A Fast Method for the Cryptanalysis of Substitution Ciphers

Thomas Jakobsen*†

January 8, 1995

Abstract

It is possible to cryptanalyze simple substitution ciphers (both mono- and polyalphabetic) by using a fast algorithm based on a process where an initial key guess is refined through a number of iterations. In each step the plaintext corresponding to the current key is evaluated and the result used as a measure of how close we are in having discovered the correct key.

It turns out that only knowledge of the digram distribution of the ciphertext and the expected digram distribution of the plaintext is necessary to solve the cipher. The algorithm needs to compute the distribution matrix only once and subsequent plaintext evaluation is done by manipulating this matrix only, and not by decrypting the ciphertext and reparsing the resulting plaintext in every iteration.

The paper explains the algorithm and it shows some of the results obtained with an implementation in Pascal. A generalized version of the algorithm can be used for attacking other simple ciphers as well.

Keywords: Cryptanalysis, automated cryptanalysis, substitution cipher, monoalphabetic cipher, polyalphabetic cipher.

1 Introduction

A **mono-alphabetic substitution cipher** is a cipher where a one-to-one mapping is used to substitute each plaintext symbol with a corresponding ciphertext symbol. Often the same set of symbols are used in both plaintext

*Address: Sdr. Roesevej 36 st., DK-2791 Dragoer, Denmark.
†Email: T.Jakobsen@mat.dtu.dk

and ciphertext. In a **polyalphabetic cipher** more than one such mapping is used.

We assume that the plaintext is in English and thus as plaintext symbols as well as ciphertext symbols we use the letters A to Z and space. Thus a monoalphabetic key is a permutation of these 27 ciphertext symbols. The **ciphertext** is obtained from the **plaintext** by replacing each symbol by the corresponding ciphertext symbol in the key. We treat space just like a letter. It is substituted during the encryption/decryption and it is also used when calculating the digram frequencies. In this way we implicitly get a frequency count on words beginning and ending with a particular letter, namely the "space-letter" and the "letter-space" digram frequencies.

2 A sketch of the algorithm

The idea behind the algorithm can be used for ciphertext-only attacks on other simple ciphers as well. The algorithm starts by making an initial guess about what the key is. This guess can be made on basis of a simple analysis of the ciphertext, it can be based on partial knowledge of the key or it can be purely random. The more correct symbols in the assumed key, the more quickly the algorithm will converge to a solution.

The algorithm then uses this guess as a key to decrypt the ciphertext. The resulting text is probably non-readable, but its contents will have a certain similarity to the expected language of the plaintext depending on how many correct symbols there were in the guess in the first place.

In the iterated loop which follows we first alter the current key a little bit, then this key is used to decrypt the ciphertext once again and finally we check if the contents of the new resulting text are closer to the expected language than those of the previously decrypted text. If they are, we keep the new key for the next iteration, if not the old one is used but modified in another way next time the loop is run through etc.

If we can construct a function which reflects "how close" the contents of a given text are to the expected language we will have a working algorithm which successively will find more and more correct symbols. We shall later define such a function.

The algorithm will be explained in details only in the case of a monoalphabetic cipher. It is straightforward to generalize to a polyalphabetic cipher - assuming that the number of alphabets used has already been determined by some standard method, for example the Kasiski test or the index of coincidence.

The initial guess will be based upon the symbol frequencies of the ciphertext. Let us say that the plaintext is in English. Average English has a certain distribution of letters (space, E, T, A, O, N as the most frequent, in that order) and thus our initial guess is that the most common ciphertext symbol is equivalent to space, the second most common is equivalent to E etc.

We now need a function, $f(t)$, which maps a text, t, into a number measuring "how close" the language of t is to the expected language of the plaintext, i.e., a function we can use for describing how close we are to having recovered the original plaintext. But first let us introduce some notation.

