Application of MCMC Algorithm to Decrypt Simple Substitution Ciphers

Simone Di Lorenzo[1], Luca Barattini[2]

Lab of Computer Security
Ca' Foscari University
Student ID: 889833[1], 888528[2]

***Abstract*—In this study, we present our efforts in designing an effective decrypting algorithm for breaking simple substitution ciphers. Employing a Metropolis-Hastings Markov Chain Monte Carlo (MCMC) approach, we implemented a novel algorithm to decipher encrypted texts. The methodology involves iteratively proposing and accepting/rejecting substitutions, allowing the algorithm to explore potential solutions in the solution space. Our results, obtained through extensive experimentation, demonstrate the algorithm's high efficacy. Specifically, we achieved an average successful decryption rate of 98% on a test text comprising half a million words. The findings underscore the potential of MCMC techniques in breaking simple substitution ciphers, offering a promising avenue for cryptanalysis in the realm of classical cryptography.**

***Index Terms*—Substitution cipher, Markov Chain Monte Carlo, Metropolis-Hasting Algorithm, Python.**

I. Introduction

In the fascinating field of cryptography, ciphers play a pivotal role in transforming clear, understandable text into an obscured format known as ciphertext. This encryption process masks the true essence of the message, ensuring confidentiality. Throughout our project, we have focused on cracking simple substitution ciphers, where each letter in the original text is substituted with an alternate letter.

Before delving deeper into the report, it's essential to introduce some of the concepts that have been the building blocks of our approach. Firstly, we will have an exploration of the n-gram model, a great tool for understanding patterns in language. Later, we will move on by looking at the scoring function of our algorithm and then, lastly, we will focus on Markov Chains and Monte Carlo distribution.

II. Substitution Cipher

A substitution cipher replaces each letter in a text with a different letter, following some established mapping. A simple example of substitution cipher is called the Caesar cipher, or shift cipher. In this approach, each letter is replaced with a letter some fixed number of positions later in the alphabet. This feature make the shift cipher easily attackable with a brute force attack, since there are only 25 possible shifts.

In this paper, we will focus on cipher text which maps are randomly established. Since this kind of cipher, have 26! possible combination, it becomes computably impossible to perform a brute force attack, at least in a reasonable time span. However, every kind of substitution cipher is vulnerable to frequency analysis attacks, because they maintain, in the encrypted text, information about the statistical features of the language.

III. The N-Gram Model

N-grams are contiguous sequences of 'n' items, typically words or characters, extracted from a text corpus. By analysing these sequences, we can observe and quantify the likelihood of certain combinations occurring in regular language use. For instance, in English, certain letter pairs or triplets are more common than others, and, in this case, an n-gram model trained on a large enough text would for sure help us to point them out.

It follows that, when we were in the first phases of designing and developing our model, we took advantage of this technique. Understanding which letter substitutions are most probable enabled our algorithm to make educated guesses about the most likely substitutions in the encrypted text. By applying the n-gram model, we move beyond random guessing and harness the predictive power of language regularities, making our decryption process not only more efficient but also more accurate.

IV. Log-Likelihood

The second theoretical concept we have decided to explore in this report is our scoring technique. However, it is crucial for the full understanding of it to start from the very beginning. As it happens quite often, in the early stages of our project, we encountered a crucial challenge with our scoring mechanism, which is evident from the original code (See Listing 1).

Here's what the original process looked like: after having applied the current decryption key to the text, the algorithm identified each bigram in the decrypted output. These bigrams were then matched against their corresponding

Listing 1: Previous (unefficient) scoring function

```
def score_text(cipher_text, bigram_frequencies):
    text = cipher_text.upper()
    text = re.sub(r'[^A-Z]', '', text)
    bigrams = [text[i:i+2] for i in range(len(text)-1)]
    bigram_counts = Counter(bigrams)

    score = 0
    for bigram, count in bigram_counts.items():
        if bigram in bigram_frequencies:
            score += bigram_frequencies[bigram] * count
    return score
```

frequencies in a reference text, as indicated by our bigram frequency model. The scoring of the decrypted text was quite straightforward, it involved the accumulation of these frequencies, summing them up to yield a final score.

