# Efficient Implementation of the Pivot Algorithm for Self-avoiding Walks

**Nathan Clisby**

**Abstract** The pivot algorithm for self-avoiding walks has been implemented in a manner which is dramatically faster than previous implementations, enabling extremely long walks to be efficiently simulated. We explicitly describe the data structures and algorithms used, and provide a heuristic argument that the mean time per attempted pivot for $N$-step self-avoiding walks is $O(1)$ for the square and simple cubic lattices. Numerical experiments conducted for self-avoiding walks with up to 268 million steps are consistent with $o(\log N)$ behavior for the square lattice and $O(\log N)$ behavior for the simple cubic lattice. Our method can be adapted to other models of polymers with short-range interactions, on the lattice or in the continuum, and hence promises to be widely useful.

**Keywords** Self-avoiding walk · Polymer · Monte Carlo · Pivot algorithm

## 1 Introduction and Results

The self-avoiding walk (SAW) model is an important model in statistical physics [16]. It models the excluded-volume effect observed in real polymers, and exactly captures universal features such as critical exponents and amplitude ratios. It is also an important model in the study of critical phenomena, as it is the $n \to 0$ limit of the $n$-vector model, which includes the Ising model ($n = 1$) as another instance. Indeed, one can straightforwardly simulate SAWs in the infinite volume limit, which makes this model particularly favorable for the calculation of critical parameters. Exact results are known for self-avoiding walks in two dimensions [14, 19] and for $d \geq 4$ (mean-field behavior has been proved for $d \geq 5$ [8]), but not for the most physically interesting case of $d = 3$.

The pivot algorithm is a powerful and oft-used approach to the study of self-avoiding walks, invented by Lal [13] and later elucidated and popularized by Madras and Sokal [17]. The pivot algorithm uses pivot moves as the transitions in a Markov chain which proceeds

N. Clisby (✉)
ARC Centre of Excellence for Mathematics and Statistics of Complex Systems, Department of Mathematics and Statistics, The University of Melbourne, Victoria 3010, Australia
e-mail: n.clisby@ms.unimelb.edu.au

as follows. From an initial SAW of length $N$, such as a straight rod, new $N$-step walks are successively generated by choosing a site of the walk at random, and attempting to apply a lattice symmetry operation, or pivot, to one of the parts of the walk; if the resulting walk is self-avoiding the move is accepted, otherwise the move is rejected and the original walk is retained. Thus a Markov chain is formed in the ensemble of SAWs of fixed length; this chain satisfies detailed balance and is ergodic, ensuring that SAWs are sampled uniformly at random.

One typical use of the pivot algorithm is to calculate observables which characterize the size of the SAWs: the squared end-to-end distance $R_e^2$, the squared radius of gyration $R_g^2$, and the mean-square distance of a monomer from its endpoints $R_m^2$. To leading order we expect the mean values of these observables over all SAWs of $N$ steps, with each SAW is given equal weight, to be $\langle R_x^2 \rangle_N \sim D_x N^{2\nu}$ ($x \in \{e, g, m\}$), with $\nu$ a universal critical exponent.

For $N$-step SAWs, the implementation of the pivot algorithm due to Madras and Sokal has estimated mean time per attempted pivot of $O(N^{0.81})$ on $\mathbb{Z}^2$ and $O(N^{0.89})$ on $\mathbb{Z}^3$; performance was significantly improved by Kennedy [9] to $O(N^{0.38})$ and $O(N^{0.74})$ respectively.

In this article, we give a detailed description of a new data structure we call the SAW-tree. This data structure allows us to implement the pivot algorithm in a highly efficient manner: we present a heuristic argument that the mean time per attempted pivot is $O(1)$ on $\mathbb{Z}^2$ and $\mathbb{Z}^3$, and numerical experiments which show that for walks of up to $N = 2^{28} - 1 \approx 2.7 \times 10^8$ steps the algorithmic complexity is well approximated by $O(\log N)$. This improvement enables the rapid simulation of walks with many millions of steps.

In a companion article [4], we describe the algorithm in general terms, and demonstrate the power of the method by applying it to the problem of calculating the critical exponent $\nu$ for three-dimensional self-avoiding walks.

Thus far the SAW-tree has been implemented for $\mathbb{Z}^2$, $\mathbb{Z}^3$, and $\mathbb{Z}^4$, but it can be straightforwardly adapted to other lattices and the continuum, as well as polymer models with short-range interactions. Other possible extensions would be to allow for branched polymers, confined polymers, or simulation of polymers in solution.

We intend to implement the SAW-tree and associated methods as an open source software library for use by researchers in the field of polymer simulation.

## 1.1 Pivot Algorithm

Madras and Sokal [17] demonstrated, through strong heuristic arguments and numerical experiments, that the pivot algorithm results in a Markov chain with short integrated autocorrelation time for global observables. The pivot algorithm is far more efficient than Markov chains which utilize local moves; see [15, 17, 22, 23] for detailed discussion.

The implementation of the pivot algorithm by Madras and Sokal utilized a hash table to record the location of each site of the walk. They showed that for $N$-step SAWs the probability of a pivot move being accepted is $O(N^{-p})$, with $p$ dimension-dependent but close to zero ($p \lesssim 0.2$). As accepted pivots typically result in a large change in global observables such as $R_e^2$, this leads to the conclusion that the pivot algorithm has integrated autocorrelation time $O(N^p)$, with possible logarithmic corrections. In addition, they argued convincingly that the CPU time per successful pivot is $O(N)$ for their implementation. Throughout this article we work with the mean time per attempted pivot, $T(N)$, which for the Madras and Sokal implementation is $O(N^{1-p})$.

Madras and Sokal argued that $O(N)$ per successful pivot is best possible because it takes time $O(N)$ to merely write down an $N$-step SAW. Kennedy [9], however, recognized that it is *not* necessary to write down the SAW for each successful pivot, and developed

**Table 1** $T(N)$, the mean time per attempted pivot for $N$-step SAWs. A tighter bound for $\mathbb{Z}^4$ is reported in Sect. 3.5, but this relies on an untested assumption

| Lattice | Madras and Sokal | Kennedy | This work | |
|---|---|---|---|---|
| | | | Predicted | Observed |
| $\mathbb{Z}^2$ | $O(N^{0.81})$ | $O(N^{0.38})$ | $O(1)$ | $o(\log N)$ |
| $\mathbb{Z}^3$ | $O(N^{0.89})$ | $O(N^{0.74})$ | $O(1)$ | $O(\log N)$ |
| $\mathbb{Z}^4$ | $o(N)$ | ? | $o(\log N)$ | $\omega(\log N)$ |
| $\mathbb{Z}^d, d > 4$ | $O(N)$ | ? | $\Theta(\log N)$ | ? |

a data structure and algorithm which cleverly utilized geometric constraints to break the $O(N)$ barrier. In this paper, we develop methods which further improve the use of geometric constraints to obtain a highly efficient implementation of the pivot algorithm.

## 1.2 Results

We have efficiently implemented the pivot algorithm via a data structure we call the SAW-tree, which allows rapid Monte Carlo simulation of SAWs with millions of steps. This new implementation can also be adapted to other models of polymers with short-range interactions, on the lattice and in the continuum, and hence promises to be widely useful.

The heart of our implementation of the algorithm involves performing intersection tests between "bounding boxes" of different sub-walks when a pivot is attempted. In [4] we generated large samples of walks with up to $2^{25} - 1 \approx 3.3 \times 10^7$ steps, but for the purpose of determining the complexity of our algorithm we have also generated smaller samples of walks of up to $2^{28} - 1 \approx 2.7 \times 10^8$ steps. For $N = 2^{28} - 1$, the mean number of intersection tests needed per attempted pivot is remarkably low: 39 for $\mathbb{Z}^2$, 158 for $\mathbb{Z}^3$, and 449 for $\mathbb{Z}^4$.

In Sect. 3 we present heuristic arguments for the asymptotic behavior of the mean time per attempted pivot for $N$-step SAWs, $T(N)$, and test these predictions with computer experiments for $N \leq 2.7 \times 10^8$. We summarize our results in Table 1; note that $O(f(N))$ indicates $T$ is bounded above by $f$ asymptotically, $o(f(N))$ indicates $f$ dominates $T$, $\omega(f(N))$ indicates $T$ dominates $f$, and $\Theta(f(N))$ indicates $f$ bounds $T$ both above and below. For comparison, we also give the algorithmic complexity of the implementations of Madras and Sokal [17], and Kennedy [9]. In Sect. 3.5, we develop an argument for the complexity of our algorithm on $\mathbb{Z}^4$; this same argument leads to an estimate for the performance of the implementation of Madras and Sokal on $\mathbb{Z}^4$. We do not know the complexity of Kennedy's implementation for $\mathbb{Z}^4$ and $\mathbb{Z}^d$ with $d > 4$, but we suspect it is $O(N^q)$ with $0.74 < q < 1$, with possible logarithmic corrections.

Our implementation is also fast in practice: for simulations of walks of length $2^{20} \approx 10^6$ on $\mathbb{Z}^3$, our implementation is almost 400 times faster when compared with Kennedy's, and close to four thousand times faster when compared with that of Madras and Sokal. We have measured $T(N)$ for each implementation over a wide range of $N$ on $\mathbb{Z}^2$, $\mathbb{Z}^3$, and $\mathbb{Z}^4$, and report these results in Sect. 6.

## 1.3 Outline of Paper

In Sect. 2, we give a detailed description of the SAW-tree data structure and associated methods which are required for implementing the pivot algorithm.

In Sect. 3 we present heuristic arguments that $T(N)$ for self-avoiding walks on $\mathbb{Z}^2$ and $\mathbb{Z}^3$ is $O(1)$, and numerical evidence which shows that for walks of up to $N = 2^{28} - 1 \approx 2.7 \times 10^8$ steps $T(N)$ is $o(\log N)$ for $\mathbb{Z}^2$ and $O(\log N)$ for $\mathbb{Z}^3$. We also discuss the behavior of our implementation for higher dimensions.

In Sect. 4 we discuss initialization of the Markov chain, including details of how many data points are discarded. We also explain why it is highly desirable to have a procedure such as **Pseudo_dimerize** for initialization (pseudo-code in Sect. 2.7) when studying very long walks, and show that the expected running time of **Pseudo_dimerize** is $\Theta(N)$.

In Sect. 5 we discuss the autocorrelation function for the pivot algorithm, and show that the batch method for estimating confidence intervals is accurate, provided the batch size is large enough. This confirms the accuracy of the confidence intervals for our data published in [4].

Finally, in Sect. 6 we compare the performance of our implementation with previous implementations of the pivot algorithm [9, 17]. We show that the SAW-tree implementation is not only dramatically faster for long walks, it is also faster than the other implementations for walks with as few as 63 steps.

## 2 Implementation Details

Self-avoiding walks (SAWs) are represented as binary trees (see e.g. [21]) via a recursive definition; we describe here the SAW-tree data structure and associated methods using pseudo-code.

These methods can be extended to include translations, splitting of walks, joining of walks, and testing for intersection with surfaces. Indeed, for SAW-like models (those with short range interactions), it should be possible to implement a wide variety of global moves and tests for SAWs of $N$ steps in time $O(\log N)$ or better. It is also possible to parallelize code by, for example, performing intersection testing for a variety of proposed pivot moves simultaneously. Parallelization of the basic operations is also possible, but would be considerably more difficult to implement.

In this section we give precise pseudo-code definitions of the data structure and algorithms. For reference, R-trees [7] and bounding volume hierarchies (see e.g. [10]) are data structures which arise in the field of computational geometry which are related to the SAW-tree.

### 2.1 The Self-avoiding Walk

For self-avoiding walks, the self-avoidance condition is enforced on sites rather than bonds, and this means that the SAW-tree is naturally defined in terms of sites. This representation also has the advantage that the basic objects, sites, have physical significance as they correspond to the monomers in a polymer. The only consequences of this choice are notational: a SAW-tree of $n$ sites has $n - 1$ steps. We adopt this notation for the remainder of this section. When discussing the complexity of various algorithms we will still use $N$ rather than $n$ in order to be consistent with the companion article and other sections of the present work.

An $n$-site SAW on $\mathbb{Z}^d$ is a mapping $\omega : \{0, 1, \ldots, n-1\} \to \mathbb{Z}^d$ with $|\omega(i+1) - \omega(i)| = 1$ for each $i$ ($|x|$ denotes the Euclidean norm of $x$), and with $\omega(i) \neq \omega(j)$ for all $i \neq j$. SAWs may be either rooted or unrooted; our convention is that the SAWs are rooted at the site which is at $\hat{\mathbf{x}}_1$ (unit vector in the first coordinate direction), i.e. $\omega(0) = \hat{\mathbf{x}}_1$. This convention simplifies some of the algebra involved in merging sub-walks, and is represented visually, e.g. in Fig. 1, by indicating a dashed bond from the origin to the first site of the walk.

**Fig. 1** A self-avoiding walk of 5 sites, which we will refer to as $\omega_a$
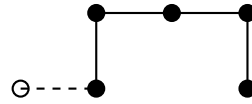


We denote the group of symmetries of $\mathbb{Z}^d$ as $G_d$, which corresponds to the dihedral group for $d = 2$, and the octahedral group for $d = 3$. This group acts on coordinates by permuting any of the $d$ coordinate directions ($d!$ choices), and independently choosing the orientation of each of these coordinates ($2^d$ choices); thus $G_d$ has $2^d d!$ elements. The group of lattice symmetries for $\mathbb{Z}^3$ therefore has 48 elements, and we use all of them except the identity as potential pivot operations; other choices are possible. We can represent the symmetry group elements as $d \times d$ orthogonal matrices, and the symmetry group elements act on the coordinates written as column vectors.

We also define the (non-unique) pivot sequence representation of a self-avoiding random walk on $\mathbb{Z}^d$ as a mapping from the integers to $G_d$, $q_\omega : \{0, 1, \ldots, n-1\} \rightarrow G_d$. The sequence elements $q(i)$ represent changes in the symmetry operator from site $i - 1$ to site $i$, while $q_{\text{abs}}(i)$ represent absolute symmetry operations, i.e. relative to the first site of the walk. We can relate this to the previous definition of a self-avoiding walk in terms of sites via the recurrence relations

$$q_{\text{abs}}(i) = q_{\text{abs}}(i-1)q(i), \tag{1}$$

$$\omega(i) = \omega(i-1) + q_{\text{abs}}(i)\hat{\mathbf{x}}_1, \tag{2}$$

with $1 \le i \le n - 1$, and initial conditions $q(0) = I$, $q_{\text{abs}}(0) = I$, and $\omega(0) = \hat{\mathbf{x}}_1$.

As noted by Madras and Sokal (footnote 10, p. 132 in [17]), for the pivot sequence representation it is possible to perform a pivot of the walk in time $O(1)$ by choosing a site $i$ uniformly at random, and multiplying $q(i)$ by a (random) symmetry group element. However, the pivot sequence representation does no better than the hash table implementation of Madras and Sokal if we wish to determine if this change results in a self-intersection, or if we wish to calculate global observables such as $R_{\text{e}}^2$ for the updated walk.

Forgetting for the moment the self-avoidance condition, and using the fact that $G_d$ has $2^d d!$ elements, we see that for random walks of $n$ sites there are $(2^d d!)^{n-1}$ possible pivot sequences, while there are only $(2d)^{n-1}$ random walks. This suggests that each random walk is represented by $(2^{d-1}(d-1)!)^{n-1}$ pivot sequences. This can be derived directly by noting that given a pivot sequence $q(1), q(2), q(3), \ldots, q(k), q(k+1), \ldots, q(n-1)$, we can insert a pivot $q$ which preserves the vector $\hat{\mathbf{x}}_1$, between two elements $q(k)$ and $q(k+1)$ as follows

$$q(0), q(1), q(2), q(3), \ldots, q(k)q, q^{-1}q(k+1), \ldots, q(n-1),$$

without altering the walk. The number of symmetry group elements which preserve $\hat{\mathbf{x}}_1$ is $2^d d!/(2d) = 2^{d-1}(d-1)!$, and there are $n - 1$ locations where these symmetry group elements can be inserted, leading to $(2^{d-1}(d-1)!)^{n-1}$ equivalent pivot representations for a random walk of $n$ sites. For $d = 2$, given $\omega(i)$ the recurrence relations in (1) and (2) only fix one of the two non-zero elements in $q(i)$, leaving the choice of sign for the other non-zero element free. For our example walk $\omega_a$ we have

$$\omega_a = ((1, 0), (1, 1), (2, 1), (3, 1), (3, 0)). \tag{3}$$

We give three of the 16 equivalent choices for the pivot representation of $\omega_a$, the first involving only proper rotations, the second with improper rotations for $q(i)$ with $1 \le i \le 4$, and

**Fig. 2** Examples of the merge
operation on SAWs: the open
circles (empty sites) of the
right-hand sub-walks are fixed to
the ends of the corresponding
left-hand sub-walks, and the
symmetries are then applied to
the right-hand sub-walks. At the
*top*, an identity symmetry
operation is applied, while at the
*bottom* the symmetry operation is
a 180° rotation
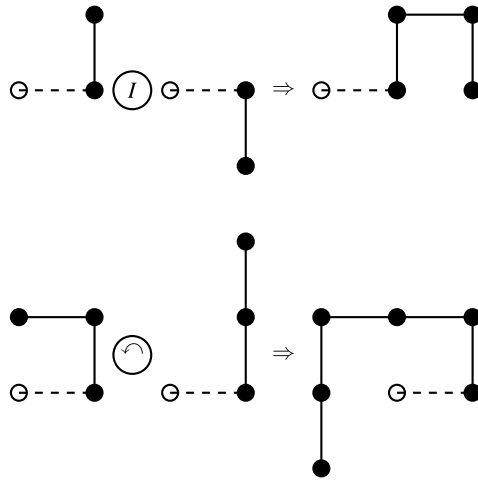
the third with proper and improper rotations alternating:

$$q_{\omega_a}^{(1)} = \left( \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}, \begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix}, \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix}, \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}, \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix} \right), \tag{4}$$

$$q_{\omega_a}^{(2)} = \left( \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}, \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}, \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}, \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}, \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \right), \tag{5}$$

$$q_{\omega_a}^{(3)} = \left( \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}, \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}, \begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix}, \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}, \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix} \right). \tag{6}$$

The non-uniqueness of the pivot representation for SAWs is due to the fact that the monomers (occupied sites) are invariant under the symmetry group $G_d$, i.e. it is not possible to distinguish the different orientations of a single site. The non-uniqueness is of no practical concern, but perhaps hints that it may be possible to derive a more succinct and elegant representation of walks than the mapping to $G_d$ defined here.

