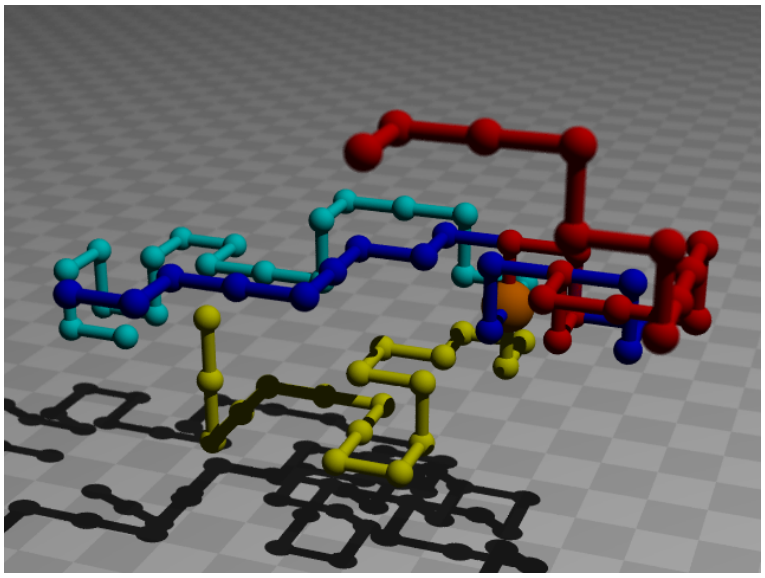
Exact enumeration of self-avoiding walks

MSc Thesis Scientific Computing

University of Utrecht



Author:
Raoul Schram

Supervisors:
Prof. Rob Bisseling
Prof. Gerard Barkema

September 8, 2011

Abstract

This thesis concerns a recently developed method for exact enumeration of self-avoiding walks, called the length-doubling algorithm. This method was created by the author and the supervisors and it substantially improves the current methods for the exact enumeration of self-avoiding walks. The basis of the algorithm and the results for the simple cubic lattice are given in the article [2]. This thesis reiterates the basis of the algorithm with slightly different terminology, and provides a more in-depth look at the implementation of the algorithm. It also adds the results of the face-centered cubic (FCC) and body-centered cubic (BCC) lattice, which are as of today yet unpublished.

Contents

Chapter 1

Introduction

A self-avoiding walk (SAW) is a walk on an underlying lattice, that does not intersect itself. An example of a self-avoiding walk of length 12 on a square lattice can be seen in Figure 1.1.

The self-avoiding walk model is a simplified model of a polymer in a good solvent. The following asymptotic scaling relations are believed to hold (strong evidence, yet unproven, see for example [1]):

$$Z_N \sim \mu^N N^{\gamma_s - 1} \tag{1.1}$$

and

$$\left\langle r_e^2(N) \right\rangle \sim N^{2\nu}, \tag{1.2}$$

where the partition sum Z_N is the total number of self-avoiding walks of length N. The parameter $\left\langle r_e^2(N) \right\rangle$ is the mean end-to-end distance of the SAW.

The importance of these relations lies in the fact that the exponents γ_s and ν are universal, which means that they do not depend on the microscopic details of the lattice, but they do depend on the dimensionality of the system. SAWs on a square lattice and a triangular lattice are expected to have the

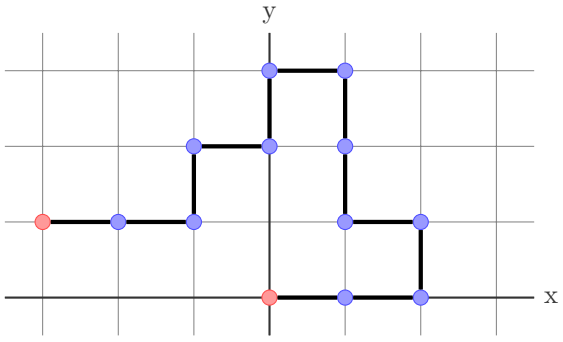


Figure 1.1: An example of a self avoiding walk with a length of 12 steps on a two-dimensional square lattice.

same γ_s and ν, but for example the fugacity μ differs between the two lattices. From the theory of critical phenomena, the exponents ν and γ_s have an even more important meaning than what was just described. The SAW model belongs to a universality class of critical systems, that extends beyond polymer physics. Each of these critical systems has a critical point, where the physics is described by power laws. In the SAW case this critical point is the limit $N \to \infty$. Another example of a critical system is the Ising model, which is a model for ferro-magnetism. The critical point in the Ising model is given by the critical temperature T_c. In a ferromagnetic system it is energetically favorable to align the spins on the lattice, which creates blocks of spins that point in the same direction. At the critical temperature, the average diameter of these blocks behaves as $\xi \sim (T - T_c)^{\nu}$.

The one-dimensional case of the SAW model is trivial: only two self-avoiding walks exist of any length N. An exact polynomial formula for Z_N or $Z_N \cdot \langle r_e^2(N) \rangle$ in two dimensions is not known, but the universal exponents are known to be: $\gamma_s = \frac{43}{32}$ and $\nu = \frac{3}{4}$. In dimensions $d > 4$ the SAW model behaves classically, with $\gamma_s = 1$ and $\nu = 0.5$. In four dimensions the exponents have the same value, but the relations (1.1) and (1.2) have to be modified by inserting a logarithmic correction. In three dimensions only crude predictions of γ_s and ν exist. This is disappointing, because most systems in nature are three-dimensional.

The best estimates in the literature for γ_s and ν are found using the SAW model. The best methods for finding these exponents can be divided into two classes: Monte Carlo methods and exact enumeration. The main advantage of Monte Carlo methods is that they can be used to simulate large ($N > 10^6$) SAWs. The finite-size effects are relatively small at these lengths, but random sampling induces a random error. On the other hand, exact enumeration calculates Z_N and $\langle r_e^2(N) \rangle$ exactly, but can only do so for small N. This thesis will reveal the details of the fastest algorithm to date [2], called length-doubling. The term "exact enumeration" is slightly deceptive in this context, because already Clisby et al. (2007) [4] showed that it is unnecessary to enumerate all SAWs of length N to obtain Z_N. In fact, their two-step algorithm is exponentially faster than complete enumeration of SAWs.

In the remaining chapters we will assume, for brevity, that for complexity estimates $Z_N \sim \mu^N$. Then any complete exact enumeration method that counts all SAWs separately, has a complexity of $O(\mu^N) \approx O(4.684^N)$ on the cubic lattice. Clisby et al. reported that their algorithm scales as $\approx O(4.0^N)$, but with a very small constant prefactor. The length-doubling algorithm has a complexity that is again exponentially faster than that, with a complexity of $O(\sqrt{2\mu}^N) \approx O(3.06^N)$.

