Contents

1 Introduction **2**
1.1 Percolation 2
1.2 First–passage percolation 2
1.3 Subadditive processes and time constants 3

2 Lower bounds **4**
2.1 The method by Janson 4
2.2 An improved method 6
2.3 Numerical results 9

3 Upper bounds **9**
3.1 The method 9
3.2 Examples of bounds 11
3.3 The uniform distribution 11
3.4 The exponential distribution 12

4 Simulation **14**
4.1 Upper confidence limits 14
4.2 Estimates 15

5 Implementation of algorithms **16**
5.1 Lower bounds 16
5.2 Simulation 18

A The Power method **19**

B The program SAWRED **20**
B.1 About the program 20
B.2 Source code 21

C Simulation **27**
C.1 The program 27
C.2 Source code 27

D Upper bound for the exponential distribution **31**

References **33**

1 Introduction

1.1 Percolation

The percolation process was introduced as a mathematical model for the spread of a fluid through a medium by Broadbent and Hammersley [2]. The term fluid has a broad interpretation and can for instance mean a liquid, an epidemic or a particle. The medium is represented by a regular lattice structure, consisting of a connected graph, with a (possible finite) countable set of *vertices (sites)* and *edges (bonds)* joining the vertices.

The fluid is then introduced at some source site(s) and spreads along the bonds. In the *site percolation* problem, the sites are the subject of a random mechanism. Each site has probability p of being open, and probability $q = 1-p$ of being closed. The fluid then spreads from the source site(s) along the bonds to the open sites. A slightly different problem is the *bond percolation* problem, where it is the bonds that are subject to the random mechanism. Each bond has probability p of being open, and probability $q = 1-p$ of being closed. One then tries to make statements about the set of sites reached by the fluid. Under some restrictions on the lattice, every bond problem may be converted to a site problem.

In *first–passage percolation*, introduced by Hammersley and Welsh [5], each bond is associated with a random time variable, representing the time for the fluid to pass the bond.

1.2 First–passage percolation

We will study the time constant for first–passage percolation on the square lattice. The vertices are the points $(x, y) \in \mathbb{Z}^2$ (all integer pairs). The edges are the lines of length 1 joining adjacent points, and with each edge e we associate a random variable X_e. We assume the variables X_e to be non–negative, independent, identically distributed and to have finite mean. X_e may be interpreted as the time for a particle to move along the edge e, or the time needed for an epidemic to spread over e. We will be interested in the time, the first–passage time, for the particle or the epidemic to reach the point $(n, 0)$, or the line $x = n$, starting from the origin.

A walk on the lattice is an alternating sequence $v_0, e_1, v_1, e_2, \ldots, e_n, v_n$ of vertices and edges. The walk is self–avoiding if all vertices are distinct. $(x(\gamma), y(\gamma))$ will be used to denote v_n, the endpoint for some walk γ, and $|\gamma|$ its length. We will need some notation for sets of walks. Γ is the set of all self–avoiding walks starting from the origin. There are some subsets of Γ that will be of use, Γ_n is the subset consisting of all self–avoiding walks of length n, $\Gamma(n, m)$ consists of all self–avoiding walks that ends in (n, m), or more general $\Gamma(R)$ for walks that ends in some non–empty set R. $F(n) = |\Gamma_n|$ will denote the number of self–avoiding walks of length n.

For a given walk γ we define $S_\gamma = \sum_{e \in \gamma} X_e$, the time for the epidemic to spread or the particle to move along the walk γ. We also define $T(R) = \inf_{\gamma \in \Gamma(R)} S_\gamma$, the time for the epidemic or particle to reach R, starting in the origin, and $T_G = \inf_{\gamma \in G} S_\gamma$, the first-passage time over the subset $G \subset \Gamma$.

1.3 Subadditive processes and time constants

A very short summary is included here. See the book by Smythe and Wierman [8] for details and references. In [5], Hammersley and Welsh introduced the concept of *subadditive processes* in order to deal with first-passage percolation. A family of random variables $\{X_{nm}\}$, where $n \leq m$, and n and m are nonnegative integers, are a subadditive process if the following conditions hold, where $g_n = E(X_{0n})$:

- $X_{nm} \leq X_{nk} + X_{km}, \quad (n \leq k \leq m)$
- The process $\{X_{n+1,m+1}\}$ has the same joint distribution as the process $\{X_{nm}\}$.
- $g_n < \infty$ for all n and $\inf_n \frac{g_n}{n} \geq A$ for some constant A.

The *time constant* (which is finite by the third condition) of the process is then defined as

$$\gamma = \inf_{n \geq 1} \frac{g_n}{n}.$$

The first and second conditions imply, through the theory of subadditive functions, that

$$\gamma = \lim_{n \to \infty} \frac{g_n}{n} \leq \frac{g_n}{n}. \tag{1}$$

Kingman proved in [7] that $\lim_{n\to\infty} \frac{X_{0n}}{n}$ exists almost surely, and with some independence condition on the process $\{X_{nm}\}$, that this limit is equal to γ almost surely.

1.3.1 Subadditive processes in first-passage percolation

There are four basic processes in first-passage percolation, two unrestricted processes, a_{mn} and b_{mn}, and two cylinder restricted processes, s_{mn} and t_{mn}, defined as follows, assuming $m < n$.

$a_{mn} \stackrel{d}{=} T(n-m, 0)$ is the first-passage time from $(m, 0)$ to $(n, 0)$.

$b_{mn} \stackrel{d}{=} T(x = n - m)$ is the first-passage time from $(m, 0)$ to the line $x = n$.

Let C_{mn} be the subset of Γ, consisting of walks lying strictly, except for the end-vertex, inside the cylinder $0 \leq x < n - m$. $t_{mn} \stackrel{d}{=} T_{C_{mn}}$ is the first-passage time from $(m, 0)$ to $(n, 0)$ over walks in C_{mn}.

Finally, $s_{mn} \stackrel{d}{=} T_{C_{mn}}(x = n - m)$ is the first-passage time from $(m, 0)$ to the line $x = n$ over walks in C_{mn}.

The point to point processes a_{mn} and t_{mn} are subadditive, but not the point to line processes, although $\psi(n) = E(s_{0n})$ is a subadditive function. The time constant τ of first-passage percolation was originally the time constant of the process a_{mn}. The following results concerning convergence has been proven

$$\frac{a_{0n}}{n} \to \tau, \frac{b_{0n}}{n} \to \tau, \frac{s_{0n}}{n} \to \tau, \frac{t_{0n}}{n} \to \tau \text{ almost surely and in mean.}$$

2 Lower bounds

Time constants may be defined in general directions as well. In [6], Janson defines the set

$$N^* = \{(a,b) \in \mathbb{R}^2 | \lim_{(m,n)\to\infty} P(T(m,n) \leq am + bn) = 0\}.$$

The usual time constant τ may then be defined by (see [6] for details)

$$\tau = \sup_{a\in\mathbb{R}^+} \{(a,0) \in N^*\}.$$

We will use the generating functions

$$F_n(s,t) = \sum_{\gamma\in\Gamma_n} s^{x(\gamma)} t^{y(\gamma)},$$

$$F(p,s,t) = \sum_n p^n F_n(s,t) = \sum_{\gamma\in\Gamma} p^{|\gamma|} s^{x(\gamma)} t^{y(\gamma)}$$

and the moment generating function for X_e

$$\psi(\nu) = E(e^{-\nu X_e})$$

to achieve criterias for a to belong to N^*, and thus lower bounds for the time constant τ.

2.1 The method by Janson

In [6], Janson derives a method for calculating lower bounds for the time constant. The method is based on counting self–avoiding walks of a certain length, and using a criteria on $F_n(s,t)$ and $\psi(\nu)$. Here, we include Janson's results for easy reference.

Lemma 2.1 *For all $\nu > 0$, $P(S_\gamma \leq z) \leq e^{\nu z}(\psi(\nu))^{|\gamma|}$.*

Proof Since $S_\gamma = \sum_{e\in\gamma} X_e$ is a sum of i.i.d . variables

$$E(e^{-\nu S_\gamma}) = \prod_{e\in\gamma} E(e^{-\nu X_e}) = (\psi(\nu))^{|\gamma|}.$$

By Markov's inequality

$$e^{-\nu z} P(S_\gamma \leq z) = e^{-\nu z} P(e^{-S_\gamma} \geq e^{-z}) \leq E(e^{-\nu S_\gamma}) = (\psi(\nu))^{|\gamma|}.$$

□

Theorem 2.2 *If $F\left(\psi(\nu), e^{a\nu}, e^{b\nu}\right) < \infty$ for some $\nu > 0$ then $(a,b) \in N^*$.*

Proof From the definition of $T(m,n)$ and lemma 2.1

$$\begin{aligned} P(T(m,n) \le am+bn) &\le \sum_{\gamma\in\Gamma(m,n)} P(S_\gamma \le am+bn) \\ &\le \sum_{\gamma\in\Gamma(m,n)} e^{\nu(am+bn)} \left(\psi(\nu)\right)^{|\gamma|} \\ &= \sum_{\gamma\in\Gamma(m,n)} e^{a\nu x(\gamma)} e^{b\nu y(\gamma)} \left(\psi(\nu)\right)^{|\gamma|}. \end{aligned}$$

