Universiteit Utrecht

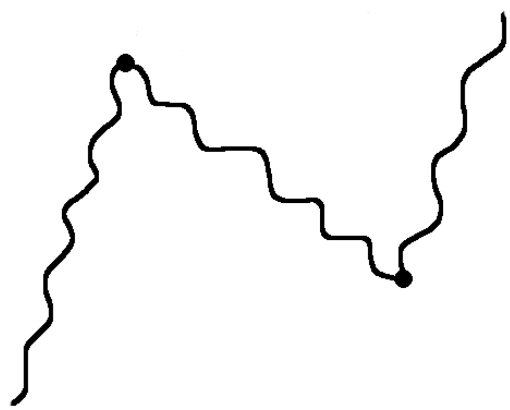
Faculteit Bètawetenschappen

Enumeration Of Self-Avoiding Walks Using Length Tripling

Bachelor Thesis

Sarita de Berg

Mathematics



Supervisor:

Prof. Dr. R. H. Bisseling
Mathematical institute, Utrecht

June 19, 2017

Abstract

In this thesis we show a new method to enumerate self-avoiding walks. The length-tripling method, which is based on the length-doubling method [12], uses three walks of length N to create walks of length $3N$. We compare this method to existing methods and find it theoretically is an improvement in some cases, but we have not seen this in practice yet.

Contents

1 Introduction

Enumeration of self-avoiding walks (SAWs) is an important combinatoral problem in statistical mechanics [9]. A self-avoiding walk is a path in a lattice, where no lattice point is visited more than once. Here, a path means that in every step we can only go to adjacent lattice points. The fundamental problem, which we study here, is counting the number of self-avoiding walks Z_N of length N. The importance of this problem derives from the use in determining critical exponents for polymers in solution, which are believed to be the same for SAWs on various lattices. If we look at Z_N, we see it behaves as

$$Z_N \approx A\mu^N N^{\gamma-1}. \tag{1}$$

Here, γ is a universal exponent which only depends on the dimension, μ is a connective constant which depends on the lattice and A is a critical amplitude. For most lattices we only have approximations for μ, for example $\mu \approx 2,63815853031$ for the square lattice [7] and $\mu \approx 4,684039931$ for the simple cubic lattice [1], but for the 2D honeycomb lattice we know that $\mu = \sqrt{2+\sqrt{2}}$ [3].

This might be an indication as to why so little research has been done to enumerate walks on the honeycomb lattice, compared to, for example, the square or cubic lattice. In [6] a short history of research to enumerate SAWs on the square lattice is given. The enumeration of SAWs on the cubic lattice [14] was first considered by Orr in 1947 [10]. He enumerated all walks up to $N = 6$ by hand. The introduction of the computer of course meant it became easier to enumerate walks. It was used by Fisher and Sykes [4] to enumerate all SAWs up to $N = 9$ in 1959. The following years this was extended further by Sykes and collaborators, until they reached 19 terms in 1972 [15]. Guttmann, who also collaborated with Sykes on reaching 19 terms, finally enumerated the walks up to 21 steps [5]. After this, some improvements were made by MacDonald et al. [8] and using a combination of the lace expansion and the two-step method SAWs were finally enumerated up to $N = 30$ by Clisby, Liang and Slade in 2007 [2]. A few years later a new method was introducted by Schram, Barkema and Bisseling [12]: the length-doubling method, where two walks of length N are used to enumerate all walks of length $2N$. Using this method, it was possible to enumerate all self-avoiding walks up to $N = 36$. This is currently the record for the simple cubic lattice.

Considering the enormous improvements made by the length-doubling method, it seems reasonable to look at the possibility of a length-tripling method, which we will consider in this thesis. In this method we use three walks of length N_1, N_2 and N_3 to enumerate all self-avoiding walks of length $N = N_1 + N_2 + N_3$. We do this in a way that is applicable to every lattice and even to other graphs. Using this method we are able to enumerate walks faster on some lattices while using less memory than previous methods.

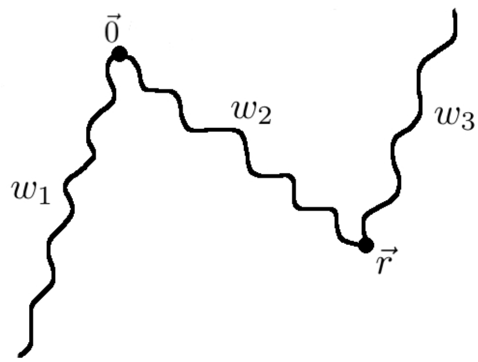


Figure 1: Construction of a walk of length N

2 The length-tripling method

In the length-tripling method, the idea is to use three walks, w_1, w_2 and w_3 of length N_1, N_2 and N_3 respectively, to create walks of length $N = N_1 + N_2 + N_3$. We construct these walks by choosing $\vec{0}$ as the starting point of w_1 and w_2 and $\vec{r}$ as the end point of w_2. Now w_3 has starting point $\vec{r}$ and, like w_1, this walk has no fixed end point. This construction is shown in Figure 1. We can now use this construction to count all SAWs of length N. We do this by first counting all self-avoiding combinations of w_1, w_2 and w_3 under these restrictions and then changing $\vec{r}$ to a new possible end point of w_2. We again count all SAWs with the new restrictions. We do this for all possible end points of w_2. Now the sum of all these counts is the number of SAWs of length N. The next section will explain how we can count the self-avoiding combinations of w_1, w_2 and w_3.

2.1 Counting combinations

We now fix $\vec{r}$. We want to count all combinations of w_1, w_2 and w_3, such that they do not intersect at any point. Because it is very hard to determine whether walks do not intersect, we look at the ones that do and based on this we can calculate our desired count. To clarify this we use the following notation

$$\begin{aligned} A &= \{(w_1, w_2, w_3) : w_1 \cap w_2 \neq \{\vec{0}\}\}, \\ B &= \{(w_1, w_2, w_3) : w_2 \cap w_3 \neq \{\vec{r}\}\}, \\ C &= \{(w_1, w_2, w_3) : w_1 \cap w_3 \neq \emptyset\}. \end{aligned}$$

Because w_1 and w_2 always intersect at $\vec{0}$ and w_2 and w_3 at $\vec{r}$, we do not consider these to be possible intersection points. We now define D as the complement of $A \cup B \cup C$. It follows that $|D|$ is the number of combinations of the three walks, such that they do not intersect each other, so this is the number we are looking for. In figure 2 it is shown how these sets are related to each other. As shown in section 3 we can determine $|A|$, $|B|$, $|C|$, $|A \cap B|$, $|A \cap C|$, $|B \cap C|$ and $|A \cap B \cap C|$ relatively easily. Using the inclusion-exclusion principle, see for instance [11], or by just looking at figure 2, we find that

$$|D| = Z_1 Z_2 Z_3 - |A| - |B| - |C| + |A \cap B| + |B \cap C| + |A \cap C| - |A \cap B \cap C|. \tag{2}$$

Here Z_n is the number of SAWs of length N_n, under the start and end point restrictions described earlier. Because the calculation of the other terms requires all walks w_1, w_2 and w_3, we immediately find Z_1, Z_2 and Z_3. An implementation of creating all these walks can be found in section 3.1, algorithm 1. In the next sections we will discuss how to calculate the other terms using walks w_1, w_2 and w_3.

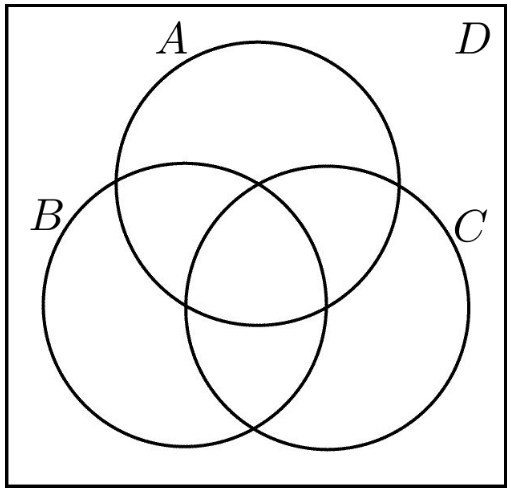


Figure 2: Venn diagram of combinations (w_1, w_2, w_3)

2.2 Calculating the first corrections

The first correction terms are $|A|$, $|B|$ and $|C|$. After we have determined all walks w_1, w_2 and w_3, we can calculate these terms using the same algorithm. The only difference in the calculation of these correction terms is whether or not $\vec{0}$ and $\vec{r}$ are considered in the calculation. In the calculation of $|A|$, we look at combinations of walks w_1 and w_2. These walks always share their starting point $\vec{0}$. This means we do not consider $\vec{0}$, but $\vec{r}$ is a possible intersection point. For $|B|$, $\vec{0}$ is considered in the calculation, but $\vec{r}$ is not. And lastly for $|C|$, we consider both $\vec{0}$ and $\vec{r}$ in the calculation.

From here on we will look at the calculation of $|A|$. This is defined as the number of walks for which $w_1 \cap w_2 \neq \{\vec{0}\}$. So we need all intersecting combinations of w_1 and w_2 and then we can combine all of these with all possible walks w_3. This results in Z_3 times something that looks at lot like the length-doubling formula, as described in [12], which determines the number of self-avoiding combinations of two walks. In the length-doubling formula, we look at all non-empty subsets S of lattice sites and for these subsets we determine the number of walks w_1 and w_2 that visit the complete subset. Because all walks have finite length, only a finite number of sites can be reached. It follows that there is only a finite number of non-empty subsets S. We define $Z_n(S)$ as the number of walks w_n that visit the entire set S. The resulting formula is

$$|A| = Z_3 \cdot \sum_{S \neq \emptyset} (-1)^{|S|+1} Z_1(S) Z_2(S). \tag{3}$$

This formula can be understood as follows. In the sum, we first add all combinations of w_1 and w_2 with at least one intersection, so $|S| = 1$. We do this by looking at all possible intersection points and adding the number of combinations that visit each of those sites. Because some of these combinations have multiple intersections, we have counted too many walks. We want to subtract all combinations that have at least two intersections. We define A_i as the set of combinations (a, b), where a behaves as w_1 and b as w_2, for which a and b visit lattice point i. We can now determine the number of combinations with at least one intersection point, by again using the inclusion-exclusion principle, which states that

$$\left| \bigcup_{i=1}^{n} A_i \right| = \sum_i |A_i| - \sum_{i<j} |A_i \cap A_j| + \sum_{i<j<k} |A_i \cap A_j \cap A_k| + ... + (-1)^{n+1} |A_1 \cap A_2 \cap ... \cap A_n| . \tag{4}$$

We defined the number of walks w_1 to visit a set S as $Z_1(S)$ and for w_2 as $Z_2(S)$. It follows that the number of combinations (a, b) that visit S is $Z_1(S) Z_2(S)$. Combining this and equation (4) we get equation (3).

2.3 Calculating the second corrections

We will now look at the calculation of the second correction terms: $|A \cap B|, |B \cap C|$ and $|A \cap C|$. We will describe the calculation of $|A \cap B|$, calculating $|B \cap C|$ and $|A \cap C|$ is done in a similar manner. This is defined as the number of combinations of walks for which $w_1 \cap w_2 \neq \{\vec{0}\}$ and $w_2 \cap w_3 \neq \{\vec{r}\}$. We now have two subsets S and T of lattice sites. Here S is the subset with points of intersection of w_1 and w_2 and T the subset with intersections of w_2 and w_3. If follows that w_1 must visit all sites in S, w_2 all sites in S and T and w_3 only the sites in T. These sets can of course contain some of the same points. Because the length of the walks is finite, it follows that only a finite number of lattice points can be reached, so we have a finite number of non-empty subsets S and T. Similarly as in calculating $|A|$, we want to look at all sets S and T and add or subtract the walks visiting these sets. We get the equation

$$|A \cap B| = \sum_{\substack{S \times T \\ S \neq \emptyset, T \neq \emptyset}} (-1)^{|S|+|T|} Z_1(S) Z_2(S \cup T) Z_3(T). \tag{5}$$

Here, we start by adding all combinations of the three walks with at least one intersection, so $|S| = |T| = 1$. But doing this we count some intersecting combinations multiple times. Now consider the case where $|S| = 2$ and $|T| = |1|$. We have already counted these walks twice, which we should not have. So we we have to subtract $Z_1(S)Z_2(S \cup T)Z_3(T)$. In the equation we get $(-1)^{|S|+|T|} = (-1)^{2+1} = -1$, so we indeed subtract this number. The case where $|T| = 2$ and $|S| = 1$ is also subtracted, following the same reasoning. But because walks can of course intersect more than just in S and T, we now have subtracted the case where $|S| = |T| = 2$ twice. This means we have to add $Z_1(S)Z_2(S \cup T)Z_3(T)$ for this case. Again we see $(-1)^{|S|+|T|} = (-1)^{2+2} = 1$. Following this argumentation for larger sizes of S and T we get equation (5).

