Vanier College - Comprehensive Assessment

May 24, 2021

Markov Chain Monte Carlo Methods in Cryptography

Peter Veroutis

Elise Fajardo

Presented to

Christian Stahn

Ivan Ivanov

Linear Algebra II

Probability and Statistics

Contents

Abstract

The purpose of this paper is to exhibit the applicability of the Metropolis algorithm, an algorithm in the family of Monte Carlo Markov Chains methods. Conventionally, Monte Carlo Markov Chain methods are used to sample from the equilibrium steady-state distribution of a Markov Chain. This paper first provides a brief overview of Markov chains to a reader unfamiliar with the topic, followed by a discussion of the Metropolis algorithm. It is later shown how such an algorithm can be deployed in cryptography and can decrypt encoded messages. The effectiveness of the Metropolis algorithm is then analyzed and the results demonstrate that an optimal guess of the initial key significantly impacts the success of the procedure. In addition, the methods are shown to decode longer strings of text more successfully since in the longer cases the algorithm is given more information. It is concluded that while the algorithm isn't used for its original traditional purpose it nevertheless solves the decryption problem and effectively decodes messages by optimization the key function through a random walk.

1 Introduction and Overview

1.1 Markov Chains

To attain a comprehensive understanding of the Markov Chain Monte Carlo and the Metropolis algorithm, used in cryptography, it is essential to have a strong understanding of Markov Chains.

1.1.1 Markov Chains

A Markov chain may be defined as a memoryless system that follows probabilistic rules when transitioning between different states. It consists of a process containing a set of states $S = \{s_1, s_2, ..., s_n\}$ that begins with an initial state s_j and moves towards a different state s_i in a step with a probability p_{ij}, referred to as a **transition probability**.

One may note that, with a given probability p_{jj}, the system can also remain in its current state at the next step. The chain obeys the Markovian property, where the probability of moving to the future states of the process depends solely on the present state. Indeed, the probability P of going to the state S_{n+1} is given by

$$P[S_{n+1}|S_0 = s_0, S_1 = s_1, ...S_n = s_n] = P[S_{n+1}|S_n = s_i]$$

The transition probabilities $P(S_n)$ may be illustrated through a transition matrix P, or by a graph with probabilities on the edges.

1.2 The Random Walk

We may consider the process of the random walk for a better understanding of the Markov chain, a special type of random walk. A **random walk** can generally be described as a process consisting of random steps on a given space.

Consider the situation of an object starting at time and position zero, $X(0) = 0$, and moving one single unit to the right or to the left for every time step. With each direction is associated a probability p and $q = 1 - p$ respectively, where $p = q = 1/2$. This particular example is the simplest case of a random walk and often referred to as a **Standard Random Walk**.

1.3 Stochastic Matrices

The transitions of a Markov Chain can be described through a transition probability matrix that contains numerous fundamental properties interconnected to probability theory.

While conventions vary, in this paper, the direction of the transition in the probability matrix will be denoted from columns to rows, where the column of the stochastic matrix represents the current state and the row indicates the new state.

One may note the following properties of any stochastic matrix:

- It is always square
- It contains entries of non-negative numbers representing the transition probabilities

Considering the decided convention of initial states represented in the columns j and final states, in rows i

- The sum of the entries of each column of this square array sum to one.

Note: For the sum of the entries of a matrix whose transition probability p_{ij} denotes transitions from $i \to j$, the previous property will apply to the rows, rather than the columns.

One can also consider that a column of such stochastic matrix is a probability vector $\vec{x}$ as it follows the condition that its entries are both non negative and sum to 1.

1.3.1 Periodic versus Aperiodic Chains

Every state present in a Markov chain has a **period** defined as the greatest common denominator of the number of steps necessary to return to that state, when the starting point is that state.

Concerning entire chains, we now refer to the property of being either periodic or aperiodic. Chains defined as **aperiodic** have a period with the strict value of 1. If the value of the period is different from 1, it is called **periodic**.

1.3.2 Irreducible Chains

A Markov chain is called **irreducible** if it is always possible to go from any state to any other state, no matter the number of steps required for this to occur.

1.4 Transition Graphs

A Markov chain may not only be represented through a matrix, but also through a **Transition Diagram**. A Markov chain's transition graph is a directed graph, that is a set of vertices representing the different states of the Markov chain and arrows representing transitions between states S. The arrows of a transition graph are only present for states S_n where the transition probability of going to the next state S_{n+1} is positive. That is $P(S_n \to S_{n+1}) > 0$.

1.5 Regular Markov Chains

A Markov chain is defined as regular if some power of its transition matrix have only strictly positive elements. One can observe that a chain that is $aperiodic + irreducible \Leftrightarrow regular$.

Let P be the transition matrix for a regular chain. Then, as $n \to \infty$, the powers P^n approach a limiting matrix P^∞ with all columns the same vector $\vec{x}\infty$.

For any probability vector $\vec{x}_0$, the state $P^k\vec{x_0}$ of the system at the time k converges to $\vec{x}_\infty$. This vector is called a **fixed probability vector**. This probability vector is also referred to as a **stationary distribution** and is one that does not change with time. That is, as this state takes any given number of steps n, it will remain unchanged. Therefore, for the distribution $\vec{x}_\infty$, given any number of steps n, $P^n\vec{x}_\infty = \vec{x}_\infty$.

1.5.1 Perron-Frobenius Theorem

One may note a theorem that arises from regular Markov chains, the **Perron-Frobenius Theorem**.

The theorem states:

Let P be a regular stochastic matrix, then there exists

- an eigenvalue $\lambda = 1$ that has algebraic and geometric multiplicity 1
- a corresponding eigenvector with positive components

An important consequence of this theorem is that the eigenvector corresponding to the eigenvalue $\lambda = 1$ is the steady-state vector of the Markov chain.

Indeed, the largest eigenvalue that will appear in a stochastic matrix for a regular Markov chain is 1. By writing the long-term solution of the system by diagonalizing P, the consequence for the steady-state vector becomes clear.