Let m denote the plaintext, c the ciphertext, and $e_k(t)$ the encryption function where k is the key used, so that $c = e_k(m)$. Correspondingly we have the decryption function $d_k(t) = e_k^{-1}(t)$.

We define $D(t)$ as the matrix in which the elements are the digram frequencies of a given text t. The row headings are the first symbols of the digram and the column headings are the second symbols.

E denotes a matrix containing the expected digram frequencies of the language in which we assume the plaintext is written. Digram frequencies for English can be obtained from several works, including [4] and [7]. We, however, compiled our own table using 30000 characters of text from Melville's Moby Dick [6]. Obviously it is better to have statistics which are close to those of the plaintext but provided enough ciphertext is available, it appears that the accuracy doesn't greatly affect the performance of the algorithm.

Now we can define the evaluation function, $f(t)$, by the equation

$$f(t) = \sum_{i,j} |D_{ij}(t) - E_{ij}|. \tag{1}$$

In other words the function is simply the sum of the numerical differences of all corresponding elements in the two digram frequency matrices. Intuitively this measure is straightforward and easy to understand, and as we shall see later a variant of this function will in fact under some assumptions yield the maximum likelihood key.

We shall use $f(d(c,k))$ to measure how "good" the key, k, is, that is how close it is to the correct key. Hopefully the more correct the key, the lower this value will be. Fig. 1 shows $f(d(c,k))$ as a function of an increasingly more inaccurate (random) key during monoalphabetic substitution. Here we used 1000 characters of text from Alice in Wonderland [3] as ciphertext.

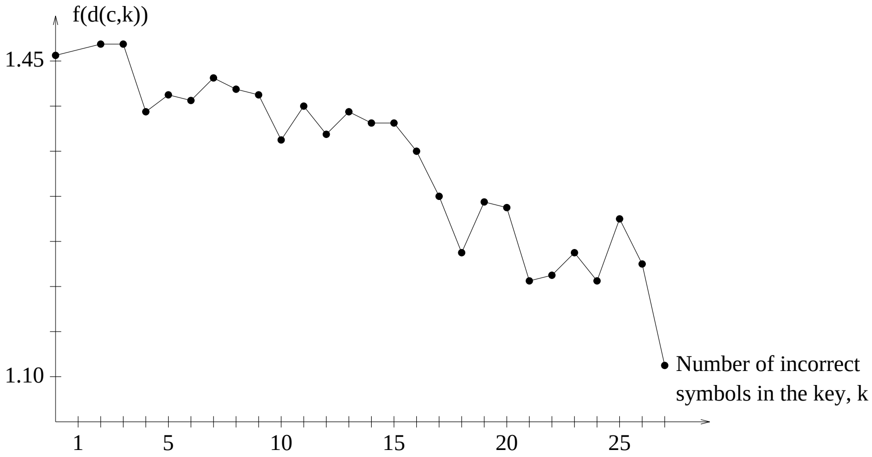

Figure 1: $f(d(c,k))$ as a function of an increasingly more inaccurate key.

Now we can state the main parts of the algorithm more formally.

Algorithm 1:

1. Construct an initial key guess, k, based upon the symbol frequencies of the expected language and the ciphertext.
2. Let $v = f(d(c,k))$.
3 Let $k' = k$.
4. Change k' by swapping two elements, α and β, in k'.
5. Let $v' = f(d(c,k'))$.
6. If $v' < v$ then let $v = v'$ and let $k = k'$.
7. Go to step 3.

Here k is the best key so far and v is the corresponding value of $f(d(c,k))$. The algorithm is terminated when the expression $v' < v$ hasn't been true for a number of iterations, say one hundred.

Remark that the exact meaning of "exchanging two elements" in step 4 is not clear yet, but it will be explained later. For now just assume that we exchange two random key elements. Of course this does not prove to be a very good strategy, though it will work.

Remark also that the calculation of $f(d(c,k'))$ is a very time-consuming operation, because we have to compute $D(d(c,k'))$ each time it is evaluated - this requires the decryption of the whole ciphertext and an analysis to find the corresponding distribution matrix. This problem is solved in the following section.

