Decipher with Tempered MCMC

Zhenan Fan, Yeming Wen

Supervisor: Prof. Jefferey S. Rosenthal

University of Toronto

September 21, 2016

Contents

1 Introduction

Markov Chain Monte Carlo (MCMC) Algorithm is a class of algorithm to approximately sample from certain distributions with high dimensionality. In specific, the algorithm constructs a Markov Chain which has the desired distribution as its equilibrium distribution. Then after enough times of iterations, the state of the Markov Chain after burn-in iteration could be viewed as the approximate sample from the desired distribution. Usually, we attain better approximation if we increase the number of iterations[1, 2].

MCMC algorithm could be extremely useful in the case where we couldn't exactly calculate the desired distribution due to the high dimensionality. For example, in the context of bayesian statistic, posterior distribution could sometimes be difficult to be calculated in a closed form since we might have too many parameters. One way to deal with it is to use MCMC algorithm to attain approximate samples from the posterior distribution and take the average[3]. Furthermore, we could even use these samples to approximate the predictive distribution.

There are several ways to implement the MCMC algorithm. The main difference is how they make a new proposal during iterations. For example, Metropolis Hastings algorithm makes a new proposal based on random walk and symmetric distribution. Gibbs sampling is based on conditional distribution. We also have slicing sampling, which based on the graph of the desired distribution. These algorithms follow a consistent rule to generate new proposal(such as fixed standard error in the Metropolis Hastings algorithm). Besides, people also develop some other variations of MCMC algorithm, such as adoptive MCMC[4, 5]. It adjusts some parameters during the MCMC iteration such as the standard error in the Metropolis Hastings algorithm.

2 MCMC Algorithm for Optimization

It is common to implement MCMC algorithm to attain samples from high dimensional distribution. But it is also possible to optimize a function via MCMC algorithm. Suppose we are given a function $f(x)$ that we want to maximize where x could be a high dimensional vector. And we could choose x_0 as a random initial x as the start point in the MCMC algorithm. Then during each iteration in the MCMC algorithm, we make a new proposal y according to some certain rules depend on x. Next, we decide whether to accept this new proposal y following a acceptance probability function. Usually, we wanna have accept the new proposal with probability one if $f(y) \geq f(x)$ and with some non-zero probability otherwise(since we don't wanna see the Markov Chain is trapped at the local maximum). After a number of iterations, suppose S is the space

of x that we accepted during the MCMC algorithm, i.e.

$$S = \{x : x \text{ is in the Markov chain}\}$$

We can take

$$\max_{x \in S} f(x)$$

as the approximate maximum for $f(x)$.

To improve the optimization performance of the MCMC algorithm, we could use tempered MCMC, which adjusts some parameters in acceptance probability function during iterations[6]. The purpose is still to avoid being trapped at the local maximum.

In the context of this report, we wanna use MCMC algorithm to decrypt a cipher text. The general idea is first we need to have a long reference text, we choose War and Peace in our project. Then we design a score function that could measure how close for a given text look like the reference text based on letter frequencies, bigram frequencies or even trigram frequencies(we will give explanation what such means later). So the procedure to decipher becomes to find a key that could give a maximal score function and we can use the MCMC algorithm here since it looks exactly the same as the optimization problem. The MCMC algorithm was introduced to break substitution ciphers by Connor [7]. Also, we can use the MCMC algorithm to decrypt the transposition cipher and substitution-transposition[8]. Our project mainly refer to this paper and we found that we can improve the results by modifying the algorithm.

3   MCMC to decipher

The MCMC algorithm to decrypt transposition In our project, we first pre-process the text into a text only involves capital letters and spaces. In specific, we remove punctuations and convert all lower case letters into capital. Now we define what bigram frequency $f(\beta_1, \beta_2)$ means.

Definition 1. $f_T(\beta_1, \beta_2)$ *is the frequency of the appearance of $\beta_1\beta_2$ where β_1, β_2 are both English letters(such as A,B and sometimes we allow it to be space character) in a given text T.*

For example, $f_R(\beta_1, \beta_2)$ stands for the bigram frequency of β_1, β_2 in the reference text.
Similarly, we could define tri-gram frequency in the way.

Definition 2. $f_T(\beta_1, \beta_2, \beta_3)$ *is the frequency of the appearance of $\beta_1\beta_2\beta_3$ where $\beta_1, \beta_2, \beta_3$ are both English letters(such as A,B and sometimes we allow it to be space character) in a given text T.*

Next we introduce the concept of encryption key. Keys could be in different formats depend on different contexts. For example, in the context of simple substitution cipher, keys are 26 unique English letters such as 'XEBPROHYAUFTIDSJLKZMWVNGQC'. And it stands for the letter 'A' in the plaintext is replaced by 'X' and the letter 'B' is substituted by 'E'. In the context of transposition cipher with length k, keys are permutations of $1, 2, 3, \ldots, k$, which stands for how we perform permutation on the plaintext.

Given a key X, suppose the plaintext is T, the reference text is R and the cipher text is C. Then we use the following notation to stand for we encrypt plaintext T with key X to get the cipher text C.

$$C = X(T)$$

Thus, our goal is to find a key W such that $W(C) = T$.

Next we can define what it is a score function.

Definition 3. *A bigram score function $\Pi(X)$ is a function from any text X to a real number. In most case, we will use*

$$\Pi(X) = \prod_{\beta_1 \beta_2 \in X} (f_R(\beta_1, \beta_2) + 1)^{f_X(\beta_1, \beta_2)}$$

where $f_R(\beta_1, \beta_2)$ and $f_X(\beta_1, \beta_2)$ are both bigram frequencies.

Similarly, we could define what tri-gram score function is.

Definition 4. *A tri-gram score function $\Pi(X)$ is a function from any text X to a real number. In most case, we will use*

$$\Pi(X) = \prod_{\beta_1 \beta_2 \beta_3 \in X} (f_R(\beta_1, \beta_2, \beta_3) + 1)^{f_X(\beta_1, \beta_2, \beta_3)}$$

where $f_R(\beta_1, \beta_2, \beta_3)$ and $f_X(\beta_1, \beta_2, \beta_3)$ are all tri-gram frequencies.

