THE UNIVERSITY OF CALGARY

Cryptanalysis Using Nature-Inspired Optimization Algorithms

by

Karel P. Bergmann

A THESIS
SUBMITTED TO THE FACULTY OF GRADUATE STUDIES
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE
DEGREE OF MASTER OF SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

CALGARY, ALBERTA

August, 2007

THE UNIVERSITY OF CALGARY

FACULTY OF GRADUATE STUDIES

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies for acceptance, a thesis entitled "Cryptanalysis Using Nature-Inspired Optimization Algorithms" submitted by Karel P. Bergmann in partial fulfillment of the requirements for the degree of MASTER OF SCIENCE.

Supervisor, Dr. Renate Scheidler
Department of Computer Science

Supervisor, Dr. Christian Jacob
Department of Computer Science

Dr. Michael Jacobson
Department of Computer Science

Dr. Norbert Sauer
Department of Mathematics and Statistics

Date

Abstract

The research presented in this document applies genetic algorithms and particle swarm optimization to the cryptanalysis of a number of cryptosystems. The two nature-inspired optimization algorithms are used in known-text attacks against cryptosystems ranging from simple classical ciphers to modern cryptographic standards. This research outlines the successful application of genetic algorithms to a variety of classical cryptosystems and provides theories as to why these techniques are not effective against modern cryptosystems. A number of mutation operators are implemented and their cryptanalytic effectiveness compared. The security of classical ciphers against genetic algorithms is communicated in terms of key length. Metrics used to gauge effectiveness include the number of decryptions necessary, making it possible to compare genetic algorithms to exhaustive search.

Acknowledgments

I would like to acknowledge a number of people and organizations for making this research possible. Firstly, I want to thank my wife Ania, my parents and my brother for their support and love. I would like to thank my supervisors, Renate Scheidler and Christian Jacob for their input, support and the time they spent reading drafts of this document. I would like to thank my lab mates, Mark Velichka, Jonathan Hammell, Ryan Vogt and Taya Krivoruchko for their help when it was needed and for making the last two years an enjoyable two years to spend in school. Finally I would like to thank iCore, the Faculty of Graduate Studies and the Department of Computer Science for their financial support, enabling me to carry out this research.

For Ania, for putting up with my eccentricities and supporting me through the course of this research.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Code breakers, or cryptanalysts, have been looking for ways to decipher hidden messages for almost as long as techniques for hiding the meanings of messages (cryptographic techniques) have been in use. In modern scenarios, cryptanalysts practice their profession for a number of reasons; some may be engaged in industrial espionage (generally frowned upon) and try to uncover manufacturing or design secrets of competing corporations. Other cryptanalysts are employed by their respective governments to attempt to decipher sensitive correspondence intercepted from rival nations. Many cryptanalysts attempt to break cryptographic methods for hiding messages (cryptosystems) in order to test their security and ensure that the cryptosystems in use are sound. Regardless of the reason, cryptanalysts generally have a common goal - to discover methods for recovering messages which have been hidden using cryptographic means without the necessary credential (key) which would normally allow a recipient to do so. Presumably, a goal of a cryptanalyst would be to discover a general-purpose algorithm which could be used to decipher hidden messages regardless of the cryptosystem used, since most cryptanalytic techniques are specific to a single cryptosystem.

A genetic algorithm is a search algorithm which uses the principle of biological evolution to evolve better and better solutions to problems. Genetic algorithms make small changes to good solutions in the hope that slightly altering a good solution will result in a better solution. If one approaches the discipline of cryptanalysis as a

search problem – the search for the correct key to decipher a hidden message then a genetic algorithm could be used to solve this search problem. Genetic algorithms provide a flexibility which may have the potential to make them the general-purpose cryptanalytic algorithm alluded to earlier.

The research examined in this thesis explores the cryptanalytic powers of genetic algorithms by applying these algorithms to a variety of classical and modern cryptosystems. As will be seen in the following pages, genetic algorithms are surprisingly flexible and in fact do provide a general-purpose cryptanalytic algorithm for certain types of cryptosystems. It turns out that genetic algorithms are an efficient method of attack for all of the classical cryptosystems examined in this research, but fall short of being successful at subverting modern cryptosystems which are in use today.

The remainder of this document will be presented as follows: A long chapter will thoroughly introduce the reader to the disciplines of cryptography and genetic algorithms, as well as detail how a genetic algorithm could be applied to subverting a cryptosystem. Next, a chapter will discuss the work previously done in this area. The bulk of this document describes the successes and failures of applying genetic algorithms and other nature-inspired optimization algorithms to a number of cryptosystems ranging from the most basic, classical cryptosystems to some of the most advanced, modern cryptosystems. A final chapter will address future work which could be done to explore the cryptanalytic power of genetic algorithms more thoroughly than could be done in the course of this research.

Chapter 2

Background

This work requires a fair amount of background information to sufficiently understand the subject areas that it integrates. The main questions this project addresses lie in the subject area of cryptology, more specifically cryptanalysis. The methods by which this research approaches these questions are based in artificial intelligence and are examples of nature-inspired optimization techniques. Both genetic algorithms and particle swarm optimization are techniques which mimic phenomena seen in nature in order to arrive at a solution.

These subject areas appear disjoint and unrelated; however, if applied properly, nature-inspired optimization techniques can be useful for cryptanalysis.

2.1 Cryptology

Modern cryptology has evolved to encompass a number of different sub-fields, but all of these can be seen as trying to ensure the *security* of some information which needs to be transmitted across a channel. All of the sub-fields which are now considered part of cryptology are in some way inspired by the notion of a classical *cryptosystem*. A classical cryptosystem is a pair of functions which specify how a piece of secret information should be transformed in order that nobody but the intended recipient is able to recover the secret information. Such a classical, or *symmetric* cryptosystem requires that both the sender and receiver of the secret message have on a previous

occasion shared a secret key which is used to scramble and unscramble the message. The algorithms which specify symmetric cryptosystems are quite efficient, but their weakness is this need for a shared secret. These algorithms are widely used today, but usually in conjunction with a *public-key* cryptosystem [19].

A public key cryptosystem addresses the weakness of the symmetric cryptosystem in that there is no requirement for a previously shared secret; instead, a publicly known key is used by the sender to scramble a message and a secret, or *private*, key is needed by the recipient to unscramble the message [27]. This private key is the only required piece of information which must stay secret, but it doesn't need to be shared with any other party. While this seems to have resolved the only weakness of the symmetric cryptosystem, public-key systems do not render symmetric systems obsolete because where symmetric cryptosystems are quite efficient, their public-key counterparts are not. The compromise which is generally struck is that communicating parties use a public-key cryptosystem to exchange a secret key for use with a symmetric cryptosystem. Once the key has been shared, the parties revert to using the symmetric cryptosystem for reasons of efficiency.

Modern cryptology also concerns itself with *authentication codes*. These codes are not necessarily used to conceal a secret message, but rather to ensure that a scrambled message hasn't been modified in transit. When used in conjunction with a cryptosystem, authentication codes allow communicating parties to communicate without fear that a third party is modifying scrambled messages in transit. The use of a cryptosystem to scramble messages also ensures that a third party cannot intercept and read messages as they are being sent.

Cryptanalysis is considered a major part of cryptology. Cryptanalysis, simply

put is the science of code breaking. It is an adversarial pursuit, concentrating on ways to break cryptosystems in order to recover secret information. Other fields can also be lumped into what is now commonly called cryptology, but these are the most essential. The research we discuss here is concerned only with the security of symmetric cryptosystems, and a number of these will be discussed in detail. The rest of the background information provided will be tailored to symmetric cryptosystems, cryptanalytic techniques used to break them and genetic algorithms. Throughout the rest of the background information, a running example of what is commonly referred to as the *Vigenère cipher* will be used to clarify concepts and applications.

2.1.1 Cryptography

There are many cryptographic algorithms in existence, and seven such algorithms were experimented with as part of this research. This section will discuss the general concept of cryptography. Individual cryptographic algorithms or cryptosystems will be described in detail in their respective chapters.

The purpose of a cryptosystem is to transform a message in such a way that it can be transmitted over an open channel (where it is vulnerable to interception by an adversary) without the risk of an adversary being able to read or recover the original message. The cryptosystems being examined through this research are symmetric-key cryptosystems, meaning that the same key is used to both encrypt and decrypt a message. A symmetric cryptosystem is generally viewed as acting on three spaces. The first is M, the message space, which is a set of all possible messages which can be scrambled using a cryptosystem. The second space, C, is the ciphertext space. C is the set of all possible scrambled messages, and in most cases $M = C$. Finally, K,

is the key space. This is the set of all possible keys which can be used with a specific cryptosystem.

In the case of the Vigenère cipher, M is the set of all possible messages made up of one or more letters from the alphabet, so $M = [a - z]^*$. The way the Vigenère cipher works ensures that any message encrypted with any key will also yield a string of letters from the alphabet, so $C = M$. Finally, the Vigenère cipher uses a key word as a key, which is again a string of letters from the alphabet, so $K = C = M$. There is no requirement as to the length of a key $k \in K$; however, shorter keys are less secure (a Vigenère encipherment with a key length of 1 is equivalent to the Caesar cipher which is known to be very weak, as will be shown in a later chapter), while longer keys are less practical because they are hard to remember.

Cryptosystems can be formalized as the pair of functions seen in Equations 2.2 and 2.4, acting on the spaces M, C and K as seen in Equations 2.1 and 2.3.

$$E : K \times M \mapsto C \tag{2.1}$$

$$E(k, m) = c \tag{2.2}$$

$$D : K \times C \mapsto M \tag{2.3}$$

$$D(k, c) = m \tag{2.4}$$

Equation 2.2 describes the encryption function E, which given a key k from the set of all possible keys K, and a message m from the set of all possible messages M, will produce a ciphertext c, from the set of all possible ciphertexts C.

Equation 2.4 describes the decryption function D. Given a ciphertext c, and the correct key k, this function will reproduce the message m which was used to create

c via the encryption function. For fixed k, the decryption function can be viewed as the left inverse of E.

The way in which these two functions are used to accomplish the goals of cryptography is as follows. Suppose that two parties, Alice and Bob, need to conduct a secure communication. Suppose Alice wants to send a message m to Bob, and that Alice and Bob share a secret key k. Alice will compute $E(k, m)$ to produce a ciphertext c, and send c to Bob. When Bob receives c, he will compute $D(k, c)$, which will result in Alice's original message m. Assuming the cryptosystem which dictated the specific functions E and D was a secure one, an adversary, Eve, who intercepts the ciphertext c on its way from Alice to Bob will not be able to recover m without knowledge of k [29].

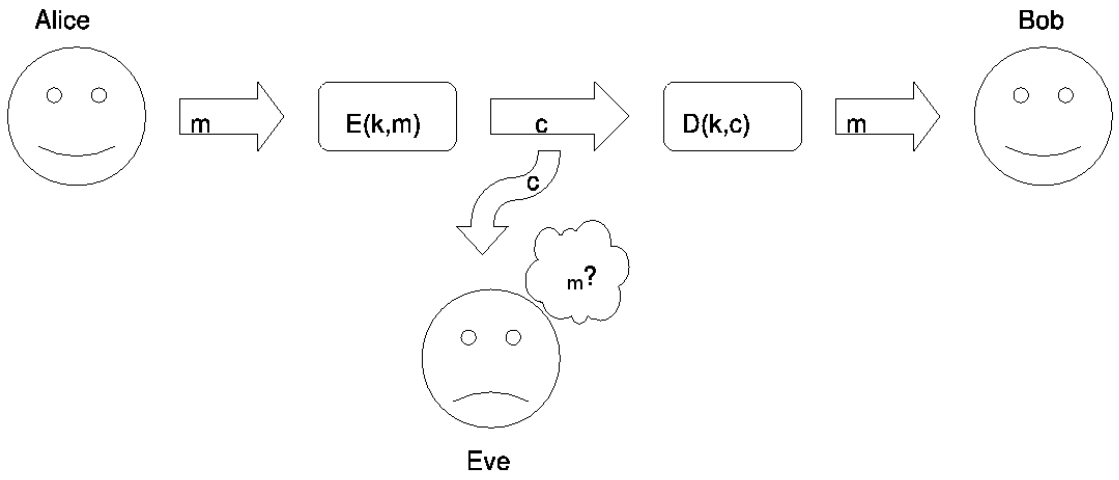

Figure 2.1: Use of cryptography to communicate securely

In the case of the Vigenère cipher, the encryption and decryption functions are

Table 2.1: Mapping letters to integers and back.

a	b	c	d	e	f	g	h	i	j	k	l	m
0	1	2	3	4	5	6	7	8	9	10	11	12
n	o	p	q	r	s	t	u	v	w	x	y	z
13	14	15	16	17	18	19	20	21	22	23	24	25

Table 2.2: Vigenère Cipher Encryption Example.

message	e	x	a	m	p	l	e	m	e	s	s	a	g	e
as integers	4	23	0	12	15	11	4	12	4	18	18	0	6	4
key	t	e	s	t	t	e	s	t	t	e	s	t	t	e
key (as integers)	19	4	18	19	19	4	18	19	19	4	18	19	19	4
addition	23	1	18	5	8	15	22	5	23	22	10	19	25	8
ciphertext	x	b	s	f	i	p	w	f	x	w	k	t	z	i

shown in Equations 2.5 and 2.6 respectively. Each letter of the message is encrypted by *adding* the *corresponding* key letter, modulo 26. The addition is accomplished by first translating each letter in the message and key to an integer from 0 to 25 as shown in Table 2.1, and then adding modulo 26. The resulting integer is then translated back to a letter, again by Table 2.1. Determining which key letter corresponds to which message letter is done by writing the key repeatedly below the message. The addition is performed on the key letters and message letters which are vertically aligned. An example of this process is given in Table 2.2. Decryption is accomplished in the same manner except that instead of adding modulo 26, one subtracts key letters from ciphertext letters modulo 26 [19].

$$E(k, m_i) = m_i + k_{i \pmod{length(k)}} \pmod{26} \tag{2.5}$$

$$D(k, c_i) = c_i - k_{i \pmod{length(k)}} \pmod{26} \tag{2.6}$$

2.1.2 Cryptanalysis

Cryptanalysis, colloquially referred to as code-breaking, has the opposite goal of cryptography. Cryptanalysts attempt to find vulnerabilities in cryptosystems which could be exploited by an adversary in order to recover a message from a ciphertext without knowledge of the key. One of the main precepts of cryptography is that the only secret which should need to be shared between two communicating parties is a secret key. It is assumed that the encryption and decryption functions, E and D, are publicly known, and thus known to the cryptanalyst. One method for uncovering a message given the corresponding ciphertext is to attempt to find the key which was used to encrypt the message. This method has an added advantage for the cryptanalyst. If the cryptanalyst is able to uncover a key which was used to encrypt a message, they can use this key to not only decrypt the ciphertext in question, but also to decrypt any other ciphertext they intercept which is encrypted with the same key. This technique of recovering a secret message by discovering the key which was used to encrypt it is used throughout this research. A genetic algorithm is used to recover a message given a ciphertext by way of discovering a key which decrypts the ciphertext to an English message.

Cryptanalysis is most commonly practiced in two arenas. Cryptographers employ cryptanalytic techniques during the creation of new cryptographic algorithms. Adopting the role of an adversary is useful in ensuring that cryptosystems which are distributed for widespread use are in fact secure. If cryptanalytic techniques uncover vulnerabilities in a cryptosystem, then the cryptosystem would require modifications which eliminate these vulnerabilities before it should be used. The second

major use of cryptanalysis is for espionage. This application of cryptanalysis can take many forms, from warring nations attempting to decrypt each others' military communications, to rival industries attempting to uncover each others' fabrication techniques (generally frowned upon), to jealous lovers reading each others' private correspondence.

As with cryptography, it is possible to formalize cryptanalysis as a function S, as seen in Equation 2.7.

$$S(c) = m, \tag{2.7}$$

where S is a "shortcut function" which attempts to recover a message m whose corresponding ciphertext c was intercepted in transit, without knowledge of the secret key k which was used to produce c. Finding a function such as S would be viewed as having successfully cryptanalyzed a cryptosystem.

The application of this function would appear as follows. Suppose that as before, Alice wants to send a secret message to Bob. Alice computes $E(k, m)$, and sends the resulting ciphertext c, to Bob. This time an adversary, Eve, intercepts c in transit to Bob and applies the short-cut function: computing $S(c)$ which yields Alice's secret message m.

ϕ-Statistic

The first cryptanalytic technique used in this research is based on the ϕ-statistic. The ϕ-statistic allows us to compare the *redundancy* (in terms of letter frequency) of a piece of text to expected values for random text and normal English text. Random text has lower redundancy than English text [9].

The ϕ-statistic is calculated by first counting the frequency with which each letter

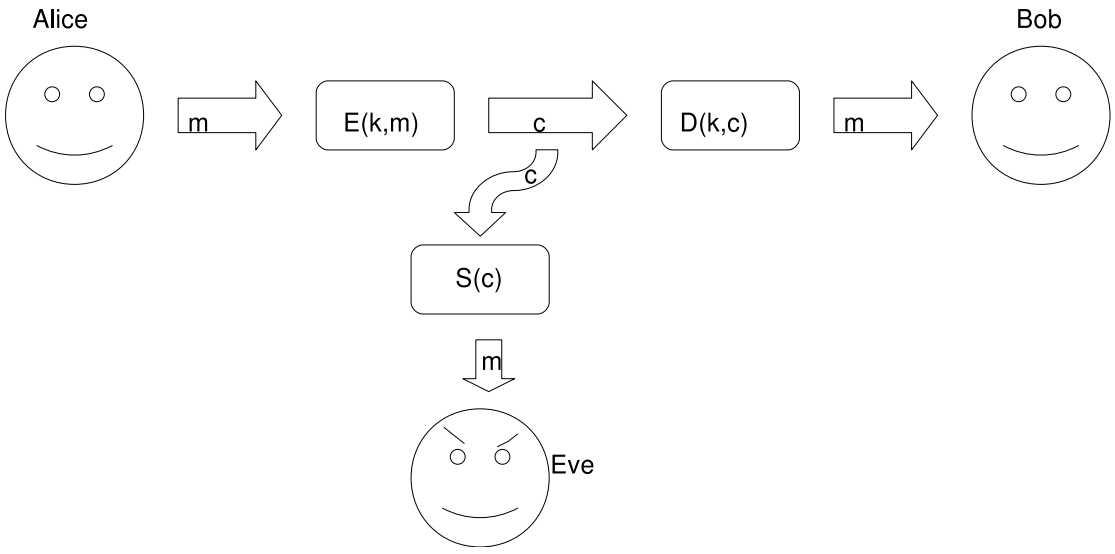

Figure 2.2: Use of cryptanalysis to intercept a message

in the alphabet appears in a piece of text (p_i for the i^{th} letter of the alphabet). Once all of the p_i have been calculated for the text, we use Equation 2.8 to calculate the ϕ-statistic for the text, where n is the number of letters in the alphabet.

$$\phi = \sum_{i=0}^{n} p_i(p_i - 1) \tag{2.8}$$

This number alone is not useful unless we have something to compare it to. The expected ϕ-statistic for random text is calculated using Equation 2.9 and the expected ϕ-statistic for English text is calculated using Equation 2.10, where N is the number of letters in the ciphertext [9].

$$E_{\phi_{rand}} = 0.0385N(N-1) \tag{2.9}$$

$$E_{\phi_{engl}} = 0.0668N(N-1) \tag{2.10}$$

The ϕ-statistic is very useful for the cryptanalysis of the Vigenère and related ciphers.

The first step in cryptanalyzing the Vigenère cipher is to determine the length of the key which was used to produce a ciphertext. This can be done through a guess-and-check approach. A guess is made at the key length, and then verified using the ϕ-statistic. Once a guess at a key length has been made (n), then the ciphertext is split into n different *subtexts*. The first subtext includes the first ciphertext letter and every n^{th} subsequent letter. The second subtext includes the second ciphertext letter and every n^{th} subsequent letter and so on for each of the subtexts. The intuition here is that each of the subtexts has been encrypted with the same key letter. If the message is in English and the guess of n is correct, or a multiple of the actual key length, then each of the subtexts will have near English-like redundancy, as calculated using Equation 2.10. If the guess is incorrect, then the subtexts will have ϕ-values distant from English-like and probably closer to that of random text, as calculated using Equation 2.9.

For example, consider the ciphertext "lwiumkpngbsbeuwsvehsjpncpqhiuykxnixzt-pjmkiavmkiie". If a cryptanalyst were given this ciphertext they might guess 3 as the key length, and subsequently divide the ciphertext into three subtexts $s_1 = \{l, u, p, b, e, s, h, p, p, i, k, i, t, m, a, k, e\}$, $s_2 = \{w, m, n, s, u, v, s, n, q, u, x, x, p, k, v, i\}$ and $s_3 = \{i, k, g, b, w, e, j, c, h, y, n, z, j, i, m, i\}$. To calculate the ϕ-value for s_1, we only need to identify which letters occur two or more times in the subtext. The letter p occurs 3 times, e occurs 2 times and i occurs 2 times, so $\phi_{s_1} = 3{\times}2{+}2{\times}1{+}2{\times}1 = 10$. Similarly, $\phi_{s_2} = 10$ and $\phi_{s_3} = 8$. Since s_1 consists of 17 letters, $\phi_{s_{1_{Eng}}} = 0.0668{\times}17 = 18.17$ and $\phi_{s_{1_{Rand}}} = 0.0385 \times 17 = 10.47$. Similarly $\phi_{s_{2_{Eng}}} = \phi_{s_{3_{Eng}}} = 16.03$, and $\phi_{s_{2_{Rand}}} = \phi_{s_{3_{Rand}}} = 9.24$. As can be seen, the actual ϕ-values for the subtexts are much closer to those one would expect for random text, and not to those one would

Table 2.3: Analysis Using Five Subtexts

Subtext	Length	ϕ_s	$\phi_{s_{Eng}}$	$\phi_{s_{Rand}}$
1	10	14	6.012	3.465
2	10	4	6.012	3.465
3	10	14	6.012	3.465
4	10	6	6.012	3.465
5	9	6	4.8096	2.772

expect for English text. So the guess for the key length is likely wrong.

This is good evidence that the original guess of there being three letters in the key word is wrong. If the cryptanalyst tries again, this time with a guess of a key length of five, the situation is different. Now we have different ϕ-values for the five subtexts. The results for this analysis are given in Table 2.3. As can be seen, the letters in the five subtexts are in most cases more redundant that what one would expect in English, but this is still an indication that the key length was guessed correctly. Now that the key length has been determined, the cryptanalyst can dedicate himself to finding the key word itself. The ϕ-statistic can be used in the cryptanalysis of many ciphers, but the rest of the process for this ciphertext is specific to the Vigenère cipher. A complete Vigenère cryptanalysis can be seen in Section 5.2.

The Vigenère cipher is what is referred to as a *polyalphabetic* substitution cipher because multiple cipher alphabets are used during encryption (one for each letter of the key). If one were to use a one-letter key with the Vigenère cipher (what is typically referred to as the Caesar Cipher), then it would be considered a *monoalphabetic* substitution cipher as only one cipher alphabet would be used. The ϕ-statistic has another major use in cryptography in that it can be used to differentiate ciphertexts which are monoalphabetically enciphered from those which are polyalphabetically

enciphered. Monoalphabetically encrypted ciphertexts have ϕ-values similar to that of English which polyalphabetically encrypted ciphertexts have ϕ-values closer to that of random text.

Hamming Distance

A second cryptanalytic technique which could be used is the popular *Hamming Distance* construct [12]. If the cryptanalyst has decided on a potential key and happens to know that a certain piece of text appears in a certain location in the message they are cryptanalyzing, then the cryptanalyst can decrypt the ciphertext and use Hamming distance to check how close the resulting message is to containing the known piece of text.

The cryptanalyst provides a piece of known plaintext (of length n) which they know should appear in the message and the starting position of the text segment in the message. The ciphertext is decrypted with the potential key and a counter i is initialized to zero. Beginning at the first position where the known text should appear, i is incremented every time the character in the known text corresponds to the character in the decrypted message at the expected location. When there are no more characters in the expected text segment, the algorithm terminates. Thus, the higher the Hamming distance value, the closer the message segment is to the known text, to a maximum of n. If the value n is attained, this would mean that the known text appears in its entirety where it is expected in the decrypted ciphertext. This would be a good indication that the key which was used to decrypt the ciphertext was the correct key. Lower values may indicate that the key is partially correct, depending on the cryptosystem being used. An example is provided in Figure

2.3. Given the known text in the example and the expected starting position, the Hamming Distance score of Candidate 1 is 1, because only one letter in that text matches with what it should be if the word "general" occurred in the expected location. All seven letters correspond to the expected text at the expected location when considering Candidate 2, thus the Hamming Distance score is 7. While this

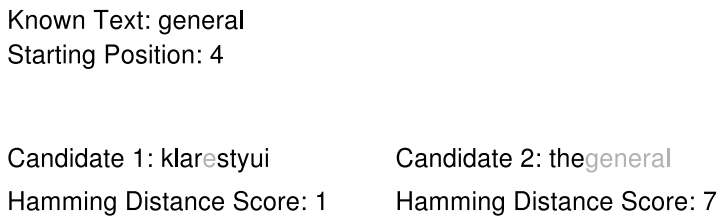

Figure 2.3: Hamming Distance scores of two different texts

type of attack requires the cryptanalyst to know something about the message they are decrypting, it is justifiable. Such known text segments have been extensively used in cryptanalysis since the Enigma cipher of WWII and earlier. In the case of the Enigma cipher, these known plaintext segments were called cribs and were used extensively by cryptanalysts to decipher messages [24]. If a cryptanalyst knew that a communication was destined for a general, they might assume that the first word would be "general", simplifying the cryptanalytic task to identifying which key

settings would decipher the first seven letters of the ciphertext to the word “general”. A similar approach is used in this research.

2.2 Genetic Algorithms

The presentation of genetic algorithms will include generic information on genetic algorithms as well as specific information about modifications that were made to make them more usable for the cryptanalysis of character-string based cryptosystems, and will be accompanied by examples relevant to cryptanalysis of the Vigenère cipher. As previously mentioned, a genetic algorithm (GA) is a nature-inspired optimization technique. We say nature-inspired because the functioning of a GA is meant to mirror that of biological evolution. A brief discussion of this topic will be provided in order to make clear how biological evolution is mirrored by GA’s. Genetic algorithms were first described by Holland in 1975 [13]. Further information on the subject can be found in Goldberg’s 1989 book [11].

2.2.1 Biological Evolution

For evolution to occur, it is necessary to have a population of organisms which reproduce. Organisms have an internal representation, called a *genotype* which is made up of all of the base-pair sequences in their deoxyribonucleic acid, DNA. For the purposes of this discussion, we will say that an organism’s genotype encodes all of the information necessary for the organism to grow and function. The genotype of an organism goes through an interpretation phase as proteins and organelles within the organism’s cells scan the DNA and execute the instructions found on it. This

interpretation phase is critical because it translates the raw information found on the genotype to the externally visible state of the organism, or *phenotype*. The importance of this interpretation phase lies greatly in the fact that the magnitude of a change in the genotype of an organism doesn't necessarily correspond to a change in the organism's phenotype of like magnitude. Small changes in genotype could result in very dramatic changes in phenotype, depending on the specifics of the process which translates from genotype to phenotype.

Different genotypes usually result in different phenotypes, and the phenotype of an organism is what determines how it will fair in its environment, or the organism's *fitness*. The survival of an organism is contingent on having a phenotype which allows it to cope with difficulties it encounters in its environment, or *selective pressures*. Populations of organisms tend towards phenotypes which are of higher fitness over time, because organisms which have high-fitness phenotypes will survive long enough to reproduce, while low-fitness individuals will either not survive long enough to reproduce, or will be out-competed by organisms of higher fitness.

When organisms reproduce, it is important that they do not create perfect replicas of themselves, otherwise the population would be stagnant. It is necessary for possible new changes to arise within a population, otherwise the population would not be able to adapt to a changing environment. During reproduction the genotype of an organism is duplicated so that the offspring of that individual may have a copy of the genotype as well. This duplication process is seldom achieved with complete accuracy. Small mistakes are made during the course of this duplication, and these mistakes are what are referred to as *mutations*. The frequency with which these mutations arise is known as the *mutation rate*.

In addition to simple mutations, organisms which reproduce sexually have one more method for introducing variety into their offspring. When two organisms reproduce sexually, both parties contribute genetic material to the genotype of the offspring. This recombination of genetic material is why offspring tend to have some traits exhibited by either parent. In sexually reproducing species this recombination of DNA occurs through a process called crossover. The genotype of each organism is contained in a number of *chromosomes*. Each chromosome contains unique information that is different from the other chromosomes possessed by an organism. Each organism requires a full set of chromosomes in order to live and grow properly, but the number of chromosomes present is different in different species of organism. During sexual reproduction each parent contributes a set of chromosomes, so at this stage, there is a duplicate copy of each chromosome in the set, one copy contributed by each parent. While the offspring now has two copies of each chromosome, these matching chromosomes aren't necessarily exactly the same. As a simplified example, consider the gene for hair colour in humans; this gene is located on the same chromosome in all people. The information encoded in this gene is necessarily not the same for all people, because some people have black hair and some people have blond hair. Thus, while at this stage of development an offspring will have two versions of each chromosome, these two versions may have slightly different information encoded within them. Since the offspring only requires a single set of chromosomes, the number present is halved as each chromosome from one parent aligns with the matching chromosome from the other parent. Once aligned, matching chromosomes intermingle and sections from one version are replaced by the corresponding section from the other version. This results in half of the genetic material present being

recombined into new chromosomes for the offspring, the other half of the genetic material is discarded.