Chapter 2

Length-doubling method

Define $\mathcal{S}_N$ as the set of self-avoiding walks of length N. All self-avoiding walks of length $2N$ can be created by connecting two SAWs of length N. In our notation we have $\mathcal{S}_{2N} \subset \mathcal{S}_N \times \mathcal{S}_N$. However, the connection of two SAWs of length N does not always yield a SAW of length $2N$. We will see how it is possible to efficiently count the number of non-self-avoiding walks of length $2N$, that are created by pairwise connection of SAWs of length N. To start, we now introduce the following (temporarily used) sets A_i:

Definition 1. $A_i = \{w \in S_N \times S_N$*: w visits position i twice*$\}$

Since any self-avoiding walk of length $2N$ does not visit any position twice, we have that $w \in \mathcal{S}_{2N} \Leftrightarrow w \notin A_i \ \forall i$. Thus, we can find $|\mathcal{S}_{2N}|$ by:

$$Z_{2N} = |\mathcal{S}_{2N}| = |\mathcal{S}_N \times \mathcal{S}_N| - |\bigcup_i A_i|. \tag{2.1}$$

It is easy to compute $|\mathcal{S}_N \times \mathcal{S}_N|$, because it is just Z_N^2, which can be found with a simple exact enumeration algorithm in $O(\mu^N)$ time. The second term in equation (2.1) might seem hard to calculate, but we can use the following result from the field of combinatorial mathematics:

$$|\bigcup_{i=1}^{n} A_i| = \sum_i |A_i| - \sum_{i<j} |A_i \cap A_j| + ... + (-1)^{n+1} |\bigcap_{i=1}^{n} A_i|. \tag{2.2}$$

This equation is known as the *inclusion-exclusion principle*. It is possible to compute $|\bigcap_{i\in I} A_i|$ for a set of positions I, from the enumeration of SAWs of length N. To see this we define the collision count c_I as:

Definition 2. $c_I = |\{w \in S_N$*: w visits all positions* $i \in I\}|$

By pairwise connecting all self-avoiding walks of length N that go through some set of positions I, we find:

$$|\bigcap_{i\in I} A_i| = c_I^2. \tag{2.3}$$

Set $c_\emptyset$ to $|\mathcal{S}_N|$, then combine equations (2.1), (2.2) and (2.3) to obtain the following concise equation:

$$|S_{2N}| = \sum_{I \in \mathcal{I}} (-1)^{|I|} c_I^2, \tag{2.4}$$

where $\mathcal{I}$ is the set of all subsets of lattice points. Thus the problem is reduced to computing all c_I and summing their squares. Consider the following algorithm: For every walk $w \in \mathcal{S}_N$ we increase the collision count c_I for all $I \in \mathcal{P}(w)$, where $\mathcal{P}(w)$ is the power set of the set of positions that w visits. Since $|\mathcal{P}(w)| = 2^N$, the total amount of work is only $O((2\mu)^N)$. This result is shown in algorithm 1.

Algorithm 1 Length-doubling

```
Input: Length N
Output: Z_2N

c := 0
Z_2N := 0
for all w ∈ S_N do
    for all I ∈ P(w) do
        c_I := c_I + 1
    end for
end for
for all c_I: c_I > 0 do
    Z_2N := Z_2N + (−1)^|I| c_I^2
end for
```

This algorithm already has the time complexity that was claimed in the introduction, but the algorithm has a major problem: the amount of memory used by the program is very large. The memory is used to store all collision counts c_I. It follows that the amount of memory we use is $k|\mathcal{I}| = O((2\mu)^N)$, for some constant k. After implementing the algorithm we empirically found that this upper bound is not tight: $|\mathcal{I}| \sim (1.6\mu)^N$. Still, the amount of memory is prohibitive and we were forced to find a solution. The solution is an algorithm that is more memory efficient, and is explained in the next chapter.

2.1 Odd lengths

At first glance, it might seem like the length-doubling algorithm is only able to compute Z_M for even M. Fortunately, for odd M the partition sum Z_M can be found by connecting two SAWs of length Z_N and Z_{N+1}, with $M = 2N+1$. The algorithm described above for odd M works analogously to the one for even N, but we need to store the collision counts for both $c_I(N)$ and $c_I(N+1)$. Then the partition sum Z_{2N+1} is given by:

$$|S_{2N+1}| = \sum_{I \in \mathcal{I}} (-1)^{|I|} c_I(N) c_I(N+1). \tag{2.5}$$

One might wonder if it could be more time efficient to connect two SAWs of unequal length. The position sets $I \in \mathcal{I}$ that have a non-zero contribution to the sum are those that occur in the set of the smallest SAWs $\mathcal{S}_N$. An algorithm based on the length-doubling algorithm has to enumerate the longest SAW of length $N + c$ (in $O(\mu^{N+c})$ time), and also has a step that sums the squares of the collision counts c_I (in $O(|\mathcal{I}(N)| \approx (1.6\mu)^N$ time). Thus, it is not inconceivable that there exists an algorithm based on length-doubling that runs in $O(\mu^{N+c} + |\mathcal{I}(N)|)$ time. Then it would be beneficial to connect two SAWs of unequal length. Unfortunately, in algorithm 1 the computation is dominated by line 7, which is independent of the number of position sets $|\mathcal{I}|$. The improved length-doubling, described in chapter 3, has a more complicated analysis regarding the complexity of the algorithm, and will be discussed there.

In the next chapters the algorithms will be given for even lengths M, but the algorithms presented there can be modified to compute Z_N and R_N for odd length. To compute Z_{2N+1}, algorithm 1 is modified such that both $c_I(N)$ and $c_I(N+1)$ are stored. This requires asymptotically roughly the same number of operations as the computation of Z_{2N+2}, because we can only skip storage of $c_I(N+1)$ with $|I| = N+1$, which is only one of the 2^{N+1} subsets I. Since it is necessary to store both $c_I(N)$ and $c_I(N+1)$, computation of Z_{2N+1} requires at most twice as much memory as computation of Z_{2N+2}.

2.2 End-to-end distance

As already noted in the introduction, another important parameter of the model, besides the partition sum is the mean end-to-end distance. In the context of exact enumeration it makes more sense to compute the summed end-to-end distance R_N, because we obtain these integer numbers exactly. The length-doubling algorithm also lends itself to this purpose with a small associated extra cost.

Instead of taking the cardinalities of the sets in equations (2.1) and (2.2), we take the weighted sum. The weight of each connected self-avoiding walk w of length N is given by its end-to-end distance $\nabla(w)$. The inclusion-exclusion principle still works and produces the sum of the squared end-to-end distances. Equation (2.1) becomes:

$$R_N = |\mathcal{S}_N|_\nabla = |\mathcal{S}_{N/2} \times \mathcal{S}_{N/2}|_\nabla - |\bigcup_i A_i|_\nabla. \tag{2.6}$$

Here, $|\cdot|_\nabla$ is the weighted sum of a set. Thus, the question remains how $|\bigcap_{i\in I} A_i|_\nabla$ can be counted, without generating all SAWs of length $2N$. Analogously to the collision count, we define the squared end-to-end distance count r_I:

$$r_I = |\bigcap_{i\in I} A_i|_\nabla. \tag{2.7}$$

Let K denote the list of end positions of the SAWs of length N that generate the set $\bigcap_{i\in I} A_i$, when these SAWs are pairwise connected. In three dimensions the end-to-end distance can then be found by:

$$r_I = \sum_{i,j \in K} (x_i - x_j)^2 + \sum_{i,j \in K} (y_i - y_j)^2 + \sum_{i,j \in K} (z_i - z_j)^2, \tag{2.8}$$

where x_i, y_i and z_i are respectively the x, y and z coordinates of the position i. Writing out equation (2.8) we obtain:

$$r_I = 2c_I \sum_{i \in K} (x_i^2 + y_i^2 + z_i^2) - 2(\sum_{i \in K} x_i)^2 - 2(\sum_{i \in K} y_i)^2 - 2(\sum_{i \in K} z_i)^2. \tag{2.9}$$

Thus, for each walk $w \in \mathcal{S}_N$ we not only need to update c_I, but also the sum of the squared end-to-end distances r^2 and the sum of the coordinates x, y and z. Obviously, this argument can be extended to any dimension. In practice this modified algorithm is slower by a factor of about 2, compared to the one as stated in Algorithm 1. Because the algorithm is exponential in N, a factor of two reduces the maximum length that can be obtained for the cubic lattice by only 0.6 steps.

