Journal of Statistical Physics, Vol. 106, Nos. 3/4, February 2002 (© 2002)

A Faster Implementation of the Pivot Algorithm for Self-Avoiding Walks

Tom Kennedy[1]

Received June 11, 2001; accepted September 26, 2001

The pivot algorithm is a Markov Chain Monte Carlo algorithm for simulating the self-avoiding walk. At each iteration a pivot which produces a global change in the walk is proposed. If the resulting walk is self-avoiding, the new walk is accepted; otherwise, it is rejected. Past implementations of the algorithm required a time $O(N)$ per accepted pivot, where N is the number of steps in the walk. We show how to implement the algorithm so that the time required per accepted pivot is $O(N^q)$ with $q < 1$. We estimate that q is less than 0.57 in two dimensions, and less than 0.85 in three dimensions. Corrections to the $O(N^q)$ make an accurate estimate of q impossible. They also imply that the asymptotic behavior of $O(N^q)$ cannot be seen for walk lengths which can be simulated. In simulations the effective q is around 0.7 in two dimensions and 0.9 in three dimensions. Comparisons with simulations that use the standard implementation of the pivot algorithm using a hash table indicate that our implementation is faster by as much as a factor of 80 in two dimensions and as much as a factor of 7 in three dimensions. Our method does not require the use of a hash table and should also be applicable to the pivot algorithm for off-lattice models.

KEY WORDS: Self-avoiding walk; pivot algorithm; polymer.

1. INTRODUCTION

The self-avoiding walk (SAW) is a simple model for polymers in dilute solution. The interest in the model extends well beyond this application since the model has critical exponents which exhibit universality.[1, 2] The pivot algorithm provides a fast Monte Carlo algorithm for simulating the model, and so it is an ideal laboratory for studying renormalization group predictions and, in two dimensions, conformal field theory predictions. This

[1] Department of Mathematics, University of Arizona, Tucson, Arizona 85721; e-mail: tgk@math.arizona.edu

algorithm first appeared in the literature in 1969.[3] When Madras and Sokal did a detailed study of its efficiency in 1988,[4] the number of papers on the pivot algorithm was still quite small. There is now a vast literature on applying the pivot algorithm to the SAW, continuum (off-lattice) models, SAW's with attractive interactions, self-avoiding polygons and star-branched polymers. Even if one only considers papers which apply the pivot algorithm to the SAW on a lattice, the list of references is substantial.[5–30] Expository accounts of the pivot algorithm may be found in refs. 31 and 32.

At each iteration in the pivot algorithm a pivot is proposed which keeps part of the walk fixed and pivots the rest of the walk. If the new walk is self-avoiding, the pivot is accepted. Most of the proposed pivots are rejected, so in discussing the efficiency of the algorithm, it is important to distinguish between the time required per iteration of the Markov chain and the time required per accepted pivot. The fraction of accepted pivots goes to zero with the length of the walk as N^{-p} for some exponent $p > 0$ which depends on the number of dimensions. So these two times differ by a factor of N^{p}. For global observables such as the distance to the endpoint of the walk, it is believed that only a few accepted pivots are needed to produce an effectively independent walk. Thus the natural measure of efficiency is to consider the time required per accepted pivot since this should roughly measure the time required to produce an essentially independent sample of a global observable.

The most naive check for self intersections takes a time $O(N^2)$. By using a hash table, this check may be done in a time $O(N)$. If one starts at the pivot point and works outwards, self intersections are typically found much faster. With this approach the average time required to check for self intersections is believed to be $O(N^{1-p})$.[4] Thus the time per accepted pivot is $O(N)$, with possibly a logarithmic correction. The time required to carry out the pivot is $O(N)$, and this need only be done once it has been decided that the pivot should be accepted. Thus the best past implementations of the pivot algorithm have only required a time $O(N)$ per accepted pivot.

It is sometimes said that the pivot algorithm cannot be any faster than this since it takes a time $O(N)$ to simply write down a walk with N steps. Nonetheless, we will show how the pivot algorithm may be implemented so that the time required per accepted pivot is $O(N^{q})$ with $q < 1$. Significant corrections to the $O(N^{q})$ behavior make an accurate estimate of q impossible. We estimate that it is less than 0.57 in two dimensions and less than 0.85 in three dimensions. The "effective" value of q depends strongly on the length of the walk, decreasing with the length of the walk. The values of 0.57 and 0.85 are roughly the effective values of q for the longest walks that we can simulate.

We have also performed simulations using the standard implementation of the pivot algorithm using a hash table. For the longest walks in two dimensions we find that our implementation is faster by a factor of around 80, and in three dimensions by a factor of around 7.

In Section 2 we explain our implementation of the pivot algorithm. We consider the theoretical time our implementation requires as a function of N in Section 3. By theoretical time we mean the time that would be required by an ideal computer with unlimited memory that was all equally fast. We also study the real time required to run our implementation of the pivot algorithm and the real time required for the implementation using a hash table for walk lengths up to 1,000,000 steps in two dimensions and up to 640,000 steps in three dimensions.

2. THE IMPLEMENTATION

The pivot algorithm is a Markov Chain Monte Carlo method for simulating the SAW. It defines a Markov chain on the set of walks of a fixed length starting at the origin so that the stationary distribution of the chain is the uniform distribution on the walks. One can then generate samples of the SAW by running the Markov chain. An iteration of the Markov chain starts by picking a random site on the walk. Then one picks a random lattice symmetry g. The section of the walk from the starting point to the randomly chosen site is not changed. The rest of the walk is "pivoted" by applying g to it with respect to the randomly chosen site. This algorithm trivially satisfies detailed balance if the probabilities of choosing g and g^{-1} are equal, and it is not hard to show it is ergodic, i.e., the Markov chain is irreducible.[31, 32]

We denote the sites in the walk by $\omega(i)$ with $0 \leqslant i \leqslant N$. Our walks start at the origin, so $\omega(0) = 0$. The nearest neighbor constraint means that $\|\omega(i) - \omega(i-1)\| = 1$ for $1 \leqslant i \leqslant N$. The self-avoiding constraint means that $\omega(i) \neq \omega(j)$ for $i \neq j$. It is misleading to think of the index i as a time, so we will refer to such indices as "locations" along the walk, rather than times.

There are two main steps in the pivot algorithm, and both limit its performance to $O(N)$ in current implementations. The first is the test for self intersections to see if the new walk should be accepted. The second is actually carrying out the pivot.

The key idea to speed up the first bottleneck is to take advantage of the fact that the walk only takes nearest neighbor steps. When we compare the walk at locations i and j, we do not simply check if $\omega(i) = \omega(j)$. Instead we compute the distance $d = \|\omega(i) - \omega(j)\|$. (The norm used should be the minimum number of nearest neighbor steps needed to get from $\omega(i)$

to $\omega(j)$. On the square and simple cubic lattices this is just the l^1 norm.) If d is nonzero then we can conclude not just that $\omega(i) \neq \omega(j)$, but also that

$$\omega(i') \neq \omega(j'), \qquad \text{if} \quad |i-i'|+|j-j'| < d \tag{1}$$

Thus we can rule out a large number of potential self intersections if d is large. By itself this observation is rather useless; one also needs an algorithm for deciding which values of i and j to check. This observation with $i' = i$ was used in ref. 33, but the resulting algorithm did not do better than $O(N)$. A hierarchical algorithm for choosing the i and j was used in ref. 34 to obtain an efficiency of $O(N)$ per attempted pivot.

