{n-\frac{1}{24}}} \right) + O(n^{-\frac{1}{4}})$$

Where N may be taken to be of order $\sqrt{n}$, and (h,k) refers to the greatest common divisor of h and k. The value of $\omega_{h,k}$ is given by $e^{\pi i s(h,k)}$ with $s(h,k)$ the Dedekind sum $D(1,h,k)$.
What's more, instead of giving the exact number, Hardy and Ramanujan's complicated formula produces an answer that is correct when rounded to the nearest whole number. So, for instance, when the formula is fed with the number 200 it outputs a value to which the nearest whole number is $3,972,999,029,388$. [1]

The subject in the coming chapters is a different one. We will not count the number of distinct partitions of n, but count something totally different: the number of distinct self-avoiding walks of length n on the square lattice. The approach does not involve number theory, or other types of mathematics used by Ramanujan and Hardy (although a solution to this currently unsolved problem might of course be found in these areas of mathematics). The problem that is considered however, is both ominously and excitingly similar in one respect: a seemingly easy question reveals a much more complicated problem under closer inspection.

So how complicated is the problem that we are talking about? Well to get an idea, not only is there no known formula that gives the number of distinct self-avoiding walks, mathematicians and computer scientists are even having a hard time finding an efficient algorithm to do the counting. The current record is set at a length of only 79! Comparing this to the problem of partitioning integers, where a simple algorithm is easily applicable[2] one might believe we are dealing with an even more unpredictable problem here. Fortunately, in a time where mathematicians are increasingly aided by computers, new methods of dealing with these problems arise. In what follows we focus mainly on algorithms, although some theoretical results are still involved.

[1] Later on, a variation of their formula would be discovered by Hans Rademacher, giving the answer on the nose. It doesn't get much prettier, mainly involving taking the first sum to infinity.

[2] Let $p(n,m)$ be the number of partitions of n using only positive integers $\leq m$. By splitting on what part we partition out of m first we get the following relation: $p(n,m) = \sum_{k=1}^{m} p(n-k,k)$. Combined with $p(n) = p(n,n) = p(n,m)$ for $m > n$ we can find the answer.

1.2 The Self-Avoiding Walk

Before going any further, let us first give a formal definition. An n-step self-avoiding walk (SAW) is a sequence of n moves on a lattice that does not visit the same point more than once. The lattice we consider is the 2D infinite square lattice, $\mathbb{L}^2$, or more formally, the graph with vertex set $\mathbb{Z}^2 \subset \mathbb{R}^2$ and edge set formed by all unit length segments between vertices in $\mathbb{Z}^2$. A self-avoiding walk of length n on such a lattice is a sequence of $n+1$ distinct vertices $(v_0, ..., v_n)$, such that $v_i \in \mathbb{Z}^2$ and $\{v_{i-1}, v_i\}$ is an edge in $\mathbb{L}^2$ for $i = 1, \ldots, n$. We let c_n denote the total number of distinct n-step self-avoiding walks on this lattice. We usually let v_0 be the origin, and view translations of v_0 as identical SAWs. It follows from the definition that rotations and reflections of SAWs are considered different.
In this setting, the number of 'non-returning' walks of length $n \geq 1$, not moving back but allowed to intersect themselves later on, is $4^1 \cdot 3^{n-1}$. (At the start 4 directions can be chosen, after which due to the non-returning constraint at each point 3 directions in which to continue are possible.) Walks of lengths up to $n = 3$ that do not move back cannot intersect themselves yet, so the first few terms can safely be given: $c_1 = 4$, $c_2 = 12$, $c_3 = 36$. To get a feel of how things change when the self-avoiding constraint kicks in, consider $c_4 = 4 \cdot 3^{4-1} - 8 = 100$, where we have had to remove the 8 cases in which a walk of length 4 can intersect itself.
The approach is obviously not suited for larger n, but does reveal a simple upper bound for c_n. A lower bound can be given by noticing that a walk that solely moves in positive directions (up and right) never intersects itself. We deduce that, for $n > 3$:

$$2^n < c_n < 4 \cdot 3^{n-1}. \tag{1.1}$$

This is a pretty rough estimate, and much better estimates are available. It is conjectured that $c_n \sim A\mu^n n^{\gamma-1}$ for large n, with A, γ and μ positive constants. Most important for our purposes is the exponential behavior $c_n \sim \mu^n$, where μ, also called the connective constant, has been approximated $\mu \approx 2.638$ [2]. Knowing the asymptotic behavior of c_n shall in this work mainly be useful for estimating the running times of different algorithms. However, the other way around, algorithms that compute c_n exactly have contributed to better approximations of the asymptotic behavior of c_n, most recently in [17].
In two dimensions it is conjectured that $\gamma = 43/32$, independent of what lattice we're in [7]. Naturally, the problem is not unique to the square lattice. Although it has only been approximated for the square lattice, the value of μ has successfully been derived for the hexagonal (honeycomb) lattice [14].

Not only the number of self-avoiding walks of length n is sought after. More general metric properties are also studied, consider for example the mean (squared) end-to-end distance of self-avoiding walks. Although the algorithms for finding c_n discussed in this work can be (and have been) extended to also give the values of such metric properties, we only consider them in this work as targeting one question: what is c_n?

1.3 Why Self-Avoiding Walks? Some Context

We first note that attempts to solve problems like the SAW problem can lead to advances elsewhere in mathematics and computer sciences. While we consider this as a sufficient reason to continue puzzling on such matters, the self-avoiding walk problem is actually of fundamental interest in statistical mechanics and polymer chemistry, where the problem was first stated [4].
That the self-avoiding walk problem originated in polymer chemistry can be understood by considering linking monomers to form polymer chains. Given the restriction that two monomers cannot find themselves on the same point in space or on a surface, we arrive at the notion of a 'self-avoiding' chain of monomers.

Initially, the best mathematical model for encoding the unique properties of such polymers was a random walk. The Swiss chemist Werner Kuhn was the first to propose this model, in 1934, using this so called 'ideal chain' approximation to apply Boltzmann's entropy formula[3] to the modeling of rubber molecules [3].
One of the assumptions made by Kuhn was that the mean squared end to end distance R^2 grew as the number of monomers N in the polymer, or $R \sim \sqrt{N}$.[4] However, years after this assumption was made, it was found to be incorrect, as R turns out to grow faster than $\sqrt{N}$. In 1953, the American chemist Paul Flory suggested that while random walks tend to occasionally 'trap' themselves, the monomers tend to bounce away from each other. Applying this so called excluded-volume constraint to polymer molecules, he was the first to create the notion of a self-avoiding walk [4].

[3]Boltzmann's equation gives the relationship between the physical concept called entropy, and the number of ways in which the atoms or molecules of a thermodynamic system can be arranged. A better understanding of the entropy of a certain system leads to more knowledge of other macroscopic properties of the system. In this way, through entropy, combinatorial problems as the one we are currently studying are more often found to have important links with physical systems.

[4]Note that this is in accordance with the theory of Brownian motion, where the mean squared displacement x^2 of a particle is proportional to the elapsed time: $x^2 \sim t$.

Although the origin of the problem lies in polymer chemistry, later on connections between the self-avoiding walk model and models in statistical physics were found, further confirming the relevance of this problem for the world beyond mathematics. Most notably French physicist Pierre-Gilles de Gennes discovered a connection between the self-avoiding walk and spin systems of classical statistical mechanics [5], see also [6], considering the self-avoiding walk as a 'zero-component' ferromagnet. (More formally the limit is taken of n goes to zero for the so-called n-vector or $O(n)$ model from statistical mechanics.) This connection has also been used to conjecture the value of the constant γ, partially describing the asymptotic behavior of c_n, mentioned earlier [7].
Inspired by the connections with statistical mechanics, soon multiple new approaches were presented considering the self-avoiding walk as a critical phenomenon[5] on its own, without appealing to specific already known models in physics [8] [9]. This has increased the possibility of the SAW-problem becoming relevant in other (perhaps currently still unknown) critical systems in physics or other fields.

1.4 Algorithm Complexity

Although there are numerous methods for approximating c_n, there currently is no known formula for determining the actual number. With the rise of computers, and without Ramanujan to help us out, another question arises: is there an efficient algorithm to find the number of self-avoiding walks?
Say the minimal number of basic computing operations required by an algorithm to give the answer c_n is given by $f(n)$. We say the algorithm is 'running in polynomial time' if $f(n)$ is a polynomial function (rather than exponential for example), however fast it grows in the 'input' n. We can talk about 'computing time' instead of 'number of operations', because we assume that the time (in seconds, hours etc) is a linear function of the number of operations, thus not changing the (e.g. polynomial) nature of f.
Currently all the algorithms that are used for this problem have an exponential running time. In the rest of this work we shall describe the complexity of the discussed algorithms by the 'running time function':

$$T(n) \sim \lambda^n \tag{1.2}$$

There is no known algorithm giving the answer in polynomial time, and the challenge has mainly been to minimize the base λ of the exponential function describing the algorithm's running time.
In some cases reasonably good upper bounds for λ can be derived. However, algorithms can also be so complicated that understanding their behavior exactly is

[5]Meaning that it can be considered as a system in a certain (scaling) limit.

nearly impossible, perhaps more difficult than the original problem of counting self-avoiding walks. In those cases numerical results are used to give estimates for λ.
The current best algorithm for counting self-avoiding walks in the square lattice, i.e. with the lowest λ (≈ 1.3), is an extension of the so-called finite lattice method. We describe this method, as well as the improvements made to it, in chapter 4.

1.5 Proof of Existence of μ

The actual behavior of c_n might be very complex, even more so than $c_n \sim A\mu^n n^{\gamma-1}$ due to many correction terms [10], it is however mainly dominated by the exponential behavior μ^n. In this thesis we do not require a very precise estimate of c_n, because we are mainly interested in determining running times of algorithms, where much rougher estimates are made. It is for this reason that our focus lies only on μ.
In this section we give a proof (taken from [11]) of the existence of μ, defined[6] as $\mu = \lim_{n\to\infty} c_n^{1/n}$. This proof does not contribute to a better understanding of the algorithms in the remainder of this thesis and might be skipped if the reader is not interested in it. We have included it, because the proof is required to justify many of the arguments concerning algorithm running times that follow in the next sections, which are all based on the exponential behavior of c_n.

Theorem 1.1. *The limit* $\lim_{n\to\infty} c_n^{1/n}$*, converges to* $\mu < \infty$.

Proof. To prove this we go back to the intuition of a self-avoiding walk. We realize two things. Firstly, every self-avoiding walk of length $n+m$ can be decomposed into two smaller SAWs of length n and m. Secondly consider combining two self-avoiding walks, one of length n and the other of length m, having them both start in the origin. In the case of no intersections, we end up with a self-avoiding walk of length $n+m$. However there must also be at least one case of intersecting SAWs (consider for example the case where the smaller SAW is entirely overlapped by the larger SAW). We can therefore say that $c_{n+m} \leq c_n \cdot c_m$. Taking the log of both sides gives $\log c_{n+m} \leq \log c_n + \log c_m$. We shall now use a lemma, also known as Fekete's lemma.

Definition. *A sequence* (a_n) *of nonnegative terms is sub-additive if* $a_{n+m} \leq a_n + a_m$ *for all* $n, m \in \mathbb{N}$

[6] This might seem like a strange way to define μ, but note that $\lim_{n\to\infty} \frac{c_n}{\mu^n} = 1$ would suggest an absence of polynomial or other smaller than exponential corrections in the large-n limit of c_n. We are not actually proving that c_n goes like μ^n for large n. For this it is obviously required that $\lim_{n\to\infty} c_n^{1/n}$ converges, but this is not a sufficient condition. In short, we cannot rule out the possibility that, for example, a polynomial term also influences c_n.