Chapter 3

Memory-efficient length-doubling implementation

The improvements described in this chapter concentrate on the memory requirements of the length-doubling algorithm. As noted in the previous chapter, it is not viable to store all collision counts c_I in memory. In this chapter we will show that these counters can be found without storing all collision counts simultaneously, but instead we store all self-avoiding walks of length N. All SAWs $\mathcal{S}_N$ can be stored in $N\mu^N$ memory blocks. The old algorithm that was described in chapter 2 uses $|\mathcal{I}| \sim (1.6\mu)^N$ memory blocks. Thus, we gain an exponential factor of $1.6^N/N$ in space complexity, compared to the old method.

In the improved length-doubling algorithm it is crucial how the SAWs are stored. Each SAW $w \in \mathcal{S}_N$ is represented by a sorted list of positions $p_1 < p_2 < \ldots < p_N$. Since the length-doubling algorithm does not depend on the order in which the lattice points are visited, we can assign any position number to any lattice point. The data structure that stores the set SAWs $\mathcal{S}_N$ is a tree $T(\emptyset) = (V, E)$, that has a root node v_0. Then a SAW w is a represented by a path from the root node of length N. If two paths start with the same sublist of positions then the paths are combined, and for the vertices in that sublist we increase the λ counter that denotes the number of SAWs that correspond to that particular vertex. It is not possible to combine two paths that have a sublist in common, but have different paths up to that sublist, because then the tree $T(\emptyset)$ would not uniquely represent the set of SAWs $\mathcal{S}_N$. An example of the tree $T(\emptyset)$ is given in Figure 3.1.

By construction (the SAWs are sorted), the lattice point with the highest position number p_h in the tree is now guaranteed to be in the leaves of $T(\emptyset)$: any path that contains p_h will end in p_h. The leaves are stored in the data structure $L(\emptyset)$. Any leaf v in this data structure is linked to its corresponding vertex in the tree $T(\emptyset)$. The collision count c_{p_h} can be found by summing all λ_{p_h} counts in the leaves that correspond to the lattice point p_h. The data structure $L(\emptyset)$ groups the leaves by their position p_h. The number of lattice points is bounded by $O(N^d)$, where d is the dimension of the system. In contrast, the number of leaves is exponential in N and scales as $O(\mu^N)$. This means that

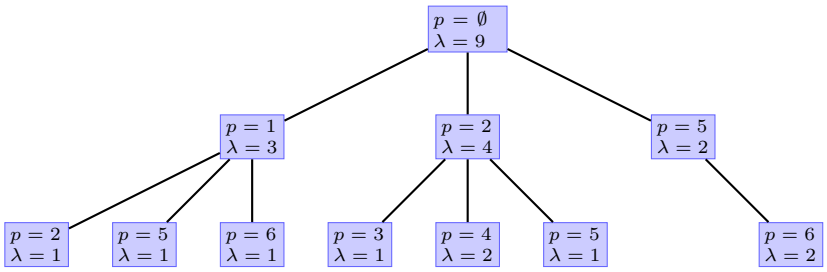


Figure 3.1: This figure shows how the SAWs are stored in memory in the improved length-doubling algorithm. It is an example of a possible initial tree of SAWs of length 2. The values of p are the positions that represent a node. The value λ denotes the number of times a node in the tree is used by the set of SAWs of length 2.

the number of leaves per position is exponential in N. The group of leaves that correspond to p_h is found in a time proportional to the number of used lattice points, which means that asymptotically the fraction of time spent on the search for the highest p_h tends to zero as N grows to infinity, i.e. it is negligible.

To proceed to the next-highest lattice point $p_{h'}$, the leaves corresponding to the position p_h are removed from the tree $T(\emptyset)$ and the data structure of the leaves $L(\emptyset)$ is updated accordingly. The tree $T(\emptyset)$ is now in an equivalent state from which we can find $p_{h'}$. Thus, the collision count $c_{\{p_{h'}\}}$ can be found in the same way as $c_{\{p_h\}}$ was found. We call this update of the leaves $L(\emptyset)$ and $T(\emptyset)$ the "*proceed*" step. By induction, all $c_{\{p\}}$ are found by applying this *proceed* step, until only the root vertex is left in the tree (the λ count of this node is equal to $c_\emptyset$).

This leaves us with the question how to find all collision counts of the form c_I. We will achieve this by introducing an "*extend*" step that allows us to compute $c_{I\cup\{e\}}$ after the computation of c_I, where e is the lattice point that extends the subset I of lattice points. The extension e is strictly smaller than all lattice points in I. Remembering that the SAWs in $T(\emptyset)$ are sorted low to high, we see that the extension e is a vertex higher up in the tree $T(\emptyset)$. All collision counts can be found by a unique series of *extend* and *proceed* steps.

To explain the *extend* step, we introduce the tree $T(I)$, in which all self-avoiding walks are represented that go through all lattice points in the set I. For the collision count of any extension (and subsequent extensions), the path that the SAW takes in $T(\emptyset)$ is irrelevant below the vertex that corresponds to the extension e. Thus, we define the tree $T(I)$ as the tree of all self-avoiding walks that go through the subset I, but with these partial paths removed, i.e. all vertices v are removed that satisfy $\exists i \in I \ \ : i \leq v$. The λ count of the root vertex v_0 of $T(I)$ is now the same as the collision count c_I. If it is possible to efficiently find $T(I \cup \{e\})$ from the tree $T(I)$ for all extensions e, then by induction, we can efficiently find all collision counts c_I.

Assume that we have found $T(I)$ for some I, then the subset I is first extended by the highest position $\{p_h\}$ (which is lower than all elements in I) in the tree $T(I)$, in the same way as we found the collision count $c_{\{p_h\}}$ in the previous paragraphs. The SAWs that contribute to the λ count of some vertex with position p_h in the leaves $L(I)$ are exactly those that contribute to the tree

$T(I \cup \{p_h\})$. Thus, the leaf data structure $L(I \cup \{p_h\})$ consists of the parent vertices of all leaves in $L(I)$ with position p_h. If two or more leaves have the same parents, then after a series of *proceed* steps, their λ counts will be added to form the λ of the parent. Then after all collision counts $c_{I \cup J}$ are found, with all elements in J smaller than every element in I, the *proceed* step described earlier is used to obtain the configuration of $T(I)$ and $L(I)$ that has $p_{h'}$ as its highest lattice point.

In both the *proceed* and the *extend* steps, only the parents of some vertices in the leaf data structure $L(I)$ are needed to compute $L(I \cup p_h)$ from $L(I)$. The parent of a vertex in $T(I)$ is equal to the parent of the same vertex in the original tree $T(\emptyset)$. Thus, the tree $T(I)$ is entirely determined by leaf data structure $L(I)$ and the structure of the original tree $T(\emptyset)$. The *proceed* and *extend* steps can be computed with the λ values of the leaf data structures. Thus, in the algorithm it is unnecessary to actually store $T(I)$. The total memory requirements are then the amount of memory it costs to store $T(\emptyset)$ (which is $O(N\mu^N)$), and all leaf data structures that need to be stored simultaneously. If a subset I is extended by the afore mentioned procedure, then we need to keep the leaf data structure $L(I)$ in memory until we have backtracked from the extension. Each vertex in the basic tree $T(\emptyset)$ is at most stored once in a leaf data structure currently in memory: if vertex v is in the leaves of $T(I)$ for some subset I, then it cannot be in any tree $T(I \cup J)$ by construction of $T(I \cup J)$. Thus, the leaf data structures also use $O(N\mu^N)$ memory (and likely much less).