2.4 Calculating the third corrections

We now look at calculating the third correction: $|A \cap B \cap C|$. According to the definition this is the number of combinations for which $w_1 \cap w_2 \neq \{\vec{0}\}$, $w_2 \cap w_3 \neq \{\vec{r}\}$ and $w_1 \cap w_3 \neq \emptyset$. To keep track of the different intersections we need three subsets of lattice sites, S, T and U. Here S contains the intersection points of w_1 and w_2, T of w_2 and w_3 and U of w_1 and w_3. Because both S and U consider sites of w_1, we need this walk to visit all sites in both S and U. The same holds for w_2, this walk has to visit S and T. And lastly w_3 must visit T and U. We again have a finite number of these subsets and look at all of those sets and add or subtract them. This results in the equation

$$|A \cap B \cap C| = \sum_{\substack{S \times T \times U \\ S \neq \emptyset, T \neq \emptyset, U \neq \emptyset}} (-1)^{|S|+|T|+|U|+1} Z_1(S \cup U) Z_2(S \cup T) Z_3(T \cup U). \tag{6}$$

The argumentation for this formula is about the same as for equation (5). The only difference is we now have three sets. This means that after adding $|S| = |T| = |U| = 1$, we have to subtract the cases where one of these cardinalities equals two and then add the cases where two of the cardinalities equal two. After this, we subtract the combinations where $|S| = |T| = |U| = 2$. Continuing this reasoning we find equation (6).

3 Algorithms and implementation

In this section we will discuss the algorithms used to do the calculations described in section 2.1. We will also discuss the implementation of the algorithms in the program. The implementation used in the program, is based on SAWdoubler [13], a program for counting walks using length doubling. To do all of our calculations, we first need to find all possible walks w_1, w_2 and w_3. We will describe how to do this in the next section.

3.1 Creating self-avoiding walks

To describe a walk, we need a unique numbering for the lattice sites. We will use the same numbering in our entire program. The reason for this is that in the length-tripling method we need to create new trees for all different $\vec{r}$, but using the same numbering we can reuse the tree with walks w_1. To determine what numbering works best for our problem, we first look at how we are going to store the walks. We do this using a tree data structure, just like described in [13]. In this tree we store all sites visited by a walk. Before we add a walk to the tree, we first sort the visited sites in increasing order. Suppose a walk of length N visits the set of sites $\{s_1, s_2, ..., s_N\}$, with $s_i < s_j$ for $i < j$. We now add the walk to the tree, such that $s_i = parent(s_{i+1})$. The only special site is the root of the tree, this node has site number -1. We cannot use the node with site number zero as the root of the tree, because this is not the starting point of all walks.

At every node we need to store some information, this is

- *site*, site number of the node;
- *count*, number of SAWs with this node as its highest site;
- *child*, first child of the node;
- *sibling*, next sibling when creating the tree, later next node with the same site number;
- *parent*, parent of the node;
- *stamp*, time stamp.

In the tree, the siblings are implemented as a linked list using *sibling*. The siblings are sorted by increasing site number, which makes searching for a child with a specific site number a bit faster. Later, when calculating the correction terms, *sibling* is used to find the next node with the same site number. When creating the tree *stamp* is not used, when traversing the tree it is used as a time stamp in the algorithm. The variable *count* is also used when traversing the tree to keep track of how many walks visit the set we consider.

We of course want to use as little memory as possible, so we want to make sure we can reuse a lot of nodes in the tree when adding new walks. The sites closest to the root are used most often, so we want to give these sites a low number. We also want a way to number the sites that is applicable to every lattice. We do this by using a breadth-first search starting at the middle of the lattice, which we call $\vec{0}$. This is the point we also use as the start of w_1 and w_2. The nodes are numbered in the order we encounter them in the BFS, this way the nodes closest to $\vec{0}$ have lowest site numbers.

Algorithm 1 Recursive algorithm to create all walks of length N

```
function FillTree(N, i, R, visited, T, end)
    if i = N then
        if end = −1 or R[i] = end then
            Sort(R)                              ▷ sort in increasing order
            InsertTree(R, T)
    else
        for all r ∈ Adj(R[i]) do
            if not visited[r] then               ▷ we cannot visit the same site twice in a walk
                R[i + 1] ← r
                visited[r] ← true
                FillTree(N, i + 1, R, visited, T, end)
    visited[R[i]] ← false
```

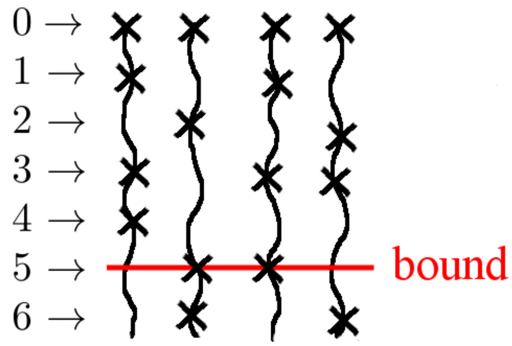


Figure 3: The bound goes up in the tree, during which we choose whether or not to add *bound* to the set S.

After we have numbered the sites, we have to create the trees. To do this we use algorithm 1, which is based on the Go function described in [13]. Before calling the function we add the root to the tree, which has -1 as site number. After this we call FillTree($N, 0, R, visited, \mathcal{T}, end$). Here N is the length of the walks we want to create, R is an array of length $N+1$ where $R[0]$ is the starting point, the array $visited$ is initially false for all values except for the starting point and $\mathcal{T}$ only contains the root node. The integer end indicates whether or not the walks need to have the same end point, if this is -1 all end points are allowed, otherwise only walks with the specified end point are added.

In the algorithm we first check the length of the walk created. If this is N and we meet the end point condition we add the sorted walk to the tree. If this is not the case, we look at all sites adjacent to r. Sites we have not visited yet we add to R and then we recursively fill R further.

3.2 The first corrections

Now that we have created the trees, we know the number of walks Z_1, Z_2 and Z_3, but as we have seen before, this is not enough. We need $|A|$, $|B|$ and $|C|$, for which we use equation (3). To calculate the different values for $Z_1(S)$ and $Z_2(S)$ we traverse up and down the tree, while adding sites to S. To clarify this we define *bound* as the maximum site that can still be included when expanding S. In figure 3 we can see what happens. The numbers on the left are site numbers. In this picture four walks of length 3 are shown. The crosses are the sites visited by the walks, for example the first walk starts in 0 and visits sites 1, 3 and 4. The sites of course do not have to be visited in this order. Because w_3 does not have 0 as starting point, it can also be the case that the lowest site number with a cross is not the starting point of the walk. Now suppose $bound = 5$, there are three options: include *bound* in S and continue expanding S, not include *bound* in S and continue expanding, include *bound* in S as its final site. After this the bound goes up to lower numbered sites, until we reach 0. This way we get all possible sets S.

In algorithm 2 we see how this is implemented. We use a bin data structure to show which nodes are active. If a node is active it means the site number of this site is included in S. The first call of the algorithm, in this case for calculating $|A|$, is $\textsc{CorrectFirstTerms}(\mathcal{T}_1, \mathcal{T}_2, Bins1, Bins2, A, r)$, where $\mathcal{T}_1$ and $\mathcal{T}_2$ are the trees that belong to the two walks we consider and $Bins1$ and $Bins2$ contain all nodes with count greater than zero, which actually are the leaves of the trees. The A shows we want to calculate $|A|$, not $|B|$ or $|C|$. Finally r is the site number of $\vec{r}$. The algorithms works as follows.

First, we determine the highest active site number, which becomes the bound. After this we check if it is possible to expand S further, if it is not we return zero. If it is possible to add more sites, we have the three previously described options.

The first option is to look at supersets $S' \supseteq S$ that do not include *bound*. Because there are no site numbers smaller than zero, we can only do this if $bound \neq 0$. We call algorithm 3 with the variable *false*, which means we do not include *bound* in the supersets. In this function we look at all nodes with site number *bound*. If the parent pv of a node v is active and we do not include *bound* in S', we add the count of the node to the count of its parent to get the number of walks that visit all sites in S and follow the same path through the tree from the root to pv. If the parent is not active we replace the count of pv by that of v and make the pv active by inserting it in the bin and giving it the current time stamp. After we have updated the counts we recursively expand S further and add the result to Z. We add this number because no sites are added to S, so the the sign in equation 3 is not changed.

The next option is to look at supersets that do include *bound*. We can only include r in S if we are looking at $|A|$ or $|C|$, so we first check if this condition holds. After this, we first have to make all nodes smaller than *bound* inactive. We do this by increasing the *time* variable and emptying the bins. Now we can use UPDATECOUNTS again, but this time *incl* is *true* because we do include *bound* in supersets. This means that for all nodes v with site number *bound* we replace the count of its parent by that of v and make the parent active. We recursively expand S further, but instead of adding we subtract this number, because we have added one site to S.

Finally we look at the contribution of $S' = S \cup bound$. To do this we need the total number of walks in $\mathcal{T}_1$ and $\mathcal{T}_2$ that visit S'. We find this by adding all counts of nodes with site number *bound* in the two different trees. We multiply these two counts like in equation (3) and add this to Z.

Algorithm 2 Recursive algorithm that calculates the first correction terms

```
function CorrectFirstTerms(T1, T2, Bins1, Bins2, mode, r, time)
    Z ← 0
    bound ←max[i : Bins1[i] ≠ ∅ or Bins2[i] ≠ ∅]                    ▷ find max active site
    if bound = −1 or (bound = 0 and mode = A) then          ▷ we cannot include zero if mode is 1
        return Z

    if bound ≠ 0 then               ▷ if bound = 0 we can only include bound in S but no more sites

        ▷ Contribution for S' ⊋ S with bound ∉ S'
        UpdateCounts(Bins1, bound, false, time)          ▷ false because we do not include bound
        UpdateCounts(Bins2, bound, false, time)
        Z ← Z + CorrectFirstTerms(T1, T2, N, Bins1, Bins2, mode, r, time)
        Restore the counts

        ▷ Contribution for S' ⊋ S with bound ∈ S'
        if bound ≠ r or mode = A or mode = C then
            time ← time + 1
            for s = 0 to bound − 1 do                                        ▷ empty the bins
                Bins1[s] = ∅
                Bins2[s] = ∅
            UpdateCounts(Bins1, bound, true, time)              ▷ true because we include bound
            UpdateCounts(Bins2, bound, true, time)
            Z ← Z − CorrectFirstTerms(T1, T2, N, Bins1, Bins2, mode, r, time)
            Restore the counts

    ▷ Contribution for S' = S ∪ {bound}
    if bound ≠ r or mode = A or mode = C then
        Z1 ← 0                                                       ▷ total walks of type 1
        Z2 ← 0                                                       ▷ total walks of type 2
        for all v ∈ Bins1[bound] do
            Z1 ← Z1 + v.count
        for all w ∈ Bins2[bound] do
            Z2 ← Z2 + w.count
        Z ← Z + Z1 · Z2
    return Z
```