The merge operation is the fundamental operation on SAWs which allows for the binary tree data structure we call the SAW-tree. This is related to the concatenation operation defined, for example, in Sect. 1.2 of [16]; for concatenation the number of bonds is conserved, whereas for the merge operation the number of sites is preserved. Merging two SAWs with $n$ and $m$ sites respectively results in a SAW with $n + m$ sites. It is convenient to also include a pivot operation, $q$, when merging the walks, and the result of merging two walks $\omega_1 = (\omega_1(0), \omega_1(1), \ldots, \omega_1(n-1))$ and $\omega_2 = (\omega_2(0), \omega_2(1), \ldots, \omega_2(m-1))$ is

$$\text{merge}(\omega_1, q, \omega_2) = (\omega_1(0), \omega_1(1), \ldots, \omega_1(n-1), \omega_1(n-1) + q\omega_2(0),$$
$$\omega_1(n-1) + q\omega_2(1), \ldots, \omega_1(n-1) + q\omega_2(m-1)). \tag{7}$$

The merge operation is represented visually in Fig. 2. To merge two sub-walks, pin the open circle of the left-hand sub-walk to the origin, and then pin the open circle of the right-hand sub-walk to the tail end of the left-hand sub-walk. Finally, apply the symmetry $q$ to the right-hand sub-walk, using the second pin as the pivot point.

## 2.2 Definitions

Here we define various quantities which are necessary for implementing our data structure and for calculating observables such as the mean-square end-to-end distance, $R_e^2$.

We first define various quantities which will be used to calculate observables which measure the size of a walk:

$$\mathbf{X}_e(\omega) \equiv \omega(n-1) \quad \text{(vector);} \tag{8}$$

$$\mathbf{X}(\omega) \equiv \sum_{i=0}^{n-1} \omega(i) \quad \text{(vector);} \tag{9}$$

$$X_2(\omega) \equiv \sum_{i=0}^{n-1} \omega(i) \cdot \omega(i) \quad \text{(scalar).} \tag{10}$$

A bounding box of a walk is a convex shape which completely contains the walk. The obvious choice of shape for $\mathbb{Z}^d$ is the rectangular prism with faces formed from the coordinate planes $\mathbf{x}_i = \text{const}$, $1 \leq i \leq d$, with the constants chosen so that the faces of the prism touch the walk, i.e. the bounding box has minimum extent. Other choices are possible, e.g. other planes can be used such as $\mathbf{x}_i \pm \mathbf{x}_j = \text{const}$, $1 \leq i < j \leq d$, and have the advantage of matching the shape of the walk more closely, but at the expense of more computational overhead and memory consumption. With closer fitting bounding boxes, fewer intersection tests need to be performed to ascertain whether two walks intersect. However, in practice, the coordinate plane rectangular prism implementation was fastest on our computer hardware (by a narrow margin), and has the benefit that it is straightforward to implement. The choice of bounding box for continuum models is not as obvious; possibilities include spheres and oriented rectangular prisms.

We note that the choice of bounding box shape determines the maximum number of sites, $b$, a SAW can have so that it is guaranteed that its bounding box contains the sites of the SAW and no others. Suppose we are given two SAWs for which the bounding boxes overlap: if each of the walks has $b$ or fewer sites, we can be certain that the two walks intersect, while if at least one of the walks has more than $b$ sites, it may be that the walks do not intersect. The value of $b$ determines the cut-off for intersection testing for the function **Intersect** in Sect. 2.6. For $\mathbb{Z}^d$ with $d \geq 2$, the bounding box with faces formed from the coordinate planes leads to the maximum number of sites being two, as there are counter-examples with three sites (e.g. the bounding box of $\omega = ((0,0), (1,0), (1,1))$ also contains $(0,1)$). For the bounding box with the faces being the coordinate planes and $\mathbf{x}_i \pm \mathbf{x}_j = \text{const}$, the maximum number of sites is three (as the bounding box of $\omega = ((0,0), (1,0), (2,0), (2,1))$ also contains $(1,1)$). It is possible to push this one step further so that the maximum number of sites is four, but five is not possible as we can see that $\omega_a$ in Fig. 1 has five sites, and an unvisited site on its convex hull, which must also therefore be interior to any bounding box.

We write bounding boxes as a product of closed intervals, in the form $B(\omega) = \times[\inf \mathbf{x}_i : \mathbf{x} \in \omega, \sup \mathbf{x}_i : \mathbf{x} \in \omega]$, where the product is taken over $1 \leq i \leq d$. Consider a walk $\omega$, with bounding box $B$, which is split into left- and right-hand sub-walks, $\omega^l = (\omega(0), \ldots, \omega(k-1))$ and $\omega^r = (\omega(k), \ldots, \omega(n-1))$, with bounding boxes $B^l = \times[a_i, b_i]$, and $B^r = \times[c_i, d_i]$ respectively. We can then define the union operation on bounding boxes,

$$B = B^l \cup B^r$$

$$\equiv \times ([a_i, b_i] \cup [c_i, d_i])$$

$$= \times [\inf\{a_i, c_i\}, \sup\{b_i, d_i\}]. \tag{11}$$

The intersection operation is defined as

$$B^l \cap B^r = \times ([a_i, b_i] \cap [c_i, d_i])$$

$$= \times [\sup\{a_i, c_i\}, \inf\{b_i, d_i\}]. \tag{12}$$

There is no guarantee that $\sup\{a_i, c_i\} \leq \inf\{b_i, d_i\}$, and we adopt the convention that an interval $[e, f]$ is considered empty if $e > f$. If any interval is empty, then the corresponding bounding box is also empty as it contains no interior sites. A quantity associated with the bounding box which we will find useful is the sum of the dimensions of the bounding box, **Perim**. If $B = \times [a_i, b_i]$, then we define

$$\mathbf{Perim}(B) = \sum_{i=1}^{d} (b_i - a_i + 1). \tag{13}$$

For $\omega_a$ (in Fig. 1) we have the following values for the various parameters:

$$n = 5; \tag{14}$$

$$\mathbf{B}(\omega_a) = [1, 3] \times [0, 1]; \tag{15}$$

$$\mathbf{X}_e(\omega_a) = (3, 0); \tag{16}$$

$$\mathbf{X}(\omega_a) = (1, 0) + (1, 1) + (2, 1) + (3, 1) + (3, 0)$$

$$= (10, 3); \tag{17}$$

$$X_2(\omega_a) = (1, 0) \cdot (1, 0) + (1, 1) \cdot (1, 1) + (2, 1) \cdot (2, 1)$$

$$+ (3, 1) \cdot (3, 1) + (3, 0) \cdot (3, 0)$$

$$= 1 + 2 + 5 + 10 + 9$$

$$= 27. \tag{18}$$

The observables $R_x^{2k}$, with $x \in \{e, g, m\}$, $1 \leq k \leq 5$, may be straightforwardly calculated from $\mathbf{X}_e$, $\mathbf{X}$, and $X_2$. We give expressions for $R_x^2$ with $x \in \{e, g, m\}$, and note that higher Euclidean-invariant moments can be obtained via $R_x^{2k} = (R_x^2)^k$ ($2 \leq k \leq 5$) (these moments are calculated for $\mathbb{Z}^2$ in [3] and for $\mathbb{Z}^3$ in [4]). In addition we introduce another observable, $\mathcal{R}_m^2$, which measures the mean-square deviation of the walk from the endpoint $\omega(n-1)$.

$$R_e^2 = |\omega(n-1) - \omega(0)|^2$$

$$= (\mathbf{X}_e - \hat{\mathbf{x}}_1) \cdot (\mathbf{X}_e - \hat{\mathbf{x}}_1) \tag{19}$$

$$R_g^2 = \frac{1}{2n^2} \sum_{i,j=0}^{n-1} |\omega(i) - \omega(j)|^2$$

$$= \frac{1}{n} X_2 - \frac{1}{n^2} \mathbf{X} \cdot \mathbf{X} \tag{20}$$

$$R_m^2 = \frac{1}{2n} \sum_{i=0}^{n-1} \left[ |\omega(i) - \omega(0)|^2 + |\omega(i) - \omega(n-1)|^2 \right]$$

$$= \frac{1}{2} + \frac{1}{2}\mathbf{X}_e \cdot \mathbf{X}_e - \frac{1}{n}\hat{\mathbf{x}}_1 \cdot \mathbf{X} - \frac{1}{n}\mathbf{X}_e \cdot \mathbf{X} + \frac{1}{n}X_2 \tag{21}$$

$$\mathscr{R}_m^2 = \frac{1}{n}\sum_{i=0}^{n-1}|\omega(i) - \omega(n-1)|^2$$

$$= \mathbf{X}_e \cdot \mathbf{X}_e - \frac{2}{n}\mathbf{X}_e \cdot \mathbf{X} + \frac{1}{n}X_2 \tag{22}$$

In [4], we chose to calculate $\mathscr{R}_m^2$ rather than $R_m^2$, as it has a slightly simpler expression, and relied on the identity $\langle\mathscr{R}_m^2\rangle = \langle R_m^2\rangle$. Compared with $R_m^2$, $\mathscr{R}_m^2$ has larger variance but smaller integrated autocorrelation time (for the pivot algorithm). Before performing the computational experiment in [4], we believed that given the same number of pivot attempts the confidence intervals for $\langle\mathscr{R}_m^2\rangle$ and $\langle R_m^2\rangle$ would be comparable. We have since confirmed that working directly with $R_m^2$ results in a standard error which is of the order of 17% smaller for $d = 3$, an amount which is not negligible; in future experiments we will calculate $R_m^2$ directly.

## 2.3 Guide to the Interpretation of Pseudo-code

Here follow some comments to aid in the interpretation of the pseudo-code description of the SAW-tree data structure and associated algorithms.

– All calls are by value, following the C programming language convention. Data structures are passed to methods via pointers.
– Pointers: the walk $w$ is a data structure whose member variables can be accessed via pointers, e.g. the vector for the end-to-end distance for the walk $w$ is $w \to \mathbf{X}_e$. The left-hand sub-walk of $w$ is indicated by $w^l$, and the right-hand sub-walk by $w^r$. This notation is further extended by indicating $w^{ll}$ for the left-hand sub-walk of $w^l$, $w^{lr}$ for the right-hand sub-walk of $w^l$, etc.
– Suggestive notation for member variables used to improve readability; all quantities, such as "$\mathbf{X}_e$" (the end-to-end vector) must correspond to a particular walk $w$. e.g. $\mathbf{X}_e \equiv w \to \mathbf{X}_e$, $\mathbf{X}_e^l \equiv w^l \to \mathbf{X}_e$ (i.e., superscript $l$ indicates that $\mathbf{X}_e^l$ is the end-to-end vector for the left sub-walk), $q^l \equiv w^l \to q$, $n^l \equiv w^l \to n$.
– Variables with subscript $t$ are used for temporary storage only.
– Comments are enclosed between the symbols /* and */ following the C convention.
– Boolean negation is indicated via the symbol "!", e.g. ! TRUE = FALSE.

## 2.4 SAW-Tree Data Structure

The key insight which has enabled a dramatic improvement in the implementation of the pivot algorithm is the recognition that *sequences* of sites and pivots can be replaced by *binary trees*. The $n$ leaves of the tree are individual sites of the walk, and thus encode no information, while each of the $n-1$ (internal) nodes of the tree contain aggregate information about all sites which are below them in the tree. We call this data structure the SAW-tree, which may be defined recursively: a SAW-tree of $n$ sites either has $n = 1$ and is a leaf, or has a left child SAW-tree with $0 < k < n$ sites, and a right child SAW-tree with the remaining $n - k$ sites.

Our implementation of the SAW-tree node is introduced in Table 2. A SAW-tree consists of one or more SAW-tree nodes in a binary tree structure; the pointers $w^l$ and $w^r$ allow traversal from the root of the tree to the leaves, while $w^p$ allows for traversal from the leaves

**Table 2** Definition of the SAW-tree node data structure; see Sect. 2.2 for definitions of each of the member variables. The node contains variables which are required for traversal of the SAW-tree ($w^p$, $w^l$, $w^r$), intersection testing ($n$, $q$, $\mathbf{X}_e$, $\mathbf{B}$), and calculation of observables $R_x^2$ with $x \in \{e, g, m\}$ ($n$, $\mathbf{X}_e$, $\mathbf{X}$, $X_2$). If the only observable to be calculated is $R_e^2$, then $\mathbf{X}$ and $X_2$ can be omitted

| SAW-tree node data structure | | |
|---|---|---|
| Type | Name | Description |
| integer | $n$ | Number of sites |
| SAW-tree ptr | $w^p$ | Parent |
| SAW-tree ptr | $w^l$ | Left-hand sub-walk |
| SAW-tree ptr | $w^r$ | Right-hand sub-walk |
| matrix | $q$ | Symmetry group element |
| vector | $\mathbf{X}_e$ | $\omega(n-1)$ |
| vector | $\mathbf{X}$ | $\mathbf{X} = \sum_{i=0}^{n-1} \omega(i)$ |
| integer | $X_2$ | $X_2 = \sum_{i=0}^{n-1} \omega(i) \cdot \omega(i)$ |
| bounding box | $\mathbf{B}$ | Convex region |

of the tree to the root. SAW-trees are created by merging other SAW-trees, with a symmetry operation acting on the right-hand walk. In particular, any *internal node* $w$ may be expressed in terms of its left child $w^l$, a symmetry operation $q \equiv w \to q$, and its right child $w^r$ via a merge operation:

$$w = \mathrm{merge}(w^l, q, w^r). \tag{23}$$

The leaves of the SAW-tree correspond to sites in a SAW, and are thus labeled from 0 to $n-1$. A binary tree with $n$ leaves has $n-1$ internal nodes, and we label these nodes from 1 to $n-1$, so that the symmetry $q(i)$ is to the left of $\omega(i), \omega(i+1), \ldots, \omega(n-1)$. The symmetry $q(0)$ is not part of the SAW-tree as it is applied to the whole walk, and thus cannot be used in a merge operation. For some applications it may be necessary to keep track of $q(0)$, e.g. when studying polymers in a confined region, but in [4] this was not necessary.

Assume that the end-to-end vectors, $\mathbf{X}_e$, and symmetry group elements, $q(i)$, for a SAW-tree and its left and right children are given. If we know the location of the anchor site of the parent node, $x_{\mathrm{abs}}$, along with the overall *absolute* symmetry group element $q_{\mathrm{abs}}$ being applied to the walk, we can then find the same information for the left and right children as follows:

$$
\begin{array}{rrcl}
\text{Left:} & x_{\mathrm{abs}} & \Leftarrow & x_{\mathrm{abs}} \\
 & q_{\mathrm{abs}} & \Leftarrow & q_{\mathrm{abs}} \\
\text{Right:} & x_{\mathrm{abs}} & \Leftarrow & x_{\mathrm{abs}} + q_{\mathrm{abs}} \mathbf{X}_e^l \\
 & q_{\mathrm{abs}} & \Leftarrow & q_{\mathrm{abs}} q^r
\end{array}
$$

Thus $\omega(i)$ can be determined for any site $i$ by iteratively performing this calculation while following the (unique) path from the root of the SAW-tree to the appropriate leaf. N.B.: $x_{\mathrm{abs}}$ must be updated before $q_{\mathrm{abs}}$.

We give explicit examples of SAW-trees in Appendix A. In Fig. 22, we give a SAW-tree representation of a SAW with $n$ sites which is precisely equivalent to the pivot sequence representation. We also give two equivalent representations of $\omega_a$ (shown in Fig. 1) in Figs. 23 and 24.

Conceptually we distinguish single-site walks (individual sites), which reside in the leaves of the tree, from multi-site walks.[1] In particular, the symmetry group element of a single site has no effect, and in the case where all monomers are identical then all single sites are identical.

If the SAW-tree structure remains fixed it is not possible to rotate part of the walk by updating a single symmetry group element, in contrast to the pivot sequence representation. This is because when we change a symmetry group element in a given node, it only alters the position of sites which are in the right child of the node. To rotate the part of the walk with sites labeled $i + 1$ and greater, we choose the $i$th internal node of the SAW-tree from the left. We then need to alter the symmetry group element of this node, and also all nodes which are above and to the right of it in the SAW-tree. If we select a random node then it will likely be near the leaves of the tree, and assuming that the SAW-tree is balanced this means that on average $O(\log N)$ symmetry group elements will need to be altered.

