

Regis University ePublications at Regis University

All Regis University Theses

Spring 2015

Man Versus Machine: Can Computers Crack Cryptography?

Brandon Ward
Regis University

Follow this and additional works at: <https://epublications.regis.edu/theses>

Recommended Citation

Ward, Brandon, "Man Versus Machine: Can Computers Crack Cryptography?" (2015). *All Regis University Theses*. 656.
<https://epublications.regis.edu/theses/656>

This Thesis - Open Access is brought to you for free and open access by ePublications at Regis University. It has been accepted for inclusion in All Regis University Theses by an authorized administrator of ePublications at Regis University. For more information, please contact epublications@regis.edu.

Regis University
Regis College
Honors Theses

Disclaimer

Use of the materials available in the Regis University Thesis Collection ("Collection") is limited and restricted to those users who agree to comply with the following terms of use. Regis University reserves the right to deny access to the Collection to any person who violates these terms of use or who seeks to or does alter, avoid or supersede the functional conditions, restrictions and limitations of the Collection.

The site may be used only for lawful purposes. The user is solely responsible for knowing and adhering to any and all applicable laws, rules, and regulations relating or pertaining to use of the Collection.

All content in this Collection is owned by and subject to the exclusive control of Regis University and the authors of the materials. It is available only for research purposes and may not be used in violation of copyright laws or for unlawful purposes. The materials may not be downloaded in whole or in part without permission of the copyright holder or as otherwise authorized in the "fair use" standards of the U.S. copyright laws and regulations.

Name: Brandon Ward

Major: Computer Science & Mathematics

MAN VERSUS MACHINE: CAN COMPUTERS CRACK CRYPTOGRAPHY

Advisor's Name: Dr. James Seibert

Reader's Name: Dr. Dennis Steele

After starting the project with the hopes of developing a program able to crack substitution ciphers via artificial life concepts, some deeper questions were arrived at. What is the line between the man and the machine? Can computers ever be capable of sentient thought? What does it mean for us as a species as we continually develop better ways to compute hard problems fast? Ultimately, I may not have the answer to these problems, but science might. I have to conclude that for now cryptography is safe, but will it always be safe? With the advent of the quantum computing era just over the horizon, the definition of a smart and intelligent computer is about to change drastically, and achievements in computing such as Deep Blue are going to become more commonplace than ever.

MAN VERSUS MACHINE: CAN COMPUTERS CRACK CRYPTOGRAPHY?

A thesis submitted to

Regis College

The Honors Program

in partial fulfillment of the requirements

for Graduation with Honors

by


Brandon Ward

May 2015

Thesis written by

Brandon Ward

Approved by



Thesis Advisor



Thesis Reader

Accepted by

Director, University Honors Program

TABLE OF CONTENTS

| | |
|--|-----|
| PREFACE and ACKNOWLEDGMENTS..... | iii |
| I. INTRODUCING PROBLEMS OF OLD WITH THEORIES OF NEW..... | 1 |
| II. CLASSICAL CRYPTOGRAPHY AND NEW-AGE COMPUTING..... | 8 |
| III. A TRADITIONAL GENETIC ALGORITHM..... | 17 |
| IV. BALANCING THEORITICAL COMPLEXITY AND COMPUTATIONAL SIMPLICITY..... | 20 |
| V. MAN VERSUS MACHINE: THE LINE IN THE SAND?..... | 25 |
| | |
| BIBLIOGRAPHY..... | 31 |
| | |
| APPENDICES: | |
| | |
| A. SOURCE CODE FROM FIRST GENETIC ALGORITHM..... | 33 |
| B. SOURCE CODE FROM SECOND GENETIC ALGORITHM..... | 43 |

PREFACE and ACKNOWLEDGMENTS

Preface. This thesis is the culmination of my studies for undergraduate degrees in mathematics and computer science, and ultimately expresses the desire I have to use both mathematics and computer science to theorize solutions to difficult problems. I ultimately hope to portray the ever-evolving aspects of these fields, and show how using them in unison can lead to amazing discoveries.

