

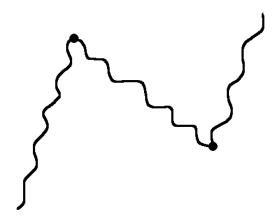
Faculteit Bètawetenschappen

# Enumeration Of Self-Avoiding Walks Using Length Tripling

BACHELOR THESIS

Sarita de Berg

Mathematics



Supervisor:

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

#### 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	1							
2	The length-tripling method         2.1       Counting combinations         2.2       Calculating the first corrections         2.3       Calculating the second corrections         2.4       Calculating the third corrections	<b>2</b> 2 3 4 4							
3	Algorithms and implementation         3.1       Creating self-avoiding walks         3.2       The first corrections         3.3       The second and third corrections	<b>5</b> 5 6 8							
4	Complexity and memory use	9							
5	A method using $k$ walks	10							
6	Results								
7	Conclusion								
A	Implementation of the length-tripling method	13							
Re	eferences								

# **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

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

Here,  $\gamma$  is a universal exponent which only depends on the dimension,  $\mu$  is a connective constant which depends on the lattice and A is a critical amplitude. For most lattices we only have approximations for  $\mu$ , 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.

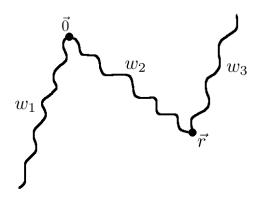


Figure 1: Construction of a walk of length N

# 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

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

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

$$|D| = Z_1 Z_2 Z_3 - |A| - |B| - |C| + |A \cap B| + |B \cap C| + |A \cap C| - |A \cap B \cap C|.$$

$$(2)$$

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  $w_3$ .

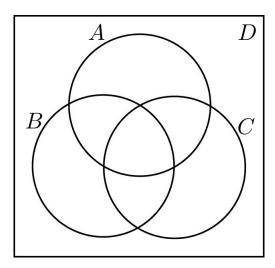


Figure 2: Venn diagram of combinations  $(w_1, w_2, w_3)$ 

#### 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

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

$$\left| \bigcup_{i=1}^{n} A_{i} \right| = \sum_{i} |A_{i}| - \sum_{i < j} |A_{i} \cap A_{j}| + \sum_{i < j < k} |A_{i} \cap A_{j} \cap A_{k}| + \dots + (-1)^{n+1} |A_{1} \cap A_{2} \cap \dots \cap A_{n}|.$$
(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 Tand  $w_3$  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 Tand add or subtract the walks visiting these sets. We get the equation

$$|A \cap B| = \sum_{\substack{S \times T \\ S \neq \emptyset, T \neq \emptyset}} (-1)^{|S| + |T|} Z_1(S) Z_2(S \cup T) Z_3(T).$$
(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| = 2and |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

```
 \begin{array}{l} \textbf{function FILLTREE}(N, i, R, visited, \mathcal{T}, end) \\ \textbf{if } i = N \textbf{ then} \\ \textbf{if } end = -1 \textbf{ or } R[i] = end \textbf{ then} \\ & \text{Sort}(R) \\ & \text{InsertTree}(R, \mathcal{T}) \\ \textbf{else} \\ \textbf{ for all } r \in Adj(R[i]) \textbf{ do} \\ & \textbf{ if not } visited[r] \textbf{ then} \\ & R[i+1] \leftarrow r \\ & visited[r] \leftarrow \text{ true} \\ & \text{FILLTREE}(N, i+1, R, visited, \mathcal{T}, end) \\ & visited[R[i]] \leftarrow false \end{array} \right) \\ \end{matrix}
```

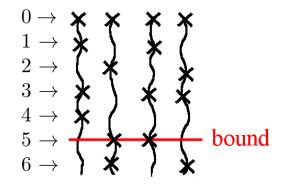


Figure 3: The bound goes up in the tree, during which we choose whether or not to add bound to the set S.

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, 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 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 Sand 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 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	at calculates the first cor	rection terms
function CorrectFirstTerms( $\mathcal{T}_1$	$\mathcal{T}_2, Bins1, Bins2, mode$	r, r, time)
$Z \leftarrow 0$		
$bound \leftarrow \max[i:Bins1[i] \neq \emptyset $ or		$\triangleright$ find max active site
$ \begin{array}{l} \mathbf{if} \ bound = -1 \ \mathbf{or} \ (bound = 0 \ \mathbf{ar} \\ \mathbf{return} \ \ Z \end{array} $	$\mathbf{nd} \ mode = A) \ \mathbf{then}$	$\triangleright$ we cannot include zero if <i>mode</i> is 1
if $bound \neq 0$ then	$\triangleright$ if $bound = 0$ we c	an only include $bound$ in $S$ but no more sites
$\triangleright \text{ Contribution for } S' \supseteq S \text{ with } UPDATECOUNTS}(Bins1, bound UPDATECOUNTS}(Bins2, bound Z \leftarrow Z + CORRECTFIRSTER)$	nd, false, time) nd, false, time)	$\triangleright$ false because we do not include <i>bound</i> ns2. mode. r. time)
Restore the counts	· · · ( · 1 ) · 2 ) ) · · · · · ) · ·	······································
$\triangleright \text{ Contribution for } S' \supseteq S \text{ wit}$ if $bound \neq r \text{ or } mode = A \text{ or}$ $time \leftarrow time + 1$		
for $s = 0$ to $bound - 1$ d $Bins1[s] = \emptyset$ $Bins2[s] = \emptyset$	o	$\triangleright$ empty the bins
UPDATECOUNTS $(Bins1, b)$ UPDATECOUNTS $(Bins2, b)$ $Z \leftarrow Z - \text{CORRECTFIRST}$ Restore the counts	ound, true, time)	$\triangleright \mbox{ true because we include } bound \\Bins2, mode, r, time)$
$\triangleright \text{ Contribution for } S' = S \cup \{bound \neq r \text{ or } mode = A \text{ or } matcheve \}$		
$Z1 \leftarrow 0$		$\triangleright$ total walks of type 1
$Z2 \leftarrow 0$		$\triangleright$ total walks of type 2
for all $v \in Bins1[bound]$ do		
$Z1 \leftarrow Z1 + v.count$		
for all $w \in Bins2[bound]$ do		
$Z2 \leftarrow Z2 + w.count$		
$Z \leftarrow Z + Z1 \cdot Z2$		
return $Z$		

Algorithm 3	<b>B</b> Algorithm to	change the	counts in the	tree to match	the number of v	valks visiting the set
-------------	-----------------------	------------	---------------	---------------	-----------------	------------------------

function UPDATECOUNTS(*Bins*, *bound*, *incl*, *time*)  $\triangleright$  incl is whether or not we include bound in the set for  $v \in Bins[bound]$  do

 $\begin{array}{l} pv \leftarrow v.parent \\ \textbf{if not } incl \ \textbf{and} \ pv.stamp = time \ \textbf{then} \\ pv.count \leftarrow pv.count + v.count \\ \textbf{else} \\ pv.count \leftarrow v.count \\ \text{INSERTBIN}(pv, Bins, time) \\ pv.stamp \leftarrow time \end{array}$ 

 $\triangleright$  parent is active

#### 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  $\mu^N$  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  $2^d N^d$ . 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  $O(2^N 2^N \mu^N) = O(4^N \mu^N)$ . Finally, we look at the third corrections. In this case we have three subsets we can combine, so we get  $O(2^N 2^N \mu^N) = O(8^N \mu^N)$ . All together this means we have a complexity of  $O(2^d N^d 8^N \mu^N)$ . When d is small of course  $2^d N^d$  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 dimensions 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

$$\mu^{kN} > 2^{\binom{k}{2} \cdot N} \cdot \mu^{N} \\ \mu > \sqrt[k-1]{2^{\binom{k}{2}}}.$$
(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  $\mu$ . 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  $\mu$ . 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
            using System;
using System.Collections.Generic;
using System.Ling;
using System.Ling;
using System.Ling;
using System.lext;
using System.Threading.Tasks;
              namespace SAW
             {
    10
                       class saw
    11
12
                        {
                                 static public int N;
    13
14
                                 static void Main()
    15
16
17
                                          Stopwatch timer = new Stopwatch();
                                          Stopwatch timer = new Stopwatch
timer.Start();
int N1, N2, N3, lattice, zero;
N1 = 2;
N2 = 2;
N3 = 2;
lattice = 0;
zero = 0;
   18
19
20
21
22
23
24
25
26
27
28
29
30
31
                                          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(time.Elapsed);
Console.ReadLine();
    32
33
34
35
36
37
38
39
40
                              }
                                 static List<int>[] CreateGraph(int lattice, ref int zero)
                                          List<int>[] graph;
switch (lattice)
{
                                                   case 0:
    41
42
43
44
45
46
47
48
49
50
51
52
53
53
                                                           graph = CreateSquare();
zero = phiSq(N, N);
break;
                                                     case 1:
                                                    graph = CreateHoneycomb();
zero = phiHc(N, N / 2 + 1);
break;
case 2:
                                                    case 2:
    graph = CreateCubic();
    zero = phiCu(N, N, N);
    break;
default:
                                                              graph = new List<int>[0];
break;
    55
56
57
58
59
60
61
62
                                          }
                                          return graph;
                               }
                                 static List<int>[] CreateSquare()
                                          Listcint>[] AdjacencyList = new Listcint>[phiSq(2 * N, 2 * N) + 1];
for (int k = 0; k < phiSq(2 * N, 2 * N) + 1; k++)
AdjacencyList[k] = new Listcint>();
//First we look at the AdjacencyList for the edges of our grid
for (int i = 1; i < 2 * N; i++)</pre>
   63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
                                            {
                                                     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++)</pre>
                                                     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));
    82
```

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