Now we sum over n and m

$$\sum_{n,m} P(T(m,n) \le am+bn) \le \sum_{\gamma\in\Gamma} e^{a\nu x(\gamma)} e^{b\nu y(\gamma)} \left(\psi(\nu)\right)^{|\gamma|}$$

$$= F\left(\psi(\nu), e^{a\nu}, e^{b\nu}\right) < \infty.$$

Therefore, $P(T(m,n) \le am+bn) \to 0$ as $n,m \to \infty$, and $(a,b) \in N^*$ by the definition of N^*. □

Lemma 2.3 *$F_n(s,t)$ is submultiplicative.*

Proof Let γ_1 and γ_2 be two self–avoiding walks, of lengths n and m and $\gamma_1 \circ \gamma_2$ their composition (which may or may not be self–avoiding). Since every self–avoiding walk γ of length $n+m$ can be decomposed as $\gamma_1 \circ \gamma_2$, for some γ_1 and γ_2,

$$\begin{aligned} F_{n+m}(s,t) &= \sum_{\gamma\in\Gamma_{n+m}} s^{x(\gamma)} t^{y(\gamma)} \le \sum_{(\gamma_1,\gamma_2)\in\Gamma_n\times\Gamma_m} s^{x(\gamma_1\circ\gamma_2)} t^{y(\gamma_1\circ\gamma_2)} \\ &= \sum_{\gamma_1\in\Gamma_n} s^{x(\gamma_1)} t^{y(\gamma_2)} \sum_{\gamma_2\in\Gamma_m} s^{x(\gamma_2)} t^{y(\gamma_2)} = F_n(s,t) F_m(s,t). \end{aligned}$$

□

Theorem 2.4 *If $F_n\left(e^{a\nu}, e^{b\nu}\right)^{\frac{1}{n}} < \frac{1}{\psi(\nu)}$ for some $\nu > 0$ and $n \ge 1$, then $(a,b) \in N^*$.*

Proof Since $F_n(s,t)$ is submultiplicative by lemma 2.3, $F_N(s,t) \le (F_n(s,t))^{\frac{N}{n}}$ if $N = nk$ for some positive integer k. Let n be fixed. By the assumptions, there exists an $\epsilon > 0$ such that $F_n\left(e^{a\nu}, e^{b\nu}\right)^{\frac{1}{n}} \le \frac{1}{\psi(\nu)+\epsilon}$. Then

$$\begin{aligned} F\left(\psi(\nu), e^{a\nu}, e^{b\nu}\right) &= \sum_N \left(\psi(\nu)\right)^N F_N\left(e^{a\nu}, e^{b\nu}\right) \\ &\le \sum_N \left(\psi(\nu)\right)^N F_n\left(e^{a\nu}, e^{b\nu}\right)^{\frac{N}{n}} \\ &\le \sum_N \left(\psi(\nu)\right)^N \frac{1}{\left(\psi(\nu)+\epsilon\right)^N} < \infty. \end{aligned}$$

The theorem follows from theorem 2.2. □

Theorem 2.4 allows us to compute lower bounds for τ. We count the number of self–avoiding walks of length n, and remembers in what x–coordinate the walk ends. We then compute the generating function, $F_n(e^{a\nu}, 1)$ to the power n^{-1}, and subtract $\frac{1}{\psi(\nu)}$. If the difference is negative then $\tau \geq a$. This is done by computer, which can easily be programmed to find the best a, for a given n.

In a note in [6], Janson points out that theorem 2.4 may be improved by realizing that F_n may be substituted by a smaller sum, namely the original sum, minus, for each x–coordinate, the walks that start in that direction so that the resulting sum is maximized. This comes from the fact that when joining two walks, for the resulting walk to be self–avoiding there are at most three possible directions for the first step of the second walk.

2.2 An improved method

In [1], Alm uses a version of a method introduced by Wakefield [9] to find upper bounds for the connective constant of self–avoiding walks. It was thought that this method could be used to improve the bounds for the time constant as well. Again, we count the number of self–avoiding walks of a given length n, but this time we also remember the m first and last steps, as well as the m:th last x–coordinate. This gives us a $F(m) \times F(m)$ matrix $\mathbf{B}$, see below. Each element in the matrix is the sum of two polynomials, one in s and one in s^{-1}, both of degree at most $n - m$. The largest eigenvalue of a matrix $\mathbf{A}$ will be denoted by $\lambda_1(\mathbf{A})$. $\mathbf{1}$ will denote a row vector of suitable length. We will use the norm $\|\mathbf{A}\| = \sum_i \sum_j a_{ij} = \mathbf{1A1}'$.

Let m be fixed and let $a_{ij}^{(n)}(k)$ be the number of self–avoiding walks of length n that starts with γ_i and ends with a translation of γ_j, $|\gamma_i| = |\gamma_j| = m$, and has m:th last x–coordinate k. Define the matrix $\mathbf{A}$ by

$$\mathbf{A}^{(n)}(k) = \left(a_{ij}^{(n)}(k)\right), 1 \leq i, j \leq F(m),$$

and the matrix $\mathbf{B}$ by

$$\mathbf{B}^{(n)}(s) = \sum_{k=-n+m}^{n-m} \mathbf{A}^{(n)}(k)s^k = \left(b_{ij}^{(n)}(s)\right), 1 \leq i, j \leq F(m).$$

Now, every self–avoiding walk of length $2n - m$ that starts with γ_i and ends with a translation of γ_j, having m:th last x–coordinate k may be constructed by joining two self–avoiding walks of length n, the first starting with γ_i, and ending with a translation of γ_l, with m:th last x–coordinate r, the second starting with γ_l, and ending with a translation of γ_j, with m:th last x–coordinate $k-r$. Their composition $\gamma_1 \circ \gamma_2$ will then have m:th last x–coordinate k. Therefore

$$a_{ij}^{(2n-m)}(k) \leq \sum_{r=-n+m}^{n-m} \sum_{l=1}^{F(m)} a_{il}^{(n)}(r) a_{lj}^{(n)}(k-r) = \sum_{r=-n+m}^{n-m} \left(\mathbf{A}^{(n)}(r)\mathbf{A}^{(n)}(k-r)\right)_{ij},$$

and, for $\mathbf{B}$,

$$
\begin{aligned}
b_{ij}^{(2n-m)}(s) &= \sum_{k=-2n+2m}^{2n-2m} a_{ij}^{(2n-m)}(k)s^k \\
&\leq \sum_{k=-2n+2m}^{2n-2m} s^k \sum_{r=-n+m}^{n-m} \sum_{l=1}^{F(m)} a_{il}^{(n)}(r)a_{lj}^{(n)}(k-r) \\
&= \sum_{k=-2n+2m}^{2n-2m} \sum_{r=-n+m}^{n-m} \sum_{l=1}^{F(m)} a_{il}^{(n)}(r)s^r a_{lj}^{(n)}(k-r)s^{k-r} \\
&= \sum_{l=1}^{F(m)} \sum_{r=-n+m}^{n-m} a_{il}^{(n)}(r)s^r \left(\sum_{k=-2n+2m}^{2n-2m} a_{lj}^{(n)}(k-r)s^{k-r} \right) \\
&= \sum_{l=1}^{F(m)} b_{il}^{(n)}(s)b_{lj}^{(n)}(s) = \left(\mathbf{B}^{(n)}(s)\mathbf{B}^{(n)}(s)\right)_{ij}.
\end{aligned}
$$

In the same way we get

$$
b_{ij}^{(k(n-m)+m)}(s) \leq \left(\mathbf{B}^{(n)}(s)\right)_{ij}^{k}. \tag{2}
$$

Let the $F(m) \times 1$ column vector $\mathbf{R}^{(m)}$ be defined by the relation

$$
F_{2n-m}(s,1) = \sum_{\gamma \in \Gamma_{2n-m}} s^{x(\gamma)} = \mathbf{1}\mathbf{B}^{(2n-m)}(s)\mathbf{R}^{(m)}.
$$

$\mathbf{R}^{(m)}$ is just a correction vector, depending only on m, with elements from the set $\{s^{-m}, s^{-(m-1)}, \ldots, s^{m-1}, s^m\}$, due to the fact that we are using the m:th last x–coordinate. By (2), we then get, for all $k \geq 1$,

$$
F_{k(n-m)+m}(s,1) = \mathbf{1}\mathbf{B}^{(k(n-m)+m)}(s)\mathbf{R}^{(m)} \leq \mathbf{1}\left(\mathbf{B}^{(n)}(s)\right)^k \mathbf{R}^{(m)}. \tag{3}
$$

Since $s = e^{a\nu}$ and $a\nu > 0$, $s > 1$, and we have $\max_{r \in \mathbf{R}^{(m)}} \mathbf{R}^{(m)} \leq s^m$ and $\min_{r \in \mathbf{R}^{(m)}} \mathbf{R}^{(m)} \geq s^{-m}$, and

$$
\begin{aligned}
\mathbf{1}\left(\mathbf{B}^{(n)}(s)\right)^k \mathbf{R}^{(m)} &\leq \mathbf{1}\left(\mathbf{B}^{(n)}(s)\right)^k \mathbf{1}' s^m = \| \left(\mathbf{B}^{(n)}(s)\right)^k \| s^m \\
\mathbf{1}\left(\mathbf{B}^{(n)}(s)\right)^k \mathbf{R}^{(m)} &\geq \mathbf{1}\left(\mathbf{B}^{(n)}(s)\right)^k \mathbf{1}' s^{-m} = \| \left(\mathbf{B}^{(n)}(s)\right)^k \| s^{-m}.
\end{aligned}
$$

