A Word-Based Genetic Algorithm for Cryptanalysis of Short Cryptograms

Ralph Morelli and **Ralph Walde**
Computer Science Department
Trinity College
Hartford, CT 06106
ralph.morelli@trincoll.edu

Abstract

This paper demonstrates the feasibility of a word-based genetic algorithm (GA) for solving short substitution cryptograms such as the kind found in newspapers and puzzle books. Although GA's based on analysis of letter, digram, or trigram frequencies have been used on substitution cryptograms, they are not able to solve short (10-30 word) cryptograms of the sort we address. By using a relatively small dictionary of frequent words to initialize a set of substitution keys, and by employing a word-based crossing mechanism, the GA achieves performance that is comparable to deterministic word-based algorithms.

Introduction

This paper describes a word-based genetic algorithm for solving simple substitution cryptograms such as the kind found in newspapers and in puzzle books. These are relatively short (10-30 word) cryptograms with word boundaries and punctuation. The *ciphertext* is created by choosing a permutation of the 26-character alphabet and using it to replace each letter in the *plaintext* message:

- **Cryptogram:** pkji oexn agnxn x gh tjxit oj nejr vjp ejr g apide jy amxteo vjpit yjfcn sxs yxis g deghkxji; g hgi rxoe ajvn gis txmfn jy exn jri.

A solution to this cryptogram must successfully choose the correct permutation from among 26! (around 4×10^{26}) possible permutations. In the solution shown here, the GA found all words except *bunch* and *champion*:

- **Plaintext :** upon this basis i am going to show you how a bun**x**h of bright young folks did find a **x**hampion; a man with boys and girls of his own.

Most approaches for automatic cryptanalysis of simple substitution ciphers are based on frequency analyses (see (8), (5), (6), (7), (9), (4)). These approaches are quite effective – achieving nearly 100 percent accuracy – provided that the ciphertext is long enough and contains a fairly normal distribution of the English letters. Most frequency-based approaches require between 400 to 1000 characters of ciphertext to be effective. However, since most puzzle cryptograms are very short, frequency-based algorithms cannot be used successfully in their solution. Note, for example, that the above cryptogram contains only 132 characters (including spaces) and has no occurrences of the letter 'e', the most frequent English letter.

A successful approach for solving short cryptograms was developed by Hart (2). This approach is based on *word frequencies* rather than letter frequencies. What's surprising about the word-based approach is that one can perform a successful analysis with a relatively small dictionary. As Hart points out, although there are over 100,000 words in English, a randomly selected word from an English text has more than a 50 percent chance of occurring in a dictionary consisting of the 135 most frequent words (2).

Hart's algorithm uses a deterministic depth-first search through a tree of word assignments, where each ciphertext token is paired with a set of possible matching words from the dictionary. As the search moves down the tree, a partial substitution key is constructed. The tree is pruned whenever a new token-word pair would be incompatible with the partial key. Hart is able to decode cryptograms containing as few as 10 words with acceptable speed. The speed and overall success of Hart's algorithm varies with the size of the dictionary. The larger the dictionary, the more words the algorithm can find, although its speed will diminish dramatically. If the message contains words not in the dictionary, a complete solution may not be found.

Building on some of Hart's ideas, this paper demonstrates the feasibility of a word-based GA for solving short cryptograms.

Related Work

Most algorithms for cryptanalyzing substitution ciphers are based on frequency analysis. Peleg and Rosenfeld and King and Bahler use a *probabilistic relaxation* algorithm to solve simple substitution ciphers (8), (5). This is a classification algorithm in which letters of the ciphertext are matched with certain plaintext letters by analyzing the letter, digram or trigram frequencies in the ciphertext. These algorithms are very effective, provided that the ciphertext contains enough characters. In Peleg and Rosenfeld's study, which used texts containing around 1000 characters, they were able to find the correct key within 15 iterations. Using a similar approach, King and Bahler found their algorithm was effective for texts of 400 characters or longer. Their algorithm converged to the correct solution after only 2 or 3 iterations. King has extended this approach with success to polyalphabetic substitution ciphers, achieving 100 percent success for texts with at least 800 characters (5). A similar approach that incorporated various speed-up techniques, with comparable results, was described by Jacobsen (4).

Genetic algorithms for analyzing substitution ciphers have also been based on frequency analysis. Spillman *et al.* used both single character frequencies and digram frequencies to calculate the fitness of their population (9). Matthews used a genetic algorithm based on frequency analysis to analyze transposition ciphers, with similarly good results (7).

