POLYTECHNIC UNIVERSITY PUERTO RICO

Cracking the Monoalphabetic Substitution Cipher

Author: Carlos Santana

Advisor: Prof. Jeffrey Duffany

Polytechnic University of Puerto Rico

Abstract

Cryptography is the cornerstone of secure communication. One of the earliest cryptographic techniques used by the Romans is the monoalphabetic substitution cipher, which replaces one letter in a message with another. This approach has one glaring weakness, in that the frequency of letters is preserved in the ciphertext. Therefore, messages encrypted via monoalphabetic substitution are vulnerable to frequency analysis attacks. A persistent attacker that has intercepted the message can conduct frequency analysis to "crack" a monoalphabetic substitution cipher. An algorithm can also be created to automate the process and conduct frequency analysis attacks on ciphertext to crack the cipher quickly, decrypting the message and exposing the plaintext and its contents to an unintended recipient.

Introduction

Cryptography is the process by which a code, or cipher, is applied to some text (the plaintext) to transform it into an obfuscated form (the ciphertext).

The two basic categories of ciphers are transposition and substitution. The former changes the order of letters without changing the letters, while the latter changes the letters without changing the order[1].

The monoalphabetic substitution cipher pairs each letter in an alphabet with another letter in the same alphabet and rewrites the message by replacing each letter with its pair[2].

Problem and Initial Approach

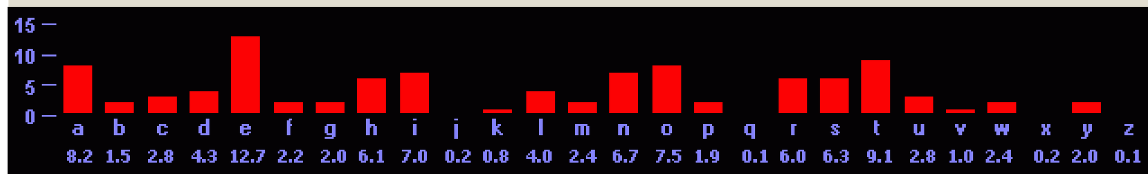

Figure 1: Letter frequencies in English

Because each occurrence of a given letter is replaced with the same letter every time, the number of times that letter appears can still be detected. The study of how frequently a letter appears within a ciphertext is called frequency analysis. This study forms the basis for decrypting the monoalphabetic substitution cipher.

As demonstrated by Figure 1, the letter E is the most common letter in the English alphabet, generally composing up to 12.7% of the letters in a document. E is followed by T, A, O, I, and N. Cryptographers recognize that these tendencies will likely hold with most pieces of plaintext. And thanks to the monoalphabetic cipher's fatal flaw, these letter frequencies are preserved when the plaintext is converted into ciphertext. Therefore, a persistent cryptographer can use frequency analysis to try and guess which letter in the ciphertext maps to which letter in the plaintext [3].

Letter frequency is the basis for this study, but other tendencies of the English language are also used for this research:

- The only one-letter words in English are "a" and "I".
- A list of common two- and three-letter words.
- A list of common double-letter pairs in English
- The most common three-letter word is "the".
- The most common letter at the end of a word is 's', followed by 'd', in plural words and past-tense words, respectively.
- The most common three-letter word ending is "–ing".
- 'W' is a rare letter, though the question words ("where", "what", "why", etc.) are common.

Methodology

```
ct = readfromfile("duffcaesar.txt")
(ctl, SPACE_INDICES, ctw, CIPHER_WORDS_WITH_DOUBLES,
WORDS_BY_SIZE) = textstats(ct)

CIPHER_FREQ = {}
(CIPHER_FREQ, ctl_lettersonly) = countletters(ct)
CIPHER_FREQ_PCT = countpcts(CIPHER_FREQ, ctl_lettersonly)
CIPHER_FREQ_SORTED = sorted(CIPHER_FREQ_PCT.items(),
key=lambda kv:(kv[1],kv[0]), reverse=True)
CIPHER_FREQ_LETTERSONLY = []
for i in CIPHER_FREQ_SORTED:
    CIPHER_FREQ_LETTERSONLY.append(i[0])

MAPPING = create_mapping(CIPHER_FREQ_SORTED, FREQ_SORTED)
MAPPING = check_mapping(MAPPING)

pt = translate(ct, MAPPING)
print("The ciphertext:\n{0}\nmay decode to the
following:\n{1}".format(ct,pt))
```