We obtain,

$$\vec{x}_k = c_1\lambda_1^k\vec{v}_1 + c_2\lambda_2^k\vec{v}_2 + ...c_n\lambda_n^k\vec{v}_n \tag{1}$$

Where $\lambda_1 = 1$ and $\lambda_2, ...\lambda_n < 1$. As we take $k \to \infty$, $\vec{x}_k \to \vec{v_1}$, all terms $\neq c_1\lambda_1^k\vec{v}_1$ will approach zero. The long-term solution will thus approach the direction of the eigenvector for $\lambda = 1$, which will correspond to the steady-state vector.

1.6 Detailed Balance in Markov Chains

An important property that may arise in Markov chains is **detailed balance**. A Markov chain is said to have detailed balance if it satisfies detailed balance equations of the following form.

Let π be the stationary distribution of a chain possessing detailed balance, then,

$$P_{ij}\pi_j = P_{ji}\pi_i \tag{2}$$

That is, with a stationary distribution π, the amount given by $P_{ij}\pi_j$, that is the quantity flowing from state $j \to i$ on a certain time step will be equivalent to the quantity, given by $P_{ji}\pi_i$, flowing from state $i \to j$. Thus, such system not only remains unchanged through time as it is in its stationary distribution, but follows the supplementary condition that the total probability leaving a state must equal the sum of the probability entering the state.

2 The Markov Chain Monte Carlo Algorithm

After our brief exploration of Markov chains, we may now address the Markov Chain Monte Carlo methods and more specifically, their usefulness and the motivation for using such methods. This discussion will provide some context to the MCMC method, the Metropolis Algorithm that is put to use in the decryption problem.

2.1 The Markov Chain Monte Carlo Algorithm

The Markov Chain Monte Carlo is a set of algorithms that samples probability distributions. The methods included in the MCMC provide a way to simulate values, and use those simulated values to conduct further analyses. More often than not, in real applications, the distribution of a sample of values is unknown, or a distribution's normalization constant Z is unknown. Consequently, the MCMC methods are incredibly practical and for this reason put to use in a wide range of areas of study.

For some background, one may note that the Markov Chain Monte Carlo were developed by physicists who wanted to model the probabilistic behavior of ensembles of atomic particles. As they could not do this through analysis, they decided to attempt using a simulation in this situation where they did not know the value of Z. They solved the problem by setting up a Markov Chain with desired distribution to be the stationary distribution and simulating until the stationary state was reached.

2.2 A Simple Monte Carlo Simulation

Suppose we are interested in a discrete random variable X that can take the values $1, 2, 6$ each with probability $1/3$. We may use a Monte-Carlo method to approximate the distribution. Such computation may be performed in python. The python code is executed as follows:

```
import numpy as np
from numpy.random import default_rng
rng = default_rng()
from collections import Counter
n_samples = 10000
samples = [rng.choice([1,2,6]) for r in range(n_samples)]
c = Counter(samples)
np.array([c[1], c[2], c[6]])/n_samples
array([0.33302, 0.33741, 0.32957])
```

In this simulation, we conducted an experiment as if the probability distribution was unknown. We sampled the distribution 10 000 times and observed the number of times each event occured. Then, dividing by the total number of samples allowed to approximate the probability distribution. We found the probability for each event to be approximately $1/3$, in accordance with the function rng.choice.

3 The Metropolis Algorithm

Before exploring the use of Markov Chain Monte Carlo methods in cryptography, it is beneficial consider the Metropolis algorithm along with a preliminary example, following a demonstration of why such an algorithm yields the equilibrium steady-state probability vector of a Markov Chain.

3.1 The Metropolis Algorithm

The Metropolis algorithm can be deployed when studying a Markov Chain whose transition matrix is large and difficult to construct. It can determine the steady-state probability distribution without ever having to construct the transition matrix. However, the algorithm is not only limited to the discrete case of stochastic matrices and is useful when sampling from probability distributions where direct sampling is difficult. The desired probability distribution must be proportional to an objective function. The objective function places a value on each state in the Markov Chain and certain states become more likely than other. At each step potential candidate states are proposed by a proposal function. The algorithm is executed as follows:
Let f denote objective function and g denote proposal function.

- begin at an arbitrary initial state x_1

For every iteration, the Metropolis algorithm generates states of a Markov Chain as follows:

- Propose a new candidate x_i state with probability $g(x_i \mid x_j)$ where $x_i \neq x_j$

- Calculate the acceptance ratio $\alpha_{ij} = f(x_i)/f(x_j)$
- If $U \leq \alpha$, where $U \sim Uniform(0, 1)$, accept the candidate state and set $x_{j+1} = x_i$
- Otherwise, reject the candidate state and set $x_{j+1} = x_j$

For the Metropolis algorithm as stated above, it's important to note that the proposal function g must be symmetric. Therefore, the following equation must hold:

$$g(x_i \mid x_j) = g(x_j \mid x_i)$$

Aside: For non-symmetric proposal functions, the Metropolis algorithm was later generalized to the Metropolis-Hastings algorithm with a multiplicative corrective term on the acceptance ratio.

3.2 Preliminary Example

Consider a discrete random variable X such that:

$$P(X = x) = \frac{1}{C} \cdot f(x), \text{ where } f(x) = x^2$$

Assuming X can take on values of 1, 2, 3 or 4, the probability mass function is

X	1	2	3	4
$p(x)$	$\frac{1}{30}$	$\frac{2}{15}$	$\frac{3}{10}$	$\frac{8}{15}$

.

Alternatively, such a distribution can be determined by deploying the Metropolis algorithm.

3.3 Complete Graph Construction

Let $f(x)$ be the objective function.

Let X be the set of possible states.