Genetic Algorithms

The basic idea behind genetic algorithms is to model the natural selection process, in which members are selected from the population and mated to each other to create a new generation of individuals (3), (1). The process begins by creating a random initial generation of individuals, sometimes called *chromosomes*, that in some way represent the problem being solved. Pairs of members of the current population are selected and "mated" with each other by means of a *crossover* operation to produce members for the succeeding generation. Randomly selected members of the current generation also undergo *mutation*, in which random portions of "genetic" material are exchanged. The fittest members of each generation, as determined by a *fitness function* are then selected for the succeeding generation (Figure 1). The crossover and mutation operations are controlled by the *crossover rate* and *mutation rate* parameters, which determine the proportion of the population that undergoes these changes.

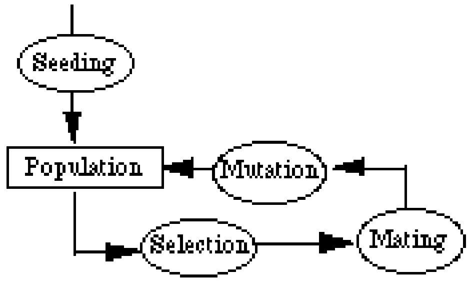

Figure 1: The basic genetic algorithm.

In a GA for breaking a substitution cryptogram, the mating and mutation processes are used to generate a search through the cryptogram's key space. The search is guided in a nondeterministic way by the fitness function, which measures how well a given key succeeds in decrypting the message. In our algorithm, the more words found in the evaluation dictionary, the higher the fitness value.

Using the GA to Break Short Cryptograms

The purpose of this study was to determine how well a word-based GA could solve short cryptograms. What attracted us to the word-based approach is that it works well on short cryptograms and it is closer to the way a human solves such cryptograms. In this section we describe the main features of our algorithm.

Representation of the Key

In our GA, the chromosomes correspond to the substitution keys, with each chromosome representing a different key. A key is represented as a permutation of the 26 letters of the alphabet, arranged in an ordered sequence, with the first letter representing the letter that would be substituted for *A*, the second for *B*, and so on. For example, in the following permutation,

```
SUQHCLADFGJKNOPTVWXZBEMRIY
```

the letter *S* would be substituted in the cryptogram for plaintext *A*, *U* for *B*, and so on. For example, the word "CAB" would be encrypted as "QSU."

Our GA uses a *pattern dictionary* to construct a collection of token-word pairs in which every token in the cryptogram is paired with all dictionary words having the same pattern. The token-word pairs are then used to seed the chromosomes. For example, suppose the first word in the cryptogram is *bzdyd*. This matches dictionary words *these*, *there*, and *where*. Therefore one of the chromosomes would be based on the pair *bzdyd=these*, with *b* substituted for *t*, *z* for *h*, and so on. Each chromosome is thus seeded with one or more randomly selected token-word pairs.

The pattern dictionary is constructed from the machine-readable Kucera-Francis word list of the *N* most frequent words (10). We've experimented with dictionaries as small as 50 words and as large as 3500. In the current implementation we use a dictionary of 50 words to seed the keys and a dictionary of 3500 words to evaluate the keys.

Fitness Function

The function used to determine the fitness of a chromosome is based on applying the key to the cryptogram and then counting the number of dictionary words produced. The more words produced, the better the key:

- Calculate the fitness of this chromosome:
  1. Use the key to decrypt the message.
  2. Fitness = the number of distinct words in the decrypted message that are in the fitness dictionary.
  3. Return fitness.

Even though it is somewhat coarse-grained – for a cryptogram containing 10 tokens the possible scores would range from 0 to 10 – this simple metric works surprisingly well.

Mating and Mutation Processes

The mating process crosses two parent chromosomes to produce two children, where each child contains "genetic material" (letters from the key) from both parents. For each pair of parents, whether a cross takes place depends on the crossover parameter. If set to 0.9, approximately 90% of possible crosses will be performed in each generation. In the current model, children compete with their parents for survival into the next generation, for which the fittest individuals are selected.

The crossover mechanism itself works as follows:

- For each parent, p1 and p2
  1. Collect those letters that are used in words found in the fitness dictionary.
  2. Exchange those letters with the corresponding letters in the other parent's key to produce the child's key.