Algorithm 3 Algorithm to change the counts in the tree to match the number of walks visiting the set

```
function UpdateCounts(Bins, bound, incl, time)    ▷ incl is whether or not we include bound in the set
    for v ∈ Bins[bound] do
        pv ← v.parent
        if not incl and pv.stamp = time then                                ▷ parent is active
            pv.count ← pv.count + v.count
        else
            pv.count ← v.count
            InsertBin(pv, Bins, time)
            pv.stamp ← time
```

3.3 The second and third corrections

The algorithms for calculating the second and third corrections have a lot in common with algorithm 2. The big difference is of course that we have two or three sets instead of one. This means there are a lot more options when traversing the tree. We use the same bound for the three trees and everytime we arrive at a new site we choose whether or not to add it to S and/or T and/or U. When determining whether or not a site is active, we use a different timer for each set. The consequences of adding a site to one of the sets and which timer(s) we have to check can easily be understood by looking at equation (5) and (6). For example, when calculating $|A \cap B|$ assume that we want to add a site to T. It follows that w_2, which has to visit $S \cup T$, must include the site, so we call UpdateCounts with the variable *true* and check the timers of S and T. We also do this for w_3, but w_1 does not have to visit this site so for this walk we call UpdateCounts with the variable *false*. When calculating the second correction there are exactly nine different options, they are:

1. Not including *bound* in S and T and continue expanding;
2. Including *bound* in S, but not in T and continue expanding;
3. Including *bound* in T, but not in S and continue expanding;
4. Including *bound* in S and T and continue expanding;
5. Including *bound* in S as its final site and not in T and continue expanding;
6. Including *bound* in S as its final site and in T and continue expanding;
7. Including *bound* in T as its final site and not in S and continue expanding;
8. Including *bound* in T as its final site and in S and continue expanding;
9. Close both sets if they have not been closed yet and add the number of walks

We see here we only add walks when we close S and T, of course one of these might already be closed before this. This means we only add each combination of sets S and T once. Of course a lot of these options are not always possible, for example we cannot add any more sites to S if we have already closed this set. When calculating the third corrections there are even more options, because in that case we have a third set U. The implementation of these algorithms can be found in appendix A.

4 Complexity and memory use

So far we have seen it is possible to do length tripling, but the question remains if it is better than previously used methods. Better can mean two things in this case: it can be faster and/or use less memory.

We first consider the complexity of different methods. The number of walks of length N grows as $Z_N \approx A\mu^N N^{\gamma-1}$, where the factor μ^N dominates. Here $\mu = \sqrt{2+\sqrt{2}}$ [3] for the honeycomb lattice, $\mu \approx 2,63815853031$ for the square lattice [7] and $\mu \approx= 4,684039931$ for the simple cubic lattice [1]. The naive method, enumerating brute forse using a backtracking algorithm, therefore takes $O(\mu^N)$ time. Using the two-step method Clisby, Liang and Slade [2] we were able to reduce this to about $O(4,0^N)$ for the simple cubic lattice. In the length-doubling method [12] walks of length N are used to create walks of length $2N$. First all walks of length N are enumerated and then for each SAW we look at all subsets S of lattice sites visited by this walk. For each SAW there are 2^N of those subsets, so the total complexity is $O(2^N\mu^N)$ which compares favorably to $O(\mu^{2N})$ when $\mu > 2$. This is the case for the square and simple cubic lattice.

Now we take a closer look at the length-tripling method. Suppose all three walks are of length N. We look at the different stages of our program and determine their complexity. First we create $\mathcal{T}_1$, which takes $O(\mu^N)$ time. After that we can fix $\vec{r}$, so all coming steps have to be done for all different $\vec{r}$. This means we have to multiply the complexities by the number of possible sites $\vec{r}$. On the square lattice there is a maximum of about $4N^2$ reachable sites and on the simple cubic lattice this is about $8N^3$. If we look at other dimensions, we see that for dimention d we get 2^dN^d. Now we can create the two other trees, which also takes $O(\mu^N)$ time. After that we use the length-doubling formula three times, so we get $O(2^N\mu^N)$. When calculating the second correction we look at all possible subsets S and combine these with all possible subsets T for each walk. It follows that this step is $O(2^N2^N\mu^N) = O(4^N\mu^N)$. Finally, we look at the third corrections. In this case we have three subsets we can combine, so we get $O(2^N2^N2^N\mu^N) = O(8^N\mu^N)$. All together this means we have a complexity of $O(2^dN^d8^N\mu^N)$. When d is small of course 2^dN^d does not play a big part. We see that this compares favorably to $O(\mu^{3N})$ if $\mu > \sqrt{8}$, which is the case for the simple cubic lattice. However, if we compare it to the length-method we find it does not always compare favorably when $\mu > \sqrt{8}$. For example, if we look at the simple cubic lattice we get $O(\sqrt{2}^N \cdot \sqrt{\mu}^N) = O(3,06^N)$ using length doubling and $O(8^{\frac{1}{3}N}\mu^{\frac{1}{3}N}) = 3,35^N$ using length tripling. If we want length tripling to be faster than length doubling we need

$$\sqrt{2} \cdot \sqrt{\mu} > 2\sqrt[3]{\mu}. \tag{7}$$

If follows that the length-tripling is profitable when $\mu > 8$, a lattice for which this holds is the FCC lattice [14].

We now we look at the memory use of the method. Storing all walks of length N takes $O(\mu^N)$ memory. It is of course possible to improve this a little by using a smart data structure, for example a tree. In our method we only need to save three trees, which use $O(\mu^N)$ memory, so we still use only $O(\mu^N)$ memory. This is a big improvement compared to the $O(\mu^{3N})$ used when using the naive method.

In conclusion, the method is definitely an improvent regarding memory use. Whether it is a faster method than previously used method depends on the lattice on which we want to enumerate the SAWs. For small dimentions and $\mu > 8$, the method is also an improvement regarding complexity.

5 A method using k walks

After doing length doubling and length tripling, the next logical step would be to combine k walks of length N to create a walk of length kN. After doing length tripling, this seems like a realistic step, although implementation might be difficult.

We first consider the number of corrections we need to do when combining k walks. The number of corrections is actually the same as the number of sets we need to describe all combinations of walks, so in our case these sets were A, B and C. We need a set for every combination of walks, so in general we have $\binom{k}{2}$ different sets. Every extra correction gives an extra term 2^N in the complexity, so the complexity of the last correction is

$$O(2^{\binom{k}{2}\cdot N}\mu^N).$$

But if we use more walks we also need to fix points $\vec{r_1}, ..., \vec{r_{k-2}}$, where r_i is the end point of w_{i+1} and the starting point of w_{i+2}. We now need to do the corrections for all combinations of these points, which means we actually get

$$O((2N)^{(k-2)d} \cdot 2^{\binom{k}{2}\cdot N} \cdot \mu^N).$$

If we just look at the last part, this would mean it is an improvement compared to the naive method when

$$\begin{aligned} \mu^{kN} &> 2^{\binom{k}{2}\cdot N} \cdot \mu^N \\ \mu &> \sqrt[k-1]{2^{\binom{k}{2}}}. \end{aligned} \tag{8}$$

Here we have also omitted that we actually need to do the calculations for every correction, except for the last one, k times. It would be very interesting to determine the best k for different μ. Of course memory use can also be taken into consideration, when determining the best k for the problem, because when k gets larger, less memory is used. All in all it is quite difficult to say when exactly this is going to be an improvement, but it is definity a possibility worth considering.

6 Results

We first implemented the length-doubling method, which is also used to calculate $|A|$, $|B|$ and $|C|$. Using this method we were able to enumerate walks on the simple cubic lattice up to $N = 19$. After this, we run out of memory. In table 1 we see Z_N and the time used by the naive method and the length-doubling method for some N. We see the length-doubling method is indeed a lot faster than the naive method. When looking at the even N, we recognise the complexity we found, which is $O(2^N \mu^N)$ for walks of length $2N$. The running time for odd lengths is always higher than expected because one of the walks has to be longer than $\frac{1}{2}N$, which means we have to look at more subsets S than when using walks of length $\frac{1}{2}N$.

We also implemented the second and third corrections. Sadly, the third correction does not give the right result yet, so we do not know the time used by the length-tripling method. The second corrections do seem to give the right results. However, when measuring the time used when only doing the first and second corrections, the time used is much longer than it should be theoretically. For example, creating walks of length 12 takes 168 seconds, which is very long compared to the 1,2 seconds when only calculating the first corrections. This probably means we go into recursion too many times, but we have not been able to find where this happens.

Hopefully, we will soon be able to get results for length tripling using the program.

N	Z_N	Naive method	Length doubling
8	387 966	0,62	0,02
9	1 853 886	3,3	0,03
10	8 809 878	13	0,12
11	41 934 150	Out of memory	0,16
12	198 842 742	Out of memory	0,22
13	943 974 510	Out of memory	1,1
14	4 468 911 678	Out of memory	1,4
15	21 175 146 054	Out of memory	7,2
16	100 121 875 974	Out of memory	8,8
17	473 730 252 102	Out of memory	52
18	2 237 723 684 094	Out of memory	63
19	10 576 033 219 614	Out of memory	381
20		Out of memory	Out of memory

Table 1: Time used in seconds when enumerating self-avoiding walks of length N

7 Conclusion

Enumerating self-avoiding walks is a problem that has been studied a lot in the past. In this thesis we have discussed a new method to enumerate SAWs: the length-tripling method. In this method we use three walks of length N to create walks of length $3N$. We have found that the method is a large improvement regarding memory. The time used by the method should theoretically be an improvement to the length-doubling method for $\mu > 8$, but in practice it might also be an improvement for smaller μ. So far, we have not been able to see this, because the program does not work optimally yet. The implementation of the length-doubling method does work very well, using this we were able to enumerate all self-avoiding walks on the simple cubic lattice up to $N = 19$. The problem for larger N is not time but memory, so hopefully we will be able to enumerate up to larger N using the length-tripling method.

A Implementation of the length-tripling method

C:\Users\Sarita de Berg\Documents\Scriptie\SAW9\SAW3\SAW\SAW\saw.cs 1

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace SAW
{
    class saw
    {
        static public int N;

        static void Main()
        {
            Stopwatch timer = new Stopwatch();
            timer.Start();
            int N1, N2, N3, lattice, zero;
            N1 = 2;
            N2 = 2;
            N3 = 2;
            lattice = 0;
            zero = 0;
            N = Math.Max(N1, N2 + N3);

            List<int>[] graph;
            graph = CreateGraph(lattice, ref zero);
            graph = NumberBFS(graph, zero);
            long walks = 0;
            walks = LengthTripling(graph, N1, N2, N3);
            Console.WriteLine(walks);
            Console.WriteLine(timer.Elapsed);
            Console.ReadLine();
        }

        static List<int>[] CreateGraph(int lattice, ref int zero)
        {
            List<int>[] graph;
            switch (lattice)
            {
                case 0:
                    graph = CreateSquare();
                    zero = phiSq(N, N);
                    break;
                case 1:
                    graph = CreateHoneycomb();
                    zero = phiHc(N, N / 2 + 1);
                    break;
                case 2:
                    graph = CreateCubic();
                    zero = phiCu(N, N, N);
                    break;
                default:
                    graph = new List<int>[0];
                    break;
            }

            return graph;
        }

        static List<int>[] CreateSquare()
        {
            List<int>[] AdjacencyList = new List<int>[phiSq(2 * N, 2 * N) + 1];
            for (int k = 0; k < phiSq(2 * N, 2 * N) + 1; k++)
                AdjacencyList[k] = new List<int>();
            //First we look at the Adjacencylist for the edges of our grid
            for (int i = 1; i < 2 * N; i++)
            {
                AdjacencyList[phiSq(i, 0)].Add(phiSq(i - 1, 0));
                AdjacencyList[phiSq(i, 0)].Add(phiSq(i + 1, 0));
                AdjacencyList[phiSq(i, 0)].Add(phiSq(i, 1));

