Computational investigations of folded self-avoiding walks related to protein folding

Jacques M. Bahi[a], Christophe Guyeux[a], Kamel Mazouzi[b], Laurent Philippe[a,1,*]

[a]*FEMTO-ST Institute, UMR 6174 CNRS, University of Franche-Comté, Besançon, France*
[b]*Mésocentre de calcul (Computing Center), University of Franche-Comté, Besançon, France*

Abstract

Various subsets of self-avoiding walks naturally appear when investigating existing methods designed to predict the 3D conformation of a protein of interest. Two such subsets, namely the folded and the unfoldable self-avoiding walks, are studied computationally in this article. We show that these two sets are equal and correspond to the whole n-step self-avoiding walks for $n \leqslant 14$, but that they are different for numerous $n \geqslant 108$, which are common protein lengths. Concrete counterexamples are provided and the computational methods used to discover them are completely detailed. A tool for studying these subsets of walks related to both pivot moves and protein conformations is finally presented.

Keywords: Self avoiding walks, protein folding, protein structure prediction

1. Introduction

Self-avoiding walks (SAWs) have been studied over decades for the extent and difficulty of the mathematical problems they provide [1, 2, 3], and for their various contexts of application in physics, chemistry, and biology [4, 5, 6]. Among other things, they are used to model polymers such as DNA, RNAs, and proteins. Numerous protein structure prediction (PSP) software iterate on self-avoiding walk subsets, often not clearly defined, of various lattices. The last produced SAW S has the length of the targeted protein P. When labeling S with the amino acids of P, S is (one of) the best solution(s) according to a scoring function that associates a value to a 2D or 3D conformation (depending on physical properties of the conformation as hydrophobic neighboring residues, etc.).

*Corresponding author
Email addresses: `jacques.bahi@femto-st.fr` (Jacques M. Bahi), `christophe.guyeux@femto-st.fr` (Christophe Guyeux), `kamel.mazouzi@univ-fcomte.fr` (Kamel Mazouzi), `laurent.philippe@femto-st.fr` (Laurent Philippe)
[1]Authors in alphabetic order

In previous studies [7, 8, 9], authors of this manuscript have investigated some dynamic protein folding models. They have shown that the possible sets of conformations reachable by these numerous PSP software are not equal. This raises severe questionings on what is indeed really predicted by such software. In particular, they have shown that software that iteratively stretch the conformation from one amino acid until a self-avoiding walk having the length n of the protein, can reach all the n-step SAWs $\mathfrak{S}_n$. Contrarily, the ones that iterate $\pm 90°$ pivot moves on the n-step straight line can only reach what they call the subset of folded self-avoiding walks $fSAW(n)$. It has been clearly established that, for some well-defined small n's, $fSAW(n) \neq \mathfrak{S}_n$. After having obtained this result, the authors' intention was then to investigate more deeply these new kind of self-avoiding walks and other related subsets of walks they called unfolded SAWs, and to determine consequences of these investigations regarding the protein structure prediction problem.

This article is the third of a series of three researches we publish in that field. In [10] we provide a general presentation of folded and unfoldable SAWs, and the collection of results we have obtained on these objects using both theoretical and computational approaches. Article [11] focuses more specifically on the mathematical study of theses subsets of self-avoiding walks, by proving in particular that the number of unfolded SAWs is infinite. This article presents our computational investigations in detail.

After recalling in the next section the basis of self-avoiding walks, of folded SAWs, and of unfoldable SAWs, we explain in Section 3 how the number of folded self-avoiding walks has been computed and how we checked the unfoldable property in practice. The various methods that have been implemented to find the shortest currently known unfoldable SAW are presented too in this section. Then, in Section 4, some heuristics that could be determinant in further studies concerning these subsets of walks are introduced. The next section contains the last contribution of this research work: a free software realized to facilitate the study of folded and unfoldable SAWs. This document ends by a conclusion section, in which all these contributions are summarized and intended future work is proposed.

2. Presentation of Folded Self-Avoiding Walks

We recall in this section various notions and properties of self-avoiding walks and of some of theirs folded subsets: folded SAWs obtained by iterating pivot moves on the straight line and unfoldable SAWs on which no pivot move can be applied without breaking the self-avoiding property. Authors that would investigate more deeply these walks are referred to [12, 4, 13] for the SAWs in general, and to [10, 11] for the folded case.

2.1. Definitions and Terminologies

Let $\mathbb{N}$ be the set of all natural numbers, $\mathbb{N}^* = \{1, 2, \ldots\}$ the set of all positive integers, and for $a, b \in \mathbb{N}$, $a \neq b$, the notation $[\![a, b]\!]$ stands for the set $\{a, a +$

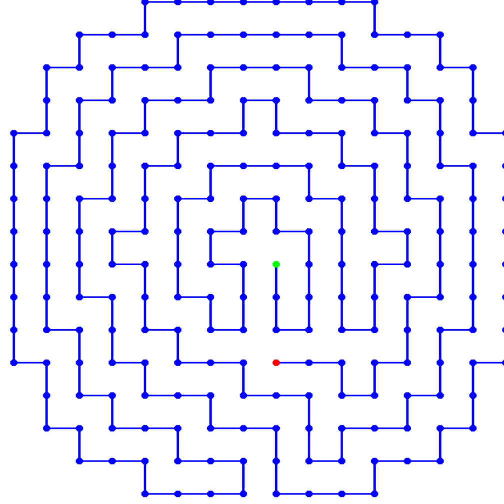


Figure 1: The first SAW shown to be not connected to any other SAW by 90° rotations (Madras and Sokal, [1]).

$1, \ldots, b-1, b\}$. $|x|$ stands for the Euclidean norm of any vector $x \in \mathbb{Z}^d, d \geqslant 1$, whereas $x_1, \ldots, x_n$ are the n coordinates of x. The $n-$th term of a sequence s is denoted by $s(n)$. Finally, $\sharp X$ is the cardinality of a finite set X. Using this material, self-avoiding walk can be defined as follows [12, 4, 13].

Definition 1 (Self-Avoiding Walk) Let $d \geqslant 1$. A $n-$step *self-avoiding walk* from $x \in \mathbb{Z}^d$ to $y \in \mathbb{Z}^d$ is a map $w : [\![0, n]\!] \rightarrow \mathbb{Z}^d$ with:

- $w(0) = x$ and $w(n) = y$,
- $|w(i+1) - w(i)| = 1$,
- $\forall i, j \in [\![0, n]\!], i \neq j \Rightarrow w(i) \neq w(j)$ (self-avoiding property).