Lemma 1.2. *Let (a_n) be a sub-additive sequence of nonnegative terms a_n. Then $\frac{a_n}{n}$ is bounded below and $\lim_{n\to\infty} \frac{a_n}{n} = \inf\left\{\frac{a_n}{n} : n \in \mathbb{N}\right\}$.*

Applying the lemma to $a_n = \log c_n$ we find that $\frac{\log c_n}{n} = \log\left(c_n^{1/n}\right)$ converges for large n. We define $\lim_{n\to\infty} \log\left(c_n^{1/n}\right) = \log\mu$. Since $y = e^x$ is continuous in $x = \log\mu$, we can use the composition law for limits to conclude that $\lim_{n\to\infty} c_n^{1/n} = \mu$, completing the proof. □

As mentioned in footnote 6, we did not actually prove that c_n goes like μ^n for large n. More precisely, what we have proved is that μ describes the exponential contribution to the asymptotic behavior. In the large-n limit $c_n^{1/n}$ is stripped of all polynomial (and other smaller than exponential) contributions, leaving only the exponential term. And because it converges, we also know that there are no larger terms (like α^{n^2} $(\alpha > 1)$ or $n!$) involved in the large n limit.
We can go a little bit further, looking at one aspect of the lemma we have not explicitly used. Fekete's lemma also gives us:

$$\begin{aligned} \inf\left\{\frac{\log c_n}{n} : n \in \mathbb{N}\right\} &= \log\mu, \text{ so} \\ \frac{\log c_n}{n} &\geq \log\mu \\ c_n &\geq \mu^n. \end{aligned}$$

2 The Simple Recursive Algorithm

In this chapter we describe an algorithm for counting self-avoiding walks. As it is the simplest algorithm that one can think of, and it is a recursive algorithm, we call it the 'Simple Recursive Algorithm' for later reference.

2.1 Introduction

The first counting method that might come to mind, is recursively moving through the lattice, at each vertex checking all directions to determine in what ways to continue 'walking'. When a completed SAW has been formed, we go back a step and check for other directions that have not been chosen previously. More formally, this method can be considered a Depth-First-Search algorithm, and we are actually explicitly searching (and thereby generating) all the SAWs before counting them.
Although this method is very inefficient, it does give a first impression of the main difficulties that we're dealing with. Namely, how it is the sheer number of self-avoiding walks that significantly limits us in our possibilities. More significantly, this algorithm will be a reference point to determine what the worst-case running time is that we have to beat, and how well other algorithms are doing.

We begin by describing the algorithm itself, working towards the main method by first describing the different variables and another method used. Next we discuss the complexity of the algorithm (the value of λ), which can in this case be derived exactly.

2.2 The Algorithm

To know which vertices have not yet been visited, we maintain a two-dimensional ($O(n^2)$) array representing the entire lattice (all vertices) a walk of length n can reach. We give a location (vertex) the output value 1 if it has already been visited by the walk, and output 0 otherwise. Because the array can't contain negative coordinates, we translate the origin from $(0, 0)$ to (n, n). For the actual counting we track the number of walks of length n already successfully reached (and counted) with the value $Total$.
We will now define the recursive method that forms the most important part of the algorithm (but does by itself not return any answers). When called upon, this method can be seen as being 'unleashed' at a certain lattice point with a number of steps allowed to be taken, expanding sequentially (forming one walk at a time) in all directions. Each time it has successfully placed all its steps, or when it has trapped itself, it will go back, continuing in other, not yet attempted, directions.

Consider x and y as describing the coordinates of the starting point of this method, and j the number of steps it has left. When $j = 0$ we will consider this as having successfully found an entire self-avoiding walk. In that case we shall add the walk to the running total $Total$. Note that the fact that the method refers to itself (in the four if-loops, one for each possible walking direction), makes this algorithm recursive.

---

function RecursiveWalk(x, y, j)
  if $j = 0$ **then**
    $Total = Total + 1$
      ▷ Successfully reached a self-avoiding walk of the desired length
  else ▷ Check what directions are possible to continue walking
    if $Lattice[x + 1, y] = 0$ **then**
      $Lattice[x + 1, y] = 1$
      RecursiveWalk$(x + 1, y, j - 1)$
    if $Lattice[x - 1, y] = 0$ **then**
      $Lattice[x - 1, y] = 1$
      RecursiveWalk$(x - 1, y, j - 1)$
    if $Lattice[x, y + 1] = 0$ **then**
      $Lattice[x, y + 1] = 1$
      RecursiveWalk$(x, y + 1, j - 1)$
    if $Lattice[x, y - 1] = 0$ **then**
      $Lattice[x, y - 1] = 1$
      RecursiveWalk$(x, y - 1, j - 1)$
  $Lattice[x, y] = 0$ ▷ Erase footprint upon returning

Although we already have a nearly working algorithm purely based on this function (just start the method in the origin with n steps left to find c_n), we can first easily make use of some symmetry. Every walk, except for the four walks that go straight in one direction, can be rotated and reflected to give a total of 8 similar walks. We can incorporate this by letting the first move go up, and letting the first move not going up to be one to the right. The full algorithm will therefore be (keeping in mind the translation of the origin to (n, n)):

```
function SelfAvoidingWalksOfLength(n)
    Total = 0
    Create new array Lattice
    Lattice[n, n] = 1                                  ▷ start at origin
    for h = 1 to n − 1 do          ▷ h = number of steps up to begin with
        Lattice[n, n + h] = 1                          ▷ walking up one extra step
        Lattice[n + 1, n + h] = 1                      ▷ first step to the right
        RecursiveWalk(n + 1, n + h, n − h − 1)
return 8 × Total + 4                                   ▷ add the 4 straight line segments
```

This algorithm gives the first 20 terms of c_n without a problem. But it takes about a minute to calculate c_{22}, 2.5 minutes to calculate c_{23}, 7 minutes to calculate c_{24} and 18 minutes to calculate c_{25}. It is obvious that the record of c_{79} will not be broken by this algorithm, but to give an impression of how terribly this gets out of hand, we did a basic exponential fit on the running times. We concluded that it would require more years than the age of the universe to find c_{57} with this algorithm.

2.3 Complexity of the Algorithm

A large running time was to be expected for a simple reason: every single SAW gets counted, and there are many SAWs. Even if we were to write an algorithm that was amazingly efficient in avoiding collisions, and miraculously only counted successful already completed self-avoiding walks, the running times still explode. This is all due to the fact that those running times grow (at least) as self-avoiding walks themselves, exponentially depending on the connective constant $\mu \approx 2.638$. We could at this moment already say that the algorithm runs in $O(\mu^n)$ time. Because $T(n)$ was defined as λ^n, we now have $\lambda \approx \mu$.
One might initially believe that much more work is done than just what is required for counting $O(\mu^n)$ finished walks, but we can easily show that this is nonetheless a reasonable approximation of the running time. The remainder of this subsection can be skipped if the reader is comfortable with the notion that $\lambda \approx \mu \approx 2.638$ for this algorithm.

For the actual analysis we first introduce t_n, the number of walks of length n that intersect with themselves on their last move. (The t stands for tadpole, resembling the shape of such a walk, as it is a combination of a self-avoiding polygon[7](the

[7]a self-avoiding polygon is a self-avoiding walk that begins and ends at neighboring lattice points, allowing us to make a full cycle

body) and a self-avoiding walk (the tail) connected to it.)
Imagine standing at the end of a walk that successfully avoided itself for $n-1$ moves, about to make the nth move. Since there are four possible directions to go in, the total number of outcomes due to the nth move is $4 \cdot c_{n-1}$. All those outcomes can either be tadpoles of length n, or self-avoiding walks of length n. It follows that, for $n > 1$:

$$4 \cdot c_{n-1} = t_n + c_n. \tag{2.1}$$

Now what do we run into when the recursive method is called upon? We can either hit ourselves, so form a tadpole of length $\leq n$ or we can continue walking, and form a self-avoiding walk of length $\leq n$. We therefore perform in total $O(1)$ operations for every tadpole and self-avoiding walk of length $\leq n$. The running time, with the use of symmetry not yet taken into account, will approximately be:

$$\begin{aligned} & \textstyle\sum_{j=2}^{n}(c_j + t_j) \\ = \quad & 4\textstyle\sum_{j=2}^{n} c_{j-1} \\ = \quad & 4\textstyle\sum_{j=1}^{n-1} c_j \\ \approx \quad & 4\textstyle\sum_{j=1}^{n-1} \mu^j \\ = \quad & 4(\mu^n - \mu)/(\mu - 1) \\ = \quad & O(\mu^n). \end{aligned}$$

We approximated c_j by its exponential contribution μ^j, assuming large j, and in the last step we neglected the $4/(\mu-1)$ term to give a running time of $O(\mu^n)$. Note that this last term $4/(\mu-1) \approx 2.38$ can actually be considered as the factor representing the 'extra work' that needs to be done due to all the steps that are not directly counting self-avoiding walks. Clearly the intuitive first guess was not far off. Note further that if we would take symmetry into account, this will just contribute by a factor of $1/8$. Approximating this as $1/\mu^2$ this can actually be seen to only increase our reachable length n by 2.
Obviously the main approximation error comes from ignoring the polynomial contribution of c_j. For now the most practical conclusion is that we need to beat $O(\mu^n)$ running times.

3 The Length-Doubling Method

In this chapter we describe another method for counting self-avoiding walks, this time focusing more on the mathematics and less on the precise implementation of the algorithm.

3.1 Introduction

One of the methods successfully beating the $O(\mu^n)$ running time of the simple recursive algorithm, is the Length-Doubling method developed by Raoul D. Schram, Gerard T. Barkema, and Rob H. Bisseling [12]. None of the following ideas are used for our own method, neither is the material discussed here a prerequisite for understanding it. However, as the length-doubling method reaches a decent $\lambda = 2.32$ in the square lattice, and, more importantly, it is the record-holder for lattices with dimensions > 2, we do believe it is worth shortly describing the mathematical derivation of this method. Details of the exact implementation can be found in [13].

3.2 Mathematical Derivation

As the name of the method might have already suggested, we will try to find the number of walks c_{2n}, assuming that we already know, and are able to reproduce, the walks of length n.
The first observation is that any two SAWs of length n can either combine to form a SAW of length $2n$, or intersect each other. Let A_i be the set of pairs of SAWs of length n that both pass through a certain lattice point i. Since any pair of walks that intersect each other must have (at least) one lattice point in common, such a pair belongs to the set $\bigcup_i A_i$. (Here i is understood as defining any point in the finite lattice reachable by walks of length n). Also, any pair of walks belonging to $\bigcup_i A_i$ belongs to at least one of the A_i and has at least one lattice point in common, therefore representing two intersecting walks. It follows that the number of intersecting pairs of walks is precisely $|\bigcup_i A_i|$. Since the total number of pairs of walks of length n is given by c_n^2 (we do not remove the possibility of walks 'pairing up with themselves') we find the following formula:

$$c_{2n} = c_n^2 - |\bigcup_i A_i|. \tag{3.1}$$

As we assumed that c_n is known, we are now interested in the term $-|\bigcup_i A_i|$. An important note is that $|A_i|$ gives the number of pairs of walks that intersect each other in lattice point i, but does not say anything about them possibly also

intersecting in another point j. Removing $|A_i|$ for all i (from the total c_n^2) will therefore be too much, since pairs of walks that intersect in precisely two points (for example) will have been removed twice, instead of once. We can correct for this by adding $|A_i \cap A_j|$ for all $i < j$, representing the pairs of walks that intersect each other in two (or more) points. With the same reasoning applied to other terms (i.e. now correcting for pairs of walks intersecting in 3 points by adding a term, etc.) the so-called inclusion-exclusion principle is found. More generally the inclusion-exclusion principle states that for n sets $A_1, A_2, \cdots, A_n$:

$$\begin{aligned} |\bigcup_{i=1}^{n} A_i| &= \sum_i |A_i| - \sum_{i<j} |A_i \cap A_j| + \sum_{i<j<k} |A_i \cap A_j \cap A_k| + \cdots \\ &+ (-1)^{n+1} |A_1 \cap A_2 \cdots \cap A_n|. \end{aligned} \tag{3.2}$$

We are interested in terms of the form $|A_{i_1} \cap A_{i_2} \cdots \cap A_{i_r}|$, where $\{i_1, \cdots, i_r\}$ is a non-empty subset of all the lattice points. We define $c_n(S)$ as the number of SAWs of length n that pass through all the sites of some subset $S = \{i_1, \cdots, i_r\}$. Any combination of two such walks is an element of $A_{i_1} \cap A_{i_2} \cdots \cap A_{i_r}$, whereas every element of $A_{i_1} \cap A_{i_2} \cdots \cap A_{i_r}$ is a combination of two such walks. We conclude that $c_n^2(S) = |A_{i_1} \cap A_{i_2} \cdots \cap A_{i_r}|$. Note also that the sign of such a term in equation 3.2 depends only on $r = |S|$.

