DRAFT - WORK IN PROGRESS

Fast, Reliable Cryptanalysis Of Simple Substitution Ciphers Without Word Divisions

JACOB GAJEK

Abstract

We present a fast and reliable method for the automated cryptanalysis of short cryptograms based on monoalphabetic substitution ciphers without word divisions. Our approach combines a letter *n*-gram model with a genetic algorithm in order to efficiently arrive at the correct solution. Empirical results significantly outperform those of previously published methods.

Introduction

Within the past two decades, a number of methods for the automated solution of simple substitution ciphers have been proposed [References: Previous papers]. Some have been quite successful in cases where word spacing has been preserved or when there is a moderate amount of ciphertext available. However, the most challenging variant of the problem, where word divisions have been eliminated and there is only a small amount of ciphertext, has until now required the manual intervention of a human cryptanalyst in order to complete the solution. Both *n*-gram models and genetic algorithms have been applied to this problem before, but never together [References]. In this paper, we demonstrate a combined approach which is able to reliably produce a full decipherment in a completely automated manner.

We start with a succinct definition of the problem. Given a short ciphertext *C* known to have been generated from a plaintext message *M* by encryption with a monoalphabetic substitution cipher, we wish to recover the substitution scheme (henceforth referred to as the decryption key *K*) that can be used to re-generate *M* from *C*. The word divisions and punctuation have been removed, but it is assumed that we know the language of the underlying plaintext. For our present purposes, we will assume that language to be English, but this need not be the case in general. An example ciphertext is shown in Figure 1.

```
LRNTN TLGOH FNPHG LESRH AIGHH PMGAX HARVJ NAWLM VKJNT HLRHC
OPFNZ XVETC HVYIG HHSRN ZRRHS RMZVA VAKSN FFKHT HGJHT RNWRC
GVNTH SRMAH NLRHG ZVAAM GSNFF XVERM FKRNT CHVZH
```

A computationally useful way of representing *K* is a permutation of the 26 letters of the English alphabet. For example, the decryption key NZPJYLREFVDTOIUBXHWSQAGMKC indicates that the letter A in the ciphertext should be replaced by N, the letter B by Z, C by P, and so on. The number of possible keys is 26!, which is a truly staggering quantity[1]. We thus quickly see the infeasibility of an exhaustive search of the key space.

Genetic Algorithms

1 $26! \sim 4.03 \times 10^{26}$. If the computing power of a distributed cluster of one billion nodes, each capable of testing one million keys per second, could be brought to bear on the task, it would take on average over 6,000 years to find the solution with a brute force search.

Our first step will be to recast our scenario as an optimization problem suitable for attacking with a genetic algorithm. This approach was first explored by [Reference: Spillman et al.] with encouraging results. We will follow its general outline, but deviate significantly in the details.

Briefly, a genetic algorithm is a heuristic search methodology inspired by the process of natural selection in the evolution of biological species. A population of strings which are encodings of solutions to an optimization problem is "evolved" toward better solutions. The initial population is usually generated randomly, although it can sometimes be seeded with partially optimized characteristics. Successive generations are then created by selecting individual solutions based on fitness, and combining/mutating them to form a new population which has a higher average fitness. This process continues until a maximum number of generations is reached or a solution with an acceptable level of fitness is found. Figure 2 shows the typical flow of a genetic algorithm.

[Figure 2: Typical flow of a genetic algorithm]

We thus have three tasks before us. They are:

1. To find a suitable fitness function which evaluates potential decryption keys according to the "Englishness" of the deciphered text they produce
2. To find suitable mating and mutation functions which lead to a steady improvement in the average fitness of the population from generation to generation
3. To empirically tune the parameters of the genetic algorithm (population size, mutation rate, number of generations, etc.) in order to maximize performance and reliability

We will consider each of these tasks in turn.

Fitness Function

In order for the genetic algorithm to be of use, there must be a way to compare any two candidate keys and decide which one is a better solution to our problem. The typical approach is to devise a fitness function that takes as input a candidate key, and outputs a numeric score as a relative measure of how "good" the key is. Presumably, when the fitness function takes on its maximum value, we will have found the key that correctly decrypts the ciphertext. However, the meaning of "good" in the context of our problem is not immediately obvious. It is simple enough to recognize when we have the correct key: The result of the decipherment will be perfectly meaningful English text. Things get more difficult, however, when we have two keys that both produce random-looking streams of letters. How do we determine their relative goodness in that situation?

Rather than proceed directly to the answer, let us first examine some fitness criteria that have been used in the past, and analyze their merits and shortcomings. This will motivate the search for a better fitness function. An initial attempt at a scoring criterion might involve the individual letter frequencies occurring in the ciphertext. It has long been recognized that alphabet-based languages such as English exhibit characteristic letter frequencies (see Figure 3) and that this information could be used in a cryptanalytic scenario. To make use of this fact, we might devise a fitness function that assigns higher scores to candidate keys which map ciphertext letters to the plaintext letters with the closest-matching relative frequencies. There are two serious shortcomings to this approach, however.