Remark 1. *Since we might use log domain during the MCMC algorithm due to some numerical reasons, then we might have $log(0)$ if $f_R(\beta_1, \beta_2) = 0$. So we add one to the base $f_R(\beta_1, \beta_2)$ to avoid this potential error. Generally it works quite well in most cases, but it might cause some problems in block substitution of size two(detailed in Section 8).*

Remark 2. *We could also define the score function from encryption key X to real number since $X(C)$ is a text. i.e.*

$$\Pi(X) = \prod_{\beta_1 \beta_2 \in X(C)} (f_R(\beta_1, \beta_2) + 1)^{f_{X(C)}(\beta_1, \beta_2)}$$

where $f_R(\beta_1, \beta_2)$ and $f_X(\beta_1, \beta_2)$ are both bigram frequencies. Sometimes we use this equation depends on the context(whether X is text or is a key). But for convenience we also write

$$\Pi(X) = \prod_{\beta_1 \beta_2 \in X} (f_R(\beta_1, \beta_2) + 1)^{f_X(\beta_1, \beta_2)}$$

even in the case where X is a key.

Let $\mathcal{X}$ be all possible keys. For the purpose of the MCMC algorithm, the score function f should satisfy the following condition

$$arg \max_{w \in \mathcal{X}} f(w(C))$$

is the key we are searching for, which means if

$$W = arg \max_{w \in \mathcal{X}} f(w(C))$$

then $W(C) = T$ or at least $W(C) \approx T$.

Another essential part in the MCMC algorithm is the acceptance probability function.

Definition 5. *An acceptance probability function a function $A(X, Y)$ from current key X and a proposal key Y to real number between $[0, 1]$, which indicates the probability we accept the new proposal key Y.*

Remark 3. *In our project, we set*

$$A(x, y) = (\frac{\Pi(y(C))}{\Pi(x(C))})^p$$

where Π is the score function and p is a scaling parameter.

Remark 4. *The scaling parameter p controls how strict we accept the new proposal key Y. In basic MCMC algorithm, p is fixed during iterations while it is not fixed in the tempered MCMC algorithm, which we will give the detail in Section 3.*

To see how the value of p will change the acceptance probability, we can just see the following two extreme examples. It is obvious that if $\Pi(Y) \geq \Pi(X)$, we will accept the proposal key Y and changing the value of p would not affect this. In the case where $\Pi(Y) < \Pi(X)$, the value of p plays an important role to decide whether to accept Y. For example, if $p = 10,000$ and $\frac{\Pi(Y)}{\Pi(X)} = 0.999$, then $A(x, y) = 0.999^{10000} = 0.00004$, which means we have a strict acceptance policy and basically we won't accept any proposal keys Y if $\Pi(Y) < \Pi(X)$. But if $p = 0.001$, even for $\frac{\Pi(Y)}{\Pi(X)} = 0.01$, the acceptance probability becomes $0.01^{(}0.001) = 0.995$, which suggests we will accpet the new proposal key Y no matter how bad it is. We won't use such extreme p's in our algorithm. These two examples are just for the purpose

of illustration.

Now we can give the general MCMC algorithm for decrypting a cipher text.

Notation

1. $R :=$ reference text, text of War and Peace in the project.

2. $P :=$ plain text, text we need to encrypt.

3. $C :=$ cipher text, text we want to decrypt.

4. $M =$ length of Markov chain. i.e. number of iterations during MCMC algorithm.

5. $\mathcal{X} :=$ set of all possible decryption keys

6. $f_R(\beta_1, \beta_2) :=$ the number of times the bigram "$\beta_1\beta_2$" appears in the reference text.

7. $f_T(\beta_1, \beta_2) :=$ the number of times the bigram "$\beta_1\beta_2$" appears in a text T.

8. $\Pi(X) :=$ score function we use through out the MCMC algorithm and X could be text or an encryption key.

9. $q(x, y) :=$ Given a key x, we propose a new key y according to the distribution $q(x, y)$. Usually we require $q(X, Y)$ to be symmetric. i.e. $q(x, y) = q(y, x)$.

10. $A(x, y) :=$ Given a current x and a proposal key y, we accpet the proposal y with probability $\min(A(x, y), 1)$. In this project, we let

$$A(x, y) = (\frac{\Pi(y(C))}{\Pi(x(C))})^p$$

11. $p :=$ scaling parameter in $A(x, y)$. It controls how strict we accept a new proposal.

Basic MCMC Algorithm for decipher

---

1 X = an initial key;

2 $keyList[0] = X$ (It stores every accepted keys);

3 $scoreList[0] = f(X(C))$ (It stores scores in every step of iterations);

4 for *i from 1 to M* **do**

5   Propose a new key Y according to the distribution $q(X, Y)$;

6   Generate U from uniform distribution;

7   **if** $u < A(x, y)$ *(we take* log *in actual implementation)* **then**

8 $X = Y$;

9   Append X to $keyList$;

10   Append $\Pi(X(C))$ to $scoreList$;

11 $j = scoreList.argmax()$ (Find the index of maximum score in the Markov chain);

12 return $keyList[j]$;

---

4 Tempered MCMC

With the basic MCMC algorithm, we could basically solve simple substitution cipher and transposition cipher. But in practice, we found the basic MCMC algorithm sometimes doesn't give a perfect result. It might be pretty closed but not exactly the key we are searching for. For example, in simple substitution, suppose the plaintext is encrypted with the key 'XEBPROHYAUFTIDSJLKZMWVNGQC'. The basic MCMC algorithm could sometimes give us 'XEBPROHYAUFTIDSZLKJMWVNGQC', which is pretty close to the decryption key but not exactly. The reason is the basic MCMC algorithm only behaves quite well if the score function $\Pi(X)$ is peak enough so that the algoritm is able to give the exact maximum, which is the decryption key we are searching for. But if the socre function $\Pi(X)$ has multiple modes, then it is possible for the basic MCMC algorithm to be trapped at some local maximum, which will give us an incorrect decryption. But since the local maximum does not differ too much from the global maximum, we could still end up with a key which is pretty close to the decryption key.

There are several ways to fix this. One obvious way it to do more iterations. Since we are taking the maximum score in the whole Markov chain, then doing more iterations first would not lead to a worse result. Also, increasing the number of iterations allow the algorithm to have higher chance to jump out of the local maximum once it get trapped. But the downside of it is obvious, too. More iterations means more running time and to jump out of the local maximum would typically require much more iterations than before. So

we consider some other methods on the purpose of efficiency.

A better idea is to use tempered MCMC. The only difference between tempered MCMC and the basic MCMC algorithm is we could change the value of scaling parameter p during the MCMC iteration. First we will give an short example on tempered MCMC and illustrate it performs better if the desired distribution has multiple modes.