Algorithm 2 Improved Length-doubling

```
1: Input: Length N, tree T(J)
2: Output: Z_{2N}(J)
3: Recursive procedure: ImprovedLD(N, T(J), J)
4: Z_{2N}(J) := 0
5: while T(J) has > 1 vertex do
6:    p_h := highest label in T(J)
7:    I := J ∪ {p_h}
8:    c_I := 0
9:    for all leaves v ∈ T(J): p(v) = p_h do
10:      c_I := c_I + λ(v)
11:   end for
12:   Z_{2N}(J) := Z_{2N}(J) + c_I^2 − ImprovedLD(N, T(I), I)
13:   Remove vertices v ∈ T(J) with p(v) = p_h
14: end while
```

The recursive algorithm is given in algorithm 2. Given the tree $T(\emptyset)$, Z_N is obtained by computing: $Z_{2N} = Z_N^2 -$ **ImprovedLD(**N**,** $T(\emptyset)$**,** $\emptyset$**)**.

The cost of the improved length-doubling algorithm is no more than that of the original length-doubling algorithm, ignoring some constant factor, which can be seen as follows. The computational cost of an efficient implementation of Algorithm 2 is dominated by a constant times the operation on line 10. Thus the work ratio between the original and improved length-doubling algorithm is a constant times the average λ count over all trees $\langle\lambda(v)\rangle$. It is hard to estimate $\langle\lambda(v)\rangle$, not in the least because it depends on the numbering of the positions.

A smart position numbering scheme reduces the number of vertices in the

trees $T(I)$, because this increases the average λ count. On the cubic lattice, we found empirically that an ordering of lattice points by the Euclidean distance to the origin performed very well: $p_1 < p_2$ if $\|p_1 - O\| < \|p_2 - O\|$, with O the origin and $\|\cdot\|$ the Euclidean norm. We can intuitively see why this scheme is efficient by looking at the top of the tree: the first level of the tree only consist of the 6 neighbouring points of the origin. Since we do not have control of the number of leaves, the best position numbering scheme produces the narrowest tree.

Chapter 4

Symmetry

It is necessary to exploit the symmetry of the cubic lattice, to create a competitive algorithm. The cubic lattice has 48 isometries, which can be obtained by a combination of reflections in the x, y and z-planes and the planes $x = y$, $y = z$ and $x = z$. Let $\Phi = \{\phi_i,\ 1 \leq i \leq 48\}$ be the set of isometries of the cubic lattice. For each set of positions I there are 48 isometries ϕ_i and corresponding sets of positions J_i, such that $\phi_i(J_i) = I$. Because all J_i are isometric to each other we have that $c_{J_i} = c_{J_j}$, for all i and j. Thus, it is only necessary to compute the collision count for one of the J_i, which we will set to J_1. This result is then multiplied by the number of distinct sets of positions in $\{J_i,\ 1 \leq i \leq 48\}$, called the multiplicity m_I of the set.

Apart from enumerating all sets J_i and counting the number of distinct sets, it is also possible to find the multiplicity by counting the number of isometries that map the set J_0 onto itself. The multiplicity is then 48 divided by this number. This can be deduced from group theory, but can also be shown with symmetry arguments. By symmetry, each set J_i must have the same number of isometries that map the set onto itself. Thus, the total number of distinct sets must be the number of isometries of the lattice (48) divided by the number of isometries that map J_i onto itself.

The length-doubling equation (2.4) then becomes:

$$|S_{2N}| = \sum_{I \in \mathcal{I}} (-1)^{|I|} m_I c_I^2, \tag{4.1}$$

where $\mathcal{I}$ is a subset of the set of lattice point subsets $\mathcal{I}$ in the old equation (2.4). The new index set $\mathcal{I}$ is essentially the old one, but it contains only one subset I of lattice points in $\{J_i,\ 1 \leq i \leq 48\}$ that are equal to I after applying an isometry ϕ_i. Thus, we need to create a "ranking" that unambiguously determines which of the subsets J_i is included in $\mathcal{I}$. The solution that is used here is a ranking by lexicography. We define the lexicographic comparison between two sublists I and J (with defined order) similarly to [7]:

Definition 1. *A sublist $I \in \mathbb{Z}^N$ is said to be **lexicographically larger** (or **smaller**) than another sublist $J \in \mathbb{Z}^N$ if $I \neq J$ and the first nonzero component of $I - J$ is positive (or negative, respectively). Symbolically we write $I >^L J$ or $I <^L J$.*

Notice that there is a difference between the subset I and the sublist I: the subset I does not have a specified ordering, while the sublist I does. Because the subset I in the algorithm grows by a sequence of *extend* steps, it is natural to identify it with the sublist I that is sorted high to low. Using this definition, $\mathcal{I}$ can now be defined as the set of the sublists of the lattice points that satisfy the condition: $I \geq^L \phi_i(I)\ \forall i$. The lexicographic comparison depends on the position numbering. Ideally, we want a position numbering scheme that ensures the following condition:

$$\exists i\ \ I <^L \phi_i(I) \Rightarrow \exists i\ \ I \cup \{e\} <^L \phi_i(I \cup \{e\}). \tag{4.2}$$

If this condition holds, it means that any extension of a not lexicographically largest subset cannot be extended to a lexicographically largest subset. If this condition is satisfied, then it is allowed to immediately backtrack after it has been found that I is not the lexicographically largest subset. It turns out that every position numbering scheme satisfies condition (4.2), as explained in the next section.

4.1 Position numbering scheme

Theorem 1. *Let $I = p_1 p_2 \dots p_n$ be a sublist of positions p_k on the lattice, with $p_k > p_{k+1}$ for all $1 \leq k < n$. Let position e be an extension to I, with $p_n > e$. Then the condition (4.2) holds:*

If the sublist I is not the lexicographically largest subset, i.e. $\exists i\ \ I <^L \phi_i(I)$, then the extended subset $I \cup \{e\}$ is not the lexicographically largest either: $I \cup \{e\} <^L \phi_i(I \cup \{e\})$.

Proof. Given the definition of I in the theorem, let $I' = \phi_i(I)$ be a sublist that is lexicographically larger than I: $I' = p'_1 p'_2 \dots p'_n$. Extend the sublist I with the position e and "extend" the sublist I' by the equivalent position $e' = \phi_i(e)$. The "extension" of I' is not a true extension as defined in the previous chapter, because e' is allowed to be larger than lattice points in I'. Application of isometry ϕ_i on $I \cup \{e\}$ yields the sorted sublist $\phi_i(I \cup \{e\}) = p'_1 p'_2 \dots p'_k e' p'_{k+1} \dots p'_n \geq^L I' e'$. This follows from the observation that given the set of lists $\mathcal{L}$ that consist of the subset of lattice points J, the list $l \in \mathcal{L}$ that is sorted high to low is the lexicographically largest. By assumption, we have that $I <^L I'$, which implies that $Ie <^L I'e'$. This in combination with the observation made earlier yields the desired result: $I \cup \{e\} <^L \phi_i(I \cup \{e\})$. □

Thus, any position numbering scheme can be used to gain efficiency by only counting c_I with I the lexicographically largest subsets. However, for an arbitrary position numbering scheme it is necessary to check if the current list of positions is the lexicographically largest after every extension. We will now propose a position numbering scheme that reduces the number of checks and also reduces the computational burden of the check itself.

The key of the proposed position numbering scheme is that the number consists of two parts: $p = (q, s)$, with q the part of the position that is invariant under symmetry operations. The symmetry part s is a number between 1 and 48 and denotes which position p corresponds to, out of the 48 positions given

by q that are equivalent. Some positions have less than 48 equivalent positions, which means that there is more than one choice for s. This conflict can be resolved arbitrarily, although from a programming perspective it can be useful to set s at least once to the highest value (48), for every q.