Before explaining our algorithm, we will explain the algorithm in ref. 33 since it helps illustrate how observation (1) can be used. Let l be the location at which the pivot is done. Fix an i between $l+1$ and N. We need to check if $\omega(i)$ is equal to $\omega(j)$ for any j between 1 and $l-1$. The naive approach would be to simply do the check for all values of j running from $l-1$ down to 1. The idea in Ref. 33 is that at a given value of j, we compute $d = \|\omega(i) - \omega(j)\|$. If $d = 0$ we have found a self intersection and we are done. If $d > 0$, then we know that $\omega(j)$ is not equal to $\omega(i)$, but we also know that $\omega(j-1),\ldots, \omega(j-d+1)$ are all not equal to $\omega(i)$ since the walk only takes nearest neighbor steps. So instead of decreasing j by just 1, we can decrease it by d. This will produce a dramatic speed-up compared to the naive approach of checking all i and j, but it will still take at least $O(N)$ operations to check a walk with no self-intersections since we must consider all values of i from $l+1$ to N. (Actually, the algorithm requires significantly more operations than that.)

Our algorithm is as follows. As before, we let l be the location at which the pivot is done. Throughout the algorithm, i and j will be locations with $j < l < i$ which have the property that

$$\omega(i') \neq \omega(j') \qquad \forall i', j' \qquad \text{such that} \quad j < j' < l < i' < i \tag{2}$$

Initially, $i = l+1$ and $j = l-1$. At each step the algorithm either decreases j or increases i in such a way that property (2) remains true. (Of course, in trying to do this we may find a self intersection, in which case the test for self-intersections ends.) The procedures for increasing i and decreasing j are completely analogous. We will only explain the procedure for increasing i. Let m_i be the distance from $\omega(i)$ to $\{\omega(k) : j < k < l\}$, i.e.,

$$m_i = \min\{\|\omega(i) - \omega(k)\| : j < k < l\} \tag{3}$$

If $m_i = 0$, then there is a self intersection. If $m_i > 0$, then we know $\omega(i') \neq \omega(j')$ for all i', j' with $i \leqslant i' < i + m_i$ and $j < j' < l$. So instead of just increasing i by 1, we can increase it by m_i. We do not need to compute m_i exactly. If we can compute a lower bound b_i on m_i, then we can increase i by b_i.

We use a loop on j' running from $l-1$ down to j to compute a lower bound b_i on m_i as follows. At the start of the loop we set $b_i = N$. Before we compare $\omega(i)$ and $\omega(j')$, b_i will be a lower bound on the distance from $\omega(i)$ to $\{\omega(k) : j'+1 \leqslant k < l\}$. Let $d = \|\omega(i) - \omega(j')\|$. Pick an integer s with $s < d$. Then the distance from $\omega(i)$ to $\{\omega(k) : j'-s \leqslant k \leqslant j'\}$ is at least $d-s$. So if we replace b_i by $\min\{b_i, d-s\}$, then b_i is a lower bound on the distance of $\omega(i)$ to $\{\omega(k) : j'-s \leqslant k < l\}$. So we can reduce j' by $1+s$. When j' reaches j, b_i will be a lower bound on m_i.

There are a lot of choices for how to choose the integer s. Recall that the only constraint is that $s < d$ where $d = \|\omega(i) - \omega(j')\|$. The simplest choice of s is to take it to be $d/2$. (If d is odd we round $d/2$ down to get s.) However, we have found that the algorithm is significantly faster with the following choice. If $d < b_i$ we take $s = d/2$. However, if $d \geqslant b_i$, then we take $s = d - b_i$. This leaves b_i unchanged and reduces j' by $1 + d - b_i$.

Our check for self-intersections will end when $j \leqslant -1$ and $i \geqslant N+1$. If we reach this point we know the pivoted walk should be accepted. As long as $j > -1$ and $i < N+1$, we are free to choose whether we attempt to increase i or decrease j. The choice we make is to attempt to increase i if it is closer to l, and attempt to decrease j otherwise. This choice means that i and j move away from l at roughly the same rate. So the algorithm checks for self intersections near l before it checks for self intersections that involve locations far from l. Once $j \leqslant -1$ or $i \geqslant N+1$ there is no choice. We can only increase i in the former case and decrease j in the latter.

Since it takes a time $O(N)$ to simply write down a walk with N steps, it seems that the second bottleneck of carrying out an accepted pivot must limit the performance to $O(N)$. To do better, the key idea is to not carry out the pivot each time a pivot is accepted. Instead we keep track of which pivots have been accepted and only carry them out after a certain number have been accepted. This implies that we do not store the present walk, but we will need to know some of the sites in it to determine if we should accept the next pivot. Thus we must store our record of the past pivots in a form which makes it possible to compute individual $\omega(i)$'s efficiently.

A pivot operation acting on the walk ω produces a new walk $\bar{\omega}$ by the equation

$$\bar{\omega}(j) = \begin{cases} \omega(j), & \text{for} \quad j \leqslant l \\ g[\omega(j) - \omega(l)] + \omega(l), & \text{for} \quad j \geqslant l \end{cases} \tag{4}$$

Here l is some location with $0 \leqslant l < N$, which we will refer to as the pivot location. g is a lattice symmetry which fixes the origin, i.e., a linear transformation which maps the lattice back into itself. (For a given lattice, there are only a finite number of possible g.) The pivot is completely determined by l and g, so one could keep track of the pivots that have been accepted by simply keeping a list of the l's and g's. This is not what we do because the time required to compute the position of a location on the walk would be significant. Instead we represent the current walk in the following way.

Suppose that we have accepted n pivots and the pivot locations are $l_1 < l_2 < \cdots < l_n$. (Note that they are in increasing order, not in the order in which they were proposed and accepted.) Let ω be the walk after these pivots, and ω' the walk before these pivots. We can think of the segment of ω' from locations l_i to l_{i+1} as being rigid. The corresponding segment of ω is obtained by applying a single lattice symmetry and translation to the segment in ω'. This motivates representing the walk ω by the following data structure. It consists of

(i) the "old" walk ω'. This is the walk some number of iterations prior to the present.

(ii) an integer n which is the number of pivots that have been accepted but not carried out yet

(iii) pivot locations, $l_1 < l_2 < \cdots < l_n$

(iv) lattice symmetries, $g_1, g_2, \ldots, g_n$

(v) lattice sites, $x_1, x_2, \ldots, x_n$

The walk ω is obtained from this data structure by the equation

$$\omega(j) = g_i \omega'(j) + x_i, \qquad \text{for} \quad l_i \leqslant j \leqslant l_{i+1} \tag{5}$$

with l_0 defined to be 0 and l_n to be N. There is some redundancy in this data structure. Using the fact that ω must be a nearest neighbor walk which starts at the origin, one can determine the x_i from the rest of the data structure. However, this requires a significant amount of time, and it is faster to simply include the x_i with the data structure.

When a pivot is proposed, we do not immediately insert it in the lists in (iii)–(v). Let ω be the walk before the pivot and $\bar{\omega}$ the walk after the pivot. To test if the proposed pivot should be accepted we need to be able to compute $\bar{\omega}(j)$ for selected values of j. We use the data structure and (5) to compute $\omega(j)$ and then use (4) to compute $\bar{\omega}(j)$.