Suppose now the function we want to maximize is $\Pi(x) = N(10,1) + 0.5N(0,1)$. Judging from the plot of function, we could clearly see that the score function has two modes. We first plot $\Pi(x)^p$ for $p = 0.001, p = 1, p = 100$ respectively. The second plot is the function we want to maximize. If we apply
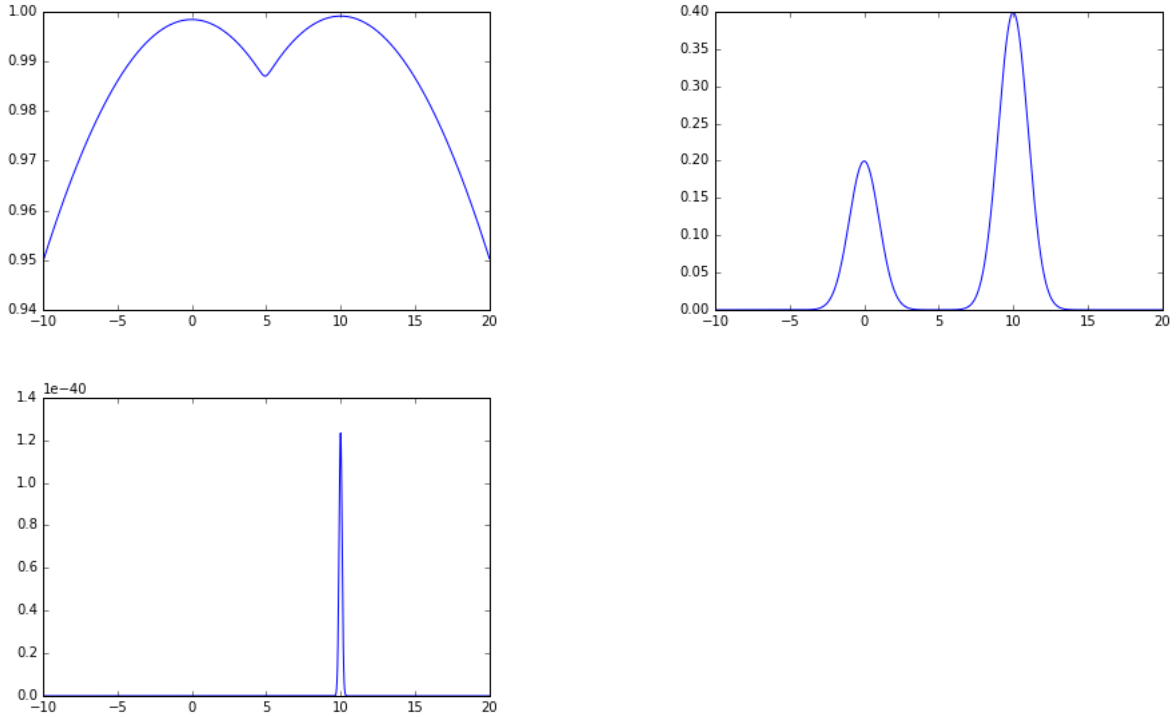
Figure 1: Plot of $\Pi(x)^p$ where $p = 0.001, 1, 100$

the basic MCMC algorithm, then in some cases the Markov Chain is trapped at $x = 0$, which it is local maximum instead of the global maximum we are searching for. To attain a better result in this example, we should let $p = 100$ and then we can attain the global maximum since $\Pi(x)^{100}$ is peak enough(not multiple modes).

Although increasing the value of p can help us find the global maximum in the above example, it is not

9

applied in all cases. In the above example, the local maximum actually differs quite much from the global maximum, so such simple method would work. However, in some cases the local maximum may be very close to the global maximum, and thus simply raising the value of p would not solve the problem. Instead we could use small value of p first. The idea is we want to search as wide as possible at the beginning of the MCMC algorithm. Then we can increase the value of p to focus on some intervals that might contain the global maximum. This is exactly the tempered MCMC algorithm. As the word tempered suggests, we change the value of p in every iteration. In specific , we use small value of p and then we increase the value of p gradually.

Tempered MCMC Algorithm for decipher

- M = length of the run

- p = scaling parameter

- C = cipher text

---

1 X = an initial key;

2 $keyList[0] = X$ (It stores every accepted keys);

3 $scoreList[0] = f(X(C))$ (It stores scores in every step of iterations);

4 for *i from 1 to M* **do**

5   Propose a new key Y according to the distribution $q(X, Y)$;

6   Generate U from uniform distribution;

7 $p = g(i)$ where $p \to \infty$ as $i \to \infty$ (For example, $p = \frac{1000(i+1)}{M}$) ;

8   **if** $u < A(x, y, p)$ *(we take* log *in actual implementation)* **then**

9 $X = Y$;

10   Append X to $keyList$;

11   Append $\Pi(X(C))$ to $scoreList$;

12 $j = scoreList.argmax()$ (Find the index of maximum score in the Markov chain);

13 return $keyList[j]$;

---

5 Simple Substitution Cipher

Simple substitution is just an one-one substitution cipher, which means we just replace each letter with another one. In this report, for illustration reason, we only substitute alphabetic letters in the context of

simple substitution. We give an example(Table1) about the encryption and decryption of a simple substitution cipher.

plain text	THE PROJECT GUTENBERG EBOOK OF OLOVER TWIST
encryption key	XEBPROHYAUFTIDSJLKZMWVNGQC
cipher text	MYR JKSURBM HWMRDERKH RESSF SO STAVRK MNAZM
decryption key	ICZNBKXGMPRQTWFDYEOLJVUAHS
decrypted text	THE PROJECT GUTENBERG EBOOK OF OLIVER TWIST

Table 1: An example of simple substitution

In simple substitution cipher, we define the accuracy to be $\frac{m_s}{n_s}$ where m_s denotes the number of letters that have been correctly revealed and n_s denotes the total number of letters in the plain text (usually 26). Also if the MCMC algorithm can give us an exact decryption key, then we say it is a successful run.

Next, we need to define the proposal function for simple substitution decipher.