The sorting routine of the improved length-doubling algorithm first sorts by the invariant part q and then by the symmetry part s. Also, in a lexicographic test between two lattice points p_1 and p_2, the invariant part is more important: $p_1 - p_2 > 0$ if $q_1 > q_2$. Only when the invariant part is equal, the symmetry parts are compared. Now, if we take a sublist of positions I, then after they are sorted all positions that have the same q are sequentially ordered in I. The invariant part q_j of I then looks for example as follows: $I_q = q_1 q_1 q_2 q_3 q_3 \dots q_k$. It is helpful to group the lattice points with the same q into groups g_j, also sorted high to low: $q(g_j) > q(g_{j+1})$. (These are not the groups from group theory, but the term is used to distinguish between the groups g_j and the sublists I.) Thus, it is also possible to write the sublist I as a list of groups: $I = g_1 g_2 \dots g_k$, where the number of groups k is smaller or equal to n, the cardinality of I. For the "average" sublist we might expect k to be almost as large as n.

The key to understanding the benefits of the structure of the proposed position numbering scheme is the observation that if we apply an isometry ϕ_i to a sublist I, the lattice points in I can only move inside the groups i.e.: $\phi_i(I) = \phi_i(g_1)\phi_i(g_2)\dots\phi_i(g_k) = g_1' g_2' \dots g_k'$. Each of the groups g_j corresponds to an invariant position part q, and every extension e has the property $q(e) \leq q(g_k)$ by the definition of the extension. This implies that the groups $g_1,\ g_2,\ \dots,\ g_{k-1}$ remain unmodified by the extension e and any subsequent extensions. We call these groups immutable. If there are not any immutable groups, the position numbering scheme will not yield any better results than any other scheme. However, the overwhelming majority of sublists I have at least one immutable group. The following explanation assumes that the sublist I has at least one immutable group.

By Theorem 1 we do not consider sublists I that are not the lexicographically largest. Thus, the first group in the set of immutable groups must be the lexicographically largest group in the set of groups $G_1 = \{\phi_i(g_1),\ 1 \leq i \leq 48\}$. If the set G_1 has 48 elements, then we have that for all isometries ϕ_i that $\phi_i(g_1) \neq g_1$, except for the identity isometry ϕ_1 that images every set onto itself. Because the lexicographic test considers the first group the most important, this means that the sublist I that starts with the group g_1 is always the lexicographically largest sublist: $I \geq^L \phi_i(I)$ for all i. Notice that this conclusion can be made without any assumption on the remaining groups $g_2 \dots g_k$. Thus, when the first group becomes immutable and only the identity isometry images the group onto itself, then it is no longer necessary to carry out lexicographic tests for any extension and subsequent extensions.

Of course, the first group does not always have a multiplicity of 48. Therefore, we will generalize this result. Define the constraint set F_j as the set of isometries that image the group g_j onto itself: $F_j = \{\phi_i :\ \phi_i(g_j) = g_j\}$. If an isometry ϕ_i is not in the constraint set F_1 of the group g_1, then $\phi_i(I) <^L I$ by the assumption that I (and thus g_1) is the lexicographically largest. This conclusion is independent of $g_2 \dots g_k$. Thus, the set of isometries that has to be considered in a lexicographic test is constrained to the subset of isometries F_1. Similarly, by assumption $g_1 g_2$ is the lexicographically largest, and we only have to consider the isometries that satisfy the condition $\phi_i(g_1 g_2) = \phi_i(g_1)\phi_i(g_2) = g_1 g_2$. Gen-

eralizing this for all immutable groups yields the set of conditions $\phi_i(g_j) = g_j$, for all $1 \leq j < k$. It is easy to see that the set of these conditions is equivalent to the following condition:

$$\phi_i \in \mathcal{F}_I = \bigcap_{1 \leq j < k} F_j \tag{4.3}$$

Note that the identity isometry ϕ_1 always satisfies this condition. Thus, if $g_1 g_2 \dots g_{k-1}$ is the lexicographically largest, then the lexicographic test of I can be reduced to the test $\phi_i \in \mathcal{F}_I$. Since the sublists I are formed by a sequence of extensions, $\mathcal{F}_I$ can be efficiently stored and updated. The number of lexicographic tests is therefore reduced from 48 to $|\mathcal{F}_I|$. If $|\mathcal{F}_I|$ only includes ϕ_1, then no test has to be performed at all: the sublist I is guaranteed to be the lexicographically largest sublist.

We can freely choose an assignment of q to any set of symmetry equivalent lattice points. As already noted in chapter 3, we chose an assignment that depends on the Euclidean distance to the origin of the lattice, because it performed the best.

Chapter 5

Memory Management and Parallelization

Even though the improvements of the length-doubling algorithm described in chapter 3 decrease the memory requirements by an exponential factor, the algorithm still needs about $O(\mu^N)$ memory. At lengths $2N > 26$ the memory requirements become prohibitive: it is impossible to store all SAWs in memory. If a tree $T(I)$ does not fit in memory (this check is done runtime), then we do calculate c_I, but since the tree cannot be completed in memory all collision counts $c_{I\cup J}$, with $i > j$ for all $i \in I$ and $j \in J$, are computed in separate jobs by the algorithm in the trees $T(I \cup j)$ for all $j < i,\ \ i \in I$. This is done recursively, starting with the job corresponding to the original tree $T(\emptyset)$.

Thus, for this to work it is necessary to efficiently construct the tree $T(I)$ from scratch. Each SAW w of length N that is represented in $T(I)$ visits the lattice points in I in a particular order. Let this order be given by the permutation $\pi(I)$. Then for each permutation $\pi(I)$ we find the SAWs that go through all lattice points in the set I in that order by a simple recursive exact enumeration algorithm. After each step the algorithm finds a lower bound on the number of remaining steps that need to be spent to visit all points in $\pi(I)$ that have not yet been visited. This lower bound is equal to the distance along the lattice edges between the current point and the first point in $\pi(I)$ that has not been visited plus the distance between all subsequent neighbouring elements of $\pi(I)$. Especially in three dimensions, this upper bound is quite tight. If the lattice points in I do not lie on a line or plane, then in almost all cases there will be some "direct" path between the lattice points.

This means that the cost of generating a self-avoiding walk in the algorithm will still cost a (slightly larger) constant amount of time, but this increase does no matter much, as the insertion of the SAW in the tree costs $O(N)$ time. The cost associated with the generation of a SAW remains roughly equal, but the number of SAWs generated increases. A self-avoiding walk $w = p_1 p_2 \dots p_n$ is generated when constructing $T(\emptyset)$, $T(\{p_1\})$, $T(\{p_2\})$, $\dots$, $T(\{p_n\})$ if the job is split only on the first level. If we split all jobs up to the nth level, then each SAW is generated $\sum_{i\le n} N!/(i!(N-i)!)$ times. This is just the number of subsets with at most n elements that can be formed by the lattice sites of a SAW of length N. Together with a factor $O(N)$, the cost of the tree building

step could exceed the collision counting step if the jobs are split at equal level. Another issue with too many jobs is that it costs time to switch between jobs (clearing temporary variables for example).

5.1 Parallelization

The program takes roughly 50,000 hours on a single processor machine for a total length $2N = 36$. Obviously, the program needs to be parallelized in order for it to finish within a reasonable amount of time. We chose the easiest (embarrassingly parallel) approach: split the work over the processors by distributing the jobs, as described by the previous paragraphs, over the processors. The number of jobs is determined by the length of the SAWs N and the memory threshold. Luckily, the number of jobs created this way is more than sufficient to keep hundreds of processors busy. For $2N = 36$ and a memory threshold of 2 GB, in total 22 million jobs were created.