This is equivalent to

$$
\begin{aligned}
\left(\| \left(\mathbf{B}^{(n)}(s)\right)^k \| s^{-m}\right)^{\frac{1}{k(n-m)+m}} &\leq \left(\mathbf{1}\left(\mathbf{B}^{(n)}(s)\right)^k \mathbf{R}^{(m)}\right)^{\frac{1}{k(n-m)+m}} \\
&\leq \left(\| \left(\mathbf{B}^{(n)}(s)\right)^k \| s^{m}\right)^{\frac{1}{k(n-m)+m}}
\end{aligned}
$$

Now let $k \to \infty$. Since

$$\lim_{k\to\infty} \left(s^{-m}\right)^{\frac{1}{k(n-m)+m}} = \lim_{k\to\infty} \left(s^{m}\right)^{\frac{1}{k(n-m)+m}} = 1$$

we get, by the Power method,

$$\lim_{k\to\infty} \left(1 \left(\mathbf{B}^{(n)}(s)\right)^k \mathbf{R}^{(m)}\right)^{\frac{1}{k(n-m)+m}} = \lim_{k\to\infty} \left(\| \left(\mathbf{B}^{(n)}(s)\right)^k \|\right)^{\frac{1}{k(n-m)+m}}$$
$$= \left(\lambda_1 \left(\mathbf{B}^{(n)}(s)\right)\right)^{\frac{1}{n-m}}.$$

And finally, by (3)

$$\left(F_{k(n-m)+m}(s,1)\right)^{\frac{1}{k(n-m)+m}} \leq \left(\lambda_1 \left(\mathbf{B}^{(n)}(s)\right)\right)^{\frac{1}{n-m}}.$$

With the help of theorem 2.4 we have thus proved

Theorem 2.5 *If* $\left(\lambda_1 \left(\mathbf{B}^{(n)}(e^{a\nu})\right)\right)^{\frac{1}{n-m}} < \frac{1}{\psi(\nu)}$ *for some* $\nu > 0$*, then* $(a, 0) \in N^*$.

We thus have a criteria for lower bounds. If the largest eigenvalue of the matrix $\mathbf{B}$ in the point $(e^{a\nu}, 1)$, to the power $(n-m)^{-1}$ is less than $\frac{1}{\psi(\nu)}$ then a is a lower bound for the time constant. For the exponential distribution this criteria gives sharper numerical results than the improved version of theorem 2.4 for all n we reasonable can handle. Already for $n = 4, m = 1$ we improve the bound given in [6].

2.2.1 Reducing B

The matrix $\mathbf{B}$ is actually unnecessary large. We can use a reduced $K(m) \times K(m)$ matrix $\tilde{\mathbf{B}} = (\tilde{b}_{ij})$, where $K(m)$ is the number of equivalence classes of walks of length m. We consider two walks equivalent if one can be mapped on the other by reflection in the x–axis. Every walk except those two that only uses horizontal steps (to $(m, 0)$ and $(-m, 0)$) has exactly one equivalent walk, so that $K(m) = \frac{F(m)}{2} + 1$. Let γ_1 and $\gamma_{K(m)}$ be the walks of length m that goes straight along the x–axis to $(m, 0)$ and $(-m, 0)$. We define $\tilde{b}_{ij} = b_{ij}$ for $j = 1$ or $j = K(m)$ and $\tilde{b}_{ij} = b_{ij} + b_{ij'}$, where γ_j and $\gamma_{j'}$ are equivalent, otherwise. The following theorem shows that we can use $\tilde{\mathbf{B}}$ instead of $\mathbf{B}$.

Theorem 2.6 $\lambda_1(\tilde{\mathbf{B}}) = \lambda_1(\mathbf{B})$

Proof Let $\tilde{\lambda}_1 = \lambda_1(\tilde{\mathbf{B}})$, with corresponding (right) eigenvector $\tilde{\mathbf{h}}$. Define $\mathbf{h}_{F(m)\times 1}$ by

$$h_j = \tilde{h}_s \text{ if } \gamma_j \text{ is equivalent to } \gamma_s.$$

If γ_i is equivalent to γ_r, then

$$\sum_{j=1}^{F(m)} b_{ij} h_j = b_{i1}\tilde{h}_1 + \sum_{s=2}^{K(m)-1} (b_{is} + b_{is'})\tilde{h}_s + b_{iK(m)}\tilde{h}_{K(m)} = \sum_{s=1}^{K(m)} \tilde{b}_{rs}\tilde{h}_s = \tilde{\lambda}_1 \tilde{h}_r,$$

so $\tilde{\lambda}_1$ is an eigenvalue for $\mathbf{B}$. It remains to show that this is the largest eigenvalue, which is easily done by the Power method (appendix A). In the recursion $\mathbf{v}^{(n)} = \frac{\mathbf{B}\mathbf{v}^{(n-1)}}{c_n}$, choose $\mathbf{v}(0) = \mathbf{h}$, and we get $c_n = \lambda_1(\tilde{\mathbf{B}})$ for all $n \geq 1$. Therefore $\lambda_1(\mathbf{B}) = \lim_{n\to\infty} c_n = \lambda_1(\tilde{\mathbf{B}})$. □

2.3 Numerical results

The results are summarized in Tables 1 and 2. In Table 1, the entries for $m = 0$ correspond to the improved version of theorem 2.4. The previous lower bounds, given in [6], where 0.29842 for the exponential distribution, by the improved version of theorem 2.4 with $n = 16$, and 0.24294 for the uniform, by a result not included here, which only uses the moment generating function $\psi(\nu)$. The best lower bounds obtained here are 0.300279 and 0.243665, respectively.

The limitation in n is the time available. When going from n to $n+1$ the time needed increases roughly by a factor 3. In m, it is the internal computer memory that is the limiting factor. $m = 7$ requires more than 128 MB of free memory, and $m = 8$ would require more than 1 GB (approximately 9 times as much). The time needed also increases in m, due to the fact that we must find the largest eigenvalue for a matrix that is roughly three times as wide. However, as can be seen from Tables 1 and 2, there is little gained by doing larger calculations, especially by increasing n.

3 Upper bounds

3.1 The method

There seems to be only one way to obtain upper bounds. All methods are based on the following simple observation.

Lemma 3.1 *If $G_1 \subset G_2$, then $T_{G_1} \geq T_{G_2}$.*

Proof Since $G_1 \subset G_2$,

$$t_{G_1} = \inf_{\gamma\in G_1} S_\gamma \geq \inf_{\gamma\in G_2} S_\gamma = t_{G_2}.$$

□

Let C_{0n} be the subset of Γ, consisting of walks lying strictly, except for the end-vertex, inside the cylinder $0 \leq x < n$.

Theorem 3.2 $\tau \leq \frac{E(T_G(R))}{n}$, *where R is the line $x = n$ and $T_G(R)$ is the first-passage time from the origin to R over walks in $G \subset C_{0n}$.*

Proof By (1) and the lemma,

$$\tau \leq \frac{E(s_{0n})}{n} \leq \frac{E(T_G(R))}{n}.$$

□

Therefore, if we can compute $t = \frac{E(T_G(R))}{n}$ for some set G, t will be an upper bound for the time constant τ.

Table 1: Lower bounds for the time constant, exponential distribution

m / n	0	1	2	3	4	5	6
2	0.286787						
3	0.289423	0.298253					
4	0.292680	0.299266	0.299631				
5	0.293828	0.299473	0.299789				
6	0.295207	0.299685	0.299968	0.300186			
7	0.295900	0.299780	0.300025	0.300201			
8	0.296518	0.299860	0.300077	0.300223	0.300245		
9	0.296934	0.299913	0.300106	0.300233	0.300252		
10	0.297292	0.299955	0.300130	0.300242	0.300258	0.300272	
11	0.297561	0.299988	0.300147	0.300247	0.300261	0.300274	
12	0.297794	0.300015	0.300161	0.300251	0.300264	0.300275	0.300277
13	0.297984	0.300037	0.300172	0.300254	0.300266	0.300276	0.300278
14	0.298150	0.300056	0.300181	0.300257	0.300267	0.300277	0.300278
15	0.298291	0.300072	0.300189	0.300259	0.300268	0.300277	0.300279
16	0.298416	0.300086	0.300196	0.300261	0.300269	0.300277	
17	0.298524	0.300098	0.300202	0.300263	0.300270	0.300278	
18	0.298621	0.300109	0.300205	0.300265	0.300271	0.300278	
19	0.298708	0.300119	0.300207	0.300266	0.300272		
20	0.298786	0.300127	0.300214	0.300267	0.300273		
21	0.298851	0.300135	0.300217	0.300268	0.300274		
22	0.298921	0.300142					

Table 2: Lower bounds for the time constant, uniform distribution

m / n	1	2	3	4	5	6
3	0.242941					
4	0.243325	0.243479				
5	0.243399	0.243518				
6	0.243468	0.243572	0.243643			
7	0.243500	0.243589	0.243647			
8	0.243526	0.243604	0.243653	0.243658		
9	0.243543	0.243613	0.243655	0.243660		
10	0.243557	0.243620	0.243657	0.243661	0.243664	
11	0.243568	0.243625	0.243658	0.243662	0.243665	
12	0.243577	0.243629	0.243659	0.243662	0.243665	0.243665
13	0.243584	0.243633	0.243659	0.243663	0.243665	0.243665
14	0.243591	0.243635	0.243659	0.243663	0.243665	0.243665
15	0.243596	0.243638	0.243659	0.243663	0.243665	0.243665
16	0.243601	0.243640	0.243661	0.243663	0.243665	
17	0.243605	0.243641	0.243661	0.243663		
18	0.243608	0.243643	0.243661	0.243664		
19	0.243614	0.243644				
20	0.243614	0.243646				

3.2 Examples of bounds

For simplicity, G will in this section denote a subset of $\mathbb{Z}^+ \times \mathbb{Z}$, and is to be interpreted as the subset of Γ of self-avoiding paths with vertices from G.