2.2. Notations

In absolute encoding [14, 15], a n-step walk $w = w(0), ..., w(n) \in \left(\mathbb{Z}^2\right)^{n+1}$ with $w(0) = (0, 0)$ is a sequence $s = s(0), ..., s(n-1)$ of elements belonging into $\mathbb{Z}/4\mathbb{Z}$, such that:

- $s(i) = 0$ if and only if $w(i+1)_1 = w(i)_1 + 1$ and $w(i+1)_2 = w(i)_2$, that is, $w(i+1)$ is at the East of $w(i)$.
- $s(i) = 1$ if and only if $w(i+1)_1 = w(i)_1$ and $w(i+1)_2 = w(i)_2 - 1$: $w(i+1)$ is at the South of $w(i)$.
- $s(i) = 2$ if and only if $w(i+1)_1 = w(i)_1 - 1$ and $w(i+1)_2 = w(i)_2$, meaning that $w(i+1)$ is at the West of $w(i)$.

- Finally, $s(i) = 3$ if and only if $w(i+1)_1 = w(i)_1$ and $w(i+1)_2 = w(i)_2 + 1$ ($w(i+1)$ is at the North of $w(i)$).

Let us now define the following functions [7].

Definition 2 The *anticlockwise fold function* is the function $f : \mathbb{Z}/4\mathbb{Z} \longrightarrow \mathbb{Z}/4\mathbb{Z}$ defined by $f(x) = x - 1 \pmod 4$ and the clockwise fold function is $f^{-1}(x) = x + 1 \pmod 4$.

Using the absolute encoding sequence s of a $n-$step SAW w that starts from the origin of the square lattice, a pivot move of $+90°$ on $w(k)$, $k < n$, simply consists to transform s into $s(0), \dots, s(k-1), f(s(k)), \dots, f(s(n))$. Similarly, a pivot move of $-90°$ consists to apply f^{-1} to the queue of the absolute encoding sequence, like in Figure 2.

2.3. A graph structure for SAWs folding process

We can now introduce a graph structure that fits the description of iterations for $\pm 90°$ pivot moves on a given self-avoiding walk.

Given $n \in \mathbb{N}^*$, the graph $\mathfrak{G}_n$, formerly introduced in [7], is defined as follows:

- its vertices are the $n-$step self-avoiding walks, described in absolute encoding;
- there is an edge between two vertices s_i, s_j if and only if s_j can be obtained by one pivot move of $\pm 90°$ on s_i, that is, if there exists $k \in [\![0, n-1]\!]$ s.t.:
  - either $s_j(0), \dots, s_j(k-1), f(s_j(k)), \dots, f(s_j(n)) = s_i$
  - or $s_j(0), \dots, s_j(k-1), f^{-1}(s_j(k)), \dots, f^{-1}(s_j(n)) = s_i$.

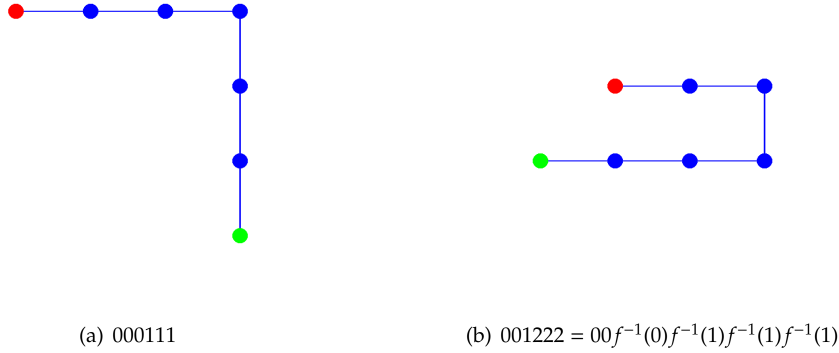
Such a digraph is depicted in Figure 3. The circled vertex is the straight line whereas strikeout vertices are walks that are not self-avoiding. Depending on the context, and for the sake of simplicity, $\mathfrak{G}_n$ will also refers to the set of SAWs in $\mathfrak{G}_n$ (*i.e.*, its vertices).

Using this graph, the folded SAWs introduced in the previous section can be redefined more rigorously.

Definition 3 $fSAW_n$ is the connected component of the straight line $00\dots0$ (n times) in $\mathfrak{G}_n$, whereas $\mathcal{S}_n$ is constituted by all the vertices of $\mathfrak{G}_n$.

The Figure 1 shows that the connected component $fSAW(223)$ of the straight line in $\mathfrak{G}_{223}$ is not equal to the whole graph: $\mathfrak{G}_{223}$ is not connected. More precisely, this graph has a connected component of size 1: it is unfoldable whereas the SAW of Fig. 4 can be folded exactly once. Indeed, to be in the same connected component is an equivalence relation $\mathcal{R}_n$ on $\mathfrak{G}_n, \forall n \in \mathbb{N}^*$, and two SAWs w, w' are considered equivalent (with respect to this equivalence relation) if and only if there is a way to fold w into w' such that all the intermediate walks are self-avoiding. When existing, such a way is not necessarily unique.

These remarks lead to the following definitions.



(a) 000111 (b) $001222 = 00f^{-1}(0)f^{-1}(1)f^{-1}(1)f^{-1}(1)$

Figure 2: Effects of the clockwise fold function applied on the four last components of an absolute encoding.

Definition 4 Let $n \in \mathbb{N}^*$ and $w \in \mathcal{S}_n$. We say that:

- w is *unfoldable* if its equivalence class, with respect to $\mathcal{R}_n$, is of size 1;
- *w is a folded self-avoiding walk* if its equivalence class contains the $n-$step straight walk $000\ldots0$ ($n-1$ times);
- *w can be folded k times* if a simple path of length k exists between w and another vertex in the same connected component of w.

Moreover, we introduce the following sets:

- $fSAW(n)$ is the equivalence class of the $n-$step straight walk, or the set of all folded SAWs.
- $fSAW(n,k)$ is the set of equivalence classes of size k in $(\mathfrak{G}_n, \mathcal{R}_n)$.
- $USAW(n)$ is the set of equivalence classes of size 1 $(\mathfrak{G}_n, \mathcal{R}_n)$, that is, the set of unfoldable walks.
- $f^1SAW(n)$ is the complement of $USAW(n)$ in $\mathfrak{G}_n$. This is the set of SAWs on which we can apply at least one pivot move of $\pm 90°$.

Example 1 Fig. 4 shows the two elements of a class belonging into $fSAW(219,2)$ whereas Fig. 1 is an element of $USAW(223)$.