Now let's consider an example scenario where we have a decrypted text "HELLO" and a reference bigram frequency model with the following probabilities {'HE': 0.05, 'EL': 0.04, 'LL': 0.03, 'LO': 0.02}. The function first converts "HELLO" into bigrams ['HE', 'EL', 'LL', 'LO']. Then, it counts the frequency of each bigram in the decrypted text. In our example, each bigram appears once, so their counts are all 1.

Last but not least, the function will compute the score by summing the product of each bigram's frequency in the decrypted text and its corresponding frequency in the reference model. For our example:

$$Score = (f\text{HE} \times P(\text{HE})) + (f\text{EL} \times P(\text{EL})) + ...$$
$$Score = (1 \times 0.05) + (1 \times 0.04) + (1 \times 0.03) + (1 \times 0.02)$$
$$Score = 0.05 + 0.04 + 0.03 + 0.02 = 0.14$$

From this example, the mathematical limitation that emerges is obvious: the function doesn't account for the relative rarity or commonness of bigrams in a more sophisticated statistical way. For example, a rare but significant bigram would contribute the same to the score as a common but less informative one.

A. *How did we solve this issue?*

Luckily, the log likelihood function came into help. This approach brought a significant shift in how we assessed the decrypted text, particularly through the use of logarithmic values for probability calculations.

Let's illustrate it with a simple example: Suppose we have two bigrams in an encrypted text, "QZ" (a rare bigram) and "TH" (a common bigram). Let's say that the probability of finding the bigram "QZ" in our reference model is very low (0.001) due to its rarity, while "TH" is quite common with a probability of 0.05. When we apply the log likelihood function, the log of 0.001 (probability of "QZ") results in a larger negative value (about -6.91 if we use natural logarithm). The log of 0.05 (probability of "TH") gives a smaller negative value (around -2.99).

Each time these bigrams occur in the decrypted text, their logarithmic probabilities contribute to the overall log likelihood score. The rarer "QZ" bigram, with its larger negative log value, impacts the score more significantly than the common "TH" bigram.

Based on this principle, it's important to note that our algorithm, trained on extensive text, generally produces a cumulative score that is close to zero from the negative side. This will happen because the text tends to contain many common bigrams, which are frequent in the reference text, and only a few rare bigrams.

V. Markov Chain Monte Carlo

The third macro-concept we have used is part of the so called MCMC methods. They are a family of algorithms that uses Markov Chains to perform Monte Carlo estimate. Its name gives us a hint about its two components, Monte Carlo methods and Markov Chains. Let us understand them separately and in their combined form.

Monte Carlo methods, with their ability to simulate random processes thousands or even millions of times, offer a powerful tool for estimating probabilities in scenarios where traditional analytical solutions are not feasible.

Let's say, for example, we were asked to calculate the area falling under the curve in Figure 1, this will require integrating over a complex analytical formula. However, using the Monte Carlo method, we will randomly generate red dots (more dots for more accuracy) in the rectangle and calculate the ratio of dots falling under the curve w.r.t dots falling in the entire rectangle—the ratio will provide us with the area, given the area of the rectangle.

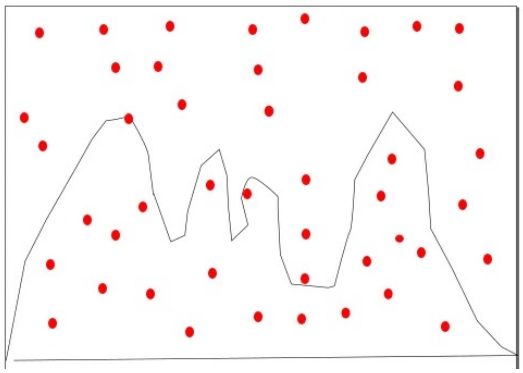

Fig. 1: Monte Carlo to estimate the area under the curve

Now, if we generalize this concept, to calculate some quantity that has a complex analytical structure, we can simply perform a simulation to generate lots of samples and use them to approximate the quantity.

