Decrypting Polyalphabetic Substitution Ciphers using Markov Chain Monte Carlo

Chou, K (995497626) & Eastman, E (999918415)

February 26, 2013

Abstract

Recent research has demonstrated the use of Markov Chain Monte Carlo (MCMC) methods to decrypt cipher text that was encoded using a monoalphabetic substitution cipher. In this paper we expand upon previous methods by considering periodic polyalphabetic substitution ciphers. We were successful in decrypting cipher text using a Metropolis-within-Gibbs algorithm for periods of length 2 and 3, but this algorithm will work for any period length. We also briefly discussed a method to infer the period in cipher text when the actual period is unknown.

Contents

1 Introduction

A polyalphabetic substitution cipher belongs to the group of classical ciphers. Classical ciphers were used to encrypt messages throughout history using substitutions, transpositions or combinations of the two. They are known as 'classical' because they have fallen out of use with the development of much more secure modern ciphers such as DES or AES. The substitution cipher works by switching the letters in the plain text with other letters or symbols, yielding an encrypted text. A simple example of this would be mapping each letter of the alphabet to another and using this 'cipher alphabet' as a key (as known as a mapping) to encrypt the text. The encrypted text is then referred to as the 'cipher text'. The case where there is only one cipher alphabet is known as a monoalphabetic substitution cipher. An example is given below:

Alphabet	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
Mapping	K	N	Q	B	G	C	R	H	Y	J	E	V	U	T	D	F	P	O	X	I	L	W	Z	M	S	A

"MONTE CARLO METHODS" → "UDTIG QKOVD UGIHDBX"

A polyalphabetic subsitution cipher will use more than one cipher alphabet to encrypt the text by using a different cipher alphabet for each word, character, or block of charaters. The example below uses a periodic polyalphabetic substition cipher, alternating two different keys every other word:

Alphabet	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
Mapping 1	K	N	Q	B	G	C	R	H	Y	J	E	V	U	T	D	F	P	O	X	I	L	W	Z	M	S	A
Mapping 2	R	D	F	S	K	L	U	Q	X	J	G	A	M	Y	E	C	N	V	P	O	Z	I	T	B	W	H

"MONTE CARLO METHODS" → "UDTIG FRVAE UGIHDBX"

With most ciphers the amount of cipher alphabets used is usually not too large. Therefore in a large text there will usually be repetition of the cipher keys or mappings. A well known example of this is the Vigenre cipher.

The decryption of cipher text encoded using a substitution cipher is usually accomplished using frequency analysis, a type of cryptanalysis. This type of problem is common in the field of cryptography. Recently, the decryption of such cipher text has been accomplished using a combination of Markov Chain Monte Carlo (MCMC) methods and frequency analysis. Frequency analysis considers the frequencies of letters and combinations of letters in the cipher text and tries to match them to the frequency that they appear in a given language. This paper will consider two different methods of frequency analysis: an 'unigram attack' and a 'bigram attack'. The implementation of MCMC methods to decrypt a text encoded with a classical cipher such as a substition cipher was first introduced by Marc Coram and Phil Beineke. They used a method involving MCMC and frequency analysis to

decrypt messages that were brought to them while working at the drop-in statistical consulting service at the Stanford Department of Statistics. The messages were provided by a psychologist working at the California state prison in an attempt to discover what the prisoners were communicating to each other. This case only involved a simple monoalphabetic substitution cipher.

A more in depth study of the decryption of substitution cipher encrypted text using MCMC methods was later performed by Connor[3]. The addition of transposition ciphers and product ciphers was then considered by Chen and Rosenthal[1]. These encryptions were more difficult to crack, but Chen and Rosenthal[1] managed to achieve very high success rates using MCMC methods. All of the ciphers considered in this previous research used monoalphabetic substitution ciphers. This paper will address text encrypted with monoalphabetic subsitution ciphers, like previous research, and then explore whether or not similar methods can be employed for text encrypted with polyalphabetic substition ciphers.

2 Frequency Analysis

2.1 Unigram Attack

One method of frequency analysis used to decrypt substitution ciphers, and perhaps the easiest method, is the unigram attack where only the frequencies of individual letters are examined. Since the frequency of individual letters used in the English language is not uniform across all 26 letters, some letters are used more often than other letters. For example, E is the most commonly used letter while Z is the least commonly used letter in the English language. In the unigram attack we simply replace the most frequently used letter in the encrypted text with the most frequently used letter in the reference text, the second most used in the encrypted text with the second most used in the reference text, and so on. We first test this method on text encrypted using a monoalphabetic substition cipher with a suitable reference text. The reference text used in this paper is *War and Peace* by Leo Tolstoy, a choice well supported by previous research. The plain text that will be encrypted and used to assess the performance of our algorithms is *Flatland* by Edwin A Abbott, which was chosen arbitrarily.

Plain text	THE PROJECT GUTENBERG EBOOK OF FLATLAND BY EDWIN A ABBOTT
Encrypted text	KTM NJCQMPK GBKMDFMJG MFCCY CI IHEKHEDV FL MVRAD E EFFCKK
Decrypted text	TRE YSAJEUT GMTEIBESG EBAAK AW WDOTDOIL BF ELPNI O OBBATT

Unforunately, this attack did not perform very well, which can seen above. Only 8/26 letters were successfully decrypted. Looking at the count of each letter that appears in the reference text, *War and Peace*, ranking from highest to lowest,

E	T	A	O	N	I	H	S	R	D	L	U	M
314833	226029	205441	192834	184158	173751	167035	162886	148059	118282	96523	65429	61644
C	W	F	G	Y	P	B	V	K	X	J	Z	Q
61258	59203	54888	51321	46266	45166	34644	26902	20416	4060	2573	2388	2330

We see that some letters have similar frequencies, such as H & S, U & M, C & W, etc., and are likely to be interchanged in an unigram attack. Thus, a simple unigram attack is not very successful in decrypting a monoalphabetic substitution cipher, and would perform even worse on any polyalphabetic substitution cipher. To demonstrate this we perform the same attack but on text encrypted using a periodic polyalphabetic substition cipher with a period = 2:

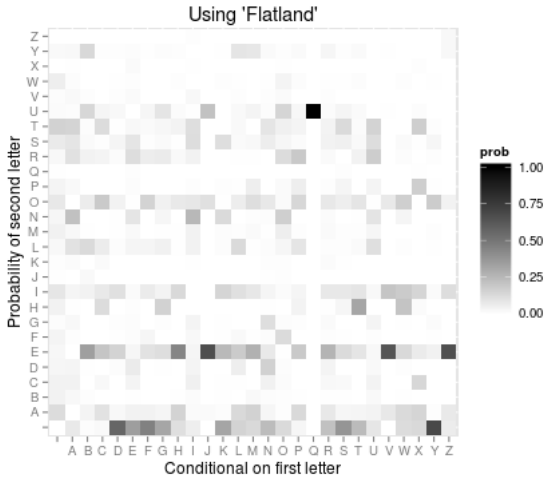
Plain text	THE PROJECT GUTENBERG EBOOK OF FLATLAND BY EDWIN A ABBOTT
Encrypted text	HTP YLHOVPX ELHPRCPDE VIHHB GS MNJXNJZK CZ VKGAZ U JIIHXX
Decrypted text	ECT RLEKNTO YLETSVTWY NZEEF HX PMIOMIDB VD NBHUD A IZZEOO

For this polyalphabetic substition cipher, we compare the mapping obtained from the unigram attack to the two different mappings used to encrypt the plain text. We find that the unigram attack only identified 1 of 26 letters correctly in the first mapping, and none in the second, obtaining an average accuracy of 0.5 letters of out 26.