Let the proposal function g be of constant probability for all proposed states given any current states. For example, if the current state is 1 then

$$g(x_2 \mid x_1) = g(x_3 \mid x_1) = g(x_4 \mid x_1) = 1/3.$$

The algorithm is implemented using python as follows:

```
>>> import numpy as np
>>> from numpy.random import default_rng
>>> rng = default_rng()
>>> from collections import Counter
>>> iter = 10000
>>> x = np.zeros(t)
>>> s = 1
>>> x[0] = s
>>> for i in range(1,iter):
...   others = [k for k in [1,2,3,4] if k != x[i-1]]
...   xprime = rng.choice(others)
...   v = (xprime)**2/ (x[i-1])**2
...   u = rng.random()
...   if u <= v:
...      s = xprime
...   else:
```

```
...      s = x[i-1]
...   x[i] = s
>>> c = Counter(x)
>>> print(np.array([c[1], c[2], c[3], c[4]])/t)
[0.0349 0.1349, 0.3005, 0.53]
```

The final print statement yields a probability vector $\vec{\pi}$ with π_X entries corresponding to the relative frequency of state X which aligns closely with the expected distribution found in the probability mass function above. Similar to all Monte Carlo methods, as the iterations increase $\vec{\pi}$ aligns closer with the expected distribution.

The probability vector $\vec{\pi}$ could have also been determined by constructing the transition matrix for a Markov chain where X is again the states. We can obtain an objective matrix F and a proposal matrix G if all states have an equal probability of being proposed,

$$\text{where } f_{ij} = \alpha_{ij} = f(x_i)/f(x_j), g_{ij} = g(x_i \mid x_j).$$

The matrix F will have zero entries along the diagonal since the Metropolis algorithm never proposes a new state x_i that's equal to the current state x_j. Furthermore, if $f(x_i)/f(x_j) > 1$ take $\alpha_{ij} = 1$ because such a ratio will always be greater than a random number sample from Uniform(0,1). If the proposed state is a greater number than the current state, the proposed state automatically becomes the new state. Therefore, the bottom triangular entries under the diagonal are all 1.

$$F = \begin{bmatrix} 0 & 1/4 & 1/9 & 1/16 \\ 1 & 0 & 4/9 & 1/4 \\ 1 & 1 & 0 & 9/16 \\ 1 & 1 & 1 & 1 \end{bmatrix}, \qquad G = \begin{bmatrix} 0 & 1/3 & 1/3 & 1/3 \\ 1/3 & 0 & 1/3 & 1/3 \\ 1/3 & 1/3 & 0 & 1/3 \\ 1/3 & 1/3 & 1/3 & 0 \end{bmatrix}.$$

Remark how $G^T = G$ which verifies that g is indeed a symmetric proposal function.

In order to construct the transition matrix P, the p_{ij} entries are the product between its corresponding f_{ij} and g_{ij} entries.

$$p_{ij} = f_{ij} \cdot g_{ij} = \alpha_{ij} \cdot g_{ij} \tag{3}$$

In the case of the zero entries along the diagonals of F and G, the diagonal entries of P are computed as $p_{jj} = 1 - \sum_{i=1}^{m} p_{ij}$.[1] By the *Perron-Frobenius theorem*, we can obtain the steady-state vector probability vector $\vec{\pi}$ through the $\ker(P - I)$

Note: $p_{ij} = P(x_i \mid x_j)$

$$P = \begin{bmatrix} 0 & 1/12 & 1/27 & 1/48 \\ 1/3 & 1/4 & 4/27 & 1/12 \\ 1/3 & 1/3 & 13/27 & 3/16 \\ 1/3 & 1/3 & 1/3 & 17/24 \end{bmatrix}, \quad \ker(P - I) = \ker\left(\begin{bmatrix} 1 & 0 & 0 & -1/16 \\ 0 & 1 & 0 & -1/4 \\ 0 & 0 & 1 & -9/16 \\ 0 & 0 & 0 & 0 \end{bmatrix} \right)$$

[1] See Figure 1 for a comprehensive transition graph under a Nearest Neighbour construction

$$= \left\{ t \begin{bmatrix} 1/16 \\ 1/4 \\ 9/16 \\ 1 \end{bmatrix}, t \in \mathbb{R} \right\}. \quad \text{By choosing } t = \tfrac{8}{15}, \vec{\pi} = \begin{bmatrix} 1/30 \\ 4/30 \\ 9/30 \\ 16/30 \end{bmatrix}$$

which also aligns with the proposed distribution. Remark that π_i entries are all multiples of $f(x_i)$ related by the normalization factor C. This result shows up in all Metropolis Algorithm. The fact that the objective function f must only be proportional and not equal to the desired distribution π makes it useful in many applications.

3.4 Nearest Neighbour Graph Construction

Remarkably, an alternative construction of the transition graph would also yield the correct distribution. Consider the following Nearest Neighbour transition graph where the possible candidates x_i are the nearest adjacent vertices of the current state x_j. In the figure below α_{ij} is the acceptance ratio which again is computed through the ratio of the objective function f at each state. Recall, $\alpha_{ij} = f(x_i)/f(x_j)$.