Figure 2: Program Main Body

```
def readfromfile(filename: str) -> str:
    with open(filename, 'r') as f:
        text = f.read().lower()
    print("Ciphertext read from: {0}".format(filename))
    return text

def textstats(text:str) -> tuple:
    textlen = len(text)
    spaces = []
    for i in range(textlen):
        if text[i] == ' ':
            spaces.append(i)
    words = text.split(' ')
    doubles = []
    for word in words:
        if has_double(word):
            doubles.append(word)
    sizes = []
    big = len(max(words,key=len))
    for i in range(1, big + 1):
        lst = []
        for word in words:
            if word[-1] not in FREQ_LETTERSONLY:
                word = word[:-1]
            if len(word) == i and word not in lst:
                lst.append(word)
        sizes.append(lst)
    return (textlen, spaces, words, doubles, sizes)
```

Figure 3: Phase 1 of Program (receive ciphertext and collect stats)

A program was created that would run in four phases. In Figure 2, the main body of the program is displayed to demonstrate execution flow. The following steps are followed:

1. Ciphertext is received from the user, and statistics of the ciphertext are derived
2. Frequency analysis is conducted on the ciphertext
3. A mapping between the ciphertext and plaintext is created and then fixed using tendency tests
4. The ciphertext is translated into the plaintext and displayed.

Figures 3-6 contain the helper functions used in each phase of the program. A few functions were omitted due to lack of space.

Phase 1 uses the following methods:

1. Readfromfile() reads from a file, converts the text into lowercase, and reports the success thereof to the user.
2. Textstats() returns information about the ciphertext, including length, indices of spaces, a list of all the words in the ciphertext, words that contain a double-letter pair, and the words sorted by lengths.
3. Has_double() returns whether a word has a double-letter pair. This function is not listed.

Methodology pt. 2

```
def countletters(text:str) -> tuple:
    counts = {}
    lettersonly = 0
    for char in text:
        if char not in FREQ_LETTERSONLY:
            continue
        counts[char] = counts.get(char, 0) + 1
        lettersonly += 1
    return (counts, lettersonly)

def countpcts(dicto:dict, nums:int) -> dict:
    counts_pct = {}
    for item in dicto.keys():
        counts_pct[item] = dicto[item]/nums
    return counts_pct
```

Figure 4: Phase 2 of program (frequency analysis)

```
def create_mapping(cipher_dict: dict, plain_dict: dict) ->
dict:
    mapping = {}
    cipherlen = len(cipher_dict)
    for i in range(cipherlen):
        mapping[cipher_dict[i][0]] = plain_dict[i][0]
    mapping.update(CONSTANT_SYMS)
    return mapping

def check_mapping(potential_map:dict) -> dict:
    map_with_fixed_the = check_the(potential_map)
    map_with_fixed_two = check_twoletters(map_with_fixed_the)
    map_with_fixed_plurals = check_plurals(map_with_fixed_two)
    map_with_fixed_w = check_w(map_with_fixed_plurals)
    map_with_fixed_end = check_endings(map_with_fixed_w)
    map_with_fixed_three =
check_threeletters(map_with_fixed_end)
    map_with_fixed_ing = check_ing(map_with_fixed_three)
    map_with_fixed_doubles = check_doubles(map_with_fixed_ing)
    newmap = {}
    newmap.update(map_with_fixed_doubles)
    newmap.update(CONSTANT_SYMS)
    return newmap
```

Figure 5: Phase 3 of program (mapping ciphertext to plaintext)

```
def translate(cipher:str, map:dict) -> str:
    plain = ""
    for letter in cipher:
        plain += map[letter]
    return plain
```

Figure 6: Phase 4 of program (translation and printing of results

Phase 2 uses the following methods:

1. Countletters() iterates over the ciphertext and counts how many times a letter appears. Symbols are not counted.
2. Countpcts() divides each value by the number of letters in the ciphertext to convert them to percentages.

