Cryptanalysis of Vigenère Cipher: Implementation and Explanation

Name : Himanshu Singh
Entry Number : 2025JCS2062

August 30, 2025

Abstract

This report describes a C++ implementation for breaking Vigenère ciphers using pattern analysis and frequency statistics. All algorithmic steps and their rationale are discussed, including preprocessing, key length detection, frequency-based decryption, and output formatting.

1 Introduction

The Vigenère cipher is a classical polyalphabetic cipher, historically thought unbreakable. Its security hinges on a secret key and use of Caesar-cipher-like shifting. This project implements a statistical attack—automatically recovering both the key and the original plaintext from ciphertext only.

2 Program Flow and Approach

The program for cryptanalysis of the Vigenère cipher is structured around several key functions, forming a pipeline that transforms raw ciphertext into decrypted plaintext and recovers the cipher key. Below is a detailed description of each major component and their roles in the program:

2.1 Preprocessing

When the ciphertext is received, it is not immediately suitable for cryptanalysis. The `preprocess` function reads the ciphertext from a file and creates two versions:

- *Original ciphertext*, which preserves formatting such as spaces and special characters.
- *Cleaned ciphertext*, which removes all non-alphabetic characters and converts all letters to uppercase.

This cleaning ensures that subsequent analyses, which operate on letters only, can be carried out effectively without interference from formatting.

2.2 Kasiski Examination

Next, the program estimates the key length using the `kasaski_method`. This function searches for all repeated substrings of length three or more within the cleaned ciphertext and calculates the differences in their positions.

Because the Vigenère cipher is a polyalphabetic cipher that repeats keys periodically, these repeated occurrences often appear at intervals that are multiples of the key length. By computing the greatest common divisor (GCD) of these distances, the function proposes a probable key length.

However, the method can be sensitive to noise—if even a single anomalous substring occurs, the GCD may incorrectly yield 1. To mitigate this, a fallback *voting mechanism* selects the most common factor dividing the distances, which is then chosen as the key length.

2.3 Index of Coincidence

To confirm the probable key length, the `index_of_coincidence` function splits the ciphertext into multiple subsequences, one for each key position modulo the key length.

For each subsequence, it calculates the Index of Coincidence (IC), a statistical measure that reflects how close the letter distribution is to normal English text. The average IC across all subsequences is then compared to typical English values (0.06 to 0.075).

If the mean IC is within this expected range, the key length is accepted; otherwise, other candidate lengths from the fallback list are tested.

2.4 Key Recovery: Mutual Index of Coincidence and Chi-Square Test

The heart of the cryptanalysis lies in the combined use of Mutual Index of Coincidence (MIC) and chi-square statistical testing:

- The `mutual_index_of_coincidence` function divides the ciphertext into subsequences by key length and uses the first subsequence as a reference. It calculates the MIC between the reference and each other subsequence to estimate the relative shifts between letters in the key.
- The `decipher` function then attempts all 26 possible shifts of the key and decrypts the ciphertext for each candidate.

To select the most plausible plaintext, the program runs a `chi_square_test` which compares the letter frequency distribution of each candidate plaintext with the expected English alphabet frequencies. The plaintext with the lowest chi-square score is selected as the correct decryption, and the corresponding key shifts form the recovered key.

2.5 Postprocessing and Output

After decryption, the `postprocess` function restores the original formatting. It re-inserts spaces and special characters, and converts letters back to the correct cases based on the original ciphertext.

Finally, the decrypted plaintext and the recovered key are output in a JSON file, facilitating further processing or evaluation.

3 Function Descriptions

preprocess(): Reads ciphertext from file, generates cleaned uppercase text and retains original with formatting.

best_factor(): Implements fallback voting on common factors dividing substring spacing differences.

gcd()/gcd_all(): Compute greatest common divisors of distances to estimate key length.

kasaski_method(): Finds repeated substrings and their positional differences, applies GCD and fallback factor voting to guess key length.

compute_ic(): Calculates the Index of Coincidence for a given text segment to indicate language likelihood.

index_of_coincidence(): Splits ciphertext by key length and averages IC values to confirm key length.

compute_mic(): Compares frequency alignments of subsequences to identify relative key shifts.

chi_square_test(): Scores how well candidate plaintext frequencies match expected English frequencies.

decipher(): Tests possible shifts across key positions to find the best matching plaintext and key.

mutual_index_of_coincidence(): Uses MIC and chi-square tests to produce final key string.

postprocess(): Restores sentence formatting, spacing, and case to decrypted plaintext.

file_output(): Writes the recovered key and plaintext in JSON format.

4 Discussion and Result Insights

The approach is effective when ciphertext is sufficiently long and resembles English language statistics. For accurate results, the encryption key should not be excessively long relative to the message size. If the ciphertext is short, highly noisy, or written in a non-English language, automated checks may fail to confirm key length and revert to fallback voting or output an error JSON file.

The implemented system runs end-to-end, requiring only a valid `input.txt` file to produce the decrypted plaintext and key in `output.json`. This design makes the tool practical and easy to use for classical cryptanalysis tasks.

5 Summary

This program demonstrates an automated cryptanalysis tool for the Vigenère cipher, incorporating classical attacks like Kasiski examination and statistical frequency tests. Its modular design, robust error handling, and output formatting create a complete pipeline that can recover keys and plaintexts from ciphertexts efficiently.