3. Computing the number of folded self-avoiding walks

3.1. The number $\sharp\mathfrak{G}_n$ of all possible s-step SAWs

In [16], Conway *et al.* have presented an algebraic technique for enumerating self-avoiding walks on a rectangular lattice. To recall in detail how the number

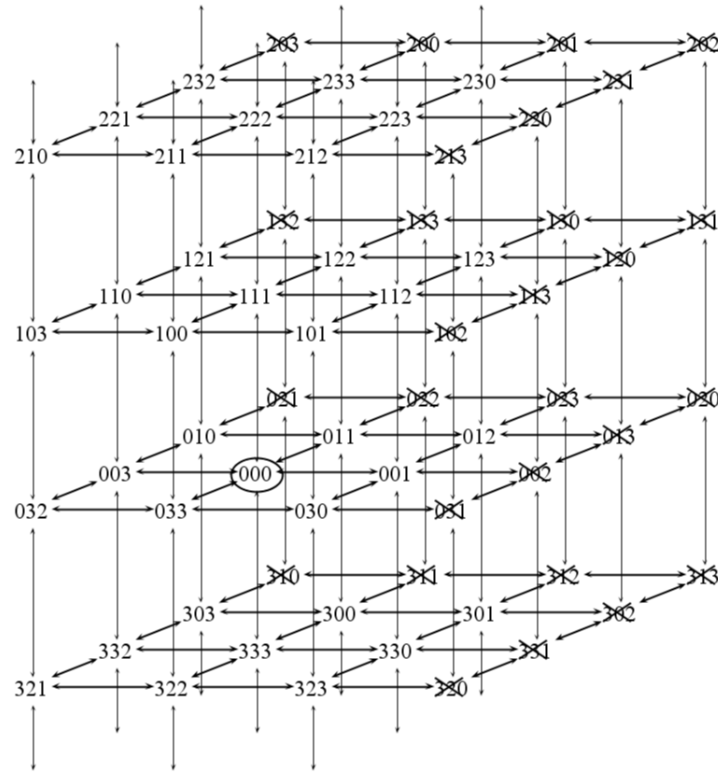


Figure 3: The digraph $\mathfrak{G}(3)$

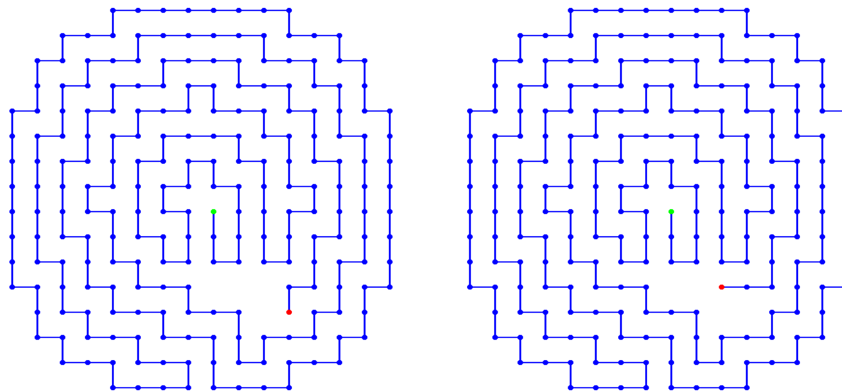


Figure 4: A self-avoiding walk in $fSAW(219, 2)$

$\sharp\mathfrak{S}_n$ of all possible SAWs has been computed until $n = 71$ is not the goal of this article. Let us just remark that they have rewritten the generating function for SAWs on the lattice using 5 irreducible components, whose generating functions are easier to calculate (see Formula 2.17 of [16]). These irreducible components are obtained by considering projections of walks onto the y axis, and classify irreducible segments by the number of y bonds they span. The obtained cardinality of $\mathfrak{S}_n$, is recalled in Table 2 for $n \leqslant 31$. For further detail, the reader is referred to [16].

Authors of this research work have tried to adapt this very interesting approach by searching a way to decompose the generating function of folded self-avoiding walks in other generating functions easier to calculate. The way to take into account pivot moves that define such folded SAWs however has not yet been discovered. This is why the algebraic method has been abandoned after unsuccessful attempts to the benefit of a constructive brute force approach detailed in the following sections.

At each of our investigations, a draft program in Python language has been released first, to test rapidly the correctness of the approach. This code has been translated in an optimized C program and deployed on the supercomputer facilities of the Mésocentre de calcul de Franche-Comté. For readability and compactness, only the python programs are presented thereafter.

3.2. Preliminaries

Python function called *walks* (Listing 5) produces the list of all possible n-step walks as follows: the n-step walks are the walks of length $n-1$ with 0, 1, 2, or 3 added to their tails (recursive call). The return is a list of walks, that is, a list of integers lists.

```
1  def walks(n):
2      if n==1:
3          return [[0]]
4      else:
5          L = []
6          for k in walks(n-1):
7              for i in range(4):
8                  L.append(k+[i])
9          return L
```

Figure 5: Obtaining all the walks

To obtain the walks belonging into $\mathfrak{S}_n$, we first introduce the function *points* which aim is to produce the list of points (two coordinates) of the square lattice that corresponds to a given walk C. The *encodes* function that encodes a walk described as a list of points in its absolute encoding is provided in Listing 6 with its *decodes* associated function.

On Listing 7, Function *is_saw* returns a Boolean: it is true if and only if the walk C satisfies the self-avoiding property. To do so, the list of its points in the lattice (its support) is produced, and it is regarded whether this list contains

```
def encodes(points):
    L=[]
    for k in range(1,len(points)):
        if points[k-1][0]==points[k][0]:
            if points[k-1][1] < points[k][1]:
                L.append(3)
            else:
                L.append(1)
        else:
            if points[k-1][0] < points[k][0]:
                L.append(0)
            else:
                L.append(2)
    return L

def decodes(C):
    L = [(0,0)]
    for c in C:
        P = L[-1]
        if c == 0:   L.append((P[0]+1,P[1]))
        elif c == 1: L.append((P[0],P[1]-1))
        elif c == 2: L.append((P[0]-1,P[1]))
        elif c == 3: L.append((P[0],P[1]+1))
    return L
```

Figure 6: Encoding and decoding walks

twice a same point (in other words, if the support has the same size than the list of points).

Function *saws* produces a generator. It returns the next SAW at each call of the *next* method on the generator. To do so, an exhaustive iteration of the list produced by *walks* is realized, and the *is_saw* function is applied to each element of this list, to test if this walk is self-avoiding.

```
def is_saw(C):
    L = decodes(C)
    return len(L) == len(list(set(L)))

def saws(n):
    for k in walks(n):
        if is_saw(k):
            yield k
```

Figure 7: Finding the self-avoiding walks

These functions have been written and optimized in C language.