Acknowledgments. Many people helped and provided encouragement during the writing of this thesis. My special thanks goes out to Dr. James Seibert – My advisor, he always pushed me to go for more, to ask deeper questions, and to offer the best explanations I could; Dr. Dennis Steele – My reader with incredible enthusiasm for the expanding capabilities of computer science. He sparked me to push what I knew about artificial life and apply what I learned in an area that interested me; Dr. Tom Howe & Dr. Tom Bowie – As directors of the Honors program in my time there, they taught me what it meant to be a lover of learning, and ultimately how to live in such a way that reflects my love of learning. A special thanks to these professors as well as every other figure I have had the pleasure of learning from during my time here at Regis. They have left their marks on me that I hope to keep with me in my upcoming journeys, and wherever life takes me.

INTRODUCING PROBLEMS OF OLD WITH THEORIES OF NEW

Two courses during my time at Regis contributed to the spark for this thesis. When I was a sophomore, I took Cryptography, which was all about the math behind cryptosystems. It covered how to create them, analyze them, and most interesting of all, how to break them (simple systems at least). The second course was Artificial Life, which is all about using known patterns of life and stripping them down to their mechanics and then using those mechanics to solve meaningful problems. The class also attempted to discover how to make a computer find results to problems that are seemingly intuitive for humans, yet extremely difficult for computers. My background as a mathematics and computer science double major suited itself for attempting to merge the two courses. Why not try to encode the theories of artificial life to make a machine solve a cryptosystem of old?

Why artificial life? Artificial life has become a large aspect of computer science, and it has the ability to solve otherwise incalculable problems by imitating some aspect of life. These problems are incalculable because there is no easily computable solution to them, and artificial life allows for a calculable estimation of the solution. The area of artificial life that I am interested in is the genetic algorithm. This is the idea that a computer could 'learn' a better estimation based on analyzing 'populations' of data, based on genes and the data's ability to thrive under our defined system of 'living well'. It is a new twist on the classic definition of 'survival of the fittest'. Just as evolution has

produced 'better' species through the years, by replicating genetics in a mathematical way I hope to create an estimation of the solution that hones into the actual truth.

What does the age of 'smart' computers mean for cryptography? Cryptography must continue to evolve, grow, and change. Tracing the history of Cryptography will give insight to where it is going, and the most insight will be given by the most recent developments, since the speed at which Cryptography is developing has greatly increased in the past 50 years. Are our secrets safe? Can our means of protecting our secrets continue to outpace our means of extracting other's secrets? There are indicators out there, and I hope to analyze and explain them in order to also explore the future of cryptography.

Lastly, I hope to cover why we care. Computers continually get faster and faster, so what makes our current cryptosystems safe? Our entire culture of digital secrecy relies on the survival and also enhancement of cryptography, and hopefully the mathematicians and computer scientists out there have the answer to keep our digital secrets safe for centuries.

For the purposes of this thesis, some background terminology is necessary. From the cryptography side of things, the plaintext is the original message we are trying to send. When the message is encrypted to be sent, it is called a ciphertext. Encrypting and decrypting are done with a key. Generally, in cryptography, it is assumed the third party who wants to steal the message knows everything about the encrypted message except the key. That is, they know the method used to encrypt it, and they have free

access to the ciphertext, yet the goal of cryptography is that without the key, the ciphertext is useless to the thief. There are four types of crypto attacks: **Ciphertext only**, where the third party only has a copy of the ciphertext, this is the most common attack, and can take many forms. The most common is the brute force attack, which essentially means try every possibility until something works; **Known plaintext**, where the third party again has the ciphertext, but they also gain access to the corresponding decrypted plaintext, allowing them to deduce the keys of encryption, further allowing them to crack future messages with the same key; **Chosen plaintext**, where the third party gains access to the encryption machine, and is able to encrypt several strings for themselves in order to deduce the key; lastly, **Chosen ciphertext**, where the third party gains access to the decryption machine, and is able to decrypt several strings for themselves in order to deduce the key(Trappe & Washington, 2-4). For this thesis, I will largely focus on ciphertext only, since this is the most common type of attack, and the main type of attack modern cryptography is geared towards preventing.