However, we note that the root node at the top of the tree has no parents, and therefore only one symmetry group element needs to be altered to rotate the right-hand part of the walk in this case. By utilizing tree-rotation operations, which alter the structure of the tree while preserving node ordering, it is possible to move the $i$th node to the root of the SAW-tree. Once this has been done, it *is* then possible to implement a rotation of part of the walk by updating a single symmetry group element. On average, $O(\log N)$ of these tree-rotation operations are required.

Binary trees are a standard data structure in computer science. By requiring trees to be balanced, i.e. so that the height of a tree with $N$ nodes is bounded by a constant times $\log N$, optimal bounds can be derived for operations such as insertion and deletion of nodes from the tree. We refer the interested reader to Sedgewick [21] for various implementations of balanced trees, such as red-black balanced trees. We have the advantage that our SAW-tree is, essentially, static, which means that we can make it perfectly balanced without the additional overhead of maintaining a balanced tree.

## 2.5 Primitive Operations

Included in this subsection are the primitive operations, which would generally not be called from the main program.

Left and right tree-rotations are modified versions of standard tree operations; for binary trees, only ordering needs to be preserved, while for SAW-trees the sequence of sites needs to be preserved, which means that symmetry group elements and other variables need to be modified.

**Procedure:**
**Merge**(*SAW-tree ptr $w^l$*, *SAW-tree ptr $w^r$*, *SAW-tree ptr $w$*)
/* Two SAWs are joined together, head to tail. Merge $w^l$ and $w^r$ into $w$, i.e.
  $w \Leftarrow \text{merge}(w^l, q, w^r)$. Pointers are not altered. See (7) and Fig. 2.            */
$n \quad \Leftarrow \quad n^l + n^r$
$\mathbf{B} \quad \Leftarrow \quad \mathbf{B}^l \cup (\mathbf{X}^l_e + q\mathbf{B}^r)$

---

[1]Technical note: as the leaves are all identical, in practice we use a sentinel node for the leaves, saving on memory usage.

**Fig. 3** Left tree-rotation applied
to a SAW-tree, where A, B, and C
are arbitrary SAW-trees. LHS and
RHS are different representations
of the same self-avoiding walk
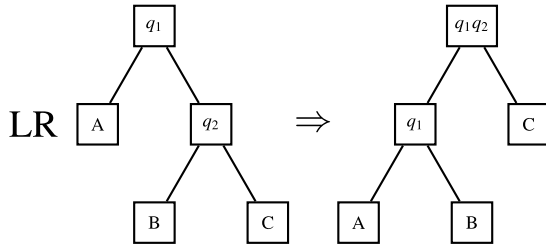


**Fig. 4** Right tree-rotation
applied to a SAW-tree, where A,
B, and C are arbitrary SAW-trees.
LHS and RHS are different
representations of the same
self-avoiding walk
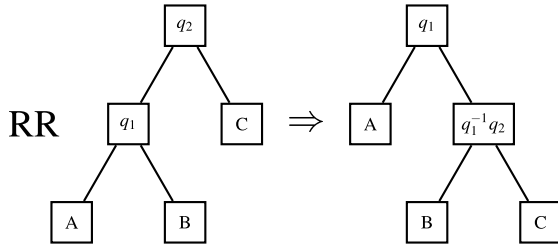


$$\begin{aligned}
\mathbf{X}_e &\Leftarrow \mathbf{X}_e^l + q\mathbf{X}_e^r \\
\mathbf{X} &\Leftarrow \mathbf{X}^l + q\mathbf{X}^r + n^r\mathbf{X}_e^l \\
X_2 &\Leftarrow X_2^l + X_2^r + 2\mathbf{X}_e^l \cdot (q\mathbf{X}^r) + n^r\mathbf{X}_e^l \cdot \mathbf{X}_e^l
\end{aligned}$$

End    /* **Merge**                                                                                  */

**Procedure:**

**LR**(*SAW-tree ptr w*)

   /*  Left tree-rotation applied to $w$. Update of pointers to parents not shown, and
      note that only one merge operation is necessary. The pseudocode is faithful to
      the implementation used in the present work and [4], but the definition of this
      operation is likely to change in future implementations. In Fig. 3, $w$ refers to
      the node with symmetry $q_1$ on the LHS, and $q_1q_2$ on the RHS. In future, we
      will adopt the convention that $w$ always refers to the same node with respect to
      left-right ordering. By this convention, $w$ would refer to node with symmetry
      $q_1$ on the RHS.                                                                              */

$$\begin{aligned}
w_t &\Leftarrow w^r \\
w^r &\Leftarrow w_t \to w^r \\
w_t \to w^r &\Leftarrow w_t \to w^l \\
w_t \to w^l &\Leftarrow w^l \\
w^l &\Leftarrow w_t \\
q_t &\Leftarrow q \\
q &\Leftarrow q_t q^l \\
q^l &\Leftarrow q_t
\end{aligned}$$

   **Merge**($w^{ll}$,$w^{lr}$,$w^l$)

End    /* **LR**                                                                                     */

**Procedure:**

**RR**(*SAW-tree ptr w*)

/* Right tree-rotation applied to $w$. Update of pointers to parents not shown. Note that only one merge operation is necessary. The pseudocode is faithful to the implementation used in the present work and [4], but the definition of this operation is likely to change in future implementations. In Fig. 4 $w$ refers to the node with symmetry $q_2$ on the LHS, and $q_1$ on the RHS. In future, we will adopt the convention that $w$ always refers to the same node with respect to left-right ordering. By this convention, $w$ would refer to node with symmetry $q_1^{-1}q_2$ on the RHS.                                                                                       */

$$
\begin{aligned}
w_t &\Leftarrow w^l \\
w^l &\Leftarrow w_t \rightarrow w^l \\
w_t \rightarrow w^l &\Leftarrow w_t \rightarrow w^r \\
w_t \rightarrow w^r &\Leftarrow w^r \\
w^r &\Leftarrow w_t \\
q_t &\Leftarrow q \\
q &\Leftarrow q^r \\
q^r &\Leftarrow q^{-1}q_t
\end{aligned}
$$

    **Merge**($w^{rl}, w^{rr}, w^r$)

End   /* **RR**                                                                                                                                  */

**Function:**
**Find_node**(*integer $n_t$*, *SAW-tree ptr $w$*)
/* Returns a pointer to the $n_t^{\text{th}}$ node from the left in the SAW-tree $w$. This may either be implemented as a numerical function, if the address of the correct node can be easily determined (such as if the nodes of $w$ are arranged in memory in pre-order fashion), or returned from a look-up array. This look-up array is static, and so will not need to be updated after it is created. Note: we require this function to take time $O(1)$, and thus it is inappropriate to use a binary search implementation which may take time $O(\log N)$. This function is required by **Attempt_pivot_fast**.    */

## 2.6  User Level Operations

Included in this subsection are user level operations which would typically be called from the main program.

**Procedure:**
**Generate_SAW-tree**(*integer $n$*)
/* Allocates memory for the SAW-tree, and creates the initial arrangement of the tree in memory. We use a strictly balanced tree layout, which guarantees $O(\log N)$ behavior for basic operations. We tested pre-order and van Emde Boas layouts; given that performance for large $N$ was memory bound, we were surprised to find that the pre-order layout was fastest, although we will experiment more with this in the future (the van Emde Boas tree layout [24] is an example of a cache-oblivious data structure [5, 12]).                                                                              */

**Function:**
**Random_integer_uniform**(*integer $a$*, *integer $b$*)
/* Returns an integer in the interval $[a, b)$ selected uniformly at random.                      */

**Function:**
**Random_integer_log**(*integer $a$*, *integer $b$*)

**Fig. 5** Shuffle up operation applied to node with symmetry $q_3$ in a SAW-tree, via a sequence of left and right tree-rotations, with A, B, C, and D arbitrary SAW-trees. LHS and RHS are different representations of the same self-avoiding walk
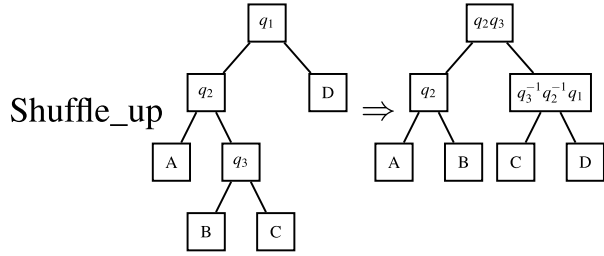


/* Returns an integer $i$ selected in the interval $[a, b)$ with probability

$$P(i) = \frac{\log(1 + \frac{1}{i-a+1})}{\log(b - a + 1)}. \tag{24}$$

This probability distribution has the property that if we set $a = 1$ for convenience and choose a length scale $1 \leq x < 2x \leq b$, then

$$P(i \in [x, 2x)) = \frac{\log(2x) - \log x}{\log b}$$

$$= \frac{\log 2}{\log b}, \tag{25}$$

i.e. the probability of $i$ lying in the semi-open interval $[x, 2x)$ is independent of $x$.    */

**Function:**
**Random_symmetry**()
/* Symmetry selected uniformly at random, excluding identity. Other choices are possible.                                                                                             */

**Procedure:**
**Accumulate_statistics**(*SAW-tree w*, *Boolean* success)
/* Accumulate statistics for Euclidean-invariant moments $R_x^{2k}$ with $x \in \{e, g, m\}$, and $1 \leq k \leq 5$. When the pivot attempt is unsuccessful there is no need to recalculate observables; in fact, by using a counter, updates to storage variables only need to be made when success = TRUE, i.e. when the last pivot attempt was successful.                                                                                           */

**Procedure:**
**Shuffle_up**(*integer* $n_0$, *SAW-tree ptr* w)
  /* Brings node $n_0$ to the root of the SAW-tree, via a series of tree-rotation operations. See Fig. 5.                                                                                */
  **if** $n_0 = n^l$ **then**
    /* Do nothing, node already at root.                                                  */
  **else if** $n_0 < n^l$ **then**
    **Shuffle_up**($n_0, w^l$)
    **RR**(w)
  **else if** $n_0 > n^l$ **then**
    **Shuffle_up**($n_0 - n^l, w^r$)
    **LR**(w)

  **end if**
  Return
**End**   /\* **Shuffle_up**                   \*/


**Procedure:**
**Shuffle_down**(*SAW-tree ptr* w)
  /\* Restores node at root to its correct place in the balanced SAW-tree, via a series
    of tree-rotation operations. Note that this operation is only necessary if one
    wishes that the SAW-tree remains balanced, and must be paired with **Shuffle-**
    **_up**.                         \*/
  $n_t \quad \Leftarrow \quad \lfloor (n+1)/2 \rfloor$
  **if** $n_t = n^l$ **then**
    /\* Do nothing, node is in correct place.            \*/
  **else if** $n_t < n^l$ **then**
    **RR**(*w*)
    **Shuffle_down**($w^r$)
  **else if** $n_t > n^l$ **then**
    **LR**(*w*)
    **Shuffle_down**($w^l$)
  **end if**
  Return
**End**   /\* **Shuffle_down**                  \*/


  The function **Intersect** is at the heart of our implementation of the pivot algorithm; we
recommend the reader consult the start of Sect. 3.2 for an explanation of how **Intersect**
works.

  **Function:**
**Intersect** (*vector* $x_{\text{abs}}^l$, *matrix* $q_{\text{abs}}^l$, *SAW-tree ptr* $w^l$,
     *vector* $x_{\text{abs}}^r$, *matrix* $q_{\text{abs}}^r$, *SAW-tree ptr* $w^r$)
  /\* Returns TRUE if $w^l$ and $w^r$ intersect, FALSE otherwise. $x_{\text{abs}}^l$ and $x_{\text{abs}}^r$ are the
    *absolute* positions of the anchor points of the walks $w^l$ and $w^r$, and $q_{\text{abs}}^l$ and
    $q_{\text{abs}}^r$ are overall symmetry group elements.            \*/
  /\* First, calculate the absolute positions of the bounding boxes. If they do not
    intersect, then $w^l$ and $w^r$ cannot intersect.           \*/
  $B_t^l \quad \Leftarrow \quad x^l + q^l B^l$
  $B_t^r \quad \Leftarrow \quad x^r + q^r B^r$
  **if** $B_t^l \cap B_t^r = \emptyset$ **then**
    Return FALSE
  **end if**
  **if** $n^l \le 2$ **and** $n^r \le 2$ **then**
    /\* The bounding boxes of $w^l$ and $w^r$ intersect, and thus $w^l$ and $w^r$ must in-
     tersect as each walk has two or fewer sites. The cut-off is dependent on the
     shape of the bounding box, and is the maximum number of sites a SAW can
     have which guarantee that the bounding box contains only the sites of the
     SAW and no others. For $\mathbb{Z}^d$ the value of the cut-off for the rectangular prism
     is two as given here; see Sect. 2.2 for further discussion.      \*/
    Return TRUE
  **end if**

**if** $n^l \geq n^r$ **then**
    /\* Split the left SAW-tree; compare the SAW-trees which are closest together
       on the chain first, as they are the most likely to intersect.     \*/
    **if Intersect**$(x_{abs}^l + q_{abs}^l \mathbf{X}_e^{ll}, q_{abs}^l q^l, w^{lr}, x_{abs}^r, q_{abs}^r, w^r)$ **then**
       Return TRUE
    **end if**
    Return **Intersect**$(x_{abs}^l, q_{abs}^l, w^{ll}, x_{abs}^r, q_{abs}^r, w^r)$
**else**
    /\* Split the right SAW-tree; compare the SAW-trees which are closest together
       on the chain first, as they are the most likely to intersect.     \*/
    **if Intersect**$(x_{abs}^l, q_{abs}^l, w^l, x_{abs}^r, q_{abs}^r, w^{rl})$ **then**
       Return TRUE
    **end if**
    Return **Intersect**$(x_{abs}^l, q_{abs}^l, w^l, x_{abs}^r + q_{abs}^r \mathbf{X}_e^{rl}, q_{abs}^r q^r, w^{rr})$
**end if**
End   /\* **Intersect**     \*/

**Shuffle_intersect** is akin to the procedure developed by Madras and Sokal [17] for intersection testing: by building new walks incrementally moving outwards from the pivot site they achieved a speed-up from $O(N)$ to $O(N^{1-p})$ for unsuccessful pivot attempts. In Sect. 3 we present a heuristic argument that the speed-up achieved for the SAW-tree implementation is from $O(\log N)$ to $O(1)$.

**Function:**
**Shuffle_intersect**(*SAW-tree ptr* $w$, *symmetry* $q_0$, *integer* was_left_child,
                  *integer* is_left_child)
    /\* Hybrid algorithm which combines elements of **Shuffle_up** and **Intersect**. $w$
       is the pivot node, and $q_0$ is the pivot operation which acts on the part of the
       walk to the right of $w$ (both ancestors and descendants). was_left_child and
       is_left_child are integer flags specifying the local tree structure. Function returns true if the left- and right-hand walks intersect, FALSE otherwise. The
       pseudo-code here reproduces the method, but some small-scale optimizations
       have been omitted for the sake of clarity. In particular, some computations are
       performed more often than strictly necessary, e.g. rotations of bounding boxes.
       For $\mathbb{Z}^3$, this results in the implementation here running approximately 40%
       slower than the optimized implementation. By using temporary variables to
       store the modified tree nodes, we guarantee that the original walk is left unmodified. Thus it is safe to execute concurrent versions of this function on the
       same SAW-tree in parallel.     \*/
    /\* Check if the left- and right- children of $w$ intersect.     \*/
**if** was_left_child $= 1$ **then**
    /\* We have already verified that $w^l$ and $w^{rl}$ do not intersect; check if $w^l$ and
       $w^{rr}$ intersect.     \*/
    **if Intersect**$(0, I, w^l, \mathbf{X}_e^l + qq_0\mathbf{X}_e^{rl}, qq_0q^r, w^{rr})$ **then**
       Return TRUE
    **end if**
**else if** was_left_child $= 0$ **then**
    /\* We have already verified that $w^{lr}$ and $w^r$ do not intersect; check if $w^{ll}$ and
       $w^r$ intersect.     \*/
    **if Intersect**$(0, I, w^{ll}, \mathbf{X}_e^l, qq_0, w^r)$ **then**

      Return TRUE
   **end if**
  **else if** was_left_child $= -1$ **then**
    /* First call of this function, and so we have no prior information; check if $w^l$
     and $w^r$ intersect.                */
    **if Intersect**$(0, I, w^l, \mathbf{X}_e^l, qq_0, w^r)$ **then**
     Return TRUE
    **end if**
  **end if**
  /* Check to see if we have reached the top of the tree.          */
  **if** $w^p =$ NULL **then**
   Return FALSE
  **end if**
  /* Not at top of tree, and so we need to perform a left or right rotation. First we
   need to determine if $w^p$ is a left or right child.         */
  **if** $w^{pp} =$ NULL **then**
   /* $w^p$ is the root of the tree so is_left_child_new will not be needed.   */
  **else if** $w^{ppl} = w^p$ **then**
   is_left_child_new   $\Leftarrow$   TRUE
  **else**
   is_left_child_new   $\Leftarrow$   FALSE
  **end if**
  /* Create a temporary SAW-tree node $w_t$ which is a copy of $w^p$ so we do not alter
   the original walk.               */
  $w_t$   $\Leftarrow$  $w^p$
  **if** is_left_child **then**
   /* right tree-rotation needed.             */
   **RR**$(w_t)$
  **else**
   /* left tree-rotation needed.             */
   **LR**$(w_t)$
  **end if**
  Return **Shuffle_intersect**$(w_t, q_0,$ is_left_child, is_left_child_new$)$
End   /* **Shuffle_intersect**              */

## 2.7 High Level Functions

Here follows a pseudo-code description of three routines which provide high level function-
ality for typical usage of the pivot algorithm.