Now suppose that we have accepted a pivot with pivot location l and lattice symmetry g. So the new walk $\bar{\omega}$ is given by (4). We must determine how to update the data structure. We do this in two steps. First we simply add the pivot location to the list. Let k be such that $l_k < l < l_{k+1}$. Then the changes in the data structure are

(ii) $n \to n+1$

(iii) $l_1, l_2,..., l_n \to l_1, l_2,..., l_k, l, l_{k+1},..., l_n$

(iv) $g_1, g_2,..., g_n \to g_1, g_2,..., g_{k-1}, g_k, g_k, g_{k+1},..., g_n$

(v) $x_1, x_2,..., x_n \to x_1, x_2,..., x_{k-1}, x_k, x_k, x_{k+1},..., x_n$

At this stage the data structure still represents the walk before the pivot. It is possible that l is equal to one of the l_i. In this case we simply skip step one.

The second step is to carry out a pivot under the assumption that the pivot location l is in the list $l_1, l_2,..., l_n$. Let $l = l_k$. Suppose that j satisfies $l_i \leqslant j \leqslant l_{i+1}$ with $i \geqslant k$. Then

$$\bar{\omega}(j) = g[\omega(j) - x] + x = g[g_i \omega'(j) + x_i - x] + x = g g_i \omega'(j) + g x_i - g x + x \tag{6}$$

where $x = \omega(l_k)$. Thus for $i \geqslant k$,

$$g_i \to g g_i \tag{7}$$

$$x_i \to g x_i - g x + x \tag{8}$$

For $i < k$, g_i and x_i are left unchanged. In both steps the "old" walk ω' is not changed.

We use this data structure to carry out the pivot algorithm as follows. We fix a large integer N_{pivot} which will be small compared to N, the number of steps in the walk. As pivots are accepted, we update the data structure as explained above. When n reaches N_{pivot}, we use (5) to compute the walk ω and replace ω' in the data structure with ω. Then we set $n = 0$ and delete the lists in (iii)–(v). Note that the integer n in (ii) of the data structure is not always equal to the number of pivots that have been accepted but not yet carried out. It can be slightly less since the pivot location will sometimes already be present in the list in (iii).

As we said before, this data structure is only useful if we can compute $\omega(j)$ quickly for a given j. This is done with Eq. (5). The nontrivial part is finding the i so that $l_i \leqslant j \leqslant l_{i+1}$. This can be done in a time of order $\ln(n)$.

3. ANALYSIS

There are three steps in the pivot algorithm for which the time required depends on the length of the walk.

1. For each proposed pivot we must decide whether to accept the pivot or not.

2. For each accepted pivot we must update elements (ii) to (v) in the above data structure.

3. For every N_{pivot} accepted pivots we must carry out the pivots implicit in the above data structure, i.e., we must update element (i) in the data structure.

One cannot determine a priori how many steps will be needed in the test for self-intersections, so the analysis of the first step will require some empirical study. In this step we must use (5) repeatedly. In this equation we are given j and must find i so that $l_i \leqslant j \leqslant l_{i+1}$ The lists in (iii)–(v) are stored as linear arrays in the order given by the condition $l_1 < l_2 < \cdots < l_n$. Using a bisection procedure, i may be found in a time of order $\ln(n)$, which on average is $O(\ln(N_{\text{pivot}}))$. Let $D(N)$ denote the number of times the distance $\|\omega(i) - \omega(j)\|$ is computed *per accepted pivot*. We assume that $D(N)$ grows as N^{σ}. Except for the steps involved in finding i in (5), the number of steps needed to check for self-intersections per accepted pivot is proportional to $D(N)$. So the time required for the first step will be $O(N^{\sigma} \ln(N_{\text{pivot}}))$ per accepted pivot. We will estimate σ by running simulations, counting the number of times $\|\omega(i) - \omega(j)\|$ must be computed, and dividing this number by the number of accepted pivots.

Now consider the second step. Since the lists in (iii)–(v) are stored as linear arrays, the number of operations required to insert the new entries is of order n. We must also carry out the updates given by (7) and (8). On average this takes order n operations too. The insertion of the new entries could be done more quickly with a more sophisticated data structure, but since (7) and (8) take order n operations, it is not clear that the improvement would be significant. So the time required for the second step is of order n, and since n increases from 0 to N_{pivot}, on average the time per accepted pivot for this step is $O(N_{\text{pivot}})$.

Finally, we consider the third step. The walk in (i) is updated using (5). The search for the index i can be avoided here. We can simply loop on i and then within the loop on i we loop on $j = l_i, \ldots, l_{i+1}$. Thus each application of (5) only takes a time of order 1 and so the update of the walk takes a time of order N. Since this need only be done for every N_{pivot} accepted pivots, the time per accepted pivot is $O(N/N_{\text{pivot}})$.

Thus the total time per accepted pivot is

$$O(N^\sigma \ln(N_{\text{pivot}})) + O(N_{\text{pivot}}) + O\left(\frac{N}{N_{\text{pivot}}}\right) \tag{9}$$

We take N_{pivot} to be proportional to $\sqrt{N}$. In our simulations we have found that $N_{\text{pivot}} = \sqrt{N/40}$ is a good choice for N_{pivot}, both in two and three dimensions. (There is a fairly wide range of values of N_{pivot} for which the algorithm takes roughly the same amount of time.) Then the total time per accepted pivot is

$$O(N^\sigma \ln(N)) + O(N^{1/2}) \tag{10}$$

We now turn to the simulations to estimate σ. We start our simulations with a walk that is a straight line. This is a very atypical walk. For example, the fraction of accepted pivots is significantly higher when one starts with this walk until the walk is well "thermalized." So it is important to first run the algorithm until this initialization bias is removed. We do this as follows. We compute the fraction of locations in the walk at which the walk turns rather than goes straight. This fraction starts at 0. This fraction is a random variable, but for long walks its variance is quite small. We run the algorithm until this fraction has reached roughly 90% of its equilibrium value. Then we run the algorithm for a total of ten times the number of iterations required to reach 90%. This is then the starting point for our estimation of σ.

In Tables I and II we give the results of simulations with our implementation of the pivot algorithm for walk lengths ranging from 1,000 steps to 1,000,000 steps on the square lattice and 1,000 steps to 640,000 steps on the simple cubic lattice. These results for the square lattice are based on 271 million iterations after the thermalization; for the simple cubic lattice they are based on 52 million iterations after the thermalization. The quantity $D(N)$ is the number of distance computations that must be done per accepted pivot. The quantity shown in the tables is the logarithm of $D(N)$.

The acceptance fractions for the pivot algorithm for the various values of N are also shown in the table. For the simple cubic lattice our values of the acceptance fraction are different from those in ref. 22. This is because their computations included the trivial pivot (which is always accepted) as a possible pivot.[35] If f is the acceptance fraction given in their table, then $(48f-1)/47$ is their acceptance fraction without the trivial pivot included. With this correction their values agree well with ours. It is important to note that while including the trivial pivot as a possible pivot is slightly

Table I. Square Lattice: *D(N)* Is the Number of Distance Computations Required per Accepted Pivot. The Acceptance Fraction Is the Ratio of the Number of Accepted Pivots to the Number of Attempted Pivots (Error Bars Are One Standard Deviation)

N	$\ln(D(N))$	Acceptance fraction
1,000	5.861836 ± 0.000175	0.253622 ± 0.000037
1,600	6.180086 ± 0.000189	0.231511 ± 0.000035
2,500	6.476449 ± 0.000213	0.212222 ± 0.000033
4,000	6.782237 ± 0.000232	0.193764 ± 0.000034
6,400	7.082294 ± 0.000248	0.176967 ± 0.000032
10,000	7.362418 ± 0.000251	0.162461 ± 0.000031
16,000	7.653438 ± 0.000279	0.148499 ± 0.000031
25,000	7.925862 ± 0.000309	0.136442 ± 0.000031
40,000	8.209963 ± 0.000314	0.124717 ± 0.000028
64,000	8.489480 ± 0.000345	0.114123 ± 0.000028
100,000	8.753149 ± 0.000382	0.104931 ± 0.000029
160,000	9.027825 ± 0.000371	0.095989 ± 0.000028
250,000	9.286426 ± 0.000408	0.088225 ± 0.000026
400,000	9.555579 ± 0.000432	0.080820 ± 0.000024
640,000	9.822300 ± 0.000455	0.074010 ± 0.000025
1,000,000	10.074441 ± 0.000450	0.068058 ± 0.000023