3 A fast approach

Just after step 5 in Algorithm 1 we have

$$v' = f(d(c,k')) = \sum_{i,j} |D_{ij}(d(c,k')) - E_{ij}| = \sum_{i,j} |D'_{ij}(d(c,k)) - E_{ij}|,$$

where $D'(t)$ is the distribution matrix, $D(t)$, for t with the modification that rows α and β have been swapped after which columns α and β have been swapped (or equivalently swapping the columns first, then the rows).

The above fact can be used to optimize the algorithm so that we have to parse the text only once to find the distribution matrix. We are now able to give a more detailed description of the optimized algorithm.

Algorithm 2:

1. Construct an initial key guess, k, based upon the symbol frequencies of the expected language and the ciphertext.
2. Let $D = D(d(c,k))$.
3. Let $v = \sum_{i,j} |D_{ij} - E_{ij}|$.
4. Let $k' = k$.
5. Let $D' = D$.
6. Swap two elements, α and β, in k'.
7. Exchange the corresponding rows in D'.
   Exchange the corresponding columns in D'.
8. Let $v' = \sum_{i,j} |D'_{ij} - E_{ij}|$.
9. If $v' >= v$ then go to step 4.
10. Let $v = v'$.
11. Let $k = k'$.
12. Let $D = D'$.
13. Go to step 6.

Again k is the best key until now, v the corresponding value of $f(d(c,k))$, and D its digram distribution matrix.

This algorithm is considerably faster because the text is analyzed only once. Apparently the only thing we need to cryptanalyze a substitution cipher is the digram distribution matrix.

The strategy used to choose the two elements, α and β, is described in the following. Let s denote the vector of ciphertext symbols ranked in order of descending frequency. Then s_1 will most likely represent space, s_2 the letter E etc. Now add the following steps to algorithm 2:

Before step 1

0. Let $a = b = 1$.

Instead of step 6 we use

6a. Let $\alpha = s_a$ and $\beta = s_{a+b}$. Swap the symbols α and β in k'.
6b. Let $a = a + 1$.
6c. If $a + b <= 27$ then go to step 7.
6d. Let $a = 1$.
6e. Let $b = b + 1$.
6f. If $b = 27$ then terminate algorithm.

After step 9 the following step is added

9b. Let $a = b = 1$.

In this way the frequencies of the key elements first swapped in k will be close to each other; first we try to exchange s_1/s_2, s_2/s_3, s_3/s_4, ..., s_{26}/s_{27}, then s_1/s_3, s_2/s_4, s_3/s_5, ..., s_{25}/s_{27}, then s_1/s_4, s_2/s_5, s_3/s_6, ..., s_{24}/s_{27} etc., and finally s_1/s_{27} - the pair with the largest difference regarding symbol frequency.

4 Polyalphabetic ciphers

For polyalphabetic ciphers with n alphabets the above algorithm is extended so that instead of one alphabet we have several. The algorithm is roughly the same. The n alphabets are treated cyclically one at a time, so that only one alphabet is "active" at a given moment. We proceed in the same way as described for monoalphabetic ciphers, the difference being that each alphabet is now assigned its own individual set of the variables a, b, k, k', v, and s, and we have to use n distribution matrices, $D^{(1)}$,..., $D^{(n)}$, each keeping track of the digrams beginning at the positions in the text encrypted by the respective alphabet.

Also to update the frequencies properly, instead of step 7, we exchange the rows α and β of the distribution matrix of the current active alphabet, and the columns α and β of the distribution matrix of the previous active alphabet.

The evaluation function becomes

$$f(t) = \sum_{h=1}^{n} \sum_{i,j} |D_{ij}^{(h)}(t) - E_{ij}|,$$

and instead of terminating in step 6f if $b = 27$, we let $b = 1$. We terminate the algorithm when all the symbol pairs of all the alphabets are exhausted without finding any two symbols to swap and at the same time lowering the evaluation value.