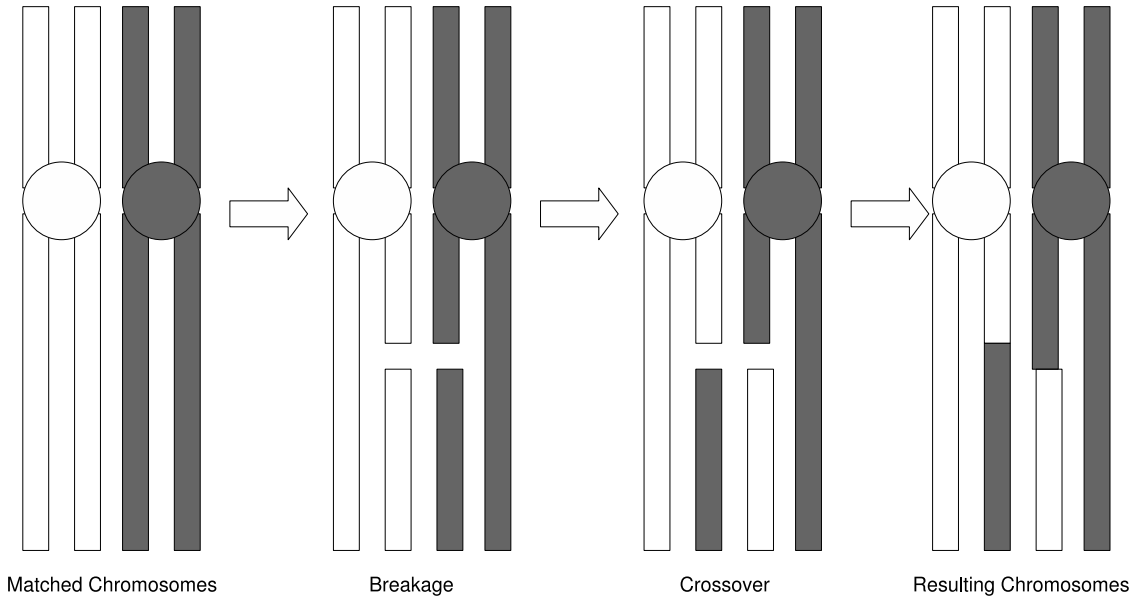

Figure 2.4: Chromosomes during crossover

2.2.2 General Function

The above model of how a population of organisms tends towards a population of higher fitness through natural selection and sexual reproduction is closely mirrored by a GA. GA's are best applied to problems with large solution spaces for which an efficient, tailored algorithm doesn't exist or is prohibitively expensive in terms of computing time. A GA is not guaranteed to find a solution to a problem, but can potentially be configured to find a *sufficiently good* solution where a perfect solution is not required.

A genetic algorithm employs a pool of solutions, or population, which evolves

over time. As the population evolves, it is the hope that the fitness of the solutions which make up the population increase to the point that an optimal or near-optimal solution is found. Figure 2.5 shows a schematic of the general operation of a genetic algorithm, as will be described in the following sections.

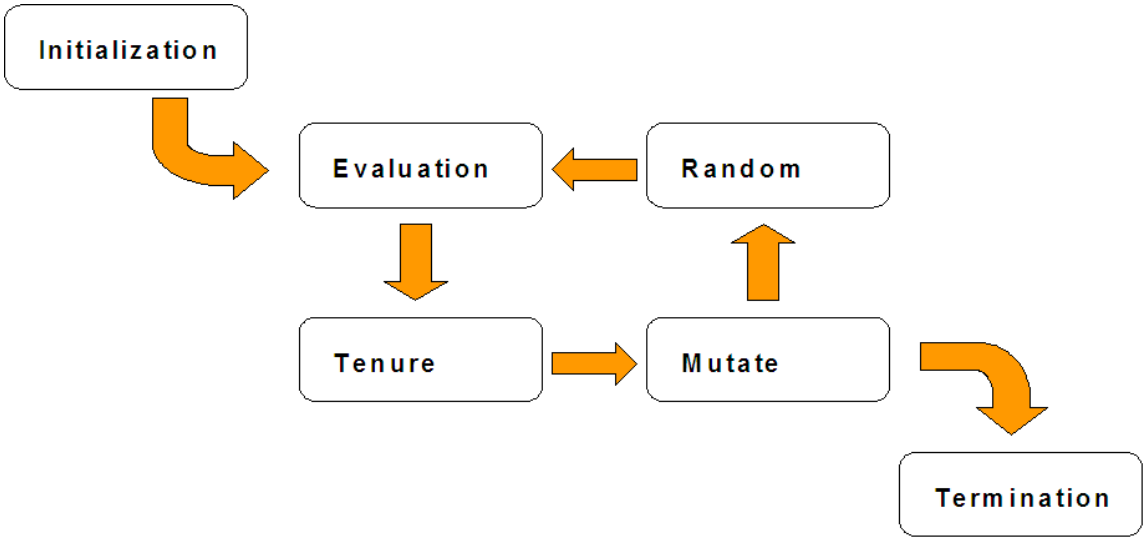

Figure 2.5: Genetic Algorithm Schematic

Initialization

To begin, a random population of individuals is selected. Each individual should represent a potential solution to the problem one is trying to solve. In our context the purpose of using a genetic algorithm is to allow us to decrypt an intercepted ciphertext in order to recover the message, so each individual in the population takes the form of a decryption key.

One of the parameters for the genetic algorithm is the population size ($pSize$), another is maximum length of the keys we want to examine ($maxKeyLength$), where a key can be a string of characters or bits depending on the specification of the cryp-

tosystem being used. To initialize the GA, *pSize* random character strings of lengths between 0 and *maxKeyLength* are selected. These keys represent the genotypes of the individuals in the population.

If one were using the GA to cryptanalyze a ciphertext encrypted with the Vigenère cipher, the cryptanalyst would need to decide on a population size and a maximum key length. If the cryptanalyst is sure the key isn't longer than 10 characters, this would be a good value to use for *maxKeyLength*; the standard population size used in many experiments was 20 individuals. To initialize the GA, 20 random Vigenère keys would be selected, each of random length, but no longer than 10 characters. Thus, "bhy", "abcdefghi" and "zzzzzzzz" would all be possible candidates for seeding the population, while "abcdefghijklmnopqrstuvwxyz" would not because it is too long.

Evaluation

Each of the individuals in the population represents a key which can be used to decrypt the ciphertext that is being cryptanalyzed. Some of these keys are better than other keys in the population and it is important that we have a way of determining which of these candidate keys are good and which ones are bad. This evaluation mechanism is known as a *fitness function*. A fitness function can be any function F which maps keys to real values between 0 and 1, as shown in Equation 2.11. High-fitness individuals should have fitness close to 1, while low-fitness individuals should have a fitness value closer to 0. While the actual key represented by each individual represents its genotype, this fitness value represents its phenotype. The fitness function F is the translation mechanism discussed previously which translates genotype to phenotype. The specific fitness function used in this research will be discussed in

Section 2.2.3. The fitness function is used to calculate the fitness of every individual in the population.

$$F\colon K \mapsto [0,1] \tag{2.11}$$

Tenuring

Once the fitness values for all the individuals in the population have been calculated it is time to decide which individuals will participate in the next generation. The genetic algorithm implementation created for this research allows the user to specify *numTenured*, or the number of individuals which will be transplanted, or allowed to "live on" into the next generation. Clearly, this number should be smaller than *pSize*. The *numTenured* individuals of highest fitness are passed on to seed the next generation of solutions.

In our running example of the Vigenère cipher, we decided on a population size of 20 and a maximum key length of 10. Each of the twenty keys in the population would be evaluated using the fitness function and associated with a fitness value in the interval $[0, 1]$. The cryptanalyst must decide on a value for the *numTenured* parameter. In many experiments the value for this parameter is 5. Since all 20 of the keys in the current generation have been evaluated using the fitness function, they can be sorted according to fitness, and the top *numTenured* keys are tenured, or moved into the next generation. The process of tenuring individuals into the next generation of the GA is important because it is possible that the offspring of these individuals, created through the application of mutation operators (described in the next section), may be of lower fitness than these five individuals. If this were the case and no tenuring was performed, then the maximum fitness of the population would

be reduced which is undesirable because it is akin to "taking a step backwards" from finding the correct key.

Reproduction

Once the *numTenured* best solutions are tenured into the next generation, they are used to create most of the rest of this next generation of solutions. The first step towards generating a new individual is the selection of a *Mutation Operator*. A mutation operator plays the role of the mutation and cross-over type transformations in a biological system. There is no reason to require a mutation operator to behave in exactly the same way as either of these transformations, so long as it is a function M which could take an arbitrary number of genotypes as input and produces a single new genotype as output, as seen in Equation 2.12; however, most mutation operators take only one or two parent genotypes as input. The specific mutation operators used in this research will be discussed in Section 2.2.4.

$$M\colon K \times K \times \ldots \times K \mapsto K \tag{2.12}$$

Once a mutation operator is selected, the required number of parent individuals is selected from the individuals which were tenured into this generation during the last step. Applying the mutation operator to the genotypes of the selected individuals produces a new genotype which is introduced into the population at this generation as a new individual. This process is repeated until the size of the population is nearly *pSize*; the process of immigration, described below, is used to provide the remainder of the required individuals.

In the case of the running example, suppose the GA was provided with only a single mutation operator which selects a random character position in a key and

changes the letter in that position to a random letter. Since this is the only mutation operator provided, it would be chosen to create all of the additional individuals required for the next generation (in addition to the tenured individuals). Since this example mutation operator is a unary operator (requires only one parent), a single individual from the individuals tenured into the next generation would be selected, the mutation operator would be applied to the selected key and the resulting key (which differs by at most one character from its parent) would be added to the next generation. This process would be repeated until the next generation contains the required number of *pSize* individuals, save for those to be provided through immigration.

Immigration

Immigration is a population characteristic which is seldom modeled in GA's; however, the introduction of immigration dramatically increases the effectiveness of the GA for cryptanalysis. Immigration occurs when individuals from outside the population migrate into the population, potentially introducing fresh genetic material to the population. The reproduction phase finishes by leaving a few *numRandom* open positions in the population. These final open positions are filled at every generation by random solutions which "immigrate" into the population. This behaviour allows a population which is stuck in a local maximum in the *fitness landscape* to resume evolving towards the global maximum if one of these random immigrants happens to have high fitness and a dissimilar genotype. The fitness landscape can be viewed as a region in n-space which represents all possible solutions to the problem being solved by the GA; thus, each location in this region of n-space has a fitness value associated

with it and there will be regions with low relative fitness which we call local minima and regions with high relative fitness which we call local maxima. The location in this region of n-space which has the highest associated fitness value of any location in the space is the global maximum and represents the best possible solution. The goal of a GA is to find this global maximum. In the context of the running example, this n-space would be representative of all Vigenère up to a maximum key length, and n would equal this maximum key length; thus, each possible key can be viewed as an n-tuple being represented by a location in the n-space.

After the immigration stage is complete, an entire new population of individuals has been established (new except for the individuals which were tenured into the generation). The GA implemented during the course of this research enforces the stipulation that all the individuals introduced into the population during the reproduction and immigration phases be distinct, so at every generation the population should contain *pSize* distinct solutions. The algorithm now loops back to the Evaluation phase unless a termination condition is reached.

In the case of the running example, the cryptanalyst may decide on 5 as the value for the *numRandom* parameter. This means that in order to establish a new generation of Vigenère keys, *numTenured* (5) keys are brought in from the previous generation. $pSize - (numTenured + numRandom)$, or 10 new keys, are created by applying the mutation operator to randomly selected, tenured keys. Finally, *numRandom*, the final 5 keys needed to make the new generation of individuals, are created randomly, in the same manner that keys were created during the initialization phase. The fitness landscape which the GA could be viewed as searching can be interpreted as an n-dimensional space, where $n = maxKeyLength$. Under this

interpretation, the i^{th} coordinate of a point in the space would correspond to the identity of the i^{th} letter of a key. Each distinct point in the fitness landscape would then map to a $maxKeyLength$-long string of letters. One of the possible identities for a letter would be the null character. If a null character were to appear in a key, it would indicate the end of that key. This provision of the null character allows us to represent all keys up to $maxKeyLength$ characters in length in the fitness landscape. For instance, consider a genetic algorithm searching for a Vigenère key which is known to be at most 10 letters in length. The space that this GA searches can be viewed as a 10-dimensional space, composed of values from 0 to 26 (inclusive) in each dimension, the integer 0 representing the null character and the integers from 1 to 26 representing the letters of the alphabet. Any n-tuple with values between 0 and 26 in every position would be representative of a Vigenère key 10 letters or less in length. The 10-tuple $(1, 1, 1, 1, 1, 1, 1, 1, 1, 1)$ would represent the key "aaaaaaaaaa", while $(1, 1, null, 1, 1, 1, 1, 1, 1, 1)$ would represent the key "aa" as the first null character signifies the end of the key. Using this representation, keys shorter than $maxKeyLength$ have multiple representations.

Termination

A genetic algorithm terminates under two possible conditions. The first termination condition occurs when a set number ($maxGen$) of generations has been reached. It would be possible to allow a GA to continue execution until a correct solution is found; however, there is no guarantee that this will happen. The user may specify the number of generations they are willing to wait before being provided with a solution. The number of generations which can be executed in a given amount of time is

variable. The base GA operations aren't computationally expensive, and normally mutation operators have quick execution times as well. The main computational cost involved in running a GA tends to be incurred by the fitness function, whose run-time can be quite slow depending on what it is calculating. The fitness function must be calculated *pSize* times per generation.

The second termination condition occurs when a solution of sufficiently high fitness enters the population. The user may specify the fitness value which represents a solution that is satisfactory. When an individual in the population reaches this threshold, the algorithm terminates and the highest-fitness solution is returned.

In the case of the running example, after numerous generations have been created and evaluated, an individual may arise which has extremely high fitness. This means that the ciphertext being cryptanalyzed, when decrypted with this high-fitness key, yields the original message, or the key is mistakenly attributed high fitness by the fitness function. This latter situation would imply that the fitness function used is not a good one because it highly favours a key which is not the correct one.

2.2.3 Fitness functions

The selection of the fitness function is the most critical decision which needs to be made when implementing a genetic algorithm. As seen in Equation 2.11, any function which maps a potential solution (in this case a character string representation of a key) to a real number between 0 and 1 will suffice. In order for the GA to operate well, this fitness function should have certain characteristics. The most obvious of these characteristics is that a solution which is correct should have very high fitness. Solutions which are incorrect should have lower fitness values which occupy a gradient

from low to high fitness depending on how *close* they are to being correct.

The GA implemented for this research allows the user to specify an arbitrary number f of fitness functions $F_1, F_2, \ldots, F_f$, each of which may judge the fitness of a key based on a different characteristic. Equation 2.13 shows how these fitness functions are amalgamated into a single fitness function which is used by the GA. This single fitness function is the average of the fitness values returned by all of the fitness functions provided to the GA.

$$F_{tot}(k) = \frac{1}{f} \sum_{i=1}^{f} F_i(k) \tag{2.13}$$

For the classical ciphers examined during this research, two fitness functions were supplied to the genetic algorithm, resulting in successful cryptanalyses of the ciphertexts which were provided. Neither of these two fitness functions work by evaluating characteristics of an individual in the population (a key) directly, but rather work by using the individual (k) to decrypt the ciphertext (c) being cryptanalyzed and analyze properties of the message that results from computing $D(k, c)$.

ϕ-Statistic

The first fitness function used in this research is based on the ϕ-statistic. This fitness function first decrypts the ciphertext with the candidate key from the GA population. The redundancy of the resulting message is used to generate a fitness value. If the ϕ-value of the message is between 0 and the expected ϕ-statistic for random text, the key that generated it is allotted a fitness of 0. ϕ-values ranging from that of random text to English text are scaled linearly from 0 to 1, where a key which decrypts a ciphertext to a message with redundancy matching that of typical English is given a fitness value of 1. Fitness values are then linearly scaled from 1 back down to 0 as the

ϕ-value of the message increases from that of typical English text to text which has double the ϕ-value of typical English text. Since this fitness function calculates the ϕ-value of the entire message, not of different subtexts as seen earlier, the ϕ-values for English messages are very close to the expected values, especially when the message being evaluated is long. ϕ-values which are very large relative to the expected value for English text do not normally occur when using this technique as they do in the classical cryptanalysis of the Vigenère cipher seen in Section 2.1.2 when analyzing individual subtexts. Figure 2.6 illustrates the scaling ϕ-statistic based fitness scores. Figure 2.7 shows a sample fitness landscape of Caesar Cipher keys (consisting of one lower-case or upper-case letter). In this case, the correct key is "k", but the figure illustrates two maxima, one for the correct key, and one for a key which would decrypt the ciphertext to an English plaintext consisting of only upper-case letters.

Hamming Distance

The other fitness function used in this research is based on a modification of the Hamming Distance and thus amounts to a *known text attack* on the ciphertext.

The cryptanalyst supplies the genetic algorithm with a piece of known plaintext (of length w) they know should appear in the message and the starting position of the text segment in the message. A counter i is initialized to zero, and beginning at the first position where the known text should appear, is incremented every time a character in the known text corresponds to the character in the decrypted message at the expected location. When there are no more characters in the expected text segment, the fitness value of the key which decrypted the ciphertext is calculated as $F_{hd}(k) = i/w$. Figure 2.8 shows a Hamming Distance fitness landscape for the

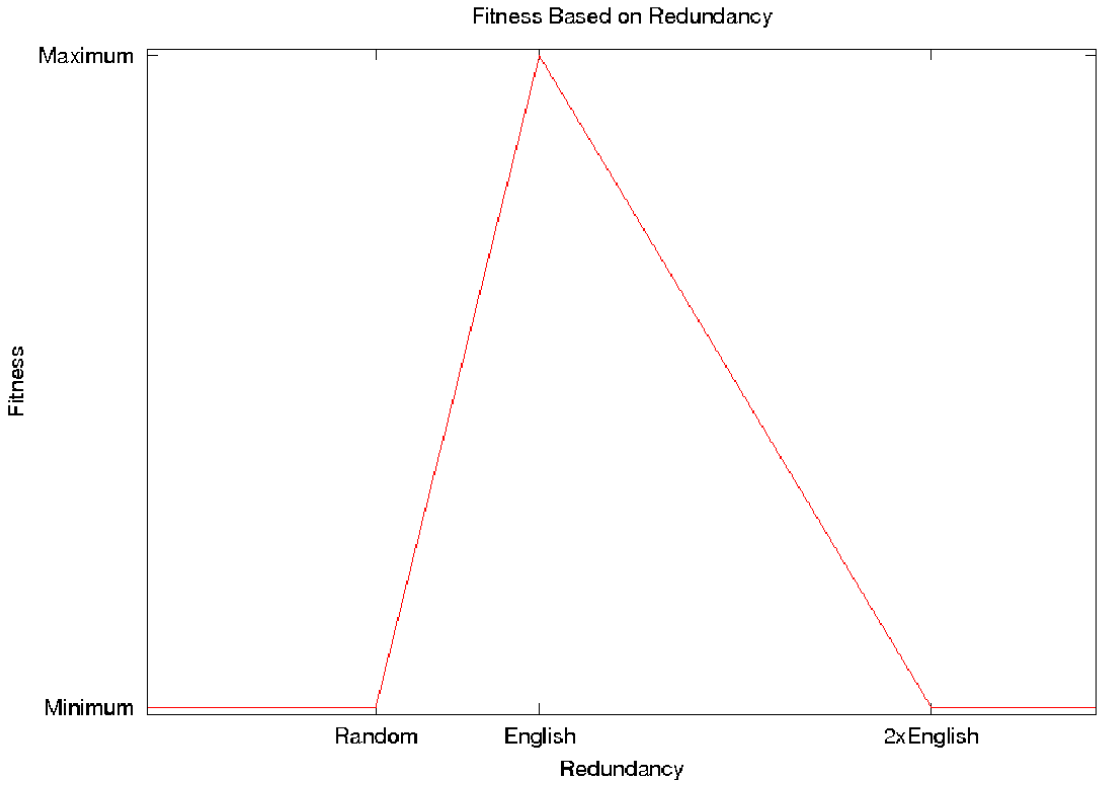

Figure 2.6: ϕ-statistic Fitness Scores

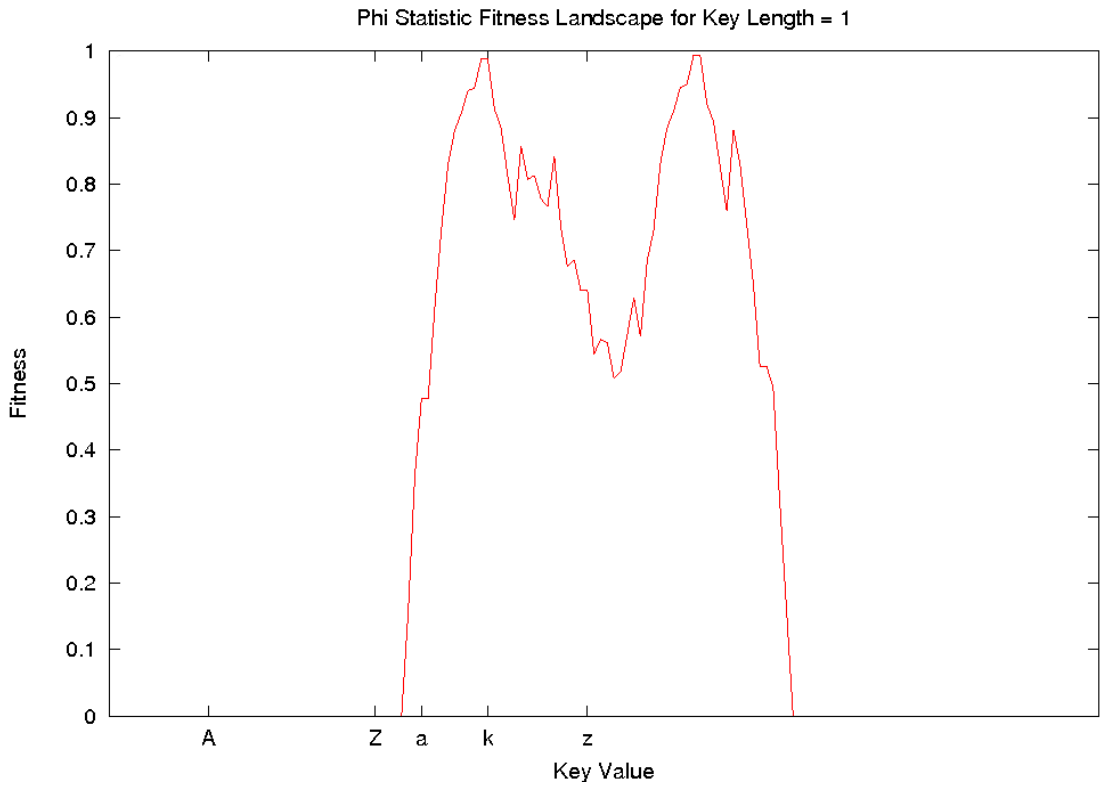

Figure 2.7: ϕ-statistic Fitness Landscape for the Caesar Cipher

Caesar Cipher where the correct key is "k". Although this landscape is not easily optimizable, the fitness function does provide useful information when used with longer keys (such as those for the Vigenère cipher), or when combined with the ϕ-statistic based fitness function (See Figure 2.10). The fitness landscape in Figure 2.8 can be described as not easily optimizable because it only consists of a single spike, with no higher-fitness regions leading up to it.

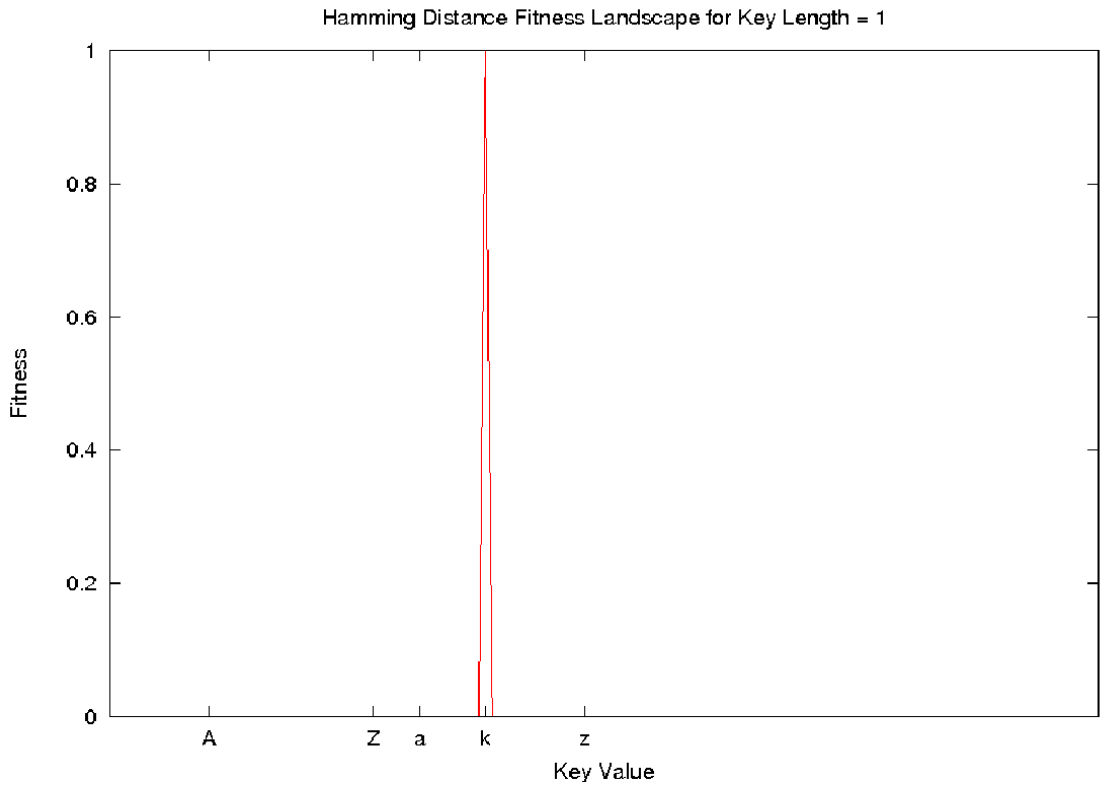

Figure 2.8: Hamming Distance Fitness Landscape for the Caesar Cipher

Combined Fitness Function

In the case of the running example of cryptanalysis of the Vigenère cipher, as was done in this research, both of the above fitness functions will be used to differentiate

between good keys and bad keys. In order to use the Hamming Distance-derived fitness function, the cryptanalyst must provide a piece of text they expect to find in the message and the starting point where it should occur. Assume that the cryptanalyst knows that the word "example" should appear in the message, starting at the 9^{th} character in the message. Recall that the ciphertext being cryptanalyzed was "lwiumkpngbsbeuwsvehsjpncpqhiuykxnixztpjmkiavmkiie". Suppose at some point, through immigration or mutation of tenured keys, the key "space" enters the population. Using the first fitness function, this key will receive a fitness value close to 1, because when the ciphertext is decrypted with it, the resulting message has redundancy very close to that of English. Using the second fitness function, "space" will receive a fitness value of 1 because the word "example" does indeed occur in the decryption of the ciphertext, starting at position 9. When these fitness values are combined using Equation 2.13, the result will again be very close to 1, indicating a very good key. Using the key "space" to decrypt the ciphertext using the Vigenère cipher, we get the message "thisisanexamplemessageforanalysisusingthephistatistic", which is indeed the original message.

Figure 2.10 shows the fitness landscape which results from the combined fitness function applied to the Caesar Cipher. This fitness landscape is both easily optimizable and unambiguous – both characteristics of good fitness functions. When comparing to Figure 2.10 to Figures 2.7 and 2.8 one can see that the fitness landscape in Figure 2.7 is easily optimizable, but ambiguous because there are two global maxima. The fitness landscape in Figure 2.8 is not easily optimizable, but is unambiguous, while the fitness landscape in Figure 2.10 is both unambiguous and easily optimizable. Figure 2.11 shows the fitness landscape which results from the same

fitness function when applied 2-letter Vigenère keys. Although less straight-forward than the fitness landscape seen in Figure 2.10, this fitness landscape shows itself to be easily searchable in Section 5.3.

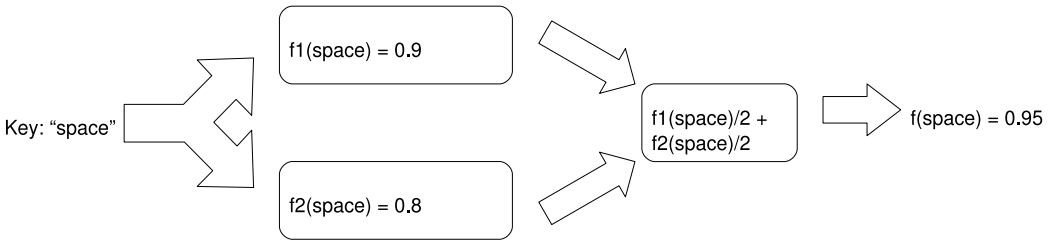

Figure 2.9: Combining fitness functions

2.2.4 Mutation Operators

Mutation operators are an important part of a genetic algorithm because they dictate in which ways genotypes can change from generation to generation. While the fitness function dictates the fitness landscape to be searched by the GA, the mutation operators dictate the search patterns used within the fitness landscape. Many genetic algorithm implementations default to using a combination of mutations and crossovers as their sole mutation operators in order to mirror the evolution of a biological population more closely. This constraint is unnecessary and limits the versatility of the genetic algorithm model. The genetic algorithm implemented as part of this research allows the flexibility of including an arbitrary number of mutation operators with no limitation on their behaviour so long as they fulfill the constraints set forth in Equation 2.12. Four such mutation operators were tested.