```
AdjacencyList[phiSq(2 * N, j)].Add(phiSq(2 * N, j - 1));
 83
                            AdjacencyList[phiSq(2 * N, j)].Add(phiSq(2 * N, j + 1));
 84
                            AdjacencyList[phiSq(2 * N, j)].Add(phiSq(2 * N - 1, j));
 85
 86
                      }
 87
                      //Now we look at all the corners
 88
                      AdjacencyList[phiSq(0, 0)].Add(phiSq(1, 0));
 89
                      AdjacencyList[phiSq(0, 0)].Add(phiSq(0, 1));
AdjacencyList[phiSq(2 * N, 0)].Add(phiSq(2 * N - 1, 0));
 90
 91
                      AdjacencyList[phiSq(2 * N, 0)].Add(phiSq(2 * N, 1));
AdjacencyList[phiSq(0, 2 * N)].Add(phiSq(0, 2 * N - 1));
 92
 93
                      AdjacencyList[phiSq(0, 2 * N)].Add(phiSq(1, 2 * N));
AdjacencyList[phiSq(2 * N, 2 * N)].Add(phiSq(2 * N - 1, 2 * N));
 94
 95
 96
                      AdjacencyList[phiSq(2 * N, 2 * N)].Add(phiSq(2 * N, 2 * N - 1));
 97
 98
                       //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++)</pre>
 99
100
101
                            {
102
                                 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));
103
104
105
                                 AdjacencyList[phiSq(i, j)].Add(phiSq(i, j + 1));
                            }
106
107
108
                      return AdjacencyList;
109
                 }
110
111
                 //Assigns a canonical numbering to every point from (0,0) to (2N, 2N) in the square lattice
112
                 static int phiSq(int i, int j)
113
                 {
114
                      return i * (2 * N + 1) + j;
115
                 }
116
117
                 static List<int>[] CreateHoneycomb()
118
                 {
                      List<int>[] AdjacencyList = new List<int>[phiHc(2 * N, N + 2) + 1];
for (int k = 0; k < phiHc(2 * N, 2 + N) + 1; k++)
119
120
                           AdjacencyList[k] = new List<int>();
121
                       //First we look at the Adjacencylist for the edges of our grid
122
                      for (int i = 1; i < 2 * N; i++)</pre>
123
                      {
124
125
                            AdjacencyList[phiHc(i, 0)].Add(phiHc(i - 1, 0));
                            AdjacencyList[phiHc(i, 0)].Add(phiHc(i + 1, 0));
126
127
128
                            AdjacencyList[phiHc(i, 2 + N)].Add(phiHc(i - 1, 2 + N));
                            AdjacencyList[phiHc(i, 2 + N)].Add(phiHc(i + 1, 2 + N));
129
130
                            if (i % 2 == 1)
131
132
                                 AdjacencyList[phiHc(i, 0)].Add(phiHc(i, 1));
133
                            else
134
                                 AdjacencyList[phiHc(i, 2 + N)].Add(phiHc(i, N + 1));
135
                       for (int j = 1; j < 2 + N; j++)</pre>
136
137
                      {
                           AdjacencyList[phiHc(0, j)].Add(phiHc(1, j));
AdjacencyList[phiHc(2 * N, j)].Add(phiHc(2 * N - 1, j));
138
139
140
                            if (j % 2 == 1)
141
142
                            {
                                 AdjacencyList[phiHc(0, j)].Add(phiHc(0, j + 1));
AdjacencyList[phiHc(2 * N, j)].Add(phiHc(2 * N, j + 1));
143
144
145
                            }
146
                            else
147
                            {
                                 AdjacencyList[phiHc(0, j)].Add(phiHc(0, j - 1));
AdjacencyList[phiHc(2 * N, j)].Add(phiHc(2 * N, j - 1));
148
149
150
                            }
151
                      }
152
                       //Now we look at all the corners
153
                      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));
154
155
156
157
                      if (N % 2 == 0)
158
159
                      {
                            AdjacencyList[phiHc(0, 2 + N)].Add(phiHc(0, 1 + N));
AdjacencyList[phiHc(2 * N, 2 + N)].Add(phiHc(2 * N, N + 1));
160
161
162
                      }
163
                      //Finally we look at the middle of the grid
164
```

C:\Users\Sarita de Berg\Documents\Scriptie\SAW9\SAW3\SAW\SAW\saw.cs 3 for (int i = 1; i < 2 \* N; i++)</pre> 165 166 for (int j = 1; j < 2 + N; j++)</pre> 167 { 168 AdjacencyList[phiHc(i, j)].Add(phiHc(i - 1, j)); 169 AdjacencyList[phiHc(i, j)].Add(phiHc(i + 1, j)); 170 if ((i + j) % 2 == 0)AdjacencyList[phiHc(i, j)].Add(phiHc(i, j - 1)); 171 else 172 AdjacencyList[phiHc(i, j)].Add(phiHc(i, j + 1)); 173 174 } 175 return AdjacencyList; 176 177 } 178 //Assigns a canonical numbering to every point from (0,0) to (2N, 1/2 N + 1) in the honeycom b 179 lattice 180 static int phiHc(int i, int j) 181 { 182 return i \* (N + 3) + j;183 } 184 185 static List<int>[] CreateCubic() 186 { List<int>[] AdjacencyList = new List<int>[phiCu(2 \* N, 2 \* N, 2 \* N) + 1]; 187 for (int k = 0; k < phiCu(2 \* N, 2 \* N, 2 \* N) + 1; k++)</pre> 188 AdjacencyList[k] = new List<int>(); 189 190 191 //First we look at the Adjacencylist for the edges of our grid 192 for (int i = 1; i < 2 \* N; i++)</pre> 193 { AdjacencyList[phiCu(i, 0, 0)].Add(phiCu(i - 1, 0, 0)); 194 AdjacencyList[phiCu(i, 0, 0)].Add(phiCu(i + 1, 0, 0)); 195 196 AdjacencyList[phiCu(i, 0, 0)].Add(phiCu(i, 1, 0)); 197 AdjacencyList[phiCu(i, 0, 0)].Add(phiCu(i, 0, 1)); 198 199 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 + 1, 2 \* N, 0)); AdjacencyList[phiCu(i, 2 \* N, 0)].Add(phiCu(i, 2 \* N - 1, 0)); 200 201 AdjacencyList[phiCu(i, 2 \* N, 0)].Add(phiCu(i, 2 \* N, 1)); 202 203 AdjacencyList[phiCu(i, 0, 2 \* N)].Add(phiCu(i - 1, 0, 2 \* N)); 204 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)); 205 206 207 208 209 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)); 210 211 AdjacencyList[phiCu(i, 2 \* N, 2 \* N)].Add(phiCu(i, 2 \* N, 2 \* N - 1)); 212 213 214 for (int j = 1; j < 2 \* N; j++)</pre> 215 { AdjacencyList[phiCu(0, j, 0)].Add(phiCu(0, j - 1, 0)); AdjacencyList[phiCu(0, j, 0)].Add(phiCu(0, j + 1, 0)); 216 217 AdjacencyList[phiCu(0, j, 0)].Add(phiCu(1, j, 0)); 218 219 AdjacencyList[phiCu(0, j, 0)].Add(phiCu(0, j, 1)); 220 AdjacencyList[phiCu(2 \* N, j, 0)].Add(phiCu(2 \* N, j - 1, 0)); 221 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)); 222 223 AdjacencyList[phiCu(2 \* N, j, 0)].Add(phiCu(2 \* N, j, 1)); 224 225 226 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)); 227 228 229 AdjacencyList[phiCu(0, j, 2 \* N)].Add(phiCu(0, j, 2 \* N - 1)); 230 231 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)); 232 AdjacencyList[phiCu(2 \* N, j, 2 \* N)].Adu(phiCu(2 \* N - 1, j, 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)); 233 234 235 for (int k = 1; k < 2 \* N; k++) 236 237 { AdjacencyList[phiCu(0, 0, k)].Add(phiCu(0, 0, k - 1)); 238 AdjacencyList[phiCu(0, 0, k)].Add(phiCu(0, 0, k + 1)); AdjacencyList[phiCu(0, 0, k)].Add(phiCu(1, 0, k)); 239 240 241 AdjacencyList[phiCu(0, 0, k)].Add(phiCu(0, 1, k)); 242 243 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)); 244 AdjacencyList[phiCu(2 \* N, 0, k)].Add(phiCu(2 \* N - 1, 0, k)); 245

<pre>AdjacemcyList[phiCu(0, 2 + N, b)].Add(phiCu(0, 2 + N, k + 1)); AdjacemcyList[phiCu(0, 2 + N, c)].Add(phiCu(0, 2 + N, k + 1)); AdjacemcyList[phiCu(0, N, 2 + N, b)].Add(phiCu(2, 2 + N, k + 1)); AdjacemcyList[phiCu(1 + N, 2 + N, k)].Add(phiCu(2 + N, 2 + N, k + 1)); AdjacemcyList[phiCu(1 + N, 2 + N, k)].Add(phiCu(2 + N, 2 + N, k + 1)); AdjacemcyList[phiCu(1 + N, 2 + N, k)].Add(phiCu(2 + N, 2 + N, k + 1)); AdjacemcyList[phiCu(0 + N, 2 + N, k)].Add(phiCu(2 + N, 2 + N + 1, k)); AdjacemcyList[phiCu(0 + 0, 0]].Add(phiCu(1 + N + 1, 0)); AdjacemcyList[phiCu(0 + 0, 0]].Add(phiCu(0 + 1, 0)); AdjacemcyList[phiCu(0 + 0, 0]].Add(phiCu(0 + 1, 0)); AdjacemcyList[phiCu(0 + 0, 0]].Add(phiCu(1 + N + 1, 0)); AdjacemcyList[phiCu(0 + 0, 0]].Add(phiCu(0 + 2 + N + 1, 0)); AdjacemcyList[phiCu(0 + 0, 0]].Add(phiCu(0 + 0, 0)); AdjacemcyList[phiCu(0 + 0, 0)].Add(phiCu(0 + 0)].000; AdjacemcyList[phiCu(0 + 0, 0)].Add(phiCu(0 + 0)].</pre>	\Users\Sari 46	<pre>ta de Berg\Documents\Scriptie\SAW9\SAW3\SAW\SAW\saw.cs</pre>
<pre>AdjacencyList[phiCu(0, 2 * m, 0]).Add(phiCu(0, 2 * m, k-1)); AdjacencyList[phiCu(0, 2 * m, 2)].Add(phiCu(2, 2 * m, 2 * m, k - 1)); AdjacencyList[phiCu(2 * n, 2 * m, k)].Add(phiCu(2 * m, 2 * m, k + 1)); AdjacencyList[phiCu(2 * n, 2 * m, k)].Add(phiCu(2 * m, 2 * m, k + 1)); AdjacencyList[phiCu(2 * n, 2 * m, k)].Add(phiCu(2 * m, 2 * m, k + 1)); AdjacencyList[phiCu(2 * n, 2 * m, k)].Add(phiCu(2 * m, 2 * m, k + 1)); AdjacencyList[phiCu(2 * n, 2 * m, k)].Add(phiCu(2 * m, 2 * m, k + 1)); AdjacencyList[phiCu(2 * n, 2 * m, k)].Add(phiCu(2 * m, 2 * m, k + 1)); AdjacencyList[phiCu(2 * n, 2 * m, k)].Add(phiCu(2 * m, 2 * m, k + 1)); AdjacencyList[phiCu(2 * m, 2 * m, k)].Add(phiCu(2 * m, 1 * m, 0)); AdjacencyList[phiCu(2 * m, 0 * 0]].Add(phiCu(2 * m, 1 * m, 0)); AdjacencyList[phiCu(2 * m, 0 * 0]].Add(phiCu(2 * m, 1 * m, 0)); AdjacencyList[phiCu(2 * m, 0 * 0]].Add(phiCu(2 * m, 1 * m, 0)); AdjacencyList[phiCu(0 * 0 * m]].Add(phiCu(0 * 1 * m, 1)); AdjacencyList[phiCu(0 * m, 0 * m]].Add(phiCu(0 * 1 * m, 1 * m, 0)); AdjacencyList[phiCu(0 * 0 * m]].Add(phiCu(0 * 1 * m, 1 * m, 0)); AdjacencyList[phiCu(0 * 0 * m]].Add(phiCu(0 * m, 2 * m, 1)); AdjacencyList[phiCu(0 * 0 * m]].Add(phiCu(0 * m, 2 * m, 1)); AdjacencyList[phiCu(0 * m, 2 * m]].Add(phiCu(0 * m, 2 * m, 1)); AdjacencyList[phiCu(0 * m, 2 * m]].Add(phiCu(0 * m, 2 * m, 1)); AdjacencyList[phiCu(0 * m, 2 * m, 0]].Add(phiCu(0 * m, 2 * m, 1 * m)); AdjacencyList[phiCu(0 * m, 2 * m, 2 * m]].Add(phiCu(0 * m, 2 * m, 1 * m)); AdjacencyList[phiCu(0 * m, 2 * m, 2 * m]].Add(phiCu(1 * m, 1 * m, 2 * m); AdjacencyList[phiCu(0 * m, 2 * m, 2 * m)].Add(phiCu(1 * m, 1 * m, 2 * m); AdjacencyList[phiCu(0 * m, 2 * m, 2 * m)].Add(phiCu(1 * m, 1 * m, 2 * m); AdjacencyList[phiCu(0 * m, 2 * m, 2 * m)].Add(phiCu(1 * m, 1 * m, 2 * m); AdjacencyList[phiCu(0 * m, 2 * m, 2 * m)].Add(phiCu(1 * m, 1 * m, 2 * m, 2 * m); AdjacencyList[phiCu(2 * m, 2 * m, 2 * m)].Add(phiCu(1 * m, 1 * m, 2 * m, 2 * m); AdjacencyList[phiCu(2 * m, 2 * m, 2 * m)].Add(phiCu(1 * m, 1 * m, 2 * m, 2 * m); AdjacencyList[phiCu(1 * m, 2</pre>		
<pre>AdjscencyList[phiCu(0, 2 * N, b].Add(phiCu(1, 2 * N, b); AdjscencyList[phiCu(2 * N, 2 * N, b].Add(phiCu(2 * N, 2 * N, k + 1)); AdjscencyList[phiCu(2 * N, 2 * N, k].Add(phiCu(2 * N, 2 * N, k + 1)); AdjscencyList[phiCu(2 * N, 2 * N, k].Add(phiCu(2 * N, 2 * N, k + 1)); AdjscencyList[phiCu(2 * N, 2 * N, k].Add(phiCu(2 * N, 2 * N, k + 1)); AdjscencyList[phiCu(2 * N, 2 * N, k].Add(phiCu(2 * N, 2 * N, k + 1)); AdjscencyList[phiCu(2 * N, 2 * N, k].Add(phiCu(2 * N, 2 * N, k + 1)); AdjscencyList[phiCu(2 * N, 0 )].Add(phiCu(0, 1, 0)); AdjscencyList[phiCu(2 * N, 0 )].Add(phiCu(0, 1, 0)); AdjscencyList[phiCu(2 * N, 0 , 0)].Add(phiCu(2 * N, 1 , 0, 0)); AdjscencyList[phiCu(2 * N, 0 , 0)].Add(phiCu(2 * N, 1 , 0)); AdjscencyList[phiCu(2 * N, 0 , 0)].Add(phiCu(0, 2 * N, 1 , 0)); AdjscencyList[phiCu(0, 0 , 0 , 0)].Add(phiCu(0, 2 * N, 1 , 0)); AdjscencyList[phiCu(0, 2 * N, 0)].Add(phiCu(0, 2 * N, 1 , 0)); AdjscencyList[phiCu(0, 2 * N)].Add(phiCu(0, 1 , 0 * N , 1 , 0)); AdjscencyList[phiCu(0, 2 * N)].Add(phiCu(0, 1 , 0 * N , 1 , 0)); AdjscencyList[phiCu(0, 2 * N)].Add(phiCu(0, 1 , 0 * N , 1 , 0)); AdjscencyList[phiCu(0, 0 * 2 * N)].Add(phiCu(0, 1 , 0 * N , 1 , 0)); AdjscencyList[phiCu(0 * 0 * 2 * N)].Add(phiCu(2 * N - 1 , 2 * N , 0)); AdjscencyList[phiCu(0 * 0 * 2 * N)].Add(phiCu(2 * N - 1 , 2 * N , 0)); AdjscencyList[phiCu(2 * N , 2 * N , 0)].Add(phiCu(2 * N - 1 , 2 * N , 0)); AdjscencyList[phiCu(2 * N , 2 * N , 0)].Add(phiCu(2 * N - 1 , 2 * N )); AdjscencyList[phiCu(2 * N , 0 * 2 * N)].Add(phiCu(2 * N - 1 , 0 * N )); AdjscencyList[phiCu(2 * N , 0 * 2 * N)].Add(phiCu(2 * N - 1 , 0 * N )); AdjscencyList[phiCu(2 * N , 0 * N )].Add(phiCu(2 * N - 1 , 0 * N )); AdjscencyList[phiCu(2 * N , 0 * N )].Add(phiCu(2 * N - 1 , 0 * N )); AdjscencyList[phiCu(2 * N , 0 * N )].Add(phiCu(2 * N - 1 , 0 * N )); AdjscencyList[phiCu(2 * N , 0 * N )].Add(phiCu(2 * N , 0 * N - 1 , 0); AdjscencyList[phiCu(2 * N , 0 * N )].Add(phiCu(2 * N , 0 * N - 1 , 0); AdjscencyList[phiCu(2 * N , 0 * N )].Add(phiCu(2 * N , 0 * N - 1 , 0 * N )); AdjscencyList[phiCu</pre>		
<pre>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, 2 * N, k - 1), k)); adjacencyList[phiCu(2 * N, 2 * N, k)].Add(phiCu(2 * N, 2 * N - 1, k)); AdjacencyList[phiCu(2 * N, 2 * N, k)].Add(phiCu(2 * N, 2 * N - 1, k)); AdjacencyList[phiCu(2 * N, 0 0].Add(phiCu(0, 1, 0)); AdjacencyList[phiCu(2 * N, 0 0].Add(phiCu(0 * N - 1, 0, 0)); AdjacencyList[phiCu(2 * N, 0 0].Add(phiCu(2 * N, -1, 0, 0)); AdjacencyList[phiCu(2 * N, 0 0].Add(phiCu(0, 2 * N - 1, 0)); AdjacencyList[phiCu(2 * N, 0 0].Add(phiCu(0, 2 * N - 0)); AdjacencyList[phiCu(2 * N, 0 0].Add(phiCu(0, 2 * N - 1, 0)); AdjacencyList[phiCu(0, 2 * N, 0].Add(phiCu(0, 1, 2 * N)); AdjacencyList[phiCu(2 * N, 0 0 + N, 0)].Add(phiCu(0, 1, 2 * N)); AdjacencyList[phiCu(0, 2 * N, 0].Add(phiCu(0, 1, 2 * N)); AdjacencyList[phiCu(0, 2 * N, 0].Add(phiCu(0, 1, 2 * N)); AdjacencyList[phiCu(0, 2 * N, 0].Add(phiCu(0, 1, 2 * N, 0)); AdjacencyList[phiCu(0, 2 * N, 0 + N, 0].Add(phiCu(0 * N - 1, 0, 2 * N, 0)); AdjacencyList[phiCu(0, 2 * N, 0 + N, 0].Add(phiCu(0 * N, 2 * N - 1)); AdjacencyList[phiCu(0, 2 * N, 2 * N)].Add(phiCu(0 * N, 2 * N - 1)); AdjacencyList[phiCu(2 * N, 2 * N, 0].Add(phiCu(0 * N, 2 * N - 1)); AdjacencyList[phiCu(2 * N, 2 * N, 0].Add(phiCu(0 * N, 2 * N - 1)); AdjacencyList[phiCu(2 * N, 2 * N, 0].Add(phiCu(0 * N, 2 * N - 1)); AdjacencyList[phiCu(2 * N, 2 * N)].Add(phiCu(0 * N, 0 * N - 1)); AdjacencyList[phiCu(1 * N, 2 * N, 2 * N)].Add(phiCu(0 * N, 0 * N - 1)); AdjacencyList[phiCu(1 * N, 0 * 2 * N)].Add(phiCu(0 * N, 0 * N - 1)); AdjacencyList[phiCu(1 * N, 0 * 2 * N)].Add(phiCu(1 * 1, 1, 2 * N); AdjacencyList[phiCu(1 * N, 0 * 2 * N)].Add(phiCu(1 * 1, 1, 2 * N); AdjacencyList[phiCu(1 * N, 2 * N, 2 * N)].Add(phiCu(1 * 1, 2 * N + 1)); AdjacencyList[phiCu(1 * N, 2 * N, 2 * N)].Add(phiCu(1 * 1, 2 * N, 2 * N)]; AdjacencyList[phiCu(1 * N, 2 * N, 2 * N)].Add(phiCu(1 * 1, 2 *</pre>		
<pre>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)); AdjacencyList[phiCu(2 * N, 2 * N, k)].Add(phiCu(2 * N, 2 * N, k)); AdjacencyList[phiCu(2 * N, 2 * N, k)].Add(phiCu(2 * N, 2 * N - 1, k)); 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 0)); 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 0)); 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 0)); 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 0)); AdjacencyList[phiCu(2 * N, 0 0]].Add(phiCu(0 0, 2 * N 0)); AdjacencyList[phiCu(2 * N, 0 0]].Add(phiCu(0 0, 2 * N 0)); AdjacencyList[phiCu(0 0, 2 * N, 0]].Add(phiCu(0 0, 2 * N 0)); AdjacencyList[phiCu(2 * N, 2 * N, 0]].Add(phiCu(0 0, 2 * N 0)); AdjacencyList[phiCu(2 * N, 2 * N, 0]].Add(phiCu(0 0, 2 * N 0)); AdjacencyList[phiCu(2 * N, 2 * N, 0]].Add(phiCu(0 0, 2 * N 0)); AdjacencyList[phiCu(2 * N, 2 * N, 0]].Add(phiCu(2 * N - 1, 2 * N)); AdjacencyList[phiCu(2 * N, 2 * N, 0]].Add(phiCu(2 * N - 1, 2 * N)); AdjacencyList[phiCu(2 * N, 2 * N, 0]].Add(phiCu(2 * N - 1, 2 * N)); AdjacencyList[phiCu(2 * N, 2 * N, 0]].Add(phiCu(2 * N - 1, 2 * N)); AdjacencyList[phiCu(2 * N, 0 * N)].Add(phiCu(2 * N - 1, 0 * N)); AdjacencyList[phiCu(2 * N, 0 * N)].Add(phiCu(2 * N - 1, 2 * N, 2 * N)); AdjacencyList[phiCu(2 * N, 0 * N)].Add(phiCu(2 * N - 1, 0 * N, 2 * N)); AdjacencyList[phiCu(2 * N, 0 * N)].Add(phiCu(2 * N - 1, 2 * N, 2 * N)); AdjacencyList[phiCu(2 * N, 0 * N)].Add(phiCu(2 * N, 0 * N - 1, 2 * N)); AdjacencyList[phiCu(2 * N, 0 * N)].Add(phiCu(2 * N, 0 * N - 1, 2 * N)); AdjacencyList[phiCu(1 * 1, 1 * 1, 1 * N)]; AdjacencyList[phiCu(1 * 1, 1 * 1, 1 * N * 1 * N)]; AdjacencyList[phiCu(1 * 1, 1 * N * 1 * N * 1 * N)]; AdjacencyList[phiCu(1 * 1, 1 * N * 1 * N * 1 * N * 1 * N * N *</pre>		
<pre>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 + 1, k)); AdjacencyList[phiCu(2 * N, 2 * N, k]].Add(phiCu(2 * N, 2 * N - 1, k)); 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 + 0)); 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 + 0)); 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 - 1, 0)); AdjacencyList[phiCu(2 * N, 0 + 0)].Add(phiCu(2 * N - 1, 0)); AdjacencyList[phiCu(2 * N, 0 + N)].Add(phiCu(0 + 2 * N - 1, 0)); AdjacencyList[phiCu(2 * N, 0 + N)].Add(phiCu(0 + 2 * N - 1, 0)); AdjacencyList[phiCu(2 * N, 0 + N)].Add(phiCu(0 + 1 + 1, 2 * N, 0)); AdjacencyList[phiCu(2 * N, 2 * N, 0)].Add(phiCu(0 + 1 + 1, 2 * N, 0)); 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 - 1, 2 * N, 0)); 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 - 1, 2 * N, 1)); AdjacencyList[phiCu(2 * N, 2 * N, 0)].Add(phiCu(2 * N - 1, 2 * N, 1)); AdjacencyList[phiCu(2 * N, 2 * N, 0)].Add(phiCu(2 * N - 1, 2 * N)); AdjacencyList[phiCu(2 * N, 0 * N)].Add(phiCu(2 * N - 1, 2 * N)); AdjacencyList[phiCu(2 * N, 0 * N)].Add(phiCu(2 * N - 1, 2 * N)); AdjacencyList[phiCu(2 * N, 0 * N)].Add(phiCu(2 * N - 1, 2 * N, 2 * N)]; AdjacencyList[phiCu(2 * N, 0 * N)].Add(phiCu(2 * N - 1, 2 * N)); AdjacencyList[phiCu(2 * N, 0 * N)].Add(phiCu(2 * N - 1, 2 * N)); AdjacencyList[phiCu(2 * N, 0 * N)].Add(phiCu(2 * N - 1, 2 * N); AdjacencyList[phiCu(2 * N, 0 * N)].Add(phiCu(2 * N - 1, 2 * N, 2 * N)); AdjacencyList[phiCu(2 * N, 0 * N)].Add(phiCu(2 * N - 1, 2 * N, 2 * N)]; AdjacencyList[phiCu(2 * N, 0 * N)].Add(phiCu(2 * N - 1, 2 * N, 2 * N)]; AdjacencyList[ph</pre>		
<pre>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)); AdjacencyList[phiCu(3 &amp; N, 0 = 0].Add(phiCu(2 * N - 1, 0 = 0)); AdjacencyList[phiCu(3 &amp; N, 0 = 0].Add(phiCu(2 * N - 1, 0 = 0)); AdjacencyList[phiCu(3 * N, 0 = 0].Add(phiCu(2 * N - 1, 0 = 0)); AdjacencyList[phiCu(3 * N, 0 = 0].Add(phiCu(2 * N, 1 = 0)); AdjacencyList[phiCu(3 * N, 0 = 0].Add(phiCu(2 * N, 1 = 0)); AdjacencyList[phiCu(3 * N, 0 = 0)].Add(phiCu(2 * N, 0 = 1)); AdjacencyList[phiCu(3 * N, 0 = 0)].Add(phiCu(2 * N, 0 = 1)); AdjacencyList[phiCu(0 = 2 * N, 0].Add(phiCu(1 = 2 * N = 1)); AdjacencyList[phiCu(0 = 2 * N, 0].Add(phiCu(0 = 0 = 2 * N = 1)); AdjacencyList[phiCu(0 = 2 * N)].Add(phiCu(1 = 0 = 2 * N = 1)); AdjacencyList[phiCu(0 = 2 * N)].Add(phiCu(2 * N = 1, 2 * N, 0)); AdjacencyList[phiCu(0 = 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, 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(2 * N = 2 * N = 1)); AdjacencyList[phiCu(2 * N, 2 * N, 0].Add(phiCu(2 * N = 1, 2 * N)); AdjacencyList[phiCu(0 = 2 * N, 2 * N)].Add(phiCu(2 * N = 1, 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 = 1, 2 * N)); AdjacencyList[phiCu(2 * N, 0 = 2 * N].Add(phiCu(2 * N = 1, 2 * N)); AdjacencyList[phiCu(2 * N, 2 * N, 2 * N].Add(phiCu(2 * N, 2 * N = 1)); AdjacencyList[phiCu(2 * N, 2 * N, 2 * N].Add(phiCu(2 * N, 2 * N = 1)); AdjacencyList[phiCu(2 * N, 2 * N, 2 * N].Add(phiCu(1 * 1, 1, 2 * N); AdjacencyList[phiCu(1 * N, 2 * N, 2 * N].Add(phiCu(1 * 1, 1, 2 * N); AdjacencyList[phiCu(1 * N, 2 * N, 2 * N].Add(phiCu(1 * 1, 1, 2 * N); AdjacencyList[phiCu(1 * N, 2 * N, 2 * N].Add(phiCu(1 * 1, 1, 2 * N); AdjacencyList[phiCu(1 * N, 2 * N, 2 * N].Add(phiCu(1 * 1, 1, 2 * N); AdjacencyList[phiCu(1 * N, 2</pre>		
<pre>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(0, 1, 0, 0)); AdjacencyList[phIcu(0, 0, 0)].Add(phIcu(0, 1, 0, 0)); 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, 0)); AdjacencyList[phIcu(2 * N, 0, 0)].Add(phIcu(2 * N, -1, 0)); AdjacencyList[phIcu(2 * N, 0, 0)].Add(phIcu(2 * N, -1, 0)); AdjacencyList[phIcu(2 * N, 0, 0)].Add(phIcu(2 * N, -1, 0)); AdjacencyList[phIcu(2 * N, 0)].Add(phIcu(2 * N, -1, 0)); AdjacencyList[phIcu(2 * N, 0)].Add(phIcu(2 * N, -1, 0)); AdjacencyList[phIcu(0, 2 * N, 0)].Add(phIcu(2 * N, -1, 0)); AdjacencyList[phIcu(0, 0, 2 * N)].Add(phIcu(2 * N, -1, 2 * N, 0)); AdjacencyList[phIcu(0, 2 * N, 2 * N, 0)].Add(phIcu(2 * N, -1, 2 * N, 0)); AdjacencyList[phIcu(0, 2 * N, 2 * N, 0)].Add(phIcu(2 * N, -1, 2 * N, 0)); AdjacencyList[phIcu(0, 2 * N, 2 * N, 0)].Add(phIcu(2 * N, -1, 2 * N, 0)); AdjacencyList[phIcu(0, 2 * N, 2 * N, 0)].Add(phIcu(2 * N, -1, 2 * N, 0)); AdjacencyList[phIcu(0, 2 * N, 2 * N, 0)].Add(phIcu(2 * N, -1, 2 * N)); AdjacencyList[phIcu(0, 2 * N, 2 * N, 0)].Add(phIcu(2 * N, -1, 2 * N)); AdjacencyList[phIcu(0, 2 * N, 2 * N)].Add(phIcu(2 * N, -1, 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, 1, 2 * N)); AdjacencyList[phIcu(2 * N, 0, 2 * N)].Add(phIcu(2 * N, 1, 2 * N)); AdjacencyList[phIcu(2 * N, 2 * N, 2 * N)].Add(phIcu(2 * N, 2 * N - 1)); AdjacencyList[phIcu(2 * N, 2 * N, 2 * N)].Add(phIcu(2 * N, 2 * N - 1, 2 * N, 2 * N)]; AdjacencyList[phIcu(2 * N, 2 * N, 2 * N)].Add(phIcu(2 * N, 2 * N - 1)); AdjacencyList[phIcu(2 * N, 2 * N, 2 * N)].Add(phIcu(2 * N, 2 * N - 1)); AdjacencyList[phIcu(2 * N, 2 * N, 2 * N)].Add(phIcu(2 * N, 2 * N - 1)); AdjacencyList[phIcu(2 * N, 2 * N, 2 * N)].Add(phIcu(2 * N, 2 * N - 1)); AdjacencyList[phIcu(2 * N, 2 * N, 2 * N)].Add(phIcu(2 * N, 2 * N - 1)); AdjacencyList[phIcu(2 * N, 2 * N, 2 * N)].Add(phIcu(2 * N,</pre>		