Table II. Simple Cubic Lattice: The Same Quantities Are Shown as in Table I

N	$\ln(D(N))$	Acceptance fraction
1,000	6.557023 ± 0.000269	0.461623 ± 0.000092
1,600	6.987372 ± 0.000386	0.437485 ± 0.000105
2,500	7.391570 ± 0.000451	0.415891 ± 0.000108
4,000	7.813830 ± 0.000427	0.394216 ± 0.000082
6,400	8.232902 ± 0.000396	0.373658 ± 0.000103
10,000	8.627014 ± 0.000468	0.355183 ± 0.000104
16,000	9.039961 ± 0.000453	0.336693 ± 0.000100
25,000	9.428706 ± 0.000473	0.320281 ± 0.000090
40,000	9.836744 ± 0.000551	0.303548 ± 0.000104
64,000	10.242274 ± 0.000557	0.287784 ± 0.000110
100,000	10.625056 ± 0.000654	0.273644 ± 0.000102
160,000	11.026357 ± 0.000710	0.259694 ± 0.000122
250,000	11.406323 ± 0.000605	0.246912 ± 0.000101
400,000	11.804262 ± 0.000698	0.234145 ± 0.000092
640,000	12.200080 ± 0.000693	0.222179 ± 0.000099

inefficient, it is a completely legitimate implementation of the pivot algorithm. No quantities in ref. 22 need to be corrected other than these acceptance fractions. All the error bars in the table are one standard deviation.

It is surprising how small $D(N)$ is, especially in two dimensions. For example, to check for self intersections in two dimensions, it takes on average only about 25,000 distance computations *per accepted pivot* for walks of length 1,000,000. In three dimensions, the number of distance computations per accepted pivot for walks of length 640,000 is only about 200,000. We believe that the reason $D(N)$ is so much smaller for two dimensions than for three is that the SAW is more spread out in two dimensions.

If $D(N) = cN^{\sigma}$, then

$$\ln(D(N)) = \ln(c) + \sigma \ln(N) \tag{11}$$

So σ and c can be found by performing a weighted least squares fit. In Fig. 1 we show the data and the resulting linear fit for $N = 10{,}000$ to $N = 1{,}000{,}000$. The data points show a systematic error with respect to the fit (11). The weighted residual sum of squares (RSS) is 14292. If the fit were

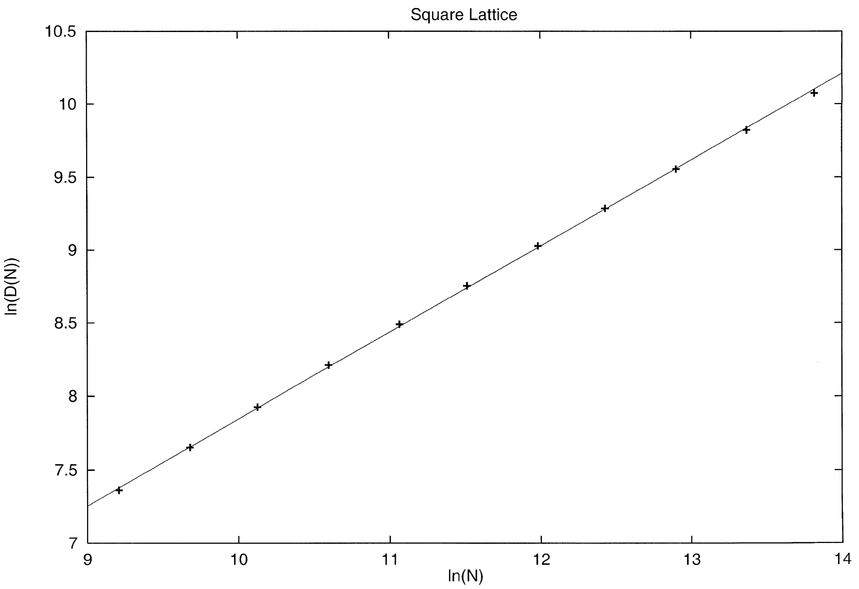

Fig. 1. Square lattice: $D(N)$ is the number of times we must compute $\|\omega(i)-\omega(j)\|$ per accepted pivot. We plot $\ln(D(N))$ as a function of $\ln(N)$. The solid line is a weighted least squares fit assuming $D(N)$ is proportional to N^{σ}.

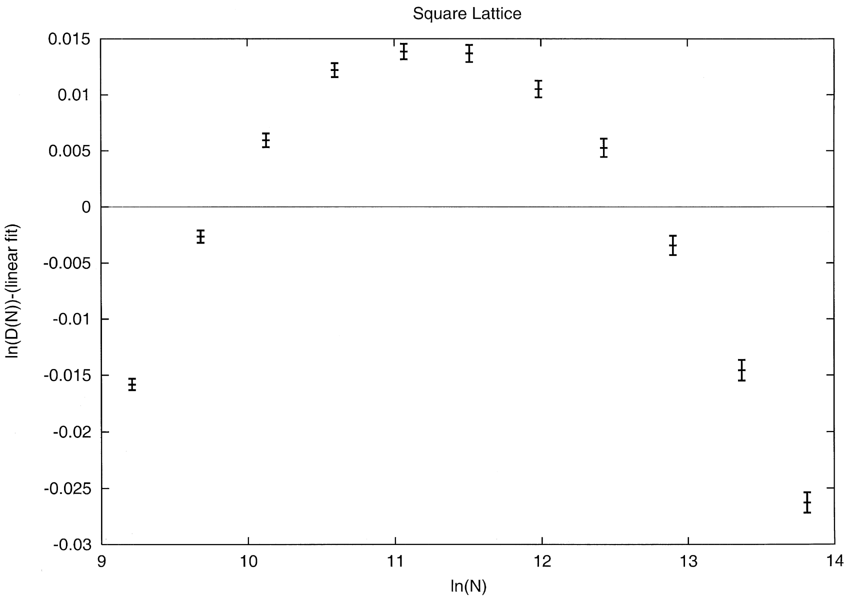

Fig. 2. Square lattice: The points are the difference between $\ln(D(N))$ and the fit in Fig. 1. So the horizontal axis is the fit in Fig. 1.

correct and the errors were independent and normally distributed then RSS would have a chi-squared distribution, and the probability of a value of RSS greater than 21.67 would be 0.01. The systematic error is more apparent in Fig. 2 in which we plot the difference of the data and the fit, i.e., $\ln(D(N))-\ln(c)-\sigma\ln(N)$. So the fit (11) is the horizontal axis in this figure.

The large value of RSS found above indicates that there are significant corrections to (11). This can also be seen by studying how σ changes as we vary the smallest N value that is included in the fit. The bottom curve in Fig. 3 shows the value of σ that comes from a weighted least squares fit to (11) as a function of $1/\ln(N_{\min})$ where $N_{\min}$ is the smallest value of N included in the fit. The figure shows that our estimate of σ depends strongly on $N_{\min}$ and shows no sign of stabilizing for the values of N that we can simulate. The RSS for all these fits are much larger than the values that would be expected if (11) was a true fit to the data. Because of the small statistical errors in our estimates of $D(N)$, the error bars for σ that come from this fit are tiny compared to the change in σ as $N_{\min}$ is changed. These error bars are not shown in the figure.