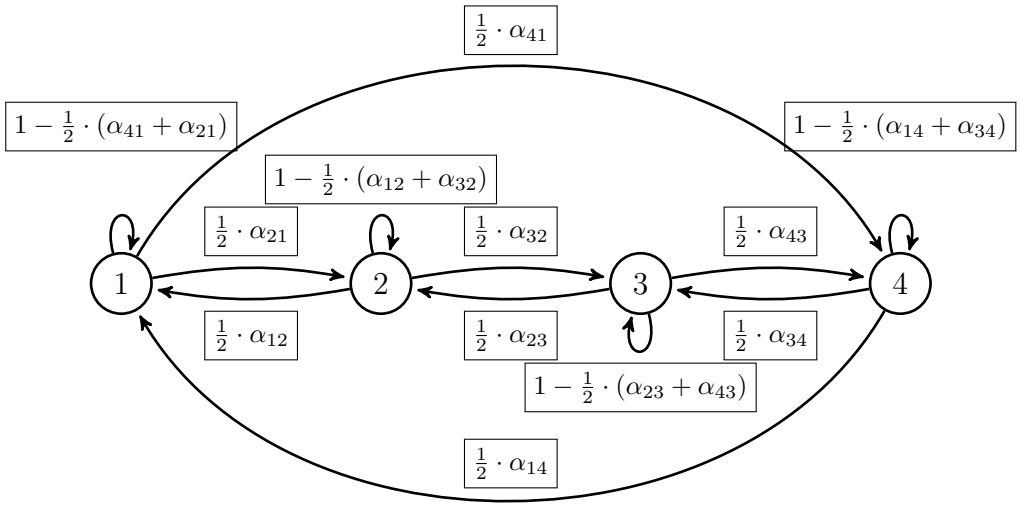


Figure 1: Nearest Neighbour Transition Graph

As opposed to the previous example where all states can be candidates x_i given any starting state x_j, in this case only the adjacent vertices of x_j can be x_i. The matrix P can be constructed using the transition graph above where the entry p_{ij} is the weight on the edge $j \to i$. Following the same process as before, we can determine the steady-state probability vector $\vec{\pi}$.

$$P = \begin{bmatrix} 0 & 1/8 & 0 & 1/32 \\ 1/2 & 3/8 & 2/9 & 0 \\ 0 & 1/2 & 5/18 & 9/32 \\ 1/2 & 0 & 1/2 & 11/16 \end{bmatrix}, \quad \ker(P - I) = \ker \left(\begin{bmatrix} 1 & 0 & 0 & -1/16 \\ 0 & 1 & 0 & -1/4 \\ 0 & 0 & 1 & -9/16 \\ 0 & 0 & 0 & 0 \end{bmatrix} \right)$$

$$= \left\{ t \begin{bmatrix} 1/16 \\ 1/4 \\ 9/16 \\ 1 \end{bmatrix}, t \in \mathbb{R} \right\}. \quad \text{By choosing } t = \tfrac{8}{15}, \vec{\pi} = \begin{bmatrix} 1/30 \\ 4/30 \\ 9/30 \\ 16/30 \end{bmatrix}$$

Therefore, regardless of its alternative construction, this transition matrix also yields the correct probability vector. Such a result can be explained through the Metropolis algorithm. Notice how Figure 1 suggests that candidate states are being proposed symmetrically. For example, in the case of states 1 and 2;

$$g(x_1 \mid x_2) = g(x_2 \mid x_1) = 1/2.$$

When considering all states, one can construct the proposal matrix G as follows:

$$G = \begin{bmatrix} 0 & 1/2 & 0 & 1/2 \\ 1/2 & 0 & 1/2 & 0 \\ 0 & 1/2 & 0 & 1/2 \\ 1/2 & 0 & 1/2 & 0 \end{bmatrix}, \qquad G^T = G$$

Under the alternative construction, a change was effected only on the proposal function g, while the objective function f remains unchanged. Nevertheless, G is still a symmetric matrix and therefore the Metropolis algorithm will effectively yield the correct probability distribution. Python code of the Metropolis algorithm under the Nearest Neighbour construction can be found in the appendix. In general, as long as G is a symmetric matrix, the algo-

rithm will yield the same distribution for the same objective matrix F, which explains why a different P matrix can determine the same probability vector.

3.5 A Proof of the Metropolis Algorithm

To prove that the Metropolis algorithm determines the steady-state distribution $\vec{\pi}$ in a stochastic matrix we first demonstrate that the algorithm satisfies detailed-balance. It can then be shown that detailed balance implies a steady-state distribution .

Proof. Let P be an stochastic $n \times n$ transition matrix

A Markov Chain with transition $P(x_i \mid x_j)$ and satisfies detailed balance if the following equation holds:

$$P(x_i \mid x_j)\pi_j = P(x_j \mid x_i)\pi_i.$$

Since the stationary distribution π is proportional to the objection function f the factor of C cancels and the equation can be expressed as

$$P(x_i \mid x_j)f(x_j) = P(x_j \mid x_i)f(x_i).$$

The acceptance ratio α can be rewritten as

$$\alpha_{ij} = \min\left\{1, \frac{f(x_i)}{f(x_j)}\right\}.$$

Following Equation 3, $P(x_i \mid x_j)$ can be written in terms of the proposal

function g and the acceptance ratio for $x_i \neq x_j$

$$P(x_i \mid x_j) = g(x_i \mid x_j) \min\left\{1, \frac{f(x_i)}{f(x_j)}\right\},$$

$$P(x_i \mid x_j) f(x_j) = g(x_i \mid x_j) f(x_j) \min\left\{1, \frac{f(x_i)}{f(x_j)}\right\},$$

$$= \min\left\{g(x_i \mid x_j) f(x_j), g(x_i \mid x_j) f(x_i)\right\}.$$

With i, j entries switched

$$P(x_j \mid x_i) = g(x_j \mid x_i) \min\left\{1, \frac{f(x_j)}{f(x_i)}\right\},$$

$$P(x_j \mid x_i) f(x_i) = g(x_j \mid x_i) f(x_i) \min\left\{1, \frac{f(x_j)}{f(x_i)}\right\},$$

$$= \min\left\{g(x_j \mid x_i) f(x_i), g(x_j \mid x_i) f(x_j)\right\}.$$

Recall that g is symmetric and by extension the last line for $P(x_i \mid x_j) f(x_j)$ and $P(x_j \mid x_i) f(x_i)$ are equal and thus the detailed balance criterion holds in the Metropolis Algorithm. Since π_i is proportional to $f(x_i)$ by a normalization factor C and $P(x_i \mid x_j)$ are the p_{ij} entries of stochastic matrix P detailed balance can be rewritten as:

$$p_{ij} \pi_j = p_{ji} \pi_i \tag{4}$$

When a steady-state distribution $\vec{\pi}$ is reached the following equation and its

i-th entry can be written as

$$P\vec{\pi} = \vec{\pi}, \quad (P\vec{\pi})_i = \sum_j^n p_{ij}\pi_j = \pi_i$$

Summing over j to n in Equation 4 yields

$$\sum_j^n p_{ij}\pi_j = \sum_j^n p_{ji}\pi_i = \pi_i \sum_j^n p_{ji} = \pi_i.$$

Recall that P is a stochastic matrix following the $j \to i$ convention therefore the sum along its columns is 1. This demonstrates that detailed balance implies that a steady-state distribution has been reached. □

4 Decryption Problem

Say we encode a message where each letter in the English alphabet has been uniquely substituted by another code symbol. The key function k that decodes or encodes the message can be called a one to one map.

$$k : (\text{code space}) \rightarrow (\text{English alphabet})$$

Applying the key function k to a coded symbol outputs a single unique letter in the English alphabet

$$k(c_n) = l_n.$$

The function acts on strings of coded symbols applying the transformation on each character in the order at which they are written in the code space.