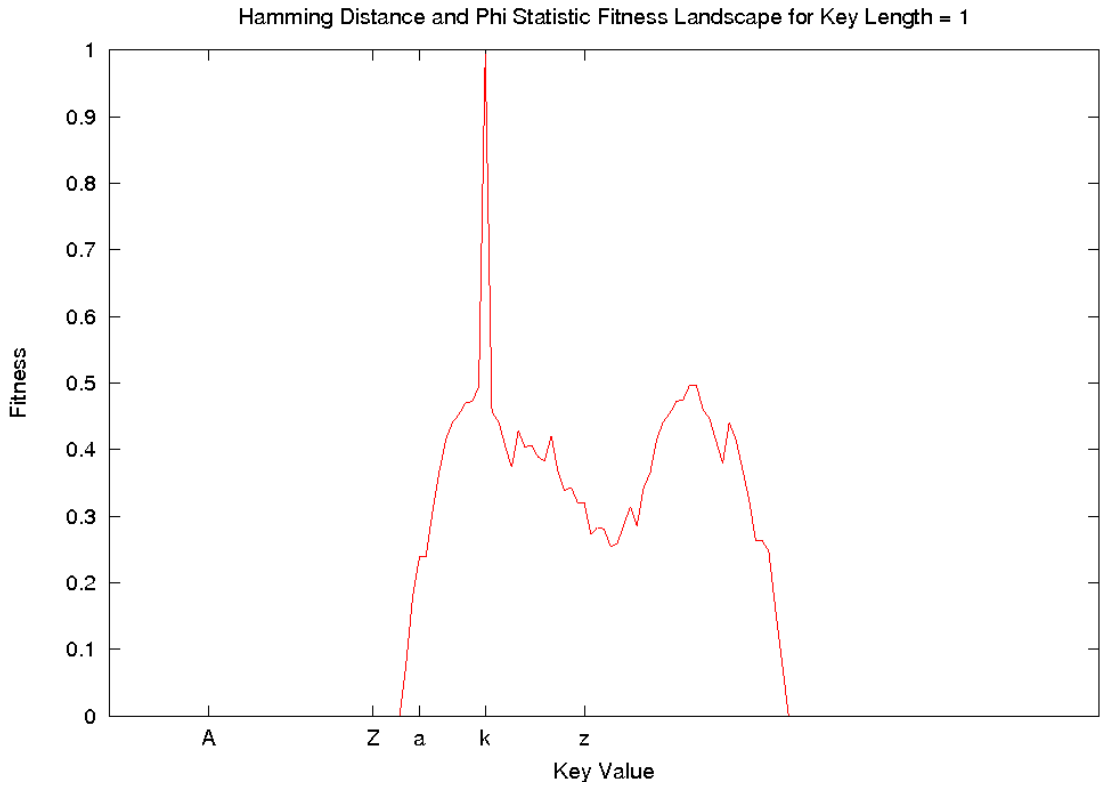

Figure 2.10: Combined Fitness Landscape for the Caesar Cipher

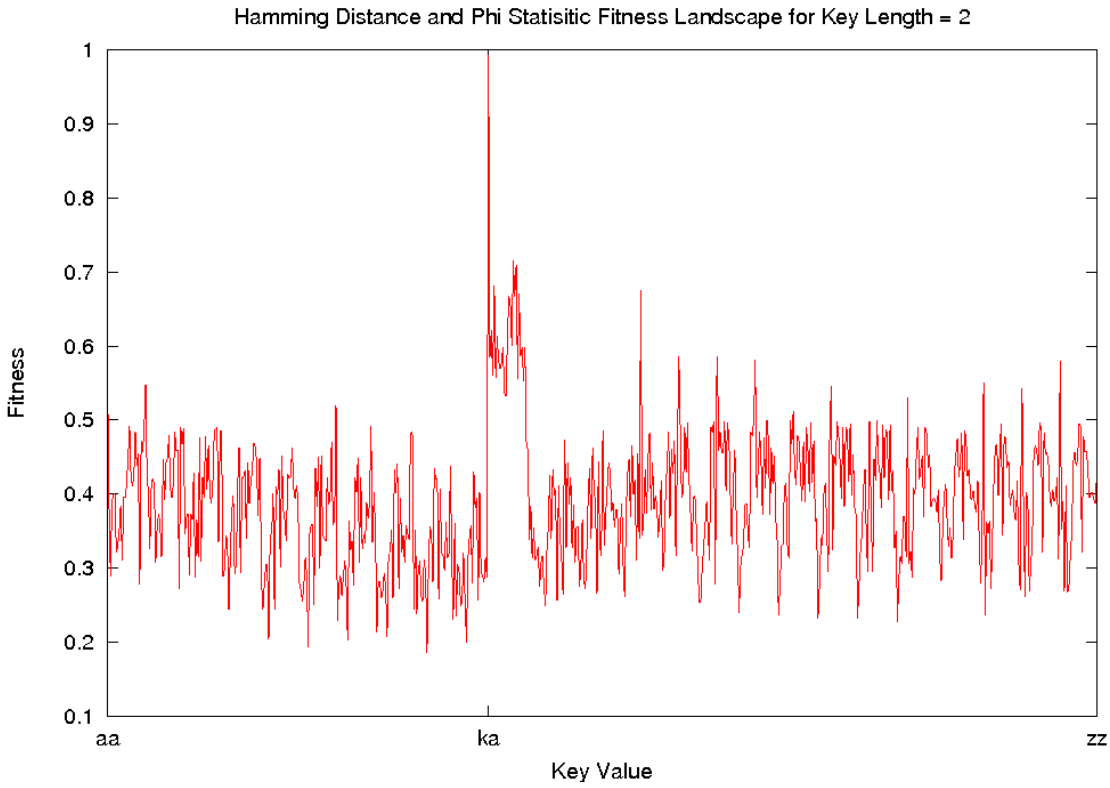

Figure 2.11: Combined Fitness Landscape for 2-letter Keys

The original description of a genetic algorithm [13] includes descriptions of three mutation operators which are used in most GA implementations. The point mutation alters a random location in a solution, the inversion reverses an interval found within a solution and the crossover exchanges a randomly selected interval between two solutions.

Resize (RS)

One mutation operator developed as part of this research is the *resize* operator. This operator changes the length of the key being mutated by one, as it randomly either grows the key by one character or shrinks it by one character. In the first case a random location within the character string is selected, a random character from the alphabet is also selected and inserted into the random location so long as this doesn't violate the condition imposed by the *maxKeyLength* parameter. In the case where the key is to shrink by one character, a random location in the character string is selected as before and removed from the key, resulting in a key with one less character.

For example in the case of Vigenère keys, both "helo" and "hellzo" are possible offspring of the key "hello" when using the resize mutation operator.

Character Flip (CF)

The *Character Flip* mutation operator is the direct analog to the mutations found in biological systems. A random location is selected within the key being mutated. The character at this location is then replaced by another character, randomly chosen from the alphabet.

In the case of Vigenère keys, both "hvllo" and "helle" are possible offspring of

"hello" when using Character Flip, but "hllo" is not because this mutation operator does not resize keys.

Mutate-By-One (MO)

The *Mutate-By-One* mutation operator is similar to the mutations seen in DNA, and is a special case of Character Flip with a constraint as to the extent of the mutation. A random location is selected in the key being mutated. The character found at this location is then modified by replacing it with either the next or the previous character found in the ASCII table. Thus any character in the key can be changed but only to a neighbouring letter in the alphabet.

In the case of Vigenère keys, both "hellp" and "gello" are valid offspring of "hello" when using Mutate-By-One, while "fello" is not.

Crossover (CO)

The *Crossover* mutation operator is again very similar to its analogue found in biological systems. Crossover requires two parent keys as input in order to produce an offspring. The crossover process begins by selecting a random interval from the first parent key. The characters found within this interval are replaced by the characters found in the same interval in the second parent key. This can also be thought of as substituting a region of one high-fitness key with the same region from another high-fitness key in the hopes of generating an offspring key of higher fitness than either of the parents'.

In the case of Vigenère keys, "axyd" is a valid offspring of the keys "abcd" and "wxyz", while "abce" is not, since Crossover doesn't introduce letters which aren't found in either parent key.

2.3 Particle Swarm Optimization

Particle Swarm Optimization (PSO) is a second nature-inspired optimization algorithm. PSO however does not model biological evolution but the flocking behaviour of animals like birds and some social insects. The solution space of the problem being optimized is modeled as Euclidean n-space, and potential solutions to the problem are modeled as particles which fly through this space based on a *social tendency* and an *individualistic tendency*. Selecting a number of dimensions for modeling the solution space can be done in any number of ways; however, an intuitive mapping is most useful. In the case of using PSO for the cryptanalysis of cryptosystems which utilize character strings as keys, one possibility (the one implemented as part of this research) is to set the number of dimensions for the n-space to be the maximum number of letters the key could contain, or the *maxKeyLength* parameter used in the GA. Once a decision has been made as to the number of dimensions for the solution space, the coordinates of any geometric point in this n-space represents a potential solution to the problem being optimized. As is the case in this research, if solutions being sought consist of discrete coordinates in $\mathbb{Z}^n$, one can still interpret the solution space as $\mathbb{R}^n$ and simply truncate the coordinates to obtain integer values. Particle swarm optimization can be seen as a direct implementation of a fitness landscape, as discussed in Section 2.2.2, along with an algorithm which dictates the locomotory behaviour of particles in that landscape. Again, the facility of the null character is used so that keys with fewer than $maxKeyLength$ characters can be represented, as was shown in Section 2.2.2.

2.3.1 Initialization

As with genetic algorithms, PSO requires a parameter *pSize* which dictates the size of the population of particles which searches the solution space. To initialize the algorithm, the number of dimensions must be specified and the ranges within these dimensions which we want to search. For the cryptanalysis of character string based cryptosystems we can specify the number of dimensions to be the maximum key length we want to look for, and the range of each of these dimensions to be the size of the alphabet. Finally, *pSize* randomly positioned particles are placed within the solution space.

2.3.2 Evaluation

At the beginning of each iteration of the algorithm, the fitness of each particle must be calculated. Since the coordinates of each particle represent a string of up to n letters, these coordinates can be directly interpreted as keys, much like the genotypes found in a genetic algorithm. The fitness of these keys can be evaluated exactly as was done in the GA as described in Sections 2.2.2 and 2.2.3. The fitness functions used in the PSO implementation are the same fitness function that were used in the GA implementation.

2.3.3 Particle Locomotion

In contrast to the update step found in a genetic algorithm where genotypes move from generation to generation while being modified by mutation operators, in PSO particles move through the solution space based on a fixed algorithm. The locomotion of particles in PSO is calculated using four items. The first item is the particle's

current position which is an n-dimensional vector of real numbers, $current[]$. The second item is the particle's velocity, also an n-dimensional vector of real numbers, $velocity[]$. The third item is referred to as $pBest[]$. $pBest[]$ is a set of coordinates in the solution space which indicates the location where the particle being considered has been in a previous iteration of the algorithm and experienced higher fitness than it has at any other location in the solution space. The final item used to calculate a particle's trajectory is referred to as $gBest[]$. $gBest[]$ is a set of coordinates in the solution space which indicates the location of highest fitness encountered by *any* particle in the system during any previous iteration. The location of a particle for the next iteration $next[]$ of the algorithm is obtained through Equations 2.14 and 2.15.

$$next[] = current[] + velocity[] \tag{2.14}$$

$$velocity[] = c_1 * rand() * (pBest - current[]) + c_2 * rand() * (gBest - current[]) \tag{2.15}$$

The $rand()$ function returns a random real number in the interval $[0, 1]$. The first term on the right hand side of Equation 2.15 is an individualistic tendency which directs the particle towards its own personally-encountered best solution. The parameter c_1 is supplied by the user and weights the individualistic tendency relative to the social tendency. The second term on the right hand side of the equation is the social tendency which directs the particle towards the globally-encountered best solution. As with c_1, c_2 weights the social tendency relative to the individualistic tendency.

Each particle is moved once at each iteration of the PSO algorithm. The PSO algorithm now loops back to the evaluation phase unless a termination condition is

reached.

2.3.4 Termination

As with genetic algorithms, PSO will terminate if one of two conditions is satisfied. The first such condition is the reaching of a user-specifiable maximum number of iterations. It is possible that PSO will never find a satisfactory solution, so a maximum number of iterations ensures that the algorithm will not run indefinitely. The second termination condition is that the solution located at the coordinates indicated by $gBest$ has a sufficiently high fitness value to indicate a satisfactory solution.

Chapter 3

Selected Previous Work

Applying genetic algorithms to cryptanalysis is not a common research topic; however, it is not a new idea either. A number of papers have been published on the topic since the mid 1990s. Most of these papers attempt to apply genetic algorithms to the cryptanalysis of one specific cryptosystem. A number of different cryptosystems have been attacked using genetic algorithms, including various substitution ciphers, transposition ciphers, knapsack ciphers and even the coherent running key cipher. None of these papers attempt to apply a general purpose genetic algorithm to an array of cryptosystems. Most of these papers are summarized in a Master's Thesis by Bethany Delman of the Rochester Institute of Technology [7]. The work summarized in the subsequent pages has been selected on the basis of providing sufficient detail or introducing ideas not seen in other work. A substantial amount of work done in this area is focused on implementation of parallel genetic algorithms, and the bulk of these papers is not relevant due to their focus on the parallelization of the problem being solved. Further material on the subject of cryptanalysis using genetic algorithms can be found in [3], [8], [4], [5], [17] and [31].

The work presented in this thesis differs from the other works in the subject area in a number of ways. Most of the previous work in this area concentrates on applying genetic algorithms to the cryptanalysis of one individual cryptosystem, whereas the work presented here applies the same genetic algorithm across a range of cryptosystems. This provides a basis for comparing the resistance different ciphers have to the

same attack. Secondly, this research is one of few to exploit the flexibility of genetic algorithms by using mutation operators beyond the standard point mutations and crossovers seen in typical GA's. Also, while other research uses various metrics for measuring the effectiveness of GA-based cryptanalysis, this research uses the number of GA generations, which can be easily converted to a number of decryptions. Measuring decryption speed in terms of the number of decryptions required forms a basis for comparison to exhaustive search-type attacks. Finally, while most research only uses one specific GA implementation, this research compares different GA implementations when one is shown to be inadequate and explores the use of other algorithms for cryptanalysis.

3.1 Substitution Ciphers

The first published paper on the subject of using genetic algorithms for cryptanalysis was published by Spillman et al. in 1993 [26]. This paper examined the effectiveness of genetic algorithms in finding the key to a simple monoalphabetic substitution cipher. The genetic algorithm implementation operated only using the standard crossover and point mutation operators which are commonly associated with genetic algorithms. The fitness function used compared the frequency of occurrence of certain letters and digrams found in the decryption under a key to those found in typical English text. The genetic algorithm implementation identified all 26 elements of the key correctly after examining only 1000 candidate keys.

In 1994 Andrew Clark published a paper [1] on the application of three different optimization algorithms, including GA's, to cryptanalysis. The length of the paper

and lack of details makes it of limited use; however, the fitness function used in the GA implementation is very similar to the one found in [26] and exploits character and digram frequencies.

In 1997, Clark and Dawson published a paper [2] where a parallel genetic algorithm is used to attack a generalized polyalphabetic substitution cipher. The work met with a large degree of success by using separate computing nodes to run individual genetic algorithms, each set to find the correct key letter for a single position in the key. The problem of finding the key length was approached using classical methods as opposed to using the genetic algorithm to identify the key length as is used in this research. The fitness function used in the paper was again frequency analysis of individual letters and digrams, although knowledge of some of the plaintext was also exploited.

3.2 Other Ciphers

R. A. J. Mathews published a paper [18] in early 1993 detailing the application of genetic algorithms to transposition ciphers. This work did not use any known text, but did use the genetic algorithm to uncover the key length. Despite high hopes, the genetic algorithm implementation did not manage to completely cryptanalyze any ciphertexts. Correct key lengths were often identified, but the seemingly more difficult task of identifying the correct permutation of the discovered length was not regularly accomplished by a pure GA.

In October of 1993, Richard Spillman published a second paper [25] on genetic algorithm-aided cryptanalysis, this time of the knapsack cipher. The knapsack cipher

is an early attempt at a public-key cryptosystem and is based on the NP-complete knapsack problem. Although a relatively modern cryptosystem, most incarnations of the knapsack cipher have been shown to have vulnerabilities. The work in [25] shows that these ciphers are also vulnerable to attack by genetic algorithms, as GA's showed themselves to be effective at at least partially solving the knapsack problem underlying the cryptosystem. Since the GA's used attacked the knapsack problem directly, the mutation operators and fitness function used in the work acted upon an internal (bit-string) representation of a knapsack. The advantage of this approach is that the GA only needs the parameters of the knapsack in order to proceed and does not make assumptions regarding the contents of the message (that it is written in English, for instance).

In 1994, Giddy and Safavi-Naini used simulated annealing to break transposition ciphers [10] with some degree of success. As in the research presented here, the effectiveness of the cryptanalysis was proportional to the ratio of key length to known plaintext length.

Another paper attacking knapsack ciphers was published in 1997 [14]. Here, the Merkle-Hellman knapsack was attacked using a genetic algorithm. Unfortunately the results are of questionable usefulness as the knapsack variant used is considered a much less difficult version of the knapsack problem than the original NP-complete problem. Furthermore, the number of super-increasing elements found in the knapsack was limited to 8, making it a relatively simple problem to solve using other means.

Levbedko and Topchy compared the performance of a number of algorithms for solving the knapsack problem in their 1998 paper [16]. These methods included

simulated annealing, local search and a number of genetic algorithm and genetic algorithm-hybrid techniques. Again, the work was based on attacking the underlying knapsack problem directly, removing the requirement of the cryptanalyst to make assumptions about any message. When combined with the other techniques, the genetic algorithms turned out to be surprisingly adept at breaking the knapsack problem on small knapsacks.

3.3 Other Work

John A. Clark published a survey paper [6] in 2003, detailing work which has been done using nature-inspired algorithms in cryptography and cryptanalysis. The paper surveys much of the work referred to in this section and notes two prevalent themes – applying genetic algorithms to classical cryptosystems and to NP-complete problems underlying modern cryptosystems. Both are referred to as starting points for applying genetic algorithms to modern cryptanalytic problems.

One of the best reference materials on the subject of genetic algorithms as applied to cryptanalysis is a Master's thesis by Bethany Delman [7] which summarizes many of the previous works in the subject area. Delman's thesis details attempts to reimplement a number of the genetic algorithm-based attacks seen in the literature with limited success due to the lack of published parameter settings.

3.4 Current Work

While a fair amount of work has been done in this area in the past, the work presented in this thesis differs from the majority of previous work in the area in a number of

ways.

One of the most marked differences between this work and other work on this subject is the testing of a variety of mutation operators to identify combinations of mutation operators which are effective for cryptanalysis. In Holland's original book on genetic algorithms [13], he proposes three mutation operators, crossover, point mutations and inversion. The inversion operator reverses the contents of randomly selected intervals of a solution. None of the researchers applying genetic algorithms to the cryptanalysis of substitution ciphers go beyond implementing these basic mutation operators. None of the other research in this area evaluates the mutation operators used in experiments for suitability. The systematic evaluation of mutation operator combinations seen in this research is unique to the field and yields interesting results. The results shown in Sections 5.3, 6.3 and 7.3 demonstrate that one of the original mutation operators (crossover) which is implemented without question in other work is not necessary for the cryptanalysis of substitution ciphers.

With the notable exception of [7], none of the other work in this area attempts to apply the same genetic algorithm implementation to the cryptanalysis of a number of different cryptosystems. The work presented in this thesis applies an identical genetic algorithm-based attack to a number of cryptosystems in order to compare their resistance to the cryptanalytic technique. While Sections 5.3, 6.3 and 7.3 show that the same genetic algorithm-based attack is effective against many polyalphabetic substitution ciphers, Section 8.5 makes it apparent that genetic algorithm-based cryptanalysis works best when tailored to the application.

The replacement rules used in this research to move solutions from one generation to the next is unique to the field. Most other work in the area adheres to the

probabilistic tenuring scheme introduced by Holland [13] and Goldberg [11] known as *roulette wheel selection*, where each solution has a chance of being selected for mating based on its fitness relative to the fitness of the other solutions in the population.

The metrics used in some of the other research in this area (most notably [7]) are not useful for comparing genetic algorithm-based cryptanalysis to other cryptanalytic methods. The metric used to gauge the success of the algorithms implemented in [7] is time, which does not provide a basis for comparing these attacks to an exhaustive search-type attack. The use of time as a metric for success is not practical because it is dependent on the hardware and software in the testing environment used as well as the genetic algorithm implementation itself. The use of the number of generations required by the genetic algorithm as a metric for success is more practical because it allows researchers to compute the number of decryptions performed by the algorithm, which can be used to compare the effectiveness of genetic algorithm-based attacks to exhaustive search. Knowing how many decryptions were performed by the genetic algorithm gives researchers the opportunity to estimate the amount of time required for the genetic algorithm to finish regardless of the computing platform used.

One final area where this research contributes to the field is that in a number of cases especially when applying a genetic algorithm to the cryptanalysis of permutation ciphers, genetic algorithms implemented as part of other work were not able to fully recover the permutations used as keys. In many cases the genetic algorithm was able to identify the length of the key but required human interaction to identify the permutation itself. The genetic algorithm implemented as part of this research is able to perform both of these tasks without supervision.

Chapter 4

Shift Cipher

4.1 Normal Operation

The shift or Caesar Cipher is perhaps one of the most widely known examples of a cryptosystem. Although this cipher has no modern applications due to the small size of the key space, it seemed like a good place to start for this work. The cipher was named for Julius Caesar who allegedly encrypted messages by shifting each letter in the message by three. The scheme can be generalized by allowing any shift from 0 to 25, equivalent to each letter in the alphabet.

The specific implementation used in this work operates as follows: The message space M is the set of all arbitrarily long strings composed of the letters a through z, or $[a-z]^*$. Although the key space K can be viewed as any number from 0 to 25, in order to make implementation consistent with that of some other cryptosystems, we chose to represent keys as a single letter from a to z, representing circular shifts from 0 to 25 characters. Encrypting a message with a key always yields another string of letters, so the ciphertext space C is the same set as the message space M.

To encrypt a message m, each letter in the message is mapped to an integer according to Table 2.1.

The key letter is mapped to an integer, also using Table 2.1. Now the integer representative of the key is added (modulo 26) to each integer in the message. An example is presented in Table 4.1, using the message "example" and the key 'd'

Table 4.1: Shift Cipher Example.

message	e	x	a	m	p	l	e
as integers	4	23	0	12	15	11	4
key	3	3	3	3	3	3	3
addition	7	0	3	15	18	14	7
ciphertext	h	a	d	p	s	o	h

(which maps to the integer 3), which yields a ciphertext of "hadpsoh".

To decrypt a ciphertext c, instead of adding the key letter to each letter in c, the key letter is subtracted (modulo 26) from each ciphertext letter. This yields the encryption and decryption functions seen in Equations 4.1 and 4.2. As can be seen, the Caesar cipher is a special case of the Vigenère cipher, where the key length is one character.

$$E(k, m_i) = m_i + k \pmod{26}, \tag{4.1}$$

$$D(k, c_i) = c_i - k \pmod{26}. \tag{4.2}$$

4.2 Classical Cryptanalysis

Cryptanalysis of the shift cipher is not complicated. Perhaps the most notable weakness of the cryptosystem is the size of the key space. Since $|K| = 26$, with modern technology it is trivial to simply try each key until the message is recovered.

Cryptanalysis of the shift cipher can be expedited if we allow for the assumption that the message which was encrypted was written in English. It has been established that the most commonly occurring letter in English text is the letter 'e'. Indeed, the letter 'e' accounts for over 12% of all letters occurring in a typical passage of English text, the next most common letter, 't' accounts for only 9%. This indicates that the

cryptanalyst can assume that the most commonly occurring letter in a ciphertext is the encrypted form of the letter ‘e’ and be correct with high probability. Once the most commonly occurring letter in a ciphertext is identified, subtracting the integer representation of the letter ‘e’ from the integer representation of this most common letter will yield the integer representation of the key letter.

For example, if the most commonly occurring letter in a ciphertext is the letter ‘l’, we take the integer representation of ‘l’, 11 according to Table 2.1, and subtract the integer representation of ‘e’ (4). This yields 7, which is interpreted as the letter ‘h’ as the most likely key.

4.3 Cryptanalysis Using Genetic Algorithms

Despite the fact that there are quick and simple ways to cryptanalyze the shift cipher, there is no fundamental reason why one couldn’t use a genetic algorithm to do this work as well. Experimenting with the shift cipher allowed us to troubleshoot the genetic algorithm implementation and ensure the correctness of the algorithm. Successful cryptanalysis of the shift cipher provided valuable feedback as to the feasibility of the technique on more complicated character string based cryptosystems. Since a proper shift cipher key is only a single letter, the genetic algorithm was restricted to looking for key strings of length 1.

Since the standard genetic algorithm parameters being used in this work included a population size of 20 (distinct) individuals, there was a high probability that one of the initial random keys selected to seed the population would in fact be the correct key. In 20 separate trials, the correct key was identified in an average of 2.2

Table 4.2: Shift Cipher Results.

Trial	Number of Generations
1	2
2	1
3	1
4	2
5	2
6	2
7	1
8	4
9	1
10	1
11	2
12	4
13	2
14	4
15	2
16	2
17	1
18	3
19	3
20	4
Average:	2.2

generations as shown in Table 4.2.

4.4 Discussion

The shift cipher is a completely insecure monoalphabetic substitution cipher. The insecurity of this cryptosystem should come as no surprise due to its age. The shift cipher falls to a very simple frequency analysis attack, and is not immune to a brute force attack due to the very small size of K. These factors along with the relative ease of implementation made this cryptosystem a good candidate for

testing the genetic algorithm implementation developed for this work. As one would expect, the genetic algorithm was able to identify the correct key, usually within the span of two generations, consisting of twenty individual keys, or maximally forty decryption attempts. Since $|K| = 26$ for the shift cipher, this is no real improvement over exhaustive search; however, success with an exceedingly simple cryptosystem validated further investigation of this cryptanalytic technique.

Chapter 5

Vigenère Cipher

5.1 Normal Operation

The Vigenère cipher is one of the best known polyalphabetic substitution ciphers and can be most simply described as a number of shift ciphers working in unison. The Vigenère cipher was first described by Giovan Batista Belaso in 1553, but later misattributed to Blaise de Vigenère. Blaise de Vigenère in fact had nothing to do with the invention of this cipher, but was responsible for the invention of the Autokey cipher which will be discussed in a later chapter. The Vigenère cipher was originally thought to be exceptionally difficult to break, although this assertion can easily be shown to be false.

Unlike the Caesar cipher, the Vigenère cipher requires an entire key word to act as a key as opposed to a key letter. While with the Caesar cipher, each letter in the message is shifted the same number of places, the Vigenère cipher requires that each letter in the message be shifted by a number of places dependent on its corresponding key letter. The correspondence between letters in the message and letters in the key is specified by writing the key underneath the message repeatedly until each message letter has a key letter written below it. Each message letter is then shifted by the number of places indicated by the key letter written beneath it to produce the ciphertext. An alternative method of conceptualizing the encryption process is to apply a shift cipher to each letter in the message. The degree of the shift

is dictated by the corresponding key letter, so an n-letter key dictates that each letter in the message be encrypted with one of the n shift ciphers. The same process is repeated to decrypt the ciphertext, except that shifts are made backwards instead of forwards. The correspondence between letters of the alphabet and numbers is again made using the scheme presented in Table 2.1. An example of encryption using the Vigenère cipher is provided in Table 2.2.

As can be seen in the example, the use of a key word of length 4 results in the application of 4 different shift ciphers. Similarly, an n-letter key would result in n different shift ciphers being applied (assuming that the message was longer than n letters).

The encryption and decryption functions for the Vigenère cipher can be seen in Equations 5.1 and 5.2, where m_i is the i^{th} letter of the message, c_i is the i^{th} letter of the ciphertext and k_i is the i^{th} letter of the key [19].

$$E(k, m_i) = m_i + k_i \pmod{length(k)} \pmod{26} \tag{5.1}$$

$$D(k, c_i) = c_i - k_i \pmod{length(k)} \pmod{26} \tag{5.2}$$

5.2 Classical Cryptanalysis

Cryptanalysis of the Vigenère cipher is more complicated than that of the shift cipher, and is usually broken down into two steps. First, the cryptanalyst must discover the length of the key that was used, and then identify each letter which was used to form this key.

The ϕ-statistic which was introduced in Section 2.2.3 can be used to confirm the correctness of a guess at the length of a Vigenère key. A guess-and-check type

approach is used to identify the key length. The ϕ-statistic is useful for confirming the length of the key because calculating the ϕ-statistic of a *subtext* and comparing this value to expected values for random text and English text allows us to identify English text as was shown in Section 2.2.3, as well as identify text which was *monoalphabetically* encrypted. The shift cipher is a monoalphabetic substitution cipher, and text encrypted with such a cipher will have a similar ϕ-value as English text.

By guessing a key-length of n, a Vigenère ciphertext can be separated into n different subtexts. The first subtext will contain the first letter of the ciphertext and every n^{th} subsequent letter. Given indices 0 to l to the letters comprising the ciphertext, Equation 5.3 can be used to identify the letters which comprise each of the n subtexts.

$$Subtext_i = (c_{i+kn} | 0 \leq i + kn \leq l) = (c_i, c_{i+n}, c_{i+2n}, \ldots) \tag{5.3}$$

The intuition here is that if the guess at the key-length (n) is correct, then each subtext contains only letters from the ciphertext which were encrypted using the same key letter, or shifted by the same amount. Since all of the letters in a subtext were encrypted using the same key letter, the character frequencies within this subtext should be the same as that which would result from a ciphertext which was monoalphabetically encrypted and thus the same as for English text. Each of the n subtexts will have ϕ values similar to that of English, while if the guess is incorrect, then all of the subtexts will not have English-like ϕ values. An example of this process can be seen in Section 2.1.2.

Once the correct value of n has been discovered, the cryptanalysis of the ci-

phertext continues by treating each of the n subtexts as separate shift ciphers. As was described in Section 4.2, the cryptanalyst can assume that the most commonly occurring letter, α, in each subtext is the encryption of the letter e. Subtracting the integer representation of the letter e (4) from $\alpha \pmod{26}$ will yield the integer representation of the key letter for that subtext. This process is repeated for each subtext, resulting in the entire key word.

5.3 Cryptanalysis Using Genetic Algorithms

Experiments on the Vigenère cipher were the foundation for much of the rest of this research, so much testing was done on the application of genetic algorithms to the cryptanalysis of this cipher. The first investigation to be conducted regarded the selection of mutation operators. A pool of mutation operators described in Section 2.2.4 was implemented. The genetic algorithm was allowed to run with the following parameters.

- Population size: 20
- Number of individuals tenured per generation: 5
- Number of random immigrants per generation: 5
- Number of generations: 1000
- Key length: 5
- Maximum key length: 8
- Ciphertext length: 522

- Known text length: 5
- Number of runs per mutation operator combination: 100

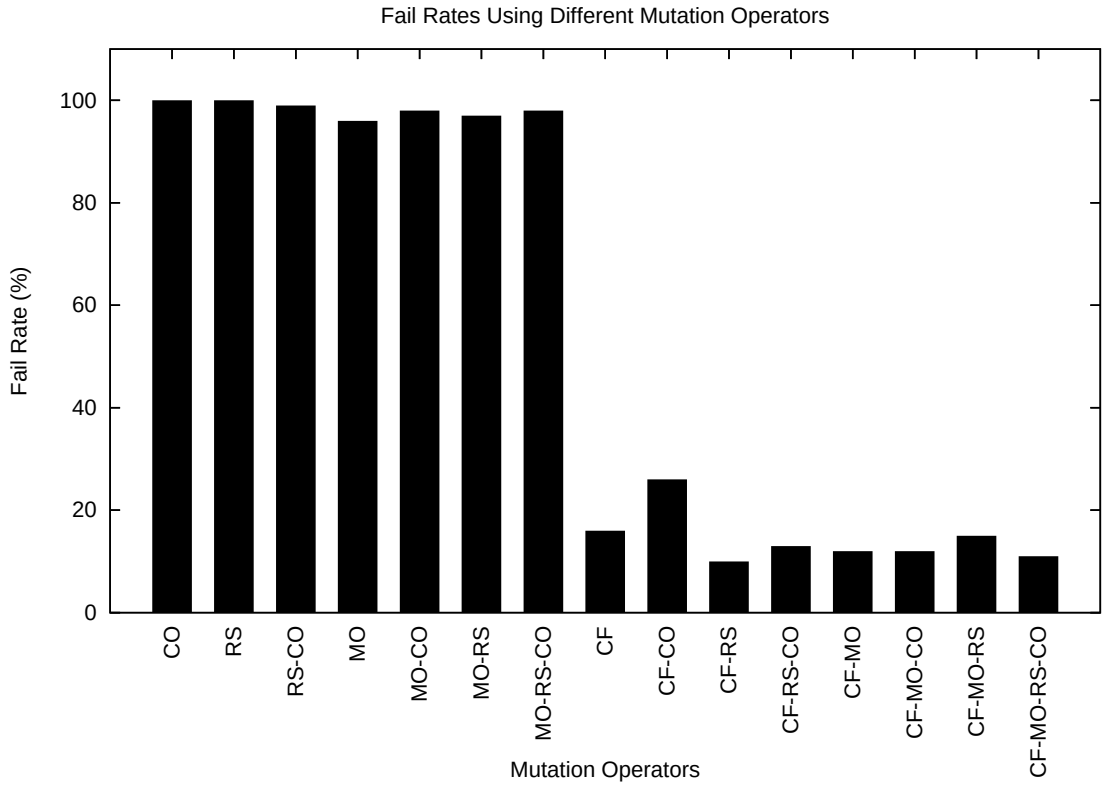

Figure 5.1: Vigenère Fail Rates Using Different Mutation Operators

The GA was allowed to run for up to 1000 generations; a test that took longer than 1000 generations to find the correct key was considered a failure. Since the goal of this research was to cryptanalyze a ciphertext as efficiently as possible using GA's, a small population size was chosen in an effort to minimize the number of decryptions needed to accomplish the task. The number of random individuals added at each generation as well as the number of individuals tenured at each generation was 5. This means that the top 1/4 of the population was always moved into the next generation in

case their offspring were of lower fitness. Another 1/4 of the population at each generation consisted of random individuals, preventing the population from being stuck in a local maximum of the fitness landscape. The final 1/2 of the population was formed by applying mutation operators to the 5 tenured solutions. This was done so that a large proportion of each generation would be devoted to attempting to improve upon the fitness scores of the best individuals found so far by recombining their genomes. At this stage of the research, only a key of length 5 was used so that the task of finding it wouldn't be trivial, but also not too difficult. The ciphertext which was selected was a long excerpt from an article on .NET programming found on the popular IT News website, *The Register*, which appeared to be written in fairly normal English. The length of the known text segment was chosen so that the Hamming distance-based fitness function would be able to return results with some granularity, but also so that it would not assume that the cryptanalyst knew an unreasonable amount of information about the message they were cryptanalyzing. The use of a respectable amount of known text is likely important to the performance of the algorithm because if very little known text is provided, the Hamming Distance fitness function has a very small range of values it can return. When more known text is provided, there are more intermediate fitness values which can be returned by the fitness function, resulting in a more easily optimizable fitness landscape.