When it comes to artificial life, this falls into a category of Evolutionary Computation. Evolutionary Computation is the idea that a “system was to evolve a population of candidate solutions to a given problem, using operators inspired by natural genetic variation and natural selection” (Mitchell ,2). What this means for the genetic algorithm is that our population is the set of all possible solutions to the ciphertext. The gene is pure data, and in this case, the gene is a one to one mapping of the alphabet that will allow for the reversal of a substitution cipher. Just as in nature,

the concepts of Darwinism are prevalent, meaning survival of the fittest; the genetic algorithm needs to replicate this by allowing the strongest members to survive. When the 'creatures' we are studying are pure data though, what defines them as strong? The gene that decrypts the ciphertext to the most understandable message then should be considered the strongest. Its genetic content will be used as a parent for the next child, and the goal is to create better and better decrypts until we are left with a plain English decryption. Some important terms to remember are: **genes** are the basic unit within the genetic algorithm, and the unit upon which we operate, evaluate, and use; **fitness** is the defined measure of how 'healthy' a given gene is, meaning how closely it comes to being the solution; **fitness function** is the formula that computes fitness using the data provided by the gene in order to create the meaningful measure of success per gene (Goldberg, 60-2). The crux of a genetic algorithm is the fitness function. The more complex the problem, the more important a well-crafted fitness function becomes. The fitness function is the measurement of success, and without a strong fitness function the chances of inconclusive results is higher.

To take a simple example, say we are trying to solve the largest value of x^2 for binary strings 5 digits long. Since we are trying to solve x^2 , a logical choice for a fitness function is $f(x) = x^2$, where in 5 digit binary strings $x^2 = (x_4 * 2^4 + x_3 * 2^3 + x_2 * 2^2 + x_1 * 2^1 + x_0 * 2^0)^2$ and $x_i = 0|1$. We know that the highest value will be 11111; the trick is making the computer realize that too. The specific version of genetic algorithms that I thought best for my problem is the hill climbing algorithm. For

illustration on how this works on this particular problem, we will start with the gene as a random binary string, say 00100. To find the next best solution to our problem, we randomly change 1 bit of our gene, and test if it is better. If the x_2 position is randomly changed, our gene would be 00000 which has a fitness of 0 as opposed to 00100 which has a fitness of 16. The new gene is lower, so we ignore it, and try again. This time, if we change the x_4 bit, the result would be 10100 which has a fitness of 400. This is better, and so it becomes the new gene and the process repeats. Eventually, the genetic algorithm will produce the 'victor' with the gene pattern of 11111 with a fitness of 961, the highest possible value of $f(x) = x^2$ for 5 digit binary strings. Using this formula, I want to modify it to crack an encoded message without knowing anything about the plaintext or original message.

Beyond just the mathematics of this problem, in order to obtain any form of results, any fitness function I use needs to be based in language. It needs to capture key information about the problem I am trying to solve. How can I turn a garbled scramble of letters back into that highly classified plain-text message with unknown secrets? The heart of the problem is not new. Essentially, we need to teach the computer how to learn. By giving it a string of letters that mean nothing to it, and trying to extract a meaningful language, it feels like the computer needs to be alive. The computer will learn what combination of letters makes sense, and what combinations leave the letters just as un-decipherable as before. Is this not the same thing our bodies do while we live our day to day lives? We decrypt our surroundings, and in our minds we piece them

together in the way that makes the most sense based on past experiences, current surroundings, and our interpretation of the situation. We are constantly learning, which is what a genetic algorithm hopes to replicate.

When does using a genetic algorithm to solve problems make sense? There are several conditions that experts have agreed are important to satisfy before proceeding.

“Many researchers share the intuitions that if the space to be searched is large, is known not to be perfectly smooth and unimodal, or is not well understood, or if the fitness function is noisy, and if the task does not require a global optimum to be found” (Mitchell, 155-6).