$R = \{(1, y) | y \in \mathbb{Z}\}, \quad G = \{(0, 0), (1, 1)\}$

$$t = E(T_G(R)) = E(X)$$

This gives the upper bounds 0.5 and 1 for the uniform and the exponential distribution, respectively.

$R = \{(1, y) | y \in \mathbb{Z}\}, \quad G = \{(a, b) | a \in \{0, 1\}, b \in \{-1, 0, 1\}\}$

$$t = E(T_G(R)) = E(\min\{X_1, X_2 + X_3, X_4 + X_5\})$$

For the uniform distribution this gives the bound 0.425, and for the exponential distribution we get the bound 0.62963.

$R = \{(1, y) | y \in \mathbb{Z}\}, \quad G = \{(a, b) | a \in \{0, 1\}, b \in \mathbb{Z}\}$

See [5] for full details on this bound, or Section 3.3 for similar bounds.

$$t = E(T_G(R)) = \int_0^\infty Q(x) \left[G(x)\right]^2 dx,$$

where $1 - Q(x)$ is the distribution function for X, and $G(x)$ is the solution to the integral equation

$$G(x) = Q(x) - \int_0^x Q(x - y) G(x - y) Q'(y) dy.$$

For the exponential distribution

$$G(x) = e^{1 - x - e^{-x}}$$

giving the bound 0.59726.

For the uniform distribution, this integral equation seems hard to solve exactly. An approximate solution using power series gives the value 0.41788753, useful not as an upper bound, but as an indication of how good bounds we can get using first-passage times to the line $x = 1$.

3.3 The uniform distribution

Let $U(t) = 1 - F_X(t) = 1 - t$, and $Q_n(t) = P(Y_n > t)$, where Y_n is the first-passage time to the line $x = 1$ over walks with all vertices except the end-vertex of the kind $(0, y), 0 < y \leq n$. Note that $Q_0(t) = P(Y_0 > t) = 1$. Now, for $n > 0$, Y_n is the convolution of X with the minimum of another X and Y_{n-1}, that is

$$Y_n = X_1 + \min(X_2, Y_{n-1}).$$

Table 3: Upper bounds for the time constant, uniform distribution

n	exact	decimal
1	$\frac{17}{40}$	0.425
2	$\frac{50621}{120960}$	0.41849372
3	$\frac{433746463}{1037836800}$	0.41793321
4	$\frac{100093210103}{239520153600}$	0.41789056
5	$\frac{11296442538134659}{27032244535296000}$	0.41788770
6	$\frac{4666120493015797357}{11165972028456960000}$	0.41788753
7	$\frac{1974700453102096679920211}{4725435302089520578560000}$	0.41788753
8	$\frac{397807962861801589313025688 79}{951949846456631050213785600000}$	0.41788753

This gives an expression for $Q_n(t)$ in terms of $Q_{n-1}(t)$

$$
\begin{aligned}
Q_0(t) &= 1 \\
Q_n(t) &= 1 - \int_0^t 1 - (U(t-s)Q_{n-1}(t-s))f_X(s)ds \\
&= U(t) + \int_0^t U(t-s)Q_{n-1}(t-s)ds \\
&= U(t) + \int_0^t U(s)Q_{n-1}(s)ds.
\end{aligned}
$$

The first-passage time T_n to the line $x = 1$ over walks with vertices $(0, y)$, $-n \leq y \leq n$ can now be written as the minimum of one X and two Y_n, all independent,

$$T_n = \min(X, Y_n^{(1)}, Y_n^{(2)}).$$

The expected first-passage time is then

$$E(T_n) = \int_0^t U(t)(Q_n(t))^2 dt.$$

Already for $n = 6$ we get the same value, 0.417888, as the numerical solution to the integral equation in the previous section, indicating that further improvements need other methods. See Table 3 for details.

3.4 The exponential distribution

The nice properties of the exponential distribution makes it easy to compute upper bounds by hand. One starts with only the origin infected. The expected first-passage time, $E(E_1)$, over the area under consideration is then rewritten with the law of total probability conditioning on the first step, giving an expression in terms of expected first-passage times with two infected vertices. All edges now have, by the lack of memory property, the same exponential distribution, and we can continue in this way, successively conditioning on the next

step, until we get explicit expressions for the expected first-passage times. Let $X_{[1]}$ denote the minimum of $\{X_1, \ldots, X_N\}$, and $E_m^{k_m,\ldots,k_1}$ the first-passage time with m infected vertices, and with k_1 as the first edge used, k_2 as the second, and so on. Then,

$$\begin{aligned}
E(E_1) &= \sum_{k_1=1}^{N} E(E_1|X_{k_1} = X_{[1]})P(X_{k_1} = X_{[1]}) \\
&= \sum_{k_1=1}^{N} E(X_{k_1} + E_2^{k_1}|X_{k_1} = X_{[1]})P(X_{k_1} = X_{[1]}) \\
&= P(X_1 = X_{[1]}) \left[\sum_{k_1=1}^{N} E(X_{k_1}|X_{k_1} = X_{[1]}) + E(E_2^{k_1}|X_{k_1} = X_{[1]}) \right] \\
&= \frac{1}{N} + \frac{1}{N} \sum_{k_1=1}^{N} E(E_2^{k_1}) = \frac{1}{N} + \frac{1}{N} \sum_{k_2=1}^{N} \left[\frac{1}{N_{k_1}} + \frac{1}{N_{k_1}} \sum_{k_1=1}^{N_{k_1}} E(E_3^{k_2,k_1}) \right] = \cdots
\end{aligned}$$

There are several ways to simplify the calculations, the easiest using symmetry. Another way is by realizing that if for two areas, the sets of infected vertices are the same, then the expected first-passage times are equal.

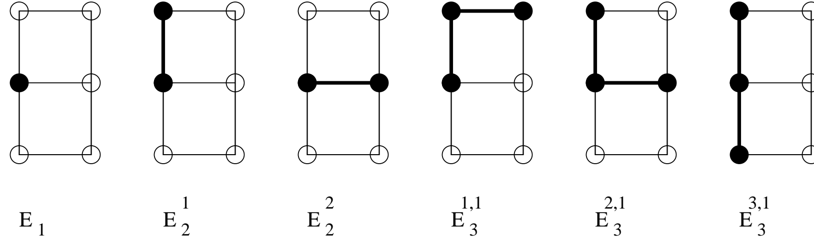

Figure 1: A simple example

3.4.1 Example

A simple example should clarify the details. Consider the area in Figure 1.

$$\begin{aligned}
E(E_1) &= \frac{1}{3}\left(1 + 2E(E_2^1)\right) \\
E(E_2^1) &= \frac{1}{3}\left(1 + E(E_3^{3,1})\right) \\
E(E_3^{3,1}) &= \frac{1}{3} \quad \Longrightarrow \\
E(E_2^1) &= \frac{4}{9} \quad \Longrightarrow \\
E(E_1) &= \frac{17}{27},
\end{aligned}$$

giving an upper bound of 0.62963, which of course is the same as in the second example of Section 3.2.

3.4.2 Lowest upper bound

The largest area considered in this work(and therefore giving the lowest bound) is the rectangle $(0,2),(2,2),(2,-2),(0,-2)$. The expected first-passage time of this rectangle is

$$E(E_1) = \frac{19849958502281}{35286632640000} \leq .56253480, \tag{4}$$

giving an upper bound of 0.562535. The previous lowest bound, 0.587073, was given by Alm (unpublished). The equations, as entered in Maple V, are found in appendix D.

4 Simulation

A simulation study of the cylinder restricted process t_{0n} has been performed. The purpose of this study is to estimate the time constant, and to obtain upper confidence limits for the time constant. A program originally written by Alm was rewritten in C++ to suit the author's need. The program generates a number of independent realizations of the process, and outputs means of first-passage times to each line from $x = 1$ up to some predetermined line $x = n$, as well as the y-coordinate for the first hit on the line $x = n$ for each simulation. To avoid unnecessary and time consuming programming the walks are restricted in y-direction as well. This is not a problem since it is easy to choose the restriction such that the probability that a walk will be restricted in the y-direction is virtually zero, which can in part be confirmed by the data of hitting points on the line $x = n$.

4.1 Upper confidence limits

The first-passage times to the line $x = n$ are used to compute upper confidence limits for the time constant. Since

$$\tau < E\left(\frac{s_{0n}}{n}\right),$$

an upper confidence limit for $E\left(\frac{s_{on}}{n}\right)$ will also be an upper confidence limit for τ, with higher level of confidence. The confidence limits are constructed in the usual way, using the Gaussian 95% quantile.

4.1.1 Numerical results

For the exponential distribution 500 realizations of the process $s_{0,500}$ were generated. The estimated expected first-passage time to the line $x = 500$ was 204.833, with standard deviation 2.77138, giving an 95% upper confidence limit of 0.410074.

Since the simulations for the uniform distribution are less time consuming than for the exponential distribution, a larger sample of 2500 realizations of the process $s_{0,500}$ could be generated. The estimated expected first-passage time to the line $x = 500$ was 158.153, with standard deviation 1.92335, giving an 95% upper confidence limit of 0.316447. This is in fact lower than the previous estimate of τ, 0.32. See for example [4], and the references therein for further information.