[Figure 3: Characteristic letter frequencies for English]

First, the characteristic letter frequencies hold only in situations where the plaintext is very long, typically on the order of 10,000 characters or more. When much less ciphertext is available, the observed letter frequencies may deviate widely from their statistical averages, and in that case the characteristic frequencies can only be a rough starting point for human cryptanalysts, not an exact law[2]. Since our goal is the automated decipherment of short cryptograms, we need a more dependable method of attack. Second, clever adversaries (read: cryptogram puzzle creators) may have gone to great lengths to manipulate the word selection and sentence structure of their plaintext message so as to make any attempt at letter frequency analysis a futile endeavour[3].

Continuing the search for a suitable fitness function, we might next try to incorporate digram frequencies in the scoring formula. Digrams, i.e. two letters occurring next to each other, also exhibit characteristic frequencies in long runs of English text (see Figure 4). It is easy to see why digram frequencies are more reliable as an indicator for our purposes than the frequencies of individual letters alone: Unigram statistics treat the text as a sequence of independent random variables, and do not take into account localized patterns and inter-dependencies between neighbouring letters.

[Figure 4: Characteristic digram frequencies for English]

In the early 1900's, the Russian mathematician A. A. Markov carried out a statistical analysis of the text from Pushkin's novel Eugene Onegin. By re-arranging the text in various ways and examining vowel frequencies, he was able to demonstrate the following simple but important result: "As we can see, the probability of a letter to be a vowel changes considerably depending upon which letter – vowel or consonant – precedes it" [Reference: David Link] . Generalizing Markov's astute observation leads directly to a viable fitness function, as we shall see next.

_n_-gram Language Models

The idea of approximating written language as a stochastic process originated with C. E. Shannon in his seminal work on information theory [Reference: Shannon]. If we imagine that a string of English text of length L is generated by an automaton that emits one letter at a time, with a probability that depends only on the previously generated letters, then the probability that a string $S = s_1...s_L$ will be emitted is given by:

$$P(S) = P(s_1)P(s_2|s_1)P(s_3|s_1s_2)...P(s_L|s_1s_2...s_{L-2}s_{L-1})$$

If we now make the Markov assumption that only the last *n-1* of the previously emitted letters determine the probability of the next letter, then the probability of *S* becomes:

$$P(S) = P(s_1s_2...s_{n-1})\prod_{i=n}^{L} P(s_i|s_{i-n+1}...s_{i-1})$$

2 In fact, it may be argued that no universally applicable frequency characteristics exist, since various linguistic styles, dialects and literary genres will have their own unique characteristics.

3 Lipograms are extreme examples of letter frequency manipulation. The 1939 novel Gadsby by Ernest Vincent Wright is a 50,000-word composition without a single instance of the letter 'E', the most frequent letter in the English alphabet.

This probability distribution for strings *S* constitutes the *n*-gram model of order *n* for English. It should be noted that such models can also be constructed on the basis of words instead of letters[4].

We are now ready to define our fitness function: Given a candidate key *K*, we decrypt the ciphertext *C*, thus obtaining a "plaintext" $M_K = m_1 \ldots m_L$, where *L* is the length of M_K in characters. For each character m_i in M_K, the *n*-gram model assigns a conditional probability $P(m_i \mid m_{i-n+1} \ldots m_{i-1})$ based on the preceding *n-1* characters. We define the fitness score as the product:

$$FITNESS(K) = P(m_1 m_2 \ldots m_{n-1}) \prod_{i=n}^{L} P(m_i | m_{i-n+1} \ldots m_{i-1})$$

This score can be interpreted as the probability that the string M_K would be the outcome if we were to use our *n*-gram model to stochastically generate an English-like text string of length *L*. If our *n*-gram model is accurate and consistently produces strings resembling actual English, then we can reasonably expect that the higher this probability, the more closely M_K resembles a meaningful message.

Training *n*-gram Models

The process of counting *n*-grams in a large, representative corpus of text is known as "training" the *n*-gram model. The frequency data thus obtained are used to estimate the conditional probability distribution. As a rule, higher-order *n*-gram models require larger training corpora, because there are more *n*-grams to estimate probabilities for. The genre of the training material matters as well.