As can be seen in Figure 5.1, the single most important mutation operator is the *Character Flip*, CF mutation operator. All combinations of mutation operators which do not include CF fail to decrypt the ciphertext 100% of the time, while all combinations which do include CF decrypt the ciphertext most of the time. The two most successful combinations of mutation operators are CF-RS (Character Flip and

Resize) which correctly decrypts the ciphertext within 1000 generations 91% of the time and CF-CO (Character Flip, Crossover) which manages the same task 91% of the time as well. One potential explanation for the effectiveness of the CF mutation operator is that this operator makes is possible for high fitness keys to be mutated to keys with even higher fitness. Consider a key with the letter "a" in a position where the correct key has the letter "e", but is otherwise correct. The CF operator makes it possible for the GA to mutate this key to the correct key within one generation. Although the MO operator functions very similarly to CF, this type of mutation is not possible with MO because MO can only change a letter to a neighboring letter in the alphabet. Since vowels and other commonly used letters in the English language are generally not found next to each other in the alphabet, an MO mutation is less likely to result in a key with higher fitness than a CF mutation.

The second investigation which was conducted attempted to find the combination of mutation operators which would be able to decrypt the ciphertext as quickly as possible. As can be seen in Figure 5.2, the most efficient combination is CF-CO. When equipped with this combination of mutation operators, a GA can successfully decrypt a ciphertext in an average of 67 generations. These tests were run using the same GA parameters as the previous experiments. 67 generations is a surprisingly low number considering the size of the key space. Since the maximum key length parameter was set to eight, the size of the key space being searched was $26^8 = 208827064576$, almost 209 billion distinct keys. Since the GA took 67 generations to find the correct key, with a population size of 20, this means that a maximum of $67 \times 20 = 1340$ keys were searched before the correct one was found, comprising 0.000000642% of the key space. Results of this experimentation can be seen in

Figure 5.2. Interestingly, the MO operator on its own rarely succeeds at decrypting the ciphertext correctly, but when it does, it does so very quickly. There is no obvious explanation for this. Since this pattern is not repeated in the analyses of the other substitution ciphers, perhaps this phenomenon can be explained as a few "lucky" runs.

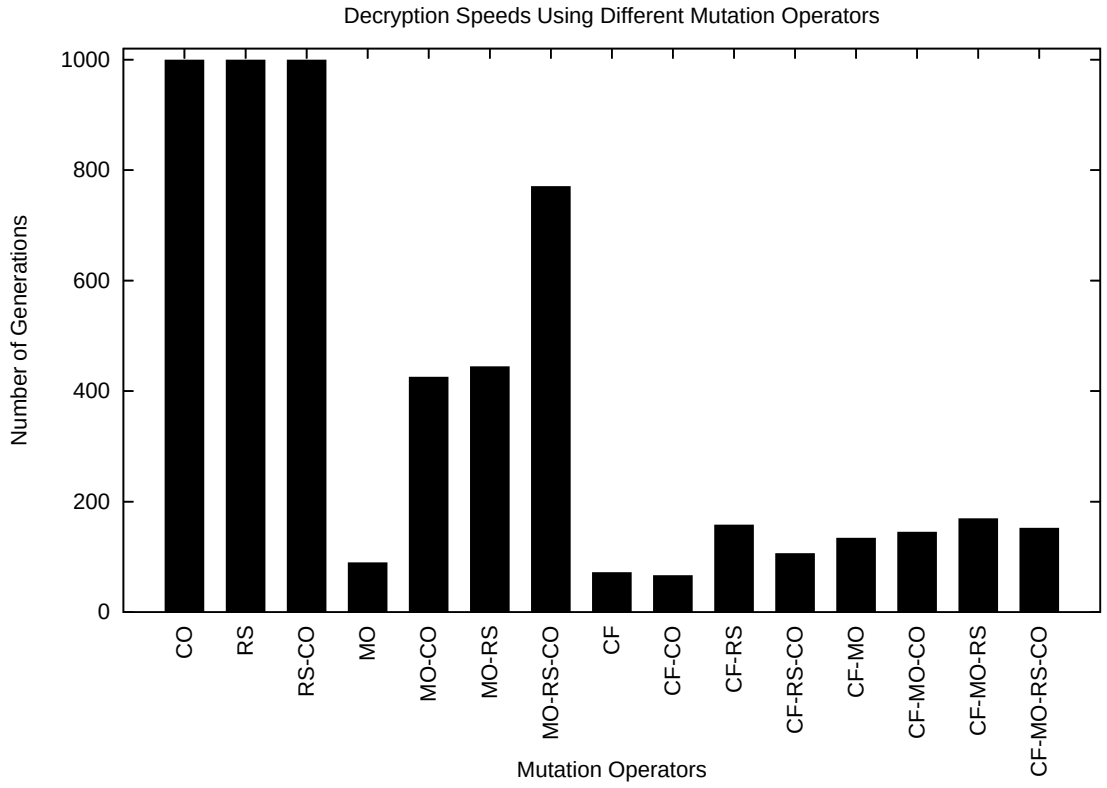

Figure 5.2: Vigenère Decryption Speeds Using Different Mutation Operators

The final tests which were run attempted to find how decryption time corresponds to the length of the key used to encrypt the ciphertext. The data points for each key length were produced by running the genetic algorithm with most of the parameter settings identical to those described above. Notable exceptions were the key length, the *maxKeyLength* and the amount known text. For each key length tested the

maxKeyLength parameter was set to 1.5 times the actual key length used, giving the potential cryptanalyst a wide margin of error when estimating the key length. For all tests, the genetic algorithm was provided with 16 characters of known text. At each key length, the genetic algorithm was run on all 9 combinations of 3 different messages and 3 different keys, using the CF-MO-RS combination of mutation operators which proved to be effective for all of the classical cryptosystems. Each of the 9 combinations keys and messages were run 10 times at each key length, resulting in 90 trials per key length. One of the messages used was the same 522-letter excerpt from the *The Register* which was used in previous experiments, the second message was an excerpt from the End User License Agreement (EULA) from Microsoft Windows Vista and the third message used is an excerpt from Tolstoy's War and Peace [28]. The messages, known text segments and keys used are provided in Section A.7. These same parameters were applied to the key length analyses for the other classical ciphers which were investigated.

The results from these experiments can be seen in Figure 5.3 which shows a number of interesting trends. The first pattern which emerges is that the longer the Vigenère key is, the less likely the genetic algorithm is to succeed, as would be expected. What is unexpected is that the number of generations required by the GA to cryptanalyze a ciphertext does not increase with the key length. The data shows that GA-based cryptanalysis of the Vigenère cipher shouldn't require more than 250 generations or 5000 decryptions, else it is unlikely to succeed. This is an unexpected result because the size of the key space increases exponentially with the length of the key, but the number of generations required to break the cipher does not. Only the likelihood of success of the GA seems dependent on the length of the

key. Possibly this indicates that the initial seeding is very important to the success or failure of the genetic algorithm despite the immigration of random solutions at each generation, but this explanation is only speculative.

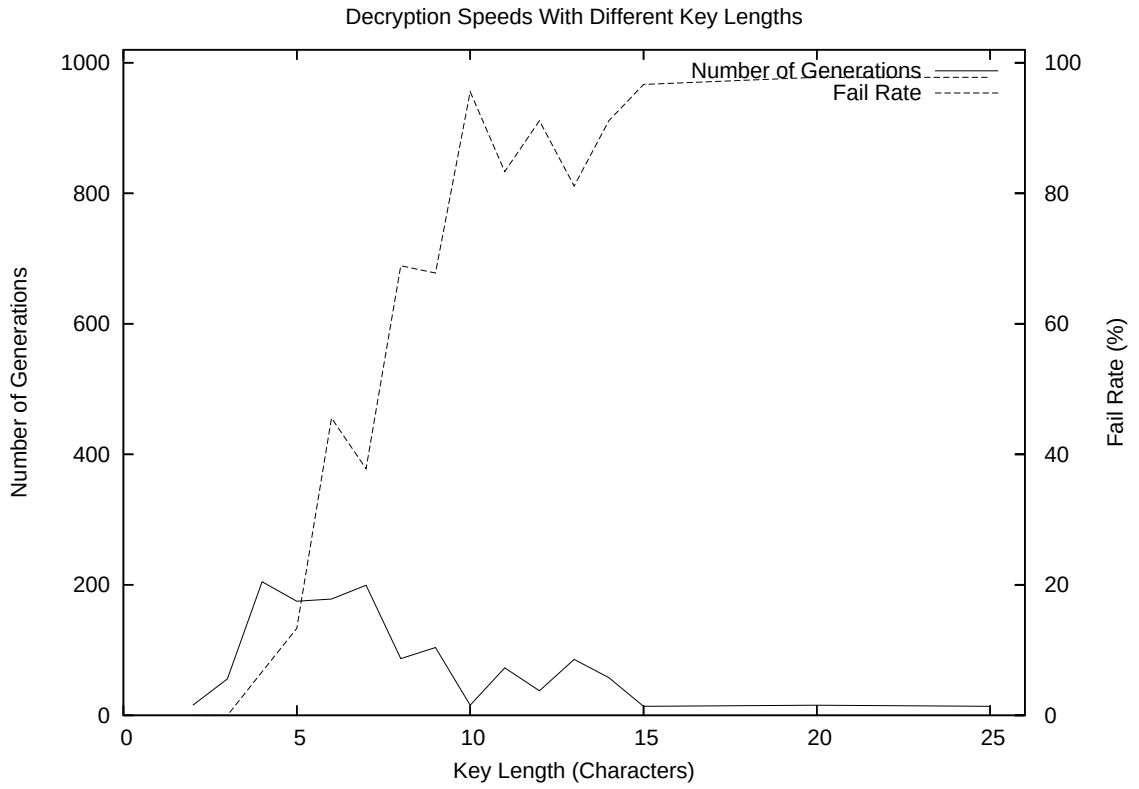

Figure 5.3: Vigenère Decryption Speeds With Different Key Lengths

5.4 Discussion

Since the Vigenère cipher was the first cryptosystem with an appreciable key space size that was tested for vulnerability to cryptanalysis by a genetic algorithm, the experiments on the Vigenère cipher form the foundation for the rest of this research. Genetic Algorithm cryptanalysis proved to be surprisingly efficient at finding the

correct decryption key from a very large set of potential candidates, searching less than one one-millionth of a percent of the key space before finding the correct key.

While this is a positive result, it may not be as surprising as the numbers imply. The fitness functions used by the genetic algorithm bear a very close resemblance to the techniques an expert system might use to cryptanalyze the Vigenère cipher if it were to use similar techniques to the one described in Section 5.2. The ϕ-statistic-based fitness function provides feedback as to whether the lengths of the keys in the population are correct, although this information isn't interpreted quite as intelligently by the GA. A low fitness value simply tells the GA that the corresponding key is a bad one, but not necessarily that its length is incorrect. The Hamming distance-based fitness function also provides the GA with information that could be used to reconstruct the key through simple subtraction by an expert system (assuming that the known text segment is as long as the key); however, again, the feedback provided by this fitness function isn't interpreted in that way by the GA. The GA only sees that some keys are better than others but has no intuitive understanding of why. So while it may not be surprising that the GA-based cryptanalysis works so well given that the fitness functions used are so close to the techniques an expert system might use, it is surprising that the genetic algorithm does its task so well given the limited amount of analysis the GA does with the information provided by the fitness functions.

In subsequent chapters the same cryptanalytic approach is applied to other polyalphabetic substitution ciphers, as well as a few modern cryptosystems, with varying levels of success.

Chapter 6

Mixed Vigenère Cipher

6.1 Normal Operation

The mixed Vigenère cipher is quite similar in operation to the original Vigenère cipher described in Section 5.1, but is more complicated to cryptanalyze. The difference between the two is that in the original Vigenère cipher the cipher alphabets used are regular alphabets in that a shift of one position from the letter a results in a b, and so on. In the mixed Vigenère cipher the cipher alphabet has a permutation applied to it before it is used for encryption. This permutation is induced by the same key word that is used to determine the shifts for each letter in the message.

To permute the cipher alphabet, the key word is first written down with the duplicate letters removed. The rest of the alphabet, without the letters which already appear in the key word, are then written in order after the key word. This process results in a permuted version of the alphabet as can be seen in Table 6.1, using the keyword "secret".

Once the cipher alphabet has been permuted using the key word, encryption and

Table 6.1: Permuted Alphabet

s	e	c	r	t	a	b	d	f	g	h	i	j
0	1	2	3	4	5	6	7	8	9	10	11	12
k	l	m	n	o	p	q	u	v	w	x	y	z
13	14	15	16	17	18	19	20	21	22	23	24	25

Table 6.2: Mixed Vigenère Cipher Encryption Example.

message	e	x	a	m	p	l	e	m	e	s	s	a	g	e
as integers	4	23	0	12	15	11	4	12	4	18	18	0	6	4
key	s	e	c	r	e	t	s	e	c	r	e	t	s	e
key (as integers)	18	4	2	17	4	19	18	4	2	17	4	19	18	4
addition	22	1	2	3	19	4	22	16	6	9	22	19	24	8
ciphertext	w	e	c	r	q	t	w	n	b	g	w	q	y	f

decryption is done as as in the original Vigenère cipher, but with the permuted cipher alphabet. In the above example, a shift of one position from s now yields e as opposed to the result t that would have been produced without a permuted cipher alphabet. The encryption and decryption functions shown in Equations 5.1 and 5.2 are still used, but when computing $m_i + k_{i \pmod{length(k)}} \pmod{26}$ and $c_i - k_{i \pmod{length(k)}} \pmod{26}$, the letter to integer correspondence shown in Table 6.1 is used instead of the one in Table 2.1. Encryption and decryption examples using the mixed Vigenère cipher can be found in Tables 6.2 and 6.3.

The encryption in Table 6.2 is constructed as follows. The first line is the message. The second line is the integer representation of the message according to Table 2.1. The third line is the key word written repeatedly below the message. The forth line is the integer representation of the key, again using Table 2.1. The fifth line is produced by adding lines 2 and 4 $\pmod{26}$. The final line is the letter representation of the fifth line, this time using Table 6.1.

The decryption in Table 6.3 is obtained similarly. Line 1 is the ciphertext. The second line is the integer representation of the ciphertext using Table 6.1. The third line is the key written repeatedly below the ciphertext. The forth line is the integer representation of the key using Table 2.1. The fifth line is obtained by subtracting

Table 6.3: Mixed Vigenère Cipher Decryption Example.

ciphertext	w	e	c	r	q	t	w	n	b	g	w	q	y	f
as integers	22	1	2	3	19	4	22	16	6	9	22	19	24	8
key	s	e	c	r	e	t	s	e	c	r	e	t	s	e
key (as integers)	18	4	2	17	4	19	18	4	2	17	4	19	18	4
subtraction	4	23	0	12	15	11	4	12	4	18	18	0	6	4
message	e	x	a	m	p	l	e	m	e	s	s	a	g	e

the forth line from the second line (mod 26). Finally, the sixth line is obtained by converting the integers on the fifth line to letters using Table 2.1 [19].

6.2 Classical Cryptanalysis

Cryptanalysis of the mixed Vigenère cipher is more involved than that of the original Vigenère cipher and requires that the cryptanalyst has a significant amount of ciphertext to work with. As with the original Vigenère cipher, the initial phase of cryptanalysis focuses on determining the key length. As described in Section 5.2, a guess at the key length is made and then verified using the ϕ-statistic. Adding the permutation to the cipher alphabet of each subtext doesn't change the fact that each subtext is monoalphabetically enciphered, nor the useful property that such a subtext will have an English-like ϕ value.

After identifying the number of cipher alphabets (key length) using the same method as with the Vigenère cipher, it is possible to identify certain letters in the cipher alphabets. Each of these cipher alphabets will be the same permuted alphabet, but with shifts of different numbers of positions occurring in each. This knowledge allows us to use the principle of *Symmetry of Position* [15] to uncover the rest of these cipher alphabets. The letters we can fix in each ciphertext are the letters which

encrypt to the letters e, t, a and o because these are the four most commonly used letters in the English language (in descending order of frequency). The four most common letters occurring in each subtext will with high likelihood decrypt to these letters. The ability to anticipate this mapping relies on the assumption that the original message was in fact written in (typical) English. This simple frequency analysis allowed us to fix four letters in each cipher alphabet; given fortuitous identities of these fixed letters, symmetry of position can be used to fix much of the rest.

As mentioned, symmetry of position relies on the fact that all of the cipher alphabets used are the same permuted version of the alphabet. If a specific pair of letters is separated by some number of positions in one cipher alphabet, this same pair of letters will be separated by the same number of positions in all other cipher alphabets. Being able to exploit this technique requires that there be overlap between the known letters of the different cipher alphabets. Consider the four alphabet representations of e, t, a and o shown in Table 6.4, and the partially completed four cipher alphabets in Table 6.5. All of the additional letters in Table 6.5 were placed solely by symmetry of position and the information provided in Table 6.4. Since we see that the letter r occurs four positions after the letter f in the first cipher alphabet (based solely on letters placed by frequency analysis), we can infer that r will occur four positions after f in all of the other cipher alphabets (because they are all permuted by the same key word). If we repeat this process for every pair of letters obtained through the frequency analysis phase, the partially completed cipher alphabets in Table 6.5 are obtained. At this point in the cryptanalysis, no further letters can be fixed in the cipher alphabets by symmetry of position. Obtaining a more complete view of the cipher alphabets can be accomplished by decrypting the

Table 6.4: Mixed Vigenère Cipher Alphabets after Frequency Analysis Phase.

a	b c d	e	f g h i j k l m n	o	p q r s	t	u v w x y z
f		r		b		x	
s		g		r		c	
y		s		f		h	
h		c		x		v	

Table 6.5: Mixed Vigenère Cipher Alphabets after Applying Symmetry of Position.

a	b c d	e	f g h i j k l m n	o	p q r s	t	u v w x y z
f		r	h c y	b	s	x	g y
s	x	g	v f	r	h	c	y b
y	b	s	x g v	f	r	h	c
h		c	y b s	x	g	v	f r

ciphertext with the partial look-up table in Table 6.5. When this partial decryption is complete, certain words in the message may become obvious, allowing us to place a letter (perhaps more) into one cipher alphabet and then apply symmetry of position to place this newly identified letter into the remaining cipher alphabets. A new partial decryption can now be obtained, hopefully yielding enough information to complete more of the cipher alphabets until the message is completely decrypted.

6.3 Cryptanalysis Using Genetic Algorithms

Since the parameters applied to the genetic algorithm worked well in the cryptanalysis of the Vigenère cipher, the same parameters as described in Section 5.3 were used for the cryptanalysis of the mixed Vigenère cipher.

Despite being significantly more difficult to cryptanalyze by hand, the mixed Vigenère cipher is not significantly more difficult to cryptanalyze using a genetic algorithm as compared to the original Vigenère cipher. As can be seen in Figure 6.1,

the CF-RS combination of mutation operators is again most likely to be successful in cryptanalyzing a ciphertext within 1000 generations. The CF-RS combination which was the most effective on the original Vigenère cipher is capable of successfully cryptanalyzing the mixed Vigenère cipher within 1000 generations 90% of the time.

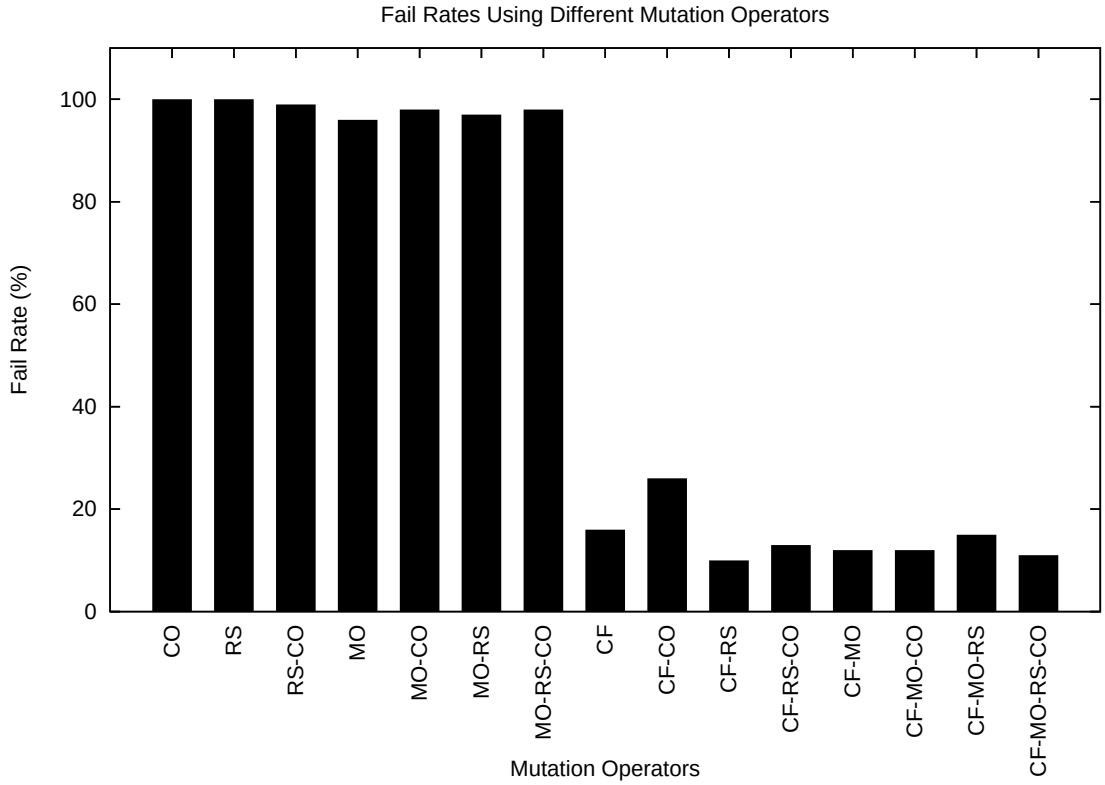

Figure 6.1: Mixed Vigenère Fail Rates Using Different Mutation Operators

The CF combination of mutation operators decrypts the fastest, successfully decrypting the ciphertext in an average of 67 generations, or examining a maximum of 1340 distinct keys out of the 209 billion keys comprising the key space. As can be seen in Figure 6.2, a number of other mutation operator combinations decrypted the ciphertext at a similar rate, though not quite as quickly as the CF combination. Comparing Figure 6.2 with Figure 5.2, we see that while there are minor differences in

the effectiveness of mutation operator combinations between the Vigenère and Mixed Vigenère cipher, a very distinct pattern emerges. All mutation operator combinations which include the CF operator are able to decrypt the ciphertext quickly and with a larger chance of success than mutation operator combinations which do not include CF. A potential explanation for this was offered in Section 5.3. It is puzzling that the Resize (RS) operator isn't just as critical to the success of the algorithm as the CF operator, as finding the correct key length is a prerequisite for identifying the contents of the key. One could speculate that since there are relatively few possibilities for the length of a key, a key of the correct length is introduced into the population during the initial seeding of the population or during one of the immigration phases. Once a key of the correct length is introduced into the population, the other mutation operators seem capable of finishing the cryptanalytic task, so long as the CF operator is present.

As can be seen in Figure 6.3, in conjunction with Figure 5.3, Figure 11.1 and Figure 11.2, for some key lengths the genetic algorithm was able to decrypt a message encrypted with the mixed Vigenère cipher even more quickly than messages encrypted with the original Vigenère cipher. This is an interesting result given the difficulty of classical cryptanalysis of the mixed Vigenère cipher and the added security thought to be added by using permuted cipher alphabets. The use of permuted cipher alphabets doesn't seem to make the cipher any more resistant to cryptanalysis by a genetic algorithm, in fact the opposite is true at some key lengths. An explanation for this phenomenon is not forth-coming, especially given the fact that the same ciphertexts were used as in the analysis of the classical Vigenère cipher.

Despite the above mentioned anomaly, the experiments show that genetic algorithm-

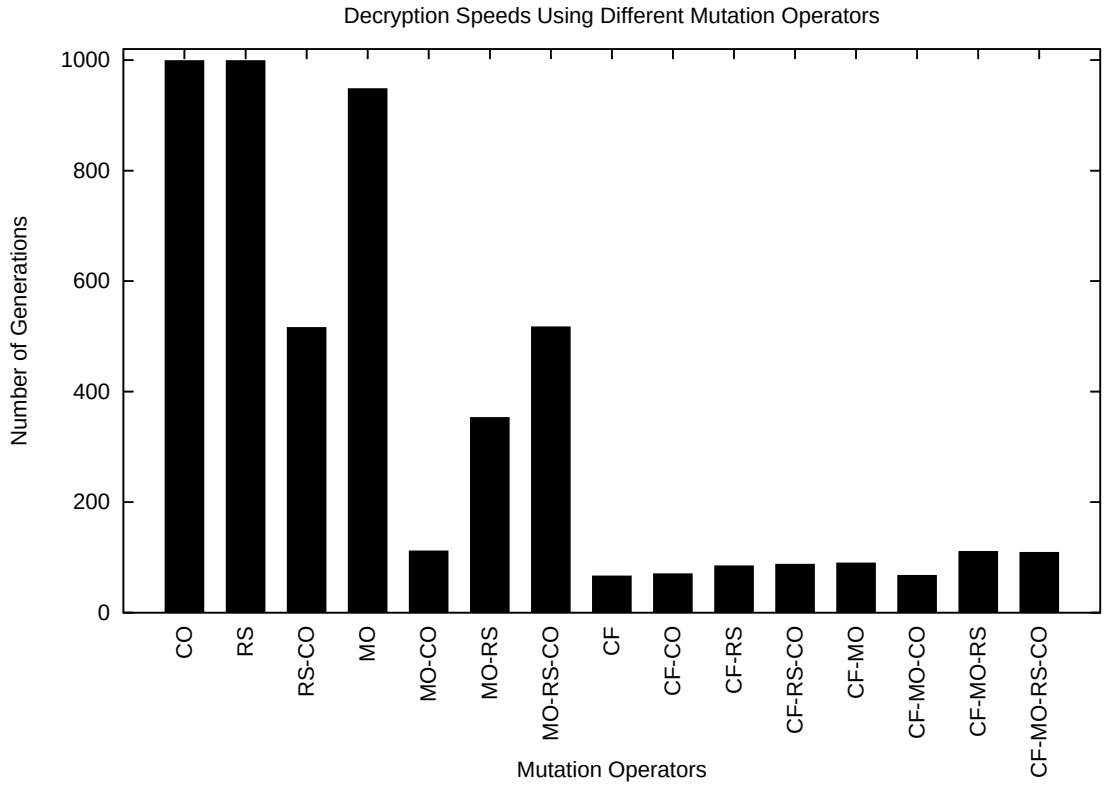

Figure 6.2: Mixed Vigenère Decryption Speeds Using Different Mutation Operators

based cryptanalysis of the mixed Vigenère cipher follows the same general patterns seen in the cryptanalysis of the original Vigenère cipher. Increasing the key length reduces the likelihood with which the algorithm will succeed, but not necessarily the number of generations required. Figure 6.3 implies that letting a genetic algorithm run for more than 250 generations is futile when cryptanalyzing the mixed Vigenère cipher.

Figure 6.3 shows lower failure rates at key lengths of 7, 11, and 13 which seem out of character with the rest of the data. These dips in failure rates correspond to spikes in the number of generations required by the genetic algorithm. While it is unclear what causes this phenomenon, it seems that at these key lengths, the mixed Vigenère cipher is more vulnerable to genetic algorithm-based cryptanalysis, but that the GA requires more generations than usual to succeed.