Closer inspection of equation 3.2 reveals that every relevant (reachable by at least one walk of length n) non-empty subset S of the lattice appears exactly one time in the sums on the right-hand side. (And again, every single term on the right-hand side can be uniquely linked to a specific subset S.) It follows that we might as well sum over the subsets S. Using the fact that $c_n^2(S)$ is the size of each term, and that the sign of each term depends only on the size of the set $|S|$, we achieve the following formula for c_{2n}:

$$c_{2n} = c_n^2 + \sum_{S \neq \emptyset} (-1)^{|S|} c_n^2(S). \tag{3.3}$$

3.3 Complexity of the Algorithm

To find $c_n(S)$ we generate each SAW of length n. Each SAW of length n contributes to 2^n different subsets S. Assuming $O(1)$ time to increment the counter of a specific subset, we will know all $c_n(S)$ in $O(2^n c_n)$ time, or $O(2^n \mu^n)$, approximating c_n by its exponential behavior for large n. Defining the running time as $T(n) = O(\lambda^n)$ we find that $\lambda^n = 2^{n/2} \mu^{n/2} = (\sqrt{2\mu})^n$, so that $\lambda = \sqrt{2\mu} \approx 2.32$ for this algorithm.

4 The Finite Lattice Method

In this chapter we describe the method that forms the basis of what is currently the best known method for counting self-avoiding walks on the square lattice.

4.1 Introduction

The name 'Finite Lattice Method' (FLM) is based on the notion that SAWs are now considered embedded in finite rectangular graphs, rather than free to span in all directions. The essence of the Finite Lattice Method lies in the so called transfer matrix (TM) method[8], used for counting the number of SAWs in those rectangular graphs. It shall be described in great detail in section 4.3. The TM method has also inspired the development of our own new recursive method, which will be described in chapter 5. A slightly more complex variant of the TM method is actually incorporated in our algorithm.

The Finite Lattice Method was first used in 1980 by Ian Enting [20] to enumerate polygons up to length 38. His pioneering work, extended to enumerating SAWs by Conway, Enting and Guttman [19] in 1993, forms the basis of the so-called 'Pruning Method' with which the record of calculating c_{79} has been reached by Iwan Jensen in 2013 [17].
For completeness, as it still holds the record, we summarize how this pruning method works in section 4.4. However, because our own method does not use any features of the pruning method, the focus of this chapter lies on the more general transfer matrix method.
In the remaining chapters of this thesis generating functions play an important role. If the reader is unfamiliar with those functions, we recommend first reading appendix A.

[8]The name 'transfer matrix method' referring to the way self-avoiding walks (or polygons) are counted, is inspired by the more widely known transfer-matrix method in statistical mechanics. The latter is used when a physical system can be broken into a sequence of subsystems that interact only with adjacent subsystems. Although the use of this name shall therefore shortly be understood, we prefer to consider the two methods as totally different. For instance, the transfer matrix method used for counting self-avoiding walks has nothing to do with the mathematical concept of a matrix.

4.2 Walks in an $L \times H$ Rectangle

We will set aside c_n for a moment, and consider first the question of how many SAWs there are in an $L \times H$ rectangle. By this we mean a rectangular graph, of L edges long and H edges high, or equivalently $L+1$ vertices long and $H+1$ vertices high. Consider for example Figure 4.1, where a SAW is shown, part of it as a dashed line, in a 8×6 rectangle. We don't specify an origin: SAWs can begin (and end) everywhere as long as they remain within the rectangle. We do assume, however, that each path visits both the left-most side and the right-most side of the rectangle, so that it spans at least L edges. Due to symmetry we only have to consider rectangles with $H \leq L$, and if we are only interested in paths of length n, we can add the restriction that $L \leq n$.
Let $R_{L,H}(n)$ be the number of SAW's of length n in a rectangle of size $L \times H$ touching both the left and the right side of the rectangle. Let $R^*_{L,H}(n)$ specify that we consider paths that fully span the rectangles, i.e. they also touch the bottom and top of the rectangle. Then:

$$R^*_{L,H}(n) = R_{L,H}(n) - 2R_{L,H-1} + R_{L,H-2}(n). \tag{4.1}$$

We remove $R_{L,H-1}$ twice because of the two ways that such a rectangle fits in the bigger $L \times H$ rectangle. We need to add $R_{L,H-2}$ to correct for the 'middle' part of the $L \times H$ rectangle, which is removed twice but should have only been removed once. If we define $R_{L,H} = 0$ for negative H, the above formula works for $H \geq 0$.
After having added the restriction that SAW's fully span the rectangle, so when working with R^* instead of R, the shortest possible SAW fitting in the rectangle will be of length $L+H$, so we only have to consider rectangles with $L+H \leq n$. Because every SAW can be placed in one unique 'smallest' rectangle (meaning that it fully spans this rectangle), the following formula can be derived for c_n:

$$c_n = 2 \sum_{\substack{H+L \leq n \\ L > H}} R^*_{L,H}(n) + \sum_{2H \leq n} R^*_{H,H}(n) \tag{4.2}$$

The first term on the right-hand side includes a factor 2, due to the fact that rotating these rectangles by 90 degrees will result in another rectangle. (Rotating even further or flipping the rectangle does not lead to different walks, because these symmetries are already accounted for within the rectangle itself.) For squares this is obviously not the case, hence the second term is taken out of the sum. Based on equations 4.1 and 4.2 we conclude that it is sufficient to know all $R_{L,H}(n)$ for $H \leq L \leq n$ and $L + H \leq n$.

4.3 The Transfer Matrix Method

The transfer matrix method is a way of computing the number of SAWs spanning a given rectangle. A vertical 'cut-line' is created, intersecting the rectangle (and possible SAWs within it). This cut-line is gradually moved from left to right, until the full $L \times H$ rectangle has been crossed. By carefully deriving the number of different ways in which such a cut-line intersects self-avoiding walks, a form of dynamic programming becomes applicable, where old configurations of the problem contribute to newer configurations (in which the cut-line has moved). In this process all the values $R_{l,H}$ for $1 \leq l \leq L$ will be counted.

4.3.1 Moving through the Rectangle - The Cut-line

What the cut-line of a transfer matrix calculation looks like, and what information it contains, can be best explained with a picture. Below we consider, as an example, a SAW embedded in a 8×6 rectangle. We have intersected this rectangle at two places. The cut-line, moving from left to right, does not go through vertices of the lattice, but can rather be seen as a set of $H + 1$ (horizontal) edges that are intersected.

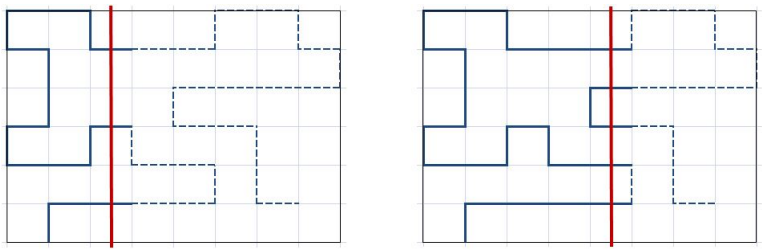


Figure 4.1: A self-avoiding walk embedded in a 8×6 rectangle, intersected in two ways by a red cut-line.

We first look at the picture on the left. Imagining the SAW literally 'cut' by the cut-line, it is now broken into several pieces, SAW segments if you will, to the left and right of this red cut-line. We will only focus on what the segments on the left of the cut-line look like. We can classify these segments in two ways:

- If a segment of the SAW has two of its edges intersecting the cut-line, such a segment is called an 'arc'. Consider again Figure 4.1 where an arc is shown in the top left of the first picture.
- If a segment of the SAW has only one edge intersecting the cut-line, it must necessarily still be connected to one of the endpoints of the SAW (making

no difference between ending and origin of the SAW at this moment). We will call such a segment a 'free end'. Naturally there can be at most two free ends given any cut-line.

We will now describe the entire cut-line, according to the edges that it intersects. These edges can be part of the SAW, or they can be 'empty' edges, meaning that the SAW does not pass through those edges. For the edges that are part of the SAW, we make a further distinction based on whether they are part of an arc or a free end. We give the edges connected to the free ends the label '3', calling them 'free edges', and the edges connected to an arc label '1' or '2', depending on which one is higher, '1' being the lower edge. Labeling empty edges intersected by the cut-line '0', we can now describe the entire red cut-line (from bottom to top) as: (0301020), looking at the left picture of Figure 4.1.
More formally, any configuration along the cut-line is represented by a set of edge states, a signature if you will, $S = \{\sigma_i\}$ (with $i = 1$ representing the bottom edge and $i = H + 1$ representing the top edge), where:

$$\sigma_i = \begin{cases} 0 & \text{empty edge} \\ 1 & \text{lower edge (of an arc)} \\ 2 & \text{upper edge (of an arc)} \\ 3 & \text{free edge} \end{cases} \tag{4.3}$$

As an extra example, consider the picture on the right, after the cut-line has moved to the right twice. This time the cut-line has signature $S = (0311220)$. [9]
We can now with reasonably short notation describe any intersection of the $L \times H$ rectangle for every possible SAW in it. How many possible signatures are there? A first guess might be 4^{H+1}, but this is not entirely true. With the notation just explained, many constraints are added to what a signature can look like.[10] We come back to the number of signatures, when discussing the complexity of this method. For now let us first explain how these signatures are going to be used.

Let us go back to the example in Figure 4.1. The left figure can be seen as depicting a possible way to 'complete' the left-hand side according to signature

[9] In the case of more (pairs of) '1's and '2's in one signature there are no ambiguities as to which lower arc ends and upper arc ends belong together to form one arc. Since crossings are not permitted, the labeling uniquely describes how the edges that are part of an arc (labels '1' or '2') are connected.

[10] An alternative notation, setting '(' as lower part of an arc and ')' as the top part of an arc, might be more illuminating, especially for those familiar with the 'bracket subsequence' problem in dynamic programming. With this notation the signatures of Figure 4.1 become 030(0)0 and 03(())0. Most importantly, it must be understood that a signature is only valid if it has at most two '3's, and if its brackets form a regular bracket sequence.