   We define two versions of the function **Attempt_pivot**, one which is conceptually simple,
while in Sect. 3 we argue that the other version is asymptotically faster for $\mathbb{Z}^2$ and $\mathbb{Z}^3$.

  **Function:**
  **Attempt_pivot_simple**(*SAW-tree ptr w, integer $n_t$, symmetry $q_t$*)
   /* Attempt to apply a pivot to $w$ at site $n_t$ with symmetry $q_t$; if successful update
    the walk, otherwise leave walk unchanged. Return a boolean value to indicate
    whether the pivot operation successfully changed the walk.       */
  **Shuffle_up**$(n_t, w)$
  $q$   $\Leftarrow$  $qq_t$
  intersection    $\Leftarrow$   **Intersect**$(0, I, w^l, \mathbf{X}_e^l, q, w^r)$

  **if** intersection **then**
   /\* Reject pivot, restore original symmetry.                \*/
   $q \quad \Leftarrow \quad qq_t^{-1}$
  **end if**
  **Shuffle_down**$(w)$
  /\* Pivot was successful if sub-walks *did not* intersect.      \*/
  Return !intersection
End  /\* **Attempt_pivot_simple**                 \*/


**Function:**
**Attempt_pivot_fast**(*SAW-tree ptr w*, *integer* $n_t$, *symmetry* $q_t$)
  /\* Attempt to apply a pivot to $w$ at site $n_t$ with symmetry $q_t$; if successful update
    the walk, otherwise leave walk unchanged. Return a boolean value to indicate
    whether the pivot operation successfully changed the walk.     \*/
  $w_t \quad \Leftarrow \quad$ **Find_node**$(n_t)$
  **if** $w_t^{pl} =$ NULL **then**
   /\* Dummy argument, is_left_child will not be required.       \*/
   is_left_child  $\Leftarrow$  TRUE
  **else if** $w_t^{pl} = w_t$ **then**
   is_left_child  $\Leftarrow$  TRUE
  **else**
   is_left_child  $\Leftarrow$  FALSE
  **end if**
  intersection  $\Leftarrow$  **Shuffle_intersect**$(w_t, q_t, -1, \text{is\_left\_child})$
  **if** !intersection **then**
   /\* Accept pivot, perform symmetry operation.         \*/
   **Shuffle_up**$(n_t, w)$
   $q \quad \Leftarrow \quad qq_t$
   **Shuffle_down**$(w)$
  **end if**
  /\* Pivot was successful if sub-walks *did not* intersect.      \*/
  Return !intersection
End  /\* **Attempt_pivot_fast**                 \*/


**Procedure:**
**Pseudo_dimerize**(*SAW-tree w*)
  /\* Uses merge and pivot operations to generate an initial $N$-step SAW in time
    $\Theta(N)$ which is difficult to distinguish from a SAW sampled uniformly at ran-
    dom. See Sect. 4 for discussion about why it is highly desirable to have such a
    procedure, and also for analysis of the algorithmic complexity.    \*/
  /\* Generate initial left- and right-hand sub-walks.         \*/
  **Pseudo_dimerize**$(w^l)$
  **Pseudo_dimerize**$(w^r)$
  /\* Perform pivot operations on each of the sub-walks while attempting to merge
    them. When the two sub-walks are mutually avoiding, exit loop.    \*/
  **do**
   $n_t \quad \Leftarrow \quad n^l -$ **Random_integer_log**$(1, n^l)$
   $q_t \quad \Leftarrow \quad$ **Random_symmetry**()
   **Attempt_pivot**$(w^l, n_t, q_t)$

$n_t$   $\Leftarrow$   **Random_integer_log**$(1, n^r)$
$q_t$   $\Leftarrow$   **Random_symmetry**()
**Attempt_pivot**$(w^r, n_t, q_t)$
$q$   $\Leftarrow$   **Random_symmetry**()
**while Intersect**$(0, I, w^l, \mathbf{X}_e^l, q, w^r)$
**Merge**$(w^l, w^r, w)$
/* Perform additional pivots on the SAW, preferentially sampling close to the joint, in an attempt to reduce any sampling bias. We can attempt $o(n/\log n)$ pivots without changing the asymptotic behavior of the algorithm; we choose to attempt $n^{1/2}$ pivots. We do not believe that these additional pivots are strictly necessary.      */
**for** $i = 1$ to $n^{1/2}$ **do**
  $n_t$   $\Leftarrow$   $n^l -$ **Random_integer_log**$(1, n^l)$
  $q_t$   $\Leftarrow$   **Random_symmetry**()
  **Attempt_pivot**$(w, n_t, q_t)$
  /* Now the walks are combined, we have to shift node label by $n^l$.      */
  $n_t$   $\Leftarrow$   $n^l +$ **Random_integer_log**$(0, n^r)$
  $q_t$   $\Leftarrow$   **Random_symmetry**()
  **Attempt_pivot**$(w, n_t, q_t)$
**end for**
Return
End  /* **Pseudo_dimerize**      */

## 2.8 Main Program

Here follows a pseudo-code description of the main program to generate self-avoiding walks via the pivot algorithm.

**Procedure:**
**Main**()
/* Create SAW-tree structure.      */
$w$   $\Leftarrow$   **Generate_SAW-tree**$(n)$
/* Initialize SAW-tree.      */
**Pseudo_dimerize**$(w)$
/* Warm up the Markov chain by discarding $n_{\text{discard}}$ time steps.      */
**for** $i = 1$ to $n_{\text{discard}}$ **do**
  $n_t$   $\Leftarrow$   **Random_integer_uniform**$(1, n)$
  $q_t$   $\Leftarrow$   **Random_symmetry**()
  **Attempt_pivot**$(w, n_t, q_t)$
**end for**
/* Perform the computer experiment, collecting data on global observables at every time step.      */
**for** $i = 1$ to $n_{\text{sample}}$ **do**
  $n_t$   $\Leftarrow$   **Random_integer_uniform**$(1, n)$
  $q_t$   $\Leftarrow$   **Random_symmetry**()
  success $=$ **Attempt_pivot**$(w, n_t, q_t)$
  **Accumulate_statistics**$(w, \text{success})$
**end for**
End  /* **Main**      */

## 3 Algorithmic Complexity

Our goal in this section is to calculate the mean time per attempted pivot, $T(N)$, which can in turn be expressed in terms of the mean time for successful pivots, $T_S(N)$, and the mean time for unsuccessful pivots, $T_U(N)$. This is done as follows:

$$
\begin{aligned}
T(N) &= \Pr(\text{successful})T_S(N) + \Pr(\text{unsuccessful})T_U(N) \\
&= O(N^{-p})T_S(N) + O(1)T_U(N) \\
&= N^{-p}T_S(N) + T_U(N),
\end{aligned}
\tag{26}
$$

where the probability of a pivot being successful is $O(N^{-p})$ as mentioned in Sect. 1.1, and constant factors have been dropped. Note that this expression is valid for $\mathbb{Z}^2$ and $\mathbb{Z}^3$, while higher dimensions are discussed in Sect. 3.5.

The thread of argument is complicated by the fact that we have two implementations we wish to characterize, **Attempt_pivot_simple** and **Attempt_pivot_fast**. As the names indicate, we will find that **Attempt_pivot_fast** is asymptotically faster, and so $T(N)$ may be regarded as being the mean time per attempted pivot for the procedure **Attempt_pivot-_fast**.

We first introduce notation and do a minor calculation in Sect. 3.1. In Sect. 3.2 we give a detailed explanation of the behavior of the procedure **Intersect**, and develop a heuristic argument that $T_S(N)$ is $\Theta(\log N)$. In Sect. 3.3 we show that for **Attempt_pivot_simple** $T_U(N) = \Theta(\log N)$, and therefore $T(N) = \Theta(\log N)$, while in Sect. 3.4 we provide a heuristic argument that for **Attempt_pivot_fast** $T_U(N) = O(1)$, which leads to the prediction that $T(N) = O(1)$. In Sect. 3.5 we discuss the complexity of the algorithm for dimensions greater than three, and finally in Sect. 3.6 we examine the numerical evidence. There is modest numerical evidence supporting our conclusion that $T_S(N)$ is $\Theta(\log N)$ for $\mathbb{Z}^2$ and $\mathbb{Z}^3$, while the numerical evidence suggests $T_U(N) = o(\log N)$ for $\mathbb{Z}^2$ and $T_U(N) = I(\log N)$ for $\mathbb{Z}^3$. This is consistent with $T_U(N) = O(1)$, but far from conclusive.

A far simpler version of the argument for our prediction that $T(N) = O(1)$ for $\mathbb{Z}^2$ and $\mathbb{Z}^3$ is given in the companion article [4].

### 3.1 Background

We introduce the notation of the *level*, $l$, of a node, which is the number of generations that separate the node from the leaves of the tree. The leaves are at level zero, their parents are at level one, etc. This is non-standard notation, as usually the concept of depth is used to describe trees, with the root at depth zero, its children at depth one, etc. For a perfectly balanced tree with $2^k$ leaves (sites), the nodes at any fixed level represent sub-walks of the same number of sites independent of $k$, and for this reason we find the level notation to be useful.

We will specialize to the case where we have a SAW-tree, $W$, with $n = 2^k$ sites; this makes things somewhat simpler to deal with, but makes no difference otherwise to the algorithmic complexity. $W$ therefore has $2^k$ leaves, $2^k - 1$ internal nodes, and the level of the root is $k$.

The first step of the pivot algorithm is to select a pivot node uniformly at random, from the $n - 1$ internal nodes of $W$. The average level of this node is

$$
\mathbb{E}(l) = \frac{1}{2^k - 1} \sum_{j=1}^{k} j 2^{k-j}
$$

$$= \frac{2^{k+1} - 2 - k}{2^k - 1}$$

$$= 2 - O\left(\frac{k}{2^k}\right). \tag{27}$$

We note that the discussion below is sufficiently general that it applies to related models of polymers in the excluded volume limit, such as the Domb-Joyce model, interacting self-avoiding walks, self-avoiding walks on other lattices, or self-avoiding walk models in the continuum. SAW-tree implementations of these other models do not yet exist. The discussion is valid for dimension $d \geq 2$, but there will be some subtleties for $d \geq 4$ which will be explored in Sect. 3.5.

## 3.2 Successful Pivot

To estimate the expected running time of **Intersect**, we first need to get a clear understanding of what the algorithm does. **Intersect** recursively tests for the intersection of the bounding boxes of the left and right SAW-trees, and searches until either an intersection has been found, or until it has been shown that no intersection is possible (i.e. no bounding boxes intersect). The search proceeds by recursively splitting the longer of the left and right SAW-trees, and although the action of **Shuffle_up** on the SAW-trees complicates things somewhat, the comparison typically occurs between left and right SAW-trees which are of the same length, up to a factor of two. We note that the search method chosen is *depth-first search*, which minimizes memory usage, and also that the choice of splitting procedure is a key feature in the performance of the algorithm. By choosing to split the larger walk, and then first testing for intersection between the walks on the left and right hand sides which are closest together on the chain, this means that intersections are more likely to be found rapidly, allowing the search to terminate after only a small portion of the walk has been examined.

For convenience we will only consider testing for intersection between the left- and right-hand sides of $W$, i.e. we assume the pivot node is the root of the SAW-tree. This will simplify the discussion of the algorithm, but the conclusions drawn will be valid for any balanced SAW-tree, with any number of sites.

We will first examine the behavior of **Intersect** in the case that a pivot attempt is successful, when there is no intersection. Later we will consider what happens when an intersection is found.

When applied to $W$, **Intersect** compares sub-walks on the left-hand side at level $j$ with sub-walks on the right-hand side at levels $j$ and $j + 1$. When bounding boxes overlap, the search proceeds to the next lowest level in the tree, until there are no overlaps. The nodes that this search procedure visits within $W$ induces a binary tree $W'$ with internal nodes that are sub-walks whose bounding boxes have been found to overlap sub-walks on the other side, and whose leaves are sub-walks which do not overlap sub-walks on the other side. As $W'$ is a binary tree, the number of leaves is exactly one more than the number of internal nodes. The algorithmic complexity of **Intersect**, $T_S(N)$, is the number of intersection tests performed between sub-walks on the left- and right-hand sides of $W'$ (neglecting constant factors).

We seek to simplify the characterization of the counting problem, by first noting that the number of intersection tests which involve leaves of $W'$ is at most four times the number of intersection tests involving nodes of $W'$, as we only test for intersection between two sub-walks when both of their parents intersect. Thus $T_S(N)$ is asymptotically equal to the

**Fig. 6** $\omega_b$, a self-avoiding walk
of 16 sites
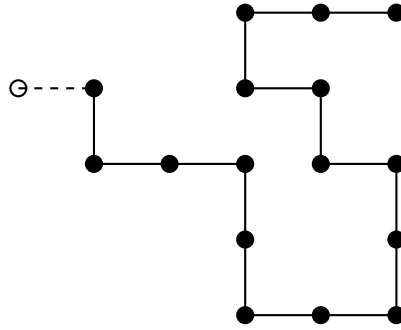


**Fig. 7** A SAW-tree
representation of $\omega_b$, with *arrows*
drawn between nodes on the *left-*
and *right-hand* sides which
approach each other
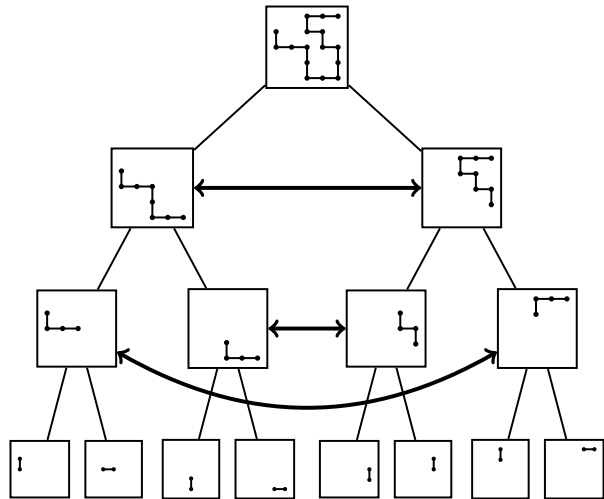


number of pairs of sub-walks with overlapping bounding boxes, where sub-walks on the left-hand side are at level $j$, and sub-walks on the right-hand side are at levels $j$ and $j + 1$. We then note that the number of pairs of sub-walks with overlapping bounding boxes with these rules are at most twice the number of pairs of sub-walks with overlapping bounding boxes on the same level. $T_S(N)$ is therefore given by the number of pairs of sub-walks on the left- and right-hand sides of $W$ with overlapping bounding boxes which are at the same level.

We introduce a new notation, describing the overlap between the bounding boxes of two sub-walks at level $j$ in $W$ as a "$j$-approach", while any overlap between boxes of sub-walks which are at the same level we denote as an "approach". Thus $T_S(N)$ is given by the expected number of *approaches* between the left- and right-hand sides of $W$.

We give an example of a SAW with 16 sites in Fig. 6, and show its corresponding SAW-tree in Fig. 7, with arrows drawn between sub-walks on the left- and right-hand sides which approach each other.

To this point our argument is exact, but we must now resort to heuristic arguments to count the expected number of approaches for $W$.

We first note that although there are typically $O(N)$ nearest neighbor contacts for a SAW of length $N$, the number of contacts between two halves of a SAW is typically $O(1)$, as shown via renormalization group [18] and Monte Carlo [2] methods. When we attempt to

pivot part of a SAW, it is guaranteed that each of the two sub-walks remain self-avoiding, and hence we only need to determine if the left- and right-hand sub-walks intersect. If the resulting walk is self-avoiding, then we expect, on average, that there will be a constant number of contacts between the two sub-walks.

We now consider the renormalization group transformation along the polymer chain described by Kremer et al. [11] in the context of a Monte Carlo renormalization group calculation (see also [6, 20]), which maps a polymer chain with hard sphere interactions to a new chain by rescaling the number of monomers ($n$), the separation between monomers ($L$), and the hard sphere diameter ($D$), so as to keep the mean-square end-to-end distance fixed. The interaction strength is defined as $\delta = D/L$. Kremer et al. concluded that this renormalization group transformation converged to a non-trivial (SAW) fixed point, with interaction strength $\delta^\star > 0$. In the neighborhood of the fixed point the renormalization transformations are

$$n' = \frac{n}{s}, \tag{28}$$

$$L' = As^\nu L, \quad \text{and} \tag{29}$$

$$D' = As^\nu D, \tag{30}$$

for some positive constant $A$.