                AdjacencyList[phiSq(i, 2 * N)].Add(phiSq(i - 1, 2 * N));
                AdjacencyList[phiSq(i, 2 * N)].Add(phiSq(i + 1, 2 * N));
                AdjacencyList[phiSq(i, 2 * N)].Add(phiSq(i, 2 * N - 1));
            }
            for (int j = 1; j < 2 * N; j++)
            {
                AdjacencyList[phiSq(0, j)].Add(phiSq(0, j - 1));
                AdjacencyList[phiSq(0, j)].Add(phiSq(0, j + 1));
                AdjacencyList[phiSq(0, j)].Add(phiSq(1, j));

```

```
                    AdjacencyList[phiSq(2 * N, j)].Add(phiSq(2 * N, j - 1));
                    AdjacencyList[phiSq(2 * N, j)].Add(phiSq(2 * N, j + 1));
                    AdjacencyList[phiSq(2 * N, j)].Add(phiSq(2 * N - 1, j));
                }

                //Now we look at all the corners
                AdjacencyList[phiSq(0, 0)].Add(phiSq(1, 0));
                AdjacencyList[phiSq(0, 0)].Add(phiSq(0, 1));
                AdjacencyList[phiSq(2 * N, 0)].Add(phiSq(2 * N - 1, 0));
                AdjacencyList[phiSq(2 * N, 0)].Add(phiSq(2 * N, 1));
                AdjacencyList[phiSq(0, 2 * N)].Add(phiSq(0, 2 * N - 1));
                AdjacencyList[phiSq(0, 2 * N)].Add(phiSq(1, 2 * N));
                AdjacencyList[phiSq(2 * N, 2 * N)].Add(phiSq(2 * N - 1, 2 * N));
                AdjacencyList[phiSq(2 * N, 2 * N)].Add(phiSq(2 * N, 2 * N - 1));

                //Finally we look at the middle of the grid
                for (int i = 1; i < 2 * N; i++)
                    for (int j = 1; j < 2 * N; j++)
                    {
                        AdjacencyList[phiSq(i, j)].Add(phiSq(i - 1, j));
                        AdjacencyList[phiSq(i, j)].Add(phiSq(i + 1, j));
                        AdjacencyList[phiSq(i, j)].Add(phiSq(i, j - 1));
                        AdjacencyList[phiSq(i, j)].Add(phiSq(i, j + 1));
                    }

                return AdjacencyList;
            }

            //Assigns a canonical numbering to every point from (0,0) to (2N, 2N) in the square lattice
            static int phiSq(int i, int j)
            {
                return i * (2 * N + 1) + j;
            }

            static List<int>[] CreateHoneycomb()
            {
                List<int>[] AdjacencyList = new List<int>[phiHc(2 * N, N + 2) + 1];
                for (int k = 0; k < phiHc(2 * N, 2 + N) + 1; k++)
                    AdjacencyList[k] = new List<int>();
                //First we look at the Adjacencylist for the edges of our grid
                for (int i = 1; i < 2 * N; i++)
                {
                    AdjacencyList[phiHc(i, 0)].Add(phiHc(i - 1, 0));
                    AdjacencyList[phiHc(i, 0)].Add(phiHc(i + 1, 0));

                    AdjacencyList[phiHc(i, 2 + N)].Add(phiHc(i - 1, 2 + N));
                    AdjacencyList[phiHc(i, 2 + N)].Add(phiHc(i + 1, 2 + N));

                    if (i % 2 == 1)
                        AdjacencyList[phiHc(i, 0)].Add(phiHc(i, 1));
                    else
                        AdjacencyList[phiHc(i, 2 + N)].Add(phiHc(i, N + 1));
                }
                for (int j = 1; j < 2 + N; j++)
                {
                    AdjacencyList[phiHc(0, j)].Add(phiHc(1, j));
                    AdjacencyList[phiHc(2 * N, j)].Add(phiHc(2 * N - 1, j));

                    if (j % 2 == 1)
                    {
                        AdjacencyList[phiHc(0, j)].Add(phiHc(0, j + 1));
                        AdjacencyList[phiHc(2 * N, j)].Add(phiHc(2 * N, j + 1));
                    }
                    else
                    {
                        AdjacencyList[phiHc(0, j)].Add(phiHc(0, j - 1));
                        AdjacencyList[phiHc(2 * N, j)].Add(phiHc(2 * N, j - 1));
                    }
                }

                //Now we look at all the corners
                AdjacencyList[phiHc(0, 0)].Add(phiHc(1, 0));
                AdjacencyList[phiHc(2 * N, 0)].Add(phiHc(2 * N - 1, 0));
                AdjacencyList[phiHc(0, 2 + N)].Add(phiHc(1, 2 + N));
                AdjacencyList[phiHc(2 * N, 2 + N)].Add(phiHc(2 * N - 1, 2 + N));
                if (N % 2 == 0)
                {
                    AdjacencyList[phiHc(0, 2 + N)].Add(phiHc(0, 1 + N));
                    AdjacencyList[phiHc(2 * N, 2 + N)].Add(phiHc(2 * N, N + 1));
                }

                //Finally we look at the middle of the grid
```

```
            for (int i = 1; i < 2 * N; i++)
                for (int j = 1; j < 2 + N; j++)
                {
                    AdjacencyList[phiHc(i, j)].Add(phiHc(i - 1, j));
                    AdjacencyList[phiHc(i, j)].Add(phiHc(i + 1, j));
                    if ((i + j) % 2 == 0)
                        AdjacencyList[phiHc(i, j)].Add(phiHc(i, j - 1));
                    else
                        AdjacencyList[phiHc(i, j)].Add(phiHc(i, j + 1));
                }

            return AdjacencyList;
        }

        //Assigns a canonical numbering to every point from (0,0) to (2N, 1/2 N + 1) in the honeycomb b
          lattice
        static int phiHc(int i, int j)
        {
            return i * (N + 3) + j;
        }

        static List<int>[] CreateCubic()
        {
            List<int>[] AdjacencyList = new List<int>[phiCu(2 * N, 2 * N, 2 * N) + 1];
            for (int k = 0; k < phiCu(2 * N, 2 * N, 2 * N) + 1; k++)
                AdjacencyList[k] = new List<int>();

            //First we look at the Adjacencylist for the edges of our grid
            for (int i = 1; i < 2 * N; i++)
            {
                AdjacencyList[phiCu(i, 0, 0)].Add(phiCu(i - 1, 0, 0));
                AdjacencyList[phiCu(i, 0, 0)].Add(phiCu(i + 1, 0, 0));
                AdjacencyList[phiCu(i, 0, 0)].Add(phiCu(i, 1, 0));
                AdjacencyList[phiCu(i, 0, 0)].Add(phiCu(i, 0, 1));

                AdjacencyList[phiCu(i, 2 * N, 0)].Add(phiCu(i - 1, 2 * N, 0));
                AdjacencyList[phiCu(i, 2 * N, 0)].Add(phiCu(i + 1, 2 * N, 0));
                AdjacencyList[phiCu(i, 2 * N, 0)].Add(phiCu(i, 2 * N - 1, 0));
                AdjacencyList[phiCu(i, 2 * N, 0)].Add(phiCu(i, 2 * N, 1));

                AdjacencyList[phiCu(i, 0, 2 * N)].Add(phiCu(i - 1, 0, 2 * N));
                AdjacencyList[phiCu(i, 0, 2 * N)].Add(phiCu(i + 1, 0, 2 * N));
                AdjacencyList[phiCu(i, 0, 2 * N)].Add(phiCu(i, 1, 2 * N));
                AdjacencyList[phiCu(i, 0, 2 * N)].Add(phiCu(i, 0, 2 * N - 1));

                AdjacencyList[phiCu(i, 2 * N, 2 * N)].Add(phiCu(i - 1, 2 * N, 2 * N));
                AdjacencyList[phiCu(i, 2 * N, 2 * N)].Add(phiCu(i + 1, 2 * N, 2 * N));
                AdjacencyList[phiCu(i, 2 * N, 2 * N)].Add(phiCu(i, 2 * N - 1, 2 * N));
                AdjacencyList[phiCu(i, 2 * N, 2 * N)].Add(phiCu(i, 2 * N, 2 * N - 1));
            }
            for (int j = 1; j < 2 * N; j++)
            {
                AdjacencyList[phiCu(0, j, 0)].Add(phiCu(0, j - 1, 0));
                AdjacencyList[phiCu(0, j, 0)].Add(phiCu(0, j + 1, 0));
                AdjacencyList[phiCu(0, j, 0)].Add(phiCu(1, j, 0));
                AdjacencyList[phiCu(0, j, 0)].Add(phiCu(0, j, 1));

                AdjacencyList[phiCu(2 * N, j, 0)].Add(phiCu(2 * N, j - 1, 0));
                AdjacencyList[phiCu(2 * N, j, 0)].Add(phiCu(2 * N, j + 1, 0));
                AdjacencyList[phiCu(2 * N, j, 0)].Add(phiCu(2 * N - 1, j, 0));
                AdjacencyList[phiCu(2 * N, j, 0)].Add(phiCu(2 * N, j, 1));

                AdjacencyList[phiCu(0, j, 2 * N)].Add(phiCu(0, j - 1, 2 * N));
                AdjacencyList[phiCu(0, j, 2 * N)].Add(phiCu(0, j + 1, 2 * N));
                AdjacencyList[phiCu(0, j, 2 * N)].Add(phiCu(1, j, 2 * N));
                AdjacencyList[phiCu(0, j, 2 * N)].Add(phiCu(0, j, 2 * N - 1));

                AdjacencyList[phiCu(2 * N, j, 2 * N)].Add(phiCu(2 * N, j - 1, 2 * N));
                AdjacencyList[phiCu(2 * N, j, 2 * N)].Add(phiCu(2 * N, j + 1, 2 * N));
                AdjacencyList[phiCu(2 * N, j, 2 * N)].Add(phiCu(2 * N - 1, j, 2 * N));
                AdjacencyList[phiCu(2 * N, j, 2 * N)].Add(phiCu(2 * N, j, 2 * N - 1));
            }
            for (int k = 1; k < 2 * N; k++)
            {
                AdjacencyList[phiCu(0, 0, k)].Add(phiCu(0, 0, k - 1));
                AdjacencyList[phiCu(0, 0, k)].Add(phiCu(0, 0, k + 1));
                AdjacencyList[phiCu(0, 0, k)].Add(phiCu(1, 0, k));
                AdjacencyList[phiCu(0, 0, k)].Add(phiCu(0, 1, k));

                AdjacencyList[phiCu(2 * N, 0, k)].Add(phiCu(2 * N, 0, k - 1));
                AdjacencyList[phiCu(2 * N, 0, k)].Add(phiCu(2 * N, 0, k + 1));
                AdjacencyList[phiCu(2 * N, 0, k)].Add(phiCu(2 * N - 1, 0, k));
```

```
                AdjacencyList[phiCu(2 * N, 0, k)].Add(phiCu(2 * N, 1, k));

                AdjacencyList[phiCu(0, 2 * N, k)].Add(phiCu(0, 2 * N, k - 1));
                AdjacencyList[phiCu(0, 2 * N, k)].Add(phiCu(0, 2 * N, k + 1));
                AdjacencyList[phiCu(0, 2 * N, k)].Add(phiCu(1, 2 * N, k));
                AdjacencyList[phiCu(0, 2 * N, k)].Add(phiCu(0, 2 * N - 1, k));

                AdjacencyList[phiCu(2 * N, 2 * N, k)].Add(phiCu(2 * N, 2 * N, k  - 1));
                AdjacencyList[phiCu(2 * N, 2 * N, k)].Add(phiCu(2 * N, 2 * N, k + 1));
                AdjacencyList[phiCu(2 * N, 2 * N, k)].Add(phiCu(2 * N - 1, 2 * N, k));
                AdjacencyList[phiCu(2 * N, 2 * N, k)].Add(phiCu(2 * N, 2 * N - 1, k));
            }

            //Now we look at the corners
            AdjacencyList[phiCu(0, 0, 0)].Add(phiCu(1, 0, 0));
            AdjacencyList[phiCu(0, 0, 0)].Add(phiCu(0, 1, 0));
            AdjacencyList[phiCu(0, 0, 0)].Add(phiCu(0, 0, 1));