Proposal Method for substitution Cipher[8]: Propose a new key by swapping 2 randomly selected letters in the current key. So each swap has probability $\frac{1}{n^2}$, which means it is a symmetric proposal. Formally, we can write down the algorithm **swap(substitution-key)**

---

1 l = length(current_key);

2 Randomly select p_1 from $\{0, 1, ..., l-1\}$;

3 Randomly select p_2 from $\{0, 1, ..., l-1\}$;

4 propose_key = current_key;

5 $propose\_key[p_1] = current\_key[p_2]$;

6 $propose\_key[p_2] = current\_key[p_1]$;

7 **return** $propose\_key$;

Basic MCMC Algorithm for deciphering the Simple substitution cipher

- M = length of the run

- p = scaling parameter

- C = cipher text

11

1 $X =$ an initial key;

2 $keyList[0] = X$ (It stores every accepted keys);

3 $scoreList[0] = f(X(C))$ (It stores scores in every step of iterations);

4 for *i from 1 to M* **do**

5 $p = 1$;

6 $Y = swap(X)$;

7   Generate U from uniform distribution;

8   **if** $u < (\frac{\pi(Y)}{\pi(X)})^p$ **then**

9 $X = Y$;

10   Append X to $keyList$;

11   Append $\Pi(X(C))$ to $scoreList$;

12 return $\arg\max\limits_{x \in keyList} \pi(x)$;

Tempered MCMC Algorithm for deciphering the Simple substitution cipher

---

1 $X =$ an initial key;

2 $keyList[0] = X$ (It stores every accepted keys);

3 $scoreList[0] = f(X(C))$ (It stores scores in every step of iterations);

4 for *i from 1 to M* **do**

5 $p = (i+1)^{\frac{3}{4}}$;

6 $Y = swap(X)$;

7   Generate U from uniform distribution;

8   **if** $u < (\frac{\pi(Y)}{\pi(X)})^p$ **then**

9 $X = Y$;

10   Append X to $keyList$;

11   Append $\Pi(X(C))$ to $scoreList$;

12 return $\arg\max\limits_{x \in keyList} \pi(x)$;

There are several questions for simple substitution cipher. First we wanna know how much MCMC iteration we need to decipher. Next, does it make any difference to change the length of cipher text? And we also want to see whether tempered MCMC improve the result. Finally, we want to try different score functions, such as bigram score function and tri-gram score function.

1. We explore how much MCMC iteration we need. The following is the result we attain. We can see there is no much need to increase the number of iterations to 10,000. Since after 3000 iterations, we don't really any improvement and more iteration would increase the running time. So for simple substitution with 2,000 cipher text length and basci MCMC algorithm, 3,000 iterations should be enough.
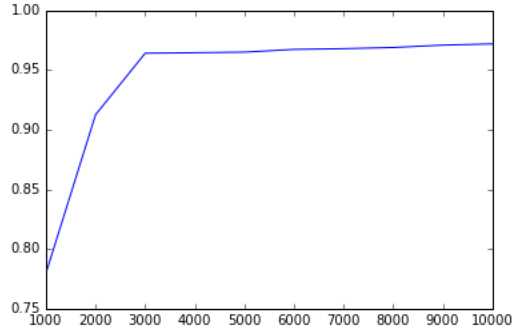
Figure 2: number of iterations vs average accuracy(Basic MCMC)

Iterations	Accuracy	Running time
1000	0.78	1.2 secs
2000	0.9127	2.7 secs
3000	0.9642	3.8 secs
4000	0.9646	5.1 secs
5000	0.9651	5.9 secs
6000	0.9674	7.4 secs
8000	0.9698	10.2 secs
10000	0.9720	13.1 secs

2. Now we check whether tempered MCMC can bring any benefits. We use the above tempered MCMC algorithm and attained the following result. We could see tempered MCMC doesn't improve the result too much in term of the accuracy. But we found we only need 2000 iterations for tempered MCMC to converge to maximum score, which means tempered MCMC is more efficient for cipher text with length 2,000.

Figure 3: number of iterations vs average accuracy(Tempered MCMC)

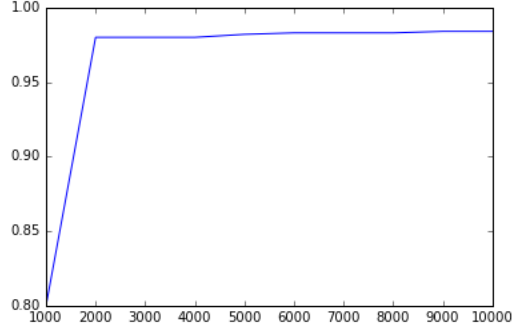3. Next, we explore different length of cipher text. In this part, we use tempered MCMC with 3,000 iterations and test the algorithm on cipher text with length 500, 1,000, 2,000 and 5,000. The output is the following.

Length	Accuracy	Successful run out of 100
500	0.8942	49
1000	0.9131	61
2000	0.9797	85
5000	0.9820	91

If we decrease the length of the text, the accuracy would not decrease too much, but the number of successful runs will drop significantly.

4. In the above part, the score function is based on bigram. We could also try tri-gram score function. And the result is in the following table. From the table, we could see there is no much difference

Number of iterations	Accuracy	Successful run out of 100
1000	0.7633	0
2000	0.9633	75
3000	0.971	86
4000	0.971	87

between bigram score function and tri-gram function. To be consistent, we will choose bigram score

14

function in the following section.

6   Transposition Cipher

Transposition cipher is also called permutation cipher. The formal Transposition cipher is gotten by splitting the plain text into fixed sized blocks and then permuting each block with the same permutation key.

We give an example(Table1) about the encryption and decryption of a transiposition cipher.

plain text	THE PROJECT GUTENBERG EBOOK OF
encryption key	1937045862
cipher text	HC JTPREOE RUBTTEENG FB GOOOKE
decryption key	4092568371
decrypted text	THE PROJECT GUTENBERG EBOOK OF

Table 2: An example of transiposition

Important idea : Instead sliding move of a single decryption position, everytime we use slide move involving entire blocks of decryption positions[8]

Formal procedure:

Assume length of key $= k$. Then everytime we randomly choose n from $\{0, 1, ..., k - 2\}$.

Set the lenth of the block to be n, and thus the possible positions are $\{0, 1, ..., k - n + 1\}$

Then we randomlly choose $k_1, k_2 \in \{0, 1, ..., k - n + 1\}$, and slide move the block of size n from k_1 to k_2

We can define function **slidemove(key_transiposition)**

---

1 k = length(key_transiposition);

2 Randomly select n from $\{0, 1, ..., k - 2\}$ ▷ Size of the block;

3 Randomly select p_1 from $\{0, 1, ..., k - n + 1\}$ ▷ Initial position of the block;

4 Randomly select p_2 from $\{0, 1, ..., k - n + 1\}$ ▷ Position want to move the block to;

5 $part1 = key\_transiposition[0 : p_1]$;

6 $part2 = key\_transiposition[p_1 : p_1 + n]$;

7 $part3 = key\_transiposition[p_1 + n : p_2 + n]$;

8 $part4 = key\_transiposition[p_2 + n : k]$;