For example it acts on the following string,

$$k(c_3c_{10}c_8c_{16}) = k(c_3)k(c_{10})k(c_8)k(c_{16}) = l_3l_{10}l_8l_{16}.$$

While function k appears to factorize and be commutative, each code symbol get mapped to a unique letter in the alphabet where order matters. A string of symbols in the code space get transformed character by character while preserving the order at which they appear in the string. In order to decode an unfamiliar text, we must find the correct k that converts a collection of code symbols to understandable English text. Considering that the stan-

dard English alphabet consists of 26 letters and other various symbols must be accounted for, k is a discrete function with 27 values. By extension, there are 27! possible permutations functions k to sift through in order to find the correct one. The brute force approach of testing around 10^{28} key functions would be computationally near impossible. The following presents a viable method to decode text that can be deployed to relatively any message in any language under the condition that function k is a one to one map.

4.1 Random Walk Approach

Rather than running 10^{28} trials to find the correct k, a Metropolis discrete random walk along the space of k functions is deployed in order to determine the correct k, in a relatively low number of number iterations. First, a quantitative plausibility of being correct must be assigned to each k. In other words, we need a way to measure how effective a certain key function is at decoding text to comprehensive English. This can be thought of as a score that rates each k.

Consider a matrix M that has transition $M(l_j \mid l_i)$ which corresponds to the proportion of consecutive letters from i to j. The rows and columns of M are constructed starting from all miscellaneous symbols such as spaces to a, b, c ... z. Therefore, it is of size 27×27. (e.g. the m_{13} entry is the proportion of spaces that are followed by the letter b.)

$$M = \begin{array}{c} \\ \text{␣} \\ \text{a} \\ \text{b} \\ \vdots \\ \text{z} \end{array} \begin{array}{c} \begin{array}{cccc} \text{␣} & \text{a} & \text{b} \cdots & \text{z} \end{array} \\ \begin{bmatrix} m_{11} & \cdots & m_{1n} \\ & & \\ \vdots & \ddots & \vdots \\ & & \\ m_{n1} & \cdots & m_{nn} \end{bmatrix} \end{array}$$

In *The Markov Chain Monte Carlo Revolution*, Persi Diaconis suggest to compute the plausibility of each key as

$$\text{Pl}(k) = \prod_n M[k(c_{n+1}) \mid k(c_n)],$$

where c_n corresponds to a symbol in the code space and n is it's position in the string of symbols.

To gain a better understanding of the plausibility function, consider the following example that deals with a string of symbols and demonstrates how the plausibility function is computed for two different key functions.

$$\clubsuit\sharp\triangle\triangle\bigstar\diamondsuit,$$

$$k_1(\clubsuit\sharp\triangle\triangle\bigstar\diamondsuit) = \text{summer}, \quad \text{Pl}(k_1) = m_{20,22} \cdot m_{22,14} \cdot m_{14,14} \cdot m_{14,6} \cdot m_{6,19},$$

$$k_2(\clubsuit\sharp\triangle\triangle\bigstar\diamondsuit) = \text{tfccga}, \quad \text{Pl}(k_2) = m_{21,7} \cdot m_{7,4} \cdot m_{4,4} \cdot m_{4,8} \cdot m_{8,2}.$$

Intuitively, k_1 is more effective at decoding the string of symbols into understandable English and this is measured quantitatively by the Pl function. In English, the letter t is almost never followed by the letter f, the letter f

is almost never followed by the letter c, so on and so forth[2]. On the other hand, the word summer is very plausible in terms of the proportion of letters that follow each other. As a result, $\mathrm{Pl}(k_1) > \mathrm{Pl}(k_2)$ and k_1 is favoured by the Metropolis algorithm .

However, over an increasingly long string of text Diaconis' definition of the Pl function yields numbers that are much too small to work with. Therefore, the logarithm of the Pl function is used and Pla can be defined as

$$\log(\mathrm{Pl}(k)) = \mathrm{Pla}(k) = \sum_n \log(M[k(c_{n+1}) \mid (k(c_n)]).$$

Key functions k with large $\mathrm{Pla}(k)$ are the most effective at decoding text. Therefore, we want to find k that maximizes the plausibility function. To accomplish this, a Metropolis discrete random walk on the state space of k is executed similar to the preliminary example with states X. Now the proposal function f is the Pla function and the algorithm is deployed as follows:

- begin at a guessed initial state k_1

For every iteration, the Metropolis algorithm generates each state of a Markov Chain as follows:

- Propose a new candidate key function k_i with probability $g(k_i \mid k_j)$ where $k_i \neq k_j$
- Calculate the acceptance ratio $\alpha_{ij} = \mathrm{Pla}(k_i)/\mathrm{Pla}(k_j)$

[2]Remark that the letter t is actually never followed by letter f, which would result in $m_{21,7} = 0$ and $Pl(k_2) = 0$. In an application, this becomes a problem for the algorithm since some key functions become impossible. A simple solution is to substitute the zero entries in M by relatively small numbers.

- If $U \leq \alpha$, where $U \sim Uniform(0,1)$, accept the candidate state and set $k_{j+1} = k_i$, where k_{j+1} is taken as the key function at the next iteration.
- Otherwise, reject the candidate state and set $k_{j+1} = k_j$

Each candidate key function k_i is constructed by performing a random transposition of two values assigned to the previous key function k_j. By extension, there are $\binom{27}{2}$ possible candidate states proposed at each transition. The standard Metropolis Algorithm can be deployed since

$$g(k_i \mid k_j) = g(k_j \mid k_i) = \frac{1}{\binom{27}{2}}$$

therefore the proposal function is symmetric.