2.2 Bigram Attacks

Unlike the unigram attack where only the frequencies of individual letters were examined, a bigram attack looks at the frequencies of two consecutive letters. For example, the letter 'Q' is always followed by a 'U', making 'QU' far more likely to appear than 'Q' followed by any other letter. Pairs such as 'ER' or 'TH' are also common.

Using *War and Peace* as the reference text and *Flatland* as the cipher text, we can examine the frequency of each pair on consecutive letters that appear in both texts:



The graphs above show the probabilities of the second letter, conditional on the letter preceding it. For example, the probability that a 'U' follows a 'Q' is 100% and the probability that an 'E' follows a 'H' is $\sim$ 70%. The frequency plots above for both *War and Peace* and *Flatland* are very similar, indicating that they have similar pair-wise frequencies of letters. This is the central idea behind frequency analysis; more specifically, bigram attacks.

Cipher text can be decrypted using this information and a Markov Chain Monte Carlo algorithm. In this paper we will look into two kinds of MCMC algorithms, a Metropolis Algorithm and a Metropolis-within-Gibbs algorithm.

3 MCMC Methods

3.1 Overview of the Metropolis Algorithm

Let $\mathcal{X}$ represent the sample space of some desired distribution, π. The Metropolis algorithm is as follows:

- For a starting value select some $X_0 \in \mathcal{X}$
- For $i = 1, 2, 3, \cdots$
  - Propose $Y_n \in \mathcal{X}$, by letting $Y_n = f(X_{n-1})$, for some symmetric proposal density f.
  - Generate $U_n \sim$ Uniform[0,1], independent of the X_i's and Y_i's
  - If $U_n < \frac{\pi(Y_n)}{\pi(X_{n-1})}$, then 'accept' and set $X_n = Y_n$. Otherwise 'reject' and set $X_n = X_{n-1}$

As shown in Chen & Rosenthal[1], this algorithm can be applied to decipher encrypted text by letting the sample space, $\mathcal{X}$ be the space of all possible keys/mappings. Since there are 26 letters in the English alphabet, $|\mathcal{X}| = 26! \approx 4 \times 10^{26}$. The proposals in each iteration are generated by switching two random letters with each other from the current key/mapping (letters can be swapped with themselves. ie, no change), hence the probability of any proposal is $\frac{1}{n^2}$. Thus, the proposals are symmetric and the Markov Chain is aperiodic. By Chen & Rosenthal[1], the algorithm will converge to solutions with approximately maximal score functions, which is defined below.

3.2 Score Function

The score, $\pi(x)$, used in the MCMC algorithm is the criterion used for assessing the merit of the current key/mapping for each iteration. For each pair of characters, β_1 and β_2, let $r(\beta_1, \beta_2)$ represent the number of times "$\beta_1\beta_2$" appears in the reference text. Then, for some decryption key $x \in \mathcal{X}$, let $f_x(\beta_1, \beta_2)$ represent the number of times "$\beta_1\beta_2$" appears in the cipher text when decrypted using the key, x. For example, if β_1='Q' and β_2='U', then $r(\beta_1, \beta_2)$ would be the frequency that 'QU' appears in the reference text and $f_x(\beta_1, \beta_2)$ would be the frequency that 'QU' appears in the cipher text after decrypted using key x. We then define the score function as,

$$\pi(x) = \prod_{\beta_1,\beta_2} r(\beta_1, \beta_2)^{f_x(\beta_1,\beta_2)} \qquad \text{or} \qquad \log(\pi(x)) = \sum_{\beta_1,\beta_2} f_x(\beta_1, \beta_2) \log(r(\beta_1, \beta_2))$$

The log of the score function is often used to prevent computational calculation errors. The frequency of paired letters of the cipher text should be similar to the reference text, so as the decryption key x gets closer to the correct key we would expect $f_x(\beta_1, \beta_2)$ to get larger. Thus, the higher the score function, the more likely the current decryption key x is correct.

3.3 Metropolis Algorithm

Recall, monoalphabetic substitution ciphers uses one encryption key/mapping to encrypt the cipher text and polyalphabetic substitution ciphers use two or more encryption keys/mappings. For simplicity, in this paper we only deal with periodic polyalphabetic ciphers, where each key/mapping is repeated for a fixed period. For example if period=3 the $1^{st}, 4^{th}, 7^{th}, ...$, word uses one key, the $2^{nd}, 5^{th}, 8^{th}, ...$, words uses another key, and the $3^{rd}, 6^{th}, 9^{th}, ...$ word uses yet another key.

We propose two different algorithms: a Metropolis algorithm which updates all keys/mappings at the same time at each iteration, and a Metropolis-within-Gibbs algorithm which updates only 1 key/map at each iteration.

If the text was encrypted with periodic polyalphabetic substition cipher with period of n, the Metropolis algorithm is as follows:

- For a period of n, let $x_1, x_2, \cdots, x_n$ be n random keys from $\mathcal{X}$.
- Divide the cipher text into n substrings. The first substring containing the $1^{st}, (n+1)^{th}, (2n+1)^{th}, \cdots$, word, the second substring containing the $2^{nd}, (n+2)^{th}, (2n+2)^{th}, \cdots$, word, and so on.
- for $i = 1, 2, 3, \cdots$
  - Propose new mappings, $y_1, \cdots, y_n \in \mathcal{X}$ by switching the position of two letters in each of $x_1, \cdots, x_n$
  - Generate $U \sim$ Uniform[0,1] independant of the x_i's and y_i's
  - If $U_n < \frac{\pi'(y_1, \cdots, y_n)}{\pi'(x_1, \cdots, x_n)}$, then 'accept' and set $x_1 = y_1, \cdots, x_n = y_n$. Otherwise 'reject' and set $x_1 = x_1, \cdots, x_n = x_n$

Where $\pi'(x_1, \cdots, x_n) = \sum_{i=1}^{n} \pi(x_i)$, the sum of the original score function on each of the substrings. Note: $n = 1$ corresponds to the monoalphabetic cipher case done in previous studies.

3.4 Metropolis-within-Gibbs

In the Metropolis-within-Gibbs algorithm, only one of the n keys/mappings are updated in each iteration. The key/mapping to be updated is chosen randomly, making this a random scan Metropolis-within-Gibbs algorithm.