We can translate this to our SAW-tree representation of a self-avoiding walk with $2^k$ sites if we regard the sub-walks of each node at a particular level in the tree as the monomers of a renormalized self-avoiding walk. We consider the transformation that occurs as we pass from level $j$ to level $j + 1$, with $1 \ll 2^j \ll 2^k$. Sub-walks at level $j$ have $n_j = 2^{k-j}$ monomers, and as each sub-walk contains $2^j$ sites the mean separation between monomers is therefore $L_j = A_1(2^j)^\nu$ for some constant $A_1$. The corresponding quantities for level $j+1$ are

$$n_{j+1} = 2^{k-j-1} = \frac{n_j}{2}, \tag{31}$$

$$L_{j+1} = A_1(2^{j+1})^\nu = 2^\nu L_j. \tag{32}$$

We note that the expectation of the perimeter of the bounding box of a self-avoiding walk with $n$ monomers is $A_2 n^\nu$ for some constant $A_2$ (see (13) in Sect. 2.2 for the definition of the perimeter). If we take the (effective) hard sphere diameter at this particular level $j$ to be some constant multiplied by the expected perimeter at level $j$, $P_j$, $D_j = A_3 P_j = A_3 A_2(2^j)^\nu$, and we pass to level $j + 1$, then in order to keep the interaction strength fixed we must have

$$\begin{aligned}
D_{j+1} &= 2^\nu D_j \\
&= 2^\nu A_3 P_j \\
&= 2^\nu A_3 A_2(2^j)^\nu \\
&= A_3 A_2(2^{j+1})^\nu \\
&= A_3 P_{j+1}, \tag{33}
\end{aligned}$$

i.e. the new effective hard sphere diameter is given by the same constant $A_3$ multiplied by $P_{j+1}$, and hence by induction $D_m = A_3 P_m \; \forall m \geq j$. Thus the nodes at level $j$ in a SAW-tree of total depth $k$, where $1 \ll 2^j \ll 2^k$, may be regarded as a self-avoiding walk with $2^{k-j}$ monomers, separated by links of length $L_j = A_1(2^j)^\nu$, with hard sphere monomers of

diameter $D_j = A_3 P_j$. This renormalization procedure becomes exact in the limit of large $j$ and $k$.

We now note that *approaches* for the SAW-tree correspond to *contacts* for the renormalized walks, where the minimum distance at which two monomers of the walk are said to be in contact is a fixed multiple of the hard sphere diameter. For fixed values of the interaction strength parameter and contact distance the number of contacts between the left- and right-hand sides of a walk is of $O(1)$, and we now know that as we proceed up the tree from the leaves to the root we are converging to a fixed value of the interaction strength. Thus the expected number of contacts between the two halves of the walk also converges to a constant.

Let the expected number of approaches at level $j$ for a walk with $n$ sites be $c_{n,j}$. The previous arguments imply that $\lim_{n\to\infty} c_{n,j} = c_j$ for some positive constant $c_j$, where the sequence is not necessarily monotonic. We then expect that the limit $\lim_{j\to\infty} c_j = c^\star$ exists, and that the $c_{n,j}$ are bounded, i.e. there exists a constant $c^\dagger$ such that $c_{n,j} < c^\dagger \, \forall n, j$. Thus

$$
\begin{aligned}
T_S(n) &= \sum_{j=0}^{\log_2 n} c_{n,j} \\
&< \sum_{j=0}^{\log_2 n} c^\dagger \\
&= c^\dagger \log_2 n,
\end{aligned}
\tag{34}
$$

and so $T_S(N) = O(\log N)$. In fact, we expect that $c_{n,j} \approx c^\star$ for sufficiently large $n$ and $j$, and so $T_S(N) = \Theta(\log N)$.

### 3.3 Algorithmic Complexity of **Attempt_pivot_simple**

On each iteration **Attempt_pivot_simple** executes the procedures **Shuffle_up** and **Shuffle_down** as well as the function **Intersect**.

The expected running time of both **Shuffle_up** and **Shuffle_down** is independent of whether the pivot attempt is successful or not. From Sect. 3.1, the expected level of a SAW-tree node selected uniformly at random is two. Thus the expected number of tree-rotations that **Shuffle_up** must perform in order to bring a pivot node to the top of a SAW-tree with $k$ levels, or **Shuffle_down** must perform to restore the node to its correct place in the tree, is $k - 2 + O(k/2^k)$. Given that $k = \log_2 n$, the algorithmic complexity of both **Shuffle_up** and **Shuffle_down** is therefore $O(\log N)$.

We already know **Intersect** takes time $\Theta(\log N)$ in the case that the pivot attempt is successful, which combined with the $\Theta(\log N)$ complexity of the shuffle operations gives

$$
T_S(N) = \Theta(\log N).
\tag{35}
$$

When the pivot attempt is unsuccessful and an intersection is found, **Intersect** terminates early and thus takes time $O(\log N)$ (we will argue in Sect. 3.4 that it is in fact $O(1)$, but this makes no difference to the argument here). Combined with the $\Theta(\log N)$ complexity of the shuffle operations this results in

$$
T_U(N) = \Theta(\log N).
\tag{36}
$$

For **Attempt_pivot_simple** we therefore have overall algorithmic complexity of

$$
\begin{aligned}
T(N) &= N^{-p} T_S(N) + T_U(N) \\
&= N^{-p} \Theta(\log N) + \Theta(\log N) \\
&= \Theta(\log N).
\end{aligned}
\tag{37}
$$

### 3.4 Algorithmic Complexity of **Attempt_pivot_fast**

On each iteration **Attempt_pivot_fast** executes the function **Shuffle_intersect**, while it only executes the procedures **Shuffle_up** and **Shuffle_down** when a pivot attempt is successful.

The key point to understand regarding the action of **Shuffle_intersect**, is that this function shuffles the pivot node up the SAW-tree only as far as necessary to find an intersection between the left- and right-hand sub-walks. In the case that the pivot attempt is successful, **Shuffle_intersect** must therefore shuffle the pivot node up to the top of the SAW-tree, but in the case where the pivot attempt is unsuccessful the pivot node is only shuffled up just far enough to be able to find this first intersection.

When the pivot attempt is successful, **Shuffle_intersect** must search the whole SAW-tree for intersections, and so is exactly equivalent to performing **Shuffle_up**, **Intersect**, and **Shuffle_down**. As shown in Sects. 3.2 and 3.3, all of these procedures take time $\Theta(\log N)$, and thus

$$
T_S(N) = \Theta(\log N).
\tag{38}
$$

The situation is considerably more complicated when the pivot attempt is unsuccessful and an intersection is found. We suppose that the pivot node is the $j^{\text{th}}$ (internal) node from the left, which has $j$ sites (leaves) to the left, and $n - j$ sites to the right. **Shuffle_intersect** will then search $W$ until it finds the intersection with $j - i_l$ being the label of the left-hand site and $j + i_r$ the label of the right-hand site such that $\sup(i_l, i_r)$ is minimum. Note: this description is approximate, as the SAW-tree structure complicates the issue of exactly which pair of intersecting sites is found, and the actual $i_l$ and $i_r$ which are found may be different from the optimal values. However, we are confident that for the average case this argument is correct up to a constant factor in the size of $i_l$ and $i_r$.

The sites $j - i_l$ and $j + i_r$ are exactly the pair of intersecting sites which would be discovered in the usual implementation of the pivot algorithm [17], where the new walk is incrementally built by moving outwards from the pivot site. Let us assume, without loss of generality, that $i_r > i_l$. We define $\Pr(i)$ as the probability that $i$ is the intersection that is found, i.e. the conditional probability that the sub-walk with sites $[j - i, j + i]$ has a self-intersection, but there are no intersections between sites in the open interval $(j - i, j + i)$. Madras and Sokal [17] convincingly argued that the expected value of $i_r$ is

$$
\begin{aligned}
\mathbb{E}(i_r) &= \sum_{i_r=1}^{2^k - j} i_r \Pr(i_r) \\
&= O(N^{1-p}),
\end{aligned}
\tag{39}
$$

with $p$ positive and close to zero for $\mathbb{Z}^2$ and $\mathbb{Z}^3$.

The interval $[j - i_l, j + i_r]$ corresponds to a self-avoiding loop, and it may be quite difficult to characterize the probability space of these configurations. We strongly suspect that

these configurations are sufficiently close to self-avoiding walks so that the mean number of approaches between the intervals $[j - i_l, j)$ and $[j, j + i_r]$ is $O(\log i_r)$. However, we do not have an argument to support this statement, and so we now make the *assumption* that the mean time for **Shuffle_intersect** to confirm that the sub-walk $(j - i_r, j + i_r)$ is non-intersecting is $O(\log i_r)$. To then find the intersection between $j - i_l$ and $j + i_r$ will take additional time of $O(\log i_r)$.

We now take $j$ to be a typical node, which places it near the middle of the walk with $j \approx 2^{k-1}$, and at level two in the SAW-tree. To determine the expected time to find the intersection between $i_l$ and $i_r$, we need to determine the probability distribution for $i_r$, given $j = 2^{k-1} = N/2$ (fixing $j$ makes the argument simpler, but does not change the result). Equation (39) suggests that for sufficiently large $i_r$, $\Pr(i_r) = O(i_r^{-1-p})$. We therefore neglect sub-dominant terms and take the leading order approximation $\Pr(i_r) \approx A/i_r^{1+p}$; this may be a very bad approximation for small $i_r$.

Using this approximation, we can now proceed to determine $T_U(N)$:

$$T_U(N) = \sum_{i_r=1}^{N/2} (\text{Time to find intersection}) \Pr(i_r)$$

$$= \sum_{i_r=1}^{N/2} \log i_r \frac{A}{i_r^{1+p}}. \tag{40}$$

We observe that for large $N$ $T_U(N)$ approaches a constant from below, i.e. $T_U(\infty) = O(1)$. To determine the rate of approach, we examine $T_U(\infty) - T_U(N)$:

$$T_U(\infty) - T_U(N) \approx \int_{N/2}^{\infty} \log i_r \frac{A}{i_r^{1+p}} di_r$$

$$= \frac{A}{p^2} \left(\frac{N}{2}\right)^{-p} \left(p \log \frac{N}{2} + 1\right) \tag{41}$$

Without detailed understanding of the exact form of $\Pr(i_r)$, it is impossible to estimate $T_U(\infty) - T_U(N)$ for any particular value of $N$. However, we can gain a rough idea of the rate of approach to the constant $T_U(\infty)$ by determining a natural length scale, $L$, from the solution of

$$\frac{T_U(\infty) - T_U(L)}{T_U(\infty) - T_U(1)} = \frac{1}{e}, \tag{42}$$

i.e. $L$ is the length at which our approximation for the deviation from the limiting value has decayed to $1/e$ of its initial value. For $\mathbb{Z}^2$, $p \approx 0.19$ which gives $L \approx 1.6 \times 10^5$, while for $\mathbb{Z}^3$, $p \approx 0.11$ and so $L \approx 5.8 \times 10^8$. We can clearly see that convergence to constant behavior may indeed be very slow, particularly for $\mathbb{Z}^3$.

For **Attempt_pivot_fast** we therefore have overall algorithmic complexity of

$$T(N) = N^{-p} T_S(N) + T_U(N)$$

$$= O\left(N^{-p} \log N + 1\right)$$

$$= O(1). \tag{43}$$

In practice, sub-leading terms may result in behavior which is quite different from $O(1)$ for lengths of the order of millions or even billions of steps.

## 3.5 $d > 3$

For $d > 3$ the time spent on successful pivots is non-negligible, and this changes the overall mean time per attempted pivot.

As discussed by Madras and Sokal [17], we expect that the exponent associated with the acceptance fraction, $p$, is of the same order of magnitude as the critical exponent $\gamma$, with a heuristic argument that $p \lesssim \gamma - 1$. It appears to be the case for $\mathbb{Z}^2$ and $\mathbb{Z}^3$ that $p < \gamma - 1$; if $p \leq \gamma - 1$ holds for $d > 3$, then this implies that $p$ is zero for $d \geq 4$, with perhaps a logarithmic correction for $d = 4$.

For $d = 4$, the number of self-avoiding walks of length $N$, $c_N$, is given asymptotically by $c_N \sim A(\log N)^{1/4}\mu^N$. Therefore, the probability of being able to successfully merge two SAWs (see Sect. 2.1 for definition of the merge operation) of $N$ steps ($N + 1$ sites) to form a SAW of length $2N + 1$ ($2N + 2$ sites) is

$$
\frac{c_{2N+1}}{c_N^2} \approx \frac{A\,(\log(2N+1))^{1/4}\,\mu^{2N+1}}{A^2\,(\log N)^{1/2}\,\mu^{2N}}
$$
$$
= O\left((\log N)^{-1/4}\right). \tag{44}
$$

We have already observed that $\gamma - 1 \neq p$ for $\mathbb{Z}^3$, and so we expect that the exponent of $\log N$ in (44) is unlikely to be the same as for the probability of a successful pivot. A plausible guess for the probability is $\Pr(\text{successful}) = O((\log N)^{-\kappa})$, with $0 < \kappa < 1$. We have performed a preliminary analysis of the data for the acceptance fraction, $f$, for $\mathbb{Z}^4$, which suggests that $\kappa \lesssim 0.42$; however, the plot of $\log(\log(-f))$ versus $\log\log N$ has not settled down to linearity by $N = 2^{28} - 1 \approx 2.7 \times 10^8$, and so the error on this estimate may well be quite large. Assuming this functional form is correct, we obtain

$$
\Pr(i_r) = -\frac{d}{di_r}(\log i_r)^{-\kappa}
$$
$$
= O\left(i_r^{-1}(\log i_r)^{-\kappa-1}\right) \tag{45}
$$

$$
T_{\mathrm{U}}(N) \approx \int_1^N \log i_r\, i_r^{-1}(\log i_r)^{-\kappa-1}\,di_r
$$
$$
= \int_1^N i_r^{-1}(\log i_r)^{-\kappa}\,di_r
$$
$$
= O\left((\log N)^{1-\kappa}\right). \tag{46}
$$

Assuming that $0 < \kappa < 1$, we have

$$
T(N) = \Pr(\text{successful})T_{\mathrm{S}}(N) + \Pr(\text{unsuccessful})T_{\mathrm{U}}(N)
$$
$$
= (\log N)^{-\kappa}\,T_{\mathrm{S}}(N) + T_{\mathrm{U}}(N)
$$
$$
= O\left((\log N)^{-\kappa}\log N + (\log N)^{1-\kappa}\right)
$$
$$
= O\left((\log N)^{1-\kappa}\right). \tag{47}
$$

Note that the time spent on successful pivots is of precisely the same order as the time spent on unsuccessful pivots. Once again, we mention that the above expression is based on a

plausible but untested assumption. We are confident that $T(N) = \omega(1)$, and also $T(N) = o(\log N)$ (this is the expression reported in Tables 1 and 3).

For $\mathbb{Z}^2$ and $\mathbb{Z}^3$, we could see that as $p \to 0$, the natural length scale $L$ obtained from (42) diverges. For this reason, we expect that deviations from the leading order behavior of $T(N)$ for $\mathbb{Z}^4$, where $p = 0$ with logarithmic corrections, may be large even for extremely long walks with $10^9$ steps or more.

For $d > 4$, we can use the mean-field approximation and ignore long range correlations within the chain, by assuming that a sub-walk of length $N$ consists of sites selected uniformly at random from a sphere of radius $r = O(N^\nu) = O(N^{1/2})$. Assuming that we already have a self-avoiding sub-walk with sites in the interval $(j - i_r, j + i_r)$, then the probability that the site $j + i_r$ intersects with one of the previous sites is $\Pr(i_r) = O(i_r/r^d) = O(i_r^{1-d/2})$. Therefore

$$T_U(N) \approx \int_1^N \log i_r \, i_r^{1-d/2} di_r$$

$$= O(1). \qquad (48)$$

Finally, we have

$$T(N) = \Pr(\text{successful})T_S(N) + \Pr(\text{unsuccessful})T_U(N)$$

$$= O(1)T_S(N) + O(1)T_U(N)$$

$$= O(\log N + 1)$$

$$= O(\log N). \qquad (49)$$

As $T_S(N) = \Theta(\log N)$, this leads to the stronger statement that $T(N) = \Theta(\log N)$. Note that the complexity of the algorithm is now dominated by the time spent on successful pivots.

3.6 Numerical Evidence

In Fig. 8 we present $T(N)$ for **Attempt_pivot_fast** ($\mathbb{Z}^2$ and $\mathbb{Z}^3$), and **Attempt_pivot_simple** ($\mathbb{Z}^2$), for lengths from $N = 2^7 - 1$ to $N = 2^{28} - 1 \approx 2.7 \times 10^8$. In Fig. 9 we present

$$\Delta T(N) \equiv T(N\sqrt{2}) - T(N/\sqrt{2}); \qquad (50)$$

the domain of $N$ for this plot is reduced due to the crossover in algorithm performance which occurs at $N \approx 10^4$. These estimates for $T(N)$ were obtained in a separate data run from the main computer experiment in the companion article [4]. The computers used were SunFire X4600M2 machines with 8 quad-core AMD Barcelona CPUs with clock frequency 2.3 GHz, and 64 GB memory.