3.3. A backtracking method to discover unfoldable walks

The folding property is checked thanks to the functions given on Listing 8. All the possible pivot moves (either in the clockwise direction, or in the anticlockwise) are checked to determine if a self-avoiding walk is unfoldable. The *fold* function tests, considering a given walk, a pivot move on residue number *position* following the given *direction* (+1 or -1, if clockwise or not). Function *is_unfoldable* applies the *fold* function to each residue of the candidate,

and for the two possible directions. The function returns True if and only if no pivot move is possible.

```
1  def fold(walk, position, direction):
2      if position == 0:
3          return walk
4      new = []
5      for k in range(len(walk)):
6          if k<abs(position):
7              new.append(walk[k])
8          else:
9              new.append((walk[k]+direction)%4)
10     return decodes(new)
11
12 def is_unfoldable(saw):
13     for k in range(1,len(saw)):
14         if is_saw(fold(saw, k, -1)):
15             return False
16         elif is_saw(flod(saw, k, +1)):
17             return False
18     return True
```

Figure 8: Testing whether a self-avoiding walk is unfoldable

Listing 9 details how to enumerate walks in $f^1SAW(n)$ (the complement of $USAW(n)$) by constructing with a backtracking method all the k-step walks for k lower than a given threshold n, and increasing a counter at each time the walk is not unfoldable and of length n. The factor 4 at the end of the program is necessary as we always consider the first step of these walks to be 1 (South).

```
1  def stretch(P,c):
2      if c == 0:   return (P[0]+1,P[1])
3      elif c == 1: return (P[0],P[1]-1)
4      elif c == 2: return (P[0]-1,P[1])
5      else :       return (P[0],P[1]+1)
6
7  def backtrack(w,n):
8      global nb
9      if len(w)>=n:
10         return
11     for a in range(4):
12         u = stretch(w[-1],a)
13         if u not in set(w):
14             w1 = w+[u]
15             if not is_unfoldable(encodes(w1)):
16                 if len(w1)==n-1:
17                     nb += 1
18
19             backtrack(w1,n)
20
21 nb=0
22 n=13
23 backtrack(decodes([1]),n+2)
24 print n,4*nb
```

Figure 9: Backtracking method of $USAW(n)$

Backtracking method of Listing 9 has been translated in C language, optimized, paralleled, and launched on the supercomputer facilities (at each time,

SAWs are separated over available processors using MPI routines) as follows.

A first version of the program, with the print line above turned as non commented, has been launched with $N = 14$ and $N = 20$. By doing so, all the walks of $\mathfrak{G}_N$ that start in the East direction have been obtained (for $N = 20$, they represent 224424291 walks stored in 3 gigabytes of data). Then each of these $|\mathfrak{G}_N|/4$ self-avoiding walks has been the starting point of another backtracking discovery until $n > N$, and the unfoldable property of each of these n-step walk has been finally tested. This systematic approach as been successively launched until reaching $n = 28$, see Table 2. For $(N, n) = (20, 28)$, 64 processors have been used during 70 hours in order to test the unfoldable property of 2351378582244 28-step self-avoiding walks, whereas no result has been obtained after 20 days of computation using the same facilities with $(N, n) = (28, 30)$. We can summarize these results as follows.

Proposition 1 $\forall n \leqslant 28, f^1SAW(n) = \mathfrak{G}_n$.

3.4. Investigating the $fSAW(n)$ set

In the previous section, self-avoiding walks that can be folded at least once have been enumerated. That is, the cardinality of $f^1SAW(n)$ has been obtained for $n \leqslant 28$. A breadth first search is now presented to show how $\mathfrak{G}_n = fSAW(n)$ has been obtained for $n \leqslant 14$.

```
1  from networkx import *
2
3  def fold(walk, position, direction):
4      new = []
5      for k in range(len(walk)):
6          if k<abs(position):
7              new.append(walk[k])
8          else:
9              new.append((walk[k]+direction)%4)
10     return new
11
12 def toString(encoding_list):
13     return "".join([str(k) for k in encoding_list])
```

Figure 10: Preliminaries for $fSAW(n)$ investigations

The graph structure used in the Python draft program has been provided by the networkx library [17], which is thus imported in the first line of Listing 10. The graph G representing $\mathfrak{G}(n)$ is instantiated in the first line of Listing 12, and the three following lines of this listing adds the node of the n-step straight line, shown as a word of n zeros. The association between relative encoding of a n-step walk described as an integer list and words on the alphabet $\{0, 1, 2, 3\}$ of length n in the nodes of G is provided by function *toString* of Listing 10.

Let us recall that there is an edge between two nodes in G if and only if the SAW of the second node can be obtained by a pivot move on the first walk. This pivot moves is realized on the integer lists of the relative encoding of the walk using the *fold* function presented in the previous section. This latter must

```
1  def explore(G,node):
2      future_nodes = []
3      for k in range(1,len(node)):
4          for direction in [-1,1]:
5              new = fold(node, k, direction)
6              newString = toString(new)
7              if not G.has_node(newString):
8                  if is_saw(new):
9                      future_nodes.append(new)
10                     G.add_edge(toString(node), newString)
11             else:
12                 G.add_edge(toString(node), newString)
13     return (G,future_nodes)
```

Figure 11: The breadth first search *explore* function

be adapted a little, to match with the fact that G contains words (not lists of integers). This adaptation is also given in Listing 10.

Then the whole connected component of the straight line is constructed using a breadth first search approach: at each iteration, the list of new walks that result from a fold on the last added nodes is obtained (with function *explore* of Listing 11) and required connections are provided between last added walks and new discovered ones. The main issue in this approach is to prevent from visiting twice a given node, which is verified with the *G.has_node* method.

```
1  G = Graph()
2  n=6
3  nodes = [[0]*n]
4  G.add_node(toString(nodes[0]))
5
6  while nodes != []:
7      new_nodes=[]
8      for node in nodes:
9          [G,x] = explore(G,node)
10         for k in x:
11             if k not in new_nodes:
12                 new_nodes.append(k)
13     nodes = new_nodes
14
15 print(4*len(G))
```

Figure 12: Main program to compute the size of the connected component of the straight line.

More precisely, given a last added node, function *explore* realizes a one depth exploration starting from this node, adds the new discovered nodes to G with related edges, and returns G with the list of these new nodes x. This x is used in Listing 12 to constitute the next depth of exploration.