However, this method has a limitation, it assumes to easily sample from a probability distribution, but doing so is not always possible. Sometimes, we can't even sample from the distribution. In those cases, we make use of Markov chains to efficiently sample from a complex probability distribution.

Therefore, through the Markov Chain, we can model sequences of events or states, where the probability of transitioning from a state to another depends solely on the current state and not on the sequence of events that preceded it. In our specific application, means that to generate the proposed cipher we randomly swap two letters, without considering information generated in the previous attempts.

This 'memoryless' property drastically simplifies the analysis of complex systems. When combined, these two components can become an extremely powerful tool. MCMC leverages the random sampling of Monte Carlo methods and the probabilistic framework of Markov Chains to explore the vast and complex solution space of substitution ciphers systematically.

A. Metropolis-Hastings Algorithm

In concluding our theoretical examination of MCMC methods, let's focus on the specific algorithm implemented in our project. For our research, we have decided to use the Metropolis-Hastings Algorithm. This algorithm, initiates in a random state and, during each iteration, proposes a transition to a new state. This proposal is guided by a predefined probability function, often denoted as 'g'.

The acceptance probability of this new state, denoted as 'A', is then calculated and, now, the algorithm will enter the last probabilistic part. Indeed, according to the theory, a probabilistic 'coin flip' will take place deciding whether to accept or reject the proposed state. If the coin—flipped with a likelihood equal to the acceptance probability—lands heads, the new state is accepted; otherwise, it's rejected. This decision-making process is repeated over a significant duration, allowing the algorithm to explore various states thoroughly.

VI. Actual Real Coding

Now, we'll transition from the theoretical framework to practical implementation and delve into the actual code that brought our project to life (For the detailed implementation, please refer to Appendix A).

A. Computing N-Gram Probabilities

Let's start with the `compute_ngram_probabilities` function (See Listing 4). The aim of this first function is to compute the n-frequency per each n-gram in the reference text. Imagine we want to compute the bigram frequency (n=2) of a text made of 900,000 characters there will be $(900,000-1)$ bigrams and, when we pass the reference text to this function, it will return the probability per each bigram in the reference text against all the others.

B. Scoring Function: Log-likelihood

Moving onto the second function (See Listing 2), here, we can see implemented what discussed in section IV-A; from summing linear probabilities, to natural log.

Listing 2: Scoring function using log-liklihood

```
def log_likelihood(text, ngram_probs,
    n):
    likelihood = 0
    text = re.sub(r'[^a-zA-Z]', '',
        text).lower()
    for i in range(len(text) - n + 1):
        ngram = text[i:i+n]
        likelihood +=
            math.log(ngram_probs.get(ngram,
            1e-8))
    return likelihood
```

Maintaining our data-driven approach carried out in the research, we decided to plot a graph showcasing the difference in 100 tests between the accuracy achieved by the new logarithmic scoring mechanism and the older one (See Fig. 2). To compute the accuracy, we compared the generated cipher with the correct cipher and looked at how many letters were in the correct place.

$$accuracy = \frac{correctly\ placed\ letters}{26} \tag{VI.1}$$

The results were clear, our new approach not only managed to maintain a significantly higher accuracy but also outperformed the older mechanism from a volatility perspective.