            AdjacencyList[phiCu(2 * N, 0, 0)].Add(phiCu(2 * N - 1, 0, 0));
            AdjacencyList[phiCu(2 * N, 0, 0)].Add(phiCu(2 * N, 1, 0));
            AdjacencyList[phiCu(2 * N, 0, 0)].Add(phiCu(2 * N, 0, 1));

            AdjacencyList[phiCu(0, 2 * N, 0)].Add(phiCu(1, 2 * N, 0));
            AdjacencyList[phiCu(0, 2 * N, 0)].Add(phiCu(0, 2 * N - 1, 0));
            AdjacencyList[phiCu(0, 2 * N, 0)].Add(phiCu(0, 2 * N, 1));

            AdjacencyList[phiCu(0, 0, 2 * N)].Add(phiCu(1, 0, 2 * N));
            AdjacencyList[phiCu(0, 0, 2 * N)].Add(phiCu(0, 1, 2 * N));
            AdjacencyList[phiCu(0, 0, 2 * N)].Add(phiCu(0, 0, 2 * N - 1));

            AdjacencyList[phiCu(2 * N, 2 * N, 0)].Add(phiCu(2 * N - 1, 2 * N, 0));
            AdjacencyList[phiCu(2 * N, 2 * N, 0)].Add(phiCu(2 * N, 2 * N - 1, 0));
            AdjacencyList[phiCu(2 * N, 2 * N, 0)].Add(phiCu(2 * N, 2 * N, 1));

            AdjacencyList[phiCu(0, 2 * N, 2 * N)].Add(phiCu(1, 2 * N, 2 * N));
            AdjacencyList[phiCu(0, 2 * N, 2 * N)].Add(phiCu(0, 2 * N - 1, 2 * N));
            AdjacencyList[phiCu(0, 2 * N, 2 * N)].Add(phiCu(0, 2 * N, 2 * N - 1));

            AdjacencyList[phiCu(2 * N, 0, 2 * N)].Add(phiCu(2 * N - 1, 0, 2 * N));
            AdjacencyList[phiCu(2 * N, 0, 2 * N)].Add(phiCu(2 * N, 1, 2 * N));
            AdjacencyList[phiCu(2 * N, 0, 2 * N)].Add(phiCu(2 * N, 0, 2 * N - 1));

            AdjacencyList[phiCu(2 * N, 2 * N, 2 * N)].Add(phiCu(2 * N - 1, 2 * N, 2 * N));
            AdjacencyList[phiCu(2 * N, 2 * N, 2 * N)].Add(phiCu(2 * N, 2 * N - 1, 2 * N));
            AdjacencyList[phiCu(2 * N, 2 * N, 2 * N)].Add(phiCu(2 * N, 2 * N, 2 * N - 1));

            //Now we look at the middle of the grid
            for (int i = 1; i < 2 * N; i++)
                for (int j = 1; j < 2 * N; j++)
                    for (int k = 1; k < 2 * N; k++)
                    {
                        AdjacencyList[phiCu(i, j, k)].Add(phiCu(i - 1, j, k));
                        AdjacencyList[phiCu(i, j, k)].Add(phiCu(i + 1, j, k));
                        AdjacencyList[phiCu(i, j, k)].Add(phiCu(i, j - 1, k));
                        AdjacencyList[phiCu(i, j, k)].Add(phiCu(i, j + 1, k));
                        AdjacencyList[phiCu(i, j, k)].Add(phiCu(i, j, k + 1));
                        AdjacencyList[phiCu(i, j, k)].Add(phiCu(i, j, k - 1));
                    }
            return AdjacencyList;
        }

        static int phiCu(int i, int j, int k)
        {
            return i * (4 * N * N + 2 * N + 2) + j * (2 * N + 1) + k;
        }

        //Assigns a new numbering to the graph, the lowest numbers have the least steps from 0
        static List<int>[] NumberBFS(List<int>[] graph, int zero)
        {
            //We first want to know how many vertices are reachable from zero
            bool[] visited = new bool[graph.Length];
            int reachable = CountBFS(graph, zero, ref visited);

            List<int>[] BFSgraph = new List<int>[reachable];
            for (int i = 0; i < reachable; i++)
                BFSgraph[i] = new List<int>();
            NewGraphBFS(graph, ref BFSgraph, zero, ref visited);
            return BFSgraph;
        }

        //Counts how many vertices are reachable by walks of length N
        static int CountBFS(List<int>[] graph, int zero, ref bool[] visited)
```

```
        {
            for (int i = 0; i < visited.Length; i++)
                visited[i] = false;

            int count = 0;
            //In the array step we save the amount of steps it take to reach a point from zero
            int[] step = new int[graph.Count()];
            step[zero] = 0;

            Queue<int> q = new Queue<int>();
            q.Enqueue(zero);
            visited[zero] = true;

            while (q.Count > 0)
            {
                int a = q.Dequeue();
                //If we need more than N steps, we are done
                if (step[a] > N)
                    break;
                count += 1;

                foreach (int b in graph[a])
                    if (!visited[b])
                    {
                        visited[b] = true;
                        q.Enqueue(b);
                        step[b] = step[a] + 1;
                    }
            }

            return count;
        }

        static void NewGraphBFS(List<int>[] graph, ref List<int>[] BFSgraph, int zero, ref bool[] visited)
        {
            for (int i = 0; i < visited.Length; i++)
                visited[i] = false;
            //The new numbering
            int number = 0;
            //In the array step we save the amount of steps it take to reach a point from zero
            int[] step = new int[graph.Count()];
            step[zero] = 0;

            //In this array we save pi(i) which represents the new site number of i
            int[] pi = new int[graph.Length];

            Queue<int> q = new Queue<int>();
            q.Enqueue(zero);
            visited[zero] = true;

            while (q.Count > 0)
            {
                int a = q.Dequeue();
                //If we need more than N steps, we are done
                if (step[a] > N)
                    break;
                pi[a] = number;
                number++;

                foreach (int b in graph[a])
                    if (!visited[b])
                    {
                        visited[b] = true;
                        q.Enqueue(b);
                        step[b] = step[a] + 1;
                    }
            }

            //We now translate edges in the original numbering to the new numbering
            //We first look at the special site zero
            foreach (int i in graph[zero])
                BFSgraph[0].Add(pi[i]);
            for (int j = 0; j < pi.Length; j++)
                //If pi[j] > 0, this means the site is used in the new numbering
                if (pi[j] > 0)
                {
                    foreach (int k in graph[j])
                        if (pi[k] > 0 || k == zero)
                            BFSgraph[pi[j]].Add(pi[k]);
                }
        }
```

```
        //Recursive function that creates a tree with all walks of length N with startpoint start en endpoint end
        //If end is -1, all possible endpoints are allowed
        //The variable walks shows the number of walks in the tree
        static List<Node> CreateTree(int N, int start, int end, List<int>[] graph, ref long walks)
        {
            bool[] visited = new bool[graph.Length];
            //In this array we save the walk before we add it to the tree
            int[] R = new int[N + 1];
            R[0] = start;

            List<Node> T = new List<Node>();
            Node tree = new Node();
            tree.newNode(-1, 0, null, null, null);
            T.Add(tree);

            visited[start] = true;
            if (N != 0)
                FillTree(N, 0, R, visited, ref T, end, graph, ref walks);
            return T;
        }

        static void FillTree(int N, int i, int[] R, bool[] visited, ref List<Node> T, int end, List<int>[] graph, ref long walks)
        {
            if (i == N)
            {
                if (end == -1 || R[i] == end)
                {
                    //We always want to have the starting point as the first element, we sort the rest of the array
                    int[] Rsort = new int[R.Length];
                    for (int j = 0; j < R.Length; j++)
                        Rsort[j] = R[j];
                    Array.Sort(Rsort);
                    walks += 1;
                    InsertTree(Rsort, ref T);
                }
            }
            else
            {
                foreach (int r in graph[R[i]])
                    if (!visited[r])
                    {
                        R[i + 1] = r;
                        visited[r] = true;
                        FillTree(N, i + 1, R, visited, ref T, end, graph, ref walks);
                    }
            }
            visited[R[i]] = false;
        }

        static void InsertTree(int[] R, ref List<Node> T)
        {
            Node current = T.First();

            //This is the first node we have to add to the tree
            Node Ri = new Node();
            T.Add(Ri);
            int i = 0;
            while (i < R.Length)
            {
                //If the current node doesn't have any children, we know we have to add the rest of R to the tree
                if (current.child != null)
                    current = current.child;
                else
                {
                    Ri.newNode(R[i], 0, null, null, current);
                    current.child = Ri;
                    break;
                }
                bool found = false;
                //We don't have to add a node to the tree if current or any of his siblings has the same site number as R[i]
                if (current.site == R[i])
                {
                    i++;
                    found = true;
                }
                else
```

```
                {
                    //We have to add a firstchild
                    if (current.site > R[i])
                    {
                        Ri.newNode(R[i], 0, null, current, current.parent);
                        current.parent.child = Ri;
                        break;
                    }
                    else while (current.sibling != null && current.sibling.site <= R[i])
                        {
                            current = current.sibling;
                            if (current.site == R[i])
                            {
                                i++;
                                found = true;
                                break;
                            }
                        }
                }
                //Because we know current node is smaller than R[i] and the next greater we know the place in the linked list of siblings we want to insert R[i]
                if (!found)
                {
                    Ri.newNode(R[i], 0, null, current.sibling, current.parent);
                    current.sibling = Ri;
                    break;
                }
            }
            //We have to add one to the count of the last site
            if (i == R.Length - 1)
                Ri.count++;
            else if (i == R.Length)
                current.count++;

            else
            {
                Node previous = Ri;
                for (int j = i + 1; j < R.Length - 1; j++)
                {
                    Node r = new Node();
                    r.newNode(R[j], 0, null, null, previous);
                    T.Add(r);
                    previous.child = r;
                    previous = r;
                }
                Node last = new Node();
                last.newNode(R[R.Length - 1], 1, null, null, previous);
                T.Add(last);
                previous.child = last;
            }
        }

        //Determines the number of SAW using three walks of length N1, N2 and N3
        static long LengthTripling(List<int>[] graph, int N1, int N2, int N3)
        {
            //The number of self avoiding walks using length tripling
            long totalSAW = 0;

            int bound = graph.Length - 1;

            long time = 0;
            long timeS = 0;
            long timeT = 0;
            long timeU = 0;
            long D;
            List<long> counts1 = new List<long>();
            List<long> counts2 = new List<long>();
            List<long> counts3 = new List<long>();

            long walks = 0;
            long Z1, Z2, Z3;
            int max1 = bound; int max2 = bound; int max3 = bound;
            List<Node> TreeR = CreateTree(N2, 0, -1, graph, ref walks);

            walks = 0;
            List<Node> T1 = CreateTree(N1, 0, -1, graph, ref walks);
            Z1 = walks;
            long[] countsT1 = SaveCounts(T1);
            //for all end points of w2
            for (int r = 1; r < bound + 1; r++)
            {
                //D is the number of walks with the restricted end point of w2
```

```
                D = 0;
                walks = 0;
                List<Node> T2 = CreateTree(N2, 0, r, graph, ref walks);

                //If there are no walks 2 that have end point r we can stop
                if (walks > 0)
                {
                    Z2 = walks;
                    long[] countsT2 = SaveCounts(T2);
                    walks = 0;
                    List<Node> T3 = CreateTree(N3, r, -1, graph, ref walks);
                    Z3 = walks;
                    long[] countsT3 = SaveCounts(T3);

                    D = Z1 * Z2 * Z3;