$S = (0301020)$, by occupying a total of 14 edges, excluding edges intersected by the cut-line.
Define $a_n(S)$ as the number of distinct possible ways of completing the left-hand side of (the cut-line belonging to) signature S occupying n edges. Now define G_S, the generating function for signature S, as $G_S(x) = \sum_{n=0} a_n(S)x^n$. This generating function contains all the required information about a certain signature. It tells us for each n how many ways there are to complete the signature on the left-hand side of the cut-line by occupying exactly n edges. Setting $x = 1$ would therefore result in summing all contributions, and thus gives the total number of ways of 'completing' the signature by explicitly constructing the corresponding walk segments of the left-hand side. Obviously it is more useful to separate these ways of completion based on the accumulated length of the segments, hence the generating functions have been introduced.

Clearly, moving the cut-line to the right alters all the generating functions, since longer segments are now possible to the left of the cut-line. However, and here comes the most important part of this method, we can still use the information of the previous collection of generating functions $\{G_{S_i}\}$ in building the new one $\{G'_{S_j}\}$. (When using indices i or j (≥ 1), we will implicitly assume that the signatures are neatly numbered in some way.) For every 'source' signature S_i we want to alter all generating functions of the 'target' signatures $\{S_j\}$ that it 'contributes' to.
The generating function of S_i, G_{S_i}, can be understood as 'contributing' to G'_{S_j}, if there are SAWs intersected by the cut-line according to signature S_i, which are intersected later on (after having moved the cut-line) according to signature S_j. Figure 4.2 shows that a source signature can contribute to (the generating functions of) many different target signatures:

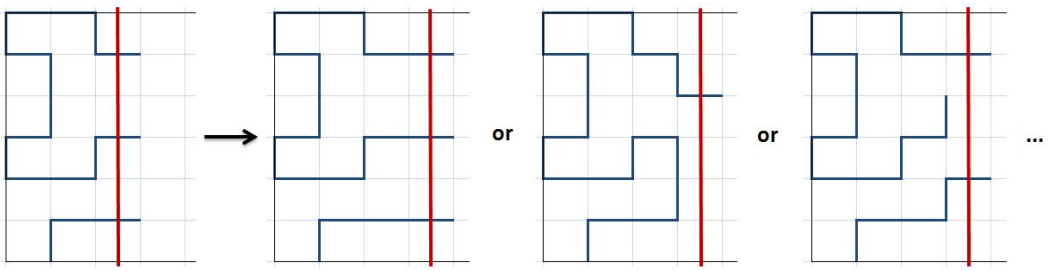


Figure 4.2: Segments of a possible SAW, intersected by the cut-line according to the 'source' signature (0301020), changing into different target signatures upon moving the cut-line. In the first case we end up with the same signature (0301020). The other possible 'target' signatures depicted are (0000300) and (0030030), but many more are possible.

Deriving for each source signature all possible target signatures is a costly (time-consuming) task. In view of the many signatures, the contributions of which must all be considered every time we move the cut-line, we would prefer to require $O(1)$ time finding the target signatures of each source signature. In order to make the switch from source to target signature more manageable, a different approach is considered than the one depicted in Figure 4.2. We do not make the step in one go, but alter the cut-line slightly. Consider the following picture.

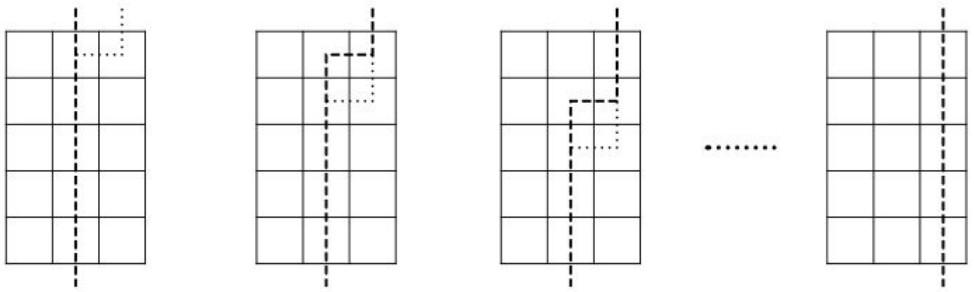


Figure 4.3: The process of moving down the kink in the cut-line in a so-called transfer matrix calculation. Each step can be seen as adding one vertex to the graph already treated on the left of the cut-line. This picture is taken from [17].

By considering the 'kinked' signatures in between, we can gradually move from one 'straight' signature to the next. Moving the kink down one step can be seen as adding one vertex to the part of the rectangular graph that we have so far already crossed. It follows that we only need to focus on the edges connected to this newly added vertex, and therefore only on two edge-labels in the signature. We just need to develop certain rules determining in what ways these two edge-labels of S can be changed into two new edge-labels of the next signature S' (where the kink is one step lower). All the logic mentioned earlier still applies, concerning the labels $\in \{0, 1, 2, 3\}$ and the generating functions. The only difference is that some signatures are now one edge-label longer since the cut-line is now intersecting $H + 2$ edges.[11]

We end this section with a few important remarks. Firstly, when moving the kink down one step, we compute all the generating functions belonging to the target signatures, doing so by adding the contributions of all relevant source signatures. Throwing away the old generating functions, and continuing with the new ones, we

[11]The signatures forming straight lines, i.e. when the kink has been moved through an entire column, completing an entire shift to the right, technically only contain $H + 1$ labels, but can also be viewed as having $H + 2$ labels of which the first (at the start of going through a column) or the last (at the end of going through a column) must be a '0'.

can view this operation as 'updating' the generating function of every signature. Secondly, although we refer to source signatures as 'contributing' to target signatures, it is of course the source generating functions that contribute to target generating functions. A detail concerning the implementation of the algorithm is that actually for every signature two generating functions are stored: an old (source) and a new (target) generating function (the latter written as G'). However, we usually refer to (source and target) signatures rather then generating functions, implicitly considering them connected.
Finally, note that the same signature can describe differently shaped cut-lines, since not only the labels in it matter, but also where the kink is located. A different notation, used in Figure 4.4, is therefore to over-line the labels of a signature corresponding to those two edges of the cut-line that are part of the 'kink' and will be altered the next time the kink moves down.

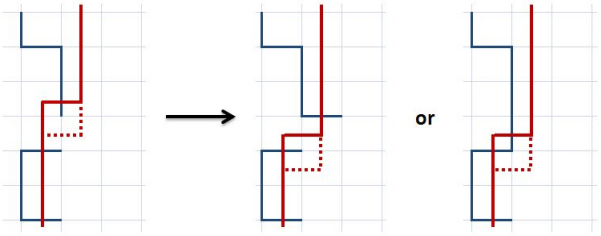


Figure 4.4: The signature on the left can be described as $(102\overline{0}3000)$. The labels corresponding to edges in the square formed by the kink and the dashed line (representing where the kink is headed towards) are over-lined. The signature can contribute to the two signatures on the right, which are $(10\overline{20}3000)$ and $(10\overline{2}\overline{3}0000)$.

Note that the two options depicted in Figure 4.4 to which the signature $(102\overline{0}3000)$ might contribute, are actually the only two valid options. Comparing this with Figure 4.2, we see how effective this kink-moving method is in making the moving of the cut-line through the rectangle more manageable.

4.3.2 Moving the Kink - The Updating Rules

During the action of moving the kink down one step, we take care of the following:

1. At every move we look for self-avoiding walks that could now be seen as completed (by adding no more edges). Consider for example the signature $(000\overline{3}\overline{0}000)$, which contributes to $(00\overline{0}\overline{0}0000)$. The generating function for the latter signature will in this case be no longer updated later on. We rather see it as containing newly found finished SAWs (of different lengths). Considering only the desired length, and say we are working through the $(l+1)$th column of vertices, we add these self-avoiding walks to the running total for $R_{l,H}$.

2. A weight x is attributed to each occupied edge. This is essential for the way we are handling the generating functions. Whenever we say that a source signature contributes to a target signature, it actually does so by adding potentially a few edges to the walk segments on the left of the cut-line. Only those edges no longer intersected by the cut-line are counted. Figure 4.4 shows that the total number of SAW-edges to the left of the cut-line increases from 5 to 6 after moving the kink.
   For each source signature S_i, we find what target signatures $\{S_j\}$ it contributes to and the number k_i (= 0, 1 or 2) of edges being added by moving the kink. We then add $x^{k_i} \cdot G_{S_i}$ to G'_{S_j}.

3. Each SAW must span the rectangle from left to right. Due to the way we add SAWs to the running total, it only remains to make sure that the SAW actually begins at the left of the rectangle. We cannot have a purely empty source signature (000000) contributing to target signatures when we start looking at the second column of vertices.

4. After going through all source signatures, updating the target signature generating functions for each of those source signatures, we now name the correctly computed target signatures as the new source signatures, throwing away the old ones. The kink can now be moved down, and a new updating operation can begin.

Given a source signature it all comes down to deriving what target signatures it contributes to. As mentioned, changing the straight cut-line to a kinked cut-line ensures that we only have to focus on a few edges, as illustrated in Figure 4.5. Considering each step of the kink as adding one vertex to the graph on the left of the cut-line, we will be concerned with the edges connected to that newly added vertex. The edges a and b, still part of the old signature, will be referred to as 'input' edges, while c and d will be referred to as 'output' edges.

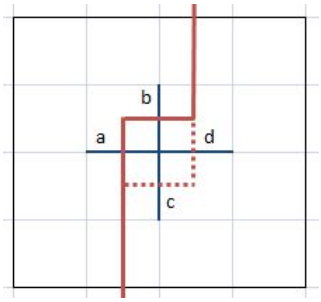


Figure 4.5: The updating process of moving the kink down one step. The edges a and b, of which the labels are part of the source signatures, are referred to as input edges. The output edges c and d contribute to the target signature. Its corresponding part of the cut-line is shown as a dotted red line.

We now have to look for all different outputs (cd) given a certain input (ab). When looking for possible outputs we keep in mind the following constraints:

1. We must make sure we keep dealing with self-avoiding segments. We do so by never letting a vertex that has just been added (by a move of the kink) be part of more than two occupied edges. In Figure 4.5 this means that if edge a or b is occupied by a SAW segment, there can be at most one occupied edge exiting it on the other side: either c or d has to remain empty (or both).

2. We must end up with a single connected component. This will be done by never letting a segment 'disconnect' itself from the cutting-line with other segments still around. As illustrated by Figure 4.4 the signature (102$\overline{0}$3000) cannot be allowed to contribute to (10$\overline{2}\overline{0}$0000).

3. There can be at most two end points of the SAW already in place. This will be realized by carefully monitoring the amount of '3's in our signatures. As a quick example, consider a signature (0030303), which can never be connected so as to form a single SAW, since already three endpoints are given.

There are many different situations we have to consider, depending on what the incoming edge labels are. For completeness we shortly describe how each input is dealt with. Readers that are not interested in further implementation details of the TM method, can safely skip the rest of this section.

We have split the updating rules according to the labels of input edges a and b, using the notation 'ab':

- 00: There are four possible outputs. We can leave c and d empty (00), start a new arc (12), or start a new free segment (03), (30). A new free segment

can only be added if the source signature has at most one free end (edge labeled '3').

- 01,10,02,20: We are dealing with one end (one edge on the cut-line) of an arc. We can either continue this arc, that starts or ends with a or b, or we can leave both output edges empty. Note that if we choose to leave both output edges empty, we are no longer dealing with an arc, but we rather have created a free segment. The edge on the cut-line that was previously the other (lower or higher) part of this arc, must now be given the label '3'. This latter option of leaving both output edges empty, because it creates a new free segment, is only possible if the source signature has at most one free end.