In Fig. 8, the plot for **Attempt_pivot_simple** appears to be linear, and this is verified in Fig. 9 where $\Delta T$ appears to be constant, providing support for the prediction that $T(N) = \Theta(\log N)$. The plot for **Attempt_pivot_fast** for $\mathbb{Z}^2$ appears to be consistent with eventually approaching a constant; in Fig. 8 we see that $T(N)$ is visibly curved for $N > 10^5$, and is plausibly $o(\log N)$, while in Fig. 9 we observe that $\Delta T(N)$ appears to be steadily declining. The final point increases somewhat, but there is an anomalous increase for all three data sets which strongly suggests this is an artifact. For the final data point a large fraction of machine memory was used, and so it is very likely that some limit with the hardware was reached, causing degraded performance. For $\mathbb{Z}^3$, there is no curvature visually apparent in

**Fig. 8** $T(N)$ for $\mathbb{Z}^2$, and $\mathbb{Z}^3$, where for $\mathbb{Z}^2$ data were collected for both the simple and fast versions of **Attempt_pivot**
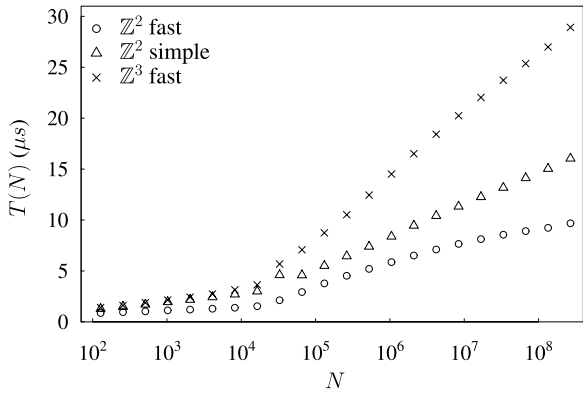


**Fig. 9** $\Delta T(N)$ for $\mathbb{Z}^2$ and $\mathbb{Z}^3$, where for $\mathbb{Z}^2$ data were collected for both the simple and fast versions of **Attempt_pivot**
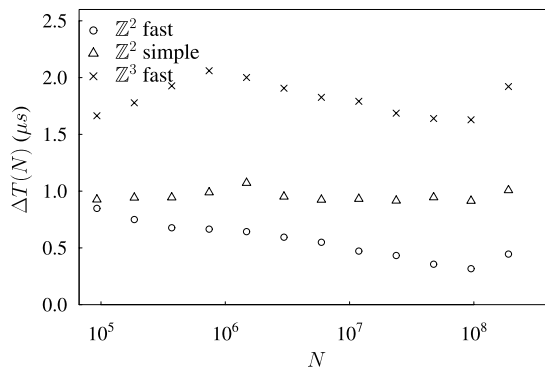


Fig. 8 which suggests $T(N) = O(\log N)$, but if we examine Fig. 9 we see weak numerical evidence supporting $T(N) = o(\log N)$, due to the decline in $\Delta T(N)$ for $N > 10^6$.

Rather than directly measuring the run-time of a computer experiment, it is also possible to directly measure the mean number of intersection tests required, $I(N)$, per attempted pivot for SAWs of $N$ steps. For **Attempt_pivot_fast**, we can relate $T(N)$ to $I(N)$ as follows:

$$T_S(N) = \log N + I_S(N), \tag{51}$$

$$T_U(N) = I_U(N), \tag{52}$$

where $I_S(N)$ and $I_U(N)$ are the mean number of intersection tests when a pivot attempt is successful and unsuccessful respectively. Determining $I(N)$ is far cleaner than directly measuring $T(N)$, which is affected by hardware, and it is also straightforward to separately determine $I_S(N)$ and $I_U(N)$.

We have performed another computer experiment where we measured $I_S(N)$, $I_U(N)$, and $I(N)$, with lengths from $N = 7$ to $N = 2^{28} - 1 \approx 2.7 \times 10^8$. In each case the initial $10^9$ configurations were discarded (independent of $N$), and then 100 batches of $10^8$ configurations were generated. The data from this computer experiment is presented visually in Figs. 10–15, where

$$\Delta I(N) \equiv I(N\sqrt{2}) - I(N/\sqrt{2}). \tag{53}$$

**Fig. 10**  $I_S(N)$ and $I_U(N)$ for **Attempt_pivot_fast** on $\mathbb{Z}^2$
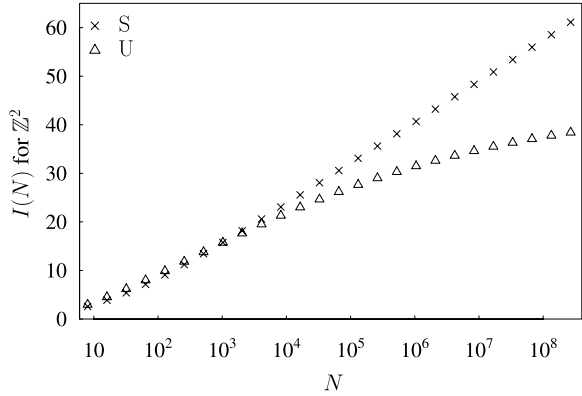


**Fig. 11**  $\Delta I_S(N)$ and $\Delta I_U(N)$ for **Attempt_pivot_fast** on $\mathbb{Z}^2$
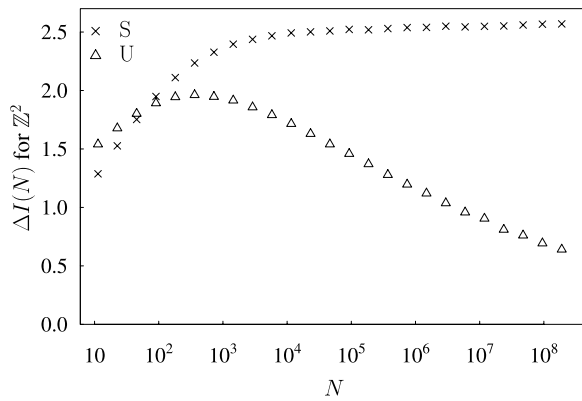


This procedure does not properly initialize the Markov chain for large $N$, but we have checked the resulting time series for any evidence of systematic errors, and we are confident that the systematic error is small compared to the statistical error. We do not show the statistical errors in Figs. 10–15, but the largest errors are smaller than the data point symbols; the small amount of scatter visible in plots of $\Delta I(N)$ is consistent with these errors.

We note in passing that the mean number of intersection tests required per attempted pivot is remarkably low: for $N = 2^{28} - 1$, $I(N)$ is 39 for $\mathbb{Z}^2$, 158 for $\mathbb{Z}^3$, and 449 for $\mathbb{Z}^4$.

Examining the evidence for $\mathbb{Z}^2$ from Figs. 10 and 11, we can see clearly that $I_S(N)$ is convex up to $N = 2^{28} - 1$, and as $N \to \infty$ the lin-log plot of $I_S(N)$ smoothly approaches a straight line strongly supporting the statement $I_S(N) = \Theta(\log N)$. This leads to $T_S(N) = \Theta(\log N)$, in accordance with our conclusion in Sect. 3.2. The evidence for $I_U(N)$ is less conclusive: $I_U(N)$ is convex for small $N$, but becomes concave once a threshold is reached. We do not feel that it is likely that there is an additional length scale beyond $N = 2^{28} - 1$ where the behavior of $I_U(N)$ would change, and therefore believe that the most likely scenario is that $\Delta I_U(N)$ will decay smoothly to zero, which would imply $I_U(N) = o(\log N)$. This argument is by no means conclusive, but it is consistent with the prediction from Sect. 3.4 that $I_U(N) = O(1)$.

The evidence for $\mathbb{Z}^3$ from Figs. 12 and 13 is less clear. In Sect. 3.4 we argued that for $\mathbb{Z}^3$ the asymptotic regime would be reached at far greater lengths than for $\mathbb{Z}^2$, and this certainly appears to be the case. The graphs for $I_S(N)$ suggest that $I_S(N) = \Theta(\log N)$, but

**Fig. 12** $I_S(N)$ and $I_U(N)$ for **Attempt_pivot_fast** on $\mathbb{Z}^3$
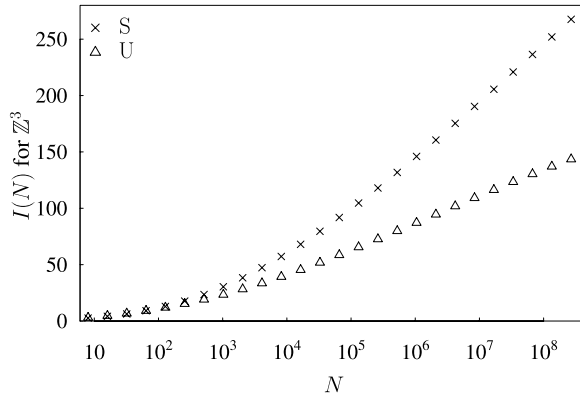


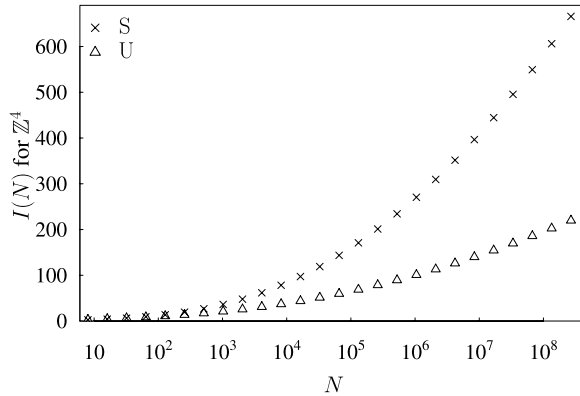**Fig. 13** $\Delta I_S(N)$ and $\Delta I_U(N)$ for **Attempt_pivot_fast** on $\mathbb{Z}^3$
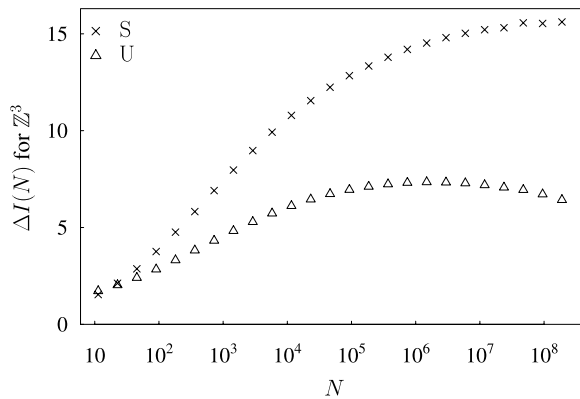


we do not believe the evidence is particularly strong, as there is no extended flat region for $\Delta I_S(N)$ (as can be seen for $\mathbb{Z}^2$). The plots for $I_U(N)$ show that $I_U(N)$ is convex for $N$ up to approximately $N = 10^6$, before becoming concave. If the downward trend in $\Delta I_U(N)$ were to continue in the same manner as for $\mathbb{Z}^2$, then this would imply $I_U(N) = o(\log N)$. However, the numerical evidence supporting $I_U(N) = o(\log N)$ is weak at best, while a stronger case can be made on the basis of the numerics that $I_U(N) = O(\log N)$.

For $\mathbb{Z}^4$, the evidence from Figs. 14 and 15 appears to support the statements $I_S(N) = \omega(\log N)$ and $I_U(N) = \omega(\log N)$, as the plots for both $I_S(N)$ and $I_U(N)$ are convex at $N = 2^{28} - 1$. This directly contradicts the predictions that $I_S(N) = \Theta(\log N)$ and $I_U(N) = o(\log N)$. However, this is not too surprising, as for $\mathbb{Z}^4$ we have $p = 0$ with logarithmic corrections, and so the asymptotic regime may not be reached until $N$ is truly large, i.e. $N \gg 2^{28}$.

In summary, the numerical evidence supports our heuristic argument that $T_S(N) = \Theta(\log N)$ for $\mathbb{Z}^2$ and to lesser extent for $\mathbb{Z}^3$. We argue that the data for $I_U(N)$ and $T_U(N)$ imply $T_U(N) = o(\log N)$ for $\mathbb{Z}^2$, and $T_U(N) = O(\log N)$ for $\mathbb{Z}^3$; the numerics are consistent with predictions that $T_U(N) = O(1)$. However, the asymptotic regime has not yet been reached at $N = 2^{28} - 1$, and so no strong conclusion regarding $T_U(N)$ can be made based on the numerics. For $\mathbb{Z}^4$ the numerical evidence does not support the conclusions from our heuristic arguments; we believe that this is due to the asymptotic regime being well beyond $N = 2^{28} - 1$.

**Fig. 14** $I_S(N)$ and $I_U(N)$ for **Attempt_pivot_fast** on $\mathbb{Z}^4$



**Fig. 15** $\Delta I_S(N)$ and $\Delta I_U(N)$ for **Attempt_pivot_fast** on $\mathbb{Z}^4$
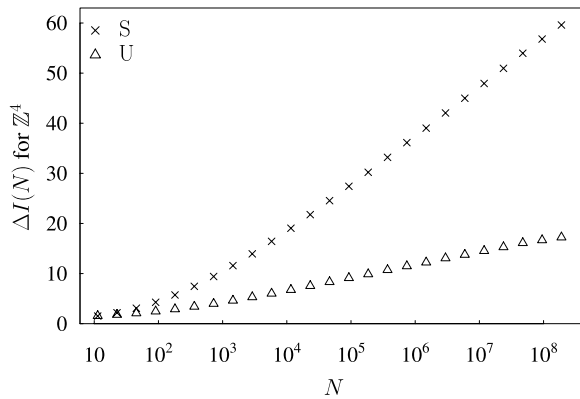


We wish to make one final point: although the performance of an $O(1)$ implementation must eventually be far superior to the performance of an $\Theta(\log N)$ implementation, we note that for $N$ as high as $2.7 \times 10^8$ the speed-up gained for $\mathbb{Z}^2$ is only a factor of 1.65 for **Attempt_pivot_fast** versus **Attempt_pivot_simple**. This factor can be expected to grow, but even for walks on $\mathbb{Z}^2$ with $10^{12}$ steps, **Attempt_pivot_fast** will be only a factor of two faster than **Attempt_pivot_simple**.

### 3.7 Summary

Although $T(N)$ is asymptotically smaller for $\mathbb{Z}^2$ and $\mathbb{Z}^3$ than for higher dimensions, this does *not* mean that our implementation is more efficient for $\mathbb{Z}^2$ and $\mathbb{Z}^3$ than for higher dimensions, as it is the integrated autocorrelation time in CPU units for global observables which is important. The integrated autocorrelation time in physical units (pivot attempts), $\tau_{\text{int}}(N)$, for observables such as $R_e^2$ is of approximately the same order as the time needed to achieve a successful pivot, perhaps with an additional logarithmic factor [17]. The best available evidence suggests that there is no logarithmic factor for $\mathbb{Z}^2$ and $\mathbb{Z}^3$ [15, 17], but there is as yet no evidence on this question for $\mathbb{Z}^d$ with $d \geq 4$. We define $\tilde{\tau}_{\text{int}}(N)$ as the integrated autocorrelation time in CPU units, which can be expressed as $\tilde{\tau}_{\text{int}}(N) = \tau_{\text{int}}(N)T(N)$. Neglecting constant factors, the relevant expressions for $T(N)$ and $\tilde{\tau}_{\text{int}}(N)$ are therefore

$$T(N) = \Pr(\text{successful})T_S(N) + \Pr(\text{unsuccessful})T_U(N); \tag{54}$$

**Table 3** $T(N)$ and $\tilde{\tau}(N)$ for $N$-step SAWs on $\mathbb{Z}^d$. The expressions for $\tilde{\tau}_{\text{int}}(N)$ may have an additional logarithmic factor [17]

|  | $\mathbb{Z}^2$ | $\mathbb{Z}^3$ | $\mathbb{Z}^4$ | $\mathbb{Z}^d, d > 4$ |
|---|---|---|---|---|
| $T(N)$ | $O(1)$ | $O(1)$ | $o(\log N)$ | $\Theta(\log N)$ |
| $\tilde{\tau}_{\text{int}}(N)$ | $O(N^{0.19})$ | $O(N^{0.11})$ | $O(\log N)$ | $\Theta(\log N)$ |

$$\tilde{\tau}_{\text{int}}(N) = \tau_{\text{int}}(N)T(N)$$

$$= \frac{1}{\text{Pr(successful)}}T(N)$$

$$= T_S(N) + \frac{\text{Pr(unsuccessful)}}{\text{Pr(successful)}}T_U(N). \tag{55}$$

This leads to the estimates for the complexity of $T(N)$ and $\tilde{\tau}_{\text{int}}(N)$ for $\mathbb{Z}^d$ given in Table 3.

We observe that our implementation of the pivot algorithm has a crossover at dimension $d = 4$: for $d < 4$, most CPU time is spent on unsuccessful pivots, while for $d > 4$, most CPU time is spent on successful pivots.

## 4 Initialization

As discussed in [17], the pivot algorithm has short integrated autocorrelation time of $O(N^p)$, but long exponential autocorrelation time of $O(N^{1+p})$ ($p$ positive, but close to zero). As it is infeasible to initialize the Markov chain for large $N$ by directly sampling from the equilibrium distribution via dimerization [1] (for $d \geq 4$ dimerization is sufficiently efficient that it can be used even for very long walks), there will necessarily be a systematic bias introduced from the initialization. To ensure that the systematic error is much less than the statistical error, it is necessary to discard a number of time steps which is significantly larger than the exponential autocorrelation time. Madras and Sokal [17] argued that the exponential autocorrelation time is $O(N/f)$, where $f$ is the acceptance fraction of the pivot algorithm, and discarded the first $20N/f$ time steps. We adopt the same procedure here. We note that for sufficiently large $N$ the time for initialization can dominate the running time of the algorithm; for the longest walks studied, initialization took approximately 2 weeks of computer time.

There is an additional complication for our implementation: although we expect the mean time per pivot attempt for the SAW-tree implementation to be $O(1)$, this is not necessarily true for atypical SAWs. In particular, if the Markov chain is initialized with a walk with long straight segments, such as a straight rod, then there can be $O(N)$ nearest neighbor contacts between two halves of a walk, and so in the worst case a successful pivot may take time $O(N)$, which is the same as the average-case performance of the implementation of Madras and Sokal, and far worse than $O(\log N)$. We have not precisely characterized the average-case behavior of the SAW-tree implementation when initialized with a straight rod, but it is clear that pivot attempts are far slower when the walk has many straight segments. It may be of interest to study the typical performance of the algorithm when initialized with a straight rod, e.g. to characterize how rapidly the CPU time per pivot attempt decays to the equilibrium value, and this will be done in a future computer experiment.