Table 4: Summary

	Exponential	Uniform
Upper bound	0.562535	0.417888
Upper confidence limit	0.410074	0.316447
Estimate	0.402	0.312
Lower bound	0.300279	0.243665

4.2 Estimates

The same data used for the confidence limits was also used to estimate the time constant. From the first-passage times we try to extrapolate towards infinity to get estimates. Here we use a simple method. We plot the (by n) normalized first-passage times against $\frac{1}{\sqrt{n}}$, and fit a regression line. The intercept will then estimate the time constant. The choice of $\frac{1}{\sqrt{n}}$ as the predictor is based only on data exploration, and not on theoretical ideas.

The removal of some data points for the lines closest to the origin results in slightly higher estimates than using the whole data set. Further removal and fitting polynomials with higher degree had in most cases only minor or no influence on the estimates. Examples are found in Figures 2 and 3. The chosen estimates are 0.402 for the exponential distribution, and 0.312 for the uniform distribution.

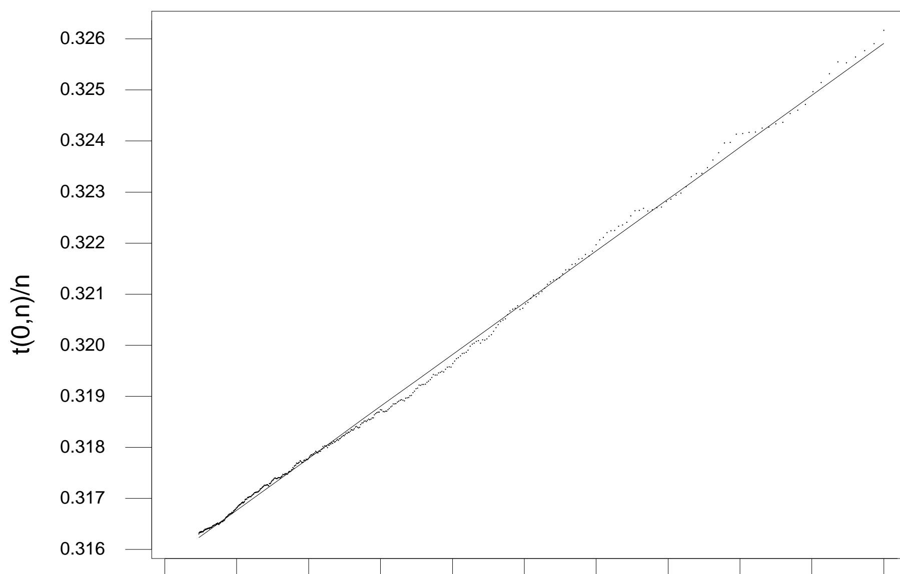

Figure 2: Estimating τ, uniform distribution

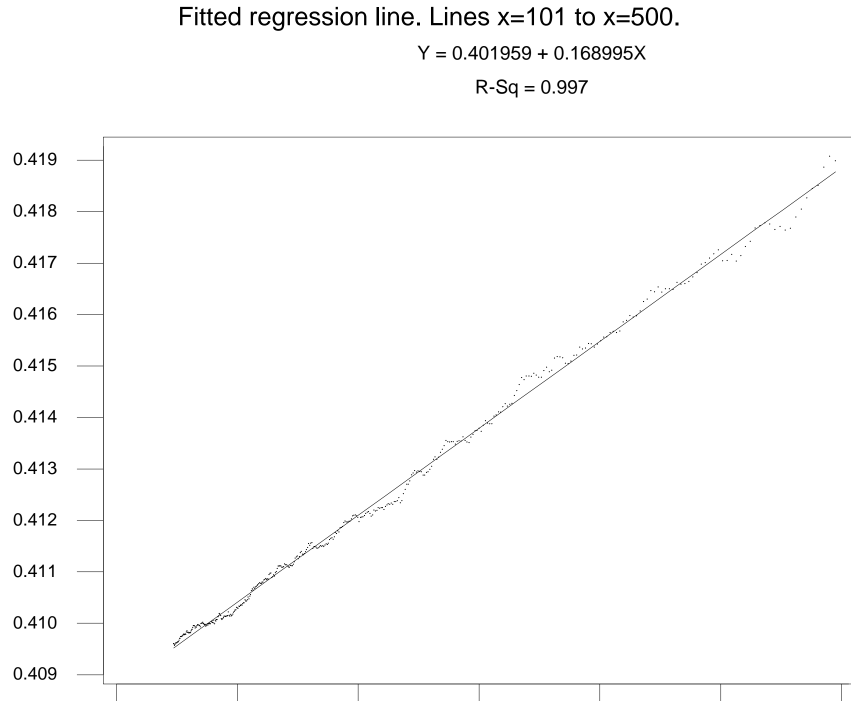

Figure 3: Estimating τ, exponential distribution

5 Implementation of algorithms

5.1 Lower bounds

The general algorithm implemented consists of three steps. First we generate an enumeration of the $K(m)$ self–avoiding walks of length m. This enumeration is then used when generating the $K(m) \times K(m)$ matrix $\tilde{\mathbf{B}}$. Then we look for a good lower bound, with the help of the Power method. The program was written in C++. The time–consuming parts are step 2 (for large n) and step 3 (for large m). More detailed information on the implementation is found in appendix B.

5.1.1 Step 1: Enumeration

The program starts by going right to the point $(n, 0)$ and remembers this walk (after checking that this walk has not been enumerated before). Then it alters the last step and enumerates the resulting walks. But since we are using only one of the walks that are equivalent with respect to reflection in the x–axis, only two walks will have been enumerated at this stage. Now the program alters the second last step , and finds the resulting walks, and so on until all $(K(m))$ walks have been enumerated.

5.1.2 Step 2: Generating $\tilde{\mathbf{B}}$

We start with the first enumerated walk. The walks under study then starts in the m:th position, and we are filling the first row in $\tilde{\mathbf{B}}$. We then proceed as in step 1, but also remembers the m last steps of the walk, as well as the m:th last x–coordinate. When we find a self–avoiding walk we increment the corresponding element in a three dimensional array. The first two indices is the position in the matrix $\tilde{\mathbf{B}}$, which is determined with help of the enumeration, and the third index is the position in the polynomial. Next we iterate through the rest of the enumerated walks until we have generated $\tilde{\mathbf{B}}$.

5.1.3 Step 3: Finding a good lower bound

Now we have the matrix $\tilde{\mathbf{B}}$ and are interested in the highest value of a so that the criteria of theorem 2.5 is fulfilled for some ν. The largest eigenvalue of $\tilde{\mathbf{B}}$ depends on both a and ν so the Power method (implemented with Aitken's Δ^2 method [3]) must be used for every candidate pair a and ν.

We start with values that we know fulfills the criteria. Then a good ν is found and held fixed. a is then increased to the best possible value. This is repeated a few times (typically two or three times). See Figure 4. Good start values of a and ν are critical for fast performance, but this is easily accomplished by extrapolation from previous n or m. Also ν is quite stable for varying n and m.

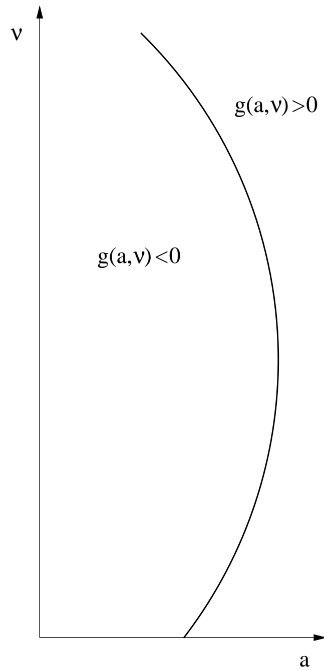

Figure 4: A typical plot of the contour $g(a,\nu) = F_n(e^{a\nu},1)^{\frac{1}{n}} - \frac{1}{\psi(\nu)} = 0$

5.2 Simulation

We start with a list of the vertices that can be infected, that is, they are adjacent to some already infected vertex. In the first step the list contains the points $(1,0),(0,-1)$ and $(1,0)$. The list is sorted with respect to the times of the forthcoming infection. The first vertex in the list (the vertex that are to be infected) is then removed from the list, and the list is expanded with eventual new vertices that are adjacent to the vertex being removed, but not yet infected. Of course only vertices inside the cylinder may be in the list. We then continue removing vertices until the infection has reached line $x = n$, and we are done.

Acknowledgement

I would like to thank my supervisor Sven Erick Alm, especially for all the good ideas that has helped me complete this work.

A The Power method

The Power method allows us to compute the largest eigenvalue of a matrix easily by an iterative algorithm. For a $n \times n$ matrix $\mathbf{A}$ we assume for the n eigenvalues λ_i

$$|\lambda_1| > |\lambda_2| \geq \cdots \geq |\lambda_n| \geq 0$$

and that the eigenvectors $\mathbf{h}_1, \ldots, \mathbf{h}_n$ are linearly independent.