3.5. In search of the shortest unfoldable self-avoiding walks

The current smallest unfoldable self-avoiding walk is a 107-step walk, as depicted in Figure 13. The production of this counterexample and the systematic exploration of the connected component of the straight walk of $\mathfrak{S}_n$ for small n's presented previously allows us to claim that (see Table 2):

Proposition 2 *Let ν_n the smallest $n \geqslant 2$ such that $USAW(n) \neq \emptyset$. Then* $15 \leqslant \nu_n \leqslant 107$*. In other words,* $\forall n \leqslant 14, fSAW(n) = \mathfrak{G}_n$*, whereas* $fSAW(107) \subsetneq \mathfrak{G}_{107}$.

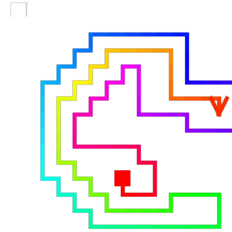


Figure 13: Current smallest (107-step) unfoldable SAW

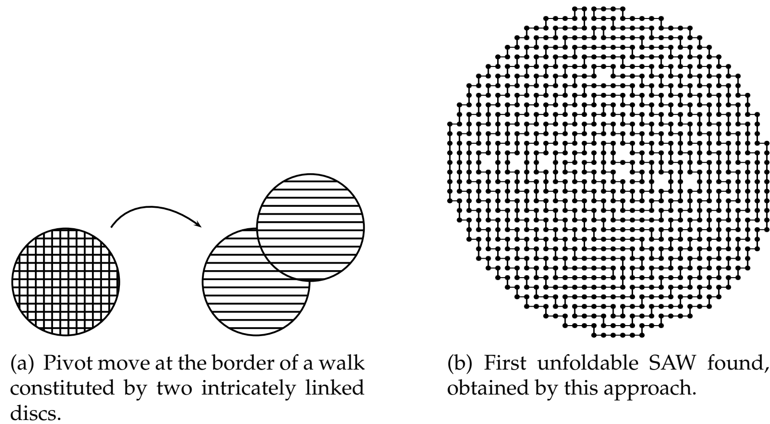
The key idea leading to our first discovery of an unfoldable self-avoiding walk is represented in Figure 14(a): a SAW constituted by two sub-walks that both fill almost 50% of a disc (approximately the same radius for the two sub-walks), when concatenated, can lead to an unfoldable self-avoiding walk if:

- when superposed, they fill almost 100% of the disc,
- the extremities of the concatenated walks are near the center of the superposed disc.

Additionally, it is required that the first sub-walk ends itself by visiting its boundary circle whereas the second sub-walk starts by visiting this circle (more precisely, an equivalent circle with a slight different radius). These two sub-walks have been obtained by following two closed spiral trajectories in the opposite direction.

By this way, no pivot move should be realized without breaking the self-avoiding property. Indeed, due to the compactness of the resulting walk having the form of a disc, no pivot move should be achieved inside the disc, whereas a pivot move at its bounds (the circle) separates the whole disc in two overlapping ones, as depicted in Figure 14(a). A first realization of such an unfoldable SAW is shown in Figure 14(b).

After obtaining our first unfoldable self-avoiding walks, the second stage was to reduce their number of steps by removing the central part of the discs and reducing, bit by bit, its radius. The first operation has been realized by removing the head of the first sub-walk and the tail of the second one, whereas the second operation consists in removing the end (resp., the beginning) of the spirals mentioned above. After having found a self-avoiding walk having the form of a smaller disc, the use of the backtracking program of Listing 9 has often been required to make this SAW as unfoldable. 1840 unfoldable self-avoiding



(a) Pivot move at the border of a walk constituted by two intricately linked discs.

(b) First unfoldable SAW found, obtained by this approach.

Figure 14: Obtaining unfoldable SAW constituted by two overlapped discs

walks have been discovered by doing so, a few of them being represented in Table 1, while they are counted according to their number of steps in Table 2.

A second approach to obtain unfoldable SAWs is to take a compact unfoldable SAW bounded by a kind of circle, and to extend it by two connected spirals covered in opposite direction, as depicted in Figure 15. A proof of the correctness of this approach, leading to an infinite number of unfoldable self-avoiding walks, is detailed in [11]. However, as by this extension-based method it is obviously impossible to obtain the shortest unfoldable self-avoiding walk, such an approach will not be discussed in detail in this research work (for further details about this approach, the reader is referred to [11]).

4. Toward Heuristic Approaches

A first quite optimistic heuristic method to discover smaller unfoldable self-avoiding walks is to follow a Monte-Carlo approach, as described in what follows. Starts first with the origin $(0,0)$, then at each iteration:

1. Pick randomly a stretching direction among the $\{0,1,2,3\}$ set.
2. If it is possible to extend the current walk in that direction while preserving the self-avoiding property, then:
   - do the extension;
   - if the obtained walk has the targeted number of steps, then tests if it is unfoldable.
3. If not, try to pick a new stretching direction again until reaching a predefined number of attempts. If this number is reached, then restart the whole process.

Of course, when picking a new direction in $\{0,1,2,3\}$, the previous one is considered beforehand and the set is adapted: if the previous move was 0, then

STEPS	RELATIVE ENCODING
109	0332223332212121210000111111222333222333333030303000000101010111111122112
	33300333323232322222121212111100011103
111	1233322233333030303000010101011112221110033003333332323232222222121212121111
	110001110003333332222303030300111000112
112	333222333303030300001010101111211103303333332323232222222121212121111110001
	1100333332223303030300111101121003333011
115	333222333303030300001010101111211103303333332323232222222121212121111110001
	1100333332223303030300111101122100033033223
123	2323232333303030300001010101111212121003030333333232323222222212121211111
	101010100033232323233030303001010101122321232330012
145	22232333230003330030100011101011122112211222322223233323333303030303003000
	0010012223222122121212111011101000010330033003232332322211121101003333321
146	22232333230003330030100011101011122112211222322223233323333303030300300000
	0100122232221221212121110111010000103300330032323323222111211010303230032

Table 1: A short list of unfoldable SAWs

the next one is picked in $\{0, 1, 3\}$, as direct reversals are banished to preserve the self-avoiding property.

An implemented improvement of this Monte-Carlo based approach for finding unfoldable self-avoiding walks is to consider that such walks are perhaps more compact than other SAWs, as any pivot move must meet the tail of the structure. To take benefits of such an assessment, assuming that to be true, we have required in a new version of the Monte-Carlo program that the walks must stay in a $N \times N$ lattice, restarting the process at each time a SAW has an height or a width exceeding N. We have experimentally limited the lattice to a square of size 13×13 and to self-avoiding walks having a number of steps lower than 100. However, no smaller unfoldable SAW has been discovered after 15 days of computation on the supercomputer facilities.