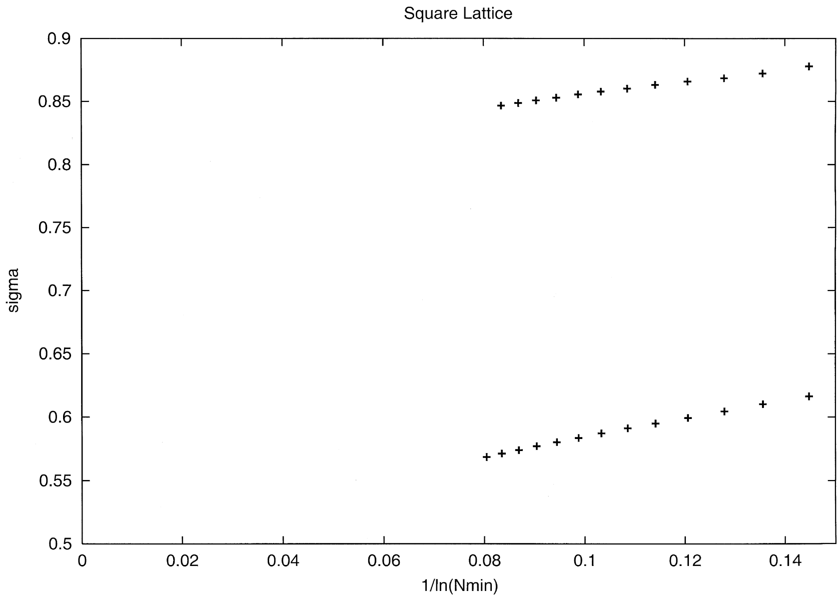

Fig. 3. The value of σ that results from a weighted least squares fit to (11) is plotted as a function of $1/\ln(N_{\min})$. $N_{\min}$ is the length of the shortest walk used in the fit. The bottom curve is for the square lattice; the top curve is for the simple cubic lattice.

One can try to improve the fit by including additional terms in (11). However, because of the lack of any a priori knowledge of the form of these corrections and the size of the corrections, it does not appear that a reliable estimate of σ is possible. As an example, suppose we assume that

$$D(N) = cN^{\sigma}[1 + bN^{-\Delta}] \tag{12}$$

Then $\ln(D(N))$ is approximately

$$\ln(D(N)) = \ln(c) + \sigma \ln(N) + bN^{-\Delta} \tag{13}$$

For a given value of Δ we can find σ and b by a weighted least squares fit. We then search over Δ to find the value that gives the smallest RSS. However, a rather wide range of values of Δ will give an acceptable value of RSS and the resulting values of σ vary considerably. For example, if we use $N_{\min} = 10{,}000$ then RSS is less than the 99% confidence level for Δ ranging from 0.072 to 0.240. The resulting values of σ range from 0.403 and 0.534. We should also note that these values of Δ are rather small and the values of b that we obtain are all greater than 3. So (13) is not a good approximation to (12).

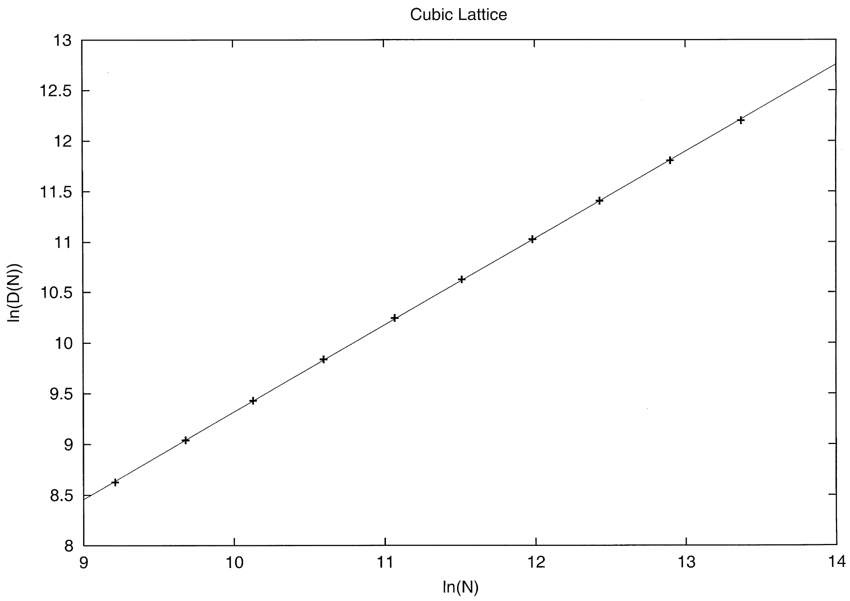

Fig. 4. $D(N)$ for the simple cubic lattice. The line is a weighted least squares fit assuming $D(N)$ is proportional to N^{σ}.

For the simple cubic lattice the data for $N = 10{,}000$ to $N = 640{,}000$ and the fit using (11) are shown in Fig. 4. There is again a systematic error in this fit; RSS is 1591 while the 99% confidence level is 20.09. It is more clearly seen in Fig. 5 in which the difference of $\ln(D(N))$ and the linear fit is plotted. The top curve in Fig. 3 shows the value of σ we obtain from the fit to (11) using different values of $N_{\min}$. As with the square lattice the value of σ depends significantly on $N_{\min}$ and shows no sign of stabilizing.

Attempts to fit the data for the simple cubic lattice with (13) produce meaningless results. The value of RSS is within the 99% confidence interval for Δ ranging from 0.43 down to almost 0. Again, the values of b we find are such that (13) is not a good approximation to (12).

It appears impossible to reliably estimate σ. However, since $\ln(D(N))$ appears to be a concave function of $\ln(N)$, we can give upper bounds on σ. For two dimensions the linear fit (11) with $N_{\min} = 250{,}000$ gives $\sigma = 0.57$. For three dimensions, using $N_{\min} = 160{,}000$ we find $\sigma = 0.85$. These are the values we have taken as upper bounds on σ. They are conservative estimates, and the true value of σ is probably significantly less.

Our analysis has only considered the time required to produce a sequence of walks. In addition, one must also compute the value of the observable on these walks. For some observables, e.g., the end to end distance, this only takes

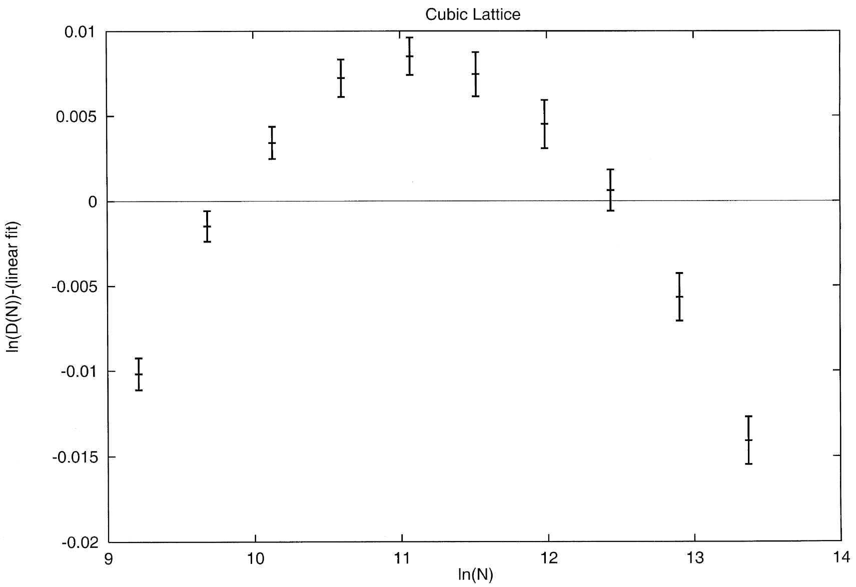

Fig. 5. Simple cubic lattice: The points are the difference between $\ln(D(N))$ and the fit in Fig. 4.

a time of order 1. But for other observables the time required could grow with N faster than the time needed to produce the walks. In this case it might be best to only compute the observable after a fixed number of Monte Carlo iterations. Given estimates of the autocorrelation time of the observable, the time needed to produce an accepted pivot, and the time needed to compute the observable, one could determine the frequency for computing the observable that would give the smallest error bars for a given amount of CPU time.