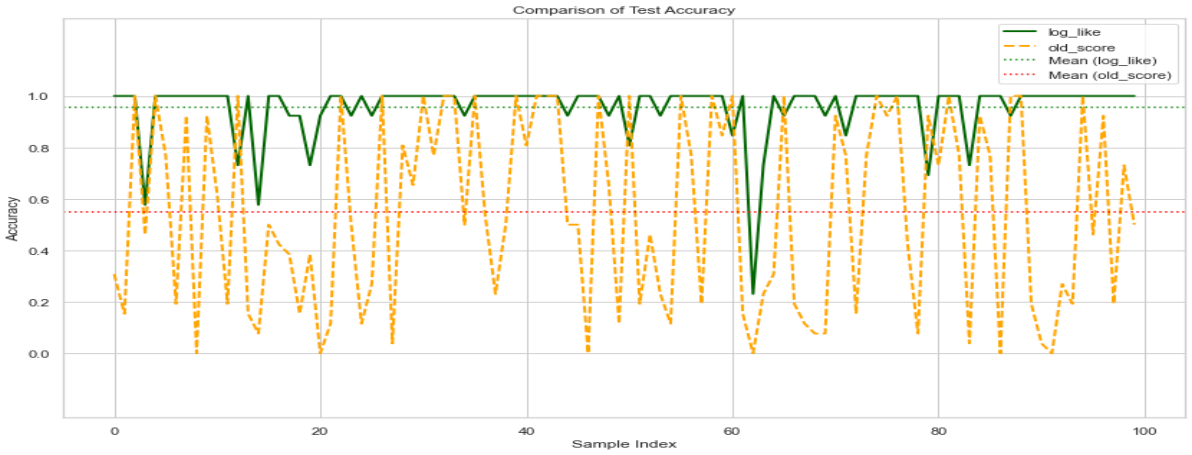

Fig. 2: Accuracy comparison of the two scoring techniques over 100 tests

C. Selection Criteria

After this clarification on the algorithm's performance, it's time to move over and have a look at our selection criteria. To do so, we have decided to firstly outline the general logic used:

1) Checks if the `proposed_score` is greater than the `current_score` and, if so, return `True`.
2) Otherwise, if the `proposed_score` is lower than the `current_score` there's still the possibility that the value gets accepted. This happens because the function will compute the exponential in the first line of the "else" statement. Then, it will generate a number between 0 and 1, and, if the number is lower than `accept_probability` the `proposed_score` gets selected.

If all of this sounds familiar to you, it's probably because it is the actual coding implementation of section V-A or, more simply, the implementation of the Metropolis-Hastings algorithm.

Listing 3: Function that manage the accepting of proposed ciphers

```
def should_accept(current_score,
    proposed_score, temperature):
    if proposed_score > current_score:
        return True
    else:
        accept_probability =
            math.exp((proposed_score -
            current_score) / temperature)
     return random.random() <
         accept_probability
```

To conclude this practical overview of the functions we implemented so far, the last one we'll analyse is the real centre of our project.

D. Application of the Markov Chain Monte Carlo Algorithm

Indeed, the `mcmc_decrypt` function (See Listing 6) will first of all conduce the n-gram analysis on the reference model by computing the n-gram probabilities and saving the result in a dictionary. Then, it will generate our original cipher, using the single letter frequency of both the reference corpus and the ciphertext we are attempting to decrypt. Finally, the algorithm enters a `for` loop where:

1) we generate a new proposed cipher, and we get its score (Listing 2);
2) we compare the new score with the current score;
3) using the function `should_accept` (Listing 3) we decide which cipher will be set as current;
4) we update our temperature metrics;
5) we repeat the loop for n iteration.

Now that we have clarified both the theoretical and practical aspects of our project, it's time to focus on the results.

VII. Testing

In the following section, we will dive into the results we achieved by testing our algorithm.

A. Testing Methodologies

To test our algorithm, we decided to consider how accurate it scored on different kind of text length and authors. To compute accuracy, we used the same approach seen in Section VI-B).

During the full testing process, we run 2000 iteration for each test, and we used bigrams to perform the n-gram analysis.

B. Testing on increasing amount of text

We took a sample text, an extract of Resurrection by Leo Tolstoy, with total characters number being equal to 483,000. Then, we split the sample text in various sizing, progressively incrementing the number of characters by 50,000. As a result, we had 9 text sample all differing by 50,000 characters, plus the full-length test text. We run 100 tests per each text sample and compared their average accuracy score.

