ST3247 Assignment: Self-Avoiding Random Walks (Draft)

Alyxia Seah

March 29, 2025

1 Introduction

The exact value of the connective constant μ of the square lattice $\mathbb{Z}^2$ remains an open problem in mathematics. We define c_L as the number of random walks on $\mathbb{Z}^2$ of length L (assumed to be centered at $(0,0)$), and we have $\mu = \lim_{N\to\infty} c_N^{1/N}$. In other words, $c_L \approx \mu^L$ for large values of L. We aim to employ Monte Carlo methods to estimate the value of μ, or equivalently, to estimate the value $\lambda = \log \mu = \lim_{N\to\infty} \log c_N^{1/N}$.

2 Basic Deterministic Methods

We first construct a simple algorithm that computes c_L, the number of self-avoiding random walks of length L, using recursive backtracking. This algorithm performs poorly for large L as c_L grows exponentially; nonetheless, we are able to compute exact values of c_L for $L \leq 20$ and verify them with known values (OEIS A001411).

In this algorithm, the function `backtrack(x,y,L)` backtracks `L` steps from a given starting coordinate `(x,y)`, and recursively calls itself until we reach $\texttt{L} = 0$, in which case we have found a self-avoiding walk. We begin the algorithm by calling `backtrack` on $(0,0)$.

Python code for the algorithm can be found on my GitHub. Some small optimizations for this code include the use of a `Set` object to store all previously visited vertices, instead of a `List`, for $O(1)$ lookup of a possible self-intersection instead of $O(N)$.

3 Basic Monte Carlo I

Next, we run a simple Monte Carlo simulation to estimate the probability P_L that a random walk of a given length L is self-avoiding. This estimation is done naively; in other words, we simply generate many random walks of length L and compute the proportion of self-avoiding walks to find an estimate of P_L.

The function `generate_random_walk(L)` generates random walks while checking if they self-intersect at each step. If an intersection is found, we return the step at which this intersection occurred (although this exact number is not required, we return the value out of convenience and to exit the function). If the random walk reaches the full length L without any self-intersection, it is self-avoiding; we estimate P_L as the proportion of self-avoiding walks generated.

Python code for this naive simulation can be found here. Running this simulation for walks of length $L = 10$ over 100000 trials yields an estimate of $P_{10} \approx 0.04186$. This agrees with the fact that the true value is $P_{10} = \frac{c_{10}}{4^{10}} \approx 0.04206$, since $c_{10} = 44100$, and the total number of random walks of length $L = 10$ is $4^{10} = 1048576$.

4 Basic Monte Carlo II

Using a different function to generate walks, we run a more sophisticated Monte Carlo algorithm to obtain a better approximation of P_L for larger values of L.

The function `generate_self_avoiding_walk(L)` ensures that a walk cannot intersect itself, by using a `filter` to identify steps which would not lead to a previously visited coordinate. If there is no possible step it can take, the path will remain at its current coordinate.

Let x_L denote a self-avoiding random walk of length L, and let X_L denote the set of all self-avoiding random walks of length L. Let z_L denote any random walk (not necessarily self-avoiding), and let Z_L denote the set of all random walks of length L. We use self-normalized importance sampling to estimate the value of c_L, which is the normalization constant for the uniform distribution $\mathbb{P}_L(x) = \frac{1}{c_L}\mathbb{1}_{z_L \in X_L}$ of all self-avoiding random walks $x_L \in X_L$ (since $\mathbb{P}_L(x)$ is a distribution, we must have $\int_{x \in X} \mathbb{P}_L(x)dx = 1$).

In particular, instead of sampling from a uniform distribution of all random walks of length L as done in Section 3, we are now sampling from a proposal distribution of random walks of length L that do not take steps which would lead to self-intersections.

As such, we implement a weight function $W(x_L)$ resembling that of the Rosenbluth algorithm [Tab15]. The weight of a particular self-avoiding walk x_L is inversely proportional to the probability that it is sampled from $P_L(z)$, the uniform distribution of all random walks of length L. Let $\sigma_i(x_L)$ be the number of possible steps a walk x_L can take at step i. Intuitively, the probability that we sample a walk from the original uniform distribution $P_L(z)$ of all random walks is higher when we have more degrees of freedom. As such, we define the weights $W(x_L) = \prod_{i=1}^{L} \sigma_i(x_L)$ and return the value $W(x_L)$ when we call `generate_self_avoiding_walk(L)`.

Finally, we are ready to compute our estimate of c_L using some number of trials t by taking the weighted sum $\sum_t (W(x_L)\mathbb{1}_{x_L \in X_L})$ for all random walks x_L generated by our function.

Python code for this section is available here. Running our estimator for random walks of length $L = 10$ over 100000 trials, we obtain an estimate of $c_{10} \approx 44158.02804$, which is extremely close to the known true value of $c_{10} = 44100$. For length $L = 20$ over 100000 trials, we obtain an estimate of $c_{20} \approx 902233353$, which is also very close to the known true value of $c_{20} = 897697164$. As such, importance sampling allows us to obtain highly accurate estimates of c_L for larger L.

References

[Tab15] Mandana Tabrizi. *Rosenbluth Algorithm Studies of Self-avoiding Walks.* York University, 2015.