                    //The first corrections
                    max1 = bound; max2 = bound; counts1.Clear(); counts2.Clear(); time = 0;
                    Node[] Bins1 = InitBins(T1, ref max1, 1, 1);
                    Node[] Bins2 = InitBins(T2, ref max2, 1, 1);
                    D = D - Z3 * CorrectFirstTerms(T1, T2, bound, Bins1, Bins2, ref time, 1, r, counts1, counts2);

                    max2 = bound; max3 = bound; counts2.Clear(); counts3.Clear(); ResetTree(T2, countsT2); time = 0;
                    Bins2 = InitBins(T2, ref max2, 1, 2);
                    Node[] Bins3 = InitBins(T3, ref max3, 1, 2);
                    D = D - Z1 * CorrectFirstTerms(T2, T3, bound, Bins2, Bins3, ref time, 2, r, counts2, counts3);

                    max1 = bound; max3 = bound; counts1.Clear(); counts3.Clear(); ResetTree(T1, countsT1); ResetTree(T3, countsT3); time = 0;
                    Bins1 = InitBins(T1, ref max1, 1, 3);
                    Bins3 = InitBins(T3, ref max3, 1, 3);
                    D = D - Z2 * CorrectFirstTerms(T1, T3, bound, Bins1, Bins3, ref time, 3, r, counts1, counts3);

                    //The second corrections
                    max1 = bound; max2 = bound; max3 = bound; counts1.Clear(); counts2.Clear(); counts3.Clear(); timeS = 0; timeT = 0; ResetTree(T1, countsT1); ResetTree(T2, countsT2); ResetTree(T3, countsT3);
                    Bins1 = InitBins(T1, ref max1, 2, 1);
                    Bins2 = InitBins(T2, ref max2, 2, 1);
                    Bins3 = InitBins(T3, ref max3, 2, 1);
                    D = D + CorrectSecondTerms(T1, T2, T3, bound, -1, -1, Bins1, Bins2, Bins3, ref timeS, ref timeT, 1, r, counts1, counts2, counts3);

                    max1 = bound; max2 = bound; max3 = bound; counts1.Clear(); counts2.Clear(); counts3.Clear(); timeS = 0; timeT = 0; ResetTree(T1, countsT1); ResetTree(T2, countsT2); ResetTree(T3, countsT3);
                    Bins1 = InitBins(T1, ref max1, 2, 2);
                    Bins2 = InitBins(T2, ref max2, 2, 2);
                    Bins3 = InitBins(T3, ref max3, 2, 2);
                    D = D + CorrectSecondTerms(T2, T1, T3, bound, -1, -1, Bins2, Bins1, Bins3, ref timeS, ref timeT, 2, r, counts2, counts1, counts3);

                    max1 = bound; max2 = bound; max3 = bound; counts1.Clear(); counts2.Clear(); counts3.Clear(); timeS = 0; timeT = 0; ResetTree(T1, countsT1); ResetTree(T2, countsT2); ResetTree(T3, countsT3);
                    Bins1 = InitBins(T1, ref max1, 2, 3);
                    Bins2 = InitBins(T2, ref max2, 2, 3);
                    Bins3 = InitBins(T3, ref max3, 2, 3);
                    D = D + CorrectSecondTerms(T3, T1, T2, bound, -1, -1, Bins3, Bins1, Bins2, ref timeS, ref timeT, 3, r, counts3, counts1, counts2);

                    //The third corrections
                    max1 = bound; max2 = bound; max3 = bound; counts1.Clear(); counts2.Clear(); counts3.Clear(); timeS = 0; timeT = 0; timeU = 0; ResetTree(T1, countsT1); ResetTree(T2, countsT2); ResetTree(T3, countsT3);
                    Bins1 = InitBins(T1, ref max1, 3, 1);
                    Bins2 = InitBins(T2, ref max2, 3, 2);
                    Bins3 = InitBins(T3, ref max3, 3, 3);
                    D = D - CorrectThirdTerms(T1, T2, T3, bound, -1, -1, -1, Bins1, Bins2, Bins3, ref timeS, ref timeT, ref timeU, r, counts1, counts2, counts3);
                    ResetTree(T1, countsT1);

                    totalSAW += D;
                }
            }
            return totalSAW;
        }

```

```
        //Stores the counts of the tree in an array
        static long[] SaveCounts(List<Node> Tree)
        {
            long[] counts = new long[Tree.Count()];
            int i = 0;
            foreach (Node node in Tree)
            {
                counts[i] = node.count;
                i++;
            }
            return counts;
        }

        //Resets the counts of the tree
        static void ResetTree(List<Node> Tree, long[] counts)
        {
            int i = 0;
            foreach (Node node in Tree)
            {
                node.count = counts[i];
                node.stamp1 = -1;
                node.stamp2 = -1;
                node.stamp3 = -1;
                i++;
            }
        }

        //Calculates the first order correction terms
        //If both walks have the same start or end point r we don't use this point as a possible intersection point
        //The int mode indicates which term we are going to calculate: 1 for |A|, 2 for |B|, 3 for | C|
        //We have to restore the counts later, so we save them in counts1 and counts2
        static long CorrectFirstTerms(List<Node> T1, List<Node> T2, int maxsite, Node[] Bins1, Node[] Bins2, ref long time, int mode, int r, List<long> counts1, List<long> counts2)
        {
            long Z = 0;

            int bound = -1;
            //We find the highest site number for which the time stamp is time, so the highest activ e site
            for (int i = maxsite; i >= 0; i--)
                if ((Bins1[i] != null && Bins1[i].stamp1 == time) || (Bins2[i] != null && Bins2[i].stamp1 == time))
                {
                    bound = i;
                    break;
                }

            if (bound == -1 || (bound == 0 && mode == 1))
                return Z;
            //If bound = 0 we can only include bound in S and no more sites
            if (bound != 0)
            {
                counts1.Clear();
                counts2.Clear();
                int max = 0;
                //We first look at the contribution for supersets of S not including bound
                UpdateCounts(Bins1, counts1, bound, false, time, ref max);
                UpdateCounts(Bins2, counts2, bound, false, time, ref max);

                Z = Z + CorrectFirstTerms(T1, T2, bound - 1, Bins1, Bins2, ref time, mode, r, counts1, counts2);
                RestoreCounts(counts1, Bins1[bound]);
                RestoreCounts(counts2, Bins2[bound]);
                counts1.Clear();
                counts2.Clear();

                if (bound != r || (mode == 1 || mode == 3))
                {
                    //empty bins and make nodes inactive by increasing the time stamp
                    time += 1;
                    for (int s = 0; s < bound; s++)
                    {
                        Bins1[s] = null;
                        Bins2[s] = null;
                    }
                    max = 0;
                    UpdateCounts(Bins1, counts1, bound, true, time, ref max);
                    UpdateCounts(Bins2, counts2, bound, true, time, ref max);
                    Z = Z - CorrectFirstTerms(T1, T2, max, Bins1, Bins2, ref time, mode, r, counts1, counts2);
                    RestoreCounts(counts1, Bins1[bound]);
```

```
                    RestoreCounts(counts2, Bins2[bound]);
                }
            }
            //We now look at the contribution of S including bound
            if (bound != r || (mode == 1 || mode == 3))
            {
                long Z1 = CalcCount(Bins1[bound]);
                long Z2 = CalcCount(Bins2[bound]);
                Z = Z + Z1 * Z2;
            }
            return Z;
        }

        //Calculates the second order correction terms
        //T1 is the tree we are going to intersect with, so suppose we want |A cap B| than T1 is from w2, T2 from w1 and T3 from w3
        //S is the intersection set of T1 and T2 and T of T1 and T3
        //The int mode also shows which tree we are going to intersect with, so in this case that is 2
        //If we close S before T or T before S, smax or tmax is the final site add to S or T, this is -1 if it has not been closed
        //TimeS and timeT are the timestamps for S and T, they are the number of include operations we have done
        static long CorrectSecondTerms(List<Node> T1, List<Node> T2, List<Node> T3, int maxsite, int smax, int tmax, Node[] Bins1, Node[] Bins2, Node[] Bins3,
            ref long timeS, ref long timeT, int mode, int r, List<long> counts1, List<long> counts2, List<long> counts3)
        {
            long Z = 0;

            int bound = -1;
            //We find the highest site number for which the time stamp is time, so the highest active site
            for (int i = maxsite; i >= 0; i--)
                if ((Bins1[i] != null && CheckTime(timeS, timeT, 1, Bins1[i])) || (Bins2[i] != null && Bins2[i].stamp1 == timeS || (Bins3[i] != null && Bins3[i].stamp2 == timeT)))
                {
                    bound = i;
                    break;
                }

            //If we are in mode 1 or 2 we can only add 0 to T, if S has not been closed we can stop
            if (bound == -1 || (bound == 0 && (mode == 1 || mode == 2) && smax <= 0))
                return Z;

            //If bound = 0 we can only add bound to T
            if (bound != 0)
            {
                int max = 0;
                //We first look at the contribution of supersets S and T not including bound
                Node site1, site2, site3;
                counts1.Clear(); counts2.Clear(); counts3.Clear();
                site1 = Bins1[bound]; site2 = Bins2[bound]; site3 = Bins3[bound];
                UpdateCounts2(Bins1, counts1, bound, false, timeS, timeT, 1, ref max);
                UpdateCounts2(Bins2, counts2, bound, false, timeS, timeT, 2, ref max);
                UpdateCounts2(Bins3, counts3, bound, false, timeS, timeT, 3, ref max);
                Z = Z + CorrectSecondTerms(T1, T2, T3, bound - 1, smax, tmax, Bins1, Bins2, Bins3, ref timeS, ref timeT, mode, r, counts1, counts2, counts3);
                RestoreCounts(counts1, site1); RestoreCounts(counts2, site2); RestoreCounts(counts3, site3);

                if (smax <= 0 && Bins2[bound] != null)
                {
                    //Now we look at supersets where S does contain bound but T does not
                    Z = Z - CorrectSec(T1, T2, T3, smax, tmax, Bins1, Bins2, Bins3, ref timeS, ref timeT, mode, r, counts1, counts2, counts3, bound, 0, 0, 1);
                    //We now consider the case where bound is the final site added to S and we do not add bound to T
                    if (tmax <= 0)
                    {
                        Z = Z + CorrectSec(T1, T2, T3, bound, tmax, Bins1, Bins2, Bins3, ref timeS, ref timeT, mode, r, counts1, counts2, counts3, bound, 0, 2, 1);
                    }
                }

                if (tmax <= 0 && !(bound == r && (mode == 2 || mode == 3)) && Bins3[bound] != null)
                {
                    //Now we look at supersets where T does contain bound but S does not
                    Z = Z - CorrectSec(T1, T2, T3, smax, tmax, Bins1, Bins2, Bins3, ref timeS, ref timeT, mode, r, counts1, counts2, counts3, bound, 0, 1, 0);

                    //We now consider the case where bound is the final site added to T and we do not add bound to S
```

```
                    if (smax <= 0)
                    {
                        Z = Z + CorrectSec(T1, T2, T3, smax, bound, Bins1, Bins2, Bins3, ref timeS, ref timeT, mode, r, counts1, counts2, counts3, bound, 0, 1, 2);
                    }
                }

                if (smax <= 0 && (tmax <= 0 && !(bound == r && (mode == 2 || mode == 3))) && Bins3[bound] != null && Bins2[bound] != null)
                {
                    //We now look at supersets where both S and T contain bound
                    Z = Z + CorrectSec(T1, T2, T3, smax, tmax, Bins1, Bins2, Bins3, ref timeS, ref timeT, mode, r, counts1, counts2, counts3, bound, 0, 0, 0);

                    //We now consider the case where bound is the final site added to S and we do add bound to T
                    Z = Z - CorrectSec(T1, T2, T3, bound, tmax, Bins1, Bins2, Bins3, ref timeS, ref timeT, mode, r, counts1, counts2, counts3, bound, 0, 2, 0);

                    //We now consider the case where bound is the final site added to T and we do add bound to S
                    Z = Z - CorrectSec(T1, T2, T3, smax, bound, Bins1, Bins2, Bins3, ref timeS, ref timeT, mode, r, counts1, counts2, counts3, bound, 0, 0, 2);
                }
            }

            if (tmax > 0 || !(bound == r && (mode == 2 || mode == 3)))
            {
                long Z1, Z2, Z3;
                if (smax <= 0)
                    smax = bound;
                if (tmax <= 0)
                    tmax = bound;
                Z1 = CalcCount(Bins1[bound]);
                Z2 = CalcCount(Bins2[smax]);
                Z3 = CalcCount(Bins3[tmax]);
                Z = Z + Z1 * Z2 * Z3;
            }
            return Z;
        }