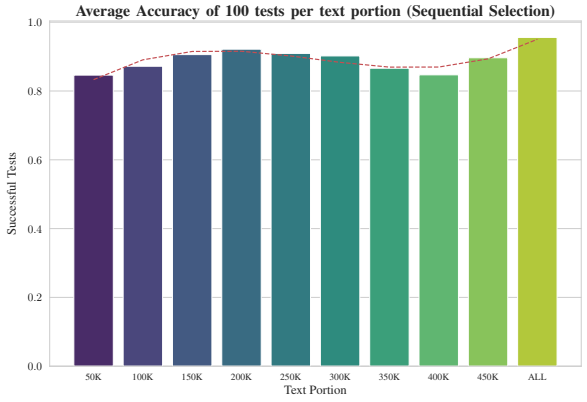

Fig. 3: Testing results of our algorithm on increasing text portion.

From the bar plot in Fig.3 we can clearly see, up to 200,000 characters, performance increase as character in the sample increases. After this threshold, counterintuitively, we see a minimal reduction in accuracy, followed by a greater increase when we reach length of 400,000 or more.

We have investigated further this phenomenon, and we concluded that this unexpected behaviour of our algorithm could signal the presence of bias in the text between 200,000 and 400,000 characters, that for our algorithm is harder to interpret. In addition, we believe we made a mistake in our testing methodology, since we take sequentially more and more text, our tests will be not interdependent one with the other, but they will include all the text present in the text portions tested before them (i.e., a good result at 200K characters will influence the results at 250K and 150K characters).

To investigate this bias further, we decided to make two additional tests:

1) Random Selection of words
2) Changing the statistical distribution of the text.

1) Random Selection Approach: With this test approach, we tried to remove any dependency between test cases. By randomly selecting words from the cipher text for each incremental test. This way, at each increasing step of characters, we are sure that the following test will not contain also the text tested in the previous attempt.

To implement the random selection approach, we kept the same methodology as the previous test, but we had to switch from selecting characters to selecting a number of words. For this reason, the label of the x-axis in Figure 4 do not match the one in Figure 3. However, the number of characters in each test case is basically the same, incrementing of roughly 50,000 each time (For the precise numbers of characters in each test See Table II).

Test_id	Number of Characters
10K	53,155
20K	106,091
30K	158,847
40K	211,297
50K	264,599
60K	317,192
70K	370,739
80K	423,181
ALL	483,080

TABLE I: Characters contained in each test case for the random selection approach (See Figure 4).

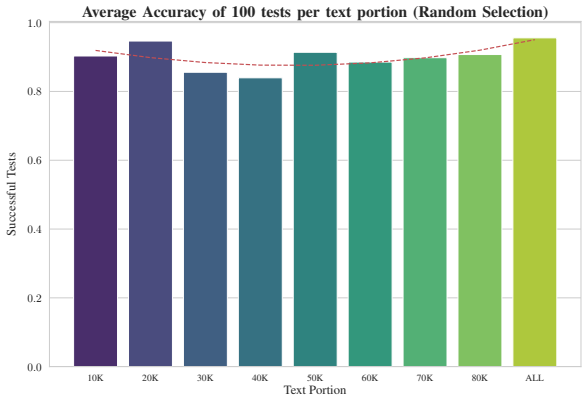

Fig. 4: Testing results of our algorithm on increasing text portion randomly selected.

We believe that removing dependency between test cases, resulted in an improved situation. The average accuracy score seam to follow a stable, slightly increasing trajectory as tests become larger. However, we were not able to totally remove the bias, in fact, we experienced a slight decrease in accuracy between 30 thousand and 40 thousand words.

2) Changing Statistical Distribution: The second approach, we attempted to remove the bias from our testing results is changing the statistical distribution of the text. To achieve a different distribution, we decided to change the way we select bigrams during the n-gram analysis. Instead of selecting bigram by taking one letter after the other, we will take one letter and the second after. In other word, the bigrams subdivision of the word “CIPHER” will change

from “CI”, “IP”, “PH”, “HE”, “ER” to ”CP”, ”IH”, “PE”, “HR”.