- 03,30: We can either continue the free end (through c or through d) or we can leave both output edges empty. The latter option will create an entirely separate component and is therefore only allowed if the resulting graph contains one single valid SAW. In this case, the partial generating function is added to the running total. Note that we can choose to only do this in the last column, so that walks that do not fully span the rectangle from left to right are not added.

- 11,22: The output must be empty, otherwise an intersection is caused because three edges are touching the same lattice point. Also the matching upper (lower) edge of the inner-most arc is relabeled as the new lower (upper) edge of the combined arc.

- 12: The lower and upper edge of a single arc are joined, forming a closed loop. This is not allowed.

- 21: Upper and lower edges of two different arcs are joined and the output must be empty. The combined arc created by joining these edges does not need further alterations: The upper edge of the combined arc was previously an upper edge (label '2'), and the lower edge of the combined arc was previously a lower edge (label '1').

- 13,31,23,32: A free end is connected to an arc, through the lower (or upper) arc edge on the cut-line. The upper (or lower) arc edge on the cut-line becomes a free edge, and must therefore be relabeled. The output edges must be empty.

- 33: Two free ends are joined. If the result is a single valid SAW, we add this generating function to the running total. Otherwise an invalid combination of multiple segments is created, and we can ignore it.

4.3.3 Complexity - The Number of Signatures

We are mainly interested in the exponential behavior of the running time, for which only the number of signatures is relevant. Every time we move the kink down one step, we go through all (source) signatures.
Assume that the number of signatures describing m edges (containing m labels) is given by $S(m)$. When introducing the signatures, we explained that the number of signatures must certainly be less than 4^m, because of some constraints. We shall now go into this in more detail.

Let $A(1)_i$ be the accumulated number of '1's in a signature up to and including the ith label (where $i = 1$ at the bottom and $i = H + 2$ at the top of the cut-line)and $A(2)_i$ similarly defined for the number of '2's. We can now state that a signature is valid only if it satisfies the following constraints:

1. There can be at most 2 free edges ('3's) in a signature
2. For every $0 < i < H + 2$ we must have $A(1)_i \leq A(2)_i$. This is because otherwise there is at least one label '2' edge that cannot be linked to a label '1' under it.
3. The total number of 1's and 2's must be equal: $A(1)_{H+2} = A(2)_{H+2}$.

For a signature belonging to a straight cut-line we can exactly calculate the total number of possible combinations following the above constraints, let's call this $S^*(m)$, the star denoting that we are dealing with a straight line, considering a signature containing m edge-labels. We can first choose either 0, 1 or 2 free ends (label '3') to be placed on the signature, leaving us with a signature of length m, $m - 1$ or $m - 2$ respectively, that does not contain edges with label '3'. Let $f(m)$ denote the number of possible signatures with length m, only consisting of arc segments or empty places (so only containing labels '0', '1' or '2'). The total number of signatures $S^*(m)$, corresponding to straight cut-lines, is given by:

$$S^*(m) = f(m) + m \cdot f(m-1) + \frac{1}{2} m \cdot (m-1) f(m-2). \tag{4.4}$$

Now what is $f(m)$?[12] To derive this we first define the following generating function, assuming $f(0) = 1$:

$$F(t) = \sum_{m=0}^{\infty} f(m) t^m. \tag{4.5}$$

[12]Some readers might recognize $f(m)$ as the number of Motzkin paths of length m. Map '0' to a horizontal step, '1' to a north-east step, and '2' to a south-east step. For instance, Motzkin paths are not allowed to dip below the $y = 0$ axis, which can be recognized as the second constraint above, $A(1)_i \leq A(2)_i$. For those that are not familiar with Motzkin paths we will do the derivation here in terms of the problem of signatures we are currently dealing with.

Now every signature begins either with a '0' or with a '1'. If we begin with a '0' we can follow up with a complete signature to formulate a valid signature. If we begin with a '1' we can follow up with a complete signature after which at some time we need a '2', and we can add another signature to the end. This can also be the 'empty' signature, with $f(0) = 1$. It follows that:

$$F(t) \;=\; 1 + tF(t) + tF(t)tF(t),$$

which can be solved to give:

$$F(t) \;=\; \frac{1 - t \pm (1 - 2t - 3t^2)^{1/2}}{2t^2}. \tag{4.6}$$

The number of possible signatures (with no labels '3' and corresponding to a straight cut-line) of length m, $f(m)$, can be derived from this generating function, and in reference [21] it is found that $f(m)$ goes like $O(3^m)$. This in turn would imply, considering equation 4.4, that $S^*(m)$ goes like $O(m^2 3^m)$. Neglecting the polynomial contribution $O(3^m)$ remains.
Returning to $S(m)$, the number of signatures with a kinked cut-line, we know that it is smaller than $S^*(m)$, since there are fewer possibilities in the neighborhood of the kink. However, $S(m)$ is larger than $S^*(m-1)$, because every signature with a kinked cut-line and the vertical edge (intersected by the horizontal part of the cut-line) having label 0 is also a valid signature of a straight cut-line, when this label 0 edge is taken out. We therefore find that $S(m)$ is bounded in the following way:

$$S^*(m-1) \leq S(m) \leq S^*(m), \tag{4.7}$$

so that we can conclude that $S(m)$ also grows asymptotically like $O(3^m)$.
Calculating all rectangles with heights $\leq H$ gives the SAWs up to $n = 2H$. If we only consider the highest (most time-consuming) rectangle, and note that its height is $O(n/2)$ we arrive at a running time for this algorithm of order $O(3^{n/2})$. It follows that the running time $T(n)$ goes like $T(n) = \lambda^n = 3^{n/2}$, resulting in $\lambda = \sqrt{3} \approx 1.73$.

This is even better than the running time for the length-doubling method, where $\lambda \approx \sqrt{2\mu} \approx 2.32$. However, the transfer matrix method does not do a good job in higher dimensional lattices, like the cubic lattice. We would have to go through 'sheets' of cuboids, and in each of those sheets through a single column of vertices. Most importantly, the number of signatures will be enormous.

4.4 The Pruning Method

The pruning method is an improvement of the transfer matrix method. The improvement (in running time as well as memory) comes from the smaller number of signatures used. Not all theoretically possible signatures of the TM method are being updated, because those signatures requiring too many edges upon completing them are 'pruned away'.

4.4.1 Introduction

As discussed, it is mainly the great number of signatures that slows down the transfer matrix method, since every signature holds a generating function that must be updated each time the kink moves. What's more, the higher the rectangle we are working with, the greater the number of signatures, increasing as 3^H.
Consider now looking for $R^*_{L,H}(n)$, the number of walks of length n that fully span a rectangle of length L and height H, thus also touching the bottom and top row of vertices. Now here comes the crux: the higher the rectangles, the fewer signatures actually contribute to valid (fully spanning the rectangle) SAWs of length n, since completing these signatures might result in walks exceeding the length n we are interested in.
This can be most easily understood by considering rectangles with H close to the maximum height $\approx n/2$. A valid SAW in such a rectangle has to be an almost staircase like line in order to touch all sides and still not exceed length n, going from one corner of the rectangle to the opposite corner. Therefore, already at the left-most column of vertices of the rectangle, many signatures are actually irrelevant, as they are certainly going to exceed the length n after the shortest possible completion of them to form a valid SAW. Note that not only do the SAW segments have to touch the bottom and top of the rectangle, also all loose ends have to be connected to form one component.
If we could somehow find a way to timely throw away the irrelevant signatures (or to not even have to consider them at the start of the TM method[13]), thus not having to update them constantly only to find out that they become too large at the end, it would result in a major improvement to the running times.

4.4.2 Different Signatures and Updating Rules

The pruning method was first developed by Iwan Jensen and Anthony Guttmann [15] in 1999, used for enumerating polygons up to length 90, and extended to counting SAWs in 2004, reaching length 71 [18]. Although those first methods generally

[13]The pruning of signatures is not efficient if at the start of this method all 3^H signatures are saved nonetheless. Instead a signature is only added to the set of signatures to be updated, when it is first encountered during the TM method as a possible target signature.

used the same signatures and updating rules as the original TM method, the latest improvement, reaching the record length of 79, is different in one major respect: the labels on the signatures now refer to the right side instead of the left side of the cut-line. Knowing how SAW segments are connected (i.e. forming arcs or free ends) on the right-hand side makes it easier to calculate what the shortest way is to create a valid finished SAW out of a given signature. This way we can more quickly decide whether a signature should be pruned away or not.

Having signatures now describing the future development of the SAW, rather than the past development, the updating rules at moves of the kink are drastically altered. We only mention a few striking differences. Again these implementation details can be skipped if the reader is only interested in the bigger picture of the pruning method. We will once more consider the relevant edges as input and exit edges as introduced in Figure 4.3, section 4.3.2.

- If the input edges are both occupied, they immediately meet each other after passing the cut-line. Since we are describing the future behavior of the SAW at the right-hand side of the cut-line, the only possible way for both input edges to be occupied is by having labels (12), as they form an arc right-away.
- If one of the input edges has label '1' or '2', we can no longer decide to discontinue this SAW segment. Because the label refers to how this segment will later be connected, it *must* be continued along one of the output edges.
- If both input edges are empty, labeled '0', and the output edges are not left empty, we must be careful: by occupying one or both of the output edges, we are creating a new SAW segment that must be linked to the rest of the signature so as not to form an unconnected component. Upon creating such new occupied edges we must immediately describe how they become part of the already defined SAW segments, and relabeling might be needed.

4.4.3 The Actual Pruning of Signatures

Now that the edge-labels refer to the right-hand side of the cut-line, we can easily see how the edges are supposed to be connected on the right. With this information we can calculate the minimum number of steps required at the right of the cut-line to complete a given signature and produce a valid walk. We shall define this as N_R, and define N_L as the minimum number of edges that succeed in building SAW segments at the *left* of the cut-line to result in a certain signature. We consider the edges currently intersected by the cut-line as belonging to N_R.

We already know the minimum number of occupied edges N_L that is needed at the *left* of the cut-line in order to build SAW segments resulting in a certain signature, since that is just the lowest nonzero order in the generating function.[14] Considering n as the length of the SAWs we're after, the pruning method now comes down to one essential action: checking if $N_R + N_L > n$ for each signature. The signatures (or rather generating functions, see footnote 14) for which this is the case can be discarded.

To determine N_R we need to focus on three contributions: the number of steps required to connect all occupied edges into a single component, the number of steps (if any) needed to ensure that the top and bottom of the rectangle are visited, and the number of steps needed to ensure that the walk spans at least H edges in length-wise direction. [15]

We shall not treat the way in which N_R is calculated in detail, as it quite tedious and mainly involves dealing with many small exceptions. To give an impression of how N_R is calculated we first note it involves the "nesting level" of arcs in the signature. The nesting level is a way of describing how many arcs or free edges are within a certain arc, that this outer arc has to loop around.
Having found the nesting levels of arcs in a given signature, we first find the shortest way to complete those arcs to form one component. What remains is an optimization problem where the shortest path is searched for multiple candidate SAW segments to satisfy the other constraints (e.g. touching top and bottom of the rectangle). The optimization problem does not have to be very hard: in most cases it is clear what candidate SAW segments are chosen and the only work lies in calculating how short the shortest path actually is.

[14] Actually four different generating functions are now maintained for each signature, having made a distinction between four cases: the segments on the left of the cut-line have already touched the bottom of the rectangle, or the top, or both, or neither. This distinction clarifies what constraints still need to be satisfied at the right of the cut-line in order to make a valid SAW. Having four generating functions slightly alters the updating rules, but these alterations include nothing complicated that we found worth mentioning.