The search space of a substitution cipher is $26!$ (26 factorial), which is incredibly large; slight changes in the alphabetical mapping will provide greatly varying fitness values, which will ensure that the fitness graph is not smooth, not unimodal for optimal fitness; lastly, even though the global optimum could be the true decryption, this is not guaranteed, so relying on the randomness of the genetic algorithm will allow us to remain open to all the ‘healthy’ solutions and will hopefully find the decryption.

Essentially, the point where this genetic algorithm meets the substitution cipher is the starting point of a smart brute-force attack on this cryptosystem. The genetic algorithm will start by allowing all possible solutions, and then through the fitness functions, will rapidly rule out weaker, clearly incorrect, solutions. The genetic algorithm will build an understanding of the ciphertext in order to construct the most logical plaintext.

I want to apply my knowledge and programming skills to build a mobile application on top of the Android framework. This app will apply various aspects of cryptology, coding theory, and artificial life. I will design the application to use a genetic algorithm to attempt to break a simple substitution cipher. In the end, even though I may or may not have a working substitution cipher decoding app, I hope to learn a little more about why such a problem is so complex and difficult, and what it means about the relationships between humans and computers.

CLASSICAL CRYPTOGRAPHY AND NEW-AGE COMPUTING

Cryptography has had many different applications throughout history. It started out as encoded hand-written messages passed between friends and colleagues, including the ancient Roman legions, but in today's age, it has evolved to the cutting edge of digital privacy. Every secret stored on a computer, is protected by some means of cryptography. The standard parties when discussing cryptography are the two parties trying to communicate, Alice and Bob, along with the third party trying to intercept the message, Eve. Throughout the years, the problem has remained the same, that is, Alice needs to get a message to Bob, and for simplicity's sake we will say they do not want Eve to be able to read the message. The way this is achieved is by Alice encrypting the message, sending the encryption over open communication channels, then when Bob receives this, he decrypts it and is able to understand Alice's intents. Eve is also able to get ahold of the encryption, but the theory is that she does not have the key, and therefore cannot read the message.

The problem has evolved because this figurative 'Eve' has continually grown smarter. Modern day mathematics and computing power, have made the cryptosystems of the ancients mere child's play. The 'secureness' of modern cryptography relies on the fact that a computer could not 'brute force', or try every possible combination, in order to crack a secret message within any 'reasonable' amount of time. Therefore, as computers get faster and we develop better algorithms,

we must also increase the complexity of our cryptosystems to outpace our own technological advances.

Cryptography is not a new science, even though the invention of computers has caused this field to explode. Some of the first true applications of Cryptography go all the way back to the Romans. Specifically, the cryptosystem to make a note of is the Caesar Cipher. It was a simple shift cipher, that can be classified as a substitution cipher, and it was as simple as shifting the alphabet by a certain number, wrapping back to the beginning if the end is reached (Trappe & Washington, 12 – 13). If the algorithm was to shift by 3 places, an 'a' became a 'd', a 'y' became a 'b', a 'z' became a 'c' and so on (Trappe & Washington, 13). Obviously, with today's modern knowledge and computing power, this is a very trivial substitution cipher, but it is still a formal birth of cryptography.

Because this cryptosystem is, compared to modern cryptosystems, trivial, it is feasible to hope to develop a brute force attack for it. As will be explained, with modern cryptography a brute force attack could take centuries to produce an answer. The best examples of modern cryptography come in the form of RSA and AES cryptography. These two cryptosystems serve to illustrate the two main applications for cryptography, public key cryptography (RSA) and symmetric key cryptography (AES). Together they paint a complete cryptographic system, because the standard practice is to use public key cryptography to securely transmit small amounts of data, such as a symmetric key, and for symmetric key cryptography to do the heavy lifting for secure communication

with large amounts of data (Trappe & Washington, 4-6). What makes these cryptosystems so secure, and also infeasible to attempt a brute force attack upon will briefly be explained before proceeding.