**Fig. 16** Initialization for
33554431-step SAWs on $\mathbb{Z}^3$, in
batches of $10^8$; means for
batches of $10^9$ configurations are
shown as *lines*. The initial
configuration was generated
using **Pseudo_dimerize**
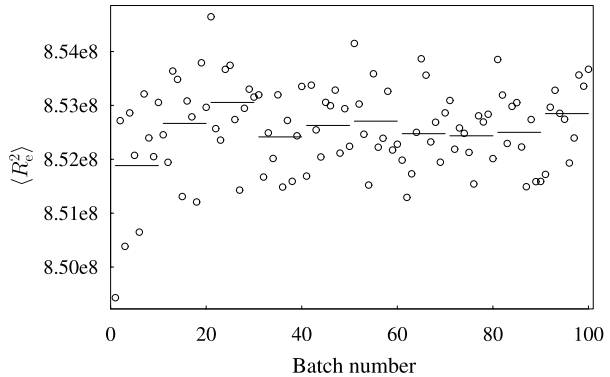


To overcome the difficulty involving straight rods, we developed the pseudo-dimerization procedure defined in Sect. 2.6, **Pseudo_dimerize**. This procedure utilizes repeated pivot operations to generate SAWs in time $\Theta(N)$ which are quite difficult to distinguish from SAWs sampled from the uniform distribution. For $N = 33554431$, $20N/f$ corresponds to approximately 50 batches of $10^8$. In Fig. 16, initialization bias is visually apparent for (at most) the first 10 batches, which suggests that discarding $20N/f$ configurations is quite conservative when initializing the Markov chain by using the pseudo-dimerization procedure. If we could argue that the pseudo-dimerization samples from a distribution which is in some sense close to the uniform distribution, and if we could quantify this, then it might be possible to spend less time on initialization. In the absence of any such argument, we strongly recommend a cautious approach, i.e. that $20N/f$ configurations should be discarded at the beginning of each data run.

Although this has not been carefully tested, it seems that there is no significant increase in the time per attempted pivot during the initialization period when the initial SAW is generated via the pseudo-dimerization procedure. It may appear that **Pseudo_dimerize** is not strictly necessary, but in practice it significantly extends the length of walks which can be studied compared to initialization with a straight rod.

### 4.1 Algorithmic Complexity of **Pseudo_dimerize**

We will now provide a theoretical argument as to why pseudo-dimerization takes time $\Theta(N)$.

The mean CPU time needed for **Pseudo_dimerize** to generate a SAW of $n$ sites may be expressed recursively from the time required to generate two SAWs of $n/2$ sites. We use the number of steps $N = n - 1$ in the following expressions for consistency with other sections, and denote the mean CPU time as $T_{\mathrm{PD}}(N)$. Thus we have:

$$T_{\mathrm{PD}}(N) = 2T_{\mathrm{PD}}(N/2) + (\text{CPU time to merge walks})$$

$$= 2T_{\mathrm{PD}}(N/2) + (\text{CPU time per pivot attempt})$$

$$\times (\text{Expected number of pivot attempts}) \qquad (56)$$

In Sect. 3 we predicted that the CPU time per pivot attempt for $\mathbb{Z}^2$ and $\mathbb{Z}^3$ is $O(1)$, and $O(\log N)$ for $\mathbb{Z}^d$ with $d \geq 2$. We will use the weaker bound $O(\log N)$ for our argument here. Calculating the expected number of pivot attempts is, however, quite subtle. Naively,

we can assume that each of the $n/2$ site walks has been sampled uniformly from SAWs of length $n/2$; we expect that this approximation is "good enough", and will only result in a small error that will not change our conclusions. The number of SAWs of $N$ steps for $\mathbb{Z}^2$ and $\mathbb{Z}^3$ is given asymptotically by $c_n \sim A\mu^N N^{\gamma-1}$, and so the probability of successfully merging two independent SAWs with $n/2$ sites is given by

$$\Pr(\text{merge}) = \frac{c_{n-1}}{c_{n/2-1}^2}$$

$$\approx \frac{A\mu^{n-1}(n-1)^{\gamma-1}}{(A\mu^{n/2-1}(n/2-1)^{\gamma-1})^2}$$

$$= O(N^{1-\gamma}) \tag{57}$$

However, successive SAWs in the Markov chain are highly correlated, and we account for this correlation by introducing an observable $C$, defined as

$$C(\omega^l, \omega^r) = \begin{cases} 1 & \omega^l \text{ and } \omega^r \text{ can be merged} \\ 0 & \omega^l \text{ and } \omega^r \text{ cannot be merged.} \end{cases} \tag{58}$$

The expected number of pivot attempts before the left and right sub-walks are successfully merged is $\tau_{\text{int}}(C)/\langle C \rangle$, and we know that

$$\langle C \rangle \equiv \Pr(\text{merge}) = O(N^{1-\gamma}).$$

In our implementation of **Pseudo_dimerize** we perform an additional $O(N^{1/2})$ pivot attempts after the sub-walks have been merged, and hence

$$T_{\text{PD}}(N) = 2T_{\text{PD}}(N/2) + O(\log N)(N^{\gamma-1}\tau_{\text{int}}(C) + N^{1/2})$$

$$= 2T_{\text{PD}}(N/2) + O(N^{\gamma-1}\log N)\tau_{\text{int}}(C) + O(N^{1/2}\log N). \tag{59}$$

Equation (57) leads to the conclusion that the mean distance of the first intersection between the left- and right-hand sub-walks from the joint where the walks meet is $O(N^{2-\gamma})$. However, there may well be other length scales which are also important for the calculation of $\tau_{\text{int}}(C)$; in particular, the shape of the walks near the joint at distance scales from $O(1)$ up to $O(N^{2-\gamma})$ will strongly affect the probability of successful merging. This will be discussed in much greater detail in a subsequent paper, in the context of our calculation of the critical exponent $\gamma$ for self-avoiding walks.

We expect that $\tau_{\text{int}}(C)$ is of at most the same order as the time needed to achieve a successful pivot on all possible length scales, i.e. the time to achieve a successful pivot in each of the ranges, in terms of distance from the joint, of $[1, 2), [2, 4), [4, 8), \ldots, [n/2, n-1)$. By choosing pivot nodes using **Random_integer_log**, pivot sites are chosen uniformly at random on a logarithmic scale in terms of their distance from the joint, and so in time $O(\log N)$ pivot *attempts* are made on all length scales. When selecting the pivot site uniformly at random (on a linear scale), the probability of a successful pivot is $O(N^{-p})$. We expect that a pivot attempt is more likely to be successful if the pivot site is near one end rather than close to the middle, and thus for pivots selected on a logarithmic scale the probability of a successful pivot will remain $O(N^{-p})$. Altogether, when using **Random_integer_log** to select pivot sites, this implies that $\tau_{\text{int}}(C) = O(N^p \log N)$. Thus, our final recurrence relation is

$$T_{\text{PD}}(N) = 2T_{\text{PD}}(N/2) + O(N^{\gamma-1+p}\log^2 N) + O(N^{1/2}\log N). \tag{60}$$

**Fig. 17** Autocorrelation function for observables $R_e^2$, $R_g^2$, and $\mathscr{R}_m^2$. See (22) for a definition of $\mathscr{R}_m^2$; the key point is that $\langle\mathscr{R}_m^2\rangle = \langle R_m^2\rangle$ for all $N$
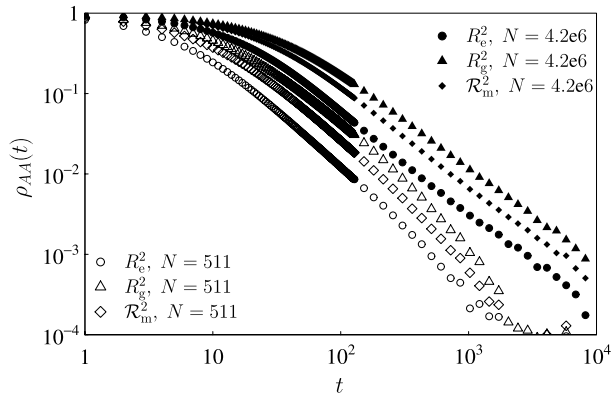


For $\lambda < 1$, the solution of the recurrence relation $T_{PD}(N) = 2T_{PD}(N/2) + O(N^{\lambda}\log^2 N) + O(N^{1/2}\log N)$ is $T_{PD}(N) = \Theta(N)$. The sequence of approximations used leads to the estimate $\lambda \approx \gamma - 1 + p \approx 0.53$ for $\mathbb{Z}^2$ and $\lambda \approx 0.27$ for $\mathbb{Z}^3$; we are confident that our approximations are sufficiently accurate that the correct $\lambda$ will be less than one, and hence that the expected running time of **Pseudo_dimerize** is indeed $\Theta(N)$. This argument can be straightforwardly repeated for $\mathbb{Z}^d$, $d \geq 4$, leading to the same conclusion that $T_{PD}(N) = \Theta(N)$.

We have clear numerical evidence from informal computer experiments that this result is correct.

## 5 Error Estimates and the Autocorrelation Function

Following [15], and given the variance of an observable, $\text{var}(A) = \langle A^2\rangle - \langle A\rangle^2$, we define the autocorrelation function for the time series measurement of an observable $A$ as

$$\rho_{AA}(t) = \frac{\langle A_s A_{s+t}\rangle - \langle A\rangle^2}{\text{var}(A)}. \tag{61}$$

We have calculated the autocorrelation function for the Euclidean-invariant moments $R_x^{2k}$ with $x \in \{e, g, m\}$, $1 \leq k \leq 5$, for $N = 2^l - 1$, $9 \leq l \leq 22$, for times $t \leq 8192$. We invested approximately 300 hours of CPU time in this endeavor, a relatively small amount compared with the 16500 CPU hours spent on the computer experiment to determine $\nu$ in [4]. The autocorrelation functions for $R_x^2$, with $N = 511$ and $N = 2^{22} - 1 = 4194303$, are shown in Fig. 17. Error bars are not shown on the graph, but the approximate size of the errors can be inferred by the degree of scatter from smooth behavior.

Despite the apparent linearity observed in Fig. 17 (particularly for $N = 4.2 \times 10^6$), it is surprisingly difficult to extract reliable estimates for the rate of decay of the tail of the autocorrelation function. Perhaps this is because the tail is not characterized by a single exponent. Madras and Sokal [17] showed that the pivot algorithm itself has a variety of exponents for the acceptance fraction for different classes of lattice symmetries, and this behavior may extend to the autocorrelation function itself. This problem certainly deserves further study, but we do not have data of sufficient quality to be able to accurately characterize the autocorrelation function.

In Fig. 18, we calculated the integrated autocorrelation time for $R_e^2$ using two different methods.

**Fig. 18** $\tau_{\text{int}}$ for $R_e^2$
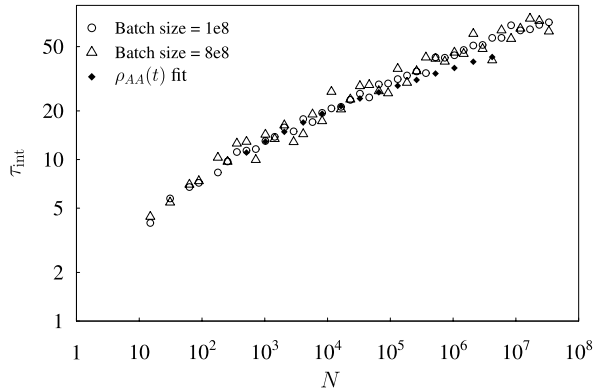


For the direct method, we calculated

$$\tau_{\text{int}}(A) = \frac{1}{2} + \sum_{t=1}^{\infty} \rho_{AA}(t) \tag{62}$$

by using direct summation for short times ($t \leq 128$), and fitting the intermediate regime with a power law truncated at exactly $t = N/f$ (where $f$ is the fraction of pivot attempts which are successful). The accuracy of this method relies on the assumptions that the exponential autocorrelation time is of $O(N/f)$, and that the intermediate regime of the autocorrelation function can be adequately fitted by a single power law. We neglect the regime $t > N/f$ where $\rho(t)$ is presumed to decay exponentially, as the contribution of this tail to $\tau_{\text{int}}$ is negligible compared to the error introduced by other approximations, e.g. the choice of $t = N/f$ rather than $t = cN/f$ with $c \neq 1$.

The indirect method used the batch estimates for the observables, and corresponding confidence intervals stdev($\bar{A}$), by solving

$$\text{stdev}(\bar{A}) = \left( \frac{2\tau_{\text{int}}(A)\text{var}(A)}{n_{\text{sample}}} \right)^{\frac{1}{2}} \tag{63}$$

for $\tau_{\text{int}}(A)$. The accuracy of this technique relies on the assumption that the batch error estimate is accurate, which in turn relies upon the degree of correlation between successive batches being negligible. Provided the exponential autocorrelation time is finite (guaranteed for a finite system), then this condition will be satisfied for sufficiently large batch size. If this condition were not satisfied then estimates of stdev($\bar{A}$), and consequently $\tau_{\text{int}}(A)$, would be systematically low. To estimate $\tau_{\text{int}}$ for $R_x^2$, with $x \in \{e, g, m\}$, we used data from the companion article [4], with 1000 batches of $10^8$ pivot attempts, and 125 batches of $8 \times 10^8$ pivot attempts.

The two indirect estimates shown in Fig. 18 are indistinguishable, although there is more scatter for the batches of $8 \times 10^8$ because of the smaller number of batches used for the estimate of stdev($\bar{A}$). This is strong evidence that a batch size of $10^8$ is sufficiently large so that the degree of correlation between successive batches is negligible up to at least $N = 2^{25} - 1 \approx 3.36 \times 10^7$. Hence, we expect the confidence intervals for estimates of observables in [4] to be accurate, and recommend the batch method for use with the pivot algorithm as a simple and reliable method for estimating confidence intervals.

The direct estimates for $\tau_{\text{int}}$ are close to the indirect estimates, but for sufficiently large $N$ the direct estimates are systematically low. This suggests that either the tail fitting procedure breaks down for large $N$, which we consider unlikely as it is clear from Fig. 17 that the tail has little curvature for large $N$. Or, for large $N$ the truncated part of the tail still contributes non-negligibly to the integrated autocorrelation time, i.e. the exponential autocorrelation time is greater than $O(N/f)$. We consider this latter explanation to be more probable, and will explore the asymptotic behavior of the exponential autocorrelation time in future work.

We refer the interested reader to [15, 17] for more information on the autocorrelation function for the pivot algorithm.

## 6 Performance: Comparison with Other Implementations

In this section we present detailed comparison of the performance of the SAW-tree implementation with the implementations of Madras and Sokal [17] and Kennedy [9], for $N$-step SAWs on $\mathbb{Z}^2$, $\mathbb{Z}^3$, and $\mathbb{Z}^4$, with $N$ ranging from 3 to 33554431.

For this section we will use the shorthand notations S-t for the SAW-tree implementation, M&S for the hash table implementation of Madras and Sokal, and K for Kennedy's implementation.

### 6.1 Experimental Details

For testing M&S we wrote our own version using the programming language C. This implementation has *not* been extensively polished for maximum efficiency, and so it is highly likely that there exist other implementations which are faster by a (small) constant factor.

For testing K, we used Kennedy's C++ program SAW_pivot v1.0, which has been released under the GNU General Public Licence. For information about this implementation, we recommend you consult the relevant article [9], as well as the source code.[2] We used the default settings for SAW_pivot, which meant that updates to the data structure were performed every $N_{\text{pivot}} = \lfloor (N/40)^{1/2} \rfloor$ successful pivots (we set $N_{\text{pivot}} = 1$ for $N \leq 40$). Kennedy [9] indicates that the performance of the algorithm is relatively insensitive to the precise choice of $N_{\text{pivot}}$, and so we expect that tuning $N_{\text{pivot}}$ would result in, at most, only modest improvement in performance.

We have observed that K is faster when walks are rod-like, presumably because the intersection testing algorithm is more efficient when SAWs are spread out.[3] For this reason the timing experiments were initialized with SAWs sampled from the equilibrium distribution, rather than straight rods; these SAWs were generated via the pivot algorithm using the SAW-tree implementation.

The computer experiment was performed on a 64-bit Linux machine with Xeon Barcelona 2.83 GHz quad-core processors. The programs were compiled with gcc version 4.1.2, with optimization flag "-O3"; other optimization flags made little if any difference.

We do not include error bars for our measurements, but we have repeated the experiment to ensure that these numbers are reproducible, and have verified that the deviations between different runs are quite small. We present our results in Table 4 of Appendix B and graphically in Figs. 19–21.