5 Maximum likelihood

If we make the assumption that the digram frequencies of a language are independent and that each one follows a Gaussian distribution with mean $\mu_{ij} = E_{ij}$ and variance σ_{ij}^2 for all i, j, and if the algorithm does not get stuck in local minima (that is, if it actually does minimize the evaluation function) then we can find an evaluation function that results in a maximum likelihood key with respect to the digram distribution.

For a maximum likelihood attack our job will be to maximize

$$q = \prod_{i,j} \frac{1}{\sigma_{ij}\sqrt{2\pi}} e^{-\frac{(D_{ij}-\mu_{ij})^2}{2\sigma_{ij}^2}} = r \prod_{i,j} e^{-\frac{(D_{ij}-E_{ij})^2}{2\sigma_{ij}^2}},$$

r being a constant. Here the optimization is done by permuting pairs of corresponding rows and columns in D. Maximizing the above expression is equivalent to minimizing the following

$$-\ln q = -\ln r + \frac{1}{2} \sum_{i,j} \frac{(D_{ij} - E_{ij})^2}{\sigma_{ij}^2},$$

which in turn is the same as minimizing

$$\sum_{i,j} \frac{(D_{ij} - E_{ij})^2}{\sigma_{ij}^2}. \tag{2}$$

When we did in fact not choose the above function as evaluation function it was because we were not in possession of the σ_{ij}-values of English language. However, if the above mentioned assumptions hold, then using (2) should result in a more accurate algorithm (albeit perhaps slower due to the extra multiplications.)

It should be mentioned that the general problem of minimizing (1) or (2) given D and E can easily be shown to be at least as hard as the graph isomorphism problem (consider the incidence matrices of the two graphs) but apparently when the plaintext is human language, cryptanalysis is quite feasible.

As evaluation functions we also tried to use $f(t) = \sum_{i,j} \sqrt{|D_{ij}(t) - E_{ij}|}$ and the obvious candidate $f(t) = \sum_{i,j} (D_{ij}(t) - E_{ij})^2$ but it showed up that generally these perform much worse than (1).

6 Results

Fig. 2 shows how successful the algorithm was in attacking texts of various lengths encrypted using both mono- and polyalphabetic substitution and a random key. Again as reference text (used to create the expected digram frequency matrix) we used 30000 characters of text from Moby Dick [6] and as ciphertext ten excerpts from Alice in Wonderland [3]. The graph depicts the average percentage of ciphertext correctly solved as a function of the text length.

For monoalphabetic substitution the number of iterations executed before the algorithm terminates ranges from about 150 to about 1500, depending on how much ciphertext is available. With five alphabets the number of iterations seldom exceeds 15000.

The algorithm seems to do better than the Carroll-Robbins algorithm [2] both regarding speed and accuracy in spite of its simplicity. In one instance the Carroll-Robbins implementation which was made in C running under Xenix on an IBM-PC/AT, used 20 minutes to decrypt a polyalphabetic cipher with three alphabets. The King-Bahler implementation [5] which uses probabilistic relaxation to cryptanalyze monoalphabetic ciphers had an average run-time of 13 CPU minutes on a HP3000 series 960.

The strength of the presented algorithm regarding speed lies in that we look through the text just once. In fact on an IBM-compatible PC (486, 33 MHz) the time used to cryptanalyze a monoalphabetic cipher ranges from only half a second to no more than a couple of seconds.