RSA is a public key cryptosystem. What this means is that there is a public encryption key that everyone is given access to, but only the receiver of the messages has the decryption key (Trappe & Washington, 164). A key-space is assigned to RSA of some very large number n , with $n = p * q$ where p and q are both very large primes. The concept of RSA is that messages are encrypted with an encryption exponent, e , which is made in such a way that the greatest common divisor of this encryption exponent and $(p - 1)(q - 1)$ is 1 i.e. $\gcd(e, (p - 1)(q - 1)) = 1$ (Trappe & Washington, 165). Even though someone looking to break this cryptosystem will know the public key, which is the encryption exponent e , and that it must be related to p and q in the above mentioned way, this is a frightfully challenging problem to solve, because factoring two very large prime numbers is an enormous computational undertaking. It is such an expensive computation that if the primes are large enough, this cryptosystem could easily take many millennia to crack.

The only issue with RSA is that when the message is converted to a number, $m < n$ must be true since this entire computation is being reduced $\text{mod}(n)$. And if $m > n$, then the message must be broken down into smaller blocks until the first condition is satisfied. And it is because of reasons such as this that public key cryptography is primarily reserved for exchanging both extremely short messages, such

as a username and password, or a private key for symmetric key cryptography for larger messages.

AES, or Advanced Encryption Standard, is a symmetric key cryptosystem. A symmetric key cryptosystem is one where both the encryption and decryption keys are known by both parties, or at the very least the encryption key is shared between parties and the decryption key can be easily computed from the encryption key (Trappe & Washington, 4). The algorithm is comprised of four basic steps that are repeated anywhere from 10 to 14 times in any one encryption (Trappe & Washington, 152). By taking the message in its binary form, it is easy to arbitrarily break up the message into 8-bit or 1 byte segments. To begin the encryption, the message is put into 4 by 4 matrices, 1 byte at a time. Each of the four steps is specifically catered to enhance the encryption by specifically making various aspects of decrypting it more difficult without the key. The first step is the byte-sub transformation, and via a built in random 16 by 16 matrix called the S-box, each byte of the message is directly substituted with another byte using the built in lookup table. This step is non-linear, making linear cryptanalysis attacks less successful (Trappe & Washington, 152-55). The next step is the shift-row transformation, where the rows of the matrix are shifted cyclically to the left, by offsets equal to the row number minus one. The next step is the mix-column transformation, which is achieved by multiplying the current 4 by 4 matrix by a built in matrix. Both the shift-row and mix-column transformations serve to diffuse the bits of the encryption over the repeated rounds (Trappe & Washington, 152-56). The final step is the add-

round-key step, where the round key is exclusively 'or'-ed with the matrix of bytes and each round key is derived from the original key(Trappe & Washington, 152-57). By doing this, AES effectively encrypts the message with as many keys as there are rounds, in addition to the added steps that make the encryption non-linear and diffused to make patterns more difficult to recognize in any given encryption.

AES is built for large amounts of data to be transmitted quickly, which is why it is not very computationally difficult, and in fact, all of the operations are computationally inexpensive assuming that the original encryption key is known. This is because all four operations used to encrypt the message have simple mathematical inverses. Each round key, by the properties of the exclusive or logic, is its own inverse, so by exclusively 'or'-ing the matrix with the round key returns the starting point for that operation. The mix-column can be undone because the original built in matrix is invertible, so multiplying by the inverse reverses that step with ease. To undo the shift-row encryption, just shift to the right by the same offsets used in the encryption. Lastly, to undo the byte-sub, there is an inverse-byte-sub lookup table that will return the original matrix and thus the message, after we have done this the same number of times as the encryption (Trappe & Washington, 158). The reason such a computationally easy cryptosystem can still be used as a secure cryptosystem is because at a minimum the data is being taken in 128 bit chunks, which means it is encrypting several letters at a time. Furthermore, the encryption key is being applied multiple times in order to make sure there are no patterns in the final encryption.