- For a period of n, let $x_1, x_2, \cdots, x_n$ be n random keys from $\mathcal{X}$.
- Divide the cipher text into n substrings. The first substring containing the $1^{st}, (n+1)^{th}, (2n+1)^{th}, \cdots$, word and the second substring containing the $2^{nd}, (n+2)^{th}, (2n+2)^{th}, \cdots$, word, and so on.

- for $i = 1, 2, 3, \cdots$
  - Generate $j \sim \text{Unif}\{1, \cdots, n\}$
  - Propose a new mapping, $y_j \in \mathcal{X}$ by switching the position of two letters in x_j
  - Generate $U \sim$ Uniform[0,1] independant of the x_i's and y_i's
  - If $U_n < \frac{\pi'(y_1, \cdots, y_n)}{\pi'(x_1, \cdots, x_n)}$, then 'accept' and set $x_j = y_j$. Otherwise 'reject' and set $x_j = x_j$

The advantage this method has over the previous method is that it only updates one of the mappings at a time, so intuitively there should be a smaller chance of rejection.

3.5 Scaling Parameter

A possible modification to algorithms above to avoid getting "stuck" in local modes is to include a scaling parameter, p, in the score function. This modification is known is 'tempering'. This would modify the accept/reject step of the above two algorithms so that the acceptance condition is now,

$$U < \left(\frac{\pi(y)}{\pi(x)}\right)^p \qquad \text{or} \qquad \log U < p(\log(\pi(y)) - \log(\pi(x)))$$

A large value for p will keep the algorithm at local modes but will lead to lower acceptance rates (thus a larger number of iterations needed for convergence). Conversely, a smaller value for p will allow for more movement around local modes, but may not converge to the correct distribution.

4 Simulations

In this section we will discuss the results from running multiple simulations using monoalphabetic and polyalphabetic ciphers. A monoalphabetic cipher was examined rigorously in previous work, therefore we will briefly consider these type of ciphers. Our attack on monoalphabetic ciphers will be motivated by the final attack discussed in Chen & Rosenthal[1]. This will help us assess the validity of our algorithm before moving on to the more complicated polyalphabetic cipher.

For polyalphabetic ciphers we will only consider periods of length 2 and 3, where the cipher key/mapping will repeat every 2 and 3 words, respectively. For each period, we will determine the optimal number of iterations necessary for convergence and whether or not a scaling parameter was needed. Based on our findings, we will then consider whether or not the implementation of our algorithm changes as the number of periods increases.

In order to determine how well our algorithm performs, we define the accuracy of our algorithm as the number of correct entries in the estimated cipher key for the current iteration. For example, suppose we have the following:

Correct Key	Q	O	W	Y	X	M	A	P	Z	B	K	I	C	T	E	R	L	N	J	D	H	S	F	U	V	G
Estimated Key	O	Q	W	Y	X	M	A	N	Z	B	K	I	C	T	E	D	L	P	J	R	H	S	F	U	V	G

Clearly, 20 of the 26 letters in the cipher key have been correctly identified in the estimated key. Therefore the accuracy would be $20/26 = 0.77$.

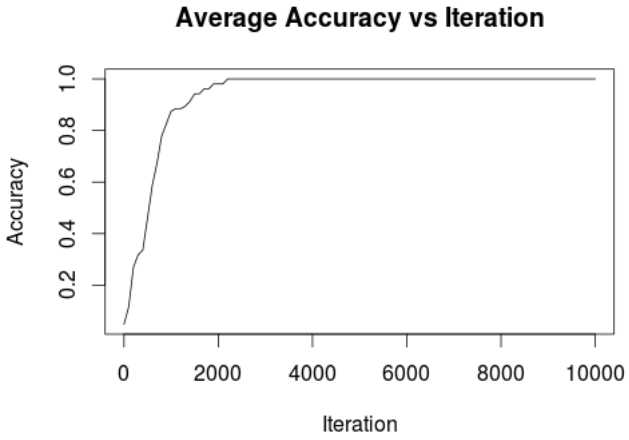
In cases where the period > 1 (so there are >1 mappings/keys), the accuracy would be defined as the average accuracy over all the keys/mappings.

4.1 Period 1

Using the same optimal scaling parameter of $p = 1$ from Chen & Rosenthal[1], we run the algorithm 5 times for 10000 iterations and graph the average accuracy.

Iterations	Accuracy	No. of successful runs (out of 5)
10000	1.00	5

Our algorithm was successful in identifying the correct key/mapping for all 5 independent runs. Next, we plot the average accuracy of all five runs against the iteration number to get a sense of the rate of convergence:



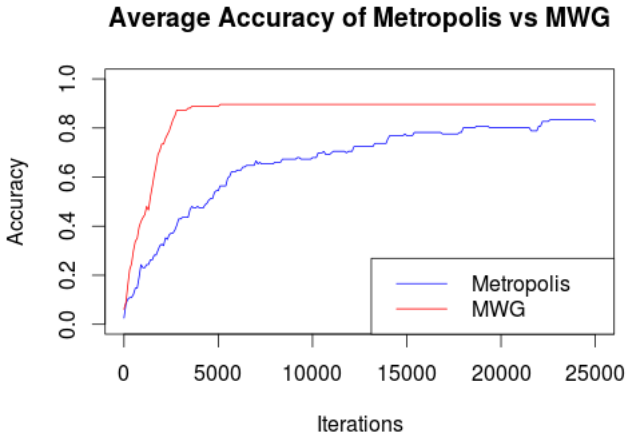
As shown above, and in previous studies, this algorithm is very successful in decoding cipher text encrypted using a monoalphabetic substitution cipher. On average, our algorithm correctly identifies the key/mapping and was able to decipher text encrypted with a monoalphabetic substitution cipher within 5000 iterations. However if we apply this algorithm to text encrypted using a polyalphabetic substition cipher, it performs poorly as shown in the following example.

Original Text	THE PROJECT GUTENBERG EBOOK OF FLATLAND
Attempted Decipher	THE TMSWONR XLTENJERX OGSSD IG KBURBUCH

We've shown above that we were able to successfully decode a monoalphabetic substitution cipher using this MCMC algorithm within 5000 iterations but was unsuccessful in decoding text when encrypted using a polyalphabetic substitution cipher. In the example above, where the text was encrypted using period=2, only 5/26 letters were successfully estimated even after 5000 iterations. Thus further methods need to be investigated in order to decrypt periodic polyalphabetic substitution ciphers.

4.2 Period 2

Two different algorithms can be used to decipher periodic polyalphabetic substitution ciphers with period ≥ 2, the Metropolis algorithm and the Metropolis-within-Gibbs algorithm. To compare the effectiveness of the Metropolis algorithm to the Metropolis-within-Gibbs algorithm, we first run both algorithms for a large number of iterations to get some understanding of the rate of convergence and accuracy of the two algorithms. We simulate 5 independent runs, at 25000 iterations each, for each algorithm and average their accuracies.