<pre>//Now we look at the corners AdjacencyList[phiCu(0, 0, 0]).Add(phiCu(1, 0, 0)); AdjacencyList[phiCu(2, 0, 0]).Add(phiCu(2, 0, 1, 0)); AdjacencyList[phiCu(2, 0, 0, 0]).Add(phiCu(2, 0, 1, 0)); AdjacencyList[phiCu(2, 0, 0, 0)].Add(phiCu(2, 0, 1, 0, 0)); AdjacencyList[phiCu(2, 0, 0, 0)].Add(phiCu(2, 0, 1, 0, 0)); AdjacencyList[phiCu(2, 0, 0, 0)].Add(phiCu(1, 0, 0, 0, 0)); AdjacencyList[phiCu(0, 0, 0, 0)].Add(phiCu(1, 0, 0, 0, 0)); AdjacencyList[phiCu(0, 0, 0, 0)].Add(phiCu(0, 1, 0, 0, 0)); AdjacencyList[phiCu(0, 0, 0, 2, 0)].Add(phiCu(0, 1, 0, 0, 0, 0)); AdjacencyList[phiCu(0, 0, 0, 2, 0, 0)].Add(phiCu(0, 1, 0, 0, 0, 0)); AdjacencyList[phiCu(0, 0, 0, 2, 0, 0)].Add(phiCu(0, 1, 0, 0, 0, 0)); AdjacencyList[phiCu(0, 0, 0, 2, 0, 0)].Add(phiCu(0, 1, 0, 1, 0, 0, 0)); AdjacencyList[phiCu(0, 0, 0, 2, 0)].Add(phiCu(1, 0, 1, 1, 0, 0, 0)); AdjacencyList[phiCu(0, 0, 0, 2, 0, 0)].Add(phiCu(2, 1, 1, 0, 2, 1, 0)); AdjacencyList[phiCu(2, 1, 0, 0, 2, 0)].Add(phiCu(1, 2, 1, 0, 2, 1, 0, 0)); AdjacencyList[phiCu(2, 1, 0, 0, 2, 0)].Add(phiCu(1, 2, 1, 0, 2, 1, 0, 0)); AdjacencyList[phiCu(2, 1, 0, 0, 2, 0)].Add(phiCu(1, 1, 1, 1, 1, 0, 0)); AdjacencyList[phiCu(2, 1, 0, 0, 2, 0)].Add(phiCu(1, 1, 1, 1, 1, 1, 0)); AdjacencyList[phiCu(2, 1, 0, 0, 2, 0, 0)].Add(phiCu(1, 1, 1, 1, 1, 0, 0)); AdjacencyList[phiCu(1, 1, 0, 0, 0, 0, 0, 0, 0)].Add(phiCu(1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,</pre>		
<pre>//Now we look at the corners AdjacencyList[phiCu(0, 0, 0]].Add(phiCu(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, 1, 0)); AdjacencyList[phiCu(2 * N, 0, 0]].Add(phiCu(2 * N, 0, 0)); AdjacencyList[phiCu(0, 2 * N, 0)].Add(phiCu(1, 2 * N, 0)); AdjacencyList[phiCu(0, 2 * N, 0)].Add(phiCu(1, 2 * N, 0)); AdjacencyList[phiCu(0, 2 * N, 0)].Add(phiCu(1, 2 * N, 0)); AdjacencyList[phiCu(0, 2 * N, 0)].Add(phiCu(1, 2 * N, 1, 0)); AdjacencyList[phiCu(0, 2 * N, 0)].Add(phiCu(1, 2 * N, 1, 0)); AdjacencyList[phiCu(0, 2 * N, 0]].Add(phiCu(1, 2 * N, 1, 0)); AdjacencyList[phiCu(0, 2 * N, 0]].Add(phiCu(1, 2 * N, 1, 0)); AdjacencyList[phiCu(0, 0 = 2 * N)].Add(phiCu(2 * N, 2 * N, 0)); AdjacencyList[phiCu(0, 0 = 2 * N)].Add(phiCu(2 * N, 2 * N, 1, 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(2 * N, 2 * N, 0]].Add(phiCu(2 * N, 2 * N, 1)); AdjacencyList[phiCu(2 * N, 2 * N, 0]].Add(phiCu(2 * N, 2 * N, 1)); AdjacencyList[phiCu(2 * N, 2 * N, 0]].Add(phiCu(2 * N, 1, 0 * 2 * N)); AdjacencyList[phiCu(2 * N, 0 = * N]].Add(phiCu(2 * N, 1 + 0, 2 * N)); AdjacencyList[phiCu(2 * N, 0 = * N]].Add(phiCu(2 * N, 1 + 0, 2 * N)); AdjacencyList[phiCu(2 * N, 0 = * N]].Add(phiCu(2 * N, 1 + 2 * N); AdjacencyList[phiCu(2 * N, 0 = * N]].Add(phiCu(2 * N, 0 = * N = 1)); AdjacencyList[phiCu(2 * N, 0 = * N]].Add(phiCu(2 * N, 0 = * N = 1)); AdjacencyList[phiCu(2 * N, 2 * N, 2 * N]].Add(phiCu(2 * N, 0 = * N = 1)); AdjacencyList[phiCu(2 * N, 2 * N, 2 * N]].Add(phiCu(2 * N - 1 , 2 * N)); AdjacencyList[phiCu(2 * N, 2 * N, 2 * N]].Add(phiCu(2 * N - 1 , 2 * N)); AdjacencyList[phiCu(2 * N, 2 * N, 2 * N]].Add(phiCu(1 + 1 , 1 , 1); AdjacencyList[phiCu(2 * N, 2 * N, 2 * N]].Add(phiCu(1 + 1 , 1 , 1);); AdjacencyList[phiCu(1 , 1 , N]].Add(phiCu(1 + 1 , 1 , 1);); AdjacencyList[phiCu(1 , 1 , N]].Add(phiCu(1 + 1 , 1 , N)); AdjacencyList[phiCu(1 , 1 , N]].Add(phiCu(1 , 1 + 1 , N)); Adjace</pre>		}
<pre>AdjacencyList[phitu(0, 0, 0)].Add(phitu(1, 0, 0, 0)); AdjacencyList[phitu(0, 0, 0)].Add(phitu(2 * N - 1, 0, 0)); AdjacencyList[phitu(2 * N, 0, 0)].Add(phitu(2 * N, 1, 0)); AdjacencyList[phitu(2 * N, 0, 0)].Add(phitu(2 * N, 0, 0)); AdjacencyList[phitu(0, 2 * N, 0)].Add(phitu(0, 2 * N, 1)); AdjacencyList[phitu(0, 2 * N, 0)].Add(phitu(0, 2 * N, 1)); AdjacencyList[phitu(0, 2 * N, 0)].Add(phitu(0, 2 * N, 1)); AdjacencyList[phitu(0, 0, 2 * N)].Add(phitu(0, 2 * N, 1)); AdjacencyList[phitu(0, 0, 2 * N)].Add(phitu(0, 2 * N, 1)); AdjacencyList[phitu(0, 0, 2 * N)].Add(phitu(0, 0, 2 * N, 1)); AdjacencyList[phitu(0, 0, 2 * N)].Add(phitu(1, 0, 2 * N, 1)); AdjacencyList[phitu(0, 0, 2 * N)].Add(phitu(2 * N, 1 * N, 0)); AdjacencyList[phitu(0, 0, 2 * N)].Add(phitu(2 * N, 1 * N, 0)); AdjacencyList[phitu(2 * N, 2 * N, 0)].Add(phitu(2 * N, 1 * N, 0)); AdjacencyList[phitu(2 * N, 2 * N, 0)].Add(phitu(2 * N, 1 * N, 0)); AdjacencyList[phitu(2 * N, 2 * N, 0)].Add(phitu(2 * N - 1, 2 * N, 1)); AdjacencyList[phitu(2 * N, 2 * N)].Add(phitu(0, 2 * N, 1 * N)); AdjacencyList[phitu(2 * N, 2 * N)].Add(phitu(0, 2 * N, 1 * N)); AdjacencyList[phitu(2 * N, 2 * N)].Add(phitu(2 * N, 0, 2 * N)); AdjacencyList[phitu(2 * N, 0 * N)].Add(phitu(2 * N, 0, 2 * N)); AdjacencyList[phitu(2 * N, 0 * N)].Add(phitu(2 * N, 0, 2 * N, 1)); AdjacencyList[phitu(2 * N, 0 * N)].Add(phitu(2 * N, 0 * N - 1)); AdjacencyList[phitu(2 * N, 2 * N, 2 * N)].Add(phitu(2 * N, 0 * N - 1)); AdjacencyList[phitu(2 * N, 2 * N, 2 * N)].Add(phitu(2 * N, 0 * N - 1)); AdjacencyList[phitu(2 * N, 2 * N, 2 * N)].Add(phitu(2 * N, 0 * N - 1)); AdjacencyList[phitu(2 * N, 2 * N, 2 * N)].Add(phitu(2 * N, 0 * N - 1)); AdjacencyList[phitu(2 * N, 2 * N, 2 * N)].Add(phitu(2 * N, 0 * N - 1, 2 * N)); AdjacencyList[phitu(1, 1, N].Add(phitu(1 + 1, 1, N)); AdjacencyList[phitu(1, 1, N].].Add(phitu(1 + 1, 1, N)); AdjacencyList[phitu(1, 1, N].].Add(phitu(1, 1 + 1, N)); AdjacencyList[phitu(1, 1, N].].Add(phitu(1, 1 + 1, N)); AdjacencyList[phitu(1, 1, N].].Add(phitu(1, 1 + 1, N)); AdjacencyList[phitu(1, 1, N</pre>		//Now we look at the corners
<pre>AdjacencyList[phiCu(0, 0, 0)].Add(phiCu(2, 0, 1)); AdjacencyList[phiCu(2 * N, 0, 0)].Add(phiCu(2 * N, 1, 0)); AdjacencyList[phiCu(0, 2 * N, 0)].Add(phiCu(1, 2 * N, 0)); AdjacencyList[phiCu(0, 2 * N, 0)].Add(phiCu(0, 2 * N, 1)); AdjacencyList[phiCu(0, 2 * N, 0)].Add(phiCu(0, 2 * N, 1)); AdjacencyList[phiCu(0, 2 * N, 0)].Add(phiCu(0, 2 * N, 1)); AdjacencyList[phiCu(0, 0, 2 * N)].Add(phiCu(0, 2 * N, 1)); AdjacencyList[phiCu(0, 0, 2 * N)].Add(phiCu(0, 2 * N, 1)); AdjacencyList[phiCu(0, 0, 2 * N)].Add(phiCu(0, 0, 2 * N, 1)); AdjacencyList[phiCu(0, 0, 2 * N)].Add(phiCu(2 * N, 2 * N, 0)); AdjacencyList[phiCu(0, 0, 2 * N)].Add(phiCu(2 * N, 1 * N, 0)); AdjacencyList[phiCu(0, 0, 2 * N, 2 * N, 0)].Add(phiCu(2 * N, 1 * N, 0)); AdjacencyList[phiCu(0, 2 * N, 2 * N, 0)].Add(phiCu(2 * N, 1 * N, 0)); AdjacencyList[phiCu(2 * N, 2 * N, 0)].Add(phiCu(2 * N - 1, 2 * N, 0)); AdjacencyList[phiCu(2 * N, 2 * N)].Add(phiCu(2 * N - 1, 2 * N)); AdjacencyList[phiCu(0, 2 * N, 2 * N)].Add(phiCu(2 * N - 1, 0, 2 * N)); AdjacencyList[phiCu(0, 2 * N, 2 * N)].Add(phiCu(2 * N - 1, 0, 2 * N)); 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, 0, 2 * N)); 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, 2 * N)); AdjacencyList[phiCu(2 * N, 2 * N, 2 * N)].Add(phiCu(2 * N, 0, 2 * N - 1)); AdjacencyList[phiCu(2 * N, 2 * N, 2 * N)].Add(phiCu(2 * N, 0, 2 * N - 1)); AdjacencyList[phiCu(2 * N, 2 * N, 2 * N)].Add(phiCu(1 - 1, 1, N)); AdjacencyList[phiCu(2 * N, 2 * N, 2 * N)].Add(phiCu(1 - 1, 1, N)); AdjacencyList[phiCu(1, 1, 1, N]].Add(phiCu(1 - 1, 1, N)); AdjacencyList[phiCu(1, 1, 1, N]].Add(phiCu(1, 1 - 1, N)); AdjacencyList[phiCu(1, 1, N]].Add(phiCu(1, 1, 1, N)); AdjacencyL</pre>		
<pre>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(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), 0)); AdjacencyList[phiCu(0, 0, 2 * N)].Add(phiCu(0, 2 * N - 1), 0)); AdjacencyList[phiCu(0, 0, 2 * N)].Add(phiCu(2 * N - 1, 2 * N, 0)); AdjacencyList[phiCu(0, 0, 2 * N)].Add(phiCu(2 * N - 1, 2 * N, 0)); AdjacencyList[phiCu(0, 0, 2 * N)].Add(phiCu(2 * N - 1, 2 * N, 0)); AdjacencyList[phiCu(2 * N, 2 * N, 0)].Add(phiCu(2 * N - 1, 2 * N, 0)); AdjacencyList[phiCu(0, 0, 2 * N)].Add(phiCu(2 * N - 1, 2 * N, 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, 2 * N)); AdjacencyList[phiCu(0, 2 * N, 2 * N)].Add(phiCu(2 * N, 2 * N - 1)); AdjacencyList[phiCu(0, 2 * N, 2 * N)].Add(phiCu(2 * N, 1, 2 * N)); AdjacencyList[phiCu(2 * N, 0, 2 * N)].Add(phiCu(2 * N, 1, 2 * N, 2 * N)); AdjacencyList[phiCu(2 * N, 0, 2 * N)].Add(phiCu(2 * N, 1, 2 * N, 2 * N)); AdjacencyList[phiCu(2 * N, 0, 2 * N)].Add(phiCu(2 * N, 1, 2 * N, 2 * N)); AdjacencyList[phiCu(2 * N, 0, 2 * N)].Add(phiCu(2 * N, 1, 2 * N, 2 * N)); 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(1, 1, 1, 1, 1, N), AdjacencyList[phiCu(1, 1, 1, 1, N], AdjacencyList[phiCu(1, 1, 1, N]), AdjacencyList[phiCu(1, 1, N]].Add(phiCu(1, 1, 1, 1, N)); AdjacencyList[phiCu(1, 1, N]].Ad</pre>		
<pre>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(0, 2 * N, 0)].Add(phicu(2 * N, 0, 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, 0)); AdjacencyList[phicu(0, 0, 2 * N)].Add(phicu(0, 1, 2 * N)); AdjacencyList[phicu(0, 0, 2 * N)].Add(phicu(2 * N, 2 * N, 1)); AdjacencyList[phicu(0, 0, 2 * N)].Add(phicu(2 * N, 2 * N, 1)); AdjacencyList[phicu(0 * N, 2 * N, 0]].Add(phicu(2 * N, 2 * N, 1)); AdjacencyList[phicu(2 * N, 2 * N, 0]].Add(phicu(2 * N, 2 * N, 1)); AdjacencyList[phicu(2 * N, 2 * N, 0]].Add(phicu(2 * N, 2 * N, 1)); AdjacencyList[phicu(2 * N, 2 * N, 0]].Add(phicu(2 * N, 2 * N, 1)); AdjacencyList[phicu(2 * N, 2 * N, 0]].Add(phicu(2 * N, 2 * N, 1)); AdjacencyList[phicu(2 * N, 2 * N)].Add(phicu(2 * N, 1, 0, 2 * N)); AdjacencyList[phicu(2 * N, 0 * N)].Add(phicu(2 * N, 1, 0, 2 * N)); AdjacencyList[phicu(2 * N, 0 * N)].Add(phicu(2 * N - 1, 0, 2 * N)); AdjacencyList[phicu(2 * N, 0 * N)].Add(phicu(2 * N, 1, 0, 2 * N)); AdjacencyList[phicu(2 * N, 0 * N)].Add(phicu(2 * N, 1, 0, 2 * N)); AdjacencyList[phicu(2 * N, 0 * N)].Add(phicu(2 * N, 1, 0, 2 * N)); AdjacencyList[phicu(2 * N, 0 * N)].Add(phicu(2 * N, 1, 0, 2 * N)); AdjacencyList[phicu(2 * N, 0 * 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)); AdjacencyList[phicu(2 * N, 2 * N, 2 * N)].Add(phicu(2 * N, 2 * N, 2 * N)); AdjacencyList[phicu(1, 1, N, 1, N); AdjacencyList[phicu(1, 1, N].Add(phicu(1 - 1, 1, N)); AdjacencyList[phicu(1, 1, N].Add(phicu(1 + 1, 1, N)); AdjacencyList[phicu(1, 1, N].Add(phicu(1 + 1, 1, N)); AdjacencyList[phicu(1, 1, N].Add(phicu(1, 1, 1, 1, N)); AdjacencyList[phicu(1, 1, N].Add(phicu(1</pre>		AdjacencyList[phiCu(0, 0, 0)].Add(phiCu(0, 0, 1));
<pre>AdjacencyList[phicu(2 * N, 0, 0)].Add(phicu(2 * N, 1, 0)); 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, 0, 2 * N)].Add(phicu(0, 1, 2 * N)); AdjacencyList[phicu(0, 0, 2 * N)].Add(phicu(0, 1, 2 * N)); AdjacencyList[phicu(0, 0, 2 * N)].Add(phicu(0, 1, 2 * N)); 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, 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, 0)); AdjacencyList[phicu(2 * N, 2 * N)].Add(phicu(2 * N, 2 * N - 1)); AdjacencyList[phicu(2 * N, 0 * N)].Add(phicu(2 * N - 1, 2 * N)); AdjacencyList[phicu(2 * N, 0 * N)].Add(phicu(2 * N - 1, 0, 2 * N)); AdjacencyList[phicu(2 * N, 0 * N)].Add(phicu(2 * N - 1, 0, 2 * N)); AdjacencyList[phicu(2 * N, 0 * N)].Add(phicu(2 * N - 1, 0, 2 * N)); AdjacencyList[phicu(2 * N, 0 * N)].Add(phicu(2 * N - 1, 0, 2 * N)); AdjacencyList[phicu(2 * N, 0 * N)].Add(phicu(2 * N - 1, 0, 2 * N)); AdjacencyList[phicu(2 * N, 0 * N)].Add(phicu(2 * N - 1, 2 * N, 2 * N)); AdjacencyList[phicu(2 * N, 0 * N)].Add(phicu(2 * N - 1, 2 * N, 2 * N)); AdjacencyList[phicu(2 * N, 2 * N, 2 * N)].Add(phicu(2 * N - 1, 2 * N)); AdjacencyList[phicu(2 * N, 2 * N, 2 * N)].Add(phicu(1 * N, 2 * N - 1)); AdjacencyList[phicu(1 * N, 2 * N, 2 * N)].Add(phicu(1 * 1, 1, 1, N)); AdjacencyList[phicu(1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1</pre>		AdjacencyList[phiCu(2 * N, 0, 0)].Add(phiCu(2 * N - 1, 0, 0));
<pre>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, 0, 2 * N)].Add(phiCu(0, 1, 2 * N)); AdjacencyList[phiCu(0, 0, 2 * N)].Add(phiCu(1, 0, 2 * N)); 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, 0)); AdjacencyList[phiCu(2 * N, 2 * N, 0)].Add(phiCu(1, 2 * N, 2 * N)); AdjacencyList[phiCu(2 * N, 2 * N, 0)].Add(phiCu(1, 2 * N, 2 * N)); AdjacencyList[phiCu(2 * N, 2 * N)].Add(phiCu(1, 2 * N, 2 * N)); AdjacencyList[phiCu(2 * N, 2 * N)].Add(phiCu(1, 2 * N, 2 * N)); AdjacencyList[phiCu(2 * N, 0 * N)].Add(phiCu(2 * N - 1, 2 * N)); AdjacencyList[phiCu(2 * N, 0, 2 * N)].Add(phiCu(2 * N - 1, 0, 2 * N)); AdjacencyList[phiCu(2 * N, 0 * N)].Add(phiCu(2 * N - 1, 0, 2 * N)); AdjacencyList[phiCu(2 * N, 0 * N)].Add(phiCu(2 * N - 1, 0, 2 * N)); AdjacencyList[phiCu(2 * N, 0 * N)].Add(phiCu(2 * N, 0, 2 * N - 1)); AdjacencyList[phiCu(2 * N, 0 * N)].Add(phiCu(2 * N, 0, 2 * N - 1)); AdjacencyList[phiCu(2 * N, 0 * N)].Add(phiCu(2 * N, 0, 2 * N - 1)); AdjacencyList[phiCu(2 * N, 2 * N, 2 * N)].Add(phiCu(2 * N, 0, 2 * N - 1)); AdjacencyList[phiCu(2 * N, 2 * N, 2 * N)].Add(phiCu(1 * N, 0, 2 * N - 1)); AdjacencyList[phiCu(2 * N, 2 * N, 2 * N)].Add(phiCu(1 * 1, 1, 1, 1, 1); AdjacencyList[phiCu(1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1</pre>		AdjacencyList[phiCu(2 * N, 0, 0)].Add(phiCu(2 * N, 1, 0));
<pre>AdjacencyList[phiCu(0, 2 * N, 0]].Add(phiCu(1, 2 * N, 0)]; AdjacencyList[phiCu(0, 2 * N, 0]].Add(phiCu(1, 0, 2 * N, 1)); AdjacencyList[phiCu(0, 0, 2 * N)].Add(phiCu(1, 0, 2 * N)); AdjacencyList[phiCu(0, 0, 2 * N)].Add(phiCu(1, 0, 2 * N)); 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, - 1, 2 * N, 0)); AdjacencyList[phiCu(2 * N, 2 * N, 0]].Add(phiCu(2 * N, - 1, 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, - 1, 0)); AdjacencyList[phiCu(2 * N, 2 * N, 0]].Add(phiCu(2 * N, - 1, 0, 2 * N)); AdjacencyList[phiCu(2 * N, 2 * N)].Add(phiCu(2 * N, - 1, 0, 2 * N)); 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, 0, 2 * N)); 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, 0, 2 * N)); 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, - 1, 2 * N, 2 * N)); 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, 2 * N)); AdjacencyList[phiCu(2 * N, 2 * N, 2 * N)].Add(phiCu(1 - 1, j, k)); AdjacencyList[phiCu(1, j, k)].Add(phiCu(1 - 1, j, k)); AdjacencyList[phiCu(1, j, k)].Add(phiCu(1 - 1, j, k)); AdjacencyList[phiCu(1, j, k)].Add(phiCu(1, j - 1, k)); AdjacencyList[phiCu(1, j, k)].Add(phiCu(1, j, k - 1)); AdjacencyList[phiCu(1, j, k)].Add(phiCu(1</pre>		AdjacencyList[phiCu(2 * N, 0, 0)].Add(phiCu(2 * N, 0, 1));
<pre>AdjacencyList[phiCu(0, 2 * N, 0)].Add(phiCu(0, 2 * N - 1, 0)); AdjacencyList[phiCu(0, 0, 2 * N)].Add(phiCu(0, 1, 2 * N)); AdjacencyList[phiCu(0, 0, 2 * N)].Add(phiCu(0, 1, 2 * N)); AdjacencyList[phiCu(0 * N, 2 * N, 0)].Add(phiCu(2 * N - 1, 2 * N, 0)); 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(1, 2 * N, 2 * N, 1)); AdjacencyList[phiCu(2 * N, 2 * N, 0)].Add(phiCu(1, 2 * N, 2 * N)); AdjacencyList[phiCu(0, 2 * N, 2 * N)].Add(phiCu(1, 2 * N, 2 * N)); AdjacencyList[phiCu(0, 2 * N, 2 * N)].Add(phiCu(1, 2 * N, 2 * N)); AdjacencyList[phiCu(2 * N, 0 * N)].Add(phiCu(2 * N - 1, 0 * N)); AdjacencyList[phiCu(2 * N, 0 * N)].Add(phiCu(2 * N - 1, 0 * N)); AdjacencyList[phiCu(2 * N, 0 * N)].Add(phiCu(2 * N - 1, 0 * N)); AdjacencyList[phiCu(2 * N, 0 * N)].Add(phiCu(2 * N, 0 * 1, 0 * N)); AdjacencyList[phiCu(2 * N, 0 * N)].Add(phiCu(2 * N, 0 * 1, 0 * N)); AdjacencyList[phiCu(2 * N, 0 * N)].Add(phiCu(2 * N, 0 * 1, 2 * N)); AdjacencyList[phiCu(2 * N, 0 * N, 2 * N)].Add(phiCu(2 * N, 0 * 1, 2 * N)); AdjacencyList[phiCu(2 * N, 2 * N, 2 * N)].Add(phiCu(2 * N, 0 * 1, 2 * N)); AdjacencyList[phiCu(2 * N, 2 * N, 2 * N)].Add(phiCu(2 * N, 0 * 1, 2 * N)); AdjacencyList[phiCu(2 * N, 2 * N, 2 * N)].Add(phiCu(1 - 1, 1, 1, 1, 1); AdjacencyList[phiCu(1, 1, N, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,</pre>		Adiacencylist[nhi(u(0, 2 * N, 0)].Add(nhi(u(1, 2 * N, 0)):
<pre>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(2 * N, 2 * N, 0)].Add(phiCu(0, 1, 2 * N); 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)); AdjacencyList[phiCu(2 * N, 2 * N, 0)].Add(phiCu(2 * N, 2 * N, 0)); AdjacencyList[phiCu(2 * N, 2 * N)].Add(phiCu(2 * N, 2 * N, 0)); AdjacencyList[phiCu(0, 2 * N, 2 * N)].Add(phiCu(1, 2 * N, 2 * N)); AdjacencyList[phiCu(0, 2 * N, 2 * N)].Add(phiCu(2 * N - 1, 2 * N)); 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, 0, 2 * N)); 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, 2 * N)); AdjacencyList[phiCu(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, 1, 2 * N, 1, 2 * N)); AdjacencyList[phiCu(2 * N, 2 * N, 2 * N)].Add(phiCu(2 * N, 2 * N - 1)); //Now we look at the middle of the grid for (int j = 1; j &lt; 2 * N; i++) for (int k = 1; k &lt; 2 * N; i++) for (int k = 1; k &lt; 2 * N; i++) for (int k = 1; k &lt; 2 * N; i++) for (int k = 1; k &lt; 2 * N; i++) for (int k = 1; k &lt; 2 * N; i++) for (int k = 1; k &lt; 2 * N; i++) for (int k = 1; k &lt; 2 * N; i++) for (int k = 1; k &lt; 2 * N; i++) for (int k = 1; k &lt; 2 * N; i++) for (int k = 1; k &lt; 2 * N; i++) for (int k = 1; k &lt; 2 * N; i++) for (int k = 1; k &lt; 2 * N; i++) for (int k = 1; k &lt; 2 * N; i++) for (int k = 1; k &lt; 2 * N; i++) for (int k = 1; k &lt; 2 * N; i++) for (int k = 0; i &lt; caselphiCu(1, j, k)].Add(phiCu(1; j, k )); AdjacencyList[phiCu(1, j, k)].Add(phiCu(1, j, k )); AdjacencyList[phiCu(1, j, k)].Add(phiCu(1, j, k + 1)); AdjacencyList[phiCu(1, j, k)].Add(phiCu(1, j, k + 1)); Ad</pre>		AdjacencyList[phiCu(0, 2 * N, 0)].Add(phiCu(0, 2 * N - 1, 0));
<pre>AdjacencyList[phiCu(0, 0, 2 * N)].Add(phiCu(1, 0, 2 * N)); AdjacencyList[phiCu(2 * N, 2 * N, 0)].Add(phiCu(0, 1, 2 * N, 0)); 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)); AdjacencyList[phiCu(2 * N, 2 * N, 0)].Add(phiCu(1, 2 * N, 2 * N, 1)); AdjacencyList[phiCu(2 * N, 2 * N)].Add(phiCu(2 * N, 2 * N, 1)); AdjacencyList[phiCu(2 * N, 2 * N)].Add(phiCu(1, 2 * N, 2 * N)); AdjacencyList[phiCu(2 * N, 2 * N)].Add(phiCu(2 * N, 2 * N, 1)); 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, 1 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 - 1, 2 * N, 2 * N)); AdjacencyList[phiCu(2 * N, 0 2 * N)].Add(phiCu(2 * N - 1, 2 * N, 2 * N)); AdjacencyList[phiCu(2 * N, 0 2 * N)].Add(phiCu(2 * N - 1, 2 * N, 2 * N)); AdjacencyList[phiCu(2 * N, 0 2 * N)].Add(phiCu(2 * N - 1, 2 * N, 2 * N)); AdjacencyList[phiCu(2 * N, 0 2 * N)].Add(phiCu(2 * N - 1, 2 * N, 2 * N - 1)); AdjacencyList[phiCu(2 * N, 0 2 * N)].Add(phiCu(2 * N - 2 * N - 1)); AdjacencyList[phiCu(1, 1 * N, 2 * N, 2 * N)].Add(phiCu(2 * N - 2 * N - 1)); AdjacencyList[phiCu(1, 1 * N, 2 * N, 2 * N)].Add(phiCu(1 - 1, 1, 1, N)); AdjacencyList[phiCu(1, 1 * N)].Add(phiCu(1 - 1, 1, 1, N)); AdjacencyList[phiCu(1, 1 * N)].Add(phiCu(1, 1, 1, 1, N)); AdjacencyList[phiCu(1, 1, N)].Add(phiCu(</pre>		
<pre>AdjacencyList[phiCu(0, 0, 2 * W)].Add(phiCu(0, 1, 2 * N)); AdjacencyList[phiCu(0, 0, 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, 2 * N)); AdjacencyList[phiCu(0, 2 * N, 2 * N)].Add(phiCu(0, 2 * N, 2 * N)); AdjacencyList[phiCu(0, 2 * N, 2 * N)].Add(phiCu(2 * N - 1, 0, 2 * N)); 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, 0, 2 * N)); 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, 2 * N)); AdjacencyList[phiCu(2 * N, 0, 2 * N)].Add(phiCu(2 * N - 1, 2 * N, 2 * N)); AdjacencyList[phiCu(2 * N, 0, 2 * N)].Add(phiCu(2 * N, 1, 2 * N, 2 * N)); 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(1 * N, 2 * N, 2 * N - 1)); AdjacencyList[phiCu(1 * N, 2 * N, 2 * N)].Add(phiCu(1 * N, 2 * N, 2 * N - 1)); AdjacencyList[phiCu(1, j, k)].Add(phiCu(1 * 1, j, k)); AdjacencyList[phiCu(1, j, k)].Add(phiCu(1 * 1, j, k)); AdjacencyList[phiCu(1, j, k)].Add(phiCu(1, j, k, 1, k)); AdjacencyList[phiCu(1, j, k)].Add(phiCu(1, j, k + 1)); AdjacencyList[phiCu(1, j, k)].Add(phiCu(1, j, k + 1)); AdjacencyList[phiCu(2 k k k * k +</pre>		Adiacencylist[nhi(ا(۵, ۵, ۲ * ۱۱) ۵dd/nhi(ا(۱ ۵ ۲ * ۱۱))،