The fact that RSA depends on a very hard mathematical computation, the factoring of 2 large primes, and AES having an enormous key-space void of patterns are the kinds of theories currently keeps our modern cryptosystems safe from attacks. However, now that there is an understanding of the basis of Cryptography, it is important to point out the advancements happening in the field that could potentially render modern cryptography obsolete. Today's computing power is growing at an ever increasing rate, which continues to improve brute force attacks against some of our current cryptographic standards. That being said, the current cryptosystems are still very secure against all current computation methods because increasing the key size effectively increases the amount of possible keys, and thus adds more necessary computation. Increasing the key size by one number consequently increases the number of possible keys by factors of 10. Even with this method available to us, it is still possible that RSA and AES will be obsolete in the future when several hard problems pertaining to computation ability are solved.

One of the methods of computation that could threaten current cryptosystems is quantum computing. Quantum computing is a particularly exciting prospect for modern computing. To describe a quantum computing system, we need to look at the current notion of parallelism in classical computing systems.

“Classically, the time it takes to do certain computations can be decreased by using parallel processors. To achieve an exponential decrease in time requires an exponential increase in the number of processors, and hence an exponential

increase in the amount of physical space needed. However, in quantum systems the amount of parallelism increases exponentially with the size of the system. Thus an exponential increase in parallelism requires only a linear increase in the amount of physical space needed. This effect is called quantum parallelism” (Rieffel & Polak, 301).

This parallelism would spell the doom of just about every single modern cryptosystem. Parallelism to this extent means that the computer could try every single combination of keys all at once so to brute force a cryptosystem such as RSA, it would only take as long as trying one combination.

While it might sound like Quantum Computing is this great computational notion, it is not without its problems. In fact, the whole challenge of Quantum Computing is quite a large one. “While a quantum system can perform massive parallel computation, access to the results is restricted. Accessing the results is equivalent to making a measurement, which disturbs the quantum state” (Rieffel & Polak, 301). So basically, reading the results also causes the results to be destroyed. But there is another catch that still needs to be solved. When reading the data from the quantum state, “we can only read the result of one parallel thread, and because measurement is probabilistic, we cannot even choose which one we get” (Rieffel & Polak, 301). It is safe to say that, even though the notion of Quantum Computing is an altogether powerful and terrifying thought to cryptography, it is still a few hard problems away from existing.

However, if a Quantum Computing device is successfully built, this will also bring about the new era of cryptography. Many of the old standards will be rendered obsolete, and the focus will have to shift towards a handful of cryptosystems that are still secure to such a powerful computing device. The most obvious system is Quantum Cryptography. The basis of Quantum Cryptography is using photons to send keys. The reason Quantum Cryptography is secure, is because “if someone tries to spy on the process and intercept the photons en route, [we] will notice too many discrepancies and conclude the line of communication is insecure”(Mone, 13). These photons are used to transmit keys, and not messages, however, the aspect that makes this so secure is that if someone begins to intercept the key, they start changing it as they read it. Quantum Cryptography is such that no one except the intended party can read the transmitted photons, otherwise the intended party is almost instantly aware that someone is trying to crack their cryptosystem. If all else were to fail, Quantum Cryptography would still provide a means of secure communications if all other systems were insecure.

Yet, this is not the case, as there are other cryptosystems that could withstand an attack by a quantum computer. One such system is the Lattice-Based Cryptosystem. Lattice Cryptography is based off the concept that it is very difficult to solve the shortest vector problem on very large lattices. A lattice is a set of linearly independent vectors that are used to compute a linear transformation (Buchmann, 108). The reason this system is so effective, is because it has traits similar to afore mentioned systems. It is efficient to compute the original transformation, which in this scenario, is our

encryption machine. However, it is extremely difficult to decrypt without knowing the key used for the decryption machine. The reason for this is that it is based on the SVP problem, or shortest vector problem which tackles finding the basis of the vector used for the transformation, knowing that any given lattice has infinitely many possible bases (Buchmann, 108-9),(Trappe & Washington, 376-7). The reason this appears to be such a promising successor and natural progression of public key cryptography is due to the “conjectured intractability of lattice problems, like approximating the shortest lattice vector (SVP), even in the quantum-era; additionally, because of their computational efficiency, lattice-based signature schemes seem to be one of the most promising replacements for current constructions” (Buchmann, 105). Ultimately, even the best algorithms designed to crack this encryption system can only produce approximations of the decryption machine, and so even though a quantum computer could surely produce approximation after approximation rapidly, they would be of little use. As I have been working on my own program designed to crack substitution ciphers, I have discovered first hand that an approximate solution will still most likely leave you with complete gibberish. If there is no algorithm that provides an absolute solution, then it is irrelevant as to how fast you are able to produce solutions. This is why lattice based crypto can survive the quantum computing era, whenever that day arrives.