Let $\mathbf{v} = a_1\mathbf{h}_1 + \ldots a_n\mathbf{h}_n$. Then

$$\mathbf{A}\mathbf{v} = \lambda_1 a_1 \mathbf{h}_1 + \lambda_2 a_2 \mathbf{h}_2 + \cdots + \lambda_n a_n \mathbf{h}_n = \lambda_1 (a_1 \mathbf{h}_1 + \frac{\lambda_2}{\lambda_1} a_2 \mathbf{h}_2 + \cdots + \frac{\lambda_n}{\lambda_1} a_n \mathbf{h}_n).$$

Also

$$\mathbf{A}^p \mathbf{v} = \lambda_1^p (a_1 \mathbf{h}_1 + \left(\frac{\lambda_2}{\lambda_1}\right)^p a_2 \mathbf{h}_2 + \cdots + \left(\frac{\lambda_n}{\lambda_1}\right)^p a_n \mathbf{h}_n) \to \lambda_1^p a_1 \mathbf{h}_1 \text{ as } p \to \infty.$$

Further

$$\frac{(\mathbf{A}^{p+1}\mathbf{v})_i}{(\mathbf{A}^p \mathbf{v})_i} \to \lambda_1 \text{ for } i = 1, 2, \ldots, n \text{ as } p \to \infty.$$

This gives us the following algorithm. Let $\mathbf{v}^{(0)}$ be an arbitrary vector, and let

$$\mathbf{v}^{(k)} = \frac{\mathbf{A}\mathbf{v}^{(k-1)}}{c_k}, \text{ where } c_k \text{ is chosen so that } \max_i \left|\mathbf{v}_i^{(k)}\right| = 1.$$

Then $c_k \to \lambda_1$ and $\mathbf{v}^{(k)} \to \mathbf{h}_1$. The convergence is usually very rapid.

B The program SAWRED

The program was written in C++, and currently has one drawback. For each pair of n and m, the program must be recompiled. The author could not find an easy way to allocate memory at run–time for the three–dimensional array $matrix[K(m)][K(m)][2n+2]$.

B.1 About the program

B.1.1 Step 1: Enumeration

First the function *start0* is called. The purpose is to allocate memory for, and initialize, variables needed. Next, we start enumerating by a call to *go0*, which calls *where0* and *next_place* until all walks have been enumerated, signaled by a false(0) return by *where0*. *where0* determines the direction for the next step (right, up, left, down, or back), making sure that the walk is self–avoiding and that we have not taken that step before, and sets the function–pointer *next_place* to the function that performs the appropriate action. There is one function for each direction, for instance *right0* takes one step to the right. If we can not reach the destination with the next step a vertex is added to the walk. Else we check if the equivalent walk have been enumerated. If not, we enumerate this walk.

The state of the current walk is stored in three arrays; *x_array, y_array* and *state*. The two first store the coordinates of the vertices, and the third remembers to what directions we have been from each vertex. We also make use of three variables; *s, t* and *steps*. The length of the current walk is stored in *steps*, and the coordinates of the last vertex of this walk in *s* and *t*. If we have been to all three (or four for the origin) possible directions from a given vertex, we go back along the walk by a call to *back*, unless we are in the origin, in which case all walks have been enumerated, and we signal this by returning from *where0* with a zero. The resulting enumeration is stored in the array $begin[K(m)][m]$.

B.1.2 Step 2: Generating $\tilde{\mathbf{B}}$

There are only small differences from step 1. Instead of starting at the origin we start with in turn each of the enumerated walks, iterating through the first index for *matrix*. The m first elements in *x–array, y–array* and *state* are calculated by *begin*. We also use an array for keeping track of the m last steps, which gives us the second (column) index for *matrix* when a self–avoiding walk is found. The walks with equivalent endings will have the same second index. The m:th last x–coordinate is also calculated and with the help of that the correct element in *matrix* is incremented. *matrix* is the representation of $\tilde{\mathbf{B}}$ used by the program. The functions used are similar (or in some cases the same) as in step 1.

B.1.3 Step 3: Finding a good lower bound

We start by requesting starting values $a1$ and $v1$ for a and ν until we have a pair that fulfills the criteria. The boundary for the area of interest is an arc, and the program makes sure to always be at the right side of that arc. $v1$ is incremented and decremented by some step–size *vstep* until the boundary is reached, giving

us $v2$ and $v3$. The new $v1$ is then $\frac{v2+v3}{2}$. Next $a1$ is incremented by *astep* until the boundary is reached. This is repeated a preset number of times.

Each evaluation requires a calculation of the largest eigenvalue of $\tilde{\mathbf{B}}$. This is done with the Power method, which is straightforward to implement, and is done in the function *power*. This function uses *the_matrix* for calculating numerical values for elements in $\tilde{\mathbf{B}}$, with the help of the representation *matrix*.

B.2 Source code

B.2.1 start0, where0, right0, go0

```
void start0() {
  s=t=steps=current=0;
  x_array = new int[m+1];
  y_array = new int[m+1];
  state = new int[m+1];
  x_array[0]=y_array[0]=state[0]=0; }

int where0() {
  if ((state[current]|8) != state[current] &&
      find(s+1,t)==0){
    next_place=right0;
    walk[steps]=walk_x[steps]=0; // Remember both the walk and the
    return 1; }                  // (x)reflected walk

  else if ((state[current]|4) != state[current] &&
   find(s,t+1)==0){
    next_place=up0;
    walk[steps]=1; walk_x[steps]=3;
    return 1; }

  else if ((state[current]|2) != state[current] &&
   find(s-1,t)==0){
    next_place=left0;
    walk[steps]=walk_x[steps]=2;
    return 1; }

  else if ((state[current]|1) != state[current] &&
   find(s,t-1)==0){
    next_place=down0;
    walk[steps]=3; walk_x[steps]=1;
    return 1; }

  else if (current==0) return 0;
  next_place=back;
  return 1; }

void right0() {
  if (m-steps==1) {
    for (int k=0; k<m; k++)
      begin[n_o_w][k]=walk[k];
    if (!find_red()) n_o_w++; // If one version of the walk has been enum.
    state[current]=(state[current]|8); // dont do this one
  }
  else {
    steps++; s++;
    append(2);
    state[current-1]=(state[current-1]|8); }}

int go0() {
  while (where0())
    next_place();
  return 0; }
```

B.2.2 start, end, where, right

```
void start() {      // Initialize variables and allocate memory
  steps=current=m; // Start in position m
  x_array = new int[n+1];
  y_array = new int[n+1];
```

```
  state = new int[n+1];
  x_array[0]=y_array[0]=0;    // The first vertice is always 0
  state[0]=15;                // The state is set to 1111 in the first
  for (int i=1; i<m+1; i++) { // m-1 vertices. The coordinates is calculated
    state[m]=15;       // with help of begin
    x_array[i]=x_array[i-1] + (begin[index0][i-1]==0)-(begin[index0][i-1]==2);
    y_array[i]=y_array[i-1] + (begin[index0][i-1]==1)-(begin[index0][i-1]==3);
  } // the last state depends on the last step
  state[m]=(2*(begin[index0][m-1]==0)+1*(begin[index0][m-1]==1))
    +8*(begin[index0][m-1]==2)+4*(begin[index0][m-1]==3);
  s=x_array[m]; t=y_array[m]; }

void end() { // Return memory to the free-store
  delete [] x_array;
  delete [] y_array;
  delete [] state; }

int where() { // Find out which way to go next
  if ((state[current]|8) != state[current] && // Done this before?
      find(s+1,t)==0){        // Been there?
    next_place=right;                         // If no, go right!
    return 1; }

  else if ((state[current]|4) != state[current] && // Else try up
   find(s,t+1)==0){
    next_place=up;
    return 1; }

  else if ((state[current]|2) != state[current] && // Else try left
   find(s-1,t)==0){
    next_place=left;
    return 1; }

  else if ((state[current]|1) != state[current] && // Else try down
   find(s,t-1)==0){
    next_place=down;
    return 1; }

  else if (current==m) return 0; // Else, if we're back in pos. m we are done
  for (int i=0; i<m-1; i++)      // e_a keeps track of the m last steps
    e_a[i]=e_a[i+1];
  next_place=back;
  return 1; }

void right() { // Go right
  for (int i=m-1; i>0; i--) // Update the last steps
    e_a[i]=e_a[i-1];        // 0=right, 1=up, 2=left, 3=down
  e_a[0]=0;
  if (n-steps==1) {  // If we can reach the destination
    int ind=index(); // Find the index for this ending
    if (ind>=0) {    // index()=-1 if the walk dont is one of the K(m)
      int x=s+1+m_l_c(ind); // Find m:th last x-coord.
      if (x>0) matrix[index0][ind][x]++; // Count the walk
      else matrix[index0][ind][n+1-x]++;
    }
    else {
      ind=index_x(); // If the walk dont is one of the K(m)
      int x=s+1+m_l_c(ind); // then reflect it in the x_axis
      if (x>0) matrix[index0][ind][x]++; // and count it
      else matrix[index0][ind][n+1-x]++;
    }
    state[current]=(state[current]|8); // Remember that we have been there
    for (int i=0; i<m-1; i++)          // Update the last steps
      e_a[i]=e_a[i+1];
  }
  else { // If dont can reach the destination
    steps++; s++; // add a vertice to the walk
    append(2);
    state[current-1]=(state[current-1]|8); }}
```

B.2.3 back, find, append

```
void back() { // Go back along the walk
  steps--;              // One step back
```

```
  s=x_array[--current]; // Get correct coordinates
  t=y_array[current]; }

int find(int u, int v) { // Search for a node
  for (int i=0; i<=current; i++)   // Look up to current
    if (x_array[i]==u)             // First coord. correct
      if (y_array[i]==v) return 1; // Both correct: return 1
  return 0; }

void append(int new_state) {// Append a vertice to the walk
  x_array[++current]=s;       // The new coordinates
  y_array[current]=t;         // Current points to the new node
  state[current]=new_state; } // Set state
```