<pre>AdjacencyList[phiCu(@, 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)); AdjacencyList[phiCu(0, 2 * N, 2 * N)].Add(phiCu(1, 2 * N, 2 * N)); AdjacencyList[phiCu(0, 2 * N, 2 * N)].Add(phiCu(1, 2 * N, 2 * N)); AdjacencyList[phiCu(0, 2 * N, 2 * N)].Add(phiCu(2 * N, 1, 2 * N)); AdjacencyList[phiCu(0, 2 * N, 2 * N)].Add(phiCu(2 * N, 1, 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, 1, 2 * N)); AdjacencyList[phiCu(2 * N, 0, 2 * N)].Add(phiCu(2 * N, 1, 2 * N)); AdjacencyList[phiCu(2 * N, 2 * N, 2 * N)].Add(phiCu(2 * N, 1, 2 * N)); 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)); AdjacencyList[phiCu(2 * N, 2 * N, 2 * N)].Add(phiCu(2 * N, 2 * N - 1)); AdjacencyList[phiCu(2 * N, 2 * N, 2 * N)].Add(phiCu(2 * N, 2 * N - 1)); AdjacencyList[phiCu(1 * N, 0, 2 * N ; 1++) for (int i = 1; i &lt; 2 * N; j++) for (int i = 1; i &lt; 2 * N; j++) for (int i = 1; i &lt; 2 * N; j++) for (int k = 1; k &lt; 2 * N; k++) { dajacencyList[phiCu(1, j, k)].Add(phiCu(1 - 1, j, k)); AdjacencyList[phiCu(1, j, k)].Add(phiCu(1, j - 1, k)); AdjacencyList[phiCu(1, j, k)].Add(phiCu(1, j - 1, k)); AdjacencyList[phiCu(1, j, k)].Add(phiCu(1, j + 1, k)); AdjacencyList[phiCu(1, j, k)].Add(phiCu(1, j + 1, k)); AdjacencyList[phiCu(1, j, k)].Add(phiCu(1, j k + 1)); } } return AdjacencyList; } //Assigns a new numbering to the graph, the lowest numbers have the least steps from static List(ath)[] BFSgraph = new List(ath)[] 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); return BFSgraph; } //Counts how many vertices are reachable; th+) BFSgraph[j] = new List(ath)[); NewGraphEFS(graph, ref BFSgraph, zero, ref visited); return BFSgraph;</pre>		
<pre>6 AdjacencyList[phiCu(2 * N, 2 * N, 0]].Add(phiCu(2 * N - 1, 2 * N, 0)]; 7 AdjacencyList[phiCu(2 * N, 2 * N, 0]].Add(phiCu(2 * N, 2 * N, 1)); 8 AdjacencyList[phiCu(0, 2 * N, 2 * N)].Add(phiCu(0, 2 * N, 2 * N, 1)); 8 AdjacencyList[phiCu(0, 2 * N, 2 * N)].Add(phiCu(0, 2 * N, 2 * N, 1)); 8 AdjacencyList[phiCu(0, 2 * N, 2 * N)].Add(phiCu(0, 2 * N, 2 * N, 1)); 8 AdjacencyList[phiCu(2 * N, 0, 2 * N)].Add(phiCu(2 * N, 1, 2 * N)); 8 AdjacencyList[phiCu(2 * N, 0, 2 * N)].Add(phiCu(2 * N, 1, 2 * N)); 8 AdjacencyList[phiCu(2 * N, 0, 2 * N)].Add(phiCu(2 * N, 1, 2 * N)); 8 AdjacencyList[phiCu(2 * N, 0, 2 * N)].Add(phiCu(2 * N, 1, 2 * N)); 8 AdjacencyList[phiCu(2 * N, 0, 2 * N)].Add(phiCu(2 * N, -1, 2 * N, 2 * N)); 8 AdjacencyList[phiCu(2 * N, 2 * N, 2 * N)].Add(phiCu(2 * N, -1, 2 * N)); 8 AdjacencyList[phiCu(2 * N, 2 * N, 2 * N)].Add(phiCu(2 * N, -1, 2 * N)); 8 AdjacencyList[phiCu(2 * N, 2 * N, 2 * N)].Add(phiCu(2 * N, -1, 2 * N)); 8 AdjacencyList[phiCu(2 * N, 2 * N, 2 * N)].Add(phiCu(2 * N, 2 * N - 1, 2 * N)); 8 AdjacencyList[phiCu(1 * N, 2 * N, 2 * N)].Add(phiCu(1 * 1, 1, 1, 2 * N)); 8 AdjacencyList[phiCu(1, 1, 1, 1, Add(phiCu(1 - 1, 1, 1, 1, 1)); 9 AdjacencyList[phiCu(1, 1, 1, 1, Add(phiCu(1 - 1, 1, 1, 1, 1)); 9 AdjacencyList[phiCu(1, 1, 1, 1, Add(phiCu(1, 1 - 1, 1, 1, 1)); 9 AdjacencyList[phiCu(1, 1, 1, 1, Add(phiCu(1, 1, 1, 1, 1, 1, 1); 9 AdjacencyList[phiCu(1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1</pre>		
<pre>AdjacencyList[phiCu(2 * N, 2 * N, 0)].Add(phiCu(2 * N, 2 * N, 1)); AdjacencyList[phiCu(8, 2 * N, 2 * N)].Add(phiCu(2 * N, 2 * N, 1)); AdjacencyList[phiCu(8, 2 * N, 2 * N)].Add(phiCu(8, 2 * N, - 1, 2 * N)); AdjacencyList[phiCu(8, 2 * N, 2 * N)].Add(phiCu(8, 2 * N, - 1, 2 * N)); 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, 0 + 2 * N)); AdjacencyList[phiCu(2 * N, 0 + 2 * N)].Add(phiCu(2 * N, 0 + 2 * N - 1)); 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, 0 + 2 * N - 1)); AdjacencyList[phiCu(2 * N, 2 * N, 2 * N)].Add(phiCu(2 * N, 0 + 2 * N - 1)); AdjacencyList[phiCu(2 * N, 2 * N, 2 * N)].Add(phiCu(2 * N, 2 * N - 1)); AdjacencyList[phiCu(2 * N, 2 * N, 2 * N)].Add(phiCu(2 * N, 2 * N - 1)); AdjacencyList[phiCu(2 * N, 2 * N, 2 * N)].Add(phiCu(2 * N, 2 * N - 1)); AdjacencyList[phiCu(2 * N, 2 * N, 2 * N)].Add(phiCu(2 * N, 2 * N, 2 * N - 1)); AdjacencyList[phiCu(1 + 1, 1 + 1)].Add(phiCu(1 + 1, 1 + 1 + 1)); AdjacencyList[phiCu(1, 1 + 1)].Add(phiCu(1 + 1, 1 + 1)); AdjacencyList[phiCu(1, 1 + 1)].Add(phiCu(1 + 1, 1 + 1)); AdjacencyList[phiCu(1, 1 + 1)].Add(phiCu(1 + 1, 1 + 1)); AdjacencyList[phiCu(1 + 1 + 1]].Add(phiCu(1 + 1 + 1 + 1)); AdjacencyList[phiCu(1 + 1 + 1]].Add(phiCu(1 + 1 + 1 + 1)); AdjacencyList[phiCu(1 + 1 + 1]].Add(phiCu(1 + 1 + 1 + 1)); AdjacencyList[phiCu(1 + 1 + 1]].Add(phiCu(1 + 1 + 1 + 1)); AdjacencyList[phiCu(1 + 1 + 1]].Add(phiCu(1 + 1 + 1 + 1)); AdjacencyList[phiCu(1 + 1 + 1]].Add(phiCu(1 + 1 + 1 + 1)); AdjacencyList[phiCu(1 + 1 + 1]].Add(phiCu(1 + 1 + 1 + 1)); AdjacencyList[phiCu(1 + 1 + 1]].Add(phiCu(1 + 1 + 1 + 1)); AdjacencyList[phiCu(1 + 1 + 1]].Add(phiCu(1 + 1 + 1 + 1)); AdjacencyList[phiCu(1 + 1 + 1]].Add(phiCu(1 + 1 + 1 + 1)); AdjacencyList[phiCu(2 + 1 + 1]].Add(phiCu(2 + 1 + 1]]; AdjacencyList[phiCu(2 + 1 + 1]].Add(phiCu(2 + 1 + 1]]]; AdjacencyList[phiCu(2 + 1 + 1]].Add(phiCu(2 + 1 + 1]]]; AdjacencyList[phiCu(2 + 1 + 1</pre>		Adjacency[ist[nhi(u/2 * N 2 * N 0)] Add(nhi(u/2 * N - 1 2 * N 0))
<pre>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, 2 * N)); AdjacencyList[phiCu(0, 2 * N, 2 * N)].Add(phiCu(2 * N, 1, 2 * N)); 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, 0, 2 * N)); AdjacencyList[phiCu(2 * N, 0, 2 * N)].Add(phiCu(2 * N - 1, 2 * N, 2); AdjacencyList[phiCu(2 * N, 0, 2 * N)].Add(phiCu(2 * N - 1, 2 * N, 2 * N)); 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 - 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 - 1, 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; 4 * N)].Add(phiCu(2 * N, 2 * N - 1, 2 * N)); AdjacencyList[phiCu(1, j, N].Add(phiCu(1 - 1, j, K)); AdjacencyList[phiCu(1, j, K]].Add(phiCu(1 - 1, j, K)); AdjacencyList[phiCu(1, j, K]].Add(phiCu(1 - 1, j, K)); AdjacencyList[phiCu(1, j, K]].Add(phiCu(1, j - 1, K)); AdjacencyList[phiCu(1, j, K]].Add(phiCu(1, j, k + 1)); AdjacencyList[phiCu(1, j, K]].Add(phiCu(1, j, K</pre>		
<pre>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(2 * N, 0, 2 * N)].Add(phiCu(0, 2 * N, 1, 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, 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, 0, 2 * N, 1, 2 * N)); AdjacencyList[phiCu(2 * N, 2 * N, 2 * N)].Add(phiCu(2 * N, 0 + 1, 2 * N, 2 * N)); AdjacencyList[phiCu(2 * N, 2 * N, 2 * N)].Add(phiCu(2 * N, 2 * N, 2 * N, 1, 2 * N)); AdjacencyList[phiCu(2 * N, 2 * N, 2 * N)].Add(phiCu(2 * N, 2 * N, 2 * N - 1)); AdjacencyList[phiCu(2 * N, 2 * N, 2 * N)].Add(phiCu(2 * N, 2 * N, 2 * N - 1)); AdjacencyList[phiCu(2 * N, 2 * N, 2 * N)].Add(phiCu(2 * N, 2 * N, 2 * N - 1)); AdjacencyList[phiCu(2 * N, 2 * N, 2 * N)].Add(phiCu(1 - 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 + 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]].Ad</pre>		
<pre>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, -1, 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, 0, 2 * N)].Add(phiCu(2 * N, 0, 2 * N - 1)); AdjacencyList[phiCu(2 * N, 2 * N, 2 * N)].Add(phiCu(2 * N, 0 + N - 1, 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 - 1, 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(1 + 1, 2 * N, 2 * N - 1)); //Now we look at the middle of the grid for (int i = 1; i &lt; 2 * N; j++) for (int i = 1; i &lt; 2 * N; j++) for (int i = 1; i &lt; 2 * N; j++) 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)); AdjacencyList[phiCu(i, j, k)].Add(phiCu(i, j, k - 1)); AdjacencyList[phiCu(i, j, k)].Add(phiCu(i, j, k - 1)); AdjacencyList; } } 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 static Listcint&gt;[] NumberBFS(Listcint&gt;[] 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); Listcint&gt;[] BFSgraph = new Listcint&gt;[reachable]; for (int i = 0; i &lt; reachable; i++) BFSgraph[i] = new Listcint&gt;[reachable]; for (int i = 0; i &lt; reachable; i++) BFSgraph[i] = new Listcint&gt;[reachable]; it return BFSgraph; } //Counts how many vertices are reachable by walks of length N</pre>		
<pre>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, 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, 1, 2 * N)); AdjacencyList[phiCu(2 * N, 2 * N, 2 * N)].Add(phiCu(2 * N, 2 * N - 1), 2 * N, 2 * N)); AdjacencyList[phiCu(2 * N, 2 * N, 2 * N)].Add(phiCu(2 * N, 2 * N, 2 * N)); AdjacencyList[phiCu(2 * N, 2 * N, 2 * N)].Add(phiCu(2 * N, 2 * N, 2 * N)); AdjacencyList[phiCu(2 * N, 2 * N, 2 * N)].Add(phiCu(2 * N, 2 * N, 2 * N)); AdjacencyList[phiCu(2 * N, 2 * N, 2 * N)].Add(phiCu(2 * N, 2 * N, 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 &lt; 2 * N; i++)</pre>		
<pre>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)); 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, 2 * N, 2 * N)); AdjacencyList[phiCu(2 * N, 2 * N, 2 * N)].Add(phiCu(2 * N, 2 * N, 2 * N)); AdjacencyList[phiCu(2 * N, 2 * N, 2 * N)].Add(phiCu(2 * N, 2 * N, 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 &lt; 2 * N; i++)</pre>		
<pre>AdjacencyList[phiCu(2 * N, 0, 2 * N)].Add(phiCu(2 * N, 0, 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, 2 * N, 2 * N)); AdjacencyList[phiCu(2 * N, 2 * N, 2 * N)].Add(phiCu(2 * N, 2 * N, 2 * N)); AdjacencyList[phiCu(2 * N, 2 * N, 2 * N)].Add(phiCu(2 * N, 2 * N, 2 * N)); AdjacencyList[phiCu(2 * N, 2 * N, 2 * N)].Add(phiCu(2 * N, 2 * N, 2 * N)); AdjacencyList[phiCu(2 * N, 2 * N, 2 * N)].Add(phiCu(2 * N, 2 * N, 2 * N - 1)); AdjacencyList[phiCu(2 * N, 2 * N; 1++) for (int i = 1; i &lt; 2 * N; i++) { AdjacencyList[phiCu(1, j, k)].Add(phiCu(i - 1, j, k)); AdjacencyList[phiCu(1, j, k)].Add(phiCu(i + 1, j, k)); AdjacencyList[phiCu(1, j, k)].Add(phiCu(i, j - 1, k)); AdjacencyList[phiCu(1, j, k)].Add(phiCu(1, j + 1, k)); AdjacencyList[phiCu(1, j, k)].Add(phiCu(1, j, k + 1)); AdjacencyList[phiCu(1, j, k)].Add(phiCu(1, j, k + 1)); AdjacencyList[phiCu(1, j, k)].Add(phiCu(1, j, k - 1)); } return AdjacencyList[phiCu(1, j, k)].Add(phiCu(1, j, k - 1)); } } 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 static List<int>[] NumberBFS(List<int>[] graph, int zero) { //We first want to know how many vertices are reachable from zero bool[j visitd = new bool[graph.length]; int reachable = CountBFS(graph, zero, ref visited); List<int>[] BFSgraph = new List<int>[reachable]; for (int i = 0; i &lt; reachable; i++) BFSgraph[j] = new List<int>[reachable]; for (int i = 0; i &lt; reachable; i++) BFSgraph[j] new List<int>[reachable]; for (int i = 0; i &lt; reachable; i++) BFSgraph[j] = new List<int>[reachable]; for (int i = 0; i &lt; reachable; i++) BFSgraph[j] = new List<int>[reachable]; for (int i = 0; i &lt; reachable; i++) BFSgraph[j] = new List<int>[reachable]; for (int i = 0; i &lt; reachable; i++) BFSgraph[j] = new List<int>[reachable]; for visited]; neturn BFSgraph, ref BFSgraph, zero, ref visited]; return BFSgraph; ref BFSgraph, zero, ref visited]; return BFSgraph;</int></int></int></int></int></int></int></int></int></int></pre>		
<pre>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 - 1)); //Now we look at the middle of the grid for (int i = 1; i &lt; 2 * N; i++) for (int i = 1; j &lt; 2 * N; i++) for (int k = 1; k &lt; 2 * N; k++) { AdjacencyList[phiCu(1, j, k)].Add(phiCu(1 - 1, j, k)); AdjacencyList[phiCu(1, j, k)].Add(phiCu(1, j - 1, k)); AdjacencyList[phiCu(1, j, k)].Add(phiCu(1, j, k - 1)); AdjacencyList[phiCu(1, j, k]].Add(phiCu(1, j, k</pre>		
<pre>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 - 1, 2 * N)); AdjacencyList[phiCu(2 * N, 2 * N, 2 * N)].Add(phiCu(2 * N, 2 * N - 1)); //Now we look at the middle of the grid for (int i = 1; i &lt; 2 * N; i++) for (int k = 1; k &lt; 2 * N; k++) { for (int k = 1; k &lt; 2 * N; k++) for (int k = 1; k &lt; 2 * N; k++)</pre>		
<pre>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 - 1)); //Now we look at the middle of the grid for (int i = 1; i &lt; 2 * N; i++) for (int i = 1; i &lt; 2 * N; i++) for (int k = 1; k &lt; 2 * N; i++) { 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)); AdjacencyList[phiCu(i, j, k)].Add(phiCu(i, j, k + 1)); AdjacencyList[phiCu(i, j, k]].Add(phiCu(i, j, k - 1)); } return AdjacencyList; } //AdjacencyList[phiCu(i, j, k]].Add(phiCu(i, j, k - 1)); } return AdjacencyList; } //Assigns a new numbering to the graph, the lowest numbers have the least steps from 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); for (int i = 0; i &lt; reachable; i++) BFSgraph[i] = new List<int>[reachable]; for (int i = 0; i &lt; reachable; i++) BFSgraph[i] = new List<int>(isted); return BFSgraph, ref BFSgraph, zero, ref visited); return BFSgraph; } //Counts how many vertices are reachable by walks of length N</int></int></int></int></pre>		
<pre>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 &lt; 2 * N; i++) for (int j = 1; j &lt; 2 * N; j++) for (int k = 1; k &lt; 2 * N; k++) {</pre>		
<pre>//Now we look at the middle of the grid for (int i = 1; i &lt; 2 * N; i++) for (int j = 1; j &lt; 2 * N; j++) for (int k = 1; k &lt; 2 * N; k++) {</pre>		
<pre>for (int i = 1; i &lt; 2 * N; i++) for (int j = 1; j &lt; 2 * N; j++) for (int k = 1; k &lt; 2 * N; k++) for (int k = 1; k &lt; 2 * N; k++) for (int k = 1; k &lt; 2 * N; k++) for (int k = 1; k &lt; 2 * N; k++) for (int k = 1; k &lt; 2 * N; k++) for (int k = 1; k &lt; 2 * N; k++) for (int k = 1; k &lt; 2 * N; k++) for (int k = 1; k &lt; 2 * N; k++) for (int k = 1; k &lt; 2 * N; k++) for (int k = 1; k &lt; 2 * N; k++) for (int k = 1; k &lt; 2 * N; k++) for (int k = 1; k &lt; 2 * N; k++) for (int k = 1; k &lt; 2 * N; k++) for (int k = 1; k &lt; 2 * N; k++) for (int k = 1; k &lt; 2 * N; k++) for (int k = 1; k &lt; 2 * N; k++) for (int k = 1; k &lt; 2 * N; k++) for (int k = 1; k &lt; 2 * N; k++) for (int k = 1; k &lt; 2 * N; k++) for (int k = 1; k &lt; 2 * N, k+) for (int j, k = 1)); for (int i, int j, int k) for eturn AdjacencyList; for static int phiCu(int i, int j, int k) for eturn i * (4 * N * N + 2 * N + 2) + j * (2 * N + 1) + k; for first want to know how many vertices are reachable from zero fool[] visited = new bool[graph.tength]; for (int i = 0; i &lt; reachable; i++) for (int i =</pre>		
<pre>for (int j = 1; j &lt; 2 * N; j++)</pre>		
<pre> 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4</pre>		
<pre>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, k + 1)); AdjacencyList[phiCu(i, j, k)].Add(phiCu(i, j, k + 1)); AdjacencyList[phiCu(i, j, k)].Add(phiCu(i, j, k - 1)); AdjacencyList; A</pre>		<pre>for (int k = 1; k &lt; 2 * N; k++)</pre>
<pre>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 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 &lt; reachable; i++) BFSgraph[i] = new List<int>(j; NewGraphBFS(graph, zero, ref visited); return BFSgraph; } //Counts how many vertices are reachable by walks of length N</int></int></int></int></int></pre>		{     AdiacencyList[nhiCu(i i k)] Add(nhiCu(i - 1 i k));
<pre>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 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 &lt; reachable; i++) BFSgraph[i] = new List<int>(j; NewGraphBFS(graph, ref BFSgraph, zero, ref visited); return BFSgraph; } //Counts how many vertices are reachable by walks of length N</int></int></int></int></int></pre>		
<pre>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 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 &lt; reachable; i++) BFSgraph[i] = new List<int>(); NewGraphBFS(graph, ref BFSgraph, zero, ref visited); //Counts how many vertices are reachable by walks of length N</int></int></int></int></int></pre>		
AdjacencyList[phiCu(i, j, k)].Add(phiCu(i, j, k - 1)); } return AdjacencyList; } return AdjacencyList; } return AdjacencyList; } 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 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 &lt; 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</int></int></int></int></int>		
<pre>} return AdjacencyList; } return AdjacencyList; } return AdjacencyList; }  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 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 &lt; 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</int></int></int></int></int></pre>		
<pre>} } 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 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 &lt; reachable; i++) BFSgraph[i] = new List<int>(); NewGraphBFS(graph, ref BFSgraph, zero, ref visited); } //Counts how many vertices are reachable by walks of length N</int></int></int></int></int></pre>		•
<pre>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 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 &lt; 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</int></int></int></int></int></pre>		
<pre>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 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 &lt; 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</int></int></int></int></int></pre>		J
<pre>return i * (4 * N * N + 2 * N + 2) + j * (2 * N + 1) + 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 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 &lt; 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</int></int></int></int></int></pre>		<pre>static int phiCu(int i, int j, int k)</pre>
<pre>9 } 1  1  2 //Assigns a new numbering to the graph, the lowest numbers have the least steps from 3 static List<int>[] NumberBFS(List<int>[] graph, int zero) 4  5 //We first want to know how many vertices are reachable from zero 6 bool[] visited = new bool[graph.Length]; 7 int reachable = CountBFS(graph, zero, ref visited); 8  9 List<int>[] BFSgraph = new List<int>[reachable]; 9 for (int i = 0; i &lt; reachable; i++) 9 BFSgraph[i] = new List<int>(); 1 BFSgraph[i] = new List<int>(); 2 NewGraphBFS(graph, ref BFSgraph, zero, ref visited); 3 return BFSgraph; 4 } 5 //Counts how many vertices are reachable by walks of length N</int></int></int></int></int></int></pre>		
<pre>//Assigns a new numbering to the graph, the lowest numbers have the least steps from 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 &lt; 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</int></int></int></int></int></pre>		
<pre>33 static List<int>[] NumberBFS(Listint&gt;[] graph, int zero) 4 { 5</int></pre>		,
<pre>{     {         //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 &lt; 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 </int></int></int></pre>		//Assigns a new numbering to the graph, the lowest numbers have the least steps from
<pre>//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); int reachable = CountBFS(graph, zero, ref visited); ist<int>[] BFSgraph = new List<int>[reachable]; for (int i = 0; i &lt; 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</int></int></int></pre>		
<pre>int reachable = CountBFS(graph, zero, ref visited); left List<int>[] BFSgraph = new List<int>[reachable]; left for (int i = 0; i &lt; reachable; i++) BFSgraph[i] = new List(int&gt;(); NewGraphBFS(graph, ref BFSgraph, zero, ref visited); return BFSgraph; left list left list list list list list list list list</int></int></pre>		•
<pre>List<int>[] BFSgraph = new List<int>[reachable]; for (int i = 0; i &lt; 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</int></int></int></pre>		
<pre>9 List<int>[] BFSgraph = new List<int>[reachable]; 9 for (int i = 0; i &lt; reachable; i++) 9 BFSgraph[i] = new List(int)(); 9 NewGraphBFS(graph, ref BFSgraph, zero, ref visited); 9 return BFSgraph; 9 //Counts how many vertices are reachable by walks of length N</int></int></pre>		<pre>int reachable = CountBFS(graph, zero, ref visited);</pre>
<pre>for (int i = 0; i &lt; 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</int></pre>		<pre>List<int>[] BFSgraph = new List<int>[reachable];</int></int></pre>
NewGraphBFS(graph, ref BFSgraph, zero, ref visited); return BFSgraph; } } //Counts how many vertices are reachable by walks of length N		<pre>for (int i = 0; i &lt; reachable; i++)</pre>
<pre>return BFSgraph; } //Counts how many vertices are reachable by walks of length N</pre>		
4 } 5 . 6 //Counts how many vertices are reachable by walks of length N		
//Counts how many vertices are reachable by walks of length N		
	.5	
	26 27	<pre>//Counts how many vertices are reachable by walks of length N static int CountBFS(List<int>[] graph, int zero, ref bool[] visited)</int></pre>

328