[15] Remember that we are actually computing $R_{H,l}$ for all $l \leq L$ in one go. Hence it is not required for a SAW to reach the right-hand side of a rectangle, since we are interested in the 'shorter' rectangles as well. However due to symmetry, as explained in section 4.3.1, we only have to consider rectangles with lengths $l \geq H$. We therefore alter the restriction that the walk must fully span the rectangle, and rather use the restriction that the walk must at least span H edges in length.

4.4.4 Complexity

The updating rules for the pruning method are slightly more complicated than those for the original TM approach. On average, however, these contributions are negligible, not just because we are dealing with polynomial contributions to the running time, but also because the signatures requiring a lot of time to be updated are precisely the ones more quickly pruned away. These are usually signatures with occupied (nonzero) labels at great distances from each-other.
We now confine ourselves to the exponential behavior of the running time, still only determined by the number of signatures. The growth rate of the number of signatures (or rather their different generating functions, see footnote 14) is too difficult to derive exactly. A different approach is chosen: While running the algorithm the maximal number of signatures generated for different values of n is measured. By inspecting this number the growth rate can be estimated.

The first pruning method (developed for SAWs in [18]), which had signatures describing the left-side of the cut-line, was found to have a running time $T(n) = \lambda^n$ with $\lambda \approx 1.334$. In this approach pruning was very complicated, since finding the (shortest) connection pattern to the right of the cut-line meant having to search through all possible ways of connecting occupied edges.

In [17] the old approach is compared with the new approach, where labels describe the right-hand side of the cut-line. The required number of signatures goes down very slightly, because the pruning is more precise in the new algorithm.
A new estimate for λ has not been given, maybe because the decrease in the number of signatures was not considered sufficiently high to have a major influence on the exponential behavior of the running time. However, due to much more faster pruning the polynomial contribution to the running time has substantially decreased. The total CPU time[16] decreases by about 70% for $n = 61$ compared to the old pruning method.

For those interested in the result of this method: using up to 400 processors, and up to 1TB of memory, it took a total of about 16500 CPU hours to enumerate SAWs up to length 79.
The self-avoiding walk depicted on the title-page has length 79. With the pruning method it was found that there are a total of:
$10, 194, 710, 293, 557, 466, 193, 787, 900, 071, 923, 676$ of such SAWs.

[16]Since a parallel algorithm is used, the work is divided over many processors. The notion of 'total CPU time' is used to describe the sum of CPU time consumed by all of the CPUs utilized.

5 The Tile Hopping Method - A New Recursive Algorithm

In this chapter we describe our own method for counting self-avoiding walks. Although we begin with the mathematical concepts forming the bigger picture, the focus of this chapter lies more on the actual implementation of the algorithm.

5.1 Introduction

The method is inspired by the Finite Lattice Method, but is more similar to the recursive algorithm described in chapter 2. Having the entire lattice divided into $M \times M$ squares, this method is actually a generalization of the simple recursive algorithm, where M would be equal to 1.
The important feature learned from treating the FLM, is that for a given square sublattice a unique signature can be determined describing how such a square is entered and exited by SAW segments, and that a generating function can be linked to such a signature.

We give a global outline of this method in section 5.2. In the sections thereafter we describe in more detail the algorithm, i.e. the different objects and methods used, working towards the main recursive method. After a discussion of the complexity of the algorithm, we end with some words on possible future improvements.

5.2 Mathematical Derivation

We begin in the origin of the lattice, but now we intersect this entire lattice with multiple cut-lines. These cut-lines form a pattern of 'tiles' on the lattice, the size of the tiles depending on the distance between the cut-lines.

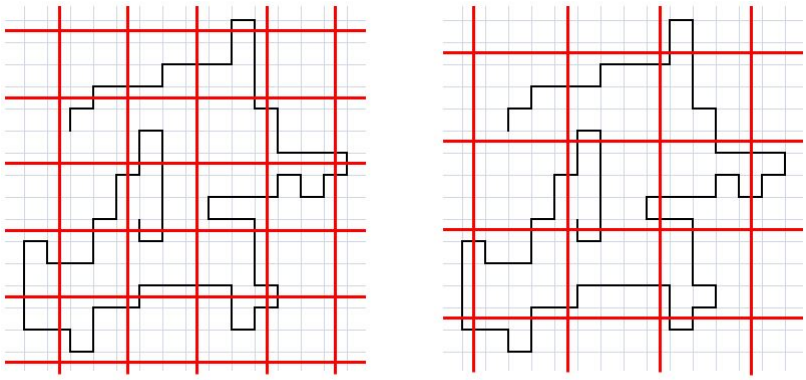


Figure 5.1: Tiling the lattice with square tiles of width 3 or 4. Any SAW can be represented in only one way, depending on how it intersects these tiles.

The tiles are placed such that their borders intersect edges. Their size is given by $M \times M$ internal vertices, with a circumference of $4M$ mid-edges being intersected. The old recursive algorithm worked by choosing one direction from {up, down, left, right} at each recursive step. In the new algorithm, at each step (after entering a tile), we choose one of the $4M - 1$ possible 'exit'-edges crossing the border, entering the next tile. We are thus constantly hopping between entry(/exit) points of the tiles, and we are never explicitly setting steps in the inner part of the tiles.

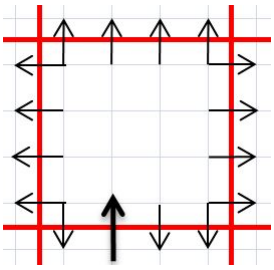


Figure 5.2: Upon entering an empty $M \times M$ tile, one of the $(4M - 1)$ exit options can be chosen to continue 'walking'.

Every tile is considered unique due to its location on the lattice. Furthermore, a tile T has a signature S_T that summarizes how (and if) the tile is being entered and exited through its $4M$ border-edges. To make sure that the signature at all times describes the previous crossings of that tile, it has to be changed whenever the same tile is crossed again. The signatures are discussed in the next section, section 5.3.

Each signature S has a generating function G_S that contains the number of ways in which a tile can receive signature S by being crossed (possibly multiple times) by a SAW. We also refer to this as the number of ways in which the tile is 'completed'.
More precisely $G_S(x) = \sum_{n=0} a_n(S)x^n$, with $a_n(S)$ the number of ways to complete a tile with signature S having placed n edges inside it. The border-edges (edges crossing into another tile) are not counted.

The main idea is now to 'walk' through the 'tiled' lattice hopping recursively from tile to tile, labeling each tile with a signature S as we pass it. Using again a recursion based on Depth-First Search techniques as in chapter 2, we now maintain a Grand Generating Function $\mathcal{G}$ that describes all possible SAWs of different lengths that correspond precisely to our trajectory of tile entries and exits. When a tile is crossed, we multiply this Grand Generating Function with the generating function

of the signature of that tile. Possibly the old generating function of a tile-signature has to be removed by division.
At each recursive step, before entering another tile, we check whether the lowest orders in the Grand Generating function are able to form valid SAWs after the next entry. If this is the case we deal with those first, before exiting to another tile. If the lowest orders have all become greater than n, we can stop.

More formally our trajectory is a specific configuration of (all) lattice-tiles with their respective signatures. It forms a 'Grand Signature' $\mathcal{S}$ that describes how the entire lattice (the set of all cut-lines forming the tiles) has been crossed, hopping from tile to tile. Every SAW is mapped to one and only one Grand Signature. The goal here is to find all possible Grand Signatures, and extract the number of SAWs of length n from them by looking at their Grand Generating functions $\mathcal{G}_{\mathcal{S}}$. Note that the bigger the tiles are, the fewer distinct Grand Signatures exist, and the more SAWs are grouped under a single Grand Signature.
We define a set of tiles and their respective signatures as 'valid' if there exists (at least) a SAW that would result in that specific combination of tiles and tile-signatures.
Let $\mathcal{T}$ be a valid set of all tiles on the lattice, and let $\mathcal{S}_{\mathcal{T}}$ be the Grand Signature of the lattice corresponding to that set. We define $\mathcal{C}(\mathcal{S})$ as the total number of occupied border-edges of a certain Grand Signature $\mathcal{S}$ ($\mathcal{C}$ stands for 'Crossings'). Remember that border-edges are not included in the generating functions of tiles. Then:

$$\mathcal{G}_{\mathcal{S}_{\mathcal{T}}}(x) = x^{\mathcal{C}(\mathcal{S}_{\mathcal{T}})} \prod_{S_t \,:\, t \in \mathcal{T}} G_{S_t}(x) \tag{5.1}$$

This function shows that we have in a way succeeded in a 'divide and conquer' approach of this problem. Having given a tile a signature, what happens inside it is completely independent of the rest of the lattice, the other tiles. We can thus focus on the tile-signatures rather than the Grand Signature $\mathcal{S}$, which we shall never need to formulate exactly. From now on we will refer to those tile-signatures just as 'signatures'.

5.3 The Signatures

The signature of a tile describes firstly which edges on the border of that tile are empty. Secondly, as regards the occupied edges, it must describe how these are mutually connected inside the tile. We label the edges clockwise, starting at the bottom left.
Having chosen this orientation of our tile, we now again use the labels {0,1,2,3}, describing an empty edge, the first part of an arc, the second part of an arc and a free end, respectively. In the case of an arc, we make sure that the label '1' always comes first in the signature, and that the label '2' is found by searching in clockwise direction.

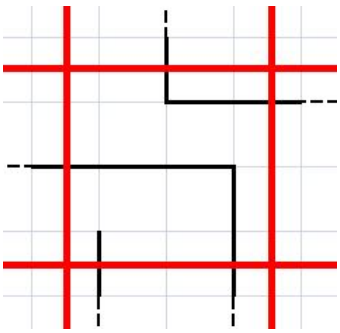


Figure 5.3: A tile being crossed multiple times. Its signature is given by (010010200203).

Signatures of a $M \times M$ tile have a length $4M$. Mapping the circumference of a tile onto a straight line of the same length, these signatures can be considered very similar to those used in the transfer matrix method for describing the cut-line. The number of signatures might be estimated by $S^*(4M)$ (see equation 4.4 in section 4.3.3). However, there are a few important differences:

1. Firstly, the corners take out some of the possibilities. If two corner edges (of the same corner) are occupied, they can only be (clockwise) labeled (12), or (21) if we are considering the corner at the bottom left.

2. Another difference is that, unlike in the case of a straight line, a tile can have so many arcs crossing it, that some border-edges must remain empty, labeled '0'. Consider for example Figure 5.4.

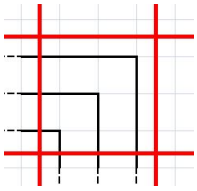


Figure 5.4: A tile with signature (111000000222). There can be no other way to connect the arcs in this tile than the way depicted. Hence any non-empty border-edge on the top or right side of the tile would result in an invalid signature.

3. Since a SAW has two ends, every tile can technically have a signature with up to two label '3' edges. However, it is actually only the tile with the origin in it that might have two 'free ends' in its signature.
   Now comes an important trick: We lay the tiles in such a way that the origin lies in the bottom left of a specific 'centre'-tile, and furthermore we let the first step be a step down, directly into the tile under it. This way we know that if a signature has two label '3's in it, one of those must be the last edge (bottom border, left-most edge). Figure 5.3 shows an example of a tile with the origin in it.
   Note also that we are using symmetry, since by deciding the direction of the first step we are counting only the SAWs corresponding to one of the four possible rotations of the lattice.

Elaborating some more on this last point, we note that there are in fact two types of signatures. There are those representing a tile without the origin in it, and those representing the 'centre' tile, with the origin in its bottom-left corner. It is important to make this distinction, because the free end at the bottom left of the tile is either (in the case of not being connected to the origin) free to become a SAW segment of any length within the tile, or it must be of length 0, immediately stopping at the origin. These two situations lead to very different generating functions, hence it is important to be able to tell them apart.