It is clear from the plot above that the Metropolis-within-Gibbs algorithm converges much faster, and with higher accuracy, than the Metropolis algorithm on average. The Metropolis-within-Gibbs algorithm appears to converge after approximately 3000 iterations, while the Metropolis algorithm doesn't seem to have quite converged yet, even after 25000 iterations. Therefore for period=2, the Metropolis-within-Gibbs algorithm is superior, both in convergence rate and in accuracy.

Next, in order to determine the optimal number of iterations needed, the Metropolis-within-Gibbs algorithm is executed several times using different number of iterations. The accuracy and number of completely successful runs is then compared in the following table:

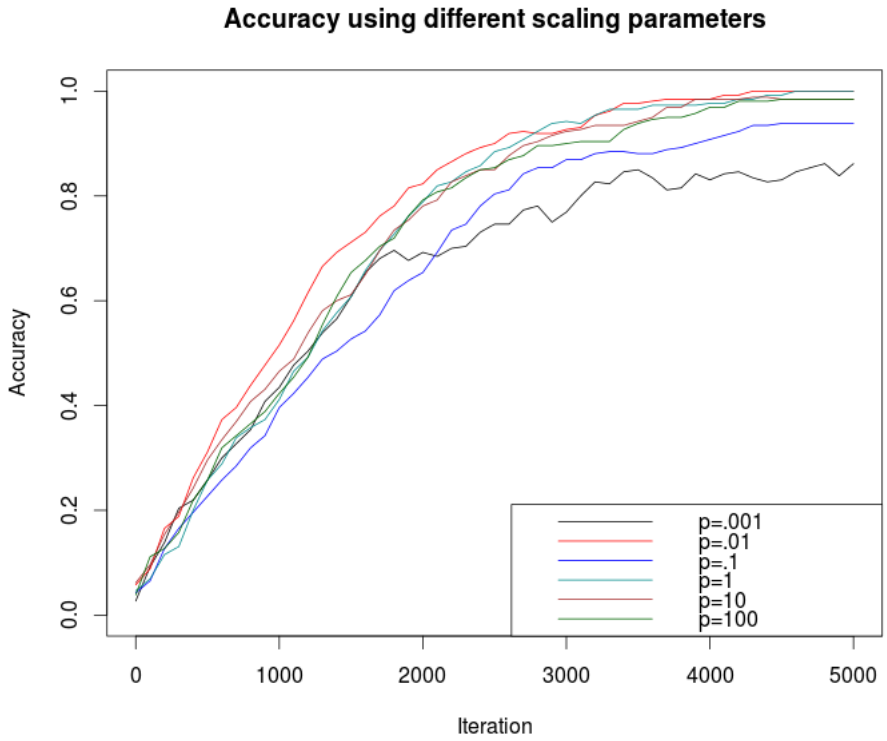
Iterations	Accuracy	No. of successful runs (out of 5)
1000	0.3241	0
5000	1.0000	5
10000	0.9345	3

Clearly, we can see that runs of only 10000 iterations did not prove to be very successful; with no successful runs and with an average accuracy of only 0.32. The accuracy and number of successful runs for the 5000 iteration and 10000 iteration runs were comparable to each other, with similar accuracies. Although the 5000 iteration runs had 5/5 successful runs while the 10000 iteration runs had 3/5 successful runs, this could just be due to chance. Together with the plot above comparing Metropolis to Metropolis-within-Gibbs, we conclude that the Metropolis-within-Gibbs algorithm converges within 5000 iterations so increasing the number of iterations would not affect anything other than the runtime of the algorithm. Therefore it appears that 5000 iterations is the optimal number of iterations to use.

Now that we've obtained the optimal number of iterations to use, we may investigate the optimal scaling parameter for the algorithm.

Scaling Parameter	Accuracy	Acceptance Rate	No. successful runs (out of 5)
0.001	0.86149	0.057	0
0.01	1.000	0.026	5
0.1	0.9114	0.028	3
1	1.000	0.026	5
10	1.000	0.028	5
100	0.985	0.026	4

Having a scaling parameter that is very small increases the acceptance rate at the cost of sacrificing accuracy and number of successful runs. The other values for the scaling parameter where $p \neq 0.001$ were all comparable to each other, with similar acceptance rates, accuracies, and number of successful runs. Furthermore, we can plot the accuracy for each scaling parameter against the number of iterations,



It appears that the scaling parameter did not have any effect on the accuracy of the algorithm in most cases. Excluding the case where $p = 0.001$, we see that the accuracy and convergence rates were all very similar to one another, much like in the above table. When the scaling parameter was very low, as in the case of $p = 0.001$, we observe that the accuracy obtained was much lower, $\approx .80$, which is a 20% reduction compared to the best performing values for the scaling parameter. As discussed previously, a lower p allows for greater movement, evident in the greater volatility seen in the case where $p = 0.001$. However, a lower p may not converge to the correct mapping. Since the scaling parameter doesn't seem to have much of an impact on the accuracy (with an exception for $p = 0.001$),

we choose $p = 1$ as the best scaling parameter to use for simplicity.

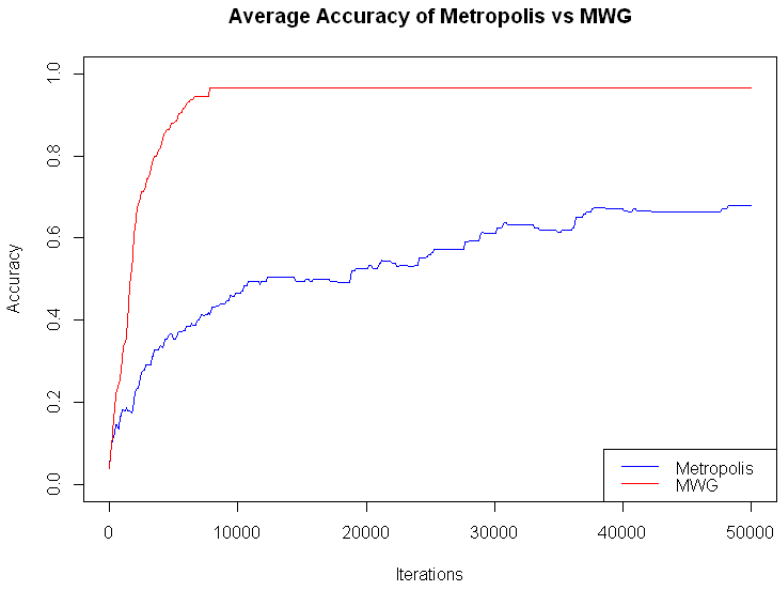
Therefore, We conclude that the best algorithm for decrypting a periodic polyalphabetic substitution cipher, with period=2, would be a Metropolis-within-Gibbs algorithm run for at least 5000 iterations, with scaling parameter $p = 1$ (ie. no scaling).