6.4 Discussion

The mixed Vigenère cipher offers significant cryptanalytic challenges over those offered by the original Vigenère cipher. The mixed Vigenère cipher's use of the key word to not only dictate shifts for each plaintext letter but also induce a permutation on the cipher alphabets used requires that the cryptanalyst apply the principle of symmetry of position in order to uncover the cipher alphabets. While the permuted cipher alphabets render classical cryptanalysis more involved and less likely to succeed, they oddly enough don't make the cipher less vulnerable to cryptanalysis by genetic algorithms at some key lengths. In fact, the genetic algorithm is in some cases able to decrypt a mixed Vigenère encrypted message more quickly than one

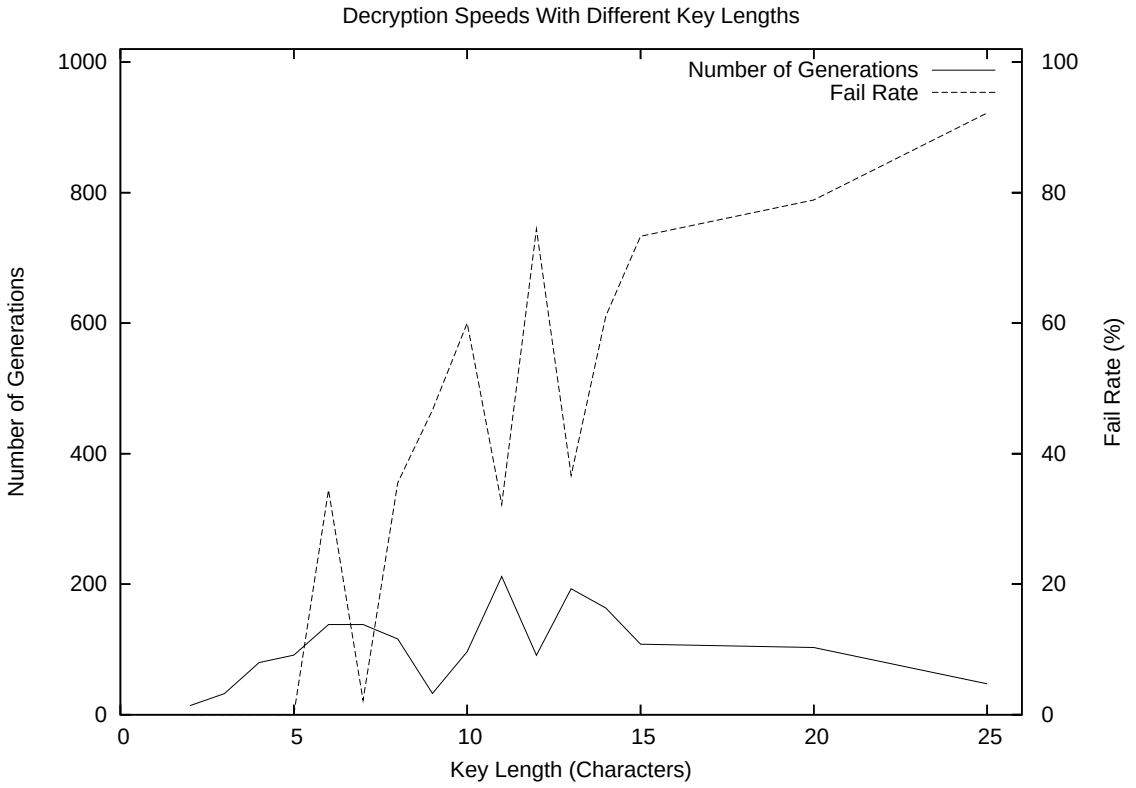

Figure 6.3: Mixed Vigenère Decryption Speeds With Different Key Lengths

encrypted with the original Vigenère cipher.

This unexpected success motivated us to attempt to apply genetic algorithm cryptanalysis to a cryptanalytically even more difficult polyalphabetic substitution cipher, the Autokey cipher. Ironically the Autokey cipher actually was invented by Blaise de Vigenère to whom the original Vigenère and subsequently the mixed Vigenère ciphers were misattributed.

Chapter 7

Autokey Cipher

7.1 Normal Operation

The Autokey cipher is another polyalphabetic substitution cipher, similar to the Vigenère and mixed Vigenère ciphers. The Autokey cipher, unlike the Vigenère cipher, actually was invented by Blaise de Vigenère and was dubbed *"le chiffre indéchiffrable"*, or "the unbreakable cipher". While not unbreakable, it did take 200 years for Charles Babbage to discover a method of breaking the cipher. The cipher operates exactly as the Vigenère cipher, but doesn't simply repeat the same key word in order to determine which shift to use for each letter of the message. In the Autokey cipher, the shift to be used for a given message letter is determined by first exhausting the key material and then moving on to the letters of the message itself. An example of Autokey encipherment can be seen in Table 7.1, with the message "example message" and the key "test".

As with the previous two ciphers, decryption is accomplished in the same manner

Table 7.1: Autokey Cipher Encryption Example.

message	e	x	a	m	p	l	e	m	e	s	s	a	g	e
as integers	4	23	0	12	15	11	4	12	4	18	18	0	6	4
key	t	e	s	t	e	x	a	m	p	l	e	m	e	s
key (as integers)	19	4	18	19	4	23	0	12	15	11	4	12	4	18
addition	23	1	18	5	19	8	4	24	19	3	22	12	10	22
ciphertext	x	b	s	f	t	i	e	y	t	d	w	m	k	w

Table 7.2: Autokey Cipher Decryption Example.

ciphertext	x	b	s	f	t	i	e	y	t	d	w	m	k	w
as integers	23	1	18	5	19	8	4	24	19	3	22	12	10	22
key	t	e	s	t	**e**	**x**	**a**	**m**	**p**	**l**	**e**	**m**	**e**	**s**
key (as integers)	19	4	18	19	4	23	0	12	15	11	4	12	4	18
subtraction	4	23	0	12	15	11	4	12	4	18	18	0	6	4
message	**e**	**x**	**a**	**m**	**p**	**l**	**e**	**m**	**e**	**s**	s	a	g	e

as encryption except that key material is subtracted from ciphertext letters instead of adding key material to message letters. If the party decrypting the ciphertext indeed has the correct key, then they decrypt the first section (the length of the key word) of the ciphertext as they normally would when using the Vigenère cipher, and afterwards they would start using the message letters they had already obtained as the key. A decryption example is provided in Table 7.2. Encryption and decryption functions for the autokey cipher can be seen in Equations 7.1 and 7.2, where m_i is the i^{th} message letter, c_i is the i^{th} ciphertext letter and k_i is the i^{th} key letter, where $k = k_1|k_2|k_3|...|k_{keylength}|m_1|m_2|m_3|...|m_{messagelength-keylength}$. The | operator indicates concatenation of letters [19].

$$E(k, m_i) = m_i + k_i \pmod{26} \tag{7.1}$$

$$D(k, c_i) = c_i - k_i \pmod{26} \tag{7.2}$$

7.2 Classical Cryptanalysis

The cryptanalysis of the Autokey cipher rests on the fact that since the message being encrypted is usually in normal English, then normal English text is being used as key material to encrypt the message. Since the key is mostly made up of normal

English text, we can assume that common English words, such as "the" must appear in the key. One way to exploit this knowledge is to decrypt parts of the ciphertext with a key word such as "the" or some other word which is likely to appear in the message. The hope is that if the proper section of the ciphertext is decrypted with one of these commonly occurring words, then a recognizable portion of the message will be revealed. If a recognizable portion of the message is revealed, the fact that the message is used as key material can be exploited. Assuming the original key word which was used for encryption is l letters in length, then the newly uncovered piece of the message, located at the c^{th} position in the message must be used as key material starting at the $(c+l)^{th}$ position of the ciphertext. It is also the case that the originally guessed word, which was used to decrypt a portion of the ciphertext must appear in the message at the $c - l^{th}$ position. Through repeated application of this process it is possible to decipher the entire ciphertext. This cryptanalytic technique requires a bit of luck and some knowledge of the message in order to succeed.

7.3 Cryptanalysis Using Genetic Algorithms

Using the same parameters as for the other cryptosystems already described, when comparing Figures 5.1, 6.1 and 7.1, it can be seen that correct cryptanalysis of the Autokey cipher is least likely of the three polyalphabetic substitution ciphers when using a genetic algorithm. The CF-MO-RS combination of mutation operators is most likely to be able to decrypt a ciphertext, succeeding 88% of the time. Figure 7.1 shows the performance of different mutation operator combinations on the autokey cipher.

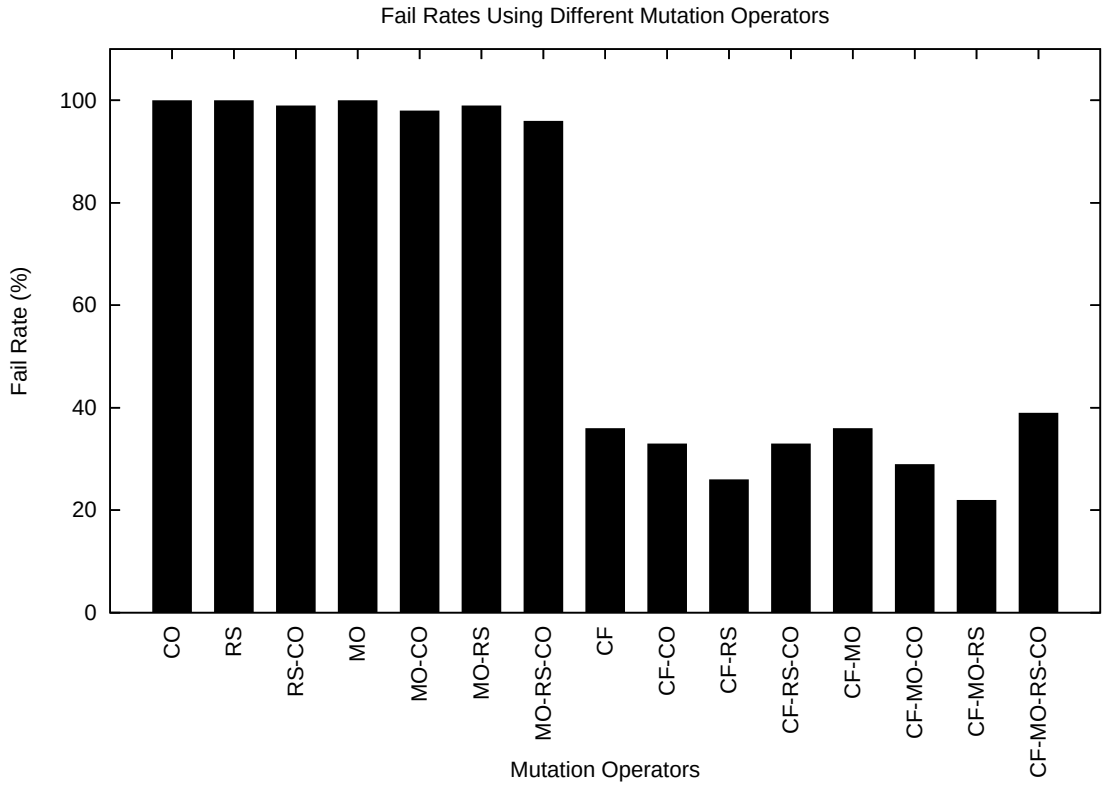

Figure 7.1: Autokey Fail Rates Using Different Mutation Operators

The CF-CO combination of mutation operators is able to achieve this task more quickly than the other mutation operator combinations. When the ciphertext is correctly cryptanalyzed, it is done within 72 generations on average. Again, since a generation size of 20 is used, this means that the genetic algorithm performs a maximum of 1440 distinct decryptions, or checks an average of 1440 keys before finding the correct one, this still only accounts for 0.00000069% of the key space. Figure 7.2 shows decryption speeds for different mutation operator combinations.

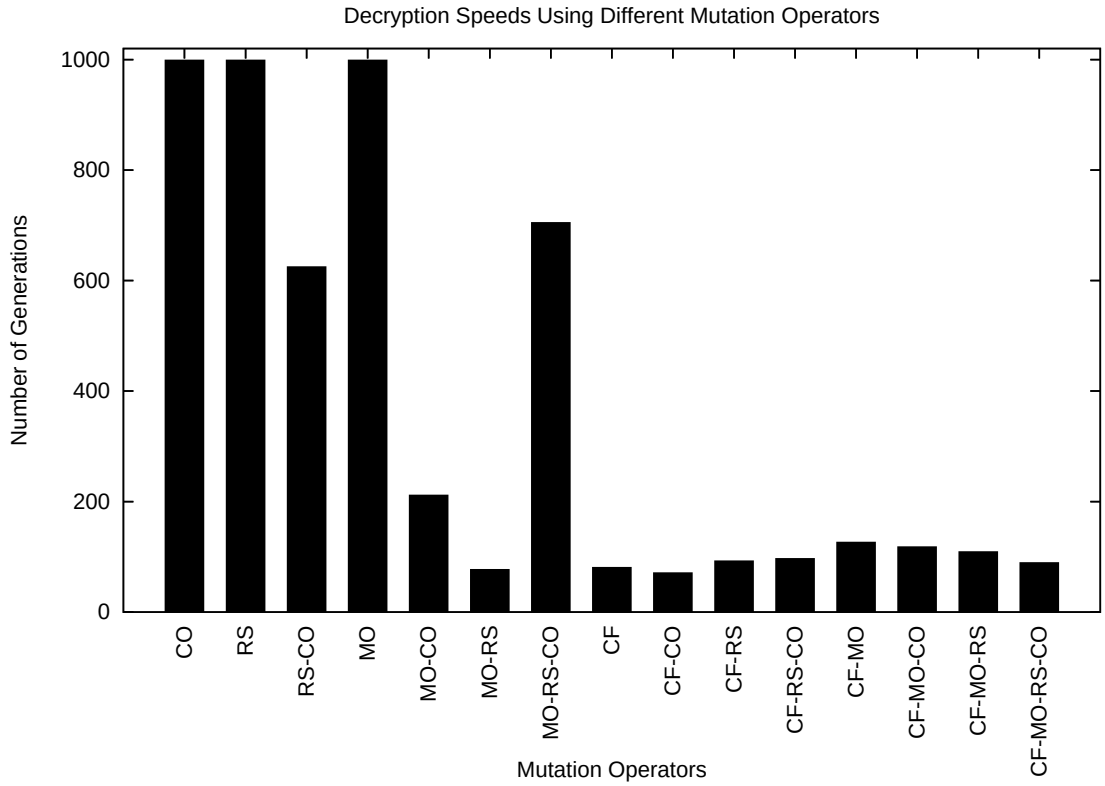

Figure 7.2: Autokey Decryption Speeds Using Different Mutation Operators

As can be seen in Figures 7.3, 11.1 and 11.2, the Autokey cipher appears to be (by a small margin) the polyalphabetic substitution cipher most susceptible to attack by a genetic algorithm. The general patterns exhibited in Figure 7.3 show that

cryptanalysis of the Autokey cipher follows the same patterns seen with the Vigenère and mixed Vigenère ciphers. Increasing the key length reduces the likelihood with which the algorithm will succeed, but does not cause an increase in the number of generations required by the GA. Figure 7.3 does not show the noticable peaks and valleys seen in Figure 6.3. For some reason the genetic algorithm performs the most consistently when confronted with the Autokey cipher, compared to other polyalphabetic substitution ciphers.

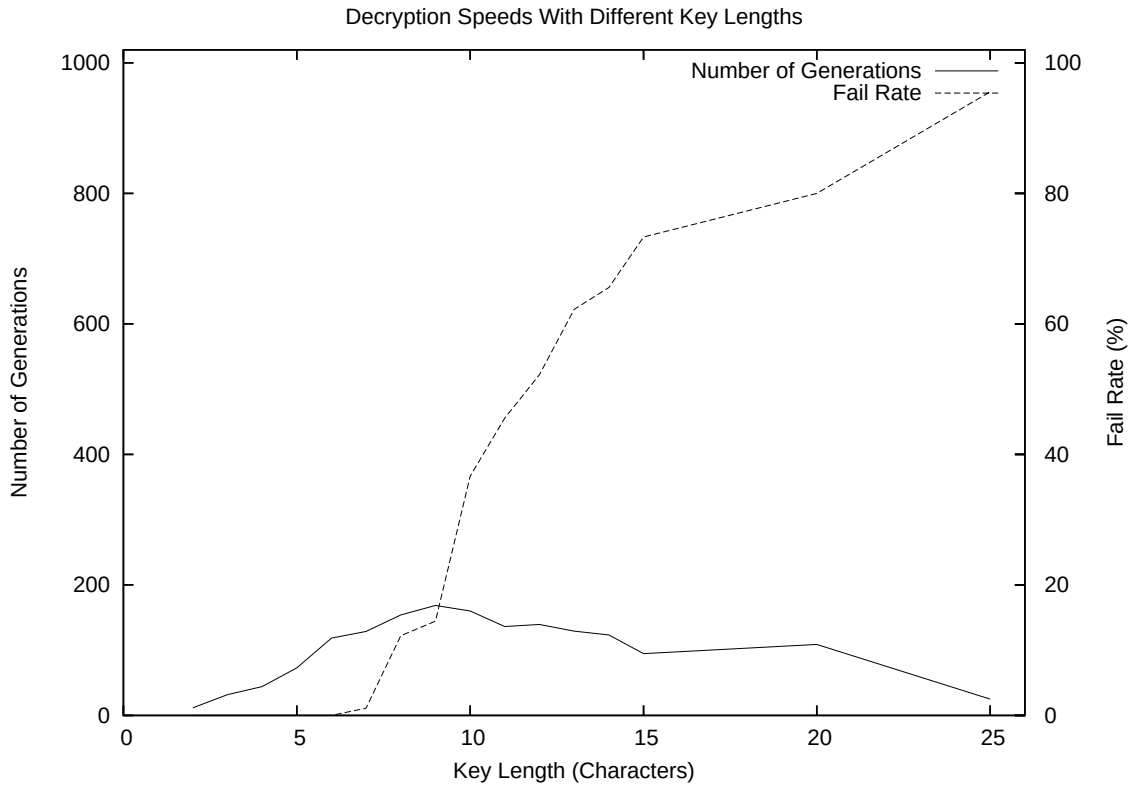

Figure 7.3: Autokey Decryption Speeds With Different Key Lengths

7.4 Discussion

While the Autokey cipher is very similar to the Vigenère and mixed Vigenère ciphers, the data gathered during the decryption of a ciphertext shows some unique characteristics. The relative invariability low failure rates with a varying key size is most puzzling because intuitively it seems that cryptanalysis of the Autokey cipher using a genetic algorithm should be hardest of all. The fitness value of a key is based on the entire decrypted ciphertext, as it was with the other cryptosystems; however, the correctness or incorrectness of the latter part of the resulting message was directly dependent on the key the genetic algorithm was testing. In the case of the Autokey cipher, the correctness/incorrectness of the latter part of the message (past the section which was encrypted using original key material), is still dependent on the key, but indirectly so. This latter part of the message is directly dependent on the decryption l characters back, which is in turn dependent on the key. The increased complexity and increased difficulty of classical cryptanalysis was not an obstacle for the genetic algorithm. It seems that despite significant implementation and cryptanalytic differences between the polyalphabetic substitution ciphers examined during this research, the performance of a genetic algorithm isn't affected in any significant way by the use of one or the other cipher.

Since identical parameter settings were used for the genetic algorithm between the testing of the implementation against the Vigenère, mixed Vigenère and Autokey ciphers, it seems that a GA can in fact be considered a general purpose cryptanalytic tool when confronted with polyalphabetic substitution ciphers. Neither of the polyalphabetic substitution ciphers seem to provide any real added security over the

others against genetic algorithm-based cryptanalysis.

Chapter 8

Columnar Transposition Cipher

The Columnar Transposition cipher, as the name implies, is a *transposition cipher* as opposed to a *substitution cipher* as are all of the previously discussed cryptosystems. In a substitution cipher the encryption and decryption functions E and D specify a mapping which is used to determine which element of the cipher alphabet is substituted for a specified message letter when encrypting a message and the reverse when decrypting a ciphertext. Potentially, a ciphertext may include none of the letters which appeared in the message or the key when using a substitution cipher because one letter is usually substituted with another as part of the encryption process. This generally isn't the case with a transposition cipher. In a transposition cipher, the encryption and decryption functions specify an algorithm for rearranging the elements (in this case letters) of a message to produce a ciphertext, depending on the key which is being used. When using a transposition cipher, the letters of the message are permuted to produce a ciphertext and then unshuffled during decryption to recover the message.

8.1 Normal Operation

The columnar transposition cipher operates by writing the text of a message underneath the key which is to be used in as many rows as are necessary so that each letter of the message appears directly below a letter of the key. The relative alphabetic

order of the key letters induces a permutation on the columns of message letters. The ciphertext is generated by reading off the columns of message letters in the order of the relative order of the key letters. An example is provided in Table 8.1 using the message "this is an example" and the key "key". When the columns are read off in the proper order, the resulting ciphertext is "hinaltsaxpiseme".

To decrypt a ciphertext generated by the columnar transposition cipher, the recipient of the ciphertext divides the length of the ciphertext by the number of letters in the key. If the number of ciphertext letters isn't divisible by the key length, the decrypter knows that some of the later columns will be shorter by one letter than the previous ones. Using division with remainder to divide the length of the ciphertext by the length of the key the decrypter discovers how many columns must be one letter shorter than the others. Since the decrypter knows the key word, and thus the correct order of the columns, they know which columns must be shorter. The ciphertext letters are written in the proper quantities underneath the key in the order dictated by the relative order of the key letters. Once this is done, the decrypter can recover the message by reading the letters off in rows. For example, consider the ciphertext "emsettsgsea" and the key "key". In this case the decrypter knows that the key is of length 3, and divides the length of the ciphertext (11) by 3 which results in 3 with a remainder of 2 indicating that the first two columns will be of length 4 and the final column will be of length 3. Since the key is "key", the first 4 letters of the ciphertext ("emse") become the middle column. The next 4 letters ("ttsg") become the first column and the final 3 letters ("sea") become the last column. Reading the rows of the resulting matrix results in the message "testmessage", as shown in Table 8.2 [19].

Table 8.1: Columnar Transposition Cipher Encryption Example.

Key	k	e	y
Relative Order	2	1	3
Message	t	h	i
	s	i	s
	a	n	e
	x	a	m
	p	l	e

Table 8.2: Columnar Transposition Cipher Decryption Example.

Key	k	e	y
Relative Order	2	1	3
Message	t	e	s
	t	m	e
	s	s	a
	g	e	

8.2 Classical Cryptanalysis

Classical cryptanalysis of the columnar transposition cipher is quite unlike that of the substitution ciphers presented earlier. One of the most crucial differences between a ciphertext produced by a transposition cipher and a ciphertext produced by a substitution cipher is that the ϕ-statistic is not useful for cryptanalysis of a transposition cipher-produced ciphertext. Since a transposition cipher dictates the shuffling of the plaintext letters to form the ciphertext, all of the letters which appear in the plaintext appear in the ciphertext in the same frequencies. Since a transposition cipher maintains the frequencies of the characters appearing in the message during encryption, the ϕ-value for the ciphertext is the same as it is for the plaintext, and thus doesn't bring the cryptanalyst any closer to discovering any secret information.

The cryptanalysis of the columnar transposition cipher relies on the cryptanalyst

Table 8.3: Encryption Possibilities With Key length=2.

Option 1		Option 2	
t	h	*	t
i	s	h	i
*	*	s	*

having some knowledge of the contents of the original message. The length of this known text must be more than that of the key if it is to be of any use. Suppose that a cryptanalyst is faced with the ciphertext shown in Table 8.1, "hinaltsaxpiseme", and knows that the word "this" appears somewhere in the plaintext. The first task of the cryptanalyst is to identify the length of the key used to encrypt the message. If the key used were of length 2, then there are 2 possibilities as to how the word "this" was encrypted. These possibilities are shown in Table 8.3, where * indicates an unknown character preceding or following the known text. Reading the letters off vertically as one would during encryption, using either *Option 1* or *Option 2* would result in the digraphs "ti" and "hs" appearing somewhere in the ciphertext. Since neither of these digraphs appear in the ciphertext the cryptanalyst can conclude that the length of the key being used was not 2. Consider the possibility that the length of the key used to produce the ciphertext was 4 or more. In this case the technique shown above will not give the cryptanalyst any information at all. Since the length of the known text is not more than the length of the key, the above technique won't yield any digraphs for the cryptanalyst to locate to verify the guess at the key length. On the other extreme, if the length of the known text is much longer than the key, the above technique will yield longer text segments for the cryptanalyst to use to verify their guess. The longer the text segment, the higher the cryptanalyst's certainty

Table 8.4: Encryption Possibilities With Key length=3.

Option 1			Option 2			Option 3		
t	h	i	*	t	h	*	*	t
s	*	*	i	s	*	h	i	s

that the guess is correct.

The only remaining possibility for the cryptanalyst given the example in question is a key length of 3. In this case, there are 3 possibilities for the encryption of the word "this". These are shown in Table 8.4. Again, when reading the columns as one would when encrypting a message, it is apparent that in either of the three cases the digraph "ts" must appear in the ciphertext. The digraph "ts" does in fact appear in the ciphertext, suggesting that the key used to produce this ciphertext was in fact three letters in length.

Now that the cryptanalyst has identified the length of the key, the cryptanalyst must identify the correct relative order of the letters in the key. The key letters themselves are unimportant and unrecoverable, only the relative order of the letters in the key word are needed to decrypt the message. Dividing the ciphertext into three equal-length segments yields the columns "hinal", "tsaxp" and "iseme". These columns can be arranged in 6 possible ways, yielding 6 different decryptions of the ciphertext. Three of these possibilities are shown in Table 8.5. The last of these possibilities, *Option 3*, yields English text while the others do not, so *Option 3* is the correct decryption. As the length of the key increases, the number of permutations of the columns increases more than exponentially [30], resulting in a quickly growing number of cases the cryptanalyst must check to recover the message. The cryptanalyst will likely not need to check all of the possibilities, but only those which

Table 8.5: Decryption Possibilities

Option 1			Option 2			Option 3		
1	2	3	1	3	2	2	1	3
h	t	i	h	i	t	t	h	i
i	s	s	i	s	s	s	i	s
n	a	e	n	e	a	a	n	e
a	x	m	a	m	x	x	a	m
l	p	e	l	e	p	p	l	e

correctly recover the known text the cryptanalyst used to establish the number of columns earlier in the process. It is also the case that partially correct permutations of the columns will result in text which is almost English, enabling the cryptanalyst to build the correct solution by improving on partially correct solutions.

8.3 Cryptanalysis Using Genetic Algorithms

Since the parameter choices used thus far have shown themselves effective at cryptanalyzing Vigenère-type ciphertexts, these choices seemed a logical starting point for the columnar transposition cipher. As can be seen in Figure 8.1, the genetic algorithm is capable of decrypting a columnar transposition ciphertext quite quickly using any combination of mutation operators; thus, the failure rates for all mutation operator combinations are 0. The most effective combination of mutation operators is the CF-CO combination, correctly decrypting the ciphertext in an average of 16.84 20-individual generations, or after 336 decryptions. Results for all combinations of mutation operators are shown in Figure 8.2. The genetic algorithm always succeeds in decrypting the ciphertext correctly for reasons discussed in Section 8.4.

Key lengths greater than 4 pose serious problems for decryption using genetic

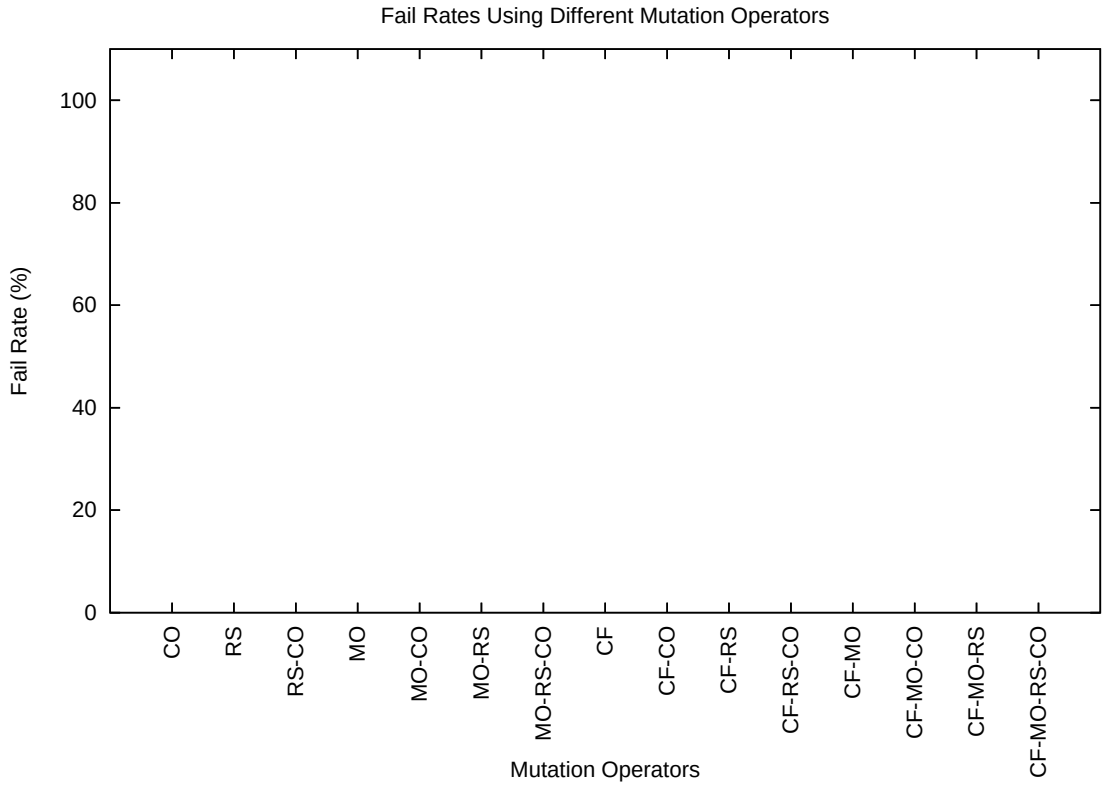

Figure 8.1: Columnar Transposition Fail Rates Using Different Mutation Operators

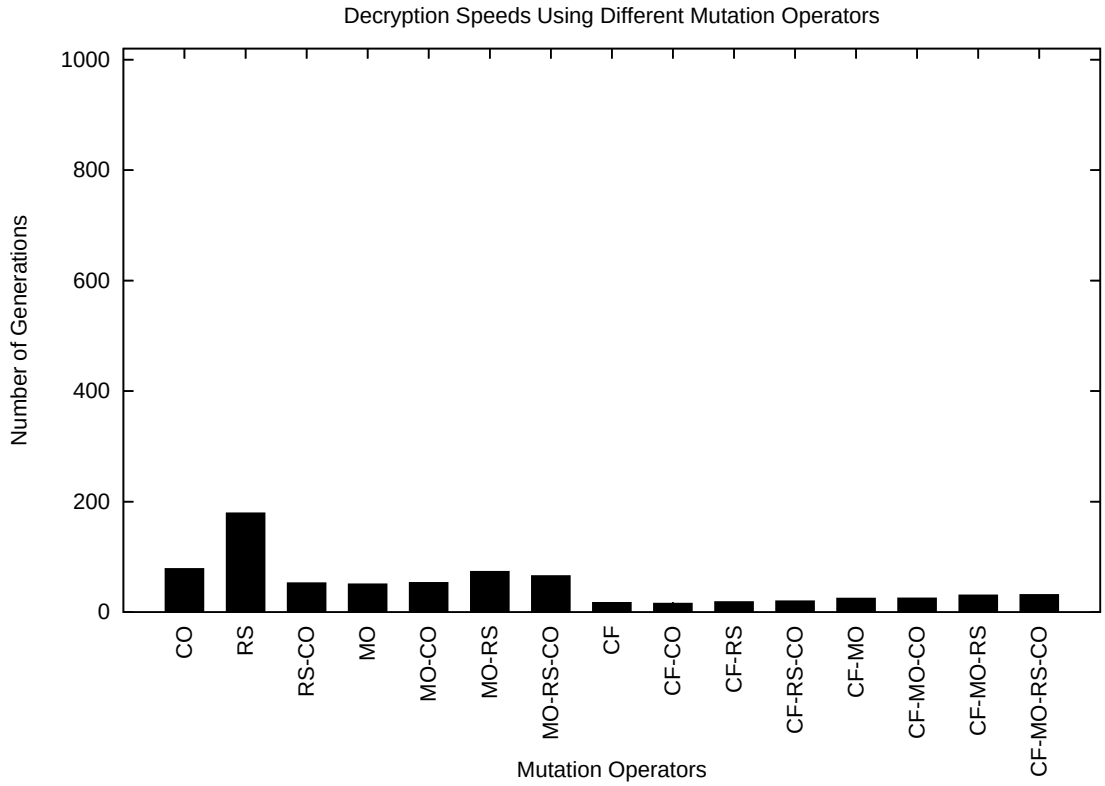

Figure 8.2: Columnar Transposition Decryption Speeds Using Different Mutation Operators

algorithms and the GA seems incapable of finding a correct key when the size of the key used is larger than 7. Figure 8.3 shows how quickly GA-based cryptanalysis falters as the key length grows when using the columnar transposition cipher. Potential reasons for this are discussed in Section 8.4. Interestingly, we see another large dip in the failure rate and number of generations required when the key length is 7. For some reason it is easier for the genetic algorithm to cryptanalyze a columnar transposition cipher-encrypted message with key length 7 than key length 6. The cause of this anomalous data point is not clear.