A TRADITIONAL GENETIC ALGORITHM

I have chosen to attack a substitution cipher because it has properties that are suited to a genetic algorithm. The first is that substitution ciphers are one to one, meaning one letter is 'substituted' for another, and there are no overlaps. Modern cryptosystems rely on block ciphering, meaning they operate on groups of letters at once, so any letter will not have a consistent representation as another letter. Substitution ciphers have many known 'smart' attacks that I am able to use as well, and the knowledge of frequency analysis, which is classified as a 'smart' attack on a substitution cipher, has helped me create the fitness function I will need for the genetic algorithm.

The task of breaking a substitution cipher with a genetic algorithm is an extremely complex dilemma, and the first way of approaching it did not work. Initially, I wanted to build a solution that was completely based on the theories of genetic algorithms, where randomness, and true randomness at that, should produce the best possible answer with the correct fitness function. Both fortunately for those who use applied cryptography and unfortunately for me, cryptography is built to have "maximal security against passive attacks" (Damgard, 445). What this means precisely is that using a true random algorithm, even while directing that randomness, the chances of decrypting the encryption are extremely slim.

In fact, I was experiencing this exact issue. I had created a genetic algorithm that could successfully operate on encrypted strings, however, the results of 'decryption' were just as encrypted as the original message. My initial method of approach was to create a gene of 2-letter 'swaps'. What my gene looked like was something like this: ... $(a,b),(c,d),(e,f)$..., and it functioned by reading the gene from left to right, initially interchanging a's with b's, then c's with d's, and finally e's with f's. It would do this for each cycle within the gene. The letters in the cycles were completely random, so in theory, the gene could have contained ... $(a,f)...(f,a)$..., which means that it would first switch the a's to f's, then back to a's. I also did not set the length of the gene to anything specific, and I arbitrarily chose a gene of 50 cycles. I did this to ensure that hopefully I was including all the letters of the alphabet in my decryption, because I was not guaranteeing that each letter would even be represented.

I had built my fitness function in such a way that it would count the occurrences of letters in the decryption, and then compare that to the actual frequency counts for English letters. By doing this, I had hoped that the letters would fall into their respective frequency counts, and that I would be left with a correct decryption. This is another area in which the randomness hurt me instead of helped me. The way the gene mating process worked was to take two genes at random and splice them at a random length. So, if I were going to mate the gene $(a,b),(c,d),(e,f),(g,h)$ with the gene $(j,k),(l,m),(n,o),(p,q)$ at position 2, the resulting genes would be $(a,b),(c,d),(n,o),(p,q)$ and $(j,k),(l,m),(e,f),(g,h)$, effectively giving me two brand new combinations to try. The

theory behind this is similar to Darwinism, in that mating the healthy genes together is going to create an even healthier child, yet 'healthy' in this population seemed to be unobtainable. Sometimes, when the healthiest genes were 'mated', the slightest change in the gene could take a gene from an 80% match back down to a 40% match, and thus I was no better off than how I had started. Essentially, I was flipping a coin with 403,291,461,126,605,635,584,000,000 (26 factorial) sides, and hoping to call it correctly. It is easy to see why this is a largely futile approach. As a result, I decided to completely re-work the problem.