9 $key\_new = part1 + part3 + part2 + part4$ ▷ Switch Part2 and Part3;

10 **return** key_new;

Basic MCMC Algorithm for decrypting the Transposition cipher

---

1 $X =$ identity permutation of length k;

2 $xlist =$ Null;

3 $scorelist =$ Null;

4 for *i from 1 to M* **do**

5 $\quad$ $Y = $ slidemove(X) ;

6 $\quad$ Random select subset S from the cipher text with length l;

7 $\quad$ Generate uniform random variable u;

8 $\quad$ **if** $u < (\frac{\pi(Y)}{\pi(X)})^p$ **then**

9 $\quad\quad$ $X = Y$;

10 $\quad$ Add X to xlist;

11 $\quad$ Add $\pi(X)$ to scorelist;

12 $index = \arg\max_{i} scorelist[i]$;

13 return $xlist[i]$;

We use the method of variable control to select parameters:

First, we want to find the best scaling parameter p, and we fix all the other parameters

- length of cipher text $= 1000$

- length of key $= 20$

- number of iteration $= 5000$

scaling parameter	Average Accuracy	number of successful runs out of 100
0.01	0.12	0
0.1	0.88	72
1	0.93	89
10	0.89	77
100	0.83	63

Table 3: Selecting scaling parameter for the basic MCMC algorithm

So as we can see from the table, we gonna choose $p = 1$ as our scaling parameter.

16

Tempered MCMC Algorithm for decrypting the Transposition cipher

1 $X =$ identity permuation of length k;

2 $xlist =$ Null;

3 $scorelist =$ Null;

4 for *i from 1 to M* **do**

5 $\quad\tau = \frac{M}{500i}$;

6 $\quad p = \frac{1}{\tau}$;

7 $\quad Y = \text{slidemove}(X)$;

8 $\quad$ Random select subset S from the cipher text with length l;

9 $\quad$ Generate uniform random variable u;

10 $\quad$ **if** $u < (\frac{\pi(Y)}{\pi(X)})^p$ **then**

11 $\quad\quad X = Y$;

12 $\quad$ Add X to xlist;

13 $\quad$ Add $\pi(X)$ to scorelist;

14 $index = \arg\max\limits_{i} scorelist[i]$;

15 return $xlist[i]$;

Then we want to compare the performance of basic MCMC and tempered MCMC on this case.

And we gonna compare them based on two aspects:

Minimum number of iteration needed with fixed key length and cipher text length

- length of cipher text = 1000

- length of key = 20

From the output we can see, if we set a lower bound for the number of successful runs out of 100 to be 70, then for basic MCMC algorithm, it needs at least 4500 iteration to achieve such that threshold while the tempered MCMC only need 3000 iterations.

Minimum cipher text length needed with fixed key length and running length

- number of iteration = 5000

- length of key = 20

Number of Iteration	Average Accuracy	number of successful runs out of 100
500	0.14	0
1000	0.40	13
1500	0.4	15
2000	0.69	41
2500	0.70	45
3000	0.80	56
3500	0.80	58
4000	0.83	63
4500	0.88	70
5000	0.93	89

Table 4: Number of iterations needed for Basic MCMC

Number of Iteration	Average Accuracy	number of successful runs out of 100
500	0.13	0
1000	0.30	5
1500	0.4	15
2000	0.71	44
2500	0.70	45
3000	0.83	71
3500	0.83	78
4000	0.83	79
4500	0.83	85
5000	0.95	91

Table 5: Number of iterations needed for Tempered MCMC

Length of Ciphertext	Average Accuracy	number of successful runs out of 100
50	0	0
100	0.01	0
300	0.12	15
500	0.41	43
1000	0.93	89

Table 6: Minimum cipher text length needed needed for Basic MCMC

Length of Ciphertext	Average Accuracy	number of successful runs out of 100
50	0	0
100	0.01	0
300	0.1	19
500	0.58	67
1000	0.95	91

Table 7: Minimum cipher text length needed for Tempered MCMC

7 Substitution-Transposition Cipher

Substitution-Transposition cipher has two different keys: one for substitution and the other for transposition. Namely, if we want to encipher a text, we follows the procedure

First Switch the letters in the plain text by using the substitution key.

Next Slide move the letters by using the transposition key.

We give an example(Table1) about the encryption and decryption of a substitution-transposition cipher.

plain text	THE PROJECT GUTENBERG EBOOK OF
encryption substitution key	XEBPROHYAUFTIDSJLKZMWVNGQC
cipher text	MYR JKSURBM HWMRDERKH RESSF SO
encryption transiposition key	1937045862
cipher text	YB UMJKRSR KWEMMRRDH OE HSSSFR
decryption substitution key	ICZNBKXGMPRQTWFDYEOLJVUAHS
decrypted text	HC JTPREOE RUBTTEENG FB GOOOKE
decryption transiposition key	4092568371
decrypted text	THE PROJECT GUTENBERG EBOOK OF

Table 8: An example of substitution-transposition

Important idea1: Make propose on both of the two keys

Let X_1, X_2 denote the key for substitution cipher and transposition cipher respectively, and we propose the new keys Y_1, Y_2 by some symmetric density $q((x_1, x_2), (y_1, y_2))$. And then run the MCMC algorithm.

Problem:

The convergence rate will become very low.

Important idea2: use multiple cycles [8]

Each cycle consists of a bigram attack on the substitution cipher, followed by a bigram attack on transposition cipher. And we use the result of the previous cycle as the starting point for next one.

Problem:

If the attack on the substitution cipher is bad, then the following attack on the transposition cipher can hardly succeed.

So based on the ideas and problems illustrated above, we decide to combine two ideas to gether, namely

1. First, we use MCMC to attack both substitution key and transiposition key

2. Then we use multi-cycles to attack substitution key and transiposition key respectively

For the proposal, we still use the function **slidemove** for substitution key and function **swap** for transiposition key

The algorithm for swap(substitution_key) is

---

1 $l = \text{length(current\_key)}$;

2 Randomly select p_1 from $\{0, 1, ..., l-1\}$;

3 Randomly select p_2 from $\{0, 1, ..., l-1\}$;

4 propose_key = current_key;

5 $propose\_key[p_1] = current\_key[p_2]$;

6 $propose\_key[p_2] = current\_key[p_1]$;

7 **return** $propose\_key$;

And the algorithm for slidemove(transiposition_key) is

---

1 $k = \text{length(key\_transiposition)}$;

2 Randomly select n from $\{0, 1, ..., k-2\}$ ▷ Size of the block;

3 Randomly select p_1 from $\{0, 1, ..., k-n+1\}$ ▷ Initial position of the block;

4 Randomly select p_2 from $\{0, 1, ..., k-n+1\}$ ▷ Position want to move the block to;

5 $part1 = key\_transiposition[0 : p_1]$;

6 $part2 = key\_transiposition[p_1 : p_1 + n]$;

7 $part3 = key\_transiposition[p_1 + n : p_2 + n]$;

8 $part4 = key\_transiposition[p_2 + n : k]$;

9 $key\_new = part1 + part3 + part2 + part4$ ▷ Switch Part2 and Part3;

10 **return** key_new;

The algorithm for decrypt(ciphertext, bigram_reference) is given as bellow.

