Solving Substitution Ciphers with Genetics Algorithm

Joe Gester

20th December 2003

Abstract

Genetic algorithms were used in an attempt to generally solve two classes of simple substitution ciphers. First, the full key space of all possible substitution ciphers was searched. When this approach was met with limited success, the simpler approach of searching the more likely used keyword generated key space was implemented. This yielded somewhat more results.

1 Background and Motivation

The simple substitution cipher has not been a secure form of communication for hundreds of years, yet a great deal about statistical analysis of cipher-text can be learned from investigating these insecure ciphers. Though certainly solvable, an attack on the general substitution cipher, with spaces and punctuation removed, is much more difficult than the cryptograms many people solve on the crossword page of the Sunday paper.

A generic substitution cipher takes as a key a mapping from each cipher-text character to a plain-text character. This is most easily described as twenty-six pairs of characters. There exist 26! or 403291461126605635584000000 such keys. This is a huge search space. Simply trying all possible keys in a brute force attack is not a viable option. Using first order statistics however and pattern matching deciphering these ciphers is almost trivial for a person with some time to work on it. [3]

Particularly interesting is breaking with these ciphers in an automated manner or through unusual means. A novel approach to the problem may reveal something interesting about cryptography in general or if successful might be applicable to other more secure ciphers.

2 Genetic Algorithms

A genetic algorithm is general method of solving problems to which no satisfactory, obvious, solution exists. It is based on the idea of emulating the evolution

of a species in nature and so the various components of the algorithm are roughly analogous to aspects of natural evolution.

Common mathematical tasks amenable to genetic solutions include computing a curve to fit a set of data or approximating NP problems. Creating AI programs to preform tasks like navigating a room or playing a complex game have also been addressed using these algorithms with great success [6]. A program which was evolved genetically even improvises live jazz with a human performer. [2]

A population of individuals is generated, typically randomly. Each of these individuals represents a possible candidate solution to the problem. These solutions usually take the form of a bit or character string, though in some cases more complex data structures like abstract program trees or multi-dimensional vectors are used. The representation of the individual solution is the most critical aspect of the algorithm. For the purposes of this project, several different individual descriptions were used.

In real evolution mutation and crossover are the driving mechanisms of genetic change. Similar genetic operators must be created to slowly modify the population of individuals. Often these operators consist of flipping a single random bit of one individual or swapping two randomly selected substrings from a pair of parents to generate a new child.

To simulate Darwinian survival of the fittest some representation of the fitness of the individuals must be generated. This is done by creating a function which applied to an individual evaluates its performance and assigns it a numerical rating of fitness. For example, when genetically evolving an algorithm to navigate an obstacle course the fitness function would be how far the individual makes it through the course in a set amount of time. Or, in trying to fit a curve with data, the fitness could be amount the curve misses the data by at each point. In this case, a lower fitness score would be desirable.

The main body of the algorithm is an iterative process of evaluating the fitness of the individuals in the population, selectively applying genetic operators to the members of the population to create a new generation and repeating. Each generation is created by randomly selecting members of the previous generation weighted according to their fitness. [5]

Genetic algorithms can be highly effective for certain types of problems and less effective in other types. They work well in situations where:

1. Known solutions are unsatisfactory for some reason, be it memory usage, run time or complexity of implementation.

2. Two very similar solutions have nearly the same fitness. That is, a small change to an individual in the population will produce a small change in fitness.

3. There exist multiple good solutions which would be satisfactory.

Violating these guidelines does not mean that a problem cannot be addressed with this method, it merely makes it more difficult.

3 Goal

Use a simple genetic algorithm to search the key space of cryptograms in an attempt to create a general solver for such problems. If this method is not satisfying, then attempt to search a smaller problem space by restricting the key searched to those generated by a keyword.

4 Prior Work

This is a solved problem. There are several good cryptogram solvers available on the Internet in a variety of languages, for many platforms. Most however rely on word frequency and dictionary matching to find their solutions. In the more general case, where spaces are not preserved after encryption these programs almost always fail. The few that do not use dictionary lookups seem to rely entirely on digram and trigram frequency to search recursively for a solution using permutations of the likely matches based on single letter frequencies. It seems that Genetic Algorithm based solvers have been attempted before as well.[8] Genetic algorithms have been used successfully to break more complex ciphers as well such as Enigma [1].