For example, suppose we have the following pairs of chromosomes.

```
   Alpha:  ABCDEFGHIJKLMNOPQRSTUVWXYZ
 Parent1:  SUQHCLADFGJKNOPTVWXZBEMRIY
 Parent2:  AZBYXCWDEVUFGTSHRIQJPKLOMN
```

Suppose that parent1's key is able to find only the word AND and that parent2's key is able to find only the word BIG. The word AND corresponds to SOH in parent1 and ATY in parent2. The word BIG corresponds to UFA in parent1 and ZEW in parent2. If we use parent1's key to initialize child1 (c1) and parent2's key to initialize child2 (c2), then the following swaps would be performed: .

```
swap(S,A) in c2   swap(U,Z) in c1
swap(O,T) in c2   swap(F,E) in c1
swap(H,Y) in c2   swap(A,W) in c1
```

It should be obvious that this mechanism preserves the validity of each key because each key is still a permutation of the 26 letters of the alphabet.

```
 Alpha:  ABCDEFGHIJKLMNOPQRSTUVWXYZ
Child1:  SZQHCLWDEGJKNOPTVAXUBFMRIY
Child2:  SZBHXCWDEVUFGOAYRIQJPKLTMN
```

Assuming that before the cross parent1 found only AND and parent2 found only BIG, then both children benefit from the swap because they both now find AND and BIG. This can be seen by noting that both keys have the same letters associated with AND (SOH) and BIG (ZEW). As a result of this cross, the children's score would increase from 1 to 2. The children would replace the parents in the population.

Note that this mechanism is not guaranteed to improve the children. Suppose in the preceding example that both parents were able to find other words beside AND and BIG. In that case while it is true that both children will decrypt AND and BIG the changes might have messed up other words. For example, suppose some of the letters in SOH were used by child2 to find the word ART. Then by swapping SOH with other letters, child2 is no longer be able to find ART. In that case its score might actually decrease. On the other hand, blocking child2 from finding ART might make it possible for it to find RANT and NOT, thereby increasing its score. However, even though it can produce children with lower scores, our experiments have revealed that this mechanism improves the fitness of the children roughly 3 times as often as it reduces their fitness. (Experimentally, a majority of crosses result in no change in the fitness.)

By using an *occurs check* it would be a simple matter to modify the crossing algorithm to guarantee that crosses never decrease the child's score. In the preceding example, because swapping SOH would drop ART from child2 the occurs check would prevent that from happening. This would guarantee that child2 could not be worse than its parent. However, in our experiments use of the occurs check did not improve the overall performance of the GA. It appears to make it more likely that the algorithm gets stuck at a local maximum rather than finding the true key. The fact that a certain percentage of keys have (erroneous) words dropped from found lists seems to act like a random perturbation that jolts the GA out of that kind of rut.

The final element of the GA is the mutation process. This process is controlled by the *mutation rate*, a parameter that determines the proportion of individuals in each generation that undergo mutation. In our experiments the mutation rate has been varied between 0.1 and 1.0. The mutation mechanism is simply to swap two characters chosen at random in the key. In the current model, mutated individuals competes with the rest of the population to survive into the next generation. This mechanism seems to play a role in the GA's ability to escape from local maxima.

Results

The algorithm described in the preceding sections is controlled by a number of parameters. We've already mentioned the *crossover rate*, *mutation rate*, and *occurs check*. Other parameters that we've varied are *population size*, the number of individuals in the population, and *dictionary size*, the number of frequent words used to construct the pattern dictionary. We've run many experiments during development and testing of the GA. Since we are mainly interested in the overall feasibility of the GA approach to this problem, the results we describe here use the settings given in Table 1.

Parameter	**Setting**
Population Size	512 individuals
Seeding Dictionary Size	50 words
Fitness Dictionary Size	3500 words
Crossover Rate	0.9
Mutation Rate	0.2
Occurs Check	Off

Table 1: Parameter settings for GA Cryptanalyzer.

We've tested the GA on a variety of plaintext messages, ranging from extremely short messages ("to be or not to be that is the question"), to messages missing the letter *E* ("upon this basis i am going to show you how a ..."), from messages whose words are all contained in the fitness dictionary to messages with 1/4 or 1/2 of their words missing from the dictionary. (Note that because the keys are initially assigned random permutation of the alphabets, the plaintext messages are no easier for the GA to solve than encrypted messages.) Table 2 summarizes the results we achieved for a representative sample of these messages. For each message we show the number of distinct tokens (*nToks*) it contains, the number of its tokens that decrypt to words in the evaluation dictionary (*nWds*), the average number of words found per trial (*nFound*), the average number of generations (maximum 500) required to find all the words (*nGens*), and the number of trial runs (*nTrials*).

msg	nToks	nWds	nFound	nGens	nTrials
1	8	8	7.92	134.6	25
2	26	21	21	15.48	25
3	22	22	22	39.4	20
4	24	19	19	35.1	15
5	21	12	11.45	244.4	20

Table 2: Performance results for five plaintext messages.