Remark that a second version of the backtracking algorithm has been written too, to take into account restrictive $N \times N$ square lattices. However, this program has not allow the author's to discover smaller unfoldable self-avoiding walks than the one depicted in Figure 13.

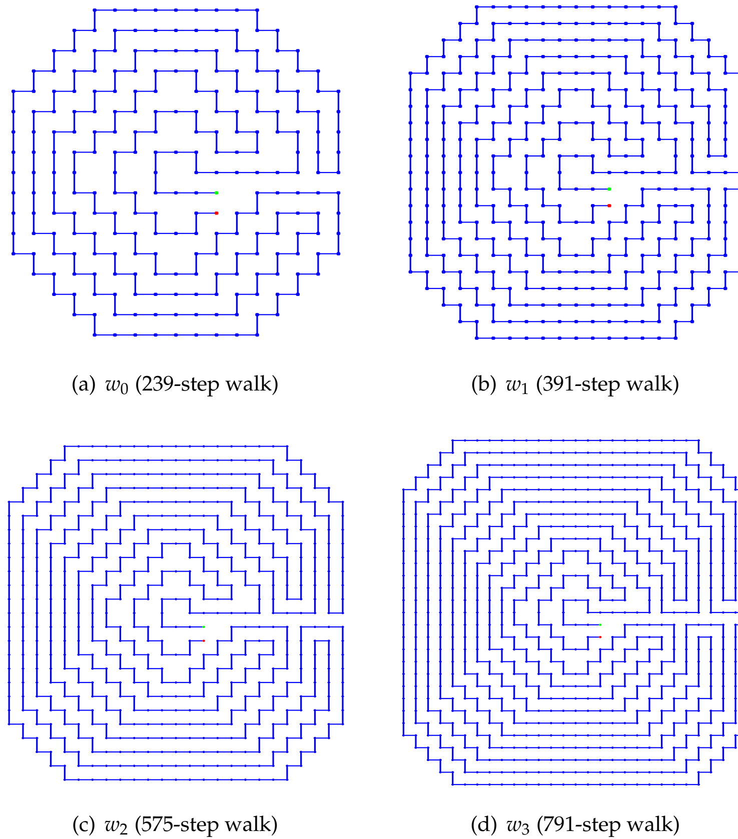
Other investigated approaches encompass stretching method presented in Figure 16, the computing of the fact that the extremities of the walk should be closed one to each other, and an adaptable probability distribution of the stretching direction set $\{0, 1, 2, 3\}$ for favoring the ones that stretch the walk in the center of the walk. They all have lead to a failure in discovering shorter unfoldable self-avoiding walks.

5. PySAW software

To investigate folded and unfoldable self-avoiding walks, we have developed a Python [18] interface hosted in `https://code.google.com/p/pysaw`

n	$\sharp\mathfrak{G}_n$	$\sharp f^1SAW(n)$	$\sharp USAW(n) = \sharp\overline{f^1SAW(n)}$	$\sharp fSAW(n)$
1	4	4	0	4
2	12	12	0	12
3	36	36	0	36
4	100	100	0	100
5	284	284	0	284
6	780	780	0	780
7	2172	2172	0	2172
8	5916	5916	0	5916
9	16268	16268	0	16268
10	44100	44100	0	44100
11	120292	120292	0	120292
12	324932	324932	0	324932
13	881500	881500	0	881500
14	2374444	2374444	0	2374444
15	6416596	6416596	0	?
16	17245332	17245332	0	?
17	46466676	46466676	0	?
18	124658732	124658732	0	?
19	335116620	335116620	0	?
20	897697164	897697164	0	?
21	2408806028	2408806028	0	?
22	6444560484	6444560484	0	?
23	17266613812	17266613812	0	?
24	46146397316	46146397316	0	?
25	123481354908	123481354908	0	?
26	329712786220	329712786220	0	?
27	881317491628	881317491628	0	?
28	2351378582244	2351378582244	0	?
29	6279396229332	?	?	?
30	16741957935348	?	?	?
31	44673816630956	?	?	?
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
107	?	?	$\geqslant 3$	?
108	?	?	$\geqslant 1$	?
111	?	?	$\geqslant 5$	?
112	?	?	$\geqslant 1$	?
113	?	?	$\geqslant 2$	?
114	?	?	$\geqslant 2$	?
115	?	?	$\geqslant 5$	?
116	?	?	$\geqslant 3$	?
117	?	?	$\geqslant 4$	?
118	?	?	$\geqslant 2$	?
119	?	?	$\geqslant 2$	?
121	?	?	$\geqslant 4$	?
122	?	?	$\geqslant 5$	?
123	?	?	$\geqslant 1$	?
132	?	?	$\geqslant 7$	?
133	?	?	$\geqslant 6$	?
134	?	?	$\geqslant 95$	?
135	?	?	$\geqslant 165$	?
136	?	?	$\geqslant 40$	?
137	?	?	$\geqslant 50$	?
138	?	?	$\geqslant 175$	?
139	?	?	$\geqslant 179$	?
140	?	?	$\geqslant 66$	?
141	?	?	$\geqslant 119$	?
142	?	?	$\geqslant 322$	?
143	?	?	$\geqslant 476$	?
144	?	?	$\geqslant 8$	?
145	?	?	$\geqslant 18$	?
146	?	?	$\geqslant 54$	?
235	?	?	$\geqslant 1$	?
239	?	?	$\geqslant 1$	?
391	?	?	$\geqslant 1$	?
575	?	?	$\geqslant 1$	?
791	?	?	$\geqslant 1$	?

Table 2: Cardinality of various subsets of SAWs



(a) w_0 (239-step walk) (b) w_1 (391-step walk)

(c) w_2 (575-step walk) (d) w_3 (791-step walk)

Figure 15: Generating walks that cannot be folded out

and freely available[2]. The interface is depicted in Figure 17 and its current functionality is detailed below.

The interface is divided in four parts: a sandbox where the SAW is drawn, two frames for configuration and backtracking, and the relative encoding of the depicted self-avoiding walk at the bottom of the window. A menu bar with common File, Edit, and Help items completes the interface.

In the sandbox frame, is is possible to construct a SAW step by step by using arrow keys (left, right, up, bottom). The initial vertex is the black square while the green square represents the last vertex. Other vertices and the remainder of the walk are represented in blue. A vertex V is represented by a circle when at least one of the two $\pm 90°$ pivot moves on V leads to a new walk

[2]This python code is maintained by Christophe Guyeux (`christophe.guyeux@univ-fcomte.fr`, any feedback is welcome).