        //Function that updates the counts and calculates the result of the recursion of the correctionterms
        //incl1, incl2, incl3 show how we want to update the count: 0 for true, 1 for false and 2 for not updating
        static long CorrectSec(List<Node> T1, List<Node> T2, List<Node> T3, int smax, int tmax, Node[] Bins1, Node[] Bins2, Node[] Bins3,
            ref long timeS, ref long timeT, int mode, int r, List<long> counts1, List<long> counts2, List<long> counts3, int bound, int incl1, int incl2, int incl3)
        {
            long result = 0;
            if (incl2 != 1) timeS++;
            if (incl3 != 1) timeT++;
            for (int s = 0; s < bound; s++)
            {
                Bins1[s] = null;
                if (incl2 != 1)
                    Bins2[s] = null;
                if (incl3 != 1)
                    Bins3[s] = null;
            }
            int max = 0;
            counts1.Clear(); counts2.Clear(); counts3.Clear();
            Node site1 = Bins1[bound]; Node site2 = Bins2[bound]; Node site3 = Bins3[bound];
            if (incl1 != 1) UpdateCounts2(Bins1, counts1, bound, true, timeS, timeT, 1, ref max);
            else UpdateCounts2(Bins1, counts1, bound, false, timeS, timeT, 1, ref max);
            if (incl2 != 1) UpdateCounts2(Bins2, counts2, bound, true, timeS, timeT, 2, ref max);
            else UpdateCounts2(Bins2, counts2, bound, false, timeS, timeT, 2, ref max);
            if (incl3 != 1) UpdateCounts2(Bins3, counts3, bound, true, timeS, timeT, 3, ref max);
            else UpdateCounts2(Bins3, counts3, bound, false, timeS, timeT, 3, ref max);

            if (incl1 != 1 && incl2 != 1 && incl3 != 1)
                result = CorrectSecondTerms(T1, T2, T3, max, smax, tmax, Bins1, Bins2, Bins3, ref timeS, ref timeT, mode, r, counts1, counts2, counts3);
            else
                result = CorrectSecondTerms(T1, T2, T3, bound - 1, smax, tmax, Bins1, Bins2, Bins3, ref timeS, ref timeT, mode, r, counts1, counts2, counts3);
            RestoreCounts(counts1, site1);
            RestoreCounts(counts2, site2);
            RestoreCounts(counts3, site3);

```

```
            return result;
        }

        //Calculates the third order correction terms
        //S is the intersection set of T1 and T2, T of T2 and T3 and U of T1 and T3
        //If we close one of the sets, smax, tmax or is the final site added to that set, this is  -1 if it has not been closed
        static long CorrectThirdTerms(List<Node> T1, List<Node> T2, List<Node> T3, int maxsite, int smax, int tmax, int umax, Node[] Bins1, Node[] Bins2, Node[] Bins3,
            ref long timeS, ref long timeT, ref long timeU, int r, List<long> counts1, List<long> counts2, List<long> counts3)
        {
            long Z = 0;

            int bound = -1;
            //We find the highest site number for which the time stamp is time, so the highest active site
            for (int i = maxsite; i >= 0; i--)
                if ((Bins1[i] != null && CheckTime2(timeS, timeT, timeU, 1, Bins1[i])) || (Bins2[i]  != null && CheckTime2(timeS, timeT, timeU, 2, Bins2[i])) || (Bins3[i] != null && CheckTime2(timeS, timeT, timeU, 3, Bins3[i])))
                {
                    bound = i;
                    break;
                }

            //If S has not been closed yet and bound is zero, we can stop
            if (bound == -1 || (bound == 0 && smax <= 0))
                return Z;

            //1: S and U, 2: S and T, 3: T and U
            //If bound = 0 we can only add bound to T and/or U
            if (bound != 0)
            {
                int max = 0;
                //We first look at the contribution of supersets S, T and U not including bound
                Node site1, site2, site3;
                counts1.Clear(); counts2.Clear(); counts3.Clear();
                site1 = Bins1[bound]; site2 = Bins2[bound]; site3 = Bins3[bound];
                UpdateCounts3(Bins1, counts1, bound, false, timeS, timeT, timeU, 1, ref max);
                UpdateCounts3(Bins2, counts2, bound, false, timeS, timeT, timeU, 2, ref max);
                UpdateCounts3(Bins3, counts3, bound, false, timeS, timeT, timeU, 3, ref max);
                Z = Z + CorrectThirdTerms(T1, T2, T3, bound - 1, smax, tmax, umax, Bins1, Bins2, Bins3, ref timeS, ref timeT, ref timeU, r, counts1, counts2, counts3);
                RestoreCounts(counts1, site1); RestoreCounts(counts2, site2); RestoreCounts(counts3, site3);

                //We cannot add any more sites if two sets are close
                if ((smax <= 0 && (tmax <= 0 || umax <= 0)) || (tmax <= 0 && umax <= 0))
                {
                    //Now bound is added to S, but not to T and U, in the second case as final site
                    if (smax <= 0 && Bins1[bound] != null && Bins2[bound] != null)
                    {
                        Z = Z - CorrectThree(T1, T2, T3, smax, tmax, umax, Bins1, Bins2, Bins3, ref timeS, ref timeT, ref timeU, r, counts1, counts2, counts3, bound, 0, 1, 1);
                        Z = Z + CorrectThree(T1, T2, T3, bound, tmax, umax, Bins1, Bins2, Bins3,  ref timeS, ref timeT, ref timeU, r, counts1, counts2, counts3, bound, 2, 1, 1);
                        //We also add bound to T
                        if (tmax <= 0 && bound != r && Bins3 != null)
                        {
                            Z = Z + CorrectThree(T1, T2, T3, smax, tmax, umax, Bins1, Bins2, Bins3, ref timeS, ref timeT, ref timeU, r, counts1, counts2, counts3, bound, 0, 0, 1);
                            Z = Z - CorrectThree(T1, T2, T3, bound, tmax, umax, Bins1, Bins2, Bins3, ref timeS, ref timeT, ref timeU, r, counts1, counts2, counts3, bound, 2, 0, 1);
                            Z = Z - CorrectThree(T1, T2, T3, smax, bound, umax, Bins1, Bins2, Bins3, ref timeS, ref timeT, ref timeU, r, counts1, counts2, counts3, bound, 0, 2, 1);
                            //We can only close both sets if U has not been closed yet
                            if (umax <= 0)
                                Z = Z + CorrectThree(T1, T2, T3, bound, bound, umax, Bins1, Bins2, Bins3, ref timeS, ref timeT, ref timeU, r, counts1, counts2, counts3, bound, 2, 2, 1);
                            //We also add bound to U
                            if (umax <= 0)
                            {
                                Z = Z - CorrectThree(T1, T2, T3, smax, tmax, umax, Bins1, Bins2, Bins3, ref timeS, ref timeT, ref timeU, r, counts1, counts2, counts3, bound, 0, 0, 0);
                                Z = Z + CorrectThree(T1, T2, T3, bound, tmax, umax, Bins1, Bins2, Bins3, ref timeS, ref timeT, ref timeU, r, counts1, counts2, counts3, bound, 2, 0, 0);
```

```
                                Z = Z + CorrectThree(T1, T2, T3, smax, bound, umax, Bins1, Bins2, Bins3, ref timeS, ref timeT, ref timeU, r, counts1, counts2, counts3, bound, 0, 2, 0);
                                Z = Z + CorrectThree(T1, T2, T3, smax, tmax, bound, Bins1, Bins2, Bins3, ref timeS, ref timeT, ref timeU, r, counts1, counts2, counts3, bound, 0, 0, 2);
                                Z = Z - CorrectThree(T1, T2, T3, bound, bound, umax, Bins1, Bins2, Bins3, ref timeS, ref timeT, ref timeU, r, counts1, counts2, counts3, bound, 2, 2, 0);
                                Z = Z - CorrectThree(T1, T2, T3, bound, tmax, bound, Bins1, Bins2, Bins3, ref timeS, ref timeT, ref timeU, r, counts1, counts2, counts3, bound, 2, 0, 2);
                                Z = Z - CorrectThree(T1, T2, T3, smax, bound, bound, Bins1, Bins2, Bins3, ref timeS, ref timeT, ref timeU, r, counts1, counts2, counts3, bound, 0, 2, 2);
                            }
                        }
                        //We do not add bound to T, but we do add it to U
                        if (umax <= 0 && Bins3[bound] != null)
                        {
                            Z = Z + CorrectThree(T1, T2, T3, smax, tmax, umax, Bins1, Bins2, Bins3, ref timeS, ref timeT, ref timeU, r, counts1, counts2, counts3, bound, 0, 1, 0);
                            Z = Z - CorrectThree(T1, T2, T3, bound, tmax, umax, Bins1, Bins2, Bins3, ref timeS, ref timeT, ref timeU, r, counts1, counts2, counts3, bound, 2, 1, 0);
                            Z = Z - CorrectThree(T1, T2, T3, smax, tmax, bound, Bins1, Bins2, Bins3, ref timeS, ref timeT, ref timeU, r, counts1, counts2, counts3, bound, 0, 1, 2);
                            if (tmax <= 0)
                                Z = Z + CorrectThree(T1, T2, T3, bound, tmax, bound, Bins1, Bins2, Bins3, ref timeS, ref timeT, ref timeU, r, counts1, counts2, counts3, bound, 2, 1, 2);
                        }
                    }
                    //We add bound to T, first without closing it then with
                    if (tmax <= 0 && bound != r && Bins2[bound] != null && Bins3[bound] != null)
                    {
                        Z = Z - CorrectThree(T1, T2, T3, smax, tmax, umax, Bins1, Bins2, Bins3, ref timeS, ref timeT, ref timeU, r, counts1, counts2, counts3, bound, 1, 0, 1);
                        Z = Z + CorrectThree(T1, T2, T3, smax, bound, umax, Bins1, Bins2, Bins3, ref timeS, ref timeT, ref timeU, r, counts1, counts2, counts3, bound, 1, 2, 1);
                        //We also add bound to U
                        if (umax <= 0 && Bins1[bound] != null)
                        {
                            Z = Z + CorrectThree(T1, T2, T3, smax, tmax, umax, Bins1, Bins2, Bins3, ref timeS, ref timeT, ref timeU, r, counts1, counts2, counts3, bound, 1, 0, 0);
                            Z = Z - CorrectThree(T1, T2, T3, smax, bound, umax, Bins1, Bins2, Bins3, ref timeS, ref timeT, ref timeU, r, counts1, counts2, counts3, bound, 1, 2, 0);
                            Z = Z - CorrectThree(T1, T2, T3, smax, tmax, bound, Bins1, Bins2, Bins3, ref timeS, ref timeT, ref timeU, r, counts1, counts2, counts3, bound, 1, 0, 2);
                            //We can only close the two sets if S has not been closed yet
                            if (smax <= 0)
                                Z = Z + CorrectThree(T1, T2, T3, smax, bound, bound, Bins1, Bins2, Bins3, ref timeS, ref timeT, ref timeU, r, counts1, counts2, counts3, bound, 1, 2, 2);
                        }
                    }
                    //We only add bound to U
                    if (umax <= 0 && Bins1[bound] != null && Bins3[bound] != null)
                    {
                        Z = Z - CorrectThree(T1, T2, T3, smax, tmax, umax, Bins1, Bins2, Bins3, ref timeS, ref timeT, ref timeU, r, counts1, counts2, counts3, bound, 1, 1, 0);
                        Z = Z + CorrectThree(T1, T2, T3, smax, tmax, bound, Bins1, Bins2, Bins3, ref timeS, ref timeT, ref timeU, r, counts1, counts2, counts3, bound, 1, 1, 2);
                    }
                }
            }

            if (tmax > 0 || bound != r)
            {
                long Z1, Z2, Z3;
                int final1 = bound;
                int final2 = bound;
                int final3 = bound;
                if (smax > 0)
                {
                    if (umax > 0)
                        final1 = Math.Min(smax, umax);
                    else if (tmax > 0)
                        final2 = Math.Min(smax, tmax);
```