4.2 Key Function State Transition Graph

The transition graph of the key function states is huge and is much to large to be contained in this margin, yet consider the following figure to get the essence of its construction.

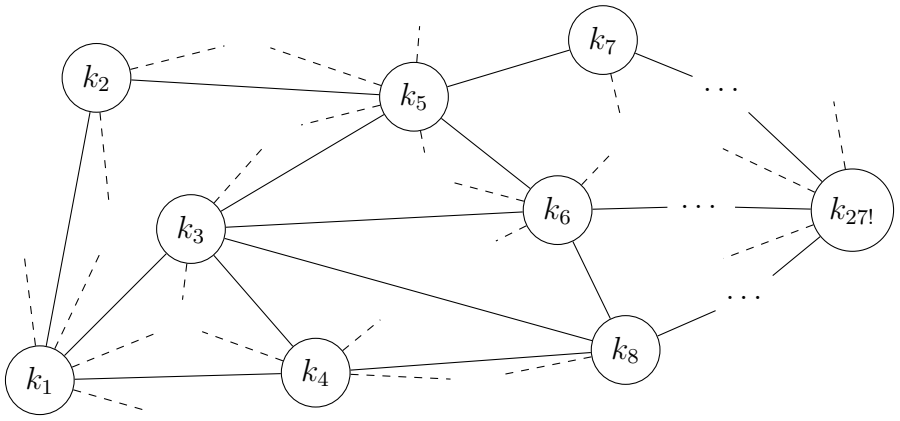


Figure 2: Key function transition graph

The transition graph T is a directed graph with 27! number of vertices. Each vertices has $\binom{27}{2}$ incoming edges as well as $\binom{27}{2}$ outgoing edges.[3] By computing the binomial coefficient the vertices have 351 in and out coming edges which is a relatively low numbers of considering T has around 10^{28} vertices.Nevertheless, T is relatively small in diameter. Intuitively, we can get from any permutation of the letters in the English alphabet to any other permutation in 25 moves or less. If we were to add one character for the spaces and miscellaneous symbols it would be 26 moves. Therefore, the diameter of T is only 26.

[3]Figure 2 displays an undirected graph for simplicity purposes, yet each edge in the Figure embodies both directed edges E_{ij} and E_{ji}.

By extension, the Metropolis Algorithm can arrive at any key function starting from any key function in a relatively low number of iterations. It avoids explicitly constructing a $27! \times 27!$ transition matrix for T. Instead, the algorithm determines the steady-state 27! entry column vector $\vec{\pi}$ with π_i entries corresponding to $\text{Pla}(k_i)$ of each k_i possible key functions. It accomplishes this thought a Metropolis random walk along the states space of k functions or alternatively along the vertices of T. Again, the objective function $\text{Pla}(k_i)$ is proportional to the steady-state vector entry π_i by a normalization factor C. The correct key function k_c becomes the mode of the distribution or alternatively the greatest entry of $\vec{\pi}$ is $\text{Pla}(k_c)$.

5 Application

The decryption methods detailed above were programmed using Python code which can be found linked in the appendix. The program was used to decode a scrambled version of the following Shakespeare exert from *Hamlet*.

ENTER HAMLET HAM TO BE OR NOT TO BE THAT IS THE QUESTION WHETHER TIS NOBLER IN THE MIND TO SUFFER THE SLINGS AND ARROWS OF OUTRAGEOUS FORTUNE OR TO TAKE ARMS AGAINST A SEA OF TROUBLES AND BY OPPOSING END

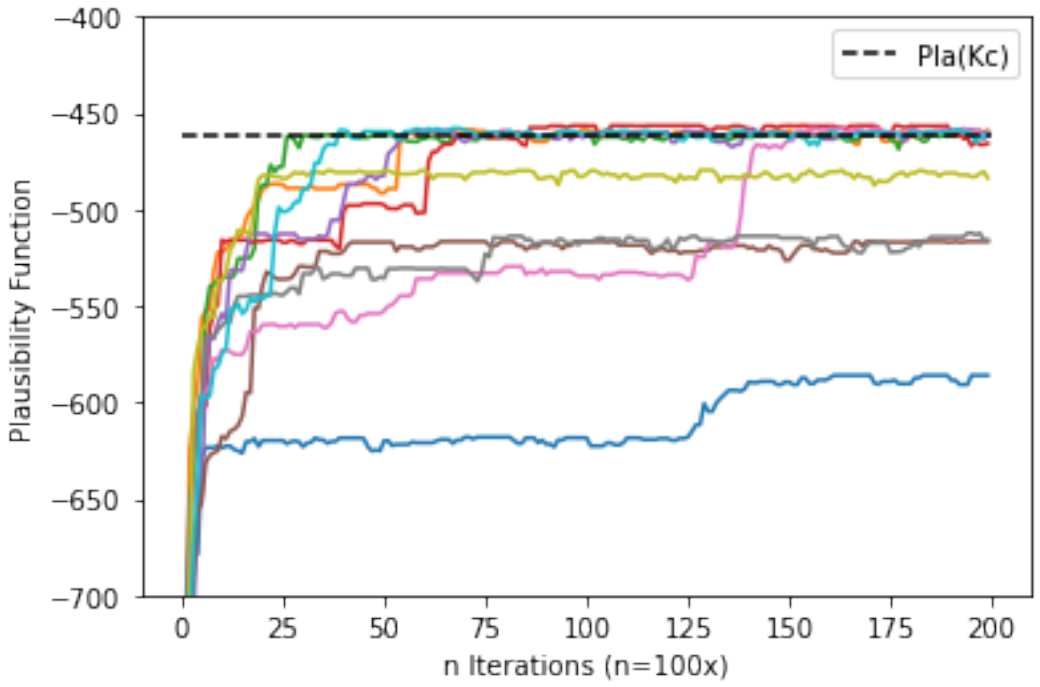
The algorithm was for the most part successful at decoding the text in a relatively small number of iterations considering the huge 27! possible key functions. After around 20000 iterations the algorithm either decoded the text entirely, decoded the text to the point where two letters were inverted or would generate a text that is still illegible. At times, the Metropolis algorithm would get stuck in a local maxima of Pla and would never reach the desired key function within a reasonably low number of iterations. Remarkably, the algorithm would also find a k_{max} for which $\text{Pla}(k_{max}) > \text{Pla}(k_c)$ given k_c is the correct key function. The k_{max} function outputs the following text with $\text{Pla}(k_{max})$ = -456.54.