5 Implementation One

5.1 Individual Description

Each individual in the population represents a single guess at the correct key for the substitution cipher. Each gene represents a translation table mapping the characters of the cipher-text to the characters of plain-text. Using the key one can preform a lookup using a character of cipher-text to discover the character of plain-text it maps to. This is implemented as a list of characters, the first of which is the character mapped to by 'a', the second mapped to by 'b' and so on. This is simply implemented as a list indexed by the characters 'a' through 'z'. So, to preform a decryption using an individual is simple, just look up each character of the plain-text using the key and record the value found.

For example if the key is [n y r f c e t p x s u a w d g i k h q j z o m b v l] and the cipher-text in question is "gfqg" we look up 'g', the seventh letter in the key and get 't'. Then look up 'f', the sixth letter, to get 'e' and so forth to reveal the plain-text "test".

5.2 Fitness Function

The fitness function for rating the quality of each individual in the population's solution is based on trigram and digram counts. A corpus of text must be supplied. From this a table of the number of times each trigram and diagram occurs is generated.

To apply the fitness function to an individual, the cipher-text is decrypted using the individual's gene as it's key. Then every trigram and digram in the decrypted text is looked up in the table of how many times it occurs in the corpus. These numbers are then summed. Thus, trigrams and bigrams that occur commonly in the corpus are more heavily rewarded than those that do not.

Single letter frequencies could have been used in the fitness function but would likely have dominated the trigram and bigram fitness measures without adding much additional utility.

A convenient aspect of the use of a corpus is that this method should succeed on text in any language without modification. Moreover, the minor advantage of having a customized corpus for known subject matter is also realized.

5.3 Mutation and Crossover

Mutation is implemented simply as a random swap between two elements of the mapping list. This could be differently implemented as a swap between probable neighbors based on the single letter frequencies of English. That is, the character 'e' might be swapped with 't' but not with 'v' or 'x'. This seems, however, overly restrictive and generally inappropriate.

Crossover is more complex. In this implementation a pair of individuals produce a single new individual. A copy of one of the parent individuals is selected and a section of the other parent's gene/key is chosen. Then the character in each element of the selected region is found in the child's key and is swapped with the character currently in the desired position.

Thus for parents, one and two with keys [a b c d e f] and [b d f e c a] the section [b c d] is chosen from [a (b c d) e f]. The child is created as a copy of parent two: [b d f e c a]. Then, [b c d] is swapped into the range it occupied in the other parent.

1. d b f e c a
2. d b c e f a
3. e b c d f a

Yielding a single child with the key [e b c d f a], which should preserve a maximal amount of the structure of both parents while maintaining the uniqueness of the letters of the key.

The general idea here is to mate two parents and produce an offspring that has as much of the structure of the two parents as possible while maintaining the one-to-one and onto mapping of the cipher-text characters to plain-text characters. To preserve the mapping properly, repeated characters must be avoided, which means swapping character's of the key around instead of simply replacing them.

In theory, as long as less than half of the key is swapped at any one time, at least the good information from one parent remains in the child and likely the majority of the good information from the other parent also remains.

5.4 Population

Several different population configurations were tested. Using only a single large population resulted in a very homogeneous solution space at times which slows or even stops the progress of the algorithm. Consequently, several smaller sub-populations where used without and interbreeding between them. Typically about thirty subpopulations of forty or fifty individuals were used though increasing or decreasing the sizes by a factor of two or four made little difference.

5.5 Trials

For each generation of the population, each individual is scored with the fitness function. Then a new population is constructed by selecting randomly either mutation or crossover. Then a parent or a pair of parents are selected by choosing randomly between the members of the previous generation weighted according to their fitness. Thus a new, random population is built favoring the heritage of the most successful members of the last generation but still allowing some of the genetic material of the less successful members to propagate forward.

5.6 Results

This implementation does not work. Very rapidly the populations reach local maxima or never seem to converge to anything resembling English. Larger populations might yield more useful results but because of the interpreted nature of the language the program is implemented in, Python, running times are already long. Within 500 generations nearly all sub-populations reach stable states which in no way resemble English text.

The first suspicion is that the fitness function is not adequately distinguishing between good and bad candidates. A brief investigation shows that this is not so. The fitness function does indeed do a good job of evaluating the fitness of a candidate. A correct solution gets a very good fitness score while a random solution gets a very poor score with a smooth distribution in between.

The method used for crossover is also suspect; it is more of a crossover coupled with several mutations rather than a simple crossover. Because of the way space must be created in the destination array for the incoming cross, swapping the various other elements of the array out of the way adds a good deal of randomness. This is the problem the next implementation will address.