B.2.4 index, index_x, m_l_c, find_red

```
int index() { // Finds out what index the ending corresponds to
  int c;
  for (int i=0; i<K_m; i++) { // Iterate through the k(m) walks
    c=1;
    for (int j=0; j<m; j++)   // and the m steps
      if (e_a[m-j-1]!=begin[i][j]) c=0;
    if (c==1)   // If all steps where correct
      return i; // return the index
  }
  return -1; }

int index_x() { // Finds out what index the
  int c;        // (x)reflected ending corresponds to
  for (int i=0; i<K_m; i++) {
    c=1;
    for (int j=0; j<m; j++)
      if ((4-e_a[m-j-1])%4!=begin[i][j]) c=0; // (4-i)%4 gives reflection
    if (c==1)                                 // in the x-axis
      return i;
  }
  return -1; }

int m_l_c(int i) { // X-coord correction
  int r=0, l=0;               // It is the m:th last vertice's
  for (int k=0; k<m; k++) {  // x-coordinate that matters
    if (begin[i][k]==0) r++; // r is the number of rightward steps
    if (begin[i][k]==2) l++; // l is the number of leftward steps
  }
  return l-r; }

int find_red() { // Used when enumerating the m-stepped walks
  int c;         // for finding out if we already have enumerated
  for (int i=0; i<n_o_w; i++) { // the (x) reflected walk
    c=1;
    for (int j=0; j<m; j++)
      if (walk_x[j]!=begin[i][j]) c=0;
    if (c==1)
      return 1; // Return 1 if that is the case
  }
  return 0; }   // and 0 otherwise
```

B.2.5 power, the_matrix, find_p

```
double power(double a, double v){ // Implements the Power method
  int k=1;                        // with Aitken's delta-square
  int p=1;
  double x[K_m];   // x will keep the eigenvector
  double y[K_m];   // So will y, under the calculations
  double u0=0;     // Starting values for the largest eigenvalue
  double u1=0;
  double u, u_out, u_old=0;
  double err;
  double tol=0.00000001;    // The tolerance
  for (int i=0; i<K_m; ++i) // Set starting value for the eigenvector
    x[i]=1;
  int stop=1;
  while (stop) { // Iterate until error<tolerance
    for (int i=0; i<K_m; i++){ // y=M*x
```

```
      y[i]=0;
      for (int j=0; j<K_m; j++)
        y[i]+=the_matrix(i,j,a,v)*x[j];
    }
    u=y[p];
    if (u-2*u1+u0==0) // If the denominator gets too small
      u_out=u;        // don't do the Aitken step
    else
      u_out=u0-pow(u1-u0,2)/(u-2*u1+u0);
    p=find_p(y,K_m);
    for (int i=0; i<K_m; i++)
      x[i]=y[i]/y[p];
    if (k>=4)        // We most do at least 4 iterations
      if ((u_old-u_out)<tol)   // due to the Aitken step
        stop=0;
    k++;
    u0=u1;
    u1=u;
    u_old=u_out;
  }
  return u_out; }

double the_matrix(int i, int j, double a, double v){ // Returns G(e^av)(i,j)
  double f=0;
  for (int k=0; k<n+1; k++)        // k is the third-last x-coord
    f+=matrix[i][j][k]*exp(a*v*k); // matrix2[i][j][k] is the number of walks
  for (int k=n+1; k<2*n+2; k++)    // for k>=n+1 the x-coord is n+1-k
    f+=matrix[i][j][k]*exp(a*v*(n+1-k)); // (negativ values)
  return f; }

int find_p(double *x, int m) { // Finds the right index for sup-norm
  int p=0;                     // p is the least index s.t. max x(i) = x(p)
  double max=0;
  for (int i=0; i<m; i++)
    if (x[i]>max) max=x[i];
  while (x[p]!=max) p++;
  return p; }
```

B.2.6 main

```
int main() { // The program!
  if (K_m!=K[m]) {
    cout << "Wrong K_m! Program will abort. Change and recompile." << endl;
    return 0;
  }

  start0(); // First we enumerate all m-stepped walks
  go0();    // that are unique w.r.t reflection in the x-axis.
  end();    // The walks are stored in begin[K_m][m]
  cout << "n=" << n << endl;
  cout << "m=" << m << endl;
  cout << "K(m)=" << n_o_w << endl;

  for (int i=0; i<K_m; i++) // Set the matrix to zero
    for (int j=0; j<K_m; j++)
      for (int k=0; k<2*n+2; k++)
        matrix[i][j][k]=0;

  for (index0=0; index0<K_m; index0++){
    start(); // For each beginning we count the number of n-stepped walks
    go();    // and generate the K(m)xK(m) matrix
    end();
  }

  // Now we will look for good values of a and v so that the largest
  // eigenvalue of the matrix (evaluated in e^av) matrix2 represents
  // is less than (v+1)^(n-m)
  double astep=0.000001;
  double vstep=0.01;
  double a1, v1, v2, v3;
  cout << "Starting values for a and v:" << endl;
  cin >> a1 >> v1;
  cout << power(a1,v1) << endl;
  while(power(a1,v1)>pow((v1+1),n-m)){
```

```
    cout << "Choose another a:" << endl;
    cin >> a1;
  }
  int donea=0;
  int donev2=0;
  int donev3=0;
  for(int i=1; i<=3; i++){
    v2=v1;
    while (power(a1,v2)<pow((v2+1),n-m)){
      donev2=1;
      v2+=vstep;
    }
    v2+=donev2*vstep;
    v3=v1;
    while (power(a1,v3)<pow((v3+1),n-m)){
      donev3=1;
      v3-=vstep;
    }
    v3+=donev3*vstep;
    v1=(v2+v3)/double(2);
    while (power(a1,v1)<pow((v1+1),n-m)){
      donea=1;
      a1+=astep;
    }
    a1-=donea*astep;
  }
  cout.precision(9);    // So that the result is displayed with more decimals
  cout << "a: " << a1 << endl;
  cout << "For v=" << v1 << endl;
  cout << "eig(G(e^av)) - (v+1)^n-m=" <<power(a1,v1)-pow((v1+1),n-m) << endl;
  cout << power(a1,v1) << endl;
return 1;
}
```

B.2.7 Header file

```
//============================================================================
// Counting self-avoiding walks, header file
// Robert Parviainen, 990530
// Filename: ~/m96rpa/perc/sawred.h
//============================================================================

#ifndef _sawred_
#define _sawred_

// These three variables must be set in compile-time!
const int m=7;    // m steps overlap
const int n=14;   // walks of length n
const int K_m=391;// number of s-a walks of length m
int K[]={0,3,7,19,51,143,391,1087,2959};
// The program will use K[m] to check K_m

int begin[K_m][m];
int e_a[m];
int walk[m];
int walk_x[m];
unsigned long matrix[K_m][K_m][2*n+2];

int s,t;             // current coord.
int current;         // current location in arrays
int steps;           // used steps
int index0;
int* x_array;        // array for the x-coords
int* y_array;        // array for the y-coords
int* state;          // array for the associated flags
int n_o_w;           // reduced number of m-stepped walks (=K(m))
void (*next_place)(void);

void right();   void up();
void left();    void down();
void right0();  void up0();
void left0();   void down0();
void back();    void append(int);
```

```
int index();    int index_x();
int m_l_c(int); int find(int, int);

#endif
```

C Simulation

C.1 The program

The core of the program is a doubly linked ordered list of the vertices that *may* be infected. The class *simulate* is the container class for the list, and the class *infection* is the node class, representing the vertices. The node class has three data fields. Two for the coordinates, and one for the time of the eventual infection. The class *infection* is declared friend to the class *simulate*, so that the latter has access to private data fields of *infection*. A global array of dimensions $(mx + 1)(2my + 1)$ of indicator variables stores information of which vertices that has been infected.

The function *insert* performs ordinary list-insertion, with the addition that nodes are inserted ordered by their data field *time1*. Infected vertices are removed as usual from the list, but before an uninfected vertex is removed a call to *new infection* is made (the vertex is becoming infected). The adjacent vertices are then checked for infection, and for all uninfected vertices we create a new node (which includes the generation of a random time), which is inserted in the list. When we remove the first vertex on the target line $x = mx$, the variable *stop* is set to zero, signalling that we are done.

Each simulation is initialized by the creation of a new *simulate* through its constructor, and started by a call to *new infection* from the origin. The function *remove* is then called until *stop* is set to zero.