---

1 M = 3000   ▷ Length of the Run;

2 X = ['0123456789','ABCDEFGHIJKLMNOPQRSTUVWXYZ']   ▷ initial cipher key;

3 x0list = Empty list;

4 x1list = Empty list;

5 x0list[0] = X[0];

6 x1list[0] = X[1];

7 scorelist = Empty list;

8 scorelist[0] = $log(\pi(X))$;

9 for *i from 0 to M-1* **do**

10 $p = \frac{i+1}{30}$ ▷ Tempered Version;

11   Y0 = slidemove(X[0]);

12   Y1 = swap(X[1]);

13   Y = [Y0,Y1];

14   score_proposal = $log(\pi(Y))$;

15   Generate u from uniform distribution;

16   **if** $log(u) < p(score\_proposal - scorelist[i])$ **then**

17   X = Y;

18   scorelist[i+1] = score_proposal;

19   **else**

20   scorelist[i+1] = scorelist[i];

21   x0list[i+1] = X[0];

22   x1list[i+1] = X[1];

23 index = $\arg\max_{i}\{score_i \in$ scorelist $\}$;

24 X0 = x0list[index];

25 X1 = x1list[index];

26 X = [X0,X1]   ▷ This is the end of joint attack;

27 for *c = 1:m*   ▷ *number of cycles* **do**

28   ciphertext = encrypt_transiposition(ciphertext, X1);

29   X0 = decrypt_substitution(ciphertext);

30   ciphertext = encrypt_substitution(ciphertext, X0);

31   X1 = decrypt_transiposition(ciphertext);

32 return *X = [X0,X1]*

Performance of the algorithm on different ciphertext and reference text

- length of ciphertext = 2000

- length of substitution key = 26

- length of transiposition key = 10

- iteration length on joint ones = 3000

- iteration length for each cycle = 2000
  (1000 on attacking substitution key and 1000 on attacking transiposition key)

Ciphertext	Reference text	Average Accuracy	number of successful runs out of 100
Oliver Twist	War and Peace	0.87	83
Ice Hockey	War and Peace	0.93	91
War and Peace	Oliver Twist	0.99	87
Ice Hockey	Oliver Twist	0.93	88
Oliver Twist	Ice Hockey	0.81	84
War and Peace	Ice Hockey	0.83	88

Table 9: Performance of the algorithm on different ciphertext and reference text

8 Data Encrytion Standard

Another popular way to encrypt a given plaintext is Data Encryption Standard(DES), which performs bit-level operation on letters or block of letters. The encryption process is much more complicated than substitution and transposition cipher, so it is much more difficult to decrypt. In this section, we first introduce how to encrypt the plaintext via SDES(Simplified Data Encryption Standard).

SDES has similar properties and structure as DES, but has been simplified to make it easier give illustration. Then we will show how we decrypt the SDES cipher and why DES is equivalent to polygraphic substitution cipher[9].

How to encrypt with SDES:

To encrypt a given plaintext by SDES, we first transform our plaintext into binary form by ACSII rule. Every letter in English(including space character) can be converted into a binary number with length 8 in an one-one fashion. For example, letter 'A' corresponds to '01000001' and letter 'B' corresponds to '01000010', and so a bigram 'AB' would correspond to '0100000101000010'. SDES performs encryption on every binary number with length 8(8-bit). Therefore we just need to show how to encrypt an 8-bit binary such as '00101000'(in this example) with SDES. Everything else can be followed similarly.

In order to encrypt an 8-bit binary with SDES, we first need a initial 10-bit binary key. Let's say '1100011110'.

Figure 4: General encryption procedure of SDES[10].

Then we generate two 8-bit sub-keys K_1 and K_2 with some permutation and shifting rules, which will be used in the process of following encryption. Suppose after some permutations and shifting, we obtain $K_1 = 11101001$ and $K_2 = 10100111$. Now we define an inverse permutation which is called IP in the diagram, it is applied in the first step and the last step in the encryption process.
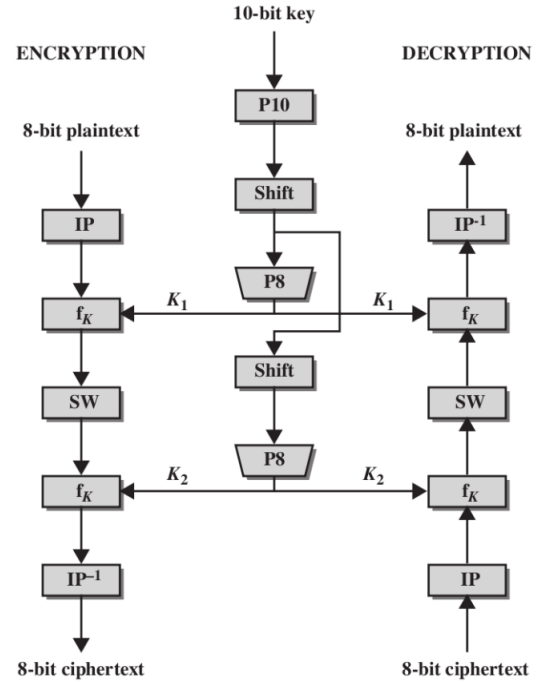
Now, according to the diagram, we need to apply encrypted function f_K to '00100010'. For notational convenience, we define $P =$'00100010'$= (L, R)$ where $L =$'0010' stands for the left component and $R =$'0010'