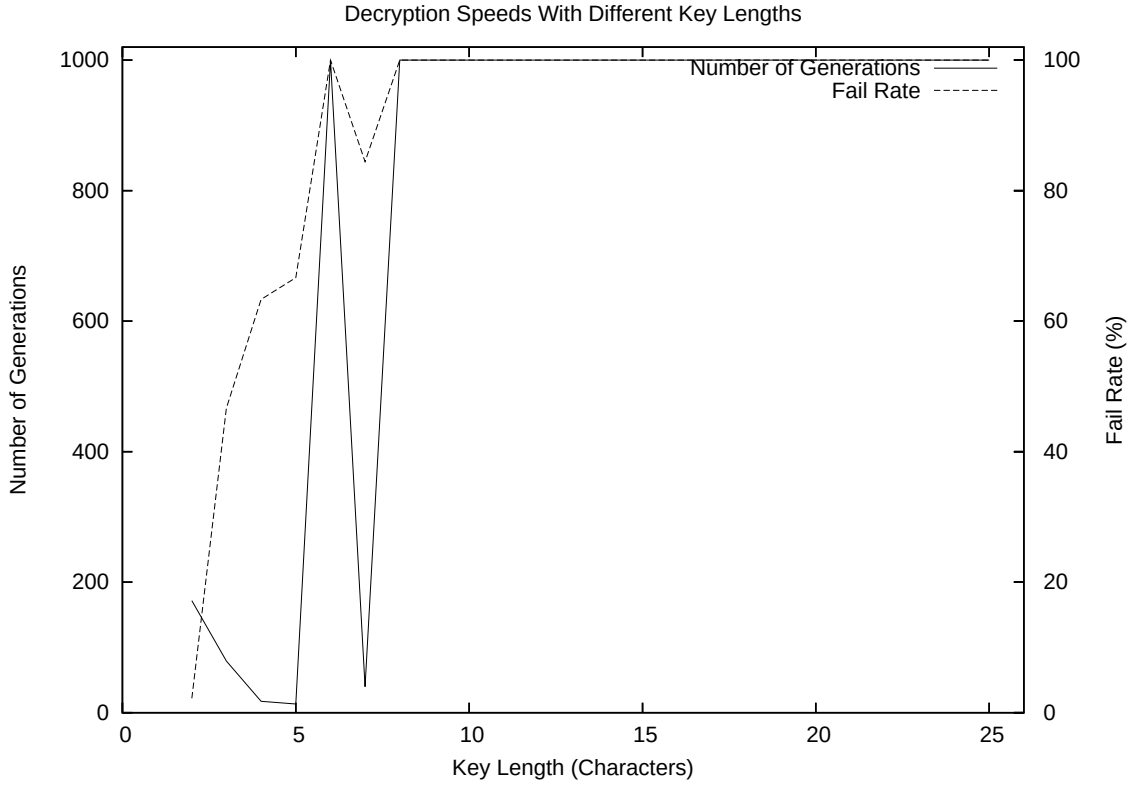

Figure 8.3: Columnar Transposition Decryption Speeds With Different Key Lengths

8.4 Discussion

Like the mixed Vigenère cipher, the columnar transposition cipher requires the cryptanalyst to have some information about the message which yielded the ciphertext being analyzed. When one examines the key space for the columnar transposition cipher in the same context as was done with the Vigenère-type cryptosystems, one obtains a falsely optimistic view of the security of the columnar transposition cipher. The reason for this is that the number of keys in the key space is not equal to the number of different strings obtainable using a set number of letters and set alphabet, it is instead equal to the number of permutations of n elements where n is the length of the key. Consider the keys "aky" and "blz". Although these keys don't share any letters between them and appear distinct, they impose the same permutation on the columns used for columnar transposition encipherment, and are therefore in essence the same key. The number of 5-letter Vigenère keys is shown in Equation 8.1 and the number of 5-letter columnar transposition keys is shown in Equation 8.2.

$$|K|_{Vigenère} = 26^5 = 11881376 \tag{8.1}$$

$$|K|_{CT} = 5! = 120 \tag{8.2}$$

This false perception of the key space and its size is possibly to blame for the relative ineffectiveness of the genetic algorithm for cryptanalysis of the columnar transposition cipher. The genetic algorithm examines keys in the context of character strings, not permutations, so it is likely that although the genetic algorithm makes changes to keys through the use of mutation operators, it doesn't get any useful feedback in the form of varying outputs from the fitness function. Small changes to a key will

usually not change the permutation induced by the resulting character string, making the genetic algorithm "run on the spot" and not achieve much progress despite numerous attempts to mutate high-fitness keys.

It was our belief that a completely different genetic algorithm implementation would be needed to be most effective at decrypting columnar transposition ciphertexts, one which evolves permutations directly and not character strings which induce permutations. It is our belief that this extra layer of abstraction hampers the performance of the genetic algorithm despite the fact that the columnar transposition key space is much smaller than that of the Vigenère-type cryptosystems, even when the key length is the same.

Finally, the issue of fitness function should be discussed. As was described in Section 8.2, a columnar transposition ciphertext should have the same ϕ-value as the message which was used to create it. For this reason, the ϕ-statistic based fitness function which was used to help produce fitness values for keys doesn't return any useful feedback. The only useful feedback is provided by the Hamming-distance based fitness function which is quite similar to the known-text attack which is normally used in the cryptanalysis of the columnar transposition cipher. The use of this fitness function is probably the reason that any success at all was seen during the course of our experiments on this cryptosystem.

8.5 Cryptanalysis Using Permutation-Based Genetic Algorithms

As mentioned previously, it was believed that a permutation-based genetic algorithm would perform better than the character string based genetic algorithm previously tested. In order to implement this permutation-based genetic algorithm, a number of refinements needed to be made.

8.5.1 Data Type

The original genetic algorithm implementation was designed to evolve character strings with few restrictions placed on them. The permutation-based GA (PGA) would need to include modifications to accommodate the evolution of permutations. The permutations were internally represented as vectors of the numbers from 0 to $n-1$ for an n-letter key. The vectors representing the permutations could only be altered using an interface which only permitted modifications that preserved the property that the modified vector would still represent a permutation.

The use of permutations was limited to decryption operations of the columnar transposition cipher, this allows users of the cipher to still encrypt messages using the more intuitive character-string key representation and the PGA to use permutations as decryption keys.

8.5.2 Mutation Operators

Mutation operators which were valid for character strings were not necessarily valid mutation operators for permutations as they may introduce duplicate entries, or any

number of other problematic combinations into the new permutation data type. New mutation operators were developed which would mutate permutations and ensured that any resulting mutant keys were still valid permutations.

A swap mutation operator would randomly select two elements in a permutation and swap them, resulting in a new permutation with the randomly selected elements swapped. An example of the application of such a mutation operator would be the permutation $(0, 1, 2, 3, 4, 5)$ being mutated to $(5, 1, 2, 3, 4, 0)$, where the first and last elements where interchanged.

Another mutation operator was implemented which would increase the size of a permutation by inserting the next required integer at a random position within the permutation. An example of this operator would be $(3, 1, 0, 2)$ being mutated to $(3, 1, 4, 0, 2)$, where the next required integer (4) was inserted into the third position of the original permutation.

Finally, a mutation operator which could decrease the size of a permutation was implemented. This operator would randomly select a position in a permutation and remove the integer found there. Unless this integer was the highest one in the permutation, it would be necessary to reduce each of the remaining integer values by one when their values were higher than the removed value in order to preserve the permutation data structure. An example of the application of this operator would be $(3, 1, 4, 0, 2)$ being mutated to $(2, 3, 0, 1)$. In this example, the second element (1) was removed from the permutation, resulting in a permutation one element shorter and necessitating the reduction of all of the integers present in the permutation (except for 0) by one.

The effectiveness of various combinations of these mutation operators was not

explored. Data produced by the PGA was done so with the algorithm using all three new mutation operators.

8.5.3 Fitness Functions

As previously mentioned, the ϕ-statistic based fitness function which is quite effective for cryptanalysis of substitution ciphers has no effect on this transposition cipher. Since the columnar transposition cipher simply rearranges the letters of the plaintext to form the ciphertext, the ϕ-value of the ciphertext is always the same as the ϕ-value of the plaintext. The ϕ-statistic based fitness function is therefore not used in the PGA.

The Hamming Distance-based fitness function still has a role to play with transposition ciphers and therefore remains unchanged in the PGA. The use of only the Hamming Distance-based fitness function was quite effective for the cryptanalysis of the columnar transposition cipher; however, it has one major caveat. In order for the PGA to find a correct decryption using only the Hamming distance-based fitness function, there must be at least as much known text as there are elements in the permutation. If the decryption of the ciphertext uses more columns than there is known text, then any key with the columns where the known text is found in the correct positions will have the optimal fitness score. Consider the situation where the known text is only two characters long, but the actual key is a permutation of the integers up to 10. Any permutation with the two columns which contain the characters which make up the known text in the correct positions will receive a fitness value of 1.0. This doesn't come close to correctly deciphering the ciphertext.

A second fitness function was therefore devised to compensate for the PGA's

unreliability when deciphering messages whose keys are longer than the known text. This second fitness function allows the programmer to specify a sequence of letters and the frequency with which the sequence should occur in the message. The fitness value of a key is scaled linearly from 0 to 1 as the incidence of the character sequence in the decryption under the key in question increases from 0 to the frequency specified by the programmer. The fitness value of the key is scaled from 1 back down to 0 as the frequency of the occurrence of the character sequence grows from the frequency specified to the double the specified frequency. This new fitness function allows the cryptanalyst to exploit additional knowledge of the English language, such as how often the character sequence "the" appears in English text. The cryptanalyst would supply the fitness function with the word "the", and specify that it should occur approximately 7 times in every 500 letters. The text being analyzed is 522 characters long and the word "the" appears 6 times, this would yield a fitness value of approximately 0.82. With the inclusion of this fitness function it is possible for the PGA to decrypt messages with less known text than key material.

8.5.4 Results

Where the original GA was able to reliably decipher ciphertexts with keys up to length 4, the figure shows that the PGA was able to decipher ciphertexts with keys of length up to and including 13 quickly and reliably. All parameter settings were identical to those used with the original GA, save for the use of the permutation-based mutation operators discussed in Section 8.5.2 and the new fitness function discussed in Section 8.5.3. This success can likely be attributed to two factors: the use of permutations as a base data type for the GA, and the additional fitness function.

The use of permutations as the base data type for the GA gives the GA better feedback after having made a change to a key. Changes made to a character string key by the original GA usually didn't result in a change in fitness value because of the number of different character strings which induce the same permutation. Changes being made to the permutation itself virtually guarantees a different fitness value for an altered key.

The additional fitness function was instrumental in the success of the PGA because it allows the algorithm to push its abilities beyond the amount of known text by favoring keys which resulted in decryptions with correct frequencies of common English letter combinations. Figure 8.4 shows a comparison of decryption speeds between the two GA implementations. Figure 8.5 shows a comparison of failure rates for the two GA implementations. In the case of the columnar transposition cipher, the tailor-built genetic algorithm outperforms the genetic algorithm which showed itself to be very effective against substitution ciphers by a wide margin.

The trends seen in Figures 8.4 and 8.4 are very different to those seen when applying genetic algorithms to substitution ciphers. When confronted by messages encrypted with substitution ciphers, the number of generations required by the genetic algorithm did not increase with the length of the key, while the failure rate did. In the case of the columnar transposition cipher both of these metrics increase with the key length. This fundamental difference in the behaviour of the genetic algorithm can likely be attributed to the very different behaviour of the columnar transposition cipher when compared to the substitution ciphers studied earlier.

The use of permutations as opposed to character strings drastically reduces the size of the key space when considering like-length keys. Figure 8.6 shows the differ-

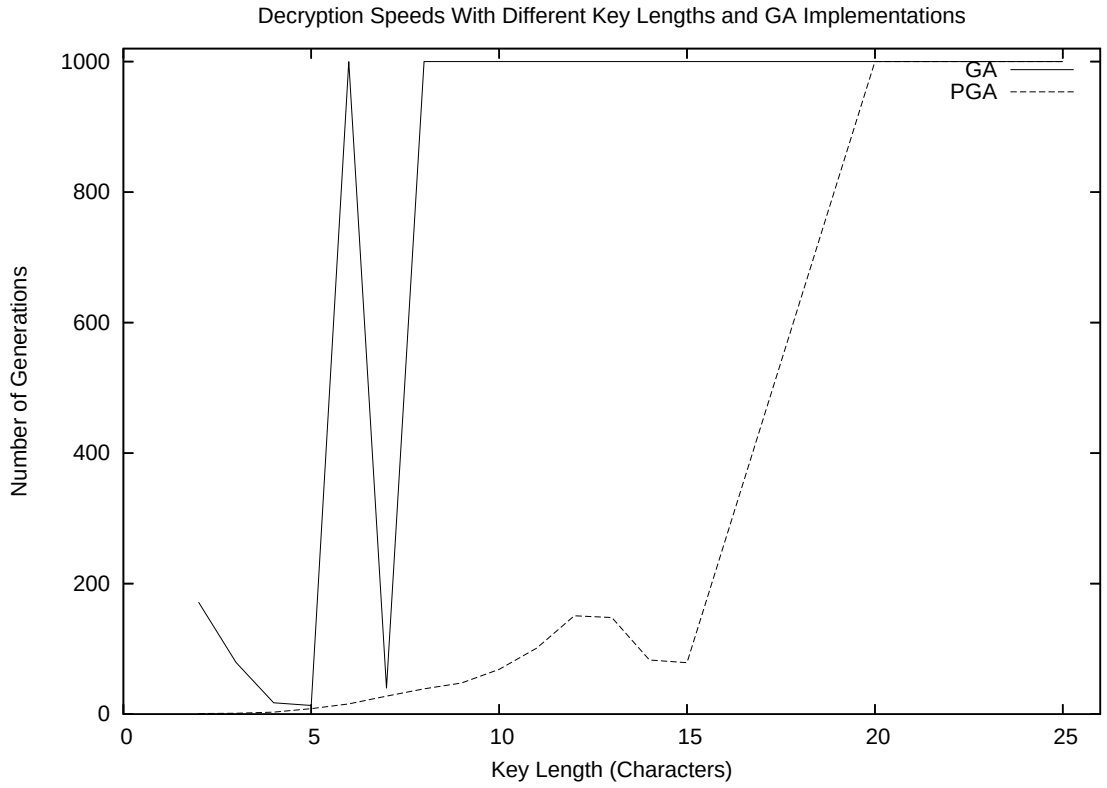

Figure 8.4: Columnar Transposition Decryption Speeds With Different Key Lengths and GA Implementations

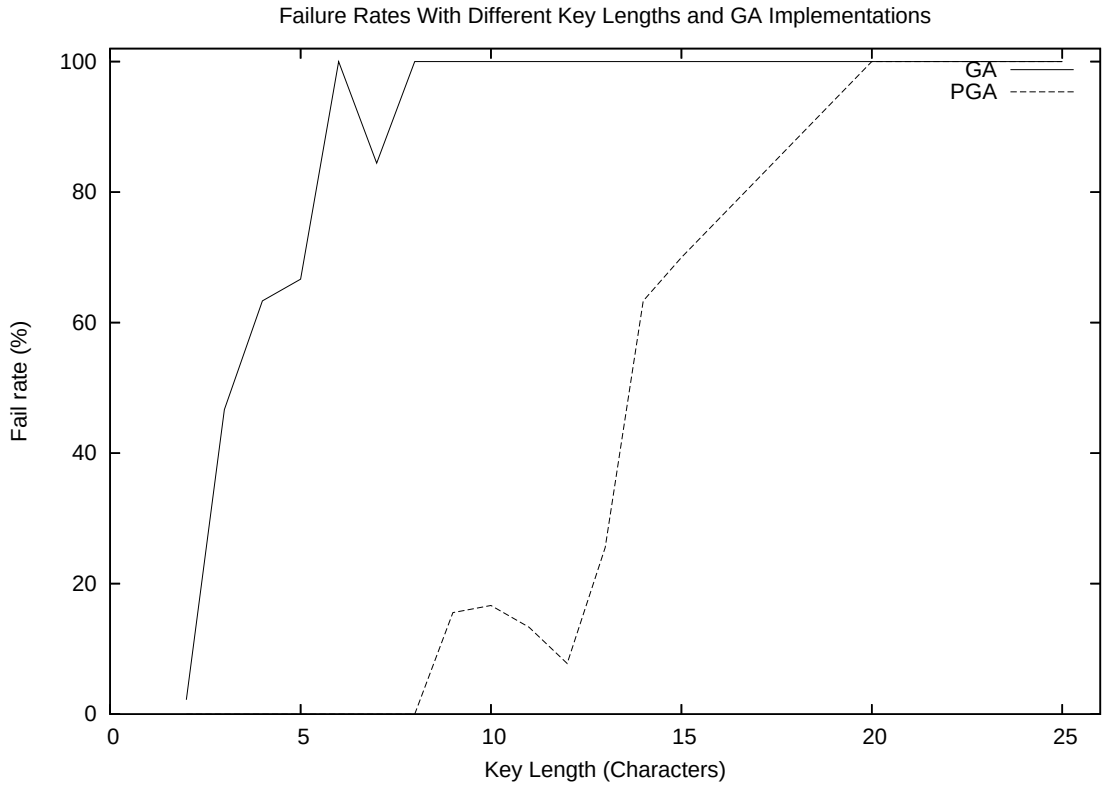

Figure 8.5: Columnar Transposition Fail Rates With Different Key Lengths and GA Implementations

ence in size of key space explored by the two different GA implementations.

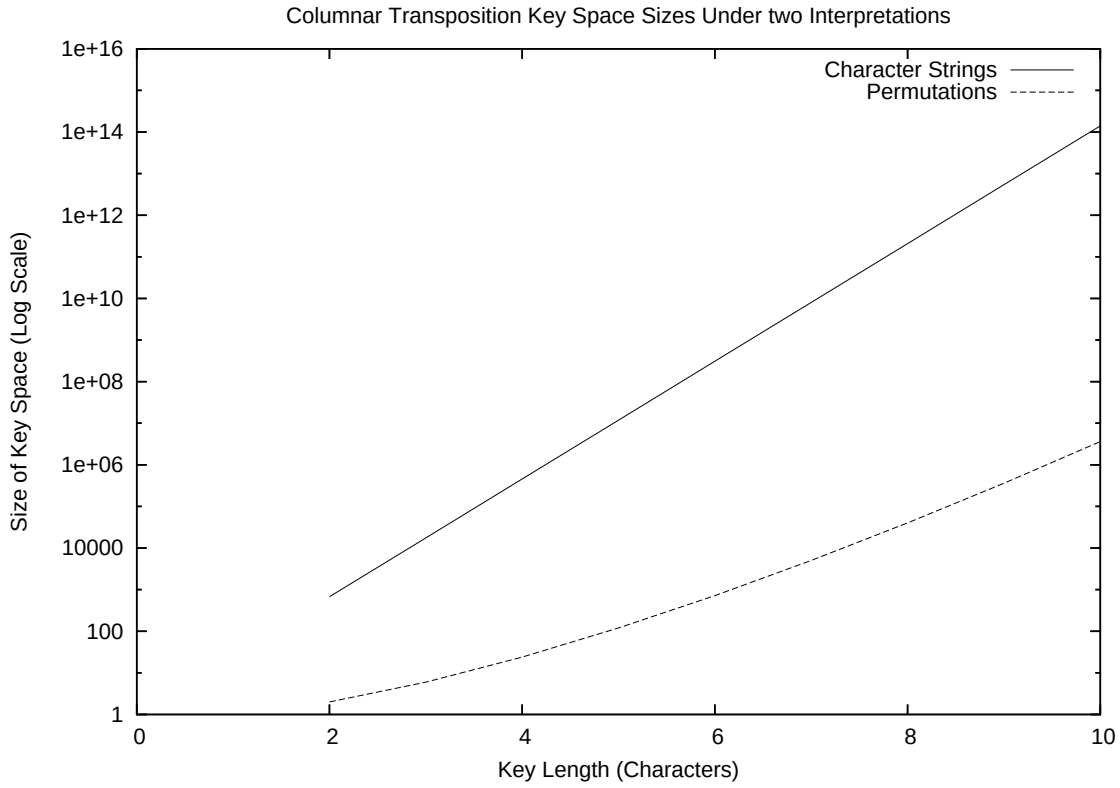

Figure 8.6: Columnar Transposition Key Space Sizes Under Two Interpretations

Chapter 9

DES

The Data Encryption Standard, or DES, is the first modern cipher that our genetic algorithm was tested on. DES's development was solicited by the US Government in order for companies to have access to a secure, standardized cryptographic algorithm. IBM submitted a cipher called Lucifer to the National Bureau of Standards which would later be renamed to NIST, and the National Security Agency (NSA) for review as a potential candidate. After some minor modifications at the behest of the NSA, Lucifer became the Data Encryption Standard. DES was used extensively from 1977 until constant improvements in conventional computers rendered it too insecure to use for sensitive data in the late 1990's. The full DES algorithm is specified in [20].

DES is quite different from the classical cryptosystems seen in previous chapters because instead of being comprised of a single encryption/decryption technique (substitution or transposition), DES utilizes a number of techniques to secure information, one applied after another. This composition of cryptographic techniques is what identifies DES as a *product cipher*. One of the advantages of DES is that encryption and decryption are virtually the same process, the combination of substitutions and permutations which are applied to encrypt data are simply repeated in reverse in order to decrypt data.

9.1 Normal Operation

Details of the DES algorithm can be found in [20] and additional explanations can be found in [29]. DES is a block cipher and thus operates on blocks of bits. The algorithm encrypts/decrypts 64-bit blocks of data, with a 56-bit key, one at a time. A message of length b bits requires $\lceil b/64 \rceil$ applications of the DES algorithm to encrypt (in Electronic Code Book mode).

DES is a *round*-based cryptosystem, so the same process is repeated a number of times (in the case of DES, 16) to encrypt or decrypt a block of data. In each round a different subset of the key material is incorporated into the encryption process, so a key schedule is needed to create 16 different 48-bit *round keys* from the single, 56-bit DES key. This is accomplished by first permuting the DES key according to a fixed permutation called *Permuted Choice 1*. The permuted key is then split into two 28-bit blocks, and each half is circularly rotated a set number of bits to the left and the two halves are rejoined. The result from this process is permuted again using a second fixed permutation (*Permuted Choice 2*) which also reduces the number of bits from 56 to 48. This whole process is repeated 15 more times with different circular shifts in order to produce the 16 round keys required by the DES algorithm.

The application of the DES algorithm can be decomposed into three sections, the Initial Permutation (IP), the 16 rounds, and the Final Permutation IP^{-1}. IP is just a permutation which rearranges the bits in the block of data being processed. IP^{-1} is the inverse of IP, and undoes the permutation applied by IP, so applying IP and IP^{-1} immediately afterwards would yield the same block of bits one started

with.

In each of the 16 rounds of DES the data block is first split into a right half and a left half. The right half is expanded to 48 bits using the *E-Bit Selection Table*, which also permutes that half of the data block. The result is XOR'ed with one of the round keys which were produced before the encryption process began. The result of this XOR is then divided up into 8 blocks of 6 bits each, which are used as indices into look-up tables called *S-Boxes*, or substitution boxes. The 48-bit block is exchanged for a different 48-bit block by using the 8 S-Boxes to dictate a series of substitutions. The result of the substitutions is then permuted again and reduced back down to 32 bits using a different permutation before being XOR'ed with the left half of the original data block and the block is reassembled. This process is repeated 15 more times to form the 16 rounds of DES. A different round key is used in each round of the algorithm, but the permutations and S-Boxes used are the same for each round.

After applying the 16 rounds, IP^{-1} is applied and this final result is the ciphertext of the original data block. Decryption is virtually the same as encryption. The same algorithm is applied, only the round keys and S-Boxes are used in the reverse order during the 16 rounds.

9.2 Classical Cryptanalysis

Although great efforts were applied to the cryptanalysis of DES, partially due to skepticism over the impartial involvement of the NSA in the selection and modification process of the standard, no major flaws or shortcuts have been found to

date in the DES algorithm. The main motivation for replacing DES with a new cryptographic standard, the Advanced Encryption Standard (AES) was that since DES required keys of only 56 bits in length, exhaustive search started to become a viable means of casual cryptanalysis of DES-encrypted messages. RSA issued a number of challenges concerning the cryptanalysis of DES, the result of which was the conclusion that a special purpose machine could be built for around $1 million to cryptanalyze DES encrypted messages using exhaustive search, requiring only 3.5 hours to complete. It is now believed that a modern computer can cryptanalyze a DES ciphertext in a matter of hours. A new encryption standard, AES, addresses this concern by allowing for different, larger key sizes, rendering exhaustive search an infeasible technique for cryptanalysis (for now). DES is still used in practice, but only in a form known as 3DES, where a message in encrypted 3 times with two or three different keys (depending on the implementation).

9.3 Cryptanalysis Using Genetic Algorithms

In order to test the resistance of DES to attack by optimization algorithms, a special implementation of DES was developed for this research, where the number of rounds used by the algorithm could be manipulated. The motivation behind this approach was the intuition that DES composed of only one or two rounds would be more vulnerable to attack than the full 16-round DES. While this was indeed shown to be the case using a technique called differential cryptanalysis [27], DES proved to be resistant to attack by genetic algorithms using the same mutation operators and fitness functions as with the substitution ciphers, even when using only one round

DES. One possible reason for the failure of the genetic algorithm is that our GA was designed to work on character string based cryptosystems, so in order to apply the GA to DES, an interface had to be built for DES so that it could accept strings as keys instead of blocks of bits. The application of this interface added a layer of "artificiality" between the cryptosystem and the genetic algorithm because manipulating keys using genetic operators on the level of characters greatly reduced granularity with which the GA could make changes to the keys as it ran. The ideal situation would be to allow the GA to make changes to keys on the bit level, instead of only what is possible on the character level. The need to implement a modified genetic algorithm became pressing not only from an aesthetic point of view, but from necessity when tests illustrated that the character-string based genetic algorithm was not able to decrypt DES-encrypted messages with any success.

To attempt to push this research further, a modified genetic algorithm was produced which would evolve bit strings instead of character strings. The basic function of the new GA is the same as that for the original GA except that keys are now represented as strings of bits instead of strings of characters, and the mutation operators now implement the cross-over and point-mutations seen before but on the bit level. The fitness functions used with this genetic algorithm implementation were the same ones as described in Section 2.2.3. To the credit of the creators of DES, this modified genetic algorithm also proved ineffective for cryptanalysis of DES-encrypted messages. As a final effort, it was decided that one more nature-inspired optimization algorithm would be applied to DES in order to probe its weaknesses or bolster the claims of its security.

9.4 Cryptanalysis Using Particle Swarm Optimization

Due to the utter failure of the application of a genetic algorithm to the cryptanalysis of DES, another nature-inspired optimization algorithm, Particle Swarm Optimization (PSO), was applied to the problem. As discussed in Section 2.3, to use PSO, one must interpret the fitness landscape as Euclidean n-space. Particles then navigate this n-space according to a set locomotion/swarming algorithm with two parameter settings, c_1 and c_2. c_1 controls the individualistic tendencies of the particles, or the extent to which they explore the fitness landscape according to their own personal experience, or tend towards $pBest$. c_2 controls the social tendency of the particles, or the extent to which they tend towards $gBest$, the best location found by any particle in the population. PSO was applied to a version of DES with greatly reduced security, where keys consisted of only 7 characters (to form the 56-bit DES key), but only the first two characters were varied, yielding a 2-dimensional fitness landscape. The number of rounds used by the DES algorithm was also varied between 1 and 16. Even when using only one round, no combination of parameters c_1 and c_2 could be found so that the particle swarm could reliably find the correct 2-letter key in a reasonable amount of time.

The magnitude of the parameters c_1 and c_2 control the step size for each particle at every iteration of the algorithm. Since the space being searched for most of the classical ciphers is quite small in any given dimension, but of high dimensionality, c_1 and c_2 values above 5 resulted in very large steps being taken across the search space at every iteration. These steps were too large to take advantage of any gradients in the fitness landscape because they simply "stepped over" the interesting portions

of the landscape. Much lower values for c_1 and c_2 (between 0.1 and 2) held the most promise. While the magnitude of the parameters controlled the step size, their relative values are actually responsible for altering the search behaviour of the particles. A number of configurations were tested with c_1 and c_2 values between 0.1 and 2. Many configurations had c_1 weighted more heavily than c_2 while other configurations had c_2 weighted more heavily than c_1. Some configurations had the two parameters weighted relatively evenly, but no configuration was found to be effective against any of the classical ciphers, let alone DES.

9.5 Discussion

Reasons why GA's and PSO were not effective at deciphering DES-encrypted messages became apparent when the PSO algorithm was unsuccessfully applied to substitution ciphers like the Vigenère cipher, and a visualization engine was created in order to determine why our attempts to apply PSO to cryptanalysis were unsuccessful.

Although PSO was implemented for the cryptanalysis of DES, the implementation and associated visualization components were tested on the substitution ciphers discussed earlier to confirm the correct functioning of the PSO algorithm. Although the PSO implementation was behaving as expected, as evidenced by observation of the visualization components, the algorithm had little success in correctly cryptanalyzing ciphertexts produced by any cryptosystem. A wide variety of c_1 and c_2 parameters were selected, but all parameter choices resulted in the particles moving in multiple dimensions within a single iteration and often bypassing the correct

solution because of this more fluid search pattern.