The previous analysis only tells us how the time required to run the algorithm will grow with N for very large N. From a practical point of view one wants to know how much time our implementation of the algorithm takes and how this time compares with the implementation of the algorithm using a hash table to check for self intersections. Tables III to VI compare the times needed to run one million iterations of the pivot algorithm for three different implementations. (By one million iterations we mean one million attempted pivots, not one million accepted pivots.) One implementation is the implementation given in this paper. Another implementation is the implementation using a hash table to check for self intersections. The third implementation uses the test for self-avoidance given in this paper, but does not use the data structure (i)–(v). This implementation is expected to take a time $O(N)$ per accepted pivot.

Table III. Square Lattice[a]

N	Time	Hash 1	Hash 2	Skip	Therm.
1,000	08.61 s	3.505	4.699	0.824	1
1,600	09.88 s	5.030	6.473	0.869	1
2,500	11.26 s	6.970	8.821	0.932	1
4,000	13.04 s	10.669	14.529	1.025	1
6,400	15.11 s	19.931	26.437	1.163	1
10,000	18.06 s	28.326	34.989	1.241	1
16,000	21.39 s	39.081	47.297	1.530	2
25,000	25.41 s	46.345	55.366	1.671	3
40,000	34.45 s	52.599	61.900	2.242	5
64,000	53.80 s	49.360	57.759	2.869	9
100,000	74.26 s	52.849	61.489	3.342	15
160,000	102.62 s	56.715	65.522	3.630	26
250,000	131.82 s	63.577	72.756	4.103	44
400,000	169.86 s	76.645	85.135	4.634	75
640,000	214.35 s	95.470	97.764	5.194	129
1,000,000	272.33 s	292.659	112.975	5.954	224

[a] The column labeled "Time" gives the time in secs required for 1,000,000 iterations of our implementation of the pivot algorithm on a PC running a 1.33 GHz AMD processor. The columns "Hash 1" and "Hash 2" are for the implementation of the pivot algorithm using a hash table. The two columns use different Hash functions. In these columns we give the ratio of the time required to the time shown in the column "Time." The column "Skip" uses the test for self-avoidance given in this paper, but does not use the data structure given in (i)–(v). It also gives the ratio of the time to the time in column "Time." The final column gives the number of iterations (in millions) carried out to thermalize the walk.

The column labeled "Time" gives the time (in secs) for the implementation in this paper. For the other two implementations, we give the ratio of the time they take to the time required by the algorithm of this paper. The column labeled "Skip" is the implementation which uses the test for self-avoidance given in this paper, but not the data structure (i)–(v). The standard implementation which uses a hash table is shown in the columns labeled "Hash 1" and "Hash 2." The difference between the two columns is only in the hash function used. Letting (x, y) or (x, y, z) be the coordinates of the point, Hash 1 uses a hash table with $L = 5{,}000{,}001$ entries and the hash function

$$(x, y) \to |(17 * x + 290 * y)| \bmod L \tag{14}$$

$$(x, y, z) \to |(17 * x + 290 * y + 4914 * z)| \bmod L \tag{15}$$

Table IV. Square Lattice: The Same Quantities Are Shown as in the Previous Table. The Only Difference Is that These Timing Tests Were Done on a PC Running a 450 MHz Pentium II

N	Time	Hash 1	Hash 2	Skip
1,000	32.82 s	2.456	2.753	1.280
1,600	37.34 s	3.216	3.658	1.372
2,500	43.76 s	4.087	4.692	1.480
4,000	51.40 s	5.352	6.164	1.623
6,400	59.62 s	7.083	8.149	1.785
10,000	68.41 s	9.594	10.908	1.986
16,000	81.29 s	13.680	15.621	2.197
25,000	94.39 s	17.614	19.815	2.499
40,000	111.25 s	23.520	26.075	2.763
64,000	139.73 s	27.786	30.547	3.174
100,000	175.69 s	32.742	35.819	3.666
160,000	227.57 s	37.587	40.850	4.333
250,000	279.65 s	44.607	47.504	4.919
400,000	346.40 s	56.945	58.078	5.719
640,000	420.47 s	79.371	69.534	6.735
1,000,000	512.97 s	378.945	84.269	7.838

Table V. Simple Cubic Lattice: The Same Quantities Are Shown as in the Previous Tables. This Table Is for the Computer Using an AMD 1.33 GHz Processor

N	Time	Hash 1	Hash 2	Skip	Therm.
1,000	53.7 s	2.334	2.661	1.183	1
1,600	70.7 s	2.785	3.140	1.296	1
2,500	93.4 s	3.422	3.961	1.397	1
4,000	128.6 s	4.559	5.672	1.434	1
6,400	165.1 s	6.614	7.682	1.610	1
10,000	234.7 s	7.376	8.495	1.672	1
16,000	315.7 s	8.486	9.687	1.801	1
25,000	451.3 s	8.613	9.783	1.852	2
40,000	693.6 s	8.379	9.457	1.868	3
64,000	1092.1 s	8.746	9.801	1.936	5
100,000	1605.5 s	8.938	9.969	1.964	8
160,000	2531.7 s	8.640	9.635	1.953	12
250,000	3767.4 s	8.331	9.254	1.925	20
400,000	5952.7 s	8.482	9.415	1.925	34
640,000	9857.1 s	7.945	8.799	1.873	56

Table VI. Simple Cubic Lattice: The Same Quantities Are Shown as in the Previous Tables. This Table Is for the Computer Using a Pentium II 450 MHz Processor

N	Time	Hash 1	Hash 2	Skip
1,000	108.5 s	1.956	2.151	1.378
1,600	148.9 s	2.218	2.420	1.482
2,500	196.3 s	2.522	2.760	1.583
4,000	264.8 s	2.846	3.099	1.697
6,400	355.2 s	3.406	3.758	1.804
10,000	479.8 s	4.074	4.473	1.900
16,000	638.8 s	5.086	5.517	1.990
25,000	937.7 s	5.730	6.150	2.129
40,000	1305.0 s	6.289	6.682	2.187
64,000	1932.4 s	6.368	6.739	2.237
100,000	2929.0 s	6.267	6.616	2.256
160,000	4180.2 s	7.012	7.405	2.333
250,000	5994.4 s	7.167	7.543	2.346
400,000	9068.9 s	7.069	7.446	2.352
640,000	14025.0 s	7.099	7.457	2.325

Hash 2 uses a hash table with $L = 5{,}000{,}000$ entries and the hash function

$$(x, y) \to |(47 * x + 2210 * y)| \bmod L \tag{16}$$

$$(x, y, z) \to |(47 * x + 2210 * y + 103824 * z)| \bmod L \tag{17}$$

The tables also give the number of iterations (in millions) we ran to achieve thermalization. These numbers are included to give the reader an idea of how much time is required to equilibriate walks of various lengths. For the longest walks the time needed for thermalization can dominate the total time used in the simulation.