IP	IP^{-1}
2 6 3 1 4 8 5 7	4 1 3 5 7 2 8 6

Bit	1 2 3 4	5 6 7 8
Initial number	0 0 1 0	1 0 0 0
After IP	0 0 1 0	0 0 1 0

stands for the right component. f_k is a complicated function, so to decompose it, we write

$$f_{SK}(P) = f_k(L, R) = (L \oplus F(R, SK), R)$$

where F is another function we will next and SK just means sub-keys. In our example, it is $K_1 == 11101001$ when we apply f_k in the first time and it is $K_2 = 10100111$ at the second time based on the diagram.
The function $F(p, k)$ takes a four-bit string p and a 8-bit binary sub-key k and produces a four-bit output. The specific steps involve expansion and permutation, XOR operation, S-BOX substitution and a P4 permutation(choose 4 and do permutation). Many steps involve keys that we need to specify before the encryption, so we omit the detailed steps and give the output here. In this example, we have

$$F(0010, K_1) = 0001 \implies f_{K_1}(00100010) = (0010 \oplus F(0010, K_1), 0010) = (0010 \oplus 0001, 0010) = (0011, 0010)$$

Now, we apply function SW(swap the left component and the right component) to the output of f_K, which it is $(0011, 0010)$ and we will have $(0010, 0011)$. Next, we apply f_K again except we use K_2 as the sub-key this. The output of f_{K_2} is $(0001, 0011)$. Finally, we perform IP^{-1} permutation to it and end up with '10001010' in this example.

How to decrypt the SDES:
It seems that the encryption process of SDES is fairly complicated and there is no hope if we run MCMC algorithm to guess the keys(there are so many of many!). But actually if we look into the encryption process we can find out that SDES can only encrypt one letter at a time. To encrypt a given plaintext, we need to apply SDES encryption algorithm letter by letter. What's more, we found that SDES encrypt each letter in an one-one fashion. In specific, letter 'A' will always be encrypted into a unique 8-bit binary number. Thus, the SDES encryption process is equivalent to substituting every letter in English(including space character) into a unique 8-bit binary number. And it performs encryption letter by letter, which suggests the method we use in simple substitution cipher could also be applied here.
Notice that the SDES procedure does not guarantee to produce a 8-bit binary that can be converted into letter again, so we need to use some tricks during decryption which would be given details next.
Our algorithm to decrypt simple substitution cipher only works if the input is in alphabetic form, but the

SDES cipher is in binary form. The first thought is to convert every 8-bit binary into a letter according to ASCII rule since we do it for every letter in the plaintext in the first place. But since the output of SDES encryption for a letter is a random 8-bit binary, then it is not necessary true that we can convert it back into English letters or space. To resolve this problem, we first pre-process the SDES cipher. Specifically, we examine every 8-bit binary in the cipher and record different patterns of them. For example, if the cipher text C is '11000010 10000110 11000010 01111100', we can collect three different patterns. After collection patterns, we substitute every 8-bit binary with the following rules. The first pattern is substituted by letter 'A' and the second pattern is replaced by 'B'. Since SDES algorithm encrypt letters in an one-one fashion, then number of possible patterns in the cipher text would not exceed 27(including space character), which means the substitution is valid. For the above cipher text example C, it becomes 'ABAC' after pre-processing. Now we can apply what did in simple substitution to decrypt the SDES cipher text. In terms of the algorithm, we just need to add a pre-process part at the beginning of the algorithm we use in simple substitution. We attach the algorithm at the end of the section.

Thoughts on DES:

DES encryption and SDES encryption share a similar encryption structure. The essential difference between them is DES performs encryption for a block of letters while SDES only encrypts one letter at a time. In the encryption process, DES takes a 64-bit binary(equivalent to 8 letters) input instead of a 8-bit binary at a time and encrypt it into another 64-bit binary. So every possible combinations of 8 letters is being encrypted into a 64-bit binary pattern. It has 27^8 different patterns where we only have 27 in the case of SDES encryption. The good news is DES encryption is still equivalent to substitution cipher but it is applied to every block of letters with length 8. We call it block substitution cipher or poly-graphic cipher. There is no hope to decrypt SDES with the previous algorithm, so we ease the problem and only consider whether we could apply the similar algorithm to the block substitution cipher with length two.

Tempered MCMC Algorithm for SDES

- $M = $ length of the run

- $p = $ scaling parameter

- $C = $ cipher text(in binary)

- $L = $ length of the cipher text(in bit)

25

1 $List = []$ (An empty list);

2 $Pattern\_Dict = dict()$ (We store different patterns in a Python dictionary);

3 $alphabet =$' ABCDEFGHIJKLMNOPQRSTUVWXYZ';

4 $k = 0$;

5 **for** *j from 1 to L with step 8* **do**

6   **if** $C[j : j + 8]$ *not in* $Pattern\_Dict.keys()$ **then**

7 $k+ = 1$;

8   List.append(alphabet[k]);

9   **else**

10   List.append(alphabet[k])

11 C = ''.join(List) (Join letters in List into a new cipher text and we finish pre-processing);

12 $X =$ an initial key;

13 $keyList[0] = X$ (It stores every accepted keys);

14 $scoreList[0] = f(X(C))$ (It stores scores in every step of iterations);

15 **for** *i from 1 to M* **do**

16 $p = (i + 1)^{\frac{3}{4}}$;

17 $Y = swap(X)$;

18   Generate U from uniform distribution;

19   **if** $u < (\frac{\pi(Y)}{\pi(X)})^p$ **then**

20 $X = Y$;

21   Append X to $keyList$;

22   Append $\Pi(X(C))$ to $scoreList$;

23 **return** $\arg \max\limits_{x \in keyList} \pi(x)$;

9   Block Substitution with size 2

As we mentioned in the last section, data encryption standard encryption is equivalent to block substitution with size eight. To simplify the analysis, we only try to decrypt block substitution cipher with length two in this section.

Key matrix:

First, we need to define the key for the block substitution cipher with length two. We adopt the idea of playfair cipher here. Since the space character is included, the key matrix in block substitution with length two is of size 27×27. The row in the key matrix stands for ' ABCDEFGHIJKLMNOPQRSTUVWXYZ' in order and the column stands for the same thing. The key matrix is filled with integers from 1 to 27^2 without replacement. Also, for notational convenience, we are going to order every possible bigram. The order is specified by the matrix with the same structure as the key matrix. But the elements in the matrix are in numerical order. So the first row is $1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, \ldots, 27$ and the second row is $28, 29, 30, \ldots, 54$. By that we can see, the second bigram is ' A', the third one is ' B' and the 28th one is 'A ' and so on. Now suppose k is the element in the key matrix position $[i, j]$. Then it stands for the ith row character and j^{th} column character is replaced by the kth bigram. For example, if the element in the key matrix with position $[2, 3]$ is 29. It means the bigram 'AB' is substituted by the bigram 'AA' in the encryption process.

