Cryptanalysis of a Class of Ciphers Based On Key Analysis and Decryption

Nowsha Islam, Harish Saravanakumar, and Josh Yam

March 17, 2019

Introduction

Our team consisted of three members; Nowsha Islam, Harish Saravanakumar, and Josh Yam. The entirety of our project is a result of equal collaboration from all three members. We discussed and decided upon all parts of our approach together. Once the steps behind our approach was decided upon, individual parts were coded separately.

1. Frequency analysis algorithm was coded by Nowsha.
2. Code to encrypt the plaintext and get the corresponding ciphertext was done by Josh.
3. Code to find the length of the key was done by Josh.
4. Using the key length, code to find the key itself was done by Nowsha and Harish.
5. Code to decrypt the ciphertext using the potential/estimated key was done by Harish
6. Code to compare the frequencies for the potential plaintext with each of the original plaintexts to determine a match was done by Harish
7. Extra credit discussing substitution/substitution ciphers was done by Josh

Detailed Informal Explanation

The first step in trying to decrypt the ciphertext is to find the key length or the period of the cipher. Because the key is repeated when encrypting the ciphertext, that means that the first letter of every key interval has the same shift. For example, if the key length is three, that means that the first, fourth, seventh, and so on letters in the ciphertext have been shifted by the same amount. We can therefore use the index of coincidence, to find the key length. If the total length of a message is n, then the probability to choose any letter would be $\frac{1}{n}$. However, for English and any other text that resembles a language instead of gibberish, the *Index of Coincidence* tends to be higher. Therefore, in our approach where we test all possible key lengths, the sequences that have the highest *IC*s should be ones that come from the key length and its multiples.

Once we retrieved the key length, our next objective was to find the key used to encrypt the plaintext. Since m[i] is shifted k[j[i]] positions, with an undisclosed algorithm dictating j, we decided to choose a method that would allow us to decipher the new unique key used every round. The method we decided on was the **shift and multiply** method. This method requires two main sets of data: the frequencies of characters (in the character space of [<space>, a , , z] in the alphabet and the ciphertext.

1. Find the subset: (pos + k*i)th letters of the ciphertext, where i is (0(ciphertext length)/key length) and **pos** = (0,1,2,..., key length)
2. Find the frequencies of the characters in this subset and order them (<space>, a, , z)
3. Multiply the subsets frequencies with the true frequencies for each character in the subset in order [meaning (subset freq of a multiplied by alphabet freq of a) and so on], then sum those results together

4. Shift the subset frequencies by one, aligning them with a different set of true frequencies (so alphabets a aligns with subsets b, and so on), then continue with step 3
5. Once the subset has been shifted every possible way [(ciphertext length/key length) times], find the largest sum recorded, and the ith shift number it occurred on
6. This shift number represents the kth key value (key[k], where k = [(0,1,..., key.length) so the first iteration of this algorithm returns key[0])]
7. Repeat Steps 1-6 as many times as the key length. Steps 1-6 represent one round of the algorithm and one portion of the key. To continue to the next algorithm round, start step 1 by shifting the subset collection by 1 character (pos += 1)

The intuition behind aligning different frequencies and attempting to find the largest sum derives from a simple math property. For example, imagine we wanted to find the largest possible number by multiplying two sets of numbers together: $A = 1, 2, 3$ and $B = 1, 2, 3$. There are only six possible ways to multiply the numbers:

$$\mathbf{(1 \times 1) + (2 \times 2) + (3 \times 3) = 14}$$
$$(1 \times 1) + (2 \times 3) + (3 \times 2) = 13$$
$$(1 \times 2) + (2 \times 1) + (3 \times 3) = 13$$
$$(1 \times 2) + (2 \times 3) + (3 \times 1) = 11$$
$$(1 \times 3) + (2 \times 2) + (3 \times 1) = 10$$
$$(1 \times 3) + (2 \times 1) + (3 \times 2) = 11$$

We can clearly see that the largest sum occurs when multiplying the largest value in set A with the largest in B, the next largest value in set A with the next largest in B, and so on. This same intuition can be applied to the encrypted ciphertext. Since the key encrypts the plaintext, this means that the true frequency and the ciphertext frequencies are shifted. So in order to realign them, we can use the **shift and multiply** principle to find the largest sum. The number of shifts it took to get to this largest sum represents the first integer in the key. To reiterate, this first integer is what is used to encrypt the character, the character at index key length + 1, and all succeeding characters following this pattern. The values this method finds will not return the original key, but rather a permutation of it. This key can then be used to directly decrypt the ciphertext (rather than using the scheduling algorithm to find which key values are being used).

An example of our method:
The frequencies recorded for a sample ciphertext are as follows:
If the key = 7 every 7th letter in a sample of the ciphertext selected:
gkjvfbu**x**vftqws**d**icxruo**i**hyozzx**w**uotguv**t**jlieor**u**euxisp**f**pixlpu**s**rvrbxm**a**uezkjb**v**kwavvf**s**bojjej**h**ej...
The frequencies of the selected characters are as follows:
20 29 20 26 9 29 22 33 22 15 22 33 21

We then convert these frequencies into fractions and put them in order.

The following table shows the shift and multiply method for just the first three letters of the alphabet. In reality this is done for the entire message space:
True frequencies for the characters a, b, c are $0.0653, 0.0125, 0.0223$

$a = 0.0653$	$b = 0.0125$	$c = 0.0223$	Sum	Shift
0.8037	0.9413	0.4457	$(0.8037 \times 0.0653) + (0.9413 \times 0.0125) + (0.4457 \times 0.0223) = 0.0102$	0
0.9413	0.4457	0.8037	$(0.9413 \times 0.0653) + (0.4457 \times 0.0125) + ()0.8037 \times 0.0223) = 0.01806$	1
0.4457	0.8037	0.9413	$(0.4457 \times 0.0653) + (0.8037 \times 0.0125) + (0.9413 \times 0.0223) = \mathbf{0.0212}$	2

The max sum occurs when the shift is 2 so that is the first key value.