5.4 Creating the Generating Functions for the Signatures

As with the TM method, we have a cut-line moving through the square tile. However this time, instead of having a signature describing only that cut-line, it is actually the entire circumference (of border-edges) of the tile that is described.
Every time a new column of edges is being worked through, by moving the kink from top to bottom, the circumference of the region we have already crossed, grows. At the start of such a new column we deal with a special updating step, having only one input edge and three output edges. At this updating step smaller

signatures (describing the previous circumference) contribute to larger signatures (Figure 5.5).

No SAWs are counted, no pruning is done, and it is no problem if some columns remain empty or if we end up with disconnected segments (as long as those segments are connected to the graph outside the tile). After completing the last column we are done, because having all the possible signatures, linked to their respective generating functions, is the only thing we were after. They are saved in a tree that is accessible throughout the method.

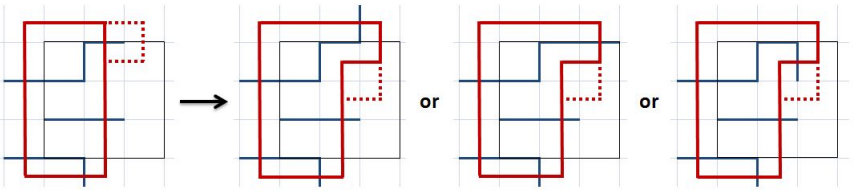


Figure 5.5: Working our way through a 3×3 tile. The signature $(10100\overline{2}03020)$ can contribute to $(1010002\overline{00}3020)$, as well as $(1010000\overline{20}3020)$ and $(1010000\overline{20}3020)$. Note that we do not choose to create another free end (by leaving the output edges empty), as we already have a label '3' in this signature. We only need signatures with two free ends, if one of them is the last label (at the bottom left).

5.5 The Algorithm

The remaining details of this method are described according to how they were implemented in the actual algorithm.

5.5.1 The Objects

In this algorithm four objects are used: Tiles, Signatures, Generating Functions and Exit Options. We shall use capital letters when referring to those objects.

- Tiles: Each Tile has a pointer referring to its current Signature. When a tile is crossed, its pointer will be changed accordingly.
  The Tiles are saved in a two-dimensional array.

- Signatures: Firstly a Signature contains an array with all its labels stored in it. Also it contains another array, with the same size, containing Exit Options, which will be explained below. Finally each Signature is pointing to a certain Generating Function.

The Signatures are stored in a tree with 'branches' determined by the labels of the Signature-array.

- Generating Functions: Since a Generating Function is actually just a polynomial in x, this object contains an array which separates the different orders of x in different slots. In the following subsection this (dense) array will be referred to as 'contents' in pseudo-code. The maximum and minimum orders are also maintained, so that they can be given at any moment in $O(1)$ time.

- Exit Options: When entering a tile, we want to quickly know all border-edges through which the SAW can exit again, that is, without having to cross arcs that are already in place. For this purpose each signature contains an array of Exit Options, with the location of the Exit Option in that array corresponding to the location of its label in the signature label-array. At first these Exit Options can be seen as a set describing all empty border-edges. However, this set is divided into one or more subsets, based on which Exit Options (border-edges) can be linked together by an arc. The structure used for these subsets is that of a circular linked list[17]. While entering a given tile, we can now consider sequentially all possible Exit Options, by first going to the Exit Option through which we are currently entering (which is thus no longer an actual 'exit option'). This Exit Option serves as a handle to the linked list. From here we can find the next and previous exit options, until the full subset of possibilities has been considered.
  Finally every Exit Option contains sufficient information to give the next Tile that is entered by choosing that specific Exit Option.

5.5.2 The Methods

The actual program consists of more than 20 different methods. We only describe those considered most important, two of them with the use of pseudo-code. In the pseudo-code N refers to the length of walks that we are after, while M refers to the width of the tiles. For the sake of clarity we do not give comments in the pseudo-code itself but shortly explain the code separately. The actual program has been written in C♯.

- The End-Walk Method. When we enter a tile, this is the method that creates a new free end in the signature, rather than an arc, thus finishing the SAW at this tile. It has as input the signature of the tile in question, the entry

[17] A doubly linked list is a series of objects that is connected through pointers 'next' and 'previous'. If an object has no next or previous element then these pointers return 'null'. In a circular linked list, as opposed to a linear linked list, a pointer to any node of the list serves as a handle to the whole list.

location (an integer between 1 and $4M-1$), the number of crossings of our previous path and the grand generating function of our previous path. This method is where the actual SAWs are found and counted, accumulated into a variable *TotalSAWs* that is accessible in the entire algorithm.

```
1: function End-Walk(SignatureOld, EntryLocation, Crossings, GrandGF)
2: Create an array Temparray, which is a copy of the label-array in SignatureOld
3: temparray[EntryLocation] = 3
4: Search for the signature belonging to Temparray, call this SignatureNew
5:     if SignatureNew is found to be a valid signature then
6:         Multiply GrandGF with the generating function of SignatureNew
7:         Divide the result by the generating function of SignatureOld
8:         Call this GrandGFNew.
9:         if N ≥ Crossings and GrandGFNew.MaxOrder+Crossings ≥ N then
10:            TotalSAWs = TotalSAWs + GrandGFNew.contents[N-Crossings]
```

The if-loop in line 5 is necessary, because even though the entry point is certainly an empty-edge, placing a free end at this edge might not be possible. This is due to situations like those in Figure 5.4, where the insides of the tile are already too full.
In line 6-8 we make sure that the old contribution (due to the old tile-signature) to the Grand Generating Function is replaced by the new one.
In line 9 we perform final checks to make sure that there are actually SAWs of length N to be found, or more precisely, that there is a nonzero contribution of order $x^{N-Crossings}$ in the generating function. Remember that the crossings are not accounted for in the generating function, and that their contribution is by separate multiplication, see also equation 5.1.

- The Recursive-Walk Method. This method has the same input as the End-Walk method, except we are now giving it the Tile that is entered directly as input, instead of its Signature. We need the Tile explicitly, because we need to change what Signature it points to, after we have crossed it. In the pseudo-code, *CurrentTile* refers to this Tile, however, in the actual program the Tile is represented by two coordinates that specify its location (in the array of all Tiles, and the lattice).

```
1: function RECURSIVE-WALK(CurrentTile, EntryLoc, Crossings, GrandGF)
2: CurrentSignature = the signature of CurrentTile
3:     if Crossings + GrandGF.MaxOrder + M^2 − 1 ≥ N then
4:         EndWalk(CurrentSignature, EntryLocation, Crossings, GrandGF)
5:     if Crossings+GrandGF.MinOrder ≤ N then
6:         Find the ExitOption belonging to EntryLoc of CurrentSignature
7:         Call this Option
8:         Option = Option.next
9:         while Option ≠ null And Option.Location ≠ EntryLoc do
10:            Find the next Tile based on CurrentTile and Option.Location
11:            Call this NextTile
12:            Find the next Entry Location on that Tile, call this NextEntry
13:            if NextEntry on NextTile is no dead end,
14:                Or there might be a possible SAW found here then
15:                if Changing the labels of EntryLoc and Option in
16:                    CurrentSignature leads to a new valid signature then
17:                    Change the Signature of CurrentTile, forming NewSignature
18:                    Divide GrandGF by the gen. function of CurrentSignature
19:                    Multiply the result by the gen. function of NewSignature
20:                    Call this GrandGFNew
21:                    Recursive-Walk(NextTile, NextEntry, Crossings+1,
22:                    GrandGFNew)
23:            Change Signature of CurrentTile back to CurrentSignature
24:            Option = Option.next
```

In line 3 and 4 we are checking for SAWs that might be long enough to be counted. Because there are M^2 vertices in a given tile, there can at most be $M^2 - 1$ edges added to the SAW. Adding this to the number of crossings and the maximal order in *GrandGenFunc* we find an upper bound for the maximal length after ending in the current Tile.
In line 5 we check if there are SAWs short enough to make it worth continuing our recursion.
In line 6-8 we access the linked list of Exit Options. The access point of the relevant subset of possibilities is found at *EntryLoc*. This point is not an actual option itself, so we immediately move to the next option.
In the while-loop from line 9-23, we go through all exit options sequentially, until there are none left in 'next' direction, or until we have come full circle. If we have not made a full circle, actually a second while loop starts after this one, going in the 'prev' direction. We have left this second loop out of

the pseudo-code, since it is similar to the first one.
In line 13, to prevent doing futile work that involves changing the tile-signature and the Grand Generating Function, we first check for each Exit Option if it does not result in a dead end in the next tile that will be entered. Checking for a dead-end comes down to checking if 'next' and 'prev' of the Exit Option in the linked list of the next tile are equal to 'null'.
After this check, there might still be situations like those in Figure 5.4 where there is no entry possible because the Tile is too full. Hence another check is done at line 15.
After changing the signature of the current tile, and changing the generating function accordingly, we continue our recursion by going into the next tile at line 21, having made one extra crossing.
Line 23 can be seen as 'erasing the footprint', and line 24 completes the while-loop.

- The main method, besides initiating the creation of relevant arrays, Signatures, Generating Functions, Tiles, etc, has the important function of beginning the recursion. It does so by giving the centre-Tile an appropriate (special) signature, and immediately stepping down into the Tile under it. The recursion ends by itself, when all directions have been treated until for each case the smallest order in the Grand Generating Function has reached length n.

5.6 Complexity

The running time of this algorithm is based on two stages. First we have to prepare our database of signatures and generating functions, by using the FLM for our tiles. Next, we can start the recursion, forming the second stage of the algorithm.

5.6.1 The number of Signatures

We have to prepare all signatures and their corresponding generating functions. As the FLM is used for this, the running time depends exponentially on the length of the signatures, which is $4M$ in our case. Hence we deal with an upper bound of $O(3^{4M})$. However, as explained in section 5.3, we expect that there are actually significantly fewer signatures than $3^{4M} = 81^M$. The precise growth rate is probably too difficult to find. However, the number of signatures for small tiles are counted in our algorithm. The results are given in Table 1.
The growth factor is significantly smaller than 81, but it increases for larger tiles. When going from 3×3 to 4×4 tiles, we can approximate the growth factor by

$7.14^2 = 50.95$. This also shows an increase compared to the previous growth factor. We believe that this increase is due to the fact that some of the restrictions discussed in chapter 5.3 become less significant in larger tiles. Although we do not know how close the growth factor might come to 81 for large tiles, we are optimistic about the low factor that we have found.

The time required to prepare all signatures (i.e. along with their generating functions, exit options, etc) is not yet relevant for small tiles. However, the required memory becomes a big problem. It is for this reason that we generalized our algorithm, so that $L \times H$ tiles can be used, with L not necessarily equal to H. A Laptop with 4GB of RAM can handle tiles with size up to 3×4. The number of signatures looks small, so one might wonder why so much memory is required. However, for each signature we also save a generating function, exit options, an array with labels, etc.

Table 1: The number of signatures for different sizes of small tiles. The growth factor is given by dividing the number in the current row by the number in the previous row. Note that going from 3×3 to 3×4 tiles means adding only 2 labels to the signature, rather than 4.

Tile	**Number of Signatures**	**Growth Factor**
1×1	12	
2×2	468	39.00
3×3	21978	46.96
3×4	156870	7.14

5.6.2 Running time of the recursion

The complexity of this recursion is probably too difficult to analyze exactly. However, placing counters at specific points in the algorithm, we can monitor closely how it behaves. The exponential behavior is only determined by the number of encounters of the methods described in the previous section. Other smaller methods as well as the operations inside the Recursive-Walk and End-Walk method, only add a polynomial contribution to the running times.