```
{
                  for (int i = 0; i < visited.Length; i++)</pre>
329
330
                      visited[i] = false;
331
                  int count = 0:
332
                  333
                  int[] step = new int[graph.Count()];
step[zero] = 0;
334
335
336
                  Queue<int> q = new Queue<int>();
337
                  q.Enqueue(zero);
338
339
                  visited[zero] = true;
340
341
                  while (q.Count > 0)
342
                  {
343
                       int a = q.Dequeue();
344
                       //If we need more than N steps, we are done
345
                       if (step[a] > N)
346
                          break;
347
                      count += 1;
348
349
                       foreach (int b in graph[a])
350
                           if (!visited[b])
351
                           {
                               visited[b] = true;
352
                               q.Enqueue(b);
353
354
                               step[b] = step[a] + 1;
355
                           }
356
                  }
357
358
                  return count;
359
              }
360
              static void NewGraphBFS(List<int>[] graph, ref List<int>[] BFSgraph, int zero, ref bool[]
361
                                                                                                                       Þ
                visited)
362
              {
                  for (int i = 0; i < visited.Length; i++)
    visited[i] = false;</pre>
363
364
                  //The new numbering
365
366
                  int number = 0;
367
                  //In the array step we save the amount of steps it take to reach a point from zero
                  int[] step = new int[graph.Count()];
368
                  step[zero] = 0;
369
370
                  //In this array we save pi(i) which represents the new site number of i
371
372
                  int[] pi = new int[graph.Length];
373
374
                  Queue<int> q = new Queue<int>();
                  q.Enqueue(zero);
375
376
                  visited[zero] = true;
377
378
                  while (q.Count > 0)
379
                  {
                       int a = q.Dequeue();
380
                       //If we need more than N steps, we are done
381
382
                       if (step[a] > N)
383
                           break;
                      pi[a] = number;
384
385
                      number++;
386
                       foreach (int b in graph[a])
387
                           if (!visited[b])
388
389
                           {
390
                               visited[b] = true;
391
                               q.Enqueue(b);
392
                               step[b] = step[a] + 1;
393
                           }
394
                  }
395
396
                  //We now translate edges in the original numbering to the new numbering % \left( {{{\rm{D}}_{{\rm{D}}}} \right)
                  //We first look at the special site zero
397
398
                  foreach (int i in graph[zero])
399
                      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
400
401
402
                       if (pi[j] > 0)
403
                       {
                           foreach (int k in graph[j])
    if (pi[k] > 0 || k == zero)
404
405
406
                                    BFSgraph[pi[j]].Add(pi[k]);
407
                       }
408
              }
```

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

409 //Recursive function that creates a tree with all walks of length N with startpoint start en 410 P endpoint end //If end is -1, all possible endpoints are allowed 411 412 //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) 413 414 { 415 bool[] visited = new bool[graph.Length]; //In this array we save the walk before we add it to the tree 416 417 int[] R = new int[N + 1]; 418 R[0] = start; 419 420 List<Node> T = new List<Node>(); 421 Node tree = new Node(); 422 tree.newNode(-1, 0, null, null, null); 423 T.Add(tree); 424 425 visited[start] = true; 426 **if** (N != 0) 427 FillTree(N, 0, R, visited, ref T, end, graph, ref walks); return T; 428 } 429 430 static void FillTree(int N, int i, int[] R, bool[] visited, ref List<Node> T, int end, 431 List<int>[] graph, ref long walks) 432 { 433 if (i == N) 434 { 435 if (end == -1 || R[i] == end) 436 { 437 //We always want to have the starting point as the first element, we sort the rest  $\,$   $\,$   $\,$ of the array 438 int[] Rsort = new int[R.Length]; for (int j = 0; j < R.Length; j++)</pre> 439 440 Rsort[j] = R[j]; 441 Array.Sort(Rsort); 442 walks += 1: InsertTree(Rsort, ref T); 443 444 } 445 } else 446 447 { 448 foreach (int r in graph[R[i]]) if (!visited[r]) 449 450 { 451 R[i + 1] = r;452 visited[r] = true; 453 FillTree(N, i + 1, R, visited, ref T, end, graph, ref walks); 454 } 455 456 visited[R[i]] = false; 457 } 458 static void InsertTree(int[] R, ref List<Node> T) 459 460 { 461 Node current = T.First(): 462 //This is the first node we have to add to the tree 463 464 Node Ri = new Node(); T.Add(Ri); 465 int i = 0; 466 while (i < R.Length)</pre> 467 468 { 469 //If the current node doesn't have any children, we know we have to add the rest of R  $\,$   $\,$   $\!$ to the tree 470 if (current.child != null) 471 current = current.child; 472 else 473 { 474 Ri.newNode(R[i], 0, null, null, current); 475 current.child = Ri; 476 break: 477 } bool found = false; 478 479 //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]) 480 481 { 482 i++; 483 found = true; 484 } 485 else

C:\Users\Sarita de Berg\Documents\Scriptie\SAW9\SAW3\SAW\SAW\saw.cs 486 { //We have to add a firstchild 487 488 if (current.site > R[i]) 489 { Ri.newNode(R[i], 0, null, current, current.parent); current.parent.child = Ri; 490 491 492 break: 493 } else while (current.sibling != null && current.sibling.site <= R[i]) 494 495 { 496 current = current.sibling; 497 if (current.site == R[i]) 498 { 499 i++; 500 found = true; 501 break; 502 } 503 } 504 } 505 //Because we know current node is smaller than R[i] and the next greater we know the P place in the linked list of siblings we want to insert R[i] if (!found) 506 507 { 508 Ri.newNode(R[i], 0, null, current.sibling, current.parent); current.sibling = Ri; 509 break; 510 511 } 512 } 513 //We have to add one to the count of the last site 514 if (i == R.Length - 1) 515 Ri.count++; 516 else if (i == R.Length) 517 current.count++; 518 519 else 520 { Node previous = Ri; for (int j = i + 1; j < R.Length - 1; j++) 521 522 523 { Node r = new Node(); r.newNode(R[j], 0, null, null, previous); 524 525 T.Add(r); 526 527 previous.child = r; 528 previous = r; 529 } 530 Node last = new Node(); 531 last.newNode(R[R.Length - 1], 1, null, null, previous); 532 T.Add(last); 533 previous.child = last; 534 } 535 } 536 //Determines the number of SAW using three walks of length N1, N2 and N3  $\,$ 537 static long LengthTripling(List<int>[] graph, int N1, int N2, int N3) 538 539 { //The number of self avoiding walks using length tripling 540 long totalSAW = 0; 541 542 543 int bound = graph.Length - 1; 544 545 long time = 0; 546 long timeS = 0; 547 long timeT = 0; 548 long timeU = 0; 549 long D; 550 List<long> counts1 = new List<long>(); 551 List<long> counts2 = new List<long>(); List<long> counts3 = new List<long>(); 552 553 554 long walks = 0; 555 long Z1, Z2, Z3; int max1 = bound; int max2 = bound; int max3 = bound; List<Node> TreeR = CreateTree(N2, 0, -1, graph, ref walks); 556 557 558 559 walks = 0; 560 List<Node> T1 = CreateTree(N1, 0, -1, graph, ref walks); 561 Z1 = walks; long[] countsT1 = SaveCounts(T1); 562 563 //for all end points of w2 564 for (int r = 1; r < bound + 1; r++)</pre> 565 { 566 //D is the number of walks with the restricted end point of w2

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

567	D = 0;	
568	walks = 0;	
569	List <node> T2 = CreateTree(N2, 0, r, graph, ref walks);</node>	
570		
571	<pre>//If there are no walks 2 that have end point r we can stop</pre>	
572	if (walks > 0)	
573	{	
574	Z2 = walks;	
575	<pre>long[] countsT2 = SaveCounts(T2);</pre>	
576	walks = 0;	
577	List <node> T3 = CreateTree(N3, r, -1, graph, ref walks);</node>	
578	Z3 = walks;	
579	<pre>long[] countsT3 = SaveCounts(T3);</pre>	
580		
581	D = Z1 * Z2 * Z3;	
582		
583	//The first corrections	
584	max1 = bound; max2 = bound; counts1.Clear(); counts2.Clear(); time = 0;	
585	Node[] Bins1 = InitBins(T1, ref max1, 1, 1);	
586	<pre>Node[] Bins2 = InitBins(T2, ref max2, 1, 1);</pre>	
587	D = D - Z3 * CorrectFirstTerms(T1, T2, bound, Bins1, Bins2, ref time, 1, r, counts1, counts2);	P
588		
589	<pre>max2 = bound; max3 = bound; counts2.Clear(); counts3.Clear(); ResetTree(T2,</pre>	P
	counts72); time = 0;	
590	Bins2 = InitBins(T2, ref max2, 1, 2);	
591	Node[] Bins3 = InitBins(T3, ref max3, 1, 2);	
592	D = D - Z1 * CorrectFirstTerms(T2, T3, bound, Bins2, Bins3, ref time, 2, r,	P
	counts2, counts3);	1
593		
595 594	<pre>max1 = bound; max3 = bound; counts1.Clear(); counts3.Clear(); ResetTree(T1,</pre>	P
	counts1); ResetTree(T3, counts1); time = 0;	
595	Bins1 = InitBins(11, ref max1, 1, 3);	
596	Bins3 = InitBins(T3, ref max3, 1, 3);	
597	D = D - Z2 * CorrectFirstFerms(T), T3, bound, Bins1, Bins3, ref time, 3, r,	P
557	counts1, counts3;	
598	councis, councis),	
599	//The second corrections	
600	<pre>max1 = bound; max2 = bound; max3 = bound; counts1.Clear(); counts2.Clear();</pre>	P
501	<pre>counts3.Clear(); timeS = 0; timeT = 0; ResetTree(T1, countsT1); ResetTree(T2, countsT2); ResetTree(T3, countsT3); Bins1 = InitBins(T1, ref max1, 2, 1);</pre>	P
602	Bins2 = InitBins(T2, ref max2, 2, 1);	
603	Bins3 = InitBins(T3, ref max3, 2, 1);	
604	D = D + CorrectSecondTerms(T1, T2, T3, bound, -1, -1, Bins1, Bins2, Bins3, ref	P
004	timeS, ref timeT, 1, r, counts1, counts2, counts3;	
605		
606	<pre>max1 = bound; max2 = bound; max3 = bound; counts1.Clear(); counts2.Clear();</pre>	P
	<pre>counts3.Clear(); timeS = 0; timeT = 0; ResetTree(T1, countsT1); ResetTree(T2, countsT2); ResetTree(T3, countsT3);</pre>	₽
607	Bins1 = InitBins(11, ref max1, 2, 2);	
608	Bins2 = InitBins(T2, ref max2, 2, 2);	
609	Bins = InitBins(13, ref max3, 2, 2);	
610	D = D + CorrectSecondTerms(T2, T1, T3, bound, -1, -1, Bins2, Bins1, Bins3, ref	P
010	timeS, ref timeT, 2, r, counts2, counts1, counts3);	
611	cames, i.e. cames, z, i, countsz, countsz, countsz,	
612	<pre>max1 = bound; max2 = bound; max3 = bound; counts1.Clear(); counts2.Clear();</pre>	P
	<pre>counts3.Clear(); timeS = 0; timeT = 0; ResetTree(T1, counts1); ResetTree(T2,</pre>	P
	counts72); ResetTree(T3, counts73);	· ·
613	Bins1 = InitBins(T1, ref max1, 2, 3);	
614	Bins2 = InitBins(T2, ref max2, 2, 3);	
	Bins2 = Initeins(12, Fer max2, 2, 3); Bins3 = InitBins(T3, ref max3, 2, 3);	
615 616		P
616	D = D + CorrectSecondTerms(T3, T1, T2, bound, -1, -1, Bins3, Bins1, Bins2, ref timeS, ref timeT, 3, r, counts3, counts1, counts2);	P
617	cames, rer camer, 5, r, countss, countss, counts2),	
617 618	//The third corrections	
619	<pre>max1 = bound; max2 = bound; max3 = bound; counts1.Clear(); counts2.Clear();</pre>	P
	counts3.Clear();    timeS = 0;    timeT = 0;    timeU = 0;    ResetTree(T1, countsT1); ResetTree(T2, countsT2);    ResetTree(T3, countsT3);	P
620	Bins1 = InitBins(T1, ref max1, 3, 1);	
621	Bins2 = InitBins(T2, ref max2, 3, 2);	
622	Bins3 = InitBins(T3, ref max3, 3, 3);	
623	D = D - CorrectThirdTerms(T1, T2, T3, bound, -1, -1, -1, Bins1, Bins2, Bins3, ref	P
	<pre>timeS, ref timeU, r, counts1, counts2, counts3);</pre>	
	ResetTree(T1, countsT1);	
624		
624 625		
624 625	<pre>totalSAW += D;</pre>	
624 625 626 627	}	
624 625 626 627	}	
624 625 626 627 628 629	} } return totalSAW;	
624 625 626 627 628 629 630	}	