Message 1 is the phrase “to be or not that is the question”. Message 2 is the message shown in the introduction, which is taken from Ernest Wright’s novel *Gadsby*, a 267-page novel with no occurrences of the letter E. Message 5 is an example of a typical published cryptogram, which are frequently chosen to be difficult to solve. This message contains only 12 dictionary words out of 21 tokens:

```
I shoot the hippopotamous with bullets
made of platinum because if I use leaden
ones his hide is sure to flatten 'em.
```

Messages 1, 2, and 5 were also used in Hart’s study (2). Message 3 is a combination of “the quick brown fox jumped over the lazy dog” and “now is the time for all good men to come to the aid of their country”. What’s interesting about this message is the singleton occurrences of the infrequent letters X, Z, Q, and K. Message 4 is a passage, picked more or less at random from the Dorothy Sayers novel *Have His Carcase*.

For this study we let the algorithm run for 500 generations or until the top 3 individuals had all achieved scores equal to *nWds*. This is the best score that the algorithm can achieve given its current scoring dictionary. And this is our criterion for success. The fact that the GA cannot always find all the words – unless an extremely large dictionary is used – is a tradeoff that we discuss below. However, even for message 5, which contained only 12 words out of 21 tokens, the algorithm usually finds enough correct letter mappings to make the rest of the cryptogram solvable by visual examination.

This set of results shows that the algorithm is both highly successful at finding the key and highly efficient. Out of the trials compiled here, the algorithm achieved complete success – defined as finding all the words that could be found – in 94 out of 105 trials, a 90% success rate. The average number of generations required for all 105 trials was 94. But this average is somewhat misleading. It is skewed by the 15% of the times that the algorithm got “stuck” at a local maximum. In the 105 trials, there were 15 trials which ran over 250 generations. In 69 out of 105 trials (66%), the algorithm converged on the solution in fewer than 35 generations. One of the challenges of further research in this area would be to determine if the algorithm’s occasional propensity to “get in a rut” is something that can be addressed by design or parametric changes or is due purely to chance.

To test the relative contributions of mutation and crossover in our algorithm we ran it on the original set of messages with the mutation rate set to 1.0 and the crossover rate set to 0. The algorithm repeatedly failed to converge to a solution. This shows that both mechanisms – crossover and mutation – play a significant role in the algorithm’s success.

This is not to downplay the significant role played by mutation in our algorithm. Indeed, in further trials we found that the algorithm converged quickest with a mutation rate of 1.0. This means that mutation – the swapping of two random characters in the key – is performed on *every* individual. One possible explanation for this result is that our crossover mechanism tends to preserve those portions of the key that are associated with the words found by that key. With a low mutation rate, a key that has found the wrong words is more likely to get stuck in a rut. Alternatively, with a high mutation rate those portions of the key that code for the wrong words are more likely to get broken up. In any case, the relative contributions made by our crossover and mutation mechanisms is a question that we need to address in future studies.

We also tested the algorithm with a very large (100,000+ word) evaluation dictionary. In these trials the algorithm invariably performed worse than with the 3500-word evaluation dictionary. Our explanation for this somewhat surprising result is that using a large dictionary disproportionately rewards keys containing wrong words. Obviously, this too is an area for further study.

Finally, we have tested the algorithm on seven actual cryptograms, all of which were published by Eliot Sperber in different editions of the Sunday *Hartford Courant*. For these trials the only change in parameter settings from Table 1 is that the mutation rate was set to 1.0. These results are summarized in Table 3. For each message we show the number of distinct tokens it contains (*nToks*), the number of its tokens that decrypt to words in the the evaluation dictionary (*nWds*), the percentage of tokens that were correctly identified (*%Found*), the average number of generations (maximum 100) required to solve the message (*nGens*), and the number of trial runs (*nTrials*).

msg	nToks	nWds	%Found	nGens	nTrials
1	24	19	97.4	21.1	50
2	33	22	99.5	16.2	50
3	23	14	61.5	46.9	50
4	23	12	58.6	52.4	50
5	26	17	95.7	16.1	50
6	24	22	93.1	57.8	50
7	25	16	74.0	46.2	50

Table 3: Performance results for seven cryptograms.