It is our opinion that Particle Swarm Optimization is less versatile than a Genetic Algorithm because GA's allow the researcher two ways to make finding an optimal solution more tractable, while PSO only gives the researcher one dimension of freedom. In a genetic algorithm the researcher is free to allow the algorithm to visualize the fitness landscape in any way they choose, through the choice of fitness functions supplied to the algorithm. This is also the case with PSO, the fitness functions are supplied by the programmer, so the algorithm's view of the fitness landscape is controllable and versatile. Genetic algorithms; however, allow the researcher a freedom that PSO does not in that they allow the researcher to dictate the patterns in which the fitness landscape is explored through the researcher's choice of mutation operators. When using PSO, the researcher does not have this luxury because the locomotory patterns of the particles are largely fixed. Although the search patterns in PSO can be influenced by the researcher's choice of c_1 and c_2, the algorithm does not allow for the flexibility of search pattern provided by the genetic algorithm.

It also became apparent that the movement patterns of the particles in the PSO implementation were not conducive to cryptanalysis, while the mutation operators of the genetic algorithm were well matched to the problem. After implementing a visualization engine for the particle swarm optimizer, it also became apparent why neither algorithm had any success with DES.

Figures 9.1, 9.2 and 9.3 show two-dimensional fitness landscapes for the three classical substitution ciphers explored in previous chapters. The fitness landscapes shown are for two-letter keys only, so one axis represents the first letter in the key, and the second axis represents the second letter in the key. Thus, each square on

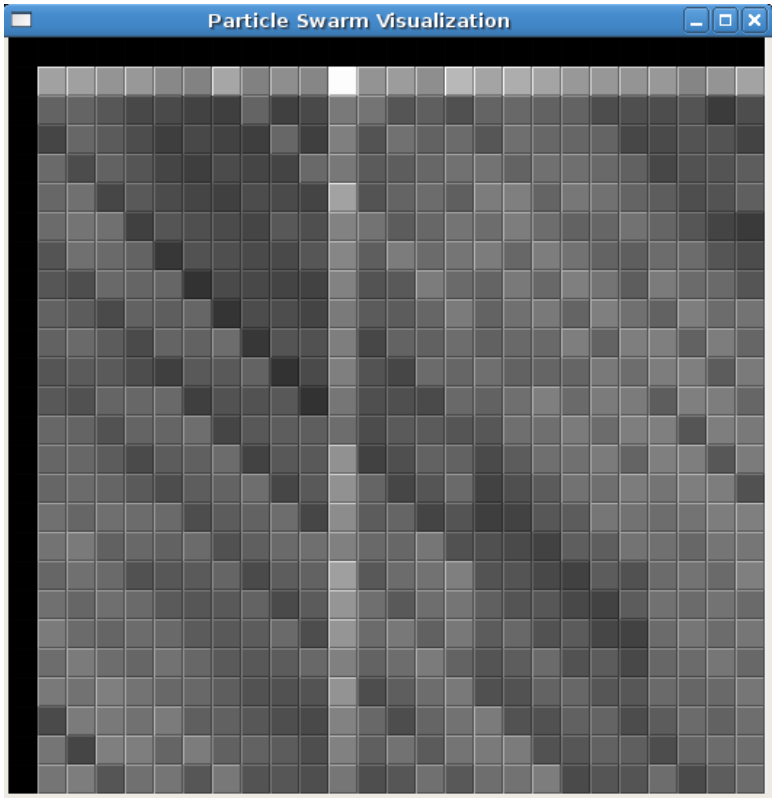

Figure 9.1: 2-Letter Vigenère Fitness Landscape

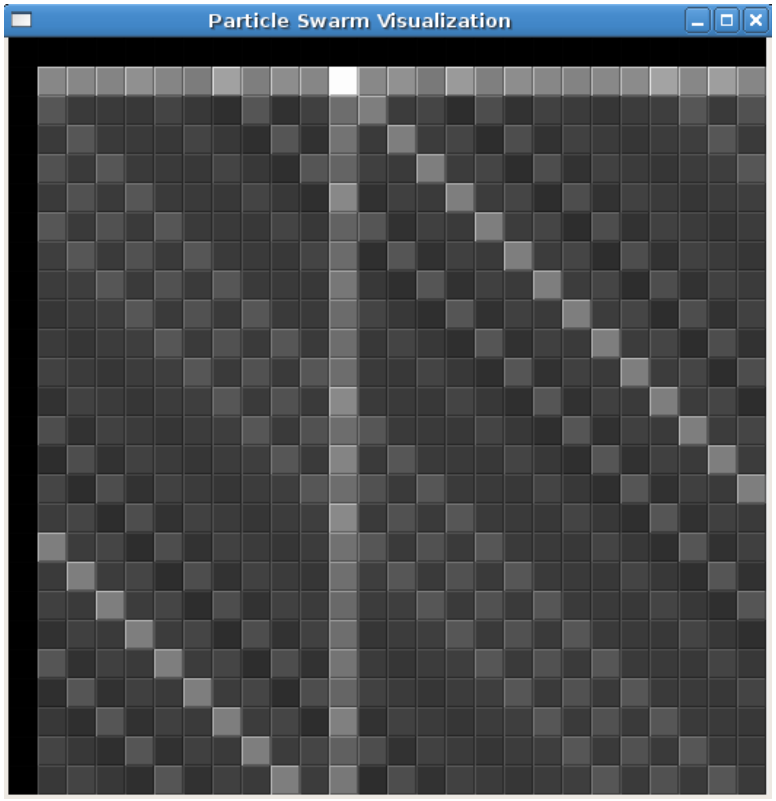

Figure 9.2: 2-Letter Mixed Vigenère Fitness Landscape

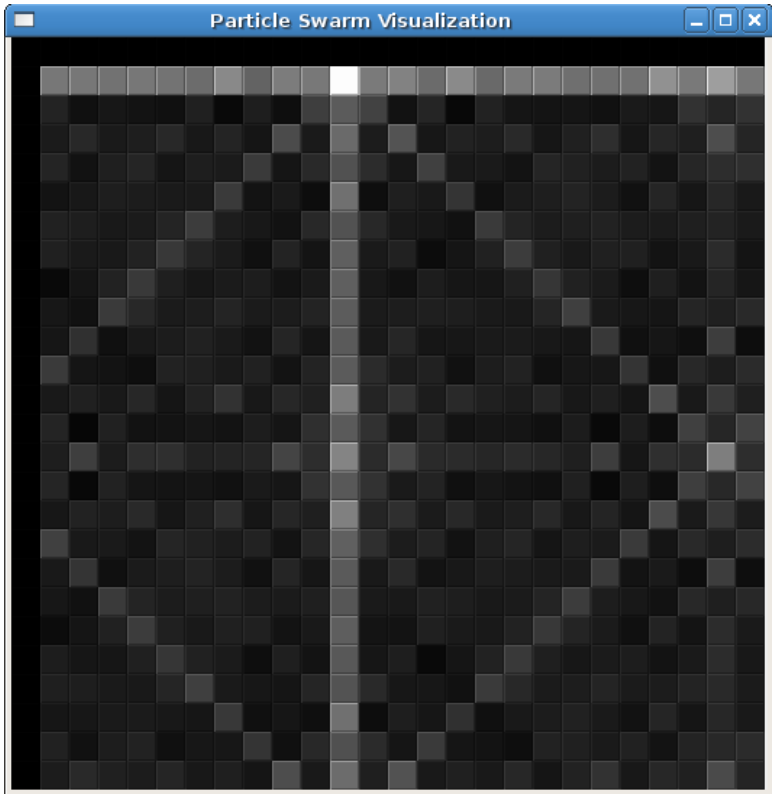

Figure 9.3: 2-Letter Autokey Fitness Landscape

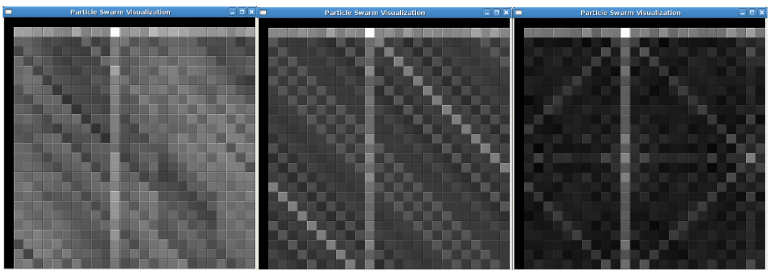

Figure 9.4: 2-Letter Substitution Cipher Fitness Landscapes

the grid represents one of the $26 \times 26 = 676$ possible two letter keys. The first row and the first column both represent the null characters, so keys along the $x - axis$ represent one-letter keys and keys along the $y - axis$ represent keys of length 0. Each square on the grid is coloured a shade of gray, ranging from black to white. The colours of the squares represent the fitness values of the keys represented by the squares in that a black square represents a key with a fitness value of 0, while a white square represents a key with the maximal fitness value of 1. In all cases, the correct key is "ka" which can be seen as the white square in the second row and twelfth column of each figure. As can be seen to varying degrees in the three figures, there are gradients of colour from black to white, these are very useful for visualizing the workings of optimization algorithms because they provide a visual basis for optimization – better keys tend to occur "close" to good keys, while keys with low fitness tend to occur in close proximity to other keys with low fitness. The Vigenère cipher seems to yield the most "optimizable" fitness landscape of the three as it has large areas of generally increasing fitness leading in the direction of the optimal solution. The Autokey cipher seems to yield the least optimizable fitness in the landscape as it has large areas of consistently low fitness where small changes to a key through the use of mutation operators of the swarming algorithm of PSO may not result in useful feedback in terms of increasing fitness. Vigenère, Mixed Vigenère and Autokey fitness landscapes are provided for side-by-side comparison in Figure 9.4.

Looking at these fitness landscapes also gives clues as to why the genetic algorithm worked so well in the cryptanalysis of these ciphers and PSO didn't work at all. The most useful fitness gradients in the landscape occur in a single dimension and this

is conducive to the pattern in which the GA searches the fitness landscape. The implemented mutation operators make small changes in a key, usually in only one letter of a key per generation, so very small changes are made to keys before feedback from those changes is examined by re-evaluating the fitness of the keys using fitness functions. This is not the case with PSO. In PSO particles move through the fitness landscape fluidly, which means that a particle may be geometrically very close to its original position after one iteration, but can subsequently move in all possible dimensions, so an apparently small change in the location of a particle could have very drastic effects on the particle's fitness. The movement patterns induced by the GA's mutation operators make it more likely that slightly altered keys have only slightly different fitness values.

Figures 9.5, 9.6, 9.7 and 9.8 show the fitness landscapes generated by 1, 2, 3, and 16-round DES. Fitness landscapes of 1, 2, 3, and 16-round DES are provided in a single figure in Figure 9.9. As can be seen by the reader, these fitness landscapes have a very different character than those of the substitution ciphers. There are few gradients from low to high fitness in any of the fitness landscapes, which makes navigating them in small steps difficult. It is also apparent in all but the 1-round DES that there is one global maximum in the fitness landscape where the correct key is located and no hints as to how to get there from any other locations in the fitness landscape. These properties make the success of either genetic algorithms or PSO highly unlikely. The seemingly random distribution of higher-fitness keys within the fitness landscape can be explained by the *avalanche criterion* which stipulates that a change of a single bit in either the message or the key used for encryption should result in half of the bits in the ciphertext being flipped [19]. Since most modern block

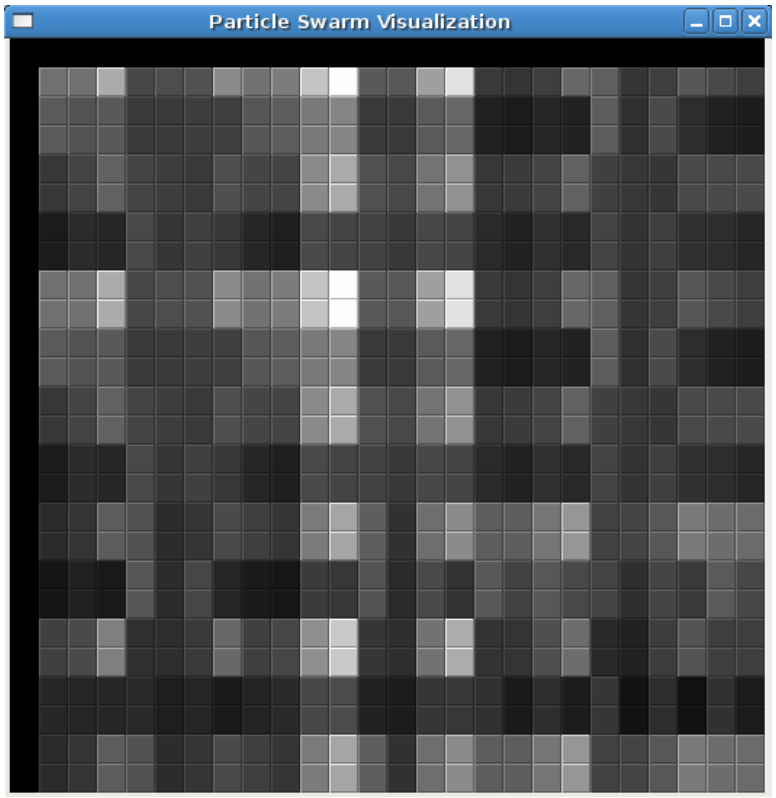

Figure 9.5: 1-Round DES Fitness Landscape

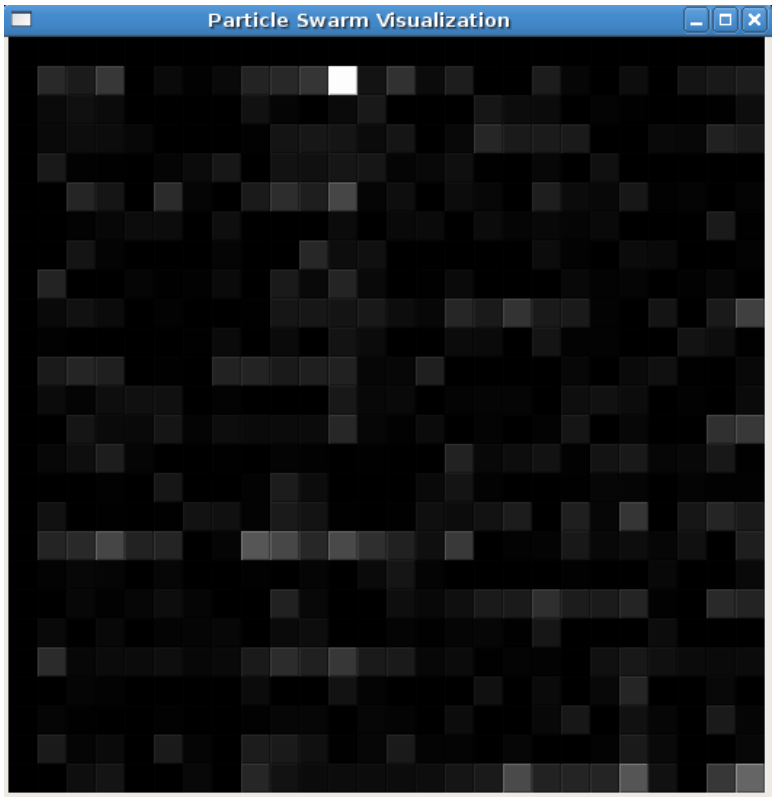

Figure 9.6: 2-Round DES Fitness Landscape

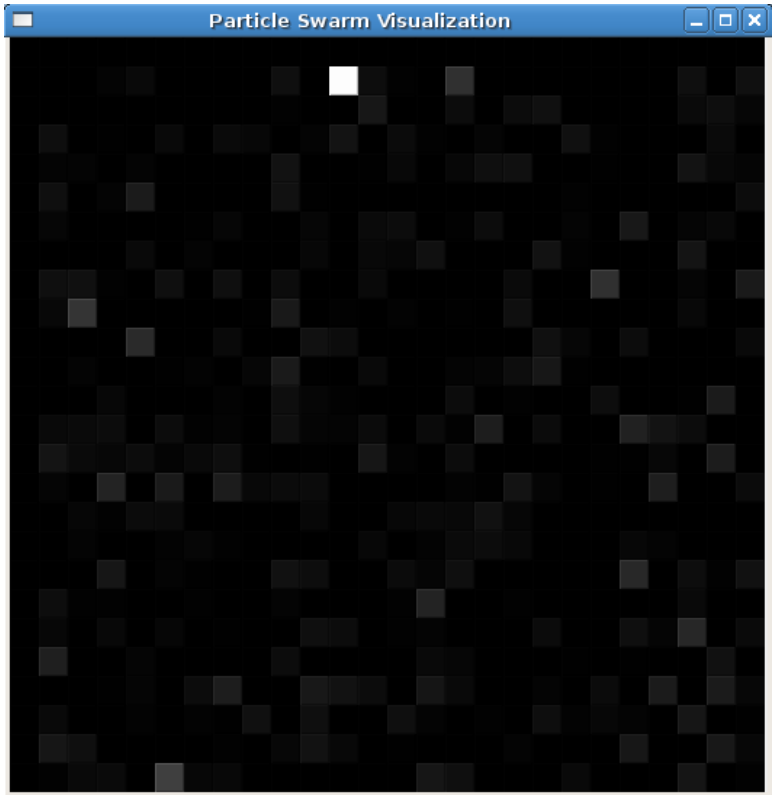

Figure 9.7: 3-Round DES Fitness Landscape

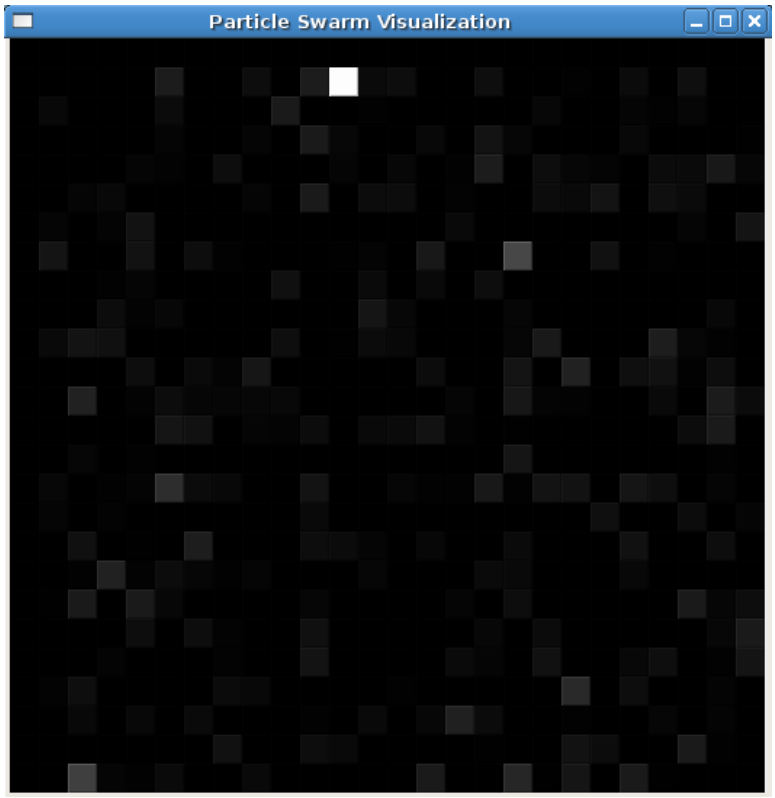

Figure 9.8: 16-Round DES Fitness Landscape

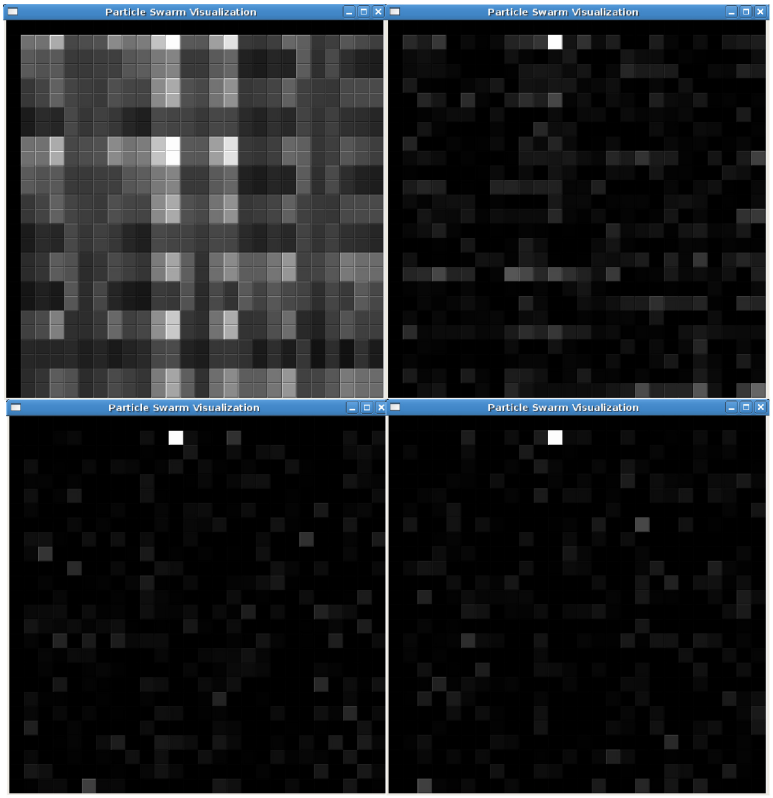

Figure 9.9: DES Fitness Landscapes

ciphers, including DES and AES, fulfill this avalanche criterion, it isn't surprising that there aren't large regions of high-fitness keys in the fitness landscapes produced by these ciphers. The avalanche criterion is discussed further as an explanation for the inability of genetic algorithms to cryptanalyze DES and AES enciphered ciphertexts in Section 11.

While it happens that our attempts at cryptanalysis of DES were unsuccessful using these techniques, that doesn't mean that it is impossible. It is conceivable that fitness functions could be developed which do produce gradients in DES fitness landscapes which would be optimizable, or mutation operators which navigate the fitness landscape in a manner which would make good solutions "close" to better solutions.

Chapter 10

AES

The Advanced Encryption Standard (AES) is a standardized version of the Rijndael cryptosystem. Like DES, AES is a block cipher which utilizes a combination of substitutions and permutations in order to achieve security. AES was developed and standardized specifically to replace the aging Data Encryption Standard (DES). As was mentioned in the previous chapter, one of the main problems with DES is that its operation only calls for a 56 bit key, making exhaustive search a more and more plausible avenue of attack as computers get faster. In order to mitigate this weakness, AES allows for much larger, and even variable key lengths. AES can be used with three different key lengths, each with a fixed plaintext block length of 128 bits. Key lengths of 128, 192 and 256 bits are specified by the standard. AES is based on arithmetic on polynomials over a finite field, but can be largely described in terms of S-Boxes and circular shifts.

10.1 Normal Operation

The full AES specification can be found in [21], with additional information in [29]. The operation of AES is in some ways quite similar to DES in that it uses a key expansion routine to create sub-keys for use in the different rounds or iterations of the algorithm. The key expansion routine uses a series of substitutions and circular shifts in order to create a round key for each round the algorithm will execute. Where

DES used 16 rounds, the number of rounds used by AES is variable and depends on the length of the key. 10 rounds are used with a 128 bit key, 12 rounds with a 192 bit key and 14 rounds with a 256 bit key.

AES is best described as a series of operations performed on a *state*. The state starts off as the data to be encrypted arranged in 4 columns of 32 bits each. The algorithm begins by adding the first round key to the state by using a simple *XOR* operation.

Next, all but the last of the rounds of the algorithm are executed. Each of these rounds consists of four steps, the first being a substitution step. This step involves using the values in the current state as indices into an S-Box and making the substitutions indicated in the S-Box in the state, much as was done in DES.

The second step in all but the last round is to shift the four rows of the state, the first row is not shifted (shifted by 0 bytes). The second row is shifted by one byte, the third by two and the last row by 3 bytes.

The third step in the rounds can be explained as interpreting each column in the state as a polynomial over the finite field $GF(2^8)$ and multiplying each column by the fixed polynomial $x^4 + 1$. This reversible process mixes the columns of the state.

The final step in each round is to add key material to the state. As was done before, the round key corresponding to the current round is XOR'ed with the state.

When all but the last round of the algorithm have executed, the final round is applied to the state. This final round is similar to the previous rounds in that all of the operations are the same as previously described except that the third column mixing step is not performed. Once this final abridged round is performed on the state, the result is the output of the algorithm, ie. the ciphertext.

Decryption of a block of AES-encrypted ciphertext is performed similarly to encryption. The algorithm iterates through the same number of rounds as it did during encryption after using the key expansion routine to generate the needed round keys. Each of the operations which comprised a round during encryption has an inverse operation, and these inverses are applied to the state during each round of decryption in a slightly modified order of what was done during encryption. When the final abridged round of decryption is completed, the resulting state is the plaintext, assuming that the correct key was used for decryption.

10.2 Cryptanalysis Using Nature-Inspired Optimization Algorithms

AES being a relatively new, thoroughly reviewed cryptosystem which was approved by the United States Government as a standard for use within its own agencies, is not known to have any insecurities. No security flaws or decryption shortcuts have so far been discovered for AES, and the algorithm is used extensively by governments and private organizations to secure data. Therefore, very limited information is available regarding the successful cryptanalysis of AES-encrypted messages.

Since AES is used to safeguard large quantities of sensitive information, it is reassuring to know that nature inspired optimization algorithms (genetic algorithms and particle swarm optimization) were not successful in cryptanalyzing AES-encrypted messages. Genetic algorithms, using the mutation operators and fitness functions described in Sections 9.3 and 2.2.4 were not successful in finding an AES key given a ciphertext. Due to the apparent limitations of particle swarm optimization for

cryptanalysis discussed in Section 9.5, it was not surprising that no working pair of c_1 and c_2 parameters could be found.

Although not in line with the original intent of this research, the failure of genetic algorithms and particle swarm optimization for cryptanalyzing an AES-encrypted ciphertext is to some extent a useful result. As with DES, AES seems to be resistant to attack by relatively simple algorithms which could be easily deployed by unscrupulous parties. This result is also indicative that the standard cryptographic algorithms used by governments and private organizations are somewhat secure.

Chapter 11

Conclusion and Future Work

The genetic algorithms implemented as part of this research were successful in cryptanalyzing simple, classical cryptosystems, but failed when confronted with modern product ciphers. Due to the nature of the problems being tackled, it is unclear whether cryptanalysis of modern product ciphers such as DES and AES is beyond the abilities of nature-inspired optimization algorithms; the creators and legitimate users of these algorithms would certainly hope so.

The research described in this document includes the development and application of a number of genetic algorithm and particle swarm optimization implementations to the cryptanalysis of a number of cryptosystems, ranging from simple classical ciphers to modern cryptographic standards. A character string based genetic algorithm was applied with success to the Caesar, Vigenère, mixed Vigenère and Autokey ciphers. The character string based genetic algorithm performed slightly differently on the different substitution ciphers, but showed that each could be readily cryptanalyzed using this technique. Tests were also run to determine the optimal combination of mutation operators to use for each cipher. The same character string based genetic algorithm was applied with limited success to the columnar transposition cipher. While discouraging, this limited success illustrated a flawed perception of the keyspace being searched, leading to a second genetic algorithm implementation. This second GA implementation worked by evolving permutations instead of character strings and showed much better performance than the character string

based GA when applied to the cryptanalysis of the columnar transposition cipher. The character string based genetic algorithm was next applied to some more modern cryptosystems, DES and AES, this time with no success. A third GA implementation, one which evolved bit strings, was developed and was also unsuccessful with the modern cryptosystems. Finally, a second nature-inspired optimization algorithm called particle swarm optimization was applied to the modern cryptosystems. PSO was also unsuccessful at cryptanalyzing DES and AES ciphertexts. Exploration of the fitness landscapes and search patterns used by GA's and PSO gave some indication as to why GA's were more successful at cryptanalysis than PSO, and why it would be very difficult to apply either algorithm successfully to DES or AES.

The effectiveness of genetic algorithms for the cryptanalysis of ciphertexts created with classical substitution based and permutation based cryptosystems has been illustrated and a comparison of the effectiveness of GA's against different classical cryptosystems is provided in Figures 11.1 and 11.2. As mentioned earlier, the general trends seen in the decryption speeds and failure rates of the 3 polyalphabetic substitution ciphers are quite similar, while the behaviour of the genetic algorithm when confronted with the columnar transposition cipher is markedly different.

It has been surmised that the combination of techniques (substitution and permutation) within a single cryptosystem yields a high level of security [23]. Both AES and DES combine substitutions with permutations during their operation, and both showed themselves to be resistant to attacks which were able to subvert cryptosystems composed of only one type of operation (substitution or permutation). This result bolsters the intuition that product ciphers offer a high level of security.