All of the programs used in the timing test were written in C++, compiled with gnu compilers, and run under the Linux operating system. As much as possible we used the same data structures and programming strategies in the three programs. In particular, all the code is written so that lattices other than the square and simple cubic lattices are easily implemented. Walks are stored as arrays of "points," where point is a class, the exact nature of which depends on the lattice. Lattice dependent operations (like the lattice symmetries) are separated out in functions. It is natural to worry that this generality slows down the code significantly. To test this, we have also run timing tests with a Fortran program based closely on code used in ref. 22. This program is specifically written for the simple cubic lattice and uses a hash table. We have found that the Fortran program runs only about 5% faster than our program using a hash table.

The time required depends of course on the type of computer used. One might expect that the ratios of times for the different algorithms do not depend much on the computer. However, we have found that when exactly the same code is run on different PC's these ratios can vary significantly. Thus we have shown the results of timing tests on two computers. Both are PC's. One uses an AMD 1.33 GHz processor, while the other uses a Pentium II 450 MHz processor. (While the code is the same, the two computers are using different versions of the compiler and operating system.) As can be seen from the tables, the implementation of this paper is faster than the implementation using a hash table for all the lengths we have considered. For the longest walks on the square lattice it is faster by roughly a factor of 80, while on the cubic lattice it is faster by roughly a factor of 7 for the longest walks. The column "Skip" gives an idea of how much of the speed-up is a result of the data structure (i)–(v). For example, Table III indicates that for the longest walks on the square lattice, the data structure is responsible for a factor of about 6 in the speed-up.

Timing computer programs is a tricky business. The analysis of the previous section assumes that we have an unlimited amount of memory, all of which can be accessed equally quickly. In a real computer, memory effects can cause the actual performance to lag behind the theoretical performance. The memory needs of our simulations are modest. Even a million step walk in three dimensions can be stored in RAM, and so there is no need to use swap space on the hard drive. However, most CPU's have a small amount of memory, cache, that is much faster than the rest of the RAM. For our implementation, a large part of the time is spent computing the distance between two lattice sites. If the two sites are both stored in the cache, this computation will be faster than if they are not. Thus as N increases, the average time for a single distance computation will increase. On PC's the cache is typically on the order of 256 K. We need 8 bytes to store a site on the walk in two dimensions and 12 bytes in three dimensions. Thus the entire walk will fit into a 256 K cache for walks up to about 32,000 steps in two dimensions and 21,000 steps in three dimensions.

The lower set of data points in Figs. 6 and 7 show log–log plots of the time *per accepted pivot* as a function of N for the square and simple cubic lattices. As a guide to the eye, two lines with slope 0.69 are shown in Fig. 6 and two lines with slope 0.91 in Fig. 7. There is a rather abrupt rise in the time required around the values of N corresponding to the longest walks that will fit in the cache.

To test if memory effects are playing a significant role, we can force the computer to use the same amount of memory for all values of N. We

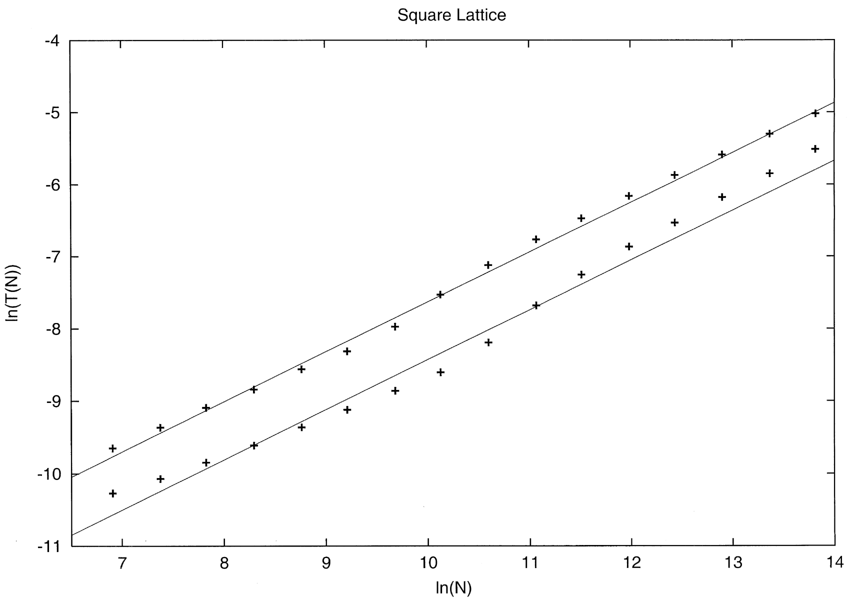

Fig. 6. $T(N)$ is the time required (in secs) per accepted pivot for the square lattice. For the lower sequence of data points, the walk is stored in the usual way. For the upper sequence, the jth site on the walk is stored in array location $1001 * j$ mod 1000001. The lines shown have slope 0.69.

do this by allocating sufficient memory for 1,000,001 steps, regardless of the value of N. We then store $\omega(j)$ in the array location $(1001j)$ mod 1,000,001. This does not necessarily make the memory effects uniform for all lengths, but by comparing the performance with the original implementation, we can see if the memory effects are significant. The resulting data are the upper sequence of points in Figs. 6 and 7. The points are shifted up from the data for the implementation in which $\omega(j)$ is simply stored in the jth array location because every time we access the array we must do a multiplication and a mod operation. The data for this implementation follows the upper line more closely than the previous data followed the lower line. In comparing the slopes of the lines in the two figures with our values of σ, we should keep in mind that there is an additional factor of $\ln(N)$ in the time required coming from the search for i in (5).

We have shown how to implement the pivot algorithm for SAW's so that the time per accepted pivot grows with the number of steps as N^q with $q < 1$. It is difficult to estimate q precisely because of large corrections to the N^q behavior. For the same reason, the precise value of q is irrelevant

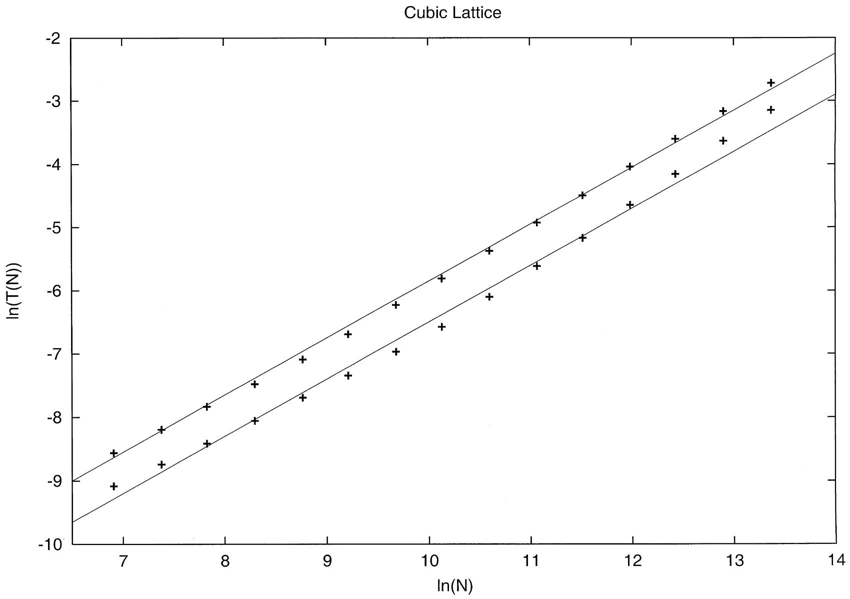

Fig. 7. The time required (in secs) per accepted pivot for the simple cubic lattice. As in Fig. 6, the two curves correspond to two different methods of storing the walk. The lines shown have slope 0.91.

from a practical point of view. Our theoretical analysis and the actual times needed to run the algorithm support the conclusion that the effective q for values of N that can be simulated is below 1.