```
//Stores the counts of the tree in an array
632
633
             static long[] SaveCounts(List<Node> Tree)
634
             {
635
                 long[] counts = new long[Tree.Count()];
636
                 int i = 0:
                 foreach (Node node in Tree)
637
638
                 {
                     counts[i] = node.count;
639
640
                     i++;
641
                 }
642
                 return counts;
643
             }
644
645
             //Resets the counts of the tree
646
             static void ResetTree(List<Node> Tree, long[] counts)
647
             {
648
                 int i = 0;
649
                 foreach (Node node in Tree)
650
                 {
651
                     node.count = counts[i];
                     node.stamp1 = -1;
node.stamp2 = -1;
652
653
                     node.stamp3 = -1;
654
655
                     i++;
                 }
656
657
             }
658
659
             //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
660
               intersection point
661
             //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
662
663
             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)
664
             {
665
                 long Z = 0;
666
                 int bound = -1;
667
                 //We find the highest site number for which the time stamp is time, so the highest active
668
                   site
                 for (int i = maxsite; i >= 0; i--)
669
                     if ((Bins1[i] != null && Bins1[i].stamp1 == time) || (Bins2[i] != null && Bins2
670
                                                                                                                  P
                       [i].stamp1 == time))
671
                      {
672
                          bound = i;
673
                          break;
674
                     }
675
676
                 if (bound == -1 || (bound == 0 && mode == 1))
                 return Z;
//If bound = 0 we can only include bound in S and no more sites
677
678
679
                 if (bound != 0)
680
                 {
                      counts1.Clear():
681
682
                      counts2.Clear();
683
                      int max = 0:
                      //We first look at the contribution for supersets of S not including bound
684
                     UpdateCounts(Bins1, counts1, bound, false, time, ref max);
685
                     UpdateCounts(Bins2, counts2, bound, false, time, ref max);
686
687
                      Z = Z + CorrectFirstTerms(T1, T2, bound - 1, Bins1, Bins2, ref time, mode, r, counts1, 🎓
688
                       counts2);
689
                      RestoreCounts(counts1, Bins1[bound]);
690
                      RestoreCounts(counts2, Bins2[bound]);
691
                      counts1.Clear();
692
                     counts2.Clear();
693
694
                      if (bound != r || (mode == 1 || mode == 3))
695
                      {
696
                          //empty bins and make nodes inactive by increasing the time stamp
697
                          time += 1;
                          for (int s = 0; s < bound; s++)</pre>
698
699
                          {
                              Bins1[s] = null;
700
                              Bins2[s] = null;
701
702
                          }
703
                          max = 0;
                          UpdateCounts(Bins1, counts1, bound, true, time, ref max);
704
705
                          UpdateCounts(Bins2, counts2, bound, true, time, ref max);
706
                          Z = Z - CorrectFirstTerms(T1, T2, max, Bins1, Bins2, ref time, mode, r, counts1,
                           counts2);
707
                          RestoreCounts(counts1, Bins1[bound]);
```

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

-	Sarita de Berg\Documents\Scriptie\SAW9\SAW3\SAW\SAW.cs	10
708	RestoreCounts(counts2, Bins2[bound]);	
709		
710	} //We now look at the contribution of S including bound	
711	if (bound $!= r   $ (mode $== 1   $ mode $== 3$ ))	
712 713	{	
714	<pre>long Z1 = CalcCount(Bins1[bound]);</pre>	
715	<pre>long Z2 = CalcCount(Bins2[bound]);</pre>	
716	Z = Z + Z1 * Z2;	
717	}	
718	return Z;	
719	}	
720		
721	//Calculates the second order correction terms	_
722	//T1 is the tree we are going to intersect with, so suppose we want  A cap B  than T1 is from //T1 is the tree we are going to intersect with, so suppose we want  A cap B  than T1 is from	P
723	w2, T2 from w1 and T3 from w3 //S is the intersection set of T1 and T2 and T of T1 and T3	
723	//The int mode also shows which tree we are going to intersect with, so in this case that is 2	
725	//If we close S before T or T before S, smax or tmax is the final site add to S or T, this is	P
	-1 if it has not been closed	
726	//TimeS and timeT are the timestamps for S and T, they are the number of include operations we	P
	have done	
727	<pre>static long CorrectSecondTerms(List<node> T1, List<node> T2, List<node> T3, int maxsite, int</node></node></node></pre>	P
	<pre>smax, int tmax, Node[] Bins1, Node[] Bins2, Node[] Bins3,</pre>	
728	<pre>ref long timeS, ref long timeT, int mode, int r, List<long> counts1, List<long> counts2,</long></long></pre>	P
720	List <long> counts3)</long>	
729 730	{ long Z = 0;	
730 731	10ng 2 - 0,	
731	<pre>int bound = -1;</pre>	
733	//We find the highest site number for which the time stamp is time, so the highest active	P
	site	
734	<pre>for (int i = maxsite; i &gt;= 0; i)</pre>	
735	<pre>if ((Bins1[i] != null &amp;&amp; CheckTime(timeS, timeT, 1, Bins1[i]))    (Bins2[i] != null &amp;&amp;</pre>	P
	Bins2[i].stamp1 == timeS    (Bins3[i] != null && Bins3[i].stamp2 == timeT)))	
736	{	
737	bound = 1;	
738	break;	
739 740	}	
740	//If we are in mode 1 or 2 we can only add 0 to T, if S has not been closed we can stop	
742	if (bound == -1    (bound == $0$ & (mode == 1    mode == 2) & sam <= $0$ )	
743	return Z;	
744		
745	//If bound = 0 we can only add bound to T	
746	if (bound != 0)	
747	{	
748	int max = 0;	
749	//We first look at the contribution of supersets S and T not including bound	
750	Node site1, site2, site3; counts1.Clear(); counts2.Clear(); counts3.Clear();	
751 752	site1 = Bins1[bound]; site2 = Bins2[bound]; site3 = Bins3[bound];	
753	UpdateCounts2(Bins1, counts1, bound, false, timeS, timeT, 1, ref max);	
754	UpdateCounts2(Bins2, counts2, bound, false, time5, timeT, 2, ref max);	
755	UpdateCounts2(Bins3, counts3, bound, false, timeS, timeT, 3, ref max);	
756	Z = Z + CorrectSecondTerms(T1, T2, T3, bound - 1, smax, tmax, Bins1, Bins2, Bins3, ref	P
	timeS, ref timeT, mode, r, counts1, counts2, counts3);	
757	RestoreCounts(counts1, site1);    RestoreCounts(counts2, site2);    RestoreCounts(counts3,	P
	si te3);	
758	if (smax <= 0.82 Rips2[bound] == null)	
759	if (smax <= 0 && Bins2[bound] != null) {	
760 761	<pre>{     //Now we look at supersets where S does contain bound but T does not</pre>	
761	Z = Z - CorrectSec(11, 12, 13, smax, tmax, Bins1, Bins2, Bins3, ref timeS, ref	P
702	timeT, mode, r, counts1, counts2, counts3, bound, 0, 0, 1);	
763	//We now consider the case where bound is the final site added to S and we do not	P
	add bound to T	
764	if (tmax <= 0)	
765	{	
766	Z = Z + CorrectSec(T1, T2, T3, bound, tmax, Bins1, Bins2, Bins3, ref timeS, ref	FP
767	timeT, mode, r, counts1, counts2, counts3, bound, 0, 2, 1);	
767 768	}	
768	}	
770	if (tmax <= 0 && !(bound == r && (mode == 2    mode == 3)) && Bins3[bound] != null)	
771		
772	//Now we look at supersets where T does contain bound but S does not	
773	Z = Z - CorrectSec(T1, T2, T3, smax, tmax, Bins1, Bins2, Bins3, ref timeS, ref	P
	timeT, mode, r, counts1, counts2, counts3, bound, 0, 1, 0);	
774		
775	//We now consider the case where bound is the final site added to T and we do not	P
	add bound to S	

C.	\llsers\Sari ta	de	Rera\Documents\Scri	riptie\SAW9\SAW3\SAW\SAW\saw.cs

C: \Users\Sar	ita de Berg\Documents\Scriptie\SAW9\SAW3\SAW\SAW\saw.cs	11
776	if (smax <= 0)	
777	{	
778	Z = Z + CorrectSec(T1, T2, T3, smax, bound, Bins1, Bins2, Bins3, ref timeS, ref timeT, mode, r, counts1, counts2, counts3, bound, 0, 1, 2);	P
779	}	
780	}	
781		
782	<pre>if (smax &lt;= 0 &amp;&amp; (tmax &lt;= 0 &amp;&amp; !(bound == r &amp;&amp; (mode == 2    mode == 3))) &amp;&amp; Bins3 [bound] != null &amp;&amp; Bins2[bound] != null)</pre>	P
783	{	
784 785	<pre>//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);</pre>	P
786 787	//We now consider the case where bound is the final site added to S and we do add	P
788	<pre>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);</pre>	₽
789		
790	//We now consider the case where bound is the final site added to T and we do add bound to S	P
791	Z = Z - CorrectSec(T1, T2, T3, smax, bound, Bins1, Bins2, Bins3, ref timeS, ref timeT, mode, r, counts1, counts2, counts3, bound, 0, 0, 2);	P
792	}	
793	}	
794	$\frac{1}{2}$ (the set of the set of	
795	if (tmax > 0    ! (bound == r && (mode == 2    mode == 3)))	
796	{	
797	long Z1, Z2, Z3;	
798	if (smax <= 0)	
799	smax = bound;	
800	if $(tmax \le 0)$	
801	tmax = bound;	
802	Z1 = CalcCount(Bins1[bound]);	
803	Z2 = CalcCount(Bins2[smax]);	
804	Z3 = CalcCount(Bins3[tmax]);	
805	Z = Z + Z1 * Z2 * Z3;	
806	}	
807	return Z;	
808	}	
809		
810		
811	//Function that updates the counts and calculates the result of the recursion of the	P
812	correctionterms //incl1, incl2, incl3 show how we want to update the count: O for true, 1 for false and 2 for	P
813	not updating static long CorrectSec(List <node> T1, List<node> T2, List<node> T3, int smax, int tmax, Node[]</node></node></node>	₽
814	<pre>Bins1, Node[] Bins2, Node[] Bins3, ref long time5, ref long timeT, int mode, int r, List<long> counts1, List<long> counts2, list<long> counts2, int incl int incl int incl int incl 2, int incl 2)</long></long></long></pre>	₽
815	List <long> counts3, int bound, int incl1, int incl2, int incl3) {</long>	
815 816	long result = 0;	
817	if (incl2 != 1) timeS++;	
818	if (incl3 != 1) timeT++;	
819	for (int s = 0; s < bound; s++)	
819		
820	{ Bins1[s] = null:	
821 822	Bins1[s] = null; if (incl2 != 1)	
822 823	Bins2[s] = nu[1;	
824	if(incl 3 != 1)	
825	Bins3[s] = null;	
	DINSS[S] = IIIII;	
826		
827	int max = 0; counts1.Clear(); counts2.Clear(); counts3.Clear();	
828	counts).clear(); counts2.clear(); counts3.clear(); Node site1 = Bins1[bound]; Node site2 = Bins2[bound]; Node site3 = Bins3[bound];	
829	Node Site1 = BINSI[bound]; Node Site2 = BINS2[bound]; Node Site3 = BINS3[bound]; if (incl1 != 1) UpdateCounts2(Bins1, counts1, bound, true, timeS, timeT, 1, ref max);	
830		
831	else UpdateCounts2(Bins1, counts1, bound, false, timeS, timeT, 1, ref max);	
832	<pre>if (incl2 != 1) UpdateCounts2(Bins2, counts2, bound, true, timeS, timeT, 2, ref max); else UpdateCounts2(Bins2, counts2, bound, false, timeS, timeT, 2, ref max);</pre>	
833		
834 835	<pre>if (incl3 != 1) UpdateCounts2(Bins3, counts3, bound, true, timeS, timeT, 3, ref max); else UpdateCounts2(Bins3, counts3, bound, false, timeS, timeT, 3, ref max);</pre>	
836		
837	if (incl1 != 1 && incl2 != 1 && incl3 != 1)	
838	result = CorrectSecondTerms(T1, T2, T3, max, smax, tmax, Bins1, Bins2, Bins3, ref timeS, ref timeT, mode, r, counts1, counts2, counts3);	₽
839	el se	
840	result = CorrectSecondTerms(T1, T2, T3, bound - 1, smax, tmax, Bins1, Bins2, Bins3, ref	P
0.44	timeS, ref timeT, mode, r, counts1, counts2, counts3);	
841	RestoreCounts(counts1, site1);	
842	RestoreCounts(counts2, site2);	
843	RestoreCounts(counts3, site3);	
844		

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

845 846	return result; }	
847	}	
	(Coloulates the third order correction terms	
848	//Calculates the third order correction terms	
849	//S is the intersection set of T1 and T2, T of T2 and T3 and U of T1 and T3	
850	<pre>//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</pre>	P
851		P
852	<pre>ref long timeS, ref long timeT, ref long timeU, int r, List<long> counts1, List<long> counts2, List<long> counts3)</long></long></long></pre>	P
853	{	
854	long $Z = 0;$	
855	·	
856	int bound = -1;	
857	<pre>//We find the highest site number for which the time stamp is time, so the highest active site</pre>	P
858 859	<pre>for (int i = maxsite; i &gt;= 0; i) if ((Bins1[i] != null &amp;&amp; CheckTime2(timeS, timeT, timeU, 1, Bins1[i]))    (Bins2[i] != null &amp;&amp; CheckTime2(timeS, timeT, timeU, 2, Bins2[i]))    (Bins3[i] != null &amp;&amp; CheckTime2(timeS, timeT, timeU, 3, Bins3[i])))</pre>	Р Р
860	{	
861	bound = i;	
862	break;	
863	}	
864		
865	//If S has not been closed yet and bound is zero, we can stop	
866	if (bound == -1    (bound == 0 && smax <= 0))	
867	return Z:	
868		
869	//1: S and U, 2: S and T, 3: T and U	
870	//if bound = 0 we can only add bound to T and/or U	
871	if (bound != 0)	
872		
873	int max = 0;	
874	//We first look at the contribution of supersets S, T and U not including bound	
875	Node site1, site2, site3;	
876	counts1.Clear();    counts2.Clear();    counts3.Clear();	
877	site1 = Bins1[bound];  site2 = Bins2[bound];  site3 = Bins3[bound];	
878	UpdateCounts3(Bins1, counts1, bound, false, timeS, timeT, timeU, 1, ref max);	
879	UpdateCounts3(Bins2, counts2, bound, false, timeS, timeT, timeU, 2, ref max);	
880 881	<pre>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);</pre>	P
882	<pre>RestoreCounts(counts1, site1); RestoreCounts(counts2, site2); RestoreCounts(counts3, site3);</pre>	P
883		
884	//We cannot add any more sites if two sets are close	
885	if ((smax <= 0 && (tmax <= 0    umax <= 0))    (tmax <= 0 && umax <= 0))	
886	{	
887	//Now bound is added to S, but not to T and U, in the second case as final site	
888	if (smax <= 0 && Bins1[bound] != null && Bins2[bound] != null)	
889	{	
890	timeS, ref timeT, ref timeU, r, counts1, counts2, counts3, bound, 0, 1, 1);	P
891	timeS, ref timeT, ref timeU, r, counts1, counts2, counts3, bound, 2, 1, 1);	P
892	//We also add bound to T	
893	if (tmax <= 0 && bound != r && Bins3 != null)	
894 895	{ Z = Z + CorrectThree(T1, T2, T3, smax, tmax, umax, Bins1, Bins2, Bins3, ref	P
00/	timeS, ref timeT, ref timeU, r, counts1, counts2, counts3, bound, 0, 0, 1);	_
896	ref timeS, ref timeT, ref timeU, r, counts1, counts2, counts3, bound, 2, 0,	P P
897		P
	ref timeS, ref timeT, ref timeU, r, counts1, counts2, counts3, bound, 0, 2,	P
000	1);	
898	//We can only close both sets if U has not been closed yet	
899	if (umax <= 0)	
900	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);	P P
901	//We also add bound to U	
902	if (umax <= 0)	
903		
904	Z = Z - CorrectThree(T1, T2, T3, smax, tmax, umax, Bins1, Bins2, Bins3,	PP
	0);	
905	Z = Z + CorrectThree(T1, T2, T3, bound, tmax, umax, Bins1, Bins2, Bins3, ref timeS, ref timeT, ref timeU, r, counts1, counts2, counts3, bound,	P

C:\Users\Sarita	a de Berg\Documents\Scriptie\SAW9\SAW3\SAW\saw.cs	13
906	Z = Z + CorrectThree(T1, T2, T3, smax, bound, umax, Bins1, Bins2, Bins3, ref timeS, ref timeT, ref timeU, r, counts1, counts2, counts3, bound,	P P
	0, 2, 0);	
907	Z = Z + CorrectThree(T1, T2, T3, smax, tmax, bound, Bins1, Bins2,	P
907		P
	Bins3, ref timeS, ref timeT, ref timeU, r, counts1, counts2, counts3, bound,	*
000	0, 0, 2); Z = Z - CorrectThree(T1, T2, T3, bound, bound, umax, Bins1, Bins2,	_
908		P
	Bins3, ref timeS, ref timeT, ref timeU, r, counts1, counts2, counts3, bound,	P
909	2, 2, 0); 7 = 7 ConnectThree(T1 T2 T2 bound three bound Pinc1 Pinc2)	P
505	Z = Z - CorrectThree(T1, T2, T3, bound, tmax, bound, Bins1, Bins2, Bins3, ref timeS, ref timeT, ref timeU, r, counts1, counts2, counts3, bound,	P
	2, 0, 2);	
910	Z = Z - CorrectThree(T1, T2, T3, smax, bound, bound, Bins1, Bins2,	P
	Bins3, ref timeS, ref timeT, ref timeU, r, counts1, counts2, counts3, bound,	P
	0, 2, 2);	
911	}	
912	}	
913	//We do not add bound to T, but we do add it to U	
914	<pre>if (umax &lt;= 0 &amp;&amp; Bins3[bound] != null)</pre>	
915	{	
916	Z = Z + CorrectThree(T1, T2, T3, smax, tmax, umax, Bins1, Bins2, Bins3, ref	P
	timeS, ref timeT, ref timeU, r, counts1, counts2, counts3, bound, 0, 1, 0);	
917	Z = Z - CorrectThree(T1, T2, T3, bound, tmax, umax, Bins1, Bins2, Bins3,	P
	ref timeS, ref timeT, ref timeU, r, counts1, counts2, counts3, bound, 2, 1,	P
		_
918	Z = Z - CorrectThree(T1, T2, T3, smax, tmax, bound, Bins1, Bins2, Bins3,	P
	ref timeS, ref timeT, ref timeU, r, counts1, counts2, counts3, bound, 0, 1,	P
010	2); if (tmpy (= 0)	
919	<pre>if (tmax &lt;= 0) Z = Z + CorrectThree(T1, T2, T3, bound, tmax, bound, Bins1, Bins2,</pre>	-
920	Bins3, ref timeS, ref timeT, ref timeU, r, counts1, counts2, counts3, bound,	8
	2, 1, 2);	
921	}	
922	}	
923	//We add bound to T, first without closing it then with	
924	if (tmax <= 0 && bound != r && Bins2[bound] != null && Bins3[bound] != null)	
925	{	
926	Z = Z - CorrectThree(T1, T2, T3, smax, tmax, umax, Bins1, Bins2, Bins3, ref	P
	timeS, ref timeT, ref timeU, r, counts1, counts2, counts3, bound, 1, 0, 1);	
927	Z = Z + CorrectThree(T1, T2, T3, smax, bound, umax, Bins1, Bins2, Bins3, ref	P
	timeS, ref timeT, ref timeU, r, counts1, counts2, counts3, bound, 1, 2, 1);	
928	//We also add bound to U	
929	if (umax <= 0 && Bins1[bound] != null)	
930	{	
931	Z = Z + CorrectThree(T1, T2, T3, smax, tmax, umax, Bins1, Bins2, Bins3, ref	P
022	timeS, ref timeT, ref timeU, r, counts1, counts2, counts3, bound, 1, 0, 0);	_
932	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,	P P
	0);	
933	Z = Z - CorrectThree(T1, T2, T3, smax, tmax, bound, Bins1, Bins2, Bins3,	P
	ref timeS, ref timeT, ref timeU, r, counts1, counts2, counts3, bound, 1, 0,	P
	2);	
934	//We can only close the two sets if S has not been closed yet	
935	if $(smax <= 0)$	
936	Z = Z + CorrectThree(T1, T2, T3, smax, bound, bound, Bins1, Bins2,	P
	Bins3, ref timeS, ref timeT, ref timeU, r, counts1, counts2, counts3, bound,	P
	1, 2, 2);	
937	}	
938	}	
939	//We only add bound to U	
940	<pre>if (umax &lt;= 0 &amp;&amp; Bins1[bound] != null &amp;&amp; Bins3[bound] != null) </pre>	
941	{     7 = 7 ConnectThree(T1 T2 T2 cmax tmax umax Pinc1 Pinc2 Pinc2 not	-
942	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);	P
042	Z = Z + CorrectThree(T1, T2, T3, smax, tmax, bound, Bins1, Bins2, Bins3, ref	P
943	timeS, ref timeT, ref timeU, r, counts1, counts2, counts3, bound, 1, 1, 2);	*
944	<pre>climes, ref climel, ref climed, r, countsi, countsi, countsi, bound, i, i, z), }</pre>	
945	}	
946	}	
947		
948	if $(tmax > 0    bound != r)$	
949	{ · · · · · · · · · · · · · · · · · · ·	
950	long Z1, Z2, Z3;	
951	<pre>int final1 = bound;</pre>	
952	<pre>int final2 = bound;</pre>	
953	<pre>int final3 = bound;</pre>	
954	if (smax > 0)	
955		
956	$if(\max > 0)$	
957	<pre>final1 = Math.Min(smax, umax); else if (tmax &gt; 0)</pre>	
958 959	<pre>else if (tmax &gt; 0)     final2 = Math.Min(smax, tmax);</pre>	

960 961 if (tmax > 0 && umax > 0)962 final3 = Math.Min(tmax, umax); Z1 = CalcCount(Bins1[final1]); 963 964 Z2 = CalcCount(Bins2[final2]): Z3 = CalcCount(Bins3[final3]); 965 966 Z = Z + Z1 \* Z2 \* Z3;967 968 } 969 return Z; 970 } 971 static long CorrectThree(List<Node> T1, List<Node> T2, List<Node> T3, int smax, int tmax, int 972 umax, Node[] Bins1, Node[] Bins2, Node[] Bins3, ref long timeS, ref long timeT, ref long timeU, int r, List<long> counts1, List<long> 973 counts2, List<long> counts3, int bound, int inclS, int inclT, int inclU) 974 { 975 long result = 0; 976 if (inclS != 1) timeS++; 977 if (inclT != 1) timeT++; if (inclU != 1) timeU++; 978 //If we add bound to the sets that belong to a walk we have to empty the bins 979 for (int s = 0; s < bound; s++) 980 981 { if (inclS != 1 || inclU != 1) 982 Bins1[s] = null; 983 984 if (inclS != 1 || inclT != 1) Bins2[s] = null; 985 986 if (inclT != 1 || inclU != 1) 987 Bins3[s] = null; 988 989 int max = 0; 990 counts1.Clear(); counts2.Clear(); counts3.Clear(); Node site1 = Bins1[bound]; Node site2 = Bins2[bound]; Node site3 = Bins3[bound]; 991 992 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); 993 if (inclS != 1 || inclT != 1) UpdateCounts3(Bins2, counts2, bound, true, timeS, timeT, 994 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, 995 996 timeU, 3, ref max); 997 else UpdateCounts3(Bins3, counts3, bound, false, timeS, timeT, timeU, 3, ref max); 998 if ((inclS != 1 && (inclT != 1 || inclU != 1)) || (inclT != 1 && inclU != 1)) 999 1000 result = CorrectThirdTerms(T1, T2, T3, max, smax, tmax, umax, Bins1, Bins2, Bins3, ref timeS, ref timeT, ref timeU, r, counts1, counts2, counts3); 1001 el se 1002 result = CorrectThirdTerms(T1, T2, T3, bound - 1, smax, tmax, umax, Bins1, Bins2, Bins3, ref timeS, ref timeT, ref timeU, r, counts1, counts2, counts3); 1003 RestoreCounts(counts1, site1); 1004 RestoreCounts(counts2, site2) 1005 RestoreCounts(counts3, site3) 1006 return result: 1007 1008 } 1009 //Initialises the bins 1010 //First max is the max reachable site, at the end it is the maximum non-empty bin 1011 //Term shows for which term we want to initialise the bins 1012 //Mode is only used when calculating the second terms to show which mode we are in 1013 static Node[] InitBins(List<Node> Tree, ref int max, int term, int mode) 1014 1015 { 1016 Node[] bins = new Node[max + 1]; max = 0;1017 1018 foreach (Node node in Tree) 1019 node.sibling = null; foreach (Node node in Tree) 1020 1021 { 1022 if (node.count > 0) 1023 { if (term == 1) InsertBin(node, bins, 0); 1024 else if (term == 2) InsertBin2(node, bins, 0, 0, mode); else if (term == 3) InsertBin3(node, bins, 0, 0, 0, mode); 1025 1026 1027 if (node.site > max) max = node.site; 1028 1029 } 1030 1031 return bins; 1032 } 1033 static void InsertBin(Node node, Node[] bin, long stamp) 1034

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

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