Phase 3 uses the following methods:

1. Create_mapping() takes each letter of the ciphertext and maps it to a letter in the plaintext. The list of letters is sorted in order from most frequent to least frequent
2. Check_mapping() runs a series of tests, called the tendency tests, on the mapping to confirm that it is correct.

The eight tendency tests conducted in phase 3 are check_the(), check_twoletters(), check_plurals(), check_w(), check_endings(), check_threeletters(), check_ing(), and check_doubles(). Their code is very extensive, so they are omitted in this presentation.

Phase 4 uses the following method:

1. Translate() iterates over a text and rewrites the text by replacing each letter in the original with its corresponding letter in the mapping. The results are printed out to the user.

Results and Conclusions

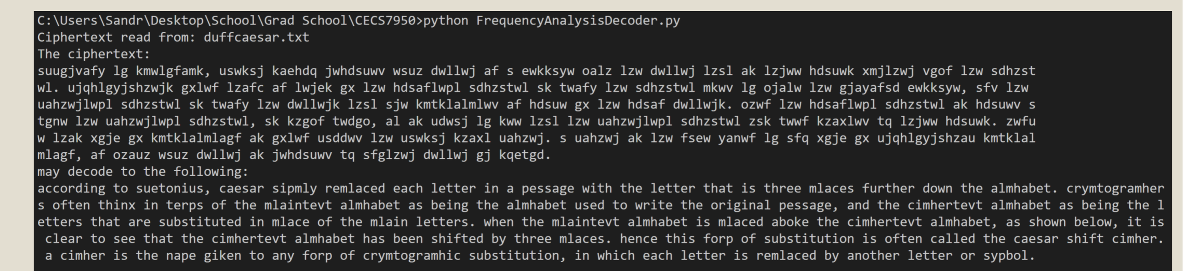

Figure 7: Program Results

The program runs in about 1 second. It performs eight partial iterations, one for each tendency test, and one full iteration for the translation. A few bugs have yet to be resolved, such as the switching between P and M.

Future Work

For future iterations of this program, the researcher would probably begin by adding a final test to the check_mapping() method. It would go through all the remaining mappings and brute-force switching them with each other to see which one sticks. This process is normally time-consuming, but by expanding the number of mappings that are proven to be correct prior to running the brute-force check, this time can be shortened greatly. Segments of unnecessary code would be deleted, so that the program would be easier to understand. The tests can also be improved so they rely less on assumptions and more on hard evidence. The researcher could also investigate more letter frequency data to create more tests from. The final bit of work would be to optimize the code to speed up the algorithm further.

Acknowledgements

I conducted this research on my own, with help from various websites across the internet, primarily for help in formatting with Python. The biggest sources of help were the Simon Singh website and the CodeDrome article[4] for providing me with the technique. Additionally, I wish to thank professor Duffany, whose guidance proved invaluable for pointing me in the right direction. Additional thanks to Dr. Denise Cobian editing my article. And finally, I wish to thank my parents who bankrolled the article edition process.

References

[1] M. Singh, "Difference between Substitution Cipher Technique and Transposition Cipher Technique – GeeksforGeeks," *GeeksforGeeks*, October 7, 2021. [Online]. Available: https://www.geeksforgeeks.org/difference-between-substitution-cipher-technique-and-transposition-cipher-technique/. [Accessed 6-May-2022].

[2] *101 Computing*, "Mono-Alphabetic Substitution Cipher.", Nov 9, 2019. [Online]. Available: https://www.101computing.net/mono-alphabetic-substitution-cipher/. [Accessed: 6-May-2022].

[3] S. Singh, "The Black Chamber - Letter Frequencies," *Simon Singh*. [Online]. Available: https://www.simonsingh.net/The_Black_Chamber/letterfrequencies.html. [Accessed May-06-2022].

[4] C. Webb, "Frequency analysis in Python," *CodeDrome*, 12-Jul-2018. [Online]. Available: https://www.codedrome.com/frequency-analysis-in-python/. [Accessed: 17-May-2022].