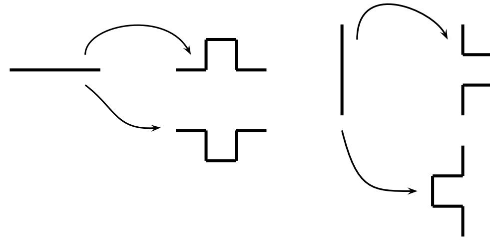


Figure 16: Yet another stretching method

satisfying the self-avoiding property, whereas it is a square when this vertex is unfoldable. Intersections in the walk, for its part, are depicted by a red square, as represented in Figure 17.

Various modification capabilities of the depicted walk have been added. Del key removes the last vertex whereas Back Space one deletes the first vertex. An item in the Edit menu allows the deletion of a sub-walk between two vertices selected by the right mouse button: the two remainder sub-walks are then concatenated. The Edit menu contains also an item that reverses the walk (maps $w(0)...w(n)$ in $w(n)...w(0)$. A click on the left mouse button on a vertex realizes a +90° pivot move on this point whereas a right click makes a pivot move in the reverse direction. Another computed facility is the capability to extend a walk between two points by a right button drag-and-drop: the walk between the two extremities (vertices) of this drag-and-drop is replaced by the walk of the mouse motion. Control-s is a fast save of the walk in *length.txt* file, while Control-q quit the program. Finally, Control-z combination undoes the last modification whereas Control-Z redoes it. Let us remark that, at each modification, the relative encoding of the walk provided at the bottom of the window is naturally updated.

The representation of the walk in the sandbox can be altered by two check boxes in the Configuration panel. The Vertices check box enables or not the representation of the vertices as circles. When enabled, a second check box entitled "Unfoldable" enables the computation of each vertex: unfoldable (square) or not (circle). In Figure 17 this check box is enabled whereas it is disabled in Figure 18. To disable this check box is a necessity for very long walks, due to computation time. Another check box specifies if origin and axes must be represented in the sandbox (the origin can be changed in the Edit menu). Other information represented in the Configuration panel are the number of steps of the walk (length), its width and height, its number of intersections and of unfoldable vertices. Such information is used when representing the walk. Indeed, when the check box "Figure adaptation" is enabled, then the representation of the walk is modified in such a way that this walk is always contained in the frame: the length of the step and the position of the walk are adapted to the frame, depending on the width and height of the walk. If not enabled, the walk can goes outside the frame during a pivot move (for instance), which can be desired to understand the effects of a given pivot move: the head of the

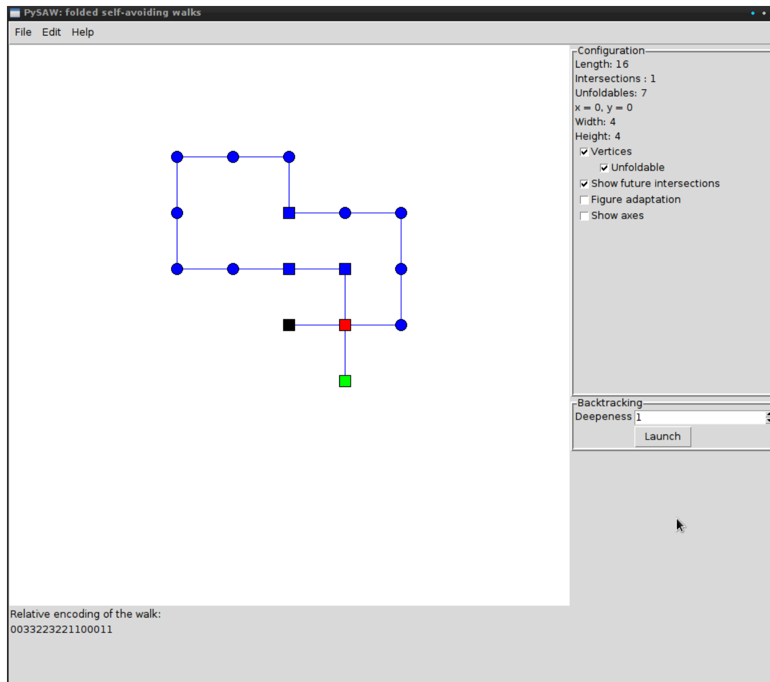


Figure 17: PySAW software

walk does not move, is not adapted to the figure, during this pivot.

The position of the mouse is given too in the Configuration panel, depending on the chosen origin. Initially, this origin is set to the first node of the walk but, as stated before, it is possible to change it using New origin of the Edit menu. When the mouse is positioned on a vertex $w(k)$ of the walk, its position k is informed in parenthesis and, if the "Show future intersection" check box is enabled, the list of intersections implied by $\pm 90°$ pivot moves on $w(k)$ is provided at the bottom of the Configuration panel.

Other functionality of the PyUSAW software encompasses a backtracking search of unfoldable SAWs, based on Listing 9. The self-avoiding walk of the sandbox is extended systematically until reaching the specified deepness and, at each time an unfoldable SAW is found, its relative encoding is printed on the terminal. Drawn self-avoiding walks can be saved as a relative encoding text file, which can contain or not the position of $w(0)$ in the square lattice, and naturally these text files can be opened by PyUSAW. Finally, the walk can be exported too in various image formats (SAWs represented in this research work have been obtained using this interface).

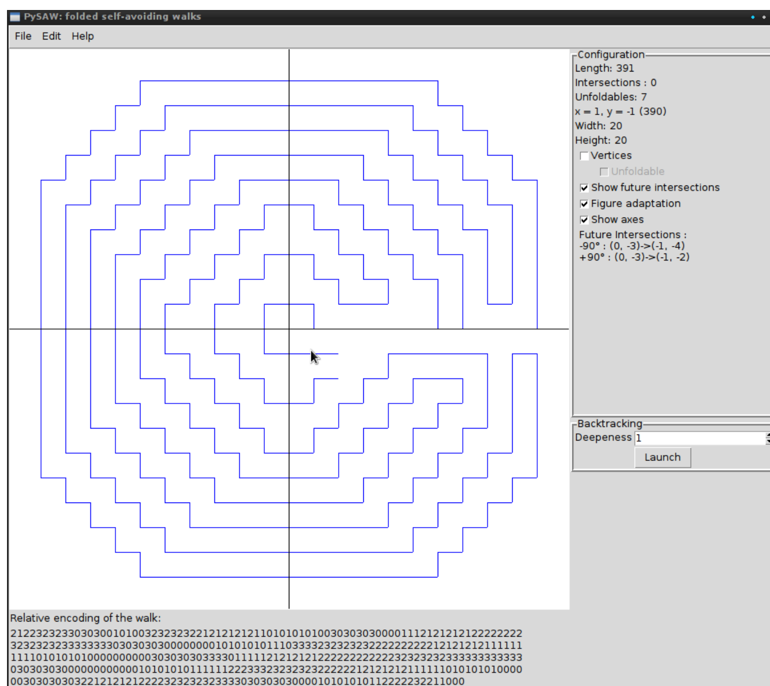


Figure 18: PySAW software

6. Conclusion

In this article, various computational methods have been proposed to investigate the newly discovered folded and unfoldable subsets of walks. These methods encompass backtracking, breadth first search, and Monte-Carlo approaches. The targeted goals was to find when the set of folded SAWs becomes different from the set of all self-avoiding walks, and to discover the shortest unfoldable SAWs. Significant advances have been achieved and explained with details, and an original Python tool to facilitate the study of these important SAW subsets has finally been presented.