We use a client-server model for the distribution of the jobs and all communication. The server keeps track of all jobs that still need to be processed. It is initially filled with all jobs that correspond to the trees $T(\{p\})$. Each client then requests one job from the server. The number of jobs in the initial queue of the server is larger than the number of processors that were used for the computation. The clients then process the jobs with two possible outcomes for each job: The first possibility is that the tree $T(I)$ fits into the memory. The result is then communicated to the server, which writes the result to a file and gives a new job (if available) to the client. The other possibility is that the job is too large to fit in memory. The only result is then c_I, and the new jobs $T(I \cup \{p\})$ are sent to the server.

If the server does not have any remaining jobs, but some processors are still busy, then each jobless client stalls and checks on a short interval (1 second) whether the queue of the server is still empty. If the server recognizes that all jobs have finished, or when the time limit has been reached, the server shuts down and all clients time-out. The jobs are sent out to the clients in a depth-first approach. This is done to prevent the queue of jobs to become exceedingly large.

The parallel program was run on the Huygens supercomputer in Amsterdam with 128 processors running concurrently for a maximum of 120 hours. Thus, each run costs about 15,000 CPU hours. If for some reason the program has stopped (either from the outside, or from some hardware or system error), then we could lose large amounts of data. Therefore the program writes the queue, including the running jobs, to the hard-disk every 10 minutes. This is not enough though, since we also want to know which results have been gathered up to that point. We store the results of the last 10 minutes with the same id as the file of the queue to keep track of this. This mechanism is also used to transfer the computation from one machine to another. This safety mechanism came into actual effect for the longest run of the cubic lattice ($2N = 36$). The server received too many connections, and the clients falsely concluded that the server had shut down. As a consequence, the parallel program then shut down completely. Luckily, the safety mechanism prevented about 17,000 CPU hours from being lost.

The first job is to compute $c^2_\emptyset$ and to create the new jobs $T(\{p\})$. The compu-

tation of $c_\emptyset$ takes $O(\mu^N)$ computation time, which although small compared to $O((2\mu)^N)$, still costs a substantial amount of time. This job was therefore computed on a single processor of the compute cluster of the University of Utrecht, called Mars. The suspended file can be transferred directly to the supercomputer Huygens, because the queue on the server is stored in a manner that is independent of the endianness of the computer it runs on. The server does not even "know" what it actually stores in the queue, which has the advantage that it does not need to be modified if the specifications of the jobs change. In theory, it is possible to connect any computer to the server for computation, with the proper configuration. One could even imagine a load-balancing set of servers, where each client is connected to a server with very low latency. For example, the supercomputer Huygens consists of nodes with 32 processors each. Thus, we use 4 nodes for our program. We could create 4 servers, with a server on each node we use. Then the client-server communication would be done in memory, while the inter-server communication could be done independently over the network, but only when rebalancing would need to be done. This would reduce the effective latency (and bandwidth also) of the parallel program. However, the latency on the LAN is already very low: <1ms, and thus there is not much benefit in implementing this configuration.

Chapter 6

Results

The results of this thesis are split up into four parts: three paragraphs concern the results of the different lattices (SC, BCC and FCC). The remaining paragraph summarizes the performance of the algorithm on those lattices. The most important result that can be obtained from the exact enumeration data are the critical exponents γ_s and ν. However, it is extremely difficult to extract these exponents from the values for Z_N and R_N due to finite size effects. Evidence suggests that the equation for Z_N has to be refined to:

$$Z_N = k_0 \mu^N N^{\gamma_s - 1}(1 + k_1 N^{-\Delta_1} + k_2 N^{-\Delta_2} + \ldots + k_i N^{-1} + \ldots) \tag{6.1}$$

The exponent Δ_1 of the leading correction term is approximately 0.46 [3]. This result was obtained by Monte Carlo simulations. Monte Carlo simulations are generally more suitable for determination of Δ_1, because for larger lengths all other correction terms become negligible compared to the leading correction term. Similarly the summed squared end-to-end distance is refined to the following equation, with different constants k_j:

$$R_N^2 = k_0 Z_N N^{2\nu}(1 + k_1 N^{-\Delta_1} + k_2 N^{-\Delta_2} + \ldots + k_i N^{-1} + \ldots) \tag{6.2}$$

In general, a simple least-squares fitting procedure on either equation with only a few correction terms will fail to produce satisfying results for γ_s and ν. A slight improvement to this fit is the introduction of an effective exponent γ_{eff}, that can be found by numerical manipulation of equation (6.1):

$$\gamma_{\text{eff}} = \gamma_s - k_1 \Delta_1 (\Delta_1 + 1) N^{-\Delta_1} + \ldots \approx 1 + \frac{2\log Z_N - \log Z_{N-1} - \log Z_{N+1}}{2\log N - \log(N+1) - \log(N-1)} \tag{6.3}$$

The higher-order correction terms are not shown in the equation above, but are still present in γ_{eff}. It is easier to fit γ_s from this equation, but there are still problems. The results will in general be biased by the first term that is not included in the fit (of course one could get lucky). Additionally, if we do not assume a value for Δ_1, $\Delta_2 \ldots$, then there exist wildly varying combinations of

Δ_j that for small N (in the exact enumeration range) produce almost equal fits. In principle we might not care about the values of Δ_j at all, but their choice heavily influences the fit of the parameter γ_{eff}. For the exponent ν we can find a similar equation for an effective exponent ν_{eff} to fit the parameter ν.

Therefore, after several attempts to produce a robust fitting procedure, we concluded we were not able to produce one. We will include the values obtained by Nathan Clisby from our data, whose methods are described in [4].

6.1 Simple cubic lattice

Of the three lattices that are covered in this thesis, the cubic lattice is arguably the most important one, for the simple reason that the critical fugacity μ is the smallest. This means that it is possible to enumerate SAWs of larger lengths, which in turn decreases the relevance of higher-order correction terms. We were able to enumerate up to length $2N = 36$, which is a major achievement. The previous record was set in 2007 by Clisby et al. [4] at the length $2N = 30$. For the largest length we used around 50,000 CPU hours at the supercomputer Huygens. Compared to the performance data given by Clisby, we see that our algorithm is approximately 1000 times as fast.

The resulting values for Z_N and R_N can be found in Appendix A.1.

The analysis of the Z_N and R_N series by Nathan Clisby yielded the following values: $\mu = 4.684015(15)$, $\gamma_s = 1.1569(2)$ and $\nu = 0.5880(2)$. The value obtained for the γ_s exponent is in agreement with the literature value $\gamma_s = 1.1573(2)$ computed with the Pruned-Enriched Rosenbluth (PERM) method by Hsu et al. [5][1]. The pivot algorithm is a Monte-Carlo algorithm that is mainly used to compute an estimate of the average end-to-end distance $r_e(N)$. The literature value $\nu = 0.587597(7)$ obtained by Clisby [3] using the pivot algorithm is in reasonable agreement with our value.

6.2 FCC lattice

At each step on the FCC lattice the self-avoiding walk has 12 choices, compared to 6 on the simple cubic lattice. These choices can be written as a combination of reflections on the $x = 0$, $y = 0$ and $z = 0$ planes and permutation of the x, y and z coordinates of the vector $(1, 1, 0)$.

The face-centered cubic lattice has the advantage that it does not have even-odd oscillations in Z_N and R_N. These occur in the SC and BCC lattices, because on these lattices, even lattice points can only be visited by an even number of steps, while the odd lattice points are always visited after an odd number of steps. The FCC lattice does not have this property. Thus, given a series of Z_N of equal length, the FCC lattice has twice the number of data points that can be used in a single fit. Unfortunately, μ is much higher for the FCC lattice compared to either the simple cubic or the body-centered cubic (BCC) lattice:

[1] In 2009 I implemented the method used by Hsu et al. [5] and created my own dataset as part of a project on the course Modeling and Simulations. Although I did not have access to a supercomputer, with patience I obtained results with a similar accuracy. The results that I obtained were $\gamma_s = 1.1570(2)$ and $\nu = 0.58763(15)$, which is remarkably close to the currently best values.