6 Revised Implementation

6.1 Individual Description

To allow for a simplified crossover mechanism it will be necessary to let the gene's have duplicate characters. This will mean that to maintain the necessary one-to-one mapping the duplicates will have to be removed later. This can be done by simply modifying the fitness function to replace each duplicate with a

letter that does not occur in the key. This should be done in a deterministic, not random, way so as to keep the calculation of fitness constant for a given individual.

6.2 Mutation and Crossover

Because, repeated characters are now allowed the crossover function can be implemented in a very straight-forward way: select a crossover point and copy everything before that point from one parent into the child and everything after that point from the other parent into the child. Thus returning to the example above, the two parents, [a b c d e f] and [b d f e c a], mate. The midpoint of the two keys is selected as the crossover point and the generated child is [a b c e c a]. This has two repeats which, when removed by the fitness function will yield [a b c e d f].

The introduction of duplicates introduces the problem of "breeding out" a character from the population. It is possible that a given character will be completely removed from all of the keys in the population. A simple way to fix this is to have a second mutation operator which replaces a duplicate character with one not found in the key.

6.3 Results

This works better but is still not very good. The average fitness does climb for several generations and occasionally climbs quite high but rarely is a usable result produced. Approximately one in fifty such results resemble English text and none were correct decryptions.

This idea is not working. Perhaps something simpler would. If the general solution is not working, tackling a subset of the problem might be easier.

7 Simplified Implementation

The most common way to implement a substitution cipher is not to use an entire mapping from a scrambled alphabet to the normally ordered alphabet. More commonly one remembers a key word and uses it to construct the mapping. For example using the keyword "textbook" one would generate the key [t e x b o k a c d f g h i j k l m n p q r s u v w y z]. This is done by removing the duplicate letters from the word and then appending the rest of the alphabet in order to it to generate the key mapping.

7.1 Individual Description

An individual in this case is simply an arbitrary length string of characters. The population is seeded with random English words of length four to ten characters. If at anytime the key should grow beyond twenty-six characters it will be truncated to twenty-six, this is an arbitrary restriction designed to prevent the strings from exploding to unbounded length.

7.2 Mutation and Crossover

Mutation can be done in the same way as in the original implementation. Two random characters are selected from the key and swapped. Also useful is a means to introduce new characters into the string by changing one character randomly into another. This will prevent the problem of "breeding out" characters from the population.

Crossover can simply be implemented by selecting a crossover point in each parent a copying as before. This also allows the lengths of the keys to change between generations, which is necessary for completeness.

7.3 Results

This final implementation does not work any better than the previous one. Fitnesses do climb but not to the correct goal but occasionally partially correct solutions are distinguishable.

8 Conclusions

It is clear based on prior work that this is a viable solution to the general substitution cipher. Precisely why it did not work in these implementations is worth exploring further.

Perhaps the fitness function is at fault here. It distinguishes well between solutions that perform well and those that perform poorly but is perhaps less well suited to find the differences between very similar solutions. If this were the case, the algorithm would have a difficult time getting started on finding a solution but once some progress had been made would more rapidly move towards higher fitnesses. This is a behavior that was at times observed.

Perhaps the crossover function is still not preserving enough good genetic information from the parents. This might be remedied by having a new individual description that does not use position in the string for any purpose. It could be an unordered list of pairs of characters.

These are all options worth exploring, however it is most certainly easier to give up on the genetic algorithm idea and pursue an ad-hoc heuristic approach to automating this sort of problem.

References

[1] A.J. Bagnall. The applications of genetic algorithms in cryptanalysis, 1996.

[2] J. Biles. Genjam: A genetic algorithm for generating jazz solos, 1994.

[3] Paul Garret. *Making, Breaking, Codes.* Prentice Hall, Upper Saddle River, 2001.

[4] Thomas Jakobsen. A fast method for cryptanalysis of substitution ciphers. *Cryptologia*, 19(3):265–274, 1995.

[5] John R. Koza. Genetically breeding populations of computer programs to solve problems in artificial intelligence. In *Proceedings of the Second International Conference on Tools for AI, Herndon, Virginia, USA*, pages 819–827. IEEE Computer Society Press, Los Alamitos, CA, USA, 6-9 1990.

[6] John R. Koza, Martin A. Keane, Jessen Yu, Forrest H Bennett III, and William Mydlowec. Automatic creation of human-competitive programs and controllers by means of genetic programming. *Genetic Programming and Evolvable Machines*, 1(1/2):121–164, 2000.

[7] Simon Singh. *The Code Book*. Anchor Books, New York, 1999.

[8] Edward Faldt Sofia Nilsson. Solving simple substitution ciphers using genetic programming, 2001.