The following example demonstrates a single run of the Metropolis-within-Gibbs algorithm with scaling parameter $p = 10$ for 10000 iterations on text encrypted using a period=2 polyalphabetic substitution cipher.

Iteration	Cipher Text
0	VCK ZAUVEPH UAVKHQKBU EFUUY ZR SQLHQLDO QS EONCD Y LFFUHH
1000	TOE PROZELS DUTENBERD EVOOK AG FGISGIND BY EDWAN L IVVOSS
2000	THE PROJECS GUTENBERG EBOOK LO FLASLAND BY EDWIN A ABBOSS
3000	THE PROJECT GUTENBERG EBOOK LO FLATLAND BY EDWIN A ABBOTT
4000	THE PROJECT GUTENBERG EBOOK OM FLATLAND BY EDWIN A ABBOTT
5000	THE PROJECT GUTENBERG EBOOK OF FLATLAND BY EDWIN A ABBOTT

4.3 Period 3

Similar to the previous case, the effectiveness of the Metropolis algorithm is compared to the Metropolis-within-Gibbs algorithm. Since the period is larger we assumed that the number of iterations needed to reach convergence will also be larger. Thus, to get an understanding of the rate of convergence and accuracy of the two algorithms, we simulate 5 independent runs, at 50000 iterations each, for each algorithm and plot their average accuracies.



Like in the case where period=2, the Metropolis-within-Gibbs algorithm converges, on average, much faster than the Metropolis algorithm and with a higher accuracy. Metropolis-within-Gibbs converges within 10000 iterations while it appears that the Metropolis algorithm takes at least 40000 iterations to converge and with a much lower accuracy. Therefore for period=3, Metropolis-within-Gibbs is again superior to Metropolis in both convergence rate and accuracy.

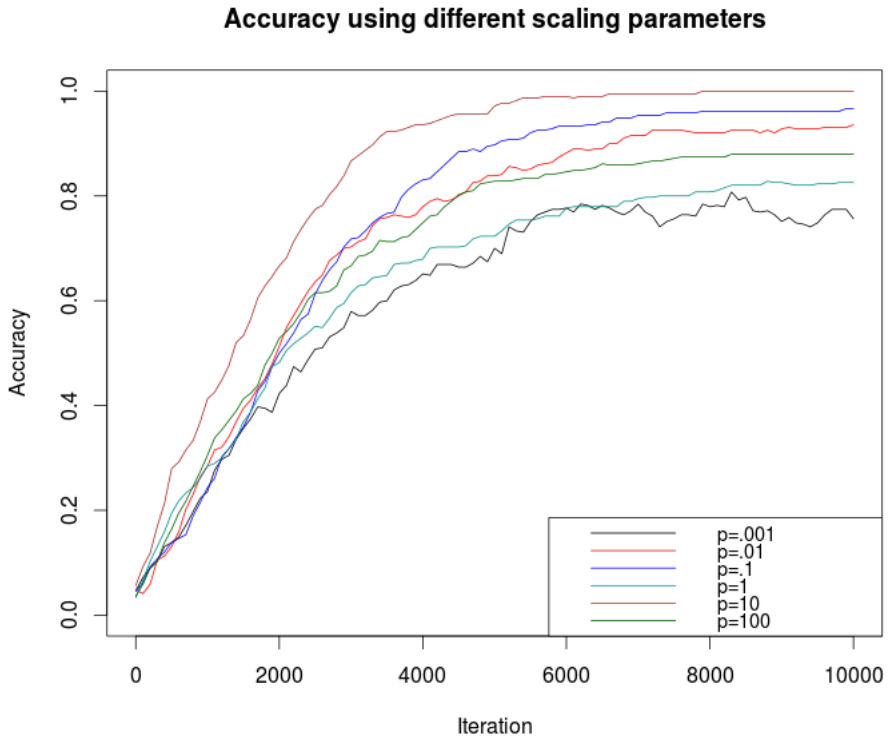
Now that we've determined the better of the two algorithms, the optimal number of iterations needed for a period=3 was determined by running the Metropolis-within-Gibbs algorithm several times with a varying number of iterations then comparing the accuracy and number of successful runs.

Iterations	Accuracy	No. of successful runs (out of 5)
1000	0.397	0
5000	0.776	1
10000	0.946	2

From the table above, we observe that the accuracy and the number of successful runs seemed to increase with the number of iterations used. The number of successful runs for 5000 iterations and 10000 iterations were comparable but could be due to chance and the small number of simulations. The 10000 iteration run had the highest accuracy and in the graph above, with 50000 iteration runs, indicates that convergence occurs around iteration 7500. Thus we conclude that 10000 iterations would be the optimal number of iterations for the case where period=3. Next we investigate the optimal value for the scaling parameter by running multiple simulations with different values of p, each with 10000 iterations.

Scaling Parameter	Accuracy	Acceptance Rate	No. successful runs (out of 5)
0.001	0.885	0.0679	0
0.01	0.936	0.0233	2
0.1	0.967	0.0217	4
1	0.826	0.0195	1
10	1.000	0.0181	5
100	0.879	0.0199	3

Similarly to when period=2, the biggest difference was when the scaling parameter was very small, $p = 0.001$. The acceptance rate was higher than the rest at the cost of accuracy and number of successful runs. The rest of the scaling parameters performed very similar to each other in terms of accuracy, acceptance rate, and the number of successful runs. One observation to note is that only 1/5 successful runs were obtained for when $p = 1$, and could be just due to bad luck. The only choice for the scaling parameter to achieve an 5/5 successful run was $p = 10$. Next we can also plot the accuracy against the iteration for the different scaling parameters to help identify the optimal choice for p.



Using accuracy and rate of convergence as the criteria for choosing the optimal scaling parameter, $p = 10$ was the most successful. Unlike the case where period=2, there is a greater divergence between the different choices of scaling parameters. It is especially evident that for a very low choice for the

scaling parameter, such as $p = 0.001$, leads to higher volatility and reduction in level of accuracy, much like in the previous section using period=2.

Therefore, the optimal algorithm for decrypting a periodic polyalphabetic substitution cipher with period=3 would a Metropolis-within-Gibbs algorithm, run for at least 10000 iterations, with a scaling parameter of $p = 10$.

The following example demonstrates a single run of the Metropolis-within-Gibbs algorithm with scaling parameter $p = 10$ for 10000 iterations on text encrypted using a period=3 polyalphabetic substitution cipher.