```
1035
              {
1036
                   int s = node.site;
                  if (bin[s] != null && bin[s].stamp1 != stamp)
1037
1038
                       bin[s] = null;
                  node.stamp1 = stamp;
1039
1040
                  node.sibling = bin[s];
1041
                  bin[s] = node;
1042
              }
1043
1044
              static void InsertBin2(Node node, Node[] bin, long timeS, long timeT, int mode)
1045
              {
                  int s = node.site;
1046
1047
                  if (bin[s] != null && !CheckTime(timeS, timeT, mode, bin[s]))
1048
                       bin[s] = null;
1049
                  node.stamp1 = timeS;
1050
                  node.stamp2 = timeT
1051
                  node.sibling = bin[s];
1052
                  bin[s] = node;
1053
              }
1054
1055
              static void InsertBin3(Node node, Node[] bin, long timeS, long timeT, long timeU, int mode)
1056
              {
1057
                  int s = node.site:
1058
                  if (bin[s] != null && !CheckTime2(timeS, timeT, timeU, mode, bin[s]))
1059
                       bin[s] = null;
                  node.stamp1 = timeS;
1060
                  node.stamp2 = timeT;
1061
                  node.stamp3 = timeU
1062
1063
                  node.sibling = bin[s];
1064
                  bin[s] = node;
1065
              }
1066
1067
              //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
1068
                                                                                                                    P
                int max)
1069
              {
                  Node v = bins[bound];
1070
                  Node pv;
1071
                  while (v != null)
1072
1073
                  {
1074
                       pv = v.parent;
                       counts. Add(pv.count);
1075
1076
                       //We only want to add the count if we do not want to include bound in supersets
                       if (pv.site != -1)
1077
1078
                       {
1079
                           if (!incl && pv.stamp1 == time)
1080
                               pv.count += v.count;
1081
                           el se
1082
                           {
1083
                               pv.count = v.count;
1084
                               if (pv.site > max)
1085
                                   max = pv.site;
                               InsertBin(pv, bins, time);
pv.stamp1 = time;
1086
1087
1088
                           }
1089
                       }
                       v = v. sibling;
1090
1091
                  }
1092
              }
1093
              //Mode is 1 when we look at bins1, 2 when looking at bins2 and 3 when looking at bins3
1094
              static void UpdateCounts2(Node[] bins, List<long> counts, int bound, bool incl, long timeS,
1095
                long timeT, int mode, ref int max)
1096
              {
1097
                  Node v = bins[bound];
1098
                  Node pv;
1099
                  while (v != null)
1100
                   {
                       pv = v.parent;
1102
                       counts.Add(pv.count);
                       //We only want to add the count if we do not want to include bound in supersets
1104
                       if (pv.site != -1)
1105
                       {
                           if (!incl && CheckTime(timeS, timeT, mode, pv))
1106
1107
                               pv.count += v.count;
1108
                           el se
1109
                           {
                               pv.count = v.count;
1110
1111
                               if (pv.site > max)
1112
                                   max = pv. site;
1113
                               InsertBin2(pv, bins, timeS, timeT, mode);
1114
                               pv.stamp1 = timeS;
```