A complimentary explanation for the failure of genetic algorithms during the

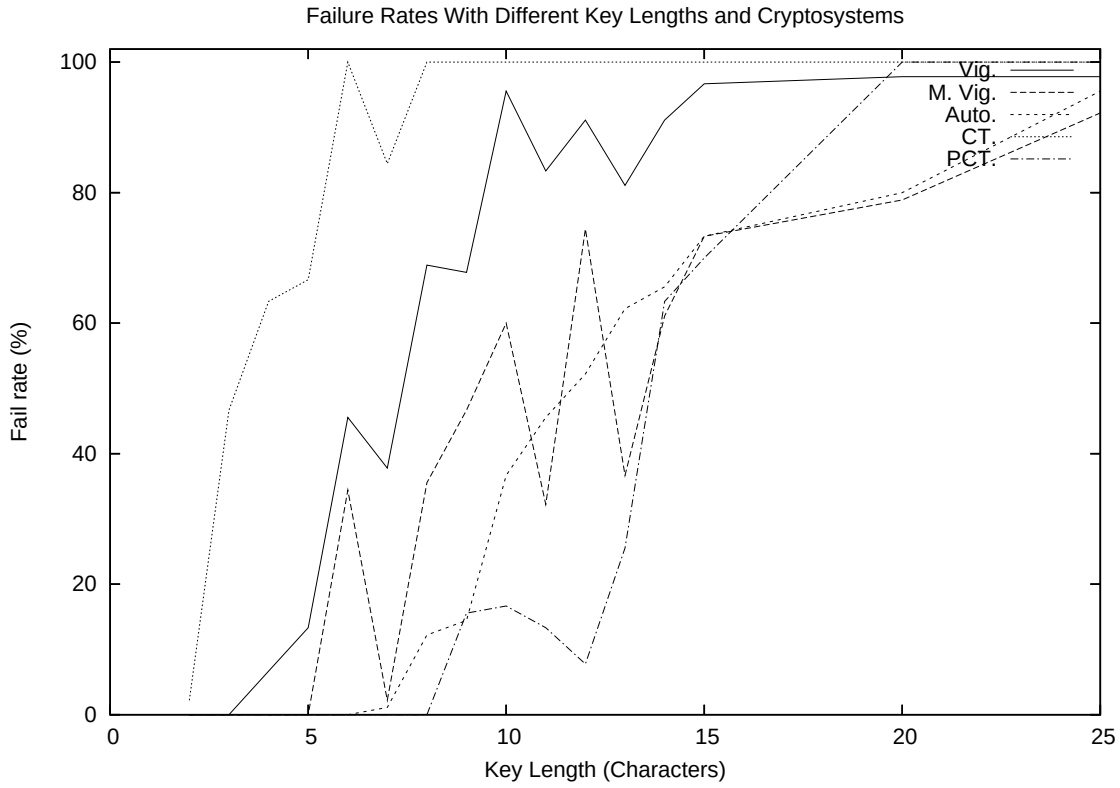

Figure 11.1: Failure Rates With Different Key Lengths and Cryptosystems

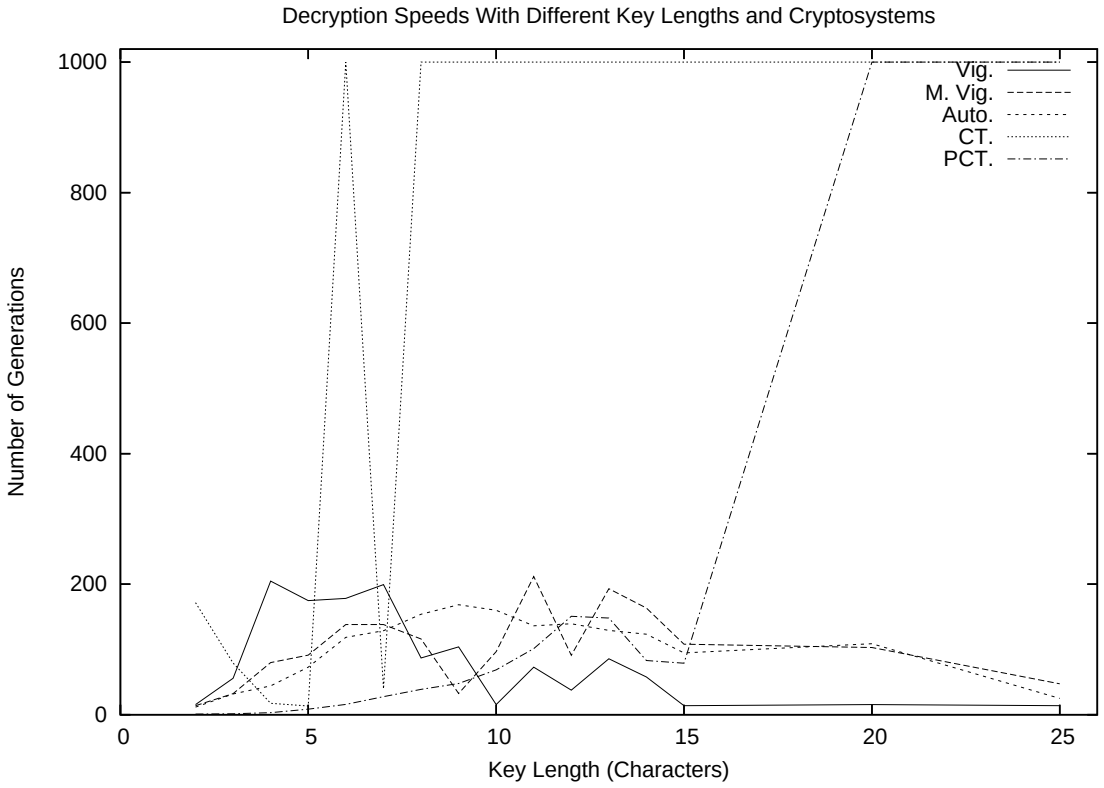

Figure 11.2: Decryption Speeds With Different Key Lengths and Cryptosystems

cryptanalysis of DES and AES is that most modern block ciphers (including DES and AES) exhibit the *Avalanche Effect* [19]. The avalanche criterion states that if any one input bit of the cipher in question (message or key bit) is changed, then then all of the cipher's output bits should change with a probability of 0.5. So changing one input bit should result in half of the output bits being changed. Attacks by genetic algorithms should not be successful against ciphers which fulfill this avalanche criterion because GA's rely on the assumption that making a small change to a good key will make a small change in the resulting decryption. Strictly speaking, the avalanche effect exhibited by modern block ciphers precludes this from working. It should, however, be mentioned that there is no strict requirement for a mutation operator to make *small* changes to keys. The successful application of such a mutation operator (as discussed in Section 11.2) is not precluded by the avalanche effect.

It should also be mentioned that the failure of genetic algorithms to cryptanalyze DES and AES-encrypted ciphertexts during the course of this research does not preclude their success at a future time. It is conceivable that a combination of fitness functions could be found which produce a fitness landscape that lends itself to navigation by an optimization algorithm. It is also possible that a combination of mutation operators could be found which successfully navigate a seemingly complex fitness landscape.

Of the nature-inspired optimization algorithms used during the course of this research, genetic algorithms showed themselves to be the most versatile for purposes of cryptanalysis. Particle swarm optimization allows the researcher latitude only in the manner in which it interprets the problem domain, through flexibility in the selection of fitness functions. Genetic algorithms allow the researcher latitude in

the selection of fitness functions – the visualization of the fitness landscape, as well as the manner in which the fitness landscape is explored – through the selection of appropriate mutation operators. The freedom to choose the manner by which the algorithm navigates the fitness landscape is not present in particle swarm optimization as the locomotory algorithm of the particles is largely fixed. These two customizable components of a GA are where further progress could be made with this research. Another avenue of experimentation with genetic algorithms and cryptanalysis lies in a more fundamental parameter of the genetic algorithm, namely the data type being evolved.

All of the code implementing genetic algorithms, particle swarm optimization and various cryptosystems are included on the compact disc accompanying this thesis.

11.1 Fitness Functions

Better fitness functions are one area where the performance of genetic algorithms could be improved. Additional metrics which measure the similarity of a decrypted ciphertext to English could be implemented. For instance, the frequency of occurrence of common words could be measured in a decryption and compared to values typical of the English language, taking the technique used in the genetic algorithm cryptanalysis of the columnar transposition cipher one step further.

The cost of being very specific in fitness functions is a loss of generality in their applicability. The use of fitness functions which use many highly specific metrics of the English language implies the assumption that the message which yielded the ciphertext was written in very typical English. The use of more sophisticated fitness

functions may yield a performance benefit for the genetic algorithm when faced with a typical English message, but may hinder the performance of the algorithm if the message was written in atypical English or another language. Knowledge of the fitness functions used in the genetic algorithm could be used as a form of anti-cryptanalysis countermeasure by the party initiating the communication by writing messages which do not fit the typical values found in English for any number of metrics, or writing in a foreign language.

The discussion of using overly specific fitness functions highlights a weakness in the technique used in this research, namely that the success of the genetic algorithm hinges on the cryptanalyst's correct assumptions as to the content of the messages. This assumption isn't always reasonable. Consider the possibility that the communicating parties choose to communicate in an obscure language for which "typical values" are not well known or well studied due to the lack of a large body of written material. Furthermore, consider the possibility that the content of the messages being sent is not meaningful text of any language. One of the most common uses of cryptography is to secure binary data such as images or spreadsheets. While each type of content adheres to a grammar, as does English, typical statistical values for these languages are likely not known.

11.2 Mutation Operators

Typical mutation operators used in genetic algorithms are just point mutations and crossover. However, mutation operators which are more pertinent to the problem domain may prove to be more effective. Although there is a set of commonly used

mutation operators, it should be pointed out that a mutation operator for a crypt-analyzing GA could make any conceivable change to one or more keys, so long as it produces a usable key for the cryptosystem as output, or implements a function M, such that $M(k_1, k_2, \ldots, k_n) = k_{new}$, for any positive value of n.

One potential example of a mutation operator being tailored to the problem domain of the columnar transposition cipher would be a mutation operator which takes a permutation as input and randomly permutes a subset of that permutation. A second example of such a mutation operator would be the use of the operations used in the AES algorithm as mutation operators. Consider a mutation operator which takes a single AES key as input and applies the transformation induced by the AES S-Boxes to that key. Both of these mutation operators are very unlike what is typically seen in a genetic algorithm, but are very relevant to the problem being tackled by the algorithm. As was previously discussed, GA's allow the researcher to tailor both the algorithm's interpretation of the fitness landscape via fitness functions and the algorithm's search pattern through the fitness landscape via mutation operators. The mutation operators just discussed exploit the latter advantage offered by the GA by tailoring a search pattern through the fitness landscape which is in tune with the cryptosystem being attacked.

The success of the CF mutation operator, which changes a randomly positioned letter in a key to a random letter, has been illustrated through much of this thesis (eg. Figures 5.1, 6.1 and 7.1). Although the MO mutation operator operates similarly, it did not prove to be as crucial to the success of the genetic algorithm. In Section 5.2 it is hypothesized that this is because the CF operator facilitates the replacement of one commonly occurring letter with another, while the MO operator does not. It is

possible that the CF mutation operator could be further improved by implementing it in such a way that it preferentially mutates letters in a key to letters which are commonly used in English instead of random letters as it currently does. Again, while such improvements would likely enhance the performance of the genetic algorithm against English messages, it would require modification to enhance the performance of the algorithm against messages in other languages. It seems that many of the possible improvements to genetic algorithms would increase the effectiveness of the algorithms but reduce their generality.

11.3 Data Types

The final area of potential future research to be mentioned is proper exploitation of the data type being evolved by the genetic algorithm. It is our belief that a GA will be most effective when the data type being evolved is as close as possible to the "native type" of the application [22]. In the case of the substitution ciphers examined as part of this research, the keys used by these cryptosystems can be most directly interpreted as strings of letters, and indeed, a genetic algorithm was effective at cryptanalysis of these cryptosystems when directed to evolve strings of letters.

While the keys used by human users in the columnar transposition cipher are strings of letters, the keys used by the cryptosystem itself are actually permutations. Upon seeing mediocre performance of a GA evolving character string keys for the columnar transposition cipher, it was decided that a GA which evolves permutations should be implemented as well. The data type evolved by this new GA was much closer to the native type used by the cryptosystem as keys, and indeed

the permutation-based GA outperformed the character string GA by a significant margin.

It isn't exactly clear what a DES key actually is, aside from the fact that we represent them as strings of 56 bits. If there is an interpretation of a DES key which is more closely related to the cipher (as a permutation is to the columnar transposition cipher, versus a character string), then it is possible that some degree of success could be realized by a genetic algorithm which would take advantage of this knowledge. While this is pure hypothesis, the results of the research done lends evidence to the notion that a genetic algorithm performs better when the data type being evolved is closer to the native type of the problem.

11.4 Improving Particle Swarm Optimization

Modifying the particle swarm optimization implementation produced as part of this research in order to make it effective for cryptanalysis may be possible. One potential improvement would be to change the locomotory algorithm used in PSO (described in Section 2.3) in order to make the search patterns used by the particles more akin to the search patterns used by the genetic algorithms which showed themselves to be effective against classical ciphers. A potential method for doing this would be to filter the results provided by the regular PSO algorithm by snapping the velocity vectors to their most prominent dimension before moving a particle. It is our belief that one of the most crucial components to the success of GA's is the use of mutation operators which only change a solution in one dimension per generation. In particle swarm optimization it isn't uncommon for a particle to move in many dimensions in a

single iteration. While this behaviour may be beneficial when applying the algorithm to other problems, it seems to hinder the algorithm's success when it is applied to cryptanalysis. Specifically, it would be possible to determine the dimension in which a particle would move the furthest at each iteration and only allow the particle to move in that one dimension. This would bring the locomotory patterns used by the particles more in line with the search patterns explored by the genetic algorithm and potentially render PSO more effective for cryptanalysis.

11.5 Hybrid Techniques

Hybrid techniques have been applied to cryptanalysis by other researchers in the field. In [16] Levbedko and Topchy apply a number of hybrid techniques to cryptanalysis including a combination of Genetic Algorithms and Local Search. A similar approach was taken in [31]. Generally speaking, the hybrid techniques perform better than pure genetic algorithms because each individual in the population examines its neighborhood for high-fitness locations in addition to using mutation operators. This technique could be used to further improve the performance of the genetic algorithm and particle swarm optimization implementations seen in this research.

11.6 Anti-GA Countermeasures

Despite the success of applying genetic algorithm to the cryptanalysis of classical ciphers, it is possible for communicating parties who are aware that their communications will by cryptanalyzed by a genetic algorithm to maintain the secrecy of their communications against such an attack. The fitness functions used in GA's require

the cryptanalyst to make assumptions about the message they are trying to recover. In most cases this includes an assumption on the language used by the communicating parties and the frequency characteristics of that language. One method of reducing a genetic algorithm's chance of success would be to violate such assumptions by either communicating in a different language, or using the words and phrases which have frequency characteristics atypical of the language.

Another method of mitigating the threat of genetic algorithm-based cryptanalysis would be to eliminate the redundancy of the messages being sent all together. One method of accomplishing this would be to use a compression utility such as *gzip* to compress a message before encrypting it and sending it. This technique would eliminate any redundancy in the ciphertext, rendering frequency-based fitness function ineffective.

Each of these methods attempt to remove the redundancy present in the ciphertext which the fitness functions used by GA's exploit. These same methods would also foil classical cryptanalytic techniques.

Bibliography

[1] A. Clark. Modern optimisation algorithms for cryptanalysis. In *Second Australian and New Zealand Conference on Intelligent Information Systems*, pages 258–262, Piscataway, NJ, 1994. IEEE Press.

[2] A. Clark and E. Dawson. A parallel genetic algorithm for cryptanalysis of the polyalphabetic substitution cipher. *Cryptologia*, 21(2):129–138, 1997.

[3] A. Clark and E. Dawson. Optimisation heuristics for the automated cryptanalysis of classical ciphers. *Journal of Combinatorial Mathematics and Combinatorial Computing*, 28:63–86, 1998.

[4] A. Clark, E. Dawson, and H. Bergen. Combinatorial optimization and the knapsack cipher. *Cryptologia*, 20(1):85–93, 1996.

[5] A. Clark, E. Dawson, and H. Nieuwald. Cryptanalysis of polyalphabetic substitution ciphers using a parallel genetic algorithm. In *IEEE International Symposium of Information and its Applications (ISITA '96)*, pages 17–20, Piscataway, NJ, September 17-20 1996. IEEE Press.

[6] J. Clark. Nature-inspired cryptography: Past, present and future. In Ruhul Sarker, Robert Reynolds, Hussein Abbass, Kay Chen Tan, Bob McKay, Daryl Essam, and Tom Gedeon, editors, *Proceedings of the 2003 Congress on Evolutionary Computation CEC2003*, pages 1647–1654, Piscataway, NJ, 8-12 December 2003. IEEE Press.

[7] B. Delman. Genetic algorithms in cryptography. Master's thesis, Rochester Institute of Technology, 2004.

[8] A. Dimovski and D. Gligoroski. Attack on the polyalphabetic substitution cipher using a parallel genetic algorithm. Technical report, Swiss-Macedonian Scientific Cooperation through SCOPES project, March 2003.

[9] W. Friedman. The index of coincidence and its applications in cryptography. Technical report, Riverbank Labs, 1920.

[10] J. P. Giddy and R. Safavi-Naini. Automated cryptanalysis of transposition ciphers. *The Computer Journal*, 17(4):429–436, 1994.

[11] D. E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley, Boston, MA, 1989.

[12] R. W. Hamming. Error detecting and error correcting codes. *Bell System Technical Journal*, 26(2):147–160, 1950.

[13] J. H. Holland. *Adaptation in Natural and Artificial Systems*. The University of Michigan Press, Ann Arbor, MI, 1975.

[14] J. Miller J. Kolodziejczyk and P. Phillips. The application of genetic algorithms in cryptanalysis of the knapsack cipher. In *Proceedings of Fourth International Conference on Pattern Recognition and Information Processing*, volume 1, pages 394–401, Piscataway, NJ, 1997. IEEE Press.

[15] A. Kerckhoffs. La cryptographie militaire. *Journel des Sciences Militaires*, 9(10), 1883.

[16] O. Levbedko and A. Topchy. On efficiency of genetic cryptanalysis for knapsack ciphers. In *Poster Proceedings of ACDM98*, Stuttgart, Germany, 1998. Springer-Verlag.

[17] F. T. Lin and C. Y. Kao. A genetic algorithm for ciphertext-only attack in cryptanalysis. In *IEEE International Conference on Systems, Man and Cybernetics*, volume 1, pages 650–654, Piscataway, NJ, 1995. IEEE Press.

[18] R. A. J. Mathews. The use of genetic algorithms in cryptanalysis. *Cryptologia*, 17(4):187–201, 1993.

[19] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, Boca Raton, FL, 1997.

[20] National Institute of Standards and Technology (NIST). *FIPS Publication 46: Announcing the Data Encryption Standard*, January 1977. Originally issued by National Bureau of Standards.

[21] National Institute of Standards and Technology (NIST). *FIPS Publication 197: Announcing the Advanced Encryption Standard (AES)*, November 2001.

[22] T. Ray. An evolutionary approach to synthetic biology: Zen and the art of creating life. *Artificial Life*, 1(1):195–226, 1994.

[23] C. Shannon. Communication theory of secrecy systems. *Bell System Technical Journal*, 24(4):656–715, 1949.

[24] S. Singh. *The Code Book*. Doubleday, New York, NY, 1999.

[25] R. Spillman. Cryptanalysis of knapsack ciphers using genetic algorithms. *Cryptologia*, 17(4):367–377, 1993.

[26] R. Spillman, M. Janssen, B. Nelson, and M. Kepner. Use of a genetic algorithm in the cryptanalysis of simple substitution ciphers. *Cryptologia*, 17(1):31–44, 1993.

[27] D. R. Stinson. *Cryptography – Theory and Practice*. CRC Press, Boca Raton, FL, 1995.

[28] L. Tolstoy. *War and Peace*. Modern Library, New York, NY, 2002.

[29] W. Trappe and L. C. Washington. *Introduction to Cryptography with Coding Theory*. Prentice Hall, Upper Saddle River, NJ, 2002.

[30] E. Weisstein. *Stirling's Approximation*. http://mathworld.wolfram.com/StirlingsApproximation.html.

[31] I. F. T. Yaseen and H. V. Sahasrabuddhe. A genetic algorithm for the cryptanalysis of the Chor-Rivest knapsack public key cryptosystem (pkc). In *Third International Conference on Computational Intelligence and Multimedia Applications (ICCIMA'99)*, pages 81–85, Piscataway, NJ, 1999. IEEE Press.

Appendix A

Collected Data

This section includes data generated and collected as part of the research detailed in this document. The data presented in subsequent sections should be sufficient to reproduce the figures seen throughout the text.

A.1 Caesar Cipher Data

Table A.1 is a copy of Table 4.2 and is included here for completeness.

A.2 Vigenère Cipher Data

Table A.2 includes data used to create Figures 5.1 and 5.2. The final column includes decryption speeds corrected to the fail rate, ie. decryption speed of successful trials only. Table A.3 includes data used to create Figure 5.3.

A.3 Mixed Vigenère Cipher Data

Table A.4 includes data used to create Figures 6.1 and 6.2. The final column includes decryption speeds corrected to the fail rate, ie. decryption speed of successful trials only. Table A.5 includes data used to create Figure 6.3.

Table A.1: Caesar Cipher Data.

Trial	Number of Generations
1	2
2	1
3	1
4	2
5	2
6	2
7	1
8	4
9	1
10	1
11	2
12	4
13	2
14	4
15	2
16	2
17	1
18	3
19	3
20	4
Average:	2.2

Table A.2: Vigenère Fail Rate and Decryption Speed Data

Mutation Operators	Fail Rate	Decryption Speed	Corrected Decryption Speed
CO	100	1000	0
RS	100	1000	0
RS-CO	100	1000	0
MO	99	990.9	90
MO-CO	99	994.26	426
MO-RS	99	994.45	445
MO-RS-CO	99	997.71	771
CF	21	267.09	72.27
CF-CO	9	150.84	66.86
CF-RS	9	234.04	158.29
CF-RS-CO	11	205.09	106.84
CF-MO	12	238.31	134.44
CF-MO-CO	17	290.7	145.42
CF-MO-RS	10	252.7	169.67
CF-MO-RS-CO	11	245.92	152.72

Table A.3: Vigenère Fail Rates and Decryption Speeds with Different Key Lengths

Key Length	Fail Rate	Decryption Speed	Corrected Decryption Speed
2	0	15.57	15.57
3	0	55.96	55.96
4	6.67	271.42	204.76
5	13.33	308.2	174.87
6	45.56	633.76	178.2
7	37.78	577.17	199.39
8	68.89	775.82	86.93
9	67.78	781.74	103.97
10	95.56	971.06	15.5
11	83.33	906.25	72.92
12	91.11	948.74	37.63
13	81.11	896.94	85.83
14	91.11	969.0	57.89
15	96.67	980.53	13.87
20	97.78	993.28	15.5
25	97.78	991.72	13.94

Table A.4: Mixed Vigenère Fail Rate and Decryption Speed Data

Mutation Operators	Fail Rate	Decryption Speed	Corrected Decryption Speed
CO	100	1000	0
RS	100	1000	0
RS-CO	99	995.17	517
MO	96	997.97	949.25
MO-CO	98	982.25	112.5
MO-RS	97	980.62	354
MO-RS-CO	98	990.36	518
CF	16	216.47	67.23
CF-CO	26	312.66	71.16
CF-RS	10	176.86	85.4
CF-RS-CO	13	206.88	88.37
CF-MO	12	199.83	90.72
CF-MO-CO	12	180	68.18
CF-MO-RS	15	244.88	111.62
CF-MO-RS-CO	11	207.72	109.80

Table A.5: Mixed Vigenère Fail Rates and Decryption Speeds with Different Key Lengths

Key Length	Fail Rate	Decryption Speed	Corrected Decryption Speed
2	0	13.92	13.92
3	0	32.41	32.41
4	0	79.97	79.97
5	0	91.43	91.43
6	34.44	482.64	138.2
7	2.22	160.38	138.16
8	35.56	471.61	116.06
9	46.67	599.31	32.64
10	60	696.64	96.64
11	32.22	533.9	211.68
12	74.44	835.44	91.0
13	36.67	559.83	193.17
14	61.11	774.63	163.52
15	73.33	841.3	107.97
20	78.89	891.83	102.94
25	92.22	969.57	47.34

Table A.6: Autokey Fail Rate and Decryption Speed Data

Mutation Operators	Fail Rate	Decryption Speed	Corrected Decryption Speed
CO	100	1000	0
RS	100	1000	0
RS-CO	99	996.26	626
MO	100	1000	0
MO-CO	98	984.25	212.5
MO-RS	99	990.78	78
MO-RS-CO	96	988.24	706
CF	36	412.36	81.81
CF-CO	33	378.2	71.94
CF-RS	26	329.12	93.41
CF-RS-CO	33	395.61	97.93
CF-MO	36	441.58	127.47
CF-MO-CO	29	374.52	119.04
CF-MO-RS	22	305.85	110.06
CF-MO-RS-CO	39	445.2	90.49

A.4 Autokey Cipher Data

Table A.6 includes data used to create Figures 7.1 and 7.2. The final column includes decryption speeds corrected to the fail rate, ie. decryption speed of successful trials only. Table A.7 includes data used to create Figure 7.3.

A.5 Columnar Transposition Cipher Data

Table A.8 includes data used to create Figures 8.1 and 8.2. The final column includes decryption speeds corrected to the fail rate, ie. decryption speed of successful trials only. Table A.9 includes data used to create Figure 8.3. Table A.10 includes additional data used to create Figures 8.4 and 8.5.

Table A.7: Autokey Fail Rates and Decryption Speeds with Different Key Lengths

Key Length	Fail Rate	Decryption Speed	Corrected Decryption Speed
2	0	11.56	11.55
3	0	31.83	31.83
4	0	44.14	44.14
5	0	72.8	72.8
6	0	118.62	118.62
7	1.11	139.89	128.78
8	12.22	276.19	153.97
9	14.44	312.91	168.47
10	36.67	526.77	160.1
11	45.56	591.77	136.21
12	52.22	661.61	139.39
13	62.22	751.32	129.1
14	65.56	778.82	123.27
15	73.33	828.08	94.74
20	80.0	908.92	108.92
25	95.56	980.68	25.12

Table A.8: Columnar Transposition Fail Rate and Decryption Speed Data

Mutation Operators	Fail Rate	Decryption Speed	Corrected Decryption Speed
CO	0	79.7	79.7
RS	0	180.52	180.52
RS-CO	0	54	54
MO	0	51.99	51.99
MO-CO	0	54.61	54.61
MO-RS	0	74.45	74.45
MO-RS-CO	0	66.74	66.74
CF	0	18.45	18.45
CF-CO	0	16.84	16.84
CF-RS	0	19.9	19.9
CF-RS-CO	0	21.14	21.14
CF-MO	0	26.09	26.09
CF-MO-CO	0	26.47	26.47
CF-MO-RS	0	31.92	31.92
CF-MO-RS-CO	0	32.71	32.71

Table A.9: Columnar Transposition Fail Rates and Decryption Speeds with Different Key Lengths

Key Length	Fail Rate	Decryption Speed	Corrected Decryption Speed
2	2.22	193.93	171.71
3	46.67	545.79	79.12
4	63.33	650.8	17.47
5	66.67	680.12	13.46
6	100	1000	1000
7	84.44	884.36	39.91
8	100	1000	1000
9	100	1000	1000
10	100	1000	1000
11	100	1000	1000
12	100	1000	1000
13	100	1000	1000
14	100	1000	1000
15	100	1000	1000
20	100	1000	1000
25	100	1000	1000

A.6 Summary Data

Tables A.11 and A.12 include data used to create Figures 11.1 and 11.2 respectively. The "decryption speed" columns include decryption speeds corrected to the fail rate, i.e. decryption speeds of successful trials only. These table are used to compare GA performance across the different classical ciphers. Columns marked "V" indicate Vigenère Cipher data, columns marked "MV" indicate Mixed Vigenère Cipher data, columns marked "A" indicate Autokey Cipher data, columns marked "CT" indicate Columnar Transposition Cipher data and columns marked "PCT" indicate Columnar Transposition Cipher data when using a permutation-based GA.

Table A.10: Columnar Transposition Fail Rates and Decryption Speeds with Different Key Lengths using Permutation GA

Key Length	Fail Rate	Decryption Speed	Corrected Decryption Speed
2	0	1.0	1
3	0	1.39	1.39
4	0	3.16	3.16
5	0	8.62	8.62
6	0	15.88	15.88
7	0	27.71	27.71
8	0	38.87	38.87
9	15.56	203.47	47.91
10	16.67	235.47	68.8
11	13.33	234.57	101.23
12	7.78	228.57	150.79
13	25.56	403.64	148.09
14	63.33	716.31	82.98
15	70.0	778.99	78.98
20	100	1000	1000
25	100	1000	1000

Table A.11: Fail Rates For Different Ciphers

Key Length	V Fail	MV. Fail	A. Fail	CT Fail	PCT Fail
2	0	0	0	2.22	0
3	0	0	0	46.67	0
4	6.67	0	0	63.33	0
5	13.33	0	0	66.67	0
6	45.56	34.44	0	100	0
7	37.78	2.22	1.11	84.44	0
8	68.89	35.56	12.22	100	0
9	67.78	46.67	14.44	100	15.56
10	95.56	60	36.67	100	16.67
11	83.33	32.22	45.56	100	13.33
12	91.11	74.44	52.22	100	7.78
13	81.11	36.67	62.22	100	25.56
14	91.11	61.11	65.56	100	63.33
15	96.67	73.33	73.33	100	70
20	97.78	78.89	80	100	100
25	97.78	92.22	95.56	100	100

Table A.12: Decryption Speeds For Different Ciphers

K. Length	V. Speed	MV. Speed	A. Speed	CT Speed	PCT Speed
2	15.57	13.92	11.55	171.71	1
3	55.96	32.41	31.83	79.12	1.39
4	204.76	79.97	44.14	17.47	3.16
5	174.87	91.43	72.8	13.46	8.62
6	178.2	138.2	118.62	1000	15.88
7	199.39	138.16	128.78	39.91	27.71
8	86.93	116.06	153.97	1000	38.87
9	103.97	32.64	168.47	1000	47.91
10	15.5	96.64	160.1	1000	68.8
11	72.92	211.68	136.21	1000	101.23
12	37.63	91	139.39	1000	150.79
13	85.83	193.17	129.1	1000	148.09
14	57.89	163.52	123.27	1000	82.98
15	13.87	107.97	94.74	1000	78.98
20	15.5	102.94	108.92	1000	1000
25	13.94	47.34	25.12	1000	1000

A.7 Messages, Keys and Known Text Fragments

The following section contains the messages, keys and known text fragments used in the key length analyses seen throughout this thesis. All of the code implementing genetic algorithms, particle swarm optimization and various cryptosystems are included on the compact disc accompanying this thesis.

Message 1: thepowerandeleganceofgenerictypeshavelong
beenacknowledgedgenericsallowdeveloperstoparameterisedatat
ypesmuchlikeyouwouldparameteriseamethodthisbringsanewdimen
sionofreusabilitytoyourtypeswithoutcompromisingexpressiven
esstypesafetyorefficiencynownetgenericsmakesthispoweravail

abletoallnetdevelopersbyintroducinggenericconceptsdirectly
intothecommonlanguageruntimeclrmicrosofthasalsocreatedthef
irstlanguageindependentgenericsimplementationtheresultisas
olutionthatallowsgenerictypestobeleveragedbyallthelanguage
softhenetplatform

Message 2: digitalrightsmanagementcontentprovidersar
eusingthedigitalrightsmanagementtechnologycontainedinthiss
oftwaretoprotecttheintegrityoftheircontentsothattheirintel
lectualpropertyincludingcopyrightinsuchcontentisnotmisappr
opriatedportionsofthissoftwareandthirdpartyapplicationssuc
hasmediaplayersusetoplaysecuresontentifthesoftwaressecurit
yhasbeencompromisedownersofsecurecontentmayrequestthatmicr
osoftrevokethesoftwaresrighttocopydisplaaandorplaysecureco
ntentrevocationdoesnotalterthesoftwaresabilitytoplayunprot
ectedcontentalist

Message 3: thesewordswereutteredinjulybyannapavlovna
schereradistinguishedladyofthecourtandconfidentialmaidofho
nourtotheempressmaryafyodorovnaitwashergreetingtoprincevas
silyamanhighinrankandofficewhowasthefirsttoarriveathersoir
eeannapavlovnahadbeencoughingforthelastfewdaysshehadanatta
ckoflagrippeasshesaidgrippewasthenanewwordonlyusedbyafewpe
opleinthenotesshehadsentroundinthemorningbyafootmaninredli

veryshehadwrittentoallindiscriminatelyifyouhavenothingbett
ertodocountorprinceandiftheprospectofspendinganeveningwith
apoorinvalidisnot

Known Text Segment 1: powerandelegance

Known Text Segment 2: italrightsmanage

Known Text Segment 3: sewordswereutter

Complete Key 1: karelhopesthisworkswellno

Complete Key 2: whilerespectingthesyntaxs

Complete Key 3: promotionalcertificatetob