As in our earlier experiments, these results show that the algorithm is efficient and successful at finding the key. Out of 350 runs, the algorithm terminated with a solution 338 times (96.6%). In 121 cases (34.6%) the algorithm got the cryptogram 100% correct – that is, it decrypted every token correctly. In 192 cases (54.8%), it got at least 90% of the tokens correct. And in 236 cases (67.4%), it got at least 75% of the tokens correct. Messages in which 75% of the tokens were correct could usually be read directly. Messages with fewer correct tokens could often be solved with additional visual analysis. Of course, for several of these messages (numbers 2, 3, 4, 5, and 7), fewer than 66% of the tokens were contained in the evaluation dictionary.

These results also show that the algorithm's performance varies considerably depending on the message itself. For example, note that for messages 2 and 5, the algorithm was highly successful even though only around two thirds of the tokens in those messages were dictionary words. On the other hand, the algorithm was less successful on message 6, in which more than 90% of its tokens were words. In our view, more analysis of these results is needed.

Discussion

There are several conclusions that can be drawn from this study. The overall conclusion is that the word-based GA compares favorably with results that have been reported in the literature for both deterministic keys searches, such as depth-first search (2), and with frequency-based GAs (9). The following specific points are also worth noting.

- Our results indicate that a word-based genetic algorithm can be highly successful at breaking short cryptograms. In the great majority of our trials the GA found the key in relatively few generations using an evaluation dictionary of only 3500 words.
- Although the GA's success rate is clearly dependent on the size of the dictionary, because of the algorithm's nondeterministic nature, its efficiency is not degraded as much as a deterministic search when the dictionary size is increased. However, its success rate does not appear to improve dramatically when a very large evaluation dictionary is used.
- Compared to frequency-based algorithms reported in the literature, which require at least 400 characters to be effective, the word-based GA is highly successful at solving short cryptograms and cryptograms with unusual character distributions.
- The main weaknesses of this approach appears to be its sensitivity to dictionary size and its heavy dependence on randomness, as indicated by its good performance with a high mutation rate. However, to some degree these are inherent tradeoffs that do not affect practical problem solving. If the GA fails on a given message on one trial, it is likely to succeed on the next.

Plans for the Future

The results presented in this paper address only the overall effectiveness of a word-based GA approach to solving short cryptograms. There are a number of issues that we have not yet examined that could be the focus for additional research.

- Is there any way to rescue the algorithm when its gets stuck? The current system "tweaks" the algorithm by replacing the population with new individuals.
- What is the relationship between words in the message and the dictionary size? Why does the algorithm succeed so easily with some messages and not others with a comparable word-to-token ratio?
- Can the GA's success with substitution ciphers be extended to other kinds of ciphers such as polyalphabetic and transposition ciphers or to other applications?

Acknowledgements

The authors would like to thank the FLAIRS reviewers. We have run additional experiments and have incorporated some of their suggestions in the paper.

References

1. D. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning.* Reading, MA: Addison-Wesley. 1989.
2. G. Hart. To decode short cryptograms. *CACM, 37(9)*, pp. 102-108, 1994.
3. J.H. Holland. *Adaptation in Natural and Artificial Systems.* Ann Arbor, MI: University of Michigan Press. 1975.
4. T. Jacobsen. A fast method for cryptanalysis of substitution ciphers. *Cryptologia, 19(3)*, 1995.
5. J.C. King and D.R. Bahler. An implementation of probabilistic relaxation in the cryptanalysis of simple substitution ciphers. *Cryptologia, 16(3)*, pp. 215-225, 1992.
6. J.C. King. An algorithm for the complete automated cryptanalysis of periodic polyalphabetic substitution ciphers. *Cryptologia, 18(4)*, pp. 332-355, 1994.
7. R.A.J. Matthews. The use of genetic algorithms in cryptanalysis. *Cryptologia, 17(2)*, pp. 187-201, 1993.
8. S. Peleg and A. Rosenfeld. Breaking substitution ciphers using a relaxation algorithm. *CACM 22(11)*, pp. 598-605, 1979.
9. R. Spillman, M. Janssen, B. Nelson and M. Kepner. Use of a genetic algorithm in the cryptanalysis of simple substitution ciphers *Cryptologia, 17(1)*, pp. 31-44, 1993.
10. MRC Psycholinguistic Database, University of Western Australia, *http://www.psy.uwa.edu.au/mrcdatabase/uwa_mrc.htm*.