Iteration	Cipher Text
0	SLM JCGQANU HUCENIEQH MGFFO GB WXBCXBNV GJ ATYID B EGGFSS
1000	SHE BSOQECT GOTENBERG EBAAX OF FMATMAND BL ERGIN A IBBASS
2000	THE PSOZECT GUTENLERG EBAAX OF FMATMAND BF EGRIN A OBBATT
3000	THE PROZECT GUTENBERG EBAAK OD FMATMAND BY EGWIN A OBBATT
4000	THE PROJECT GUTENBERG EBOOK OD FLATLAND BY EFWIN A ABBOTT
5000	THE PROJECT GUTENBERG EBOOK OD FLATLAND BY EFWIN A ABBOTT
6000	THE PROJECT GUTENBERG EBOOK OD FLATLAND BY EFWIN A ABBOTT
7000	THE PROJECT GUTENBERG EBOOK OD FLATLAND BY EFWIN A ABBOTT
8000	THE PROJECT GUTENBERG EBOOK OF FLATLAND BY EDWIN A ABBOTT
9000	THE PROJECT GUTENBERG EBOOK OF FLATLAND BY EDWIN A ABBOTT
10000	THE PROJECT GUTENBERG EBOOK OF FLATLAND BY EDWIN A ABBOTT

5 Period Inference

Thus far, in our analysis of polyalphabetic ciphers we have assumed that we know the period in which the sequence of keys/mappings repeats. This assumption was needed to analyze the performance of our algorithm for different periods, but in practice is quite unreasonable. There would be no reason to assume that we would have perfect knowledge of the actual period. Thus, in any practical applications we would first need to identify the period. One possible method of inference would be to estimate some upper bound for the period and run the Metropolis-within-Gibbs algorithm with the period set to the upper bound. In theory the algorithm should converge with repeated mappings being equal to each other. It is important to note that this will only happen if the upper bound is a multiple of the true period. For example, suppose the encryption period is 2 and the period used for decryption is 4, letting individual letters represent whole words and the key used to encrypt that word,

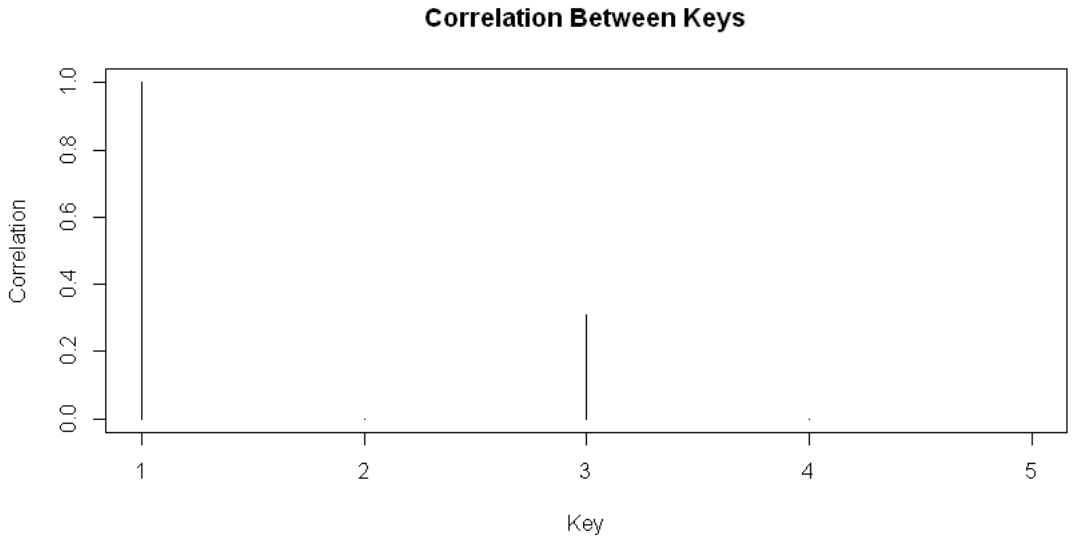
Encryption period	A	B	A	B	A	B	A	B
Decryption period	C	D	E	F	C	D	E	F

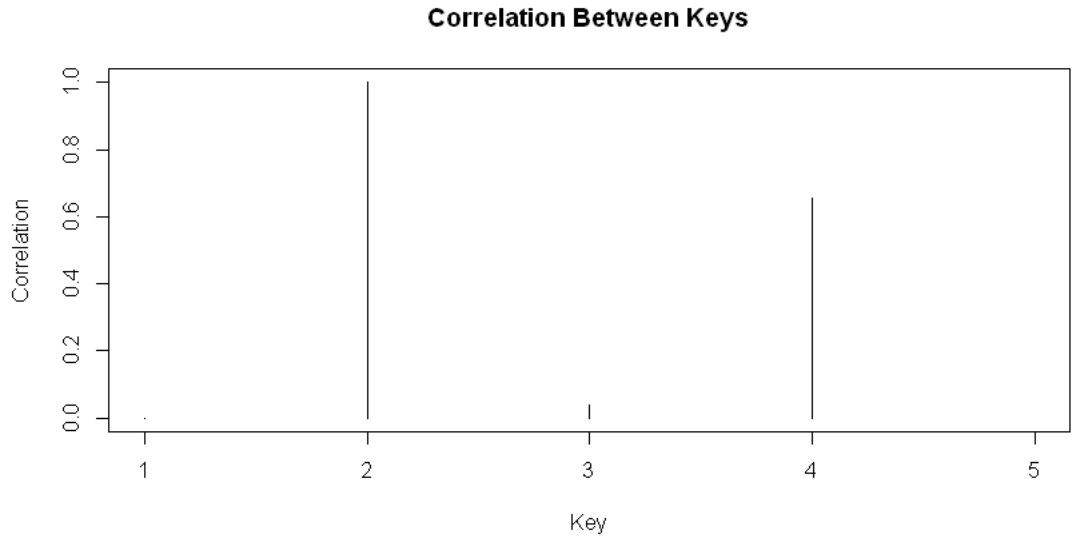
We can see in the above example that key 'C' & 'E' will converge to 'A' and 'D' & 'F' will converge to 'B', thus we see that the encryption period is 2. Now suppose the encryption period is still 2, but the decryption period is now 3, then we have the following,

Encryption period	A	B	A	B	A	B	A	B
Decryption period	C	D	E	C	D	E	C	D

Clearly in this case the algorithm will not converge as 'C', 'D', and 'E' are mapped to both 'A' and 'B'.

Since the running time of the algorithm increases as the period increases, it would be wise to run the MWG algorithm for a lower number of iterations. The number of iterations only needs to be enough to identify if there is a correlation between the keys/mappings that can be used to determine the correct period. We illustrate this using a simple example where the actual period is 2 and we believe the period ≤ 4. In this case we will run a MWG algorithm with the period equal to 4 and the scaling parameter, p=1. After 2000 iterations we check the correlations between the keys/mappings. Below are two graphs displaying the correlation between the keys. The first plot compares the first key to the others and the second plot compares the second key to the others.