---

[2]Available at http://math.arizona.edu/~tgk/

[3]Interestingly, this is in stark contrast to the behavior of the SAW-tree implementation, where long straight segments in rod-like walks result in significant performance degradation, as explained in Sect. 4.
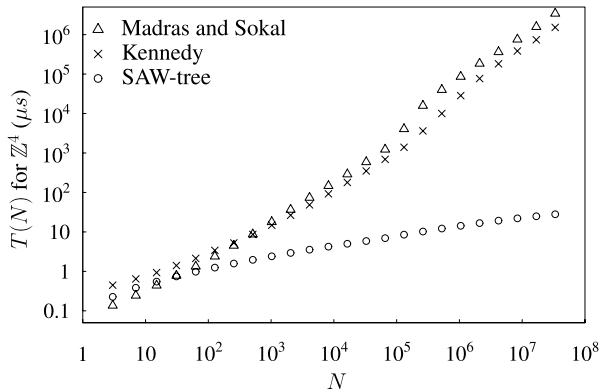
**Fig. 19** $T(N)$ for the Madras and Sokal, Kennedy, and SAW-tree implementations on $\mathbb{Z}^2$
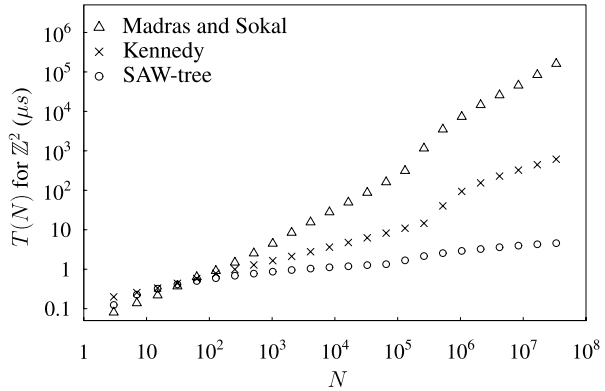


**Fig. 20** $T(N)$ for the Madras and Sokal, Kennedy, and SAW-tree implementations on $\mathbb{Z}^3$
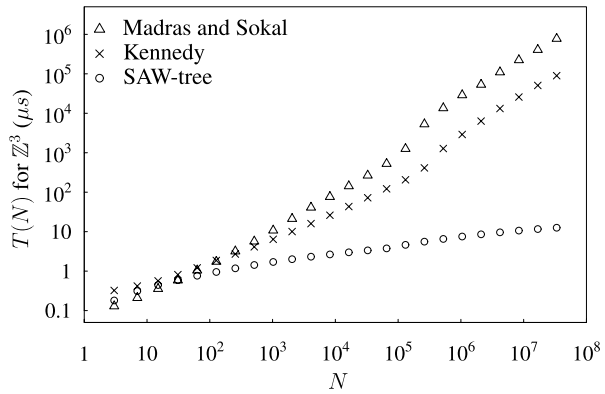


**Fig. 21** $T(N)$ for the Madras and Sokal, Kennedy, and SAW-tree implementations on $\mathbb{Z}^4$



## 6.2 Discussion

In Fig. 19, and to a lesser extent in Figs. 20 and 21, a kink is visible in each of the curves, indicating the length of walk where a hardware limit is reached and the computer programs become memory bound. This occurs at shorter lengths for S-t and M&S, as our implementations of these algorithms use significantly more memory than K.

In [4] we made the statement "For SAWs of length $N = 10^6$ on the cubic lattice, the performance gain for our implementation is approximately 200 when compared with Kennedy's, and over a thousand when compared with that of Madras and Sokal". This statement was based on testing of an earlier version of our SAW-tree implementation, on a different computer to the tests reported here, and we regard the figures in Table 4 of Appendix B as more reliable. In this table, we find that for SAWs on $\mathbb{Z}^3$ with $N = 1048576$, S-t is 385 times faster than K and 3830 times faster than M&S.

For this computer experiment the only observable calculated was $R_e^2$; it is straightforward to extend this to other observables such as $R_g^2$ and $R_m^2$ for the S-t and M&S implementations for a constant factor penalty. It is likely also possible to do the same thing for K, but despite the clear performance advantage for this algorithm over M&S, to the best of our knowledge this has not been done.

We observe from the table and graphs that S-t is lightweight, as it is comparable with M&S for short walks with as few as 15 steps. For $\mathbb{Z}^2$, $\mathbb{Z}^3$, and $\mathbb{Z}^4$, S-t is in fact faster than the other implementations for 63 or more steps.

The difference between the implementations is particularly stark for $N \gtrsim 10^6$, where S-t is significantly faster than the other implementations in all dimensions; this improvement is quite dramatic for $\mathbb{Z}^3$ and $\mathbb{Z}^4$.

On the basis of this computer experiment, where the precise timings are compiler and machine dependent, we can nevertheless draw the following robust conclusions: the SAW-tree implementation is efficient for short walks, and much more powerful than other implementations for long walks. Compared with Kennedy's implementation, there is a large performance boost for $\mathbb{Z}^2$, and a dramatic performance boost for $\mathbb{Z}^d$ with $d \geq 3$.
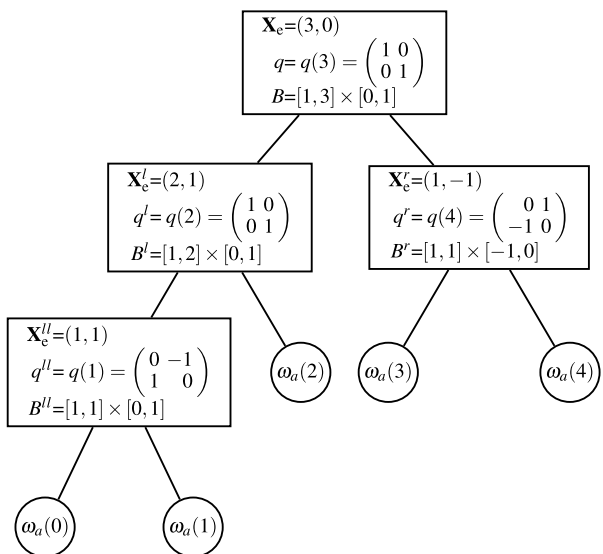
## Appendix A: Example SAW-Trees

**Fig. 22** SAW-tree which is precisely equivalent to the pivot sequence representation for a walk with $n$ sites. Note that $q(0)$ is an overall symmetry which is applied to the whole walk, and cannot be directly included in the SAW-tree data structure
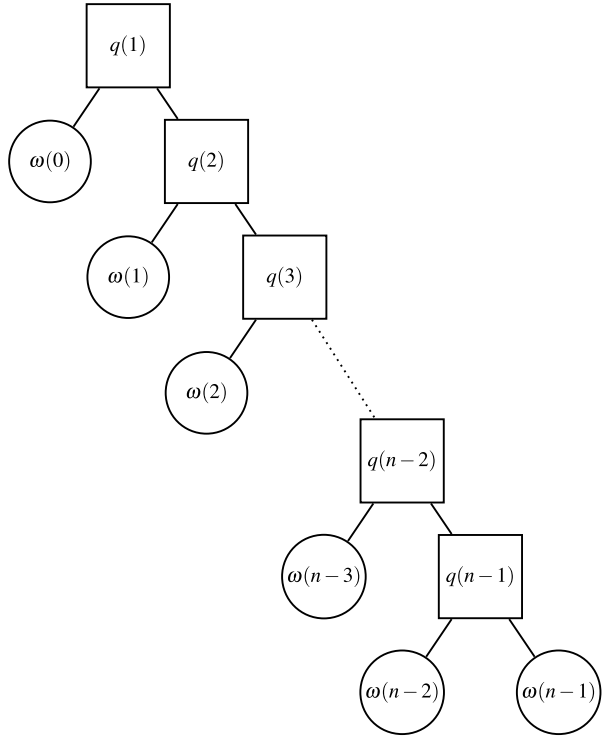
**Fig. 23** A SAW-tree representation of $\omega_a$ (from Fig. 1) involving proper rotations only

**Fig. 24** A SAW-tree
representation of $\omega_a$ (from
Fig. 1) involving proper and
improper rotations
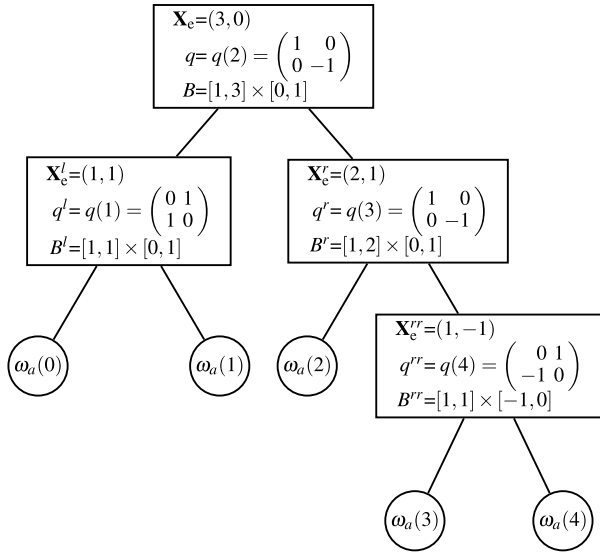


**Appendix B:  Running Times of SAW-Tree, Madras and Sokal, and Kennedy
                     Implementations of the Pivot Algorithm**

**Table 4** $T(N)$ for the SAW-tree implementation (S-t) on $\mathbb{Z}^2$, $\mathbb{Z}^3$, and $\mathbb{Z}^4$, with relative performance given
for the implementations of Madras and Sokal (M&S) [17] and Kennedy (K) [9]. These numbers are given as a
rough guide only, and are machine and compiler dependent

| $N$ | $\mathbb{Z}^2$ | | | $\mathbb{Z}^3$ | | | $\mathbb{Z}^4$ | | |
|---|---|---|---|---|---|---|---|---|---|
| | S-t | M&S/S-t | K/S-t | S-t | M&S/S-t | K/S-t | S-t | M&S/S-t | K/S-t |
| | (µs) | | | (µs) | | | (µs) | | |
| 3 | 0.12 | 0.647 | 1.6 | 0.18 | 0.715 | 1.79 | 0.23 | 0.602 | 1.98 |
| 7 | 0.23 | 0.61 | 1.13 | 0.31 | 0.671 | 1.34 | 0.38 | 0.637 | 1.68 |
| 15 | 0.32 | 0.681 | 1.04 | 0.44 | 0.802 | 1.27 | 0.55 | 0.800 | 1.69 |
| 31 | 0.41 | 0.894 | 1.06 | 0.59 | 0.981 | 1.37 | 0.75 | 1.05 | 1.89 |
| 63 | 0.50 | 1.26 | 1.15 | 0.76 | 1.37 | 1.58 | 0.98 | 1.36 | 2.18 |
| 127 | 0.59 | 1.53 | 1.32 | 0.95 | 1.83 | 1.95 | 1.25 | 1.92 | 2.73 |
| 255 | 0.68 | 2.16 | 1.45 | 1.18 | 2.70 | 2.30 | 1.58 | 2.85 | 3.32 |
| 511 | 0.77 | 3.27 | 1.66 | 1.43 | 3.97 | 2.89 | 1.96 | 4.42 | 4.41 |
| 1023 | 0.87 | 5.15 | 1.90 | 1.71 | 6.31 | 3.75 | 2.41 | 7.41 | 6.13 |
| 2047 | 0.95 | 8.85 | 2.21 | 2.01 | 10.6 | 5.01 | 2.94 | 12.5 | 8.98 |
| 4095 | 1.04 | 14.9 | 2.68 | 2.32 | 17.7 | 6.90 | 3.55 | 20.8 | 13.7 |
| 8191 | 1.12 | 25.0 | 3.25 | 2.65 | 28.8 | 9.84 | 4.24 | 34.7 | 21.8 |
| 16383 | 1.19 | 41.5 | 4.00 | 3.00 | 47.5 | 14.4 | 5.01 | 57.8 | 35.8 |
| 32767 | 1.27 | 68.6 | 4.92 | 3.36 | 79.2 | 21.5 | 5.88 | 100 | 59.6 |
| 65535 | 1.34 | 120 | 6.13 | 3.77 | 139 | 32.3 | 6.95 | 177 | 99.3 |
| 131071 | 1.68 | 186 | 6.50 | 4.63 | 272 | 44.5 | 8.52 | 476 | 164 |

**Table 4** (*Continued*)

| $N$ | $\mathbb{Z}^2$ | | | $\mathbb{Z}^3$ | | | $\mathbb{Z}^4$ | | |
|---|---|---|---|---|---|---|---|---|---|
| | S-t (µs) | M&S/S-t | K/S-t | S-t (µs) | M&S/S-t | K/S-t | S-t (µs) | M&S/S-t | K/S-t |
| 262143 | 2.15 | 536 | 6.76 | 5.58 | 949 | 74.0 | 10.23 | $1.54 \times 10^3$ | 355 |
| 524287 | 2.56 | $1.37 \times 10^3$ | 15.7 | 6.55 | $2.07 \times 10^3$ | 195 | 12.19 | $3.28 \times 10^3$ | 821 |
| 1048575 | 2.91 | $2.51 \times 10^3$ | 32.2 | 7.53 | $3.83 \times 10^3$ | 385 | 14.28 | $6.02 \times 10^3$ | $1.99 \times 10^3$ |
| 2097151 | 3.25 | $4.48 \times 10^3$ | 47.5 | 8.56 | $6.22 \times 10^3$ | 740 | 16.75 | $1.10 \times 10^4$ | $4.60 \times 10^3$ |
| 4194303 | 3.60 | $7.14 \times 10^3$ | 63.5 | 9.62 | $1.14 \times 10^4$ | $1.37 \times 10^3$ | 19.32 | $1.88 \times 10^4$ | $9.41 \times 10^3$ |
| 8388607 | 3.96 | $1.15 \times 10^4$ | 82.0 | 10.65 | $2.09 \times 10^4$ | $2.42 \times 10^3$ | 22.07 | $3.42 \times 10^4$ | $1.74 \times 10^4$ |
| 16777215 | 4.29 | $1.96 \times 10^4$ | 104 | 11.65 | $3.49 \times 10^4$ | $4.36 \times 10^3$ | 24.93 | $6.31 \times 10^4$ | $2.97 \times 10^4$ |
| 33554431 | 4.57 | $3.52 \times 10^4$ | 134 | 12.58 | $6.17 \times 10^4$ | $7.13 \times 10^3$ | 28.03 | $1.22 \times 10^5$ | $5.44 \times 10^4$ |

## References

1. Alexandrowicz, Z.: Monte Carlo of chains with excluded volume: a way to evade sample attrition. J. Chem. Phys. **51**, 561–565 (1969)
2. Baiesi, M., Orlandini, E., Stella, A.L.: Peculiar scaling of self-avoiding walk contacts. Phys. Rev. Lett. **87**, 070602 (2001)
3. Caracciolo, S., Guttmann, A.J., Jensen, I., Pelissetto, A., Rogers, A.N., Sokal, A.D.: Correction-to-scaling exponents for two-dimensional self-avoiding walks. J. Stat. Phys. **120**, 1037–1100 (2005)
4. Clisby, N.: Accurate estimate of the critical exponent $\nu$ for self-avoiding walks via a fast implementation of the pivot algorithm. Phys. Rev. Lett. **104**, 055702 (2010)
5. Frigo, M., Leiserson, C.E., Prokop, H., Ramachandran, S.: Cache oblivious algorithms. In: Proceedings 40th Annual Symposium on Foundations of Computer Science, pp. 285–297 (1999)
6. Gabay, M., Garel, T.: Renormalization along the chemical sequence of a single polymer chain. J. Phys. Lett. **39**, 123–125 (1978)
7. Guttman, A.: R-trees: A dynamic index structure for spatial searching. In: Yormark, B. (ed.) SIGMOD '84, pp. 47–57. ACM, New York (1984)
8. Hara, T., Slade, G.: Self-avoiding walk in five or more dimensions I. The critical behaviour. Commun. Math. Phys. **147**, 101–136 (1992)
9. Kennedy, T.: A faster implementation of the pivot algorithm for self-avoiding walks. J. Stat. Phys. **106**, 407–429 (2002)
10. Klosowski, J.T., Held, M., Mitchell, J.S.B., Sowizral, H., Zikan, K.: Efficient collision detection using bounding volume hierarchies of $k$-dops. IEEE T. Vis. Comput. Gr. **4**, 21–36 (1998)
11. Kremer, K., Baumgärtner, A., Binder, K.: Monte Carlo renormalization of hard sphere polymer chains in two to five dimensions. Z. Phys. B, Condens. Matt. **40**, 331–341 (1981)
12. Kumar, P.: Cache Oblivious Algorithms, pp. 193–212. Springer, Berlin (2003). Chap. 9
13. Lal, M.: 'Monte Carlo' computer simulation of chain molecules. I. Mol. Phys. **17**, 57–64 (1969)
14. Lawler, G.F., Schramm, O., Werner, W.: On the scaling limit of planar self-avoiding walk. In: Fractal Geometry and Applications: a Jubilee of Benoit Mandelbrot, Part 2. Proc. Sympos. Pure Math., vol. 72, pp. 339–364. Am. Math. Soc., Providence (2004)
15. Li, B., Madras, N., Sokal, A.D.: Critical exponents, hyperscaling, and universal amplitude ratios for two- and three-dimensional self-avoiding walks. J. Stat. Phys. **80**, 661–754 (1995)
16. Madras, N., Slade, G.: The Self-Avoiding Walk. Birkhäuser, Boston (1993)
17. Madras, N., Sokal, A.D.: The pivot algorithm: A highly efficient Monte Carlo method for the self-avoiding walk. J. Stat. Phys. **50**, 109–186 (1988)
18. Müller, S., Schäfer, L.: On the number of intersections of self-repelling polymer chains. Eur. Phys. J. B **2**, 351–369 (1998)
19. Nienhuis, B.: Exact critical point and critical exponents of O($n$) models in two dimensions. Phys. Rev. Lett. **49**, 1062–1065 (1982)
20. Oono, Y.: Renormalization along the polymer chain. J. Phys. Soc. Jpn. **47**, 683–684 (1979)

21. Sedgewick, R.: Algorithms in C, Parts 1–4, 3rd edn. Addison-Wesley, Reading (1998)
22. Sokal, A.D.: Monte Carlo methods for the self-avoiding walk. arXiv:hep-lat/9405016 (1994)
23. Sokal, A.D.: Monte Carlo methods for the self-avoiding walk. Nucl. Phys. B (Proc. Suppl.) **47**, 172–179 (1996)
24. van Emde Boas, P.: Preserving order in a forest in less than logarithmic time. In: Proceedings 16th Annual Symposium on Foundations of Computer Science, pp. 75–84 (1975)