ENTER HALPET HAL TO ME OR NOT TO ME THAT IS THE BUESTION WHETHER TIS NOMPER IN THE LIND TO SUFFER THE SPINGS AND ARROWS OF OUTRAGEOUS FORTUNE OR TO TAVE ARLS AGAINST A SEA OF TROUMPES AND MY OCCOSING END

As can be seen from the text above, the k_{max} maps the code symbols to almost perfectly legible English. Therefore in its application, finding k_{max}

as opposed to k_c isn't an issue. At this point, decoding the text manually is a matter of a few simple transpositions of letters and k_{max} is useful. A discrepancy between plausibility functions of k may have been caused by the construction of the M matrix. In the python script, Arthur Conan Doyle's *The Adventures of Sherlock Holmes* was used to collect the statistics on relative letter frequencies. Yet even in the case where Shakespeare's Own *Macbeth* was used to construct M, the algorithm managed to find a k_{max} with higher plausibility than k_c.

The following graph displays the behaviour of the logarithm of the plausibly of the key functions. Over ten experiment, which correspond to the colours, the plausibility was measured as a function of the iteration of the algorithm. The black dotted line illustrate the plausibility of the correct key function.



As can be seen above, some of the red, orange and cyan curve appear above dotted black line which suggest the algorithm finds a key function that is more plausible than the correct one. The algorithm also gets stuck at a local maximum for a couple thousand iterations and is sometimes unable to reach the correct key function. At this point, in practical use, the program can be run again or we can attempt to interpret the text at the current plausibility. After 20000 iterations, the grey curve outputs a text with Pla(k) = -510.88 as,

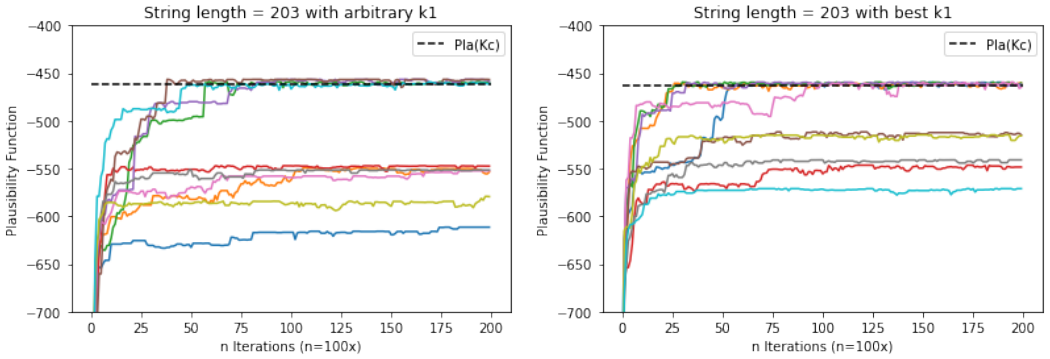
EITEN HAGLET HAG TO BE ON IOT TO BE THAT SR THE CUERTSOI WHETHEN TSRIOBLEN SI THE GSID TO RUPPEN THE RLSIMR AID ANNOWR OP OUTNAMEOUR PONTUIE ON TO TAVE ANGR AMASIRT A REA OP TNOUBLER AID BY OFFORSIM EID.

Given that $\text{Pla}(k_c) = -463.07$, the end plausibility of the grey experiment isn't far off. Some small two letter words can be distinguished yet the text is still illegible and is beyond our abilities of manual decryption. Notice how the spaces and the characters that appear quite often in the English language are mapped to the correct character. In the previous ten experiments, the initial k_1 was guessed as an arbitrary key function. However, we can give the algorithm these correct characters to begin with by guessing k_1 as the key function that maps the most frequent code symbols to the most frequent characters in the English language.

5.1 Length of Text and Optimal Initial Key Function

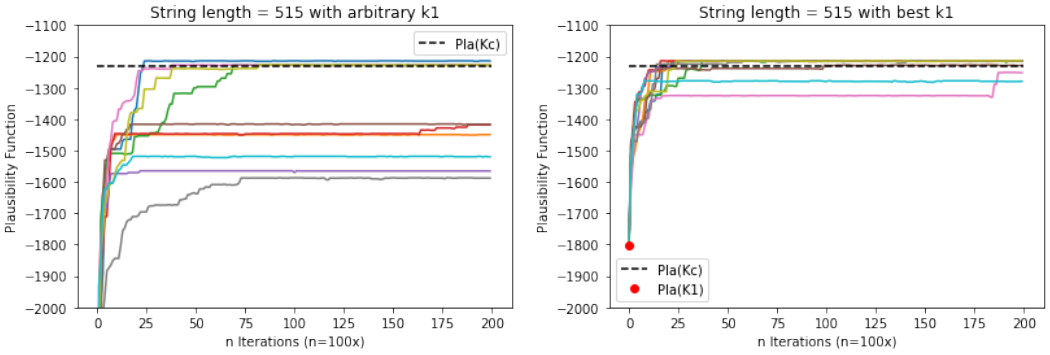
The following two graphs illustrate two cases where the same text from *Hamlet* was decoded, which is 203 characters in length. The graph on the

left illustrates the behaviour when the algorithm begins with an arbitrary key function and on the right, the algorithm begins with a key function that is constructed from letter frequencies.