Observing the first plot there it is clear there is a strong correlation between the first key and the third key with very little correlation to the second or fourth key, indicating that the period is of length 2. In the second plot there is an even stronger correlation between the second key and the fourth key, with almost no correlation to the first or third. Therefore, after only 2000 iterations it is quite evident that the period is indeed two. We may now run our optimal algorithm from earlier with a period of two to decrypt the cipher text. It is important to remember that this only worked because four is a multiple of two. If we had used a period of five instead of two it would not have converged or shown any meaningful correlations. This implies that in practice it would generally be quite difficult to identify the period as there would be a lot of time spent through trial and error.

6 Conclusion

In this paper we have managed to successfully replicate algorithms that can decrypt monoalphabetic substitution ciphers like in previous research. Since monoalphabetic substitution ciphers were been previously studied extensively, they were only briefly examined to assess the validity of our initial algorithm. We were then able to expand upon this by developing efficient algorithms that can also decrypt periodic polyalphabetic substitution ciphers. Due to lack of computational power and time we restricted the periods examined to be only of lengths 2 and 3, although our algorithm will work for any period. For polyalphabetic substitution ciphers we determined that a Metropolis-within-Gibbs algorithm performed much better than the standard Metropolis algorithm. The optimal Metropolis-within-Gibbs algorithm ran quickly, being able to decrypt cipher text within 5000 iterations for a period of length 2 with no scaling parameter used, and within 10000 iterations for a period of length 3 with scaling parameter $p = 10$. We then briefly examined possible methods of inferring

the actual period used in text encryption when it is unknown. Our methods consisted of attempts at decrypting the cipher text using a large period, and then examining the correlations between the cipher keys/mappings. However, this method leads to an abundance of trail-and-error so further study into period inference is warranted. The effectiveness of Markov Chain Monte Carlo methods used to decrypt monoalphabetic and polyalphabetic substitution ciphers has been demonstrated in this paper, further strengthening the merits of MCMC methods for the decryption of cipher texts.