$\mu = 10.037067(7)$ by our series. The highest length that we obtained was $2N = 24$.

As explained in the introduction, the critical exponents are expected to be equal to those found for the SC and BCC lattice. Indeed, Nathan Clisby extracted the following values from our data: $\gamma_s = 1.1568(3)$ and $\nu = 0.58815(25)$, which are consistent with the values found on the simple cubic lattice.

The series of the FCC lattice are found in Appendix A.2.

6.3 BCC lattice

At each step on the BCC lattice the self-avoiding walk has 8 choices, compared to 6 on the simple cubic lattice. The unit vectors that denote these choices are found by applying a combination of reflections on the $x = 0$, $y = 0$ and $z = 0$ planes and permutation of the x, y and z coordinates on the vector $(1, 1, 1)$.

In contrast to the FCC lattice, the points on the lattice can be divided into even and odd lattice points. As μ is still higher than the SC lattice, the BCC lattice has no real advantages compared to the other lattices. However, Clisby found the following (independent) estimates of the critical exponents: $\mu = 6.530521(6)$, $\gamma_s = 1.567(3)$ and $\nu = 0.5882(2)$. These values are again consistent with the results for the FCC and SC lattices.

The series for the BCC lattice up to $2N = 28$ are found in Appendix A.3.

6.4 Computation time

The fact that our algorithm improves the length of the series by a significant amount shows us that it delivers an exponential speed-up. As noted in chapter 3, the computation time t_N as a function of N is expected to behave as:

$$t_N = (2\mu)^N / \langle\lambda\rangle , \tag{6.4}$$

where $\langle\lambda\rangle$ is the average λ count over the whole course of the algorithm. It is hard to give rigorous bounds for this variable. It could be anything from a constant close to 1 to an exponential function in N. Figure 6.1 shows the time in seconds that is used per self-avoiding walk with a total length of $2N$ on a logarithmic scale. Any complete enumeration algorithm would have all data points on a horizontal line. The figure shows a clear exponential improvement, denoted by the fact that the data points lie roughly on a decreasing line. The lines that are shown in Figure 6.1 are of the form $f(N) = c(2\mu)^N$, where the constant c is fitted (by hand) to the data points corresponding to the highest lengths. The BCC series has an outlier data point at $N = 12$. I believe that this is caused by a shortage of memory on the machine of the cluster Mars. The algorithm could have allocated more memory than was available on the shared machine.

It is clear that the effective $\langle\lambda\rangle$ is not of a strong exponential nature, because then the data points would lie on a steeper line than the solid lines in Figure 6.1. The constants c are $3 \cdot 10^{-10}$, $3 \cdot 10^{-10}$ and $8 \cdot 10^{-10}$, for the SC, BCC and FCC lattice respectively. I use as a rule of thumb, that if a program theoretically needs around q operations, with each operation consisting of a relatively simple piece

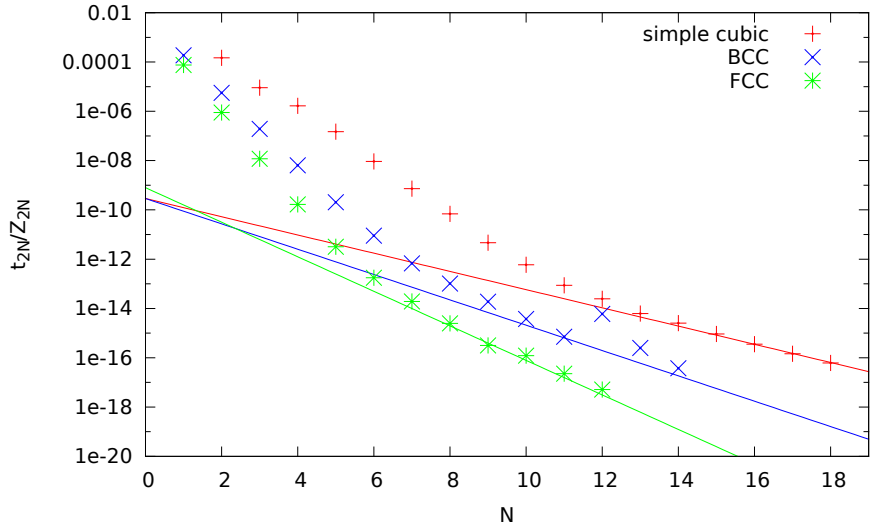


Figure 6.1: This figure shows the average number of seconds per self-avoiding walk of length $2N$ that is needed by the improved length-doubling algorithm for three lattices: simple cubic (SC), body-centered cubic (BCC) and face-centered cubic (FCC). It shows the exponential improvement of the length-doubling algorithm.

of code, then the program needs approximately $10^{-}8 \cdot q$ seconds to complete. If we correct the data for the reduction of computation time that is obtained by the exploitation of the symmetry of the lattice, this rule of thumb seems to hold. It means that $\langle\lambda\rangle$ is likely to be relatively small (order 1). Another possibility is that the effect of the extra jobs created by the algorithm on the computation time masks the λ effect. Further research is necessary to verify this.

Chapter 7

Conclusion and Discussion

The conclusion is relatively simple: the length-doubling algorithm works very efficiently after the modifications described in chapters 3 and 4. These improvements were a necessity. My bachelor thesis concerned the length-doubling algorithm without these improvements and it could also only compute the even values Z_{2N} for integer N. With that program I was barely able to find $2N = 22$, which took several hours. On top of that the scaling of the computation time of the algorithm became exponentially worse than $O((2\mu)^N)$. For comparison, our currently fastest algorithm that computes both Z_{2N} and R_{2N} computes $2N = 22$ in less than two minutes.

This is by no means the final step in the history of exact enumeration of the SAW. There are multiple leads that could bring the performance to a new level. The first idea is to form SAWs by connecting more than two SAW parts. Although I have for a long time considered this to be dead end, I still have not been able to determine tight bounds on the computation time. If the complexity of the length tripling algorithm would be close to the length-doubling algorithm, it might be interesting because I expect the memory requirements to be of the order $O(\mu^N)$ for the computation of Z_{3N}.

The next lead is an idea first brought to my attention by my supervisor and co-author Gerard Barkema. It was recently brought back to my attention by Nathan Clisby. The algorithm sums c_I dynamically, which brings the complexity to $|\mathcal{I}| = |\{I : \ c_I > 0\}| \sim (k\mu)^N$, with $k \approx 1.6$ for the SC lattice. It is a dynamic programming algorithm that follows from the equation below for the collision counts c_I for all I with $|I| < N$, which can be verified by some simple drawings:

$$c_I = \frac{1}{N - |I|} \sum_{p \notin I} c_{I \cup \{p\}} \tag{7.1}$$

Another point of interest is the computation of the tree $T(I)$ and its corresponding leaves $L(I)$. I expect this computation to be dominant when N grows to infinity. I think that this computation can be sped up significantly by using the observation that for some partial SAWs w_i in the tree $T(I)$ it can be recognized that they have equal λ count. Since it is far from trivial to show this, the proof is not discussed here, but the reader can be assured that this observation is true. The only question that remains is how many partial SAWs share the

same λ count on average. I personally expect this number to be exponential in N.

Lastly, the program that was used for the exact enumeration is not close to optimal. Careful examination of the data structures and the sorting algorithm could improve the efficiency by a constant factor or even factor logarithmic in N.