In short text, using the best initial k_1 seems to have a negligible effect on the number of curves that reach the dotted line however in longer text the effects become more apparent.

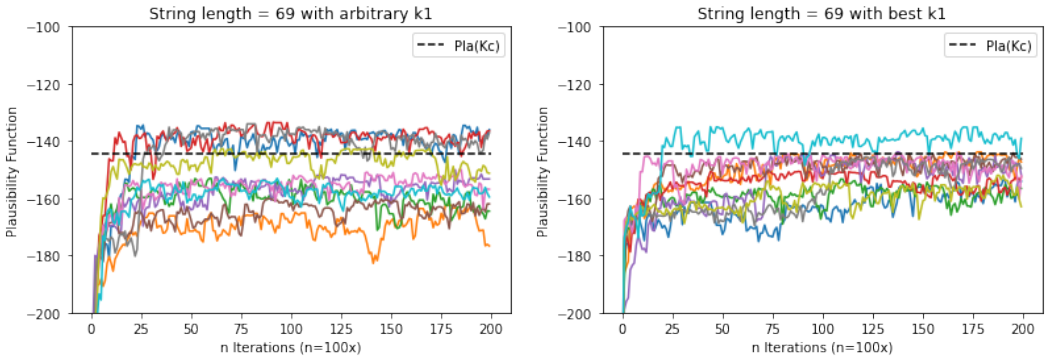
A longer portion the *Hamlet* soliloquy was decoded and the behaviour of the plausibility function is as follows



In this case, starting at the best k_1 helps significantly since in almost all experiments the longer text text is decoded successfully. The algorithm now

skips over all highly unlikely key functions and reaches Pl(k_c) = -1232.35 more frequently out of the ten experiments by starting from Pl(k_1) = -1798.57.

In shorter text, the Metropolis Algorithm seems to encounter difficulty decoding text.



The two graphs resemble a regular 1-dimensional random walk as opposed to a Metropolis random walk. In general, the methods listed above aren't as effective at decoding text which are shorter in length because the algorithm doesn't have enough information about the relative frequencies of the transition between code symbols. In this case, different key functions seems to be equally plausible since their aren't enough characters to consider. As can be seen by the curves that exceed the black dotted line the algorithm finds key functions that are more plausible that the correct one. This demonstrates that it doesn't have enough information about the text to distinguish other key functions from the correct one. In the case of longer text, it is more effective and more reliable at decoding text. Combined with starting at the best k_1, it seems that the algorithm handles longer texts optimal. It is presumed that in increasingly long texts the methods become increasingly better at decrypting.

6 Conclusion

The Markov Chain Monte Carlo method studied in this paper is effective in its application problem however it isn't used for its original intended purpose. Nevertheless, it still works for decrypting encoded messages into standard English and effectively optimizes the key function. Traditionally, the Metropolis algorithm is deployed to determine the entire steady-state probability distribution $\vec{\pi}$. However, in the decryption problem we are only interested in the greatest value of the discrete 27! value probability distribution. The greatest value in this distribution corresponds to the plausibility function of the correct k function that maps the code space to comprehensive and readable English text. Once the mode of the distribution is found, the algorithm can be terminated and the text is almost entirely if not entirely decoded. In the essence of its intended purposes, generating the steady-state $\vec{\pi}$ in the decryption Markov chain the algorithm would have to be run for over 27! iterations. However for the purposes of the problem 20,000 iterations in most cases the mode of the distribution is already visible and the correct k is determined. Taking 20,000 sample from the distribution $\vec{\pi}$ is sufficient. Hence, the Metropolis Algorithm effectively solves an optimization problem it wasn't specifically intended to solve. This Markov Chain Monte Carlo algorithm is not only remarkable due to its application in statistical physics but also in its applicability in unconventional problems.

References

[1] Diaconis, P.. "The Markov chain Monte Carlo revolution." Bulletin of the American Mathematical Society 46 (2008): 179-205.

[2] Liu, Jun S. "Monte Carlo Strategies in Scientific Computing." Springer Series in Statistics, 2004, doi:10.1007/978-0-387-76371-2.

[3] Grinstead, Charles Miller, and James Laurie Snell. Introduction to Probability. American Mathematical Society, 2006.

[4] Haugh, M. (2017). MCMC and Bayesian Modeling. New York City ; Columbia University.

[5] Pistone, G. Rogantin, M. P. (2011). The Algebra of Reversible Markov Chains. Collegio Carlo Alberto and Universita Di Genova.

[6] Speagle, J. S. (2020). A Conceptual Introduction to Markov Chain Monte Carlo Methods. Harvard; Smithsonian Center for Astrophysics , 1–1.

[7] UC Berkeley. (2018). Reversible Markov Chains. California .

Appendix A

Preliminary Example: Python Code

```
import numpy as np

from numpy.random import default_rng

rng = default_rng()
```

```
from collections import Counter n = 10000
y = np.zeros(t)
s = 1
y[0] = s
for i in range(1,n):
  if y[i-1] == 1:
    yprime = rng.choice([2,4])
  if y[i-1] == 2:
    yprime = rng.choice([1,3])
  if y[i-1] == 3:
    yprime = rng.choice([2,4])
  if y[i-1] == 4:
    yprime = rng.choice([1,3])
  v = (yprime)**2/ (y[i-1])**2
  u = rng.random()
  if u <= v:
    s = yprime
  else:
    s = y[i-1]
```

```
  y[i] = s
c = Counter(y)
np.array([c[1], c[2], c[3], c[4]])/t
```

A Simple Monte Carlo Simulation: Python Code

```
import numpy as np
from numpy.random import default_rng
rng = default_rng()
from collections import Counter
n_samples = 10000
samples = [rng.choice([1,2,6]) for r in range(n_samples)]
c = Counter(samples)
np.array([c[1], c[2], c[6]])/n_samples
array([0.33302, 0.33741, 0.32957])
```

Appendix B

Decryption Algorithm: Python Code

Colab Python Link