While one might first think that the Recursive-Walk Method is encountered far more often than the End-Walk method, this may not be the case. The branching factor of this recursion is so high that the number of 'last steps' in the algorithm becomes quite significant compared to the sum of all earlier steps.
To make this idea more plausible, remember our derivation of the complexity of the simple recursive algorithm. It sufficed to consider only the walks of length n

at the end of the recursion. This way we found a running time of $O(\mu^n)$.
Note that arriving at the End-Walk Method, thereby successfully counting a bunch of SAWs, can be considered as finding a Grand Signature. The main question is therefore how many Grand Signatures there are, for a given M and n.

Let us shortly summarize what we discovered by explicitly counting Grand Signatures in our algorithm. The complete results can be found in Appendix B.

Table 2: The running time of the recursion, expressed by the value of λ, for different sizes of small tiles. The estimate for λ is based on the growth factor for the number of Grand Signatures between $n = 19$ and $n = 20$.

Tile	**Estimate for λ**
1×1	2.68
2×2	2.46
3×3	2.25
3×4	2.19

The number of Grand Signatures might be approximated by the number of SAWs (on average) included in one Grand Generating Function. This mainly depends on how many different SAW segments are possible within a tile. This idea might offer a starting point for further analysis.

5.6.3 Comparing to the pruning method

Currently, this algorithm does not reach high n at all, due to a big polynomial contribution to the running time of the recursion. Using 3×3 tiles, we can compute c_{18} in 55 seconds and c_{19} in 127 seconds. However, we are more interested in the exponential growth of the running times. If λ is low enough, other algorithms that perform better for low n are eventually beaten for higher n.
Combining the contributions of both stages discussed in the two previous subsections we find that the running time can be given by:

$$T_M(n) = \lambda_M^n + 3^{4M}, \tag{5.2}$$

with λ_M describing the exponential behavior of the running time of the recursion when $M \times M$ tiles are used.
In comparison, the pruning method (which is currently the best method) has a running time of $T(n) \approx 1.3^n$.
With sufficient memory, and assuming λ_M falls below 1.3 in the large-M limit, there must be an M for which an N can be found such that for all $n > N$ we beat the running time of the pruning method.

Refraining from giving a formal proof of this statement, we shortly explain it in words. Firstly, M has to be chosen large enough so that $\lambda_M < 1.3$. Secondly, having chosen such an M, we must choose an N such that $1.3^N > 3^{4M}$. We end up with two terms in equation 5.2 that both grow slower than 1.3^n, if $n > N$.
In theory this could work, but the required memory becomes a problem. Also N can be much larger than the number 80 we are currently looking for to beat the record.[18] However, if we have reached a higher record than 79 at some times in the future, our method might become more relevant.

5.7 Possible Improvements

We finish by mentioning a few ideas for further improvements of this method. Some ideas can be easily implemented and only require some extra code, other ideas are more complicated to implement but deserve some further study. A better understanding of the complexity of this algorithm might also lead to new ideas.

- Firstly, there is still a factor 2 to be gained by using symmetry. A possible way in which this idea can be realized, is by "manually" (before the recursion is initiated) going through all different ways to start the SAW by moving down and later on to the right. This approach has been used in the simple recursive algorithm, but it can be a little treacherous here.

- When entering a tile, and having minimal order slightly below n in our Grand Generating Function, we are still checking *all* exit options of that tile, even though some of them might be "out of reach". The extra workload might be larger than expected. As discussed in the previous section, the high branching factor of the recursion can result in a significant number of "last steps" in the recursion.

- Not all tiles need to be of the same size. It is no problem to use smaller tiles as well as large tiles, since the memory requirement for smaller signatures is only a fraction of the memory for the largest signatures used. As an example of how using smaller tiles might be effective, consider placing them at the periphery of the lattice. When the recursion arrives at those smaller tiles, a smaller number of exit options has to be considered. More generally, we might even incorporate the periphery of the lattice in the tiles that are placed over it.

[18] Consider $N = 80$. We would then require $3^{4M} < 1.3^{80}$, implying $M \leq 4$. However, we assume that due to neglected polynomial contributions (especially in the pruning method's running time) this restriction is too tight. What's more, based on the first results, we believe that the exponential growth in the number of signatures is much lower than 3^{4M} (Table 1). On top of that it took only a few seconds to compute the signatures for 3×4 tiles. For these reasons we believe $M = 5$ or even $M = 6$ do not pose difficulties concerning computing time.

Appendix A Working with Generating Functions

What generating functions are, or more precisely, how they are used in this thesis, can be best described by a quick example. Consider the following problem.
We have 5 red marbles, 3 blue marbles and 4 green marbles. In how many different ways can we choose 6 marbles from this collection?
One can use binomial coefficients in such a problem, but let us take a different approach. It might seem strange at first, but we can reformulate this problem in the following way:
What is the order of x^6 in the following polynomial?

$$(x^0 + x^1 + x^2 + x^3 + x^4 + x^5)(x^0 + x^1 + x^2 + x^3)(x^0 + x^1 + x^2 + x^3 + x^4) \quad \text{(A.1)}$$

Now the first term can be seen to represent (the different options for taking) the red marbles, the second term represents the blue marbles, and the third term represents the green marbles.
Computing the order of x^6 of this polynomial actually comes down to going through all possible ways of combining different elements from each term in A.1 such that the total order is 6. But this is the same as choosing different numbers of marbles from each set such that a total of 6 marbles is chosen.
To elaborate some more on this idea, let us add a few restrictions. Let us assume we cannot choose only a single red marble; either all or none of the blue marbles must be chosen; and finally an even number of green marbles must be chosen. Now the polynomial becomes:

$$(x^0 + x^2 + x^3 + x^4 + x^5)(x^0 + x^3)(x^0 + x^2 + x^4) \quad \text{(A.2)}$$

Note that this way of solving the problem is not faster or more efficient than other possible tricks. It is mainly a way of reformulating the problem: the work of counting all possibilities still has to be done explicitly.
However, one can already appreciate the elegance of this approach by realizing how easy it is, after having worked out the above expressions, to add the option of choosing between 3 and 7 yellow marbles as well. We just need to multiply the polynomial we found with $(x^3 + \cdots + x^7)$.

So what is a Generating Function? In our thesis, a Generating Function is a polynomial that summarizes a problem in combinatorics, like the polynomials just presented above. It does so by having the coefficient of x^n correspond to the number of possible ways to solve the problem, having a total of n of a certain object. This n can represent a number of marbles that needs to be chosen, but also the number of edges of a self-avoiding walk.

Appendix B Further Results of the Tile Hopping Method

For different types of tiles we have counted the number of times the End-Walk and Recursive-Walk method are encountered while running the algorithm for different lengths n. The growth factor is obtained by dividing the current value by the previous value of the corresponding column.
Note that the number of times the End-Walk method is encountered corresponds to the number of Grand Signatures after multiplication by four, since we have used rotational symmetry in our algorithm.
The slight decrease in the growth factors is caused by the decreasing impact of polynomial growth of c_n.

Table 3: Using 2×2 tiles

n	**End-Walk**	**Growth factor**	**Recursive-Walk**	**Growth factor**
16	2627187	2.468	2834368	2.469
17	6466004	2.461	6985151	2.464
18	15915570	2.461	17198450	2.462
19	39100903	2.457	42297104	2.459
20	96046491	2.456	103927025	2.457

Table 4: Using 3×3 tiles

n	**End-Walk**	**Growth factor**	**Recursive-Walk**	**Growth factor**
16	1039810	2.264	1090668	2.265
17	2347504	2.258	2463833	2.259
18	5291391	2.254	5558088	2.256
19	11929937	2.255	12537097	2.256
20	26882340	2.253	28267943	2.255

Table 5: Using 3×4 tiles

n	**End-Walk**	**Growth factor**	**Recursive-Walk**	**Growth factor**
16	808616	2.200	840877	2.202
17	1772511	2.192	1844914	2.194
18	3891730	2.196	4052718	2.197
19	8531221	2.192	8889729	2.194
20	18683472	2.190	19475934	2.191

References

[1] Marcus du Sautoy (2003), The Music of the Primes: Searching to Solve the Greatest Mystery in Mathematics

[2] I. Jensen, A. J. Guttmann (1998), Self-avoiding walks, neighbor-avoiding walks and trails on semi-regular lattices, J. Phys. A 31 (40)

[3] Kuhn, Werner (1934), "Über die Gestalt fadenförmiger Moleküle in Losungen" (On the shape of filform molecules in solutions), Kolloidzeitschrift 68, p. 2.

[4] Paul Flory (1953), Principles of Polymer Chemistry. Cornell University Press.

[5] P.G. de Gennes (1972), Exponents for the excluded volume problem as derived by the Wilson method, Phys. Lett. 38A:339

[6] P.G. de Gennes (1979), Scaling Concepts in Polymer Physics. Cornell University Press, Ithaca.

[7] Bertrand Duplantier (1986), Polymer Network of fixed topology: renormalization, exact critical exponent γ in two dimensions, and $d = 4 - \epsilon$, Phys. Rev. Lett. 57, 941

[8] B. Shapiro (1978), A direct renormalization group approach for the self-avoiding walk, J. Phys. C: Solid State Phys., Vol. 11

[9] S Havlin and D Ben-Avraham (1982) New approach to self-avoiding walk as a critical phenomenon, J. Phys. A: Math. Gen. 15 L321-L328.

[10] Caracciolo S, Guttmann A J, Jensen I, Pelissetto A, Rogers A N and Sokal A D (2005), Correction-to-scaling exponents for two-dimensional self-avoiding walks J. Stat. Phys. 120 1037–1100

[11] E.J. Janse van Rensburg, The Statistical Mechanics of Interacting Walks, Polygons, Animals and Vesicles, 2nd Edition.

[12] R. D. Schram, G. T. Barkema and R. H. Bisseling (2011), Exact enumeration of self-avoiding walks, Journal of Statistical Mechanics: Theory and Experiment p06019

[13] Raoul D. Schram, Gerard T. Barkema, and Rob H. Bisseling (2013), SAW-doubler: a program for counting self-avoiding walks, Computer Physics Communications, 184 pp. 891-898.

[14] Hugo Duminil-Copin, Stanislav Smirnov (2010), The connective constant of the honeycomb lattice equals $\sqrt{2+\sqrt{2}}$

[15] Iwan Jensen and Anthony J Guttmann (1999), Self-avoiding polygons on the square lattice, J. Phys. A: Math. Gen. 32

[16] Nathan Clisby and Iwan Jensen (2012), A new transfer-matrix algorithm for exact enumerations: Self-avoiding polygons on the square lattice, J. Phys. A: Math. Gen. 45 115202

[17] Iwan Jensen (2013) A new transfer-matrix algorithm for exact enumerations: self-avoiding walks on the square lattice, eprint arXiv:1309.6709

[18] Iwan Jensen (2004), Enumeration of self-avoiding walks on the square lattice J. Phys. A: Math. Gen. 37

[19] A R Conway, I G Enting and A J Guttmann (1993), Algebraic techniques for enumerating self-avoiding walks on the square lattice J. Phys. A: Math. Gen. 26 1519–1534

[20] I G Enting (1980), Generating functions for enumerating self-avoiding rings on the square lattice J. Phys. A: Math. Gen. 13 3713–3722

[21] E J Janse van Rensburg and A Rechnitzer (2002), Exchange symmetries in Motzkin path and bargraph models of copolymer adsorption. Elect J Comb, 9, 1-24.