```
                }
                if (tmax > 0 && umax > 0)
                    final3 = Math.Min(tmax, umax);
                Z1 = CalcCount(Bins1[final1]);
                Z2 = CalcCount(Bins2[final2]);
                Z3 = CalcCount(Bins3[final3]);

                Z = Z + Z1 * Z2 * Z3;
            }
            return Z;
        }

        static long CorrectThree(List<Node> T1, List<Node> T2, List<Node> T3, int smax, int tmax, int umax, Node[] Bins1, Node[] Bins2, Node[] Bins3,
            ref long timeS, ref long timeT, ref long timeU, int r, List<long> counts1, List<long> counts2, List<long> counts3, int bound, int inclS, int inclT, int inclU)
        {
            long result = 0;
            if (inclS != 1) timeS++;
            if (inclT != 1) timeT++;
            if (inclU != 1) timeU++;
            //If we add bound to the sets that belong to a walk we have to empty the bins
            for (int s = 0; s < bound; s++)
            {
                if (inclS != 1 || inclU != 1)
                    Bins1[s] = null;
                if (inclS != 1 || inclT != 1)
                    Bins2[s] = null;
                if (inclT != 1 || inclU != 1)
                    Bins3[s] = null;
            }
            int max = 0;
            counts1.Clear(); counts2.Clear(); counts3.Clear();
            Node site1 = Bins1[bound]; Node site2 = Bins2[bound]; Node site3 = Bins3[bound];
            if (inclS != 1 || inclU != 1) UpdateCounts3(Bins1, counts1, bound, true, timeS, timeT, timeU, 1, ref max);
            else UpdateCounts3(Bins1, counts1, bound, false, timeS, timeT, timeU, 1, ref max);
            if (inclS != 1 || inclT != 1) UpdateCounts3(Bins2, counts2, bound, true, timeS, timeT, timeU, 2, ref max);
            else UpdateCounts3(Bins2, counts2, bound, false, timeS, timeT, timeU, 2, ref max);
            if (inclT != 1 || inclU != 1) UpdateCounts3(Bins3, counts3, bound, true, timeS, timeT, timeU, 3, ref max);
            else UpdateCounts3(Bins3, counts3, bound, false, timeS, timeT, timeU, 3, ref max);

            if ((inclS != 1 && (inclT != 1 || inclU != 1)) || (inclT != 1 && inclU != 1))
                result = CorrectThirdTerms(T1, T2, T3, max, smax, tmax, umax, Bins1, Bins2, Bins3, ref timeS, ref timeT, ref timeU, r, counts1, counts2, counts3);
            else
                result = CorrectThirdTerms(T1, T2, T3, bound - 1, smax, tmax, umax, Bins1, Bins2, Bins3, ref timeS, ref timeT, ref timeU, r, counts1, counts2, counts3);
            RestoreCounts(counts1, site1);
            RestoreCounts(counts2, site2);
            RestoreCounts(counts3, site3);

            return result;
        }

        //Initialises the bins
        //First max is the max reachable site, at the end it is the maximum non-empty bin
        //Term shows for which term we want to initialise the bins
        //Mode is only used when calculating the second terms to show which mode we are in
        static Node[] InitBins(List<Node> Tree, ref int max, int term, int mode)
        {
            Node[] bins = new Node[max + 1];
            max = 0;
            foreach (Node node in Tree)
                node.sibling = null;
            foreach (Node node in Tree)
            {
                if (node.count > 0)
                {
                    if (term == 1) InsertBin(node, bins, 0);
                    else if (term == 2) InsertBin2(node, bins, 0, 0, mode);
                    else if (term == 3) InsertBin3(node, bins, 0, 0, 0, mode);
                    if (node.site > max)
                        max = node.site;
                }
            }
            return bins;
        }

        static void InsertBin(Node node, Node[] bin, long stamp)
```

```
        {
            int s = node.site;
            if (bin[s] != null && bin[s].stamp1 != stamp)
                bin[s] = null;
            node.stamp1 = stamp;
            node.sibling = bin[s];
            bin[s] = node;
        }

        static void InsertBin2(Node node, Node[] bin, long timeS, long timeT, int mode)
        {
            int s = node.site;
            if (bin[s] != null && !CheckTime(timeS, timeT, mode, bin[s]))
                bin[s] = null;
            node.stamp1 = timeS;
            node.stamp2 = timeT;
            node.sibling = bin[s];
            bin[s] = node;
        }

        static void InsertBin3(Node node, Node[] bin, long timeS, long timeT, long timeU, int mode)
        {
            int s = node.site;
            if (bin[s] != null && !CheckTime2(timeS, timeT, timeU, mode, bin[s]))
                bin[s] = null;
            node.stamp1 = timeS;
            node.stamp2 = timeT;
            node.stamp3 = timeU;
            node.sibling = bin[s];
            bin[s] = node;
        }

        //The bool incl states whether or not bound is included in supersets
        static void UpdateCounts(Node[] bins, List<long> counts, int bound, bool incl, long time, ref int max)
        {
            Node v = bins[bound];
            Node pv;
            while (v != null)
            {
                pv = v.parent;
                counts.Add(pv.count);
                //We only want to add the count if we do not want to include bound in supersets
                if (pv.site != -1)
                {
                    if (!incl && pv.stamp1 == time)
                        pv.count += v.count;
                    else
                    {
                        pv.count = v.count;
                        if (pv.site > max)
                            max = pv.site;
                        InsertBin(pv, bins, time);
                        pv.stamp1 = time;
                    }
                }
                v = v.sibling;
            }
        }

        //Mode is 1 when we look at bins1, 2 when looking at bins2 and 3 when looking at bins3
        static void UpdateCounts2(Node[] bins, List<long> counts, int bound, bool incl, long timeS, long timeT, int mode, ref int max)
        {
            Node v = bins[bound];
            Node pv;
            while (v != null)
            {
                pv = v.parent;
                counts.Add(pv.count);
                //We only want to add the count if we do not want to include bound in supersets
                if (pv.site != -1)
                {
                    if (!incl && CheckTime(timeS, timeT, mode, pv))
                        pv.count += v.count;
                    else
                    {
                        pv.count = v.count;
                        if (pv.site > max)
                            max = pv.site;
                        InsertBin2(pv, bins, timeS, timeT, mode);
                        pv.stamp1 = timeS;
```

```
                        pv.stamp2 = timeT;
                    }
                }
                v = v.sibling;
            }
        }

        //Mode is 1 when we look at bins1, 2 when looking at bins2 and 3 when looking at bins3
        static void UpdateCounts3(Node[] bins, List<long> counts, int bound, bool incl, long timeS, long timeT, long timeU, int mode, ref int max)
        {
            Node v = bins[bound];
            Node pv;
            while (v != null)
            {
                pv = v.parent;
                counts.Add(pv.count);
                //We only want to add the count if we do not want to include bound in supersets
                if (pv.site != -1)
                {
                    if (!incl && CheckTime2(timeS, timeT, timeU, mode, pv))
                        pv.count += v.count;
                    else
                    {
                        pv.count = v.count;
                        if (pv.site > max)
                            max = pv.site;
                        InsertBin3(pv, bins, timeS, timeT, timeU, mode);
                        pv.stamp1 = timeS;
                        pv.stamp2 = timeT;
                        pv.stamp3 = timeU;
                    }
                }
                v = v.sibling;
            }
        }

        static bool CheckTime(long timeS, long timeT, int mode, Node v)
        {
            switch (mode)
            {
                case 1:
                    if (v.stamp1 == timeS && v.stamp2 == timeT) return true;
                    else return false;
                case 2:
                    if (v.stamp1 == timeS) return true;
                    else return false;
                case 3:
                    if (v.stamp2 == timeT) return true;
                    else return false;
            }
            return false;
        }

        static bool CheckTime2(long timeS, long timeT, long timeU, int mode, Node v)
        {
            switch (mode)
            {
                case 1:
                    if (v.stamp1 == timeS && v.stamp3 == timeU) return true;
                    else return false;
                case 2:
                    if (v.stamp1 == timeS && v.stamp2 == timeT) return true;
                    else return false;
                case 3:
                    if (v.stamp2 == timeT && v.stamp3 == timeU) return true;
                    else return false;
            }
            return false;
        }

        static void RestoreCounts(List<long> counts, Node v)
        {
            List<long>.Enumerator e = counts.GetEnumerator();
            while (v != null)
            {
                e.MoveNext();
                v.parent.count = e.Current;
                v = v.sibling;
            }
        }

```

```
        static long CalcCount(Node v)
        {
            long result = 0;
            while (v != null)
            {
                result += v.count;
                v = v.sibling;
            }
            return result;
        }
    }

    class Node
    {
        public int site; //site number of node
        public Int64 count; //number of saw's with this node as highest site number
        public Node child, sibling, parent; //first child, next sibling also used for next node with the same site number when traversing the tree, parent
        public Int64 stamp1, stamp2, stamp3; //time stamps

        public void newNode(int s, Int64 c, Node ch, Node si, Node pa)
        {
            site = s;
            count = c;
            child = ch;
            sibling = si;
            parent = pa;
            stamp1 = -1;
            stamp2 = -1;
            stamp3 = -1;
        }
    }
}
```

References

[1] Nathan Clisby. Calculation of the connective constant for self-avoiding walks via the pivot algorithm. *Journal of Physics A: Mathematical and Theoretical*, 46(24):245001, 2013.

[2] Nathan Clisby, Richard Liang, and Gordon Slade. Self-avoiding walk enumeration via the lace expansion. *Journal of Physics A: Mathematical and Theoretical*, 40(36):10973, 2007.

[3] Hugo Duminil-Copin and Stanislav Smirnov. The connective constant of the honeycomb lattice equals $\sqrt{2+\sqrt{2}}$. *Annals of Mathematics*, 175:1653–1665, 2012.

[4] Michael E Fisher and MF Sykes. Excluded-volume problem and the ising model of ferromagnetism. *Physical Review*, 114(1):45, 1959.

[5] AJ Guttmann. On the critical behaviour of self-avoiding walks. ii. *Journal of Physics A: Mathematical and General*, 22(14):2807, 1989.

[6] AJ Guttmann and AR Conway. Square lattice self-avoiding walks and polygons. *Annals of Combinatorics*, 5(3-4):319–345, 2001.

[7] Iwan Jensen. A parallel algorithm for the enumeration of self-avoiding polygons on the square lattice. *Journal of Physics A: Mathematical and General*, 36(21):5731, 2003.

[8] D MacDonald, S Joseph, DL Hunter, LL Moseley, N Jan, and AJ Guttmann. Self-avoiding walks on the simple cubic lattice. *Journal of Physics A: Mathematical and General*, 33(34):5973, 2000.

[9] Neal Madras and Gordon Slade. *The self-avoiding walk*. Springer Science & Business Media, 2013.

[10] WJC Orr. Statistical treatment of polymer solutions at infinite dilution. *Transactions of the Faraday Society*, 43:12–27, 1947.

[11] Fred Roberts and Barry Tesman. *Applied combinatorics*. CRC Press, 2009.

[12] R D Schram, G T Barkema, and R H Bisseling. Exact enumeration of self-avoiding walks. *Journal of Statistical Mechanics: Theory and Experiment*, 2011(06):P06019, 2011.

[13] Raoul D Schram, Gerard T Barkema, and Rob H Bisseling. Sawdoubler: A program for counting self-avoiding walks. *Computer Physics Communications*, 184(3):891–898, 2013.

[14] Raoul D Schram, Gerard T Barkema, Rob H Bisseling, and Nathan Clisby. Exact enumeration of self-avoiding walks on bcc and fcc lattices. *arXiv preprint arXiv:1703.09340*, 2017.

[15] MF Sykes, AJ Guttmann, MG Watts, and PD Roberts. The asymptotic behaviour of selfavoiding walks and returns on a lattice. *Journal of Physics A: General Physics*, 5(5):653, 1972.