```
1115
                               pv.stamp2 = timeT;
1116
                           }
1117
                       }
                       v = v. sibling;
                  }
1119
              }
1121
              //Mode is 1 when we look at bins1, 2 when looking at bins2 and 3 when looking at bins3
1122
              static void UpdateCounts3(Node[] bins, List<long> counts, int bound, bool incl, long timeS,
1123
                                                                                                                     P
                long timeT, long timeU, int mode, ref int max)
1124
              {
1125
                  Node v = bins[bound];
1126
                  Node pv;
1127
                  while (v != null)
1128
                  {
1129
                       pv = v.parent;
1130
                       counts. Add(pv.count);
                       //We only want to add the count if we do not want to include bound in supersets
1131
1132
                       if (pv.site != -1)
1133
                       {
                           if (!incl && CheckTime2(timeS, timeT, timeU, mode, pv))
1134
1135
                               pv.count += v.count;
                           el se
1136
1137
                           {
                               pv.count = v.count;
1138
                               if (pv.site > max)
1139
1140
                                   max = pv.site;
1141
                               InsertBin3(pv, bins, timeS, timeT, timeU, mode);
1142
                               pv.stamp1 = timeS;
1143
                               pv.stamp2 = timeT;
1144
                               pv.stamp3 = timeU;
1145
                           }
1146
                       }
                       v = v. sibling;
1147
1148
                  }
1149
              }
1150
              static bool CheckTime(long timeS, long timeT, int mode, Node v)
1152
              {
                  switch (mode)
1154
                  {
1155
                       case 1:
                           if (v.stamp1 == timeS && v.stamp2 == timeT) return true;
1156
                           else return false;
1157
1158
                       case 2:
1159
                           if (v.stamp1 == timeS) return true;
1160
                           else return false;
1161
                       case 3:
1162
                           if (v.stamp2 == timeT) return true;
1163
                           else return false;
1164
                  }
1165
                  return false;
1166
              }
1167
              static bool CheckTime2(long timeS, long timeT, long timeU, int mode, Node v)
1169
              {
                  switch (mode)
1170
1171
                  {
1172
                      case 1:
                           if (v.stamp1 == timeS && v.stamp3 == timeU) return true;
1173
                           else return false;
1174
1175
                       case 2:
1176
                           if (v.stamp1 == timeS && v.stamp2 == timeT) return true;
1177
                           else return false;
1178
                       case 3:
1179
                           if (v.stamp2 == timeT && v.stamp3 == timeU) return true;
1180
                           else return false;
1181
                  }
1182
                  return false;
1183
              }
1184
              static void RestoreCounts(List<long> counts, Node v)
1185
1186
              {
                  List<long>. Enumerator e = counts. GetEnumerator();
1187
                  while (v != null)
1188
1189
                  {
1190
                       e.MoveNext();
                       v.parent.count = e.Current;
1191
1192
                       v = v. sibling;
1193
                  }
1194
              }
1195
```

```
static long CalcCount(Node v)
1196
1197
                 {
                     long result = 0;
while (v != null)
1198
1199
1200
                      {
                          result += v.count;
1201
                          v = v.sibling;
1202
1203
                      }
1204
                     return result;
1205
                }
1206
            }
1207
1208
            class Node
1209
            {
1210
                 public int site; //site number of node
                 public Int64 count; //number of saw's with this node as highest site number
1211
                public Node child, sibling, parent; //first child, next sibling also used for next node with
the same site number when traversing the tree, parent
1212
                                                                                                                                       P
1213
                 public Int64 stamp1, stamp2, stamp3; //time stamps
1214
                 public void newNode(int s, Int64 c, Node ch, Node si, Node pa)
1215
1216
                 {
                      site = s;
1217
                     count = c;
child = ch;
1218
1219
                     sibling = cn;
sibling = si;
parent = pa;
stamp1 = -1;
stamp2 = -1;
1220
1221
1222
1223
1224
                      stamp3 = -1;
1225
                }
1226
            }
1227 }
1228
```

# 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.