Practical Considerations

When working with *n*-gram models, two practical considerations must be addressed in order to obtain satisfactory results. The first arises from the limitations of finite-precision floating-point arithmetic. Since the probabilities involved in computing the fitness score are usually very small, the multiplication over all characters m_i is likely to produce an underflow condition. Moreover, multiplication is much slower than addition on typical floating-point hardware. Therefore, the computation of the score is best carried out in logarithmic space, where the product becomes a sum:

$$FITNESS(K) = \log P(m_1 m_2 \ldots m_{n-1}) + \sum_{i=n}^{L} \log P(m_i | m_{i-n+1} \ldots m_{i-1})$$

Since the logarithm is a monotonically increasing function in the interval *(0, 1]*, it does not change the relative ranking of candidate keys. It can be much more efficient computationally, however, since the logarithms for each conditional probability can be pre-calculated and stored in a look-up table.

The second consideration arises from the sparseness of the data on which the conditional probabilities are usually based. This is illustrated in Figure 4, where a training corpus consisting of approximately 20 million words of English text covers less and less of the *n*-gram space the higher the order of the *n*-gram model. For *n*-grams which never occur in the training corpus, the model discussed above assigns

4 Word-based models are actually more common than letter-based models in linguistic applications. However, they are clearly not appropriate in our present scenario.

a probability of zero. This becomes a serious problem with higher-order models ($n > 4$), where a significant percentage of n-grams will remain unseen no matter how much training data is available. A number of techniques for estimating non-zero probabilities for unseen n-grams have been developed [Reference: Manning and Schütze]. Among these, the class of Good-Turing estimators seem to be especially effective [Reference: Chen and Goodman], although they have a reputation for being somewhat difficult to implement. Fortunately, a variant called Simple Good-Turing has been published with source code [Reference: Gale and Sampson], and we found it to work extremely well in our present scenario.

Mating Function

Having at our disposal a powerful method for evaluating the "Englishness" of text, we next turn our attention to the mating step of the genetic algorithm. The overarching goal here is to take two candidate keys K_1, K_2 (the parents) and combine them to form a new key K_3 (the child). The manner in which we accomplish this step is subject to two constraints. First, the outcome of the mating operation must be a valid decryption key, i.e. it must be a permutation of the 26 letters of the English alphabet. Second, the fitness score of the child must be at least as good as that of the parents, otherwise the algorithm would not proceed in the direction of the correct solution. Taken together, the above constraints suggest the following intuitive approach:

1. Choose the fittest of the two parent keys as the starting-off point, K_S.
2. Compare the first character in K_S with the first character in the other parent key.
3. If the characters are equal, do nothing.
4. If the characters are not equal, perform the appropriate character swap in K_S to make them equal.
5. If the fitness of K_S after performing the swap is lower than it was before the swap, undo the swap. Otherwise, keep it.
6. Repeat steps 2 to 5 for each character position in K_S, scanning from left to right.

It is easy to see that the above procedure guarantees the resulting child will be a valid decryption key (since we are performing swaps internally within one of the parent keys only), and that its fitness score will be at least as good as that of the parent keys.

It remains to be noted that our algorithm to select the parent keys serving as inputs to the mating operation is biased towards keys with a higher fitness score. In a population of size k, the key with the best fitness score is k times more likely to be selected than the key with the poorest score. The number of mating operations performed in each generation is chosen so as to keep the population size stable throughout.

Mutation Function

In order to reduce the likelihood of the genetic algorithm becoming stuck after converging to a local optimum of the fitness function, it is desirable to inject some random variation into the key population. We accomplish this by applying a mutation function to each child key immediately after the mating step. Our mutation function is very simple: any character in a child key has a slight chance[5] of being

5 The probability of mutation is controlled by the mutation rate parameter of the algorithm.

swapped with another, randomly selected character within the same key.

Initial Population

The final procedure remaining to be defined is the generation of the initial key population. Although simply generating permutations at random is certainly a viable approach, we can do somewhat better. Invoking Sukhotin's algorithm [Reference: Guy] we can identify those letters in the ciphertext which map to vowels in the original plaintext with a fair degree of accuracy. By generating random permutations of the vowels and consonants separately, and then combining them into complete keys, we can ensure that the initial key population starts off with some of the correct substitution mappings already in place.

Results

A large collection of cryptograms ranging in length from 50 to 2500 characters was used to test our method. To keep the tests fair, none of the cryptograms were derived from the corpus text on which the *n*-gram model was trained. Figure 5 shows the percentage of correctly decrypted characters as a function of cryptogram length for *n*-gram models ranging from $n = 2$ to $n = 5$. As is evident from the results, it is possible to achieve perfect accuracy on cryptogram lengths as short as 200 characters with a trigram model. With higher-order *n*-gram models, even shorter cryptograms can be reliably solved, subject to the limitation on the size of the text corpus that is available to train the model. It is conceivable that with a training corpus of sufficient size, a 6-gram or 7-gram model could be used to attack cryptograms approaching the 28-character estimate for the unicity distance of English [Reference: Shannon].

Open Source Implementation

[Pointer to the Alkindus source code, training corpus and cryptogram test collection here]