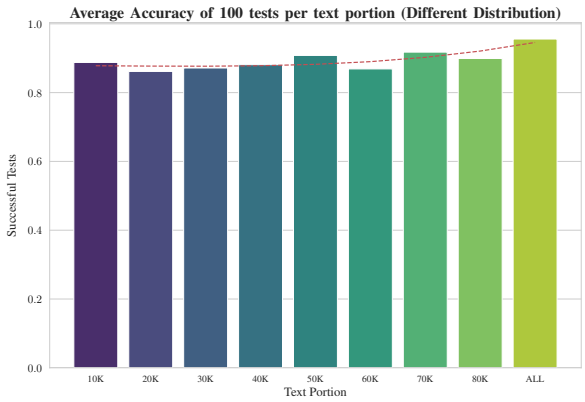

Fig. 5: Testing results of our algorithm on increasing text portion with a different statistical distribution for the text.

To perform this test, we kept the same methodologies used in the previous test (Section VII-B1). We just modified the function `compute_ngram_probabilities` in our code (Listing 5) to perform the alternative n-gram selection.

By changing our method of selecting bigram to perform the frequency analysis, we were able to remove the bias we had before. In Figure 5 we can see how we don’t have any unexpected drop in accuracy as the amount of characters increases. In this last test, our algorithm accuracy have an increasing trajectory positively correlated with the numbers of characters in the cipher text it tries to decrypt.

Test_id	Number of Characters
10K	52,626
20K	106,053
30K	159,992
40K	211,491
50K	264,420
60K	316,982
70K	369,888
80K	422,877
ALL	483,080

TABLE II: Characters contained in each test case for alternative statistical distribution approach (See Figure 5)

C. Testing on Different Authors

Up to this point, both our ciphertext and reference corpus were written by the same author. To further test our algorithm performance, we decided to keep the same reference corpus but test the algorithm on ciphertext generated from book written by diverse authors (Table III).

Test_id	Length (Char)	Total Tests	Accuracy (Avg.)
Moby Dick[1]	1,191,786	36	0.958
Resurrection [2]	483,080	101	0.948
Dorian Gray[3]	428,967	30	0.952
Dr. J & Mr. H[4]	138,461	61	0.863

TABLE III: Testing results of our algorithm on various work of literature of various length

In our testing, we concluded that our algorithm is accurate with text from different authors and of various length. In addition, we can say that, even if marginally, our algorithm’s accuracy increases with the length of the text.

VIII. Group and Activity description

Our group is formed by: Simone Di Lorenzo (889833), Luca Barattini (888528). Our work have been the result of a close collaboration on every aspect of the project.

What	Who
Initial Research on possible solutions	888528 & 889833
Research on scoring mechanisms	888528
Research on MCMC algorithm and Metropolis-Hastings	889833
Initial Implementation of the cipher.py et loadfile.py module	889833
Initial Implementation of the score.py module	888528
Deep-dive research on scoring algorithms due to initial inefficiency	888528
Implementation of the new scoring function	888528 & 889833
Implementation of the mcmc.py module	888528 & 889833

TABLE IV: Work division between the member of the group.

IX. Conclusion

Cryptography is a fascinating world where the ancient art of secret communication meets the cutting-edge of modern computational techniques, and this project has only scratched the surface of its vast and intricate landscape. Our hope is that we have successfully highlighted the nuanced capability of statistical models to adapt to complex linguistic patterns, offering a substantial leap forward from traditional frequency-based methods.