Using a 400 character long ciphertext encrypted by monoalphabetic substitution the Carroll-Robbins approach succeeds in correctly recovering 24%

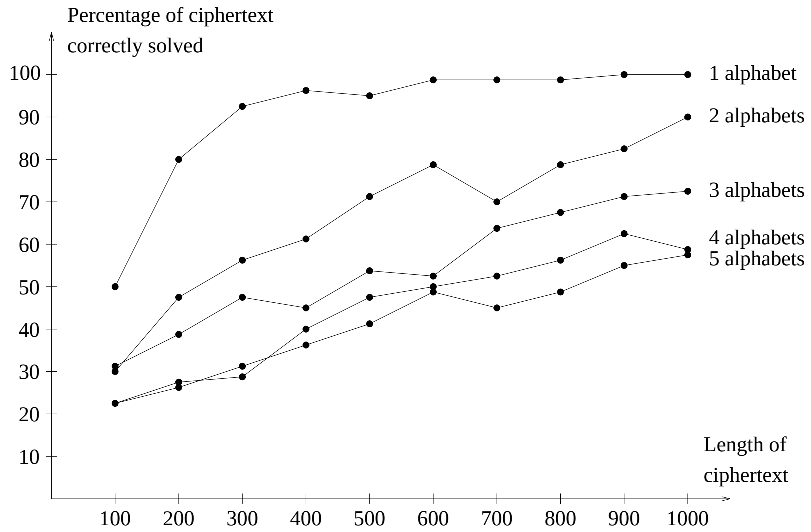

Figure 2: The percentage of text correctly solved as a function of the length.

of the ciphertext symbols whereas 93% of the text was correctly resolved using the King-Bahler algorithm (according to [2] and [5]). Using the method described in this paper resolved 98% of the text.

The software implementation was made using Turbo Pascal from Borland. The source code can be obtained on request from the author. It is also available in C.

7 Conclusion

The algorithm is quite fast compared to earlier results and it succeeds in cryptanalyzing relatively small texts though the lengths of solved texts are always much larger than the unicity distance of the ciphers we are dealing with.

As mentioned, the generalized version of the algorithm (Algorithm 1) can be applied to other simple ciphers although it is probably not possible to optimize it so that one can avoid reparsing the text in each iteration (like in Algorithm 2). Algorithm 1, however, is applicable whenever it is possible to construct a "closeness"-function. Normally this is obtainable only for ciphers where a small change in the key produces a small change in the ciphertext. Therefore this approach is not immediately useful for attacking the more modern type of encryption algorithms (IDEA, DES, etc.)

Finally it should be noted, that there is room for other small improvements as well, such as using a better method for deciding which pairs to swap. Also we might improve the evaluation function so that it not only depends on digrams, but perhaps also on trigrams, probable words and so on.

References

[1] Carroll, J. M. and S. Martin. 1986. The Automated Cryptanalysis of Substitution Ciphers. *Cryptologia.* 10(4): 193-209.

[2] Carroll, J. M. and L. Robbins. 1987. The Automated Cryptanalysis of Polyalphabetic Ciphers. *Cryptologia.* 11(4): 193-205.

[3] Carroll, L. 1991. *Alice in Wonderland.* Project Gutenberg[1].

[4] Gaines, H. F. 1956. *Cryptanalysis.* New York: Dover Publications.

[5] King, J. C. and D. R. Bahler. 1992. Probabilistic Relaxation in Cryptanalysis. *Cryptologia.* 16(3): 215-225.

[6] Melville, H. 1991. *Moby Dick.* Project Gutenberg (converted to etext by Professor Eugene F. Irey at the University of Colorado).

[7] Sinkov, A. 1966. *Elementary Cryptanalysis.* The Mathematical Association of America.

8 Acknowledgements

Thanks must go to Frank Nielsen from the Mathematical Institute at The Technical University of Denmark for commenting on this text and to Brian J. Winkel for being so patient.

9 Biographical sketch

Thomas Jakobsen, 23, is currently an undergraduate at The Technical University of Denmark. His main interests are computer science, cryptology

[1] The Project Gutenberg files are available on the Internet via anonymous ftp to mrcnext.cso.uiuc.edu.

and mathematics. This term he studies at the Institute of Signal Processing at the Swiss Federal Institute of Technology, Zurich. He hopes to do his master's thesis in cryptology.