We have restricted our attention to lattice models in this paper, but our implementation of the pivot algorithm can be carried out for off-lattice (continuum) models as well, provided the length of the steps the walk can take is bounded. It would be interesting to determine the exponent σ in such applications. In principle, our implementation could also be done for SAW's with a nearest neighbor attactive interaction. However, the walks in such a model are not as spread out as they are without this interaction. So the exponent σ may be larger for this model.

ACKNOWLEDGMENTS

The author would like to thank Alan Sokal and the referee for many useful comments. This work was supported by the National Science Foundation (DMS-9970608).

REFERENCES

1. P. G. de Gennes, *Scaling Concepts in Polymer Physics* (Cornell University Press, Ithaca, New York, 1979).
2. J. des Cloizeaux and G. Jannink, *Polymers in Solution: Their Modeling and Structure* (Oxford University Press, Oxford, New York, 1990).
3. M. Lal, "Monte Carlo" computer simulations of chain molecules I, *Molec. Phys.* **17**:57–64 (1969).
4. N. Madras and A. D. Sokal, The pivot algorithm: A highly efficient Monte Carlo method for the self-avoiding walk, *J. Statist. Phys.* **50**:109–186 (1988).
5. B. MacDonald, N. Jan, D. L. Hunter, and M. O. Steinitz, Polymer conformations through "wiggling," *J. Phys. A* **18**:2627–2631 (1985).
6. D. L. Hunter, N. Jan, and B. MacDonald, On the correction-to-scaling exponent of linear-polymers in 2 dimensions, *J. Phys. A* **19**:L543–545 (1986).
7. K. Kelly, D. L. Hunter, and N. Jan, Critical properties of the three-dimensional self-avoiding walk, *J. Phys. A* **20**:5029–5031 (1987).
8. M. Bishop and C. J. Saltiel, Application of the pivot algorithm for investigating the shapes of two-dimensional and 3-dimensional lattice polymers, *J. Chem. Phys.* **88**:6594–6596 (1988).
9. N. Madras, A. Orlitsky, and L. A. Shepp, Monte Carlo generation of self-avoiding walks with fixed endpoints and fixed length, *J. Statist. Phys.* **58**:159–183 (1990).
10. G. Zifferer, Monte Carlo simulation of tetrahedral chains. 1. Very long (athermal) chains by pivot algorithm, *Macromolecules* **23**:3166–3172 (1990).
11. S. Caracciolo, A. Pelissetto, and A. D. Sokal, Universal distance ratios for 2-dimensional self-avoiding walks—corrected conformal—invariance predictions, *J. Phys. A* **23**:L969–L974 (1990).
12. A. J. Chorin, Constrained random-walks and vortex filaments in turbulence theory, *Commun. Math. Phys.* **132**:519–536 (1990).
13. M. Bishop and J. H. R. Clarke, Investigation of the end-to-end distance distribution function for random and self-avoiding walks in two and three dimensions, *J. Chem. Phys.* **94**:3936–3942 (1991).
14. A. J. Barrett, M. Mansfield, and B. C. Benesch, Numerical study of self-avoiding walks on lattices and in the continuum, *Macromolecules* **24**:1615–1621 (1991).
15. S. Caracciolo, G. Ferraro, and A. Pelissetto, Criticality of self-avoiding walks with an excluded infinite needle, *J. Phys. A* **24**:3625–3639 (1991).
16. G. Zifferer, Monte-Carlo simulation of tetrahedral chains. 4. Size and shape of linear and star-branched polymers, *Makromol. Chem.* **192**:1555–1566 (1991).
17. M. Bishop and C. J. Saltiel, The distribution function of the radius of gyration of linear-polymers in 2 and 3 dimensions, *J. Chem. Phys.* **95**:606–607 (1991).
18. M. Bishop and C. J. Saltiel, Structure-function of linear-polymers in the ideal and excluded volume regime, *J. Chem. Phys.* **94**:6920–6923 (1991).
19. N. Eizenberg and J. Klafter, Self-avoiding walks on a simple cubic lattice, *J. Chem. Phys.* **99**:3976–3982 (1993).
20. P. Grassberger, R. Hegger, and L. Schäfer, Self avoiding walks in four dimensions: Logarithmic corrections, *J. Phys. A* **27**:7265–7282 (1994), cond-mat/9409071.
21. S. Caracciolo, G. Parisi, and A. Pelissetto, Random walks with short-range interaction and mean-field behavior, *J. Statist. Phys.* **77**:519-543 (1994).
22. B. Li, N. Madras, and A. D. Sokal, Critical Exponents, hyperscaling and universal amplitude ratios for two- and three-dimensional self-avoiding walks, *J. Statist. Phys.* **80**:661–754 (1995), hep-lat/9409003.

23. G. G. Pereira, Internal structure of polymer chains, *Physica A* **219**:290–304 (1995).
24. N. Eizenberg and J. Klafter, Critical exponents of self-avoiding walks in three dimensions, *Phys. Rev. B* **99**:5078–5081 (1996).
25. P. Grassberger, P. Sutter, and L. Schäfer, Field theoretic and Monte Carlo analysis of the Domb–Joyce model, *J. Phys. A* **30**:7039–7056 (1997).
26. S. Caracciolo, M. S. Causo, and A. Pelissetto, High-precision determination of the critical exponent gamma for self-avoiding walks, *Phys. Rev. E* **57**:1215–1218 (1998), cond-mat/9703250.
27. S. Caracciolo, M. S. Causo, and A. Pelissetto, Monte Carlo results for three-dimensional self-avoiding walks , *Nucl. Phys. Proc. Suppl.* **63**:652–654 (1998), hep-lat/9711051.
28. G. Zifferer, Shape distribution and correlation between size and shape of tetrahedral lattice chains in athermal and theta systems, *J. Chem. Phys.* **109**:3691–3698 (1998).
29. G. Zifferer, Shape distribution and correlation between size and shape of star-branched tetrahedral lattice chains in athermal and theta systems, *J. Chem. Phys.* **110**:4668–4677 (1999).
30. S. Caracciolo, M. S. Causo, and A. Pelissetto, End-to-end distribution function for dilute polymers, *J. Chem. Phys.* **112**:7693–7710 (2000).
31. N. Madras and G. Slade, *The Self-Avoiding Walk* (Birkhäuser, Boston/Basel/Berlin, 1993).
32. A. Sokal, Monte Carlo Methods for the Self-Avoiding Walk, in *Monte Carlo and Molecular Dynamics Simulations in Polymer Science*, Kurt Binder, ed. (Oxford University Press, 1995), hep-lat/9405016.
33. S. D. Stellman, M. Froimowitz, and P. J.Gans, Efficient computation of polymer conformation energy, *J. Comput. Phys.* **7**:178–181 (1971).
34. L. A. Johnson, A. Monge, and R. A. Friesner, A hierarchical algorithm for polymer simulations, *J. Chem. Phys.* **97**:9355–9365 (1992).
35. N. Madras and A. D. Sokal, private communication (2001).