For example, the order matrix(left) and an arbitrary key matrix(right) is

$$
\begin{bmatrix}
 & \text{␣} & A & \ldots & Z \\
\text{␣} & 1 & 2 & \ldots & 27 \\
A & 28 & 29 & \ldots & 54 \\
\vdots & \vdots & \vdots & \ddots & \vdots \\
Z & \ldots & \ldots & \ldots & 27^2
\end{bmatrix}
\qquad
\begin{bmatrix}
 & \text{␣} & A & \ldots & Z \\
\text{␣} & 178 & 992 & \ldots & 23 \\
A & 3671 & 500 & \ldots & 254 \\
\vdots & \vdots & \vdots & \ddots & \vdots \\
Z & \ldots & \ldots & \ldots & 3
\end{bmatrix}
$$

Initialiazation:

In the MCMC algorithm, we need to choose a initial key. When dealing with other substitution ciphers, we just randomly generate an initial key, which turns out to be a bad choice in this case. Specifically, we fix a plaintext with length 5,000 and encrypt it. If we substitute the plaintext into the bigram score function, the score is 48747, which we call theoretical max score. After the MCMC algorithm with random initialization, the max score in the Markov chain is around 33000 in average. But if we choose a good initial key(detailed next), the initial score would be around 43000, which it is a significant improvement.

To choose a good initial key, we adopt the idea of unigram attack. In the unigram attack, we just replace the letter with the highest frequency in the cipher text with the letter with the highest frequency in the reference text. And replace the letter with the second highest frequency with the corresponding one in the reference text. Similarly, we perform the replacement in terms of the bigram in this case. So we replace the bigram with the highest frequency in the cipher text with the corresponding one in the reference text. And we do it for every bigram appeared in the cipher text. Usually, we might come across the case where cipher text has more kinds of bigram than that in the reference text. If so, we just don't replace any more bigram in the cipher text after running out choices of bigram in the reference text.

How to propose a new key:

For the proposal method, the basic idea we use is still swap keys in the key matrix, but since the size of the key matrix is very big, if we only swap two keys at each iteration, the convergence rate will be very low. However, if we keep swapping more than two keys, it can hardly be accepted during the final period of the algorithm, because the key matrix should be close to the correct one during that time. So this is actually a trade-off problem.

So we define to use different proposal methods during the algorithm.

The formal procedure is given as below:

- First, assume the number of iteration is M, then we equally divide it into three sub-iterations each with length $\frac{M}{3}$

- We use swap_8keys as our proposal method in the first sub-iteration

- We use swap_4keys as our proposal method in the second sub-iteration

- We use swap_2keys as our proposal method in the third sub-iteration

And the algorithms for swap_mkeys is given as bellow

1   l = width(current_key);

2   Randomly select p_1 from $\{0, 1, ..., l-1\}$;

3   Randomly select p_2 from $\{0, 1, ..., l-1\}$;

4   a = current_key[p_1,p_2];

5   new_key = current_key;

6   **for** $i = 1\text{:}m\text{-}1$ **do**

7   Randomly select p_3 from $\{0, 1, ..., l-1\}$;

8   Randomly select p_4 from $\{0, 1, ..., l-1\}$;

9   new_key[p_1,p_2] = current_key[p_3,p_4];

10     $p_1 = p_3$;

11     $p_2 = p_4$;

12   new_key[p_1,p_2] = a;

13   **return** *new_key*;

Then we can define the formal tempered MCMC algorithm as follows:

1 X = identity permuation matrix of size $l \times l$;

2 $xlist$ = Null;

3 $scorelist$ = Null;

4 for *i from 1 to M* **do**

5 $\tau = \frac{M}{50i}$;

6 $p = \frac{1}{\tau}$;

7   **if** $i < \frac{M}{3}$ **then**

8   Y = swap_8keys(X);

9   **else**

10   **if** $i < \frac{2M}{3}$ **then**

11   Y = swap_4keys(X);

12   **else**

13   Y = swap_2keys(X);

14   Generate uniform random variable u;

15   **if** $u < (\frac{\pi(Y)}{\pi(X)})^p$ **then**

16 $X = Y$;

17   Add X to xlist;

18   Add $\pi(X)$ to scorelist;

19 $index = \arg\max_{i} scorelist[i]$;

20 return $xlist[i]$;

First, we make the size of key matrix to be 27×27, and we define

- M = 500000

- Cipher text length = 5000

And we run the algorithm for 100 times, and get the result

1. Average accuracy = 0.114

2. Number of successful runs = 0

3. Running time = 2431 secs

And sometimes we may fall in the problem that the score we get is actually higher than the theoritical max score (computed by the plaintext), but the encrypted cipher text is quite different from the plaintext. We found out that this problem is mainly caused by the movement of ␣.

For example, we may get ␣E␣ in our ciphertext, this is actually not allowed in the real text, but can still get very high score.

So we decide to fix the location of the ␣. And thus the size of key matrix to be 26×26

- M = 2000000

- Cipher text length = 5000

And we run the algorithm for 5 times, and get the result

1. Average accuracy = 0.531

2. Number of successful runs = 0

3. Running time = 1443 secs

We notice it's better than the previous one, and we won't have the problem of the ␣. But the result is still not satisfactory.

And our future work is to improve this method.

References

[1] Gilks, W.R.; Richardson, S.; Spiegelhalter, D.J. (1996). Markov Chain Monte Carlo in Practice. Chapman and Hall/CRC.

[2] Markov chain Monte Carlo(Wikipedia Page). `https://en.wikipedia.org/wiki/Markov_chain_Monte_Carlo`

[3] Tierney, Luke. Markov Chains for Exploring Posterior Distributions. Ann. Statist. 22 (1994), no. 4, 1701–1728.

[4] Y.F. Atchade and J.S. Rosenthal (2005), On Adaptive Markov Chain Monte Carlo Algorithms. Bernoulli 11, 815828.

[5] G.O. Roberts and J.S. Rosenthal, Examples of Adaptive MCMC. J. Comp. Graph. Stat. 18:349-367, 2009.

[6] Radford M. Neal, Sampling from multimodal distributions using tempered transitions, Statistics and Computing, 6(4):353366, 1996.

[7] S. Conner (2003), Simulation and solving substitution codes, Masters thesis, Department of Statistics, University of Warwick

[8] J. Chen and J.S. Rosenthal, Decrypting Classical Cipher Text Using Markov Chain Monte Carlo Stat. and Comput. 22:397-413, 2011.

[9] P. Garg (2009), Cryptanalysis of SDES via Evolutionary Computation Techniques, IJCSIS 1(1), May 2009.

[10] `https://www.youtube.com/watch?v=qHZKze24kVo`