In future work, the authors' intention is to restrict the range of uncertainties regarding the smallest n such that $fSAW(n) \neq \mathfrak{S}_n$, which is currently known to belong in $[\![15, 107]\!]$. To do so, computational methods presented in this article, often quite naive, will be enhanced and optimized. Heuristic approaches encompassing swarm particles and genetic algorithms, will be regarded too to discover shorter unfoldable SAWs, if exist. Theoretically speaking, a complexity study of the protein structure prediction problem in the subset of folded SAWs will be realized, and we will try to rewrite the generating function of folded SAWs in other generating functions easier to calculate. Finally, consequences regarding the best ways to make protein structure prediction will be investigate.

Acknowledgment

The authors wish to thank Thibaut Cholley, Raphaël Couturier, Jean-Marc Nicod, and Alain Giorgetti for their help in understanding folded and unfoldable SAWs. All the computations presented in the paper have been performed on the supercomputer facilities of the Mésocentre de calcul de Franche-Comté.

[1] N. Madras, A. D. Sokal, The pivot algorithm: A highly efficient monte carlo method for the self-avoiding walk, Journal of Statistical Physics 50 (1988) 109–186. doi:10.1007/BF01022990.
URL `http://dx.doi.org/10.1007/BF01022990`

[2] A. Bacher, M. Bousquet-Mélou, Weakly directed self-avoiding walks, J. Comb. Theory Ser. A 118 (8) (2011) 2365–2391. doi:10.1016/j.jcta.2011.06.001.
URL `http://dx.doi.org/10.1016/j.jcta.2011.06.001`

[3] N. R. Beaton, P. Flajolet, T. M. Garoni, A. J. Guttmann, Some new self-avoiding walk and polygon models, Fundam. Inf. 117 (1-4) (2012) 19–33.
URL `http://dl.acm.org/citation.cfm?id=2385103.2385107`

[4] G. Slade, The self-avoiding walk: a brief survey., Blath, Jochen (ed.) et al., Surveys in stochastic processes. Selected papers based on the presentations at the 33rd conference on stochastic processes and their applications, Berlin, Germany, July 27–31, 2009. Zürich: European Mathematical Society (EMS). EMS Series of Congress Reports, 181-199 (2011). (2011). doi:10.4171/072.

[5] P. G. de Gennes, Exponents for the excluded volume problem as derived by the Wilson method, Physics Letters A 38 (5) (1972) 339–340. doi:10.1016/0375-9601(72)90149-1.
URL `http://dx.doi.org/10.1016/0375-9601(72)90149-1`

[6] P. J. Flory, The Configuration of Real Polymer Chains, The Journal of Chemical Physics 17 (3) (1949) 303–310. doi:10.1063/1.1747243.
URL `http://dx.doi.org/10.1063/1.1747243`

[7] C. Guyeux, N. M.-L. Côté, W. Bienia, J. Bahi, Is protein folding problem really a NP-complete one? first investigations, Journal of Bioinformatics and Computational Biology * (*) (2013) ***–***, accepted manuscript. To appear.

[8] J. Bahi, N. Côté, C. Guyeux, M. Salomon, Protein folding in the 2D hydrophobic-hydrophilic (HP) square lattice model is chaotic, Cognitive Computation 4 (1) (2012) 98–114. doi:10.1007/s12559-011-9118-z.
URL `http://dx.doi.org/10.1007/s12559-011-9118-z`

[9] J. Bahi, N. Côté, C. Guyeux, Chaos of protein folding, in: IJCNN 2011, Int. Joint Conf. on Neural Networks, San Jose, California, United States, 2011,

pp. 1948–1954. doi:10.1109/IJCNN.2011.6033463.
URL http://dx.doi.org/10.1109/IJCNN.2011.6033463

[10] J. M. Bahi, C. Guyeux, J.-M. Nicod, L. Philippe, Protein structure prediction software generate two different sets of conformations. Or the study of unfolded self-avoiding walks, arXiv:1306.1439arXiv:arXiv.

[11] J. M. Bahi, A. Giorgetti, C. Guyeux, Unfoldable self-avoiding walks are infinite. Consequences for the protein structure prediction problem, arXiv.orgarXiv:arXiv.

[12] N. N. Madras, G. Slade, The self-avoiding walk, Probability and its applications, Birkhäuser, Boston, 1993.
URL http://opac.inria.fr/record=b1082524

[13] B. D. Hughes, Random walks and random environments, Volume 1: Random walks, Clarendon Press, Oxford, 1995.
URL http://www.worldcat.org/isbn/0198537883

[14] M. Hoque, M. Chetty, A. Sattar, Genetic algorithm in ab initio protein structure prediction using low resolution model: A review, in: A. Sidhu, T. Dillon (Eds.), Biomedical Data and Applications, Vol. 224 of Studies in Computational Intelligence, Springer Berlin Heidelberg, 2009, pp. 317–342.

[15] R. Backofen, S. Will, P. Clote, Algorithmic approach to quantifying the hydrophobic force contribution in protein folding (1999).

[16] A. R. Conway, I. G. Enting, A. J. Guttmann, Algebraic techniques for enumerating self-avoiding walks on the square lattice, Journal of Physics A Mathematical General 26 (1993) 1519–1534. arXiv:arXiv:hep-lat/9211062, doi:10.1088/0305-4470/26/7/012.

[17] A. A. Hagberg, D. A. Schult, P. J. Swart, Exploring network structure, dynamics, and function using NetworkX, in: Proceedings of the 7th Python in Science Conference (SciPy2008), Pasadena, CA USA, 2008, pp. 11–15.

[18] www.python.org.
URL \url{http://www.python.org/}