7 Appendix

7.1 R Code

```
# Removes all punctuation, numbers, multiple spaces from a string input
textonly <- function(x) {
  x <- str_replace_all(x, "[[:punct:]]", "")   # Removes all symbols
  x <- str_replace_all(x, "[[:digit:]]", "")   # Removes all numbers
  x2 <- str_replace_all(x, "  ", " ")
  while (x2 != x) {
    x <- x2
    x2 <- str_replace_all(x2, "  ", " ")
  }
  x2
}

# Decrypts or encrypts a string using a specified mapping
# decipher=TRUE: input an ENCRYPTED string, outputs decrypted string using mapping
# decipher=FALSE: input an UNENCRYPTED string, encrypts it using specified mapping
decipher <- function(input, mapping, decipher=TRUE) {
  mapping <- paste(mapping, collapse="")
  letter <- paste(toupper(letters), collapse="")
  input <- toupper(input)
  if (decipher) { output <- chartr(mapping, letter, input) }
  else {output <- chartr(letter, mapping, input)}
  output
}

# Calculates Score using specified mapping
score <- function(string_input, mapping) {
  string_deciphered <- decipher(string_input, mapping)
  temp <- embed(unlist(strsplit(toupper(string_deciphered), '')), 2)
  temp.count <- table( temp[,2], temp[,1])
  temp.melt <- melt(temp.count)

  let <- c(' ', toupper(letters) )
  score= 0
  for (i in let) {
    for (j in let) {
      r.ij <- pair.count$value[which(pair.count$Var1==i & pair.count$Var2==j)] + 1
      fx.ij <- temp.melt$value[which(temp.melt$Var1==i & temp.melt$Var2==j)] + 1
      if (is.na(fx.ij) || length(fx.ij) == 0) { fx.ij = 1 }
      score = score + fx.ij*log(r.ij)
    }
  }
  score
}

# Attempts to decrypt simple substitution cipher using MCMC
# inputs: encrypted string; p = scaling parameter (tempering);
# iterations = max no. of iterations;
# start.map= mapping to start with (random by default)
run.mcmc.decipher <- function(to_decipher, p=1, iterations=2000,
     start.map = sample(toupper(letters)) ) {
  mapping <- start.map
```

```
  current.decipher <- decipher(to_decipher, mapping)
  current.score <- score(to_decipher, mapping)

  max.decipher <- current.decipher
  max.score <- current.score

  i <- 1
  numaccept <- 0
  list <- NULL
  while( i <= iterations) {
    # Randomly switch 2 letters in the mapping
    proposal <- sample(1:26, 2)
    prop.mapping <- mapping
    prop.mapping[proposal[1]] = mapping[proposal[2]]
    prop.mapping[proposal[2]] = mapping[proposal[1]]

    prop.score <- score(to_decipher, prop.mapping)

    if( log(runif(1)) < p*(prop.score - current.score) ) {
      mapping = prop.mapping
      current.score <- prop.score
      current.decipher <- decipher(to_decipher, mapping)

      if( current.score > max.score) {
        max.score = current.score
        max.mapping = mapping
      }
      numaccept <- numaccept+1
    }
    cat(i, substring(current.decipher,1,50), current.score, '\n')
    if (i %% 100 ==0 ) {
      list <- rbind(list, c(i, substring(current.decipher, 1, 70), current.score))
    }
    i=i+1
  }
  ret <- list(max.score = max.score, max.mapping = max.mapping, list=list,
     numaccept= numaccept)
  return(ret)
}

# Splits a string into "period" substrings
text.split <- function(x, period) {
  x <- unlist(strsplit(x," "))  # Splits string into words
  string = matrix(0, nrow=ceiling(length(x)/period) ,ncol=period)

  for ( i in 1:ceiling(length(x)/period)) {
    temp <- NULL
    for (j in (period*(i-1) +1) :(i*period)) {
      if ( is.na(x[j])) { x[j] = ""}
      temp <- c(temp, x[j])
    }
    string[i,] = temp
  }
  out <- list()
  for (i in 1:period) {
    out[[i]] <- paste(string[,i], collapse=" ")
  }
```

```
  unlist(out)
}

# Decrypts substitution cipher with different mapping every period
decipher2 <- function(input.string, map.list, period, ...) {
  string.split <- text.split(input.string, period)
  deciphered <- list()
  for (i in 1:period) {
    deciphered[[i]] <- decipher(string.split[i], map.list[[i]], ...)
    deciphered[[i]] <- unlist(strsplit(deciphered[[i]], " "))
  }

  if (period > 1) {
    n <- length(deciphered[[1]])
    for (j in 2:period) {
      if (length(deciphered[[j]]) < n) { deciphered[[j]] <- c(deciphered[[j]], "")}
    }
  }
  recombine.mat <- NULL
  for (i in 1:period) {
    recombine.mat <- cbind(recombine.mat, deciphered[[i]])
  }
  recombined <- paste(as.vector(t(recombine.mat)), collapse=" ")
  textonly(recombined)
}

# Calculates score of each substring, and adds them up
score2 <- function( input, maplist, period) {
  input.split <- text.split(input, period)
  score.list <- NULL
  score.sum <- 0
  for (i in 1:period) {
    score.list[i] <- score(input.split[i], maplist[[i]])
    score.sum <- score.sum + score.list[[i]]
  }
  score.sum
}

# MCMC decipher with periods
run.mcmc.decipher2 <- function(to_decipher, period, p=1, iterations=2000) {
  map.list <- list()
  for (i in 1:period){
    map.list[[i]] <- sample(toupper(letters))
  }

  current.decipher <- decipher2(to_decipher, map.list, period)
  current.score <- score2(to_decipher, map.list, period)

  max.decipher <- current.decipher
  max.score <- current.score

  i <- 0
  numaccept <- 0
  list <- list()
  while( i <= iterations) {
    # Randomly switch 2 letters in the mapping
    proposal.list <- list()
```

```
    prop.mapping.list = map.list
    for (j in 1:period){
      proposal.list[[j]] <- sample(1:26, 2)
      prop.mapping.list[[j]][proposal.list[[j]][1]]=map.list[[j]][proposal.list[[j]][2]]
      prop.mapping.list[[j]][proposal.list[[j]][2]]=map.list[[j]][proposal.list[[j]][1]]
    }
    prop.score <- score2(to_decipher, prop.mapping.list, period)

    if( log(runif(1)) < p*(prop.score - current.score)) {
      map.list = prop.mapping.list
      current.score <- prop.score
      current.decipher <- decipher2(to_decipher, map.list, period)

      if( current.score > max.score) {
        max.score = current.score
        max.mapping.list = prop.mapping.list
      }
      numaccept = numaccept + 1
    }

    if (i %% 100 ==0 ) {
      accuracy.count <- rep(0,period)
      for (n in 1:period) {
        accuracy.count[n] <- sum(map.list[[n]] == starting.map[[n]])
      }
      accuracy <- accuracy.count/26
      list <- rbind(list, c(i,substring(current.decipher,1,70), current.score,accuracy))
      colnames(list) <- c('iteration', 'decoded string', 'score', rep('accuarcy',period))
    }
    cat(i, substring(current.decipher,1,50), current.score, accuracy, '\n')
    i=i+1
  }
  ret <- list(max.score = max.score, start.mapping = starting.map,
              max.mapping.list = max.mapping.list,
     list=list, numaccept = numaccept)
  return(ret)
}

# MCMC decipher with periods, using random scan updating 1 map at a time
run.mcmc.decipher.random <- function(to_decipher, period, p=1, iterations=2000) {
  map.list <- list()
  for (i in 1:period){
    map.list[[i]] <- sample(toupper(letters))
  }

  current.decipher <- decipher2(to_decipher, map.list, period)
  current.score <- score2(to_decipher, map.list, period)

  max.decipher <- current.decipher
  max.score <- current.score

  i <- 0
  numaccept <- 0
  list <- list()
  while( i <= iterations) {
    # Randomly switch 2 letters in the mapping
    # random scan through no. of periods
```

```
    prop.mapping.list = map.list
    coord <- floor(runif(1,0,period)) + 1
    proposal.switch <- sample(1:26, 2)
    prop.mapping.list[[coord]][proposal.switch[1]]=map.list[[coord]][proposal.switch[2]]
    prop.mapping.list[[coord]][proposal.switch[2]]=map.list[[coord]][proposal.switch[1]]

    prop.score <- score2(to_decipher, prop.mapping.list, period)

    if( log(runif(1)) < p*(prop.score - current.score)) {
      map.list = prop.mapping.list
      current.score <- prop.score
      current.decipher <- decipher2(to_decipher, map.list, period)

      if( current.score > max.score) {
        max.score = current.score
        max.mapping.list = prop.mapping.list
      }
      numaccept = numaccept + 1
    }

    # Saves information every 100 iterations
    if (i %% 100 ==0 ) {
      accuracy.count <- rep(0,period)
      for (n in 1:period) {
        accuracy.count[n] <- sum(map.list[[n]] == starting.map[[n]])
      }
      accuracy <- accuracy.count/26
      list <- rbind(list, c(i, substring(current.decipher, 1, 60), current.score, accuracy))
    }

    # Prints out current iteratoin info
    cat(i, substring(current.decipher,1,60), current.score, accuracy, '\n')
    i=i+1
  }
  colnames(list) <- c('iteration', 'decoded string', 'score', rep('accuarcy', period))
  ret <- list(max.score = max.score, start.mapping = starting.map,
     max.mapping.list = max.mapping.list, list=list, numaccept = numaccept)
  return(ret)
}

# Creates 'period' random mappings
create.map <- function(period) {
  map.list <- list()
  for (i in 1:period) {
    map.list[[i]] <- sample(toupper(letters))
  }
  map.list
}

# Unigram attack. replaces most frequent letter in encrypted text with most frequent
# letter in reference text, etc
unigram <- function(test) {
  let <- unlist(strsplit(training, ''))
  let.count <- table(let)
  let.count <- let.count[-1]
  let.sorted <- sort(let.count, decreasing = TRUE)
```

```
  coded.char <- unlist(strsplit(test, ''))
  coded.count <- table(coded.char)
  coded.sorted <- sort(coded.count[-which(names(coded.count) == ' ')], decreasing=TRUE)

  lettersNotInCoded<-toupper(letters)[which(toupper(letters)%in%names(coded.sorted)==FALSE)]
  lettersNotInCoded<-sample(lettersNotInCoded)
  coded.sorted2 <- c(coded.sorted, rep(0, length(lettersNotInCoded)))
  names(coded.sorted2) <- c(names(coded.sorted), lettersNotInCoded)

  mapping = NULL

  for (i in 1:length(letters)) {
    mapping[i] = names(coded.sorted2)[which(names(let.sorted) == toupper(letters)[i])]
  }
  new.test<- decipher(test, mapping)
  ret <- list(deciphered = new.test, mapping = mapping)
  ret
}


# Set up Reference Text
readfile <- readLines('warandpeace.txt')            # Read in War and Peace text file
training <- toupper(paste(readfile, collapse=' '))  # Collapses it to one line
training <- textonly(training)

# Creates a table of number of times (row letter) is followed by (column letter)
let <- unlist(strsplit(training, ''))
let2 <- embed(let, 2)
mat.count <- table( let2[,2], let2[,1] )
pair.count <- melt(mat.count)

# Divides each number in a row by its row sum. Frequency
mat.trans <- sweep( mat.count, 1, rowSums(mat.count), FUN='/')
```

References

[1] J. Chen, & J. Rosenthal (2011), *Decrypting Classical Cipher Text Using Markov Chain Monte Carlo*, Statistics and Computing, **22(2)**, 397-413

[2] P. Diaconis (2008), *The Markov Chain Monte Carlo Revolution*, Bull. Amer. Math. Soc., Nov. 2008

[3] S. Connor (2003), *Simulation and solving substitution codes*, Master's thesis, Department of Statistics, University of Warwick.

[4] Project Gutenberg, `http://www.projectgutenberg.org/`