In the end, the gain of implementing these optimizations might be currently too small to justify the effort. If we assume Moore's law (which states that the number of transistors doubles every two years), the ideas could be used in a few years to create a real dent in the current record.

Acknowledgments

I would like to thank my supervisors Rob Bisseling and Gerard Barkema for their support and ideas that immensely helped the creation of the efficient length-doubling algorithm. I also gratefully acknowledge the help provided by Nathan Clisby with the series analysis. Lastly, the computations were carried out on the Huygens supercomputer at SARA in Amsterdam. I would like to thank the Dutch National Computing Facilities Foundation for providing the resources needed to carry out our computations.

Bibliography

[1] de Gennes P-G, 1979 *Scaling Concepts in Polymer Physics* (Ithaca, NY: Cornell University Press)

[2] Schram R D, Barkema G T and R H Bisseling, *Exact enumeration of self-avoiding walks*, 2011 *J. Stat. Mech P06019*

[3] Clisby N, *Accurate estimate of the critical exponent ν for self-avoiding walks via a fast implementation of the pivot algorithm*, 2010 *Phys. Rev. Lett.* 104 055702

[4] Clisby N, Liang R, Slade G, *Self-avoiding walk enumeration via the lace expansion*, 2007 *J. Phys. Math. Theor.* 40 10973

[5] Hsu H-P, Nadler W and Grassberger P, *Scaling of star polymers with 1-80 arms*, 2004 *Macromolecules* 37 4658

[6] `http://www.math.ubc.ca/~slade/lacecounts`, 2010

[7] Bertsimas D, Tsitsiklis J N, 1997 *Introduction to Linear Optimization* (Athena Scientific, Dynamic Ideas)

[8] Guttmann A J, *On the critical behaviour of self-avoiding walks*, 1987 *J. Phys. A: Math. Gen.* 20 1839

[9] Guttmann A J, *On the critical behaviour of self-avoiding walks*, 1989 *J. Phys. A: Math. Gen.* 22 2807

[10] MacDonald D, Hunter D L, Kelly K and Jan N, *Self-avoiding walks in two to five dimensions: exact enumerations and series study*, 1992 *J. Phys. A: Math. Gen.* 25 1429

[11] MacDonald D, Joseph S, Hunter D L, Moseley L L, Jan N and Guttmann A J, *Self-avoiding walks on the simple cubic lattice*, 2000 *J. Phys. A: Math. Gen.* 33 5973

Appendix A

Tabulated exact enumeration results

A.1 SC lattice

Table A.1: Exact enumeration results for the cubic lattice. The parameter N is the total length of the enumerated SAWs. For each length N we found the number of SAWs Z_N and the summed squared end-to-end distance R_N. The results for small N are not in the table for brevity. They can be found on the website [6].

N	Z_N	R_N	Year Author
1	6	6	
...	...	...	
19	10 576 033 219 614	385 066 579 325 550	
20	49 917 327 838 734	1 933 885 653 380 544	1987 Guttmann [8]
21	235 710 090 502 158	9 679 153 967 272 734	1989 Guttmann [9]
22	1 111 781 983 442 406	48 295 148 145 655 224	
23	5 245 988 215 191 414	240 292 643 254 616 694	1992 MacDonald e.a. [10]
24	24 730 180 885 580 790	1 192 504 522 283 625 600	
25	116 618 841 700 433 358	5 904 015 201 226 909 614	
26	549 493 796 867 100 942	29 166 829 902 019 914 840	2000 MacDonald e.a. [11]
27	2 589 874 864 863 200 574	143 797 743 705 453 990 030	
28	12 198 184 788 179 866 902	707 626 784 073 985 438 752	
29	57 466 913 094 951 837 030	3 476 154 136 334 368 955 958	
30	270 569 905 525 454 674 614	17 048 697 241 184 582 716 248	2007 Clisby, Liang, Slade [4]
31	1 274 191 064 726 416 905 966	83 487 969 681 726 067 169 454	
32	5 997 359 460 809 616 886 494	408 264 709 609 407 519 880 320	
33	28 233 744 272 563 685 150 118	1 993 794 711 631 386 183 977 574	
34	132 853 629 626 823 234 210 582	9 724 709 261 537 887 936 102 872	
35	625 248 129 452 557 974 777 990	47 376 158 929 939 177 384 568 598	
36	2 941 370 856 334 701 726 560 670	230 547 785 968 352 575 619 933 376	2011 Schram, Barkema, Bisseling [2]

A.2 FCC lattice

Table A.2: Exact enumeration results for the face-centered cubic (FCC) lattice. The parameter N is the total length of the enumerated SAWs. For each length N we found the number of SAWs Z_N and the summed squared end-to-end distance R_N.

N	Z_N	R_N
1	12	12
2	132	576
3	1 404	9 816
4	14 700	144 288
5	152 532	1 951 560
6	1 573 716	25 021 536
7	16 172 148	309 080 808
8	165 697 044	3 714 659 040
9	1 693 773 924	43 714 781 448
10	17 281 929 564	505 948 384 608
11	176 064 704 412	5 777 220 825 912
12	1 791 455 071 068	65 234 797 723 584
13	18 208 650 297 396	729 724 191 726 408
14	184 907 370 618 612	8 097 639 351 530 304
15	1 876 240 018 679 868	89 239 258 469 121 912
16	19 024 942 249 966 812	977 545 487 795 069 952
17	192 794 447 005 403 916	10 651 662 728 070 257 016
18	1 952 681 556 794 601 732	115 520 552 778 504 791 136
19	19 767 824 914 170 222 996	1 247 619 751 507 795 906 248
20	200 031 316 330 580 106 948	13 423 705 093 594 869 393 216
21	2 023 330 401 919 804 218 996	143 942 374 595 787 212 970 696
22	20 458 835 772 261 851 432 748	1 538 749 219 442 520 114 999 744
23	206 801 586 042 610 941 719 148	16 403 200 314 230 418 676 555 512
24	2 089 765 228 215 904 826 153 292	174 411 223 302 510 038 302 309 440

A.3 BCC lattice

Table A.3: Exact enumeration results for the body-centered cubic (BCC) lattice. The parameter N is the total length of the enumerated SAWs. For each length N we found the number of SAWs Z_N and the summed squared end-to-end distance R_N.

N	Z_N	R_N
1	8	8
2	56	384
3	392	4 248
4	2 648	40 704
5	17 960	358 008
6	120 056	2 987 232
7	804 824	23 999 880
8	5 351 720	187 661 376
9	35 652 680	1 436 494 872
10	236 291 096	10 816 140 768
11	1 568 049 560	80 339 567 112
12	10 368 669 992	590 168 152 512
13	68 626 647 608	4 294 543 350 696
14	453 032 542 040	31 003 097 851 872
15	2 992 783 648 424	222 268 142 153 784
16	19 731 335 857 592	1 583 984 756 900 544
17	130 161 040 083 608	11 228 345 566 400 136
18	857 282 278 813 256	79 223 666 339 548 320
19	5 648 892 048 530 888	556 634 161 952 309 400
20	37 175 039 569 217 672	3 896 382 415 388 139 840
21	244,738 250 638 121 768	27 181 650 674 871 447 672
22	1 609 522 963 822 562 936	189 042 890 267 974 827 744
23	10 588 362 063 533 857 304	1 311 064 323 033 684 408 072
24	69 595 035 470 413 829 144	9 069 398 712 299 296 227 648
25	457 555 628 726 692 288 712	62 590 336 418 536 387 660 248
26	3 005 966 051 800 541 943 464	431 019 462 253 450 273 360 416
27	19 752 584 641 370 206 122 344	2 962 188 054 220 614 984 412 920
28	129 712 890 849 289 264 912 904	20 319 962 284 800 857 255 045 760