[1]Moby Dick by Herman Melvile
[2]Extract of Resurrection by Leo Tolstoy
[3]The Picture of Dorian Gray by Oscar Wilde
[4]The Strage Case of Dr. Jekyll and Mr. Hyde by Robert Louis Stevenson

APPENDIX A
CODE IMPLEMENTATIONS

Listing 4: Code for computing the n-gram probabilities of a given text

```
def compute_ngram_probabilities(text, n):
    ngrams = defaultdict(int)
    text = re.sub(r'[^a-zA-Z]', '', text).lower()
    for i in range(len(text) - n + 1):
        ngram = text[i:i+n]
        ngrams[ngram] += 1
    total_ngrams = sum(ngrams.values())
    return {ngram: count / total_ngrams for ngram, count in ngrams.items()}
```

Listing 5: Modified code for computing the n-gram probabilities of a given text skipping one letter when selecting n-gram

```
def compute_ngram_probabilities(text, n, skip=1):
    ngrams = defaultdict(int)
    text = re.sub(r'[^a-zA-Z]', '', text).lower()
    for i in range(len(text) - n + 1, skip):
        ngram = text[i:i+n]
        ngrams[ngram] += 1
    total_ngrams = sum(ngrams.values())
    return {ngram: count / total_ngrams for ngram, count in ngrams.items()}
```

Listing 6: Code for implementing the MCMC algorithm

```
def mcmc_decrypt(num_iterations, cipher_text, corpus_text, n):
    # Build the bigram model from the corpus
    corpus = lf._readfile(corpus_text)
    n_gram_freq = score.compute_ngram_probabilities(corpus, n)

    # Generate origitnal cipher
    current_cipher = cipher.create_original_cipher(corpus_text, cipher_text)
    current_decrypted_text = cipher.apply_cipher(cipher_text, current_cipher)
    current_score = score.log_likelihood(current_decrypted_text, n_gram_freq, n)

    # Temperature parameter settings
    initial_temperature = 1.0
    final_temperature = 0.01
    temperature = initial_temperature
    cooling_rate = 0.95

    for _ in tqdm(range(num_iterations)):

        proposed_cipher = cipher.generate_cipher(current_cipher)
        proposed_decrypted_text = cipher.apply_cipher(cipher_text, proposed_cipher)
        proposed_score = score.log_likelihood(proposed_decrypted_text, n_gram_freq,
            n)

        if should_accept(current_score, proposed_score, temperature):
            current_cipher = proposed_cipher
            current_score = proposed_score


        temperature = max(final_temperature, temperature * cooling_rate)
```

```
current_decryption = cipher.apply_cipher(cipher_text, current_cipher)

# Saving results on a txt file
f = open("final_decryption.txt", "w")
f.write(current_decryption)
f.close()

f = open("final_cipher.txt", "w")
f.write(current_cipher)
f.close()
```

REFERENCES

Agarwal, Rahul (2019). *Applications of MCMC for Cryptography and Optimization*. URL: https://towardsdatascience.com/applications-of-mcmc-for-cryptography-and-optimization-1f99222b7132.

Agrahari, Shivam (2021). *Monte Carlo Markov Chain (MCMC) Explained*. URL: https://towardsdatascience.com/monte-carlo-markov-chain-mcmc-explained-94e3a6c8de11.

Everton Gomede, PhD (2023). *Exploring N-gram Models in Natural Language Processing*. URL: https://medium.com/@evertongomede/exploring-n-gram-models-in-natural-language-processing-bf5852b32050.

Keng, Brian (2021). *Hamiltonian Monte Carlo (HMC) Algorithm*. URL: https://bjlkeng.io/posts/hamiltonian-monte-carlo/#hmc-algorithm.

Maccari, Leonardo (2023). In: *Lab of Computer Security, Lecture Notes*. Chap. 7.3.1, pp. 109–112.

Rohde, Maximilian (2022). *Code Breaking with Metropolis*. URL: https://maximilianrohde.com/posts/code-breaking-with-metropolis/.

V, Nithyashree (2023). *What are N-grams and How to Implement Them in Python*. URL: https://www.analyticsvidhya.com/blog/2021/09/what-are-n-grams-and-how-to-implement-them-in-python/.

A Technical content

B Deepness and soundness

C Presentation