C.2 Source code

```
//===========================================================================
// Simulating cylinder first-passage times to lines x=n.
// Original version by S.E. Alm
// This version by Robert Parviainen. 990902.
//===========================================================================

#include <iostream.h>
#include <fstream.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>

const int mx=500;              // max x-value
const int my=5*mx;             // max y-value
int stop=1;                    // stopping indicator
double time2[mx];              // f.p time to lines 1..mx
int num_of_sim=500;            // number of simulations
short infected[mx+1][2*my+1];  // indicators for infected vertices
class infection;               // forward declaration

// container class, a double linked ordered list
class simulate{
public:
  void insert(infection *s);
  void remove();
  void new_infection(int q1,int q2,double t);
  double rndm();
  simulate();
  ~simulate();
  double yhit;
private:
  infection *last;
  infection *first;
  int intr;
};
```

```
// the list elements class
class infection{
  friend simulate;    // allow use of private members
  infection();        // in class simulate
  infection(int i, int j, double t);
private:
  int p1;
  int p2;
  double time1;
  infection *prev;
  infection *next;
};

infection::infection(){ // constructor
  prev=0; next=0;
}

infection::infection(int i, int j, double t){ // constructor
  prev=0; next=0; p1=i; p2=j; time1=t;
}


// simulate methods
void simulate::insert(infection *s){ // insert vertice s
  infection *t=last;
  while (t!=0 && t->time1 > s->time1) t=t->prev; // find position for s
  if (t==0) { // s is first
    s->prev=0;
    s->next=first;
    if (first!=0) first->prev=s;
    first=s; }
  else { // s is not first
    s->prev=t;
    s->next=t->next;
    t->next=s;
  }
  if (s->next==0) // if s is last
    last=s;
  else s->next->prev=s;
}

void simulate::new_infection(int q1,int q2,double t){ // find new vertices
  int i=q1,j=q2;                                       // to infect
  infection *s;
  j++; // look up
  if (!infected[i][j+my]){
    s=new infection(i,j,t+rndm()); // create a new vertice
insert(s);       // and insert it in the list
  }
  j-=2; // look down
  if (!infected[i][j+my]){
    s=new infection(i,j,t+rndm());
    insert(s);
  }
  j++; i++; // look right
  if (!infected[i][j+my]){
    s=new infection(i,j,t+rndm());
    insert(s);
  }
  if (i-1>0) { // if possible
    i-=2;      // look left
    if (!infected[i][j+my]){
  s=new infection(i,j,t+rndm());
      insert(s);
    }
  }
}

double simulate::rndm(){ // returns random number
  double nb=drand48();   // from the exponetial distribution
  if (nb==0)             // avoid problems with ln(0)
    nb=10000;
  else
    nb=-log(nb);
```

```
  return nb;
}

void simulate::remove(){ // remove the first vertice
  infection *t;
  t=first;
  first=first->next;
  first->prev=0;

  if (!infected[t->p1][t->p2+my]){ // if not infected
    infected[t->p1][t->p2+my]=1;   // infect
    if (t->p1==mx) {
      stop=0;
      time2[mx-1]=t->time1;
  yhit=t->p2;
      }
else if (t->p1>0 && t->p1<mx)
  if (time2[t->p1-1]>t->time1) time2[t->p1-1]=t->time1;
    new_infection(t->p1,t->p2,t->time1);
  }
  delete t;
}

simulate::simulate(){ // constructor, sets up a new simulation
  last=0;first=0;intr=1;
  infected[0][my]=1;
  for(int i=0;i<=mx;i++) time2[i]=1000000;
}

simulate::~simulate(){
  infection *t;
  t=first;
  while (t->next) {
delete t;
t=t->next;
  }
}

// main program
int main(){
  double sumx[mx];
  double sumxx[mx];
  srand48(time(0));  // initialize random number generator
  ofstream fout1("fppexp.txt");
  ofstream fout2("fppexphits.txt");
  for(int i=0;i<=mx;i++) {
sumx[i]=0;
sumxx[i]=0;
  }
  for(int k=0;k<num_of_sim;k++){
    stop=1;
    for(int i=0;i<=mx;i++)  // reset infected
      for(int j=0;j<=2*my;j++) infected[i][j]=0;

    simulate sim; // create new simulation through constructor
    sim.new_infection(0,0,0);  // start simulation
    while (stop) sim.remove(); // continue until stop=0
for (int m=0;m<mx;m++) {
  sumx[m]+=time2[m];
  sumxx[m]+=time2[m]*time2[m];
}
fout2 << sim.yhit << endl;
cout << k+1 << " " << endl;
  }
  // compute statistics
  for (int k=0;k<mx;k++) {
int m=k+1;
double std=sqrt((sumxx[k]-sumx[k]*sumx[k]/num_of_sim)/(num_of_sim-1));
double std2=std/(m*sqrt(num_of_sim));
double x=sumx[k]/num_of_sim;
double xx=x/m;
double bound=xx+1.6445*std2;

cout  << m << " " << x << " " << xx << " "
```

```
  << std << " " << std2 << " " << bound << endl;
fout1 << m << " " << x << " " << xx << " "
  << std << " " << std2 << " " << bound << endl;

  }
  return 0;
}
```

D Upper bound for the exponential distribution

```
> restart;
> KKK:=1/5:
> JJJ:=(1+3*KKK)/7:
> CCC:=(1+2*KKK)/6:
> III:=(1+4*CCC)/7:
> HHH:=(1+3*KKK)/7:
> EEE:=(1+2*HHH+3*CCC)/8:
> DDD:=(1+4*EEE+3*III)/9:
> UU:=(1+3*KKK)/7:
> GGG:=(1+6*UU)/9:
> FFF:=(1+3*CCC+2*UU)/8:
> ZZ:=(1+2*UU+2*HHH)/7:
> QQ:=(1+UU+2*CCC)/6:
> AAA:=(1+2*QQ+2*III+FFF)/7:
> RR:=(1+CCC+KKK)/6:
> BBB:=(1+III+2*RR+CCC)/7:
> VV:=(1+2*QQ+EEE+ZZ)/6:
> YY:=(1+4*ZZ+GGG)/7:
> XX:=(1+2*EEE+2*FFF+2*ZZ)/8:
> TT:=(1+3*QQ+2*FFF+GGG)/8:
> SS:=(1+3*RR+FFF+UU)/8:
> PP:=(1+SS+2*BBB+AAA+QQ)/7:
> NN:=(1+2*VV+2*AAA+DDD+XX)/7:
> GG:=(1+2*RR+KKK)/7:
> MM:=(1+2*GG+2*BBB+CCC)/8:
> FF:=(1+2*RR+QQ)/6:
> LL:=(1+2*FF+AAA+2*BBB)/7:
> WW:=(1+3*RR+EEE+HHH)/8:
> OO:=(1+2*WW+3*BBB+DDD+EEE)/9:
> KK:=(1+2*WW+2*SS+XX+ZZ)/8:
> JJ:=(1+2*VV+TT+XX+YY)/6:
> II:=(1+3*GG+2*SS+UU)/9:
> HH:=(1+3*FF+2*SS+TT)/8:
> EE:=(1+II+2*MM+2*PP+QQ)/8:
> W:=(1+KK+OO+2*PP+VV+NN)/7:
> BB:=(1+2*TT+4*AAA)/7:
> LLL:=(1+2*BBB+2*RR)/7:
> DD:=(1+MM+GG+2*LLL+RR)/8:
> CC:=(1+FF+GG)/5:
> Y:=(1+2*FF+WW+VV)/6:
> AA:=(1+2*OO+2*WW+3*LLL)/9:
> Z:=(1+LL+FF+2*LLL+PP)/7:
> X:=(1+2*JJ+2*NN+BB)/5:
> V:=(1+HH+2*LL+2*PP+BB)/7:
> U:=(1+2*Y+2*LL+NN+OO)/7:
> T:=(1+2*CC+LL+MM)/6:
```

```
> S:=(1+2*Y+HH+JJ+KK)/6:
> R:=(1+3*CC+HH+II)/7:
> Q:=(1+2*CC+Y)/5:
> P:=(1+4*DD+2*GG)/9:
> OOO:=(1+FF+GG)/5:
> N:=(1+EE+2*DD+FF+2*Z)/8:
> M:=(1+T+CC+Z+DD)/6:
> L:=(1+4*Z+2*V)/7:
> K:=(1+U+Y+2*Z+AA+W)/7:
> J:=(1+S+U+V+W+X)/5:
> IIII:=(1+R+2*T+V+EE)/6:
> G:=(1+2*Q+R+S)/5:
> H:=(1+2*Q+2*T+U)/6:
> F:=(1+2*M+OOO+N+P)/7:
> E:=(1+IIII+2*M+L+N)/6:
> DDDD:=(1+2*J+2*K+L)/5:
> C:=(1+G+H+IIII+J)/4:
> B:=(1+2*E+2*F)/5:
> A1:=(1+H+Q+2*M+K)/6:
> A:=(1+A1+C+DDDD+E)/4:
> bound:=(1+2*A+B)/(3*2);
```

$$bound := \frac{19849958502281}{35286632640000}$$

```
> evalf(bound);
```

$$.5625347906$$

References

[1] S. E. Alm. Upper bounds for the connective constant of self–avoiding walks. *Combinatorics, Probability and Computing*, 2:115–136, 1993.

[2] S. R. Broadbent and J. M. Hammersley. Percolation processes I. Crystals and mazes. *Proceedings of the Cambridge Philosophical Society*, 53:629–641, 1957.

[3] R. L. Burden and J. D. Faires. *Numerical analysis.* Brooks/Cole, 6 edition, 1997.

[4] S. Finch. *Favorite mathematical constants.* http://www.mathsoft.com/asolve/constant/constant.html.

[5] J. M. Hammersley and J. D. Welsh. First–passage percolation, subadditive processes, stochastic networks, and generalized renewal theory. In *Bernoulli-Bayes-Laplace Anniversary Volume*, pages 61–110. Springer-Verlag, 1965.

[6] S. Janson. An upper bound for the velocity of first–passage percolation. *Journal of Applied Probability*, 18:256–262, 1981.

[7] J. F. C. Kingman. The ergodic theory of subadditive stochastic processes. *Journal of Royal Statistical Society Series B*, 30:499–510, 1968.

[8] R. T. Smythe and J. C. Wierman. *First-Passage Percolation on the Square Lattice.* Lecture notes in Mathematics 671. Springer-Verlag, 1978.

[9] A. J. Wakefield. Statistics of the simple cubic lattice. II. *Proceedings of the Cambridge Philosophical Society*, 47:799–810, 1951.