Our final step was to decrypt the ciphertext using the key we constructed. In order to do this, we shifted each ciphertext character (c[i]) by the keys value (k[i]). (Ex. a - 3 = y). If the key characters ran out,

we would loop back to the start of the key and continue until the ciphertext is completely decrypted. From here, we had to match the constructed text with the original plaintext. To do this, we analyzed how similar the two texts were in terms of character frequency, and in turn returned a plaintext guess. Since scheduling algorithms are deterministic, this method can be used regardless of the algorithm used.

Detailed Rigorous Description

The scheme for encrypting the plaintext was a challenge because the key was not a fixed length. The variable t, which represented the key length in the project description, could have a value from one to twenty-four. Therefore, one of the first steps to decrypt the ciphertext was finding the key length. To try to find the period of the encryption scheme, we used the Index of Coincidence to analyze the ciphertext. The Index of Coincidence represents the probability of two randomly selected letters from two texts being the same. The formula for the I.C. over a given letter-frequency distribution is:

$$IC = \frac{\sum_{i=1}^{c} n_i(n_i - 1)}{N(N-1)/c}$$

N is the length of the text, and n_i represents the frequency of each of c letters in the alphabet. Therefore, the numerator of the equation is the sum of all of the frequencies of each alphabet letter in the space $< space >, a, .., z$ in the given ciphertext. The index of coincidence is helpful in finding the key length, as when the text is split into the right number of columns, the index of coincidence will be greatest in the columns that are a multiple of the key length. In the function *getKeyLength( )*, there are three vectors that are used: seq which is a 1-D vector of characters, *aKey* which is 2-D vector of characters, and *manyKeys* which is a 3-D vector of characters. There is a triple nested for loop that splits the ciphertext up in accordance to the 24 different possible key lengths. Then for each key length,j, a corresponding j number of sequences are produced. These are calculated by taking the every j^{th}letter of the ciphertext starting at c[i] to c[j], where c represents the ciphertext. Therefore, seq stores the characters for each sequence, *aKey* represents all the sequences for a given key length, and *manyKeys* stores all sequences for all key lengths. The function *getIC( )* is called on to calculate the IC for each sequence. The formula used to calculate IC is shown above. Following calculating the IC for each sequence, the results are averaged and saved onto a vector of floating point integers, *avgIC*, representing the average IC for all sequences for all possible key lengths. The function *getKeyLength( )* returns the key length in *avgIC* that has the greatest IC.

Get Key Length Pseudocode

```
1−D character Vector seq;
2−D character Vector aKey;
3−D character Vector manyKeys;
float vector individualIC;
float vector avgIC;
int counter = largest possible key length;

for(int i =0; i<=counter; i++){
    for(int j=0; j<i; j++){
        for (int k=0; k<cipherText.length(); k+=i){
            seq.push(cipherText[k+j]);
        }
        akey.push(seq);
        individualIC.push(getIC(seq));
        seq.clear();
    }
    manyKeys.push(aKey);
    float avg = average IC for sequences for a key;
    avgIC.push(avg);
    aKey.clear();
}
```

Get Key Value Pseudocode

```
for each keyValue {
    shiftPosition = 0
    for each char in the CipherText {
        store every t characters starting from
        shiftPosition where t is the length of the
        key
    }

freq = store frequencies of each character for this subset
in a vector freq

for each char in the message space {
    for each char in the message space{
        multiply = multiply the frequenicies of each
        character in the subset with the true
        frequencies of each character in the
        alphabet and store the values in a vector
        multiply
    }

    for each number in multiply {
        sum += each number to get the total for
        this shift
    }

    rotatedSums = store each sum in a vector rotatedSums

    Rotate the sums vector by one space such that the
    second character is now the first, the third character
    is the second, ..., and the first character is now
    the last
    Ex: (1,2,3) -> (2,3,1)
}

for each number in rotatedSums {
    Find the max and its position in the vector
}

The keyValue is the position of the max sum.
Increase the shifter by one so the process repeats starting
from the next character in the cipher
}
```

The two functions below outline the ciphertext decryption method and the plaintext determination algorithm.

Decrypt Text Pseudocode

```
string alphabet (<space>, A, ..., Z);
string plainText = "";
decryptCipher (cipherText, key, keyLength){
    for (cipherText length){
        if (keyPosition > keyLength){ // Used to loop the key back to the start
        keyPosition = 0;
        }
        plainNum = alphabetSpace[cipherCharToNum − key[keyPosition]] //
The cipher character is changed to a number, then subtracted from the
corresponding key
        if(plainNum is negative){
            plainNum equals 27 + plainNum;
        }
        find the corresponding character in the alphabet space, and add this
character to plainText
    }
    return the plainText
}
letterMatches(cipherText, plainText){
    for (cipherText length){
        if (cipherText[i] matches plainText[i]){
            increment number of matches by 1;
        }
    }
    return the number of matches;
}

plainTextMatch(cipherText){
    for all plaintexts, run letterMatches(cipherText, plainText (1....5));
        add text(1...5) matches into a vector;
    for (allMatchesVector.size){
        find index with the greatest number of matches;
    }
    return the index with the greatest number of matches;
}
```

Conclusion

Although our code correctly implements our method, it does not correctly determine which plain-text was encrypted to produce the ciphertext. While the key length is consistently correct, the problem most likely lies in the actual key calculation. A potentially more thorough implementation of the key calculation method is required to successfully return the correct results.