Even though ultimately this solution did not work out, I learned quite a bit from this initial attempt. Firstly, I learned that a pure genetic algorithm, in all of its random glory, is not quite suited to find a single solution that is deemed the correct one. Secondly I learned that it is still possible, even though difficult, to teach the computer to recognize patterns and attribute them to specific English patterns. The largest downfall of this attempt is that the fitness function did not encompass enough data to paint an entire picture. With this fitness function only relying on the frequencies of individual letters, and a handful of the most common digraphs and trigraphs (2 and 3 letter patterns) in the English language, I could not hope to find any sense of communication in my garbled decryptions. I returned to the drawing board, and decided it was time to scale up the search.

BALANCING THEORITICAL COMPLEXITY AND COMPUTATIONAL SIMPLICITY

Coming to this decision meant I needed a fresh start, and so to make a complete fresh start, I abandoned a computer application entirely, and decided to rebuild this program as an Android app. Deciding to move to a mobile platform meant several other things about how I approached the problem needed to change. I needed to reduce the randomness, and therefore reduce the amount of computation, which is important to be aware of when moving into mobile development. The highest quality mobile device will only have 3GB of memory, whereas you'd be hard pressed to find a computer with less than 8GB of memory. Memory equals computing power, and so a streamlined algorithm is necessary if this program is to run on a smartphone.

As a result of this, the biggest change I made is the structure of the gene. Instead of having an unspecified length with unrestricted cycles, I have a gene that contains 26 cycles, in the form (a, _), (b, _) (c,_)... where the first part of the cycle will contain each letter of the alphabet only once. To ensure that algorithm is working with the whole alphabet, the blanks will be filled with another set of the alphabet, in any order. By definition, the fact that it is a set of the alphabet means that each letter only occurs once. Doing this means that each letter occurs once at the start of a cycle, and once at the end of a cycle.

I adapted a fitness function from python code that James Lyons had written to crack simple substitution ciphers to Java, which was necessary since Android

applications are all Java based, and that was not without its fair share of changes. For starters, Java simply could not handle the magnitude of the numbers inherently, so I had to make some modifications to scale it down and prevent overflow errors within the computation. I also gathered a text file with every possible quadragram from James Lyons, however, I had to cut that down too since the size of the file was simply too large to be included in an Android application. To combat this, I simply cut out quadragrams with extremely low frequencies and told the computer to assume a frequency of 1 if the quadragram was not found. In hindsight, Android may have been a poor choice to attempt this with simply due to memory and processing power constraints, yet none-the-less I carried forward because there are still things to be learned from a failing attempt.

The premise of this new genetic algorithm is that of a hill climbing algorithm. This means that I am not trying to swallow the entire pie all at once. Instead, I am looking for one tiny change that even slightly improves the result, from which it continues to climb. The calculation of the fitness has also changed. I am now using a text file that contains every single quadragram in the English language, along with an associated relative frequency of that quadragram. The actual fitness of a particular decryption is: $F(x) = \sum_{i=0}^n \text{Log}_{10}(\frac{Q_i}{\bar{Q}})$. The fitness is the sum of the logarithm of the frequency divided by the average of all the frequencies. The purpose of this fitness function is to allow quadragrams such as 'tion' with a frequency score in the hundreds

of thousands to be strongly weighted over quadragrams like 'axcq', which only has a frequency score of 1. In fact, if a decryption was filled with many quadragrams with low relative frequency scores, $\frac{Q_i}{Q}$ will be less than 1, and the logarithm of anything less than 1 is negative, so bad quadragrams will detract from the fitness score instead of boosting them up, which is exactly what is necessary.

The algorithm that is described above should work; however, I never achieved any meaningful results with it on the Android platform. The entire premise of it is to feed the computer enough information about the English language along with a formula to judge the information it receives so that it can guess how to turn a given string of encoded letters back into plain English. As far as complexity goes, when I passed in an encrypted string of over 200 characters, the resulting decryption took over 10 minutes to process. The biggest reason that it took so long was because I was running it on an Android smartphone, with limited memory, which meant limited processing power. Every few seconds, the log of information notifying me on the status of my app would note that it was freeing up memory, along with how long the process took. Over 25% of the app's runtime was dedicated to keeping enough memory freed to even run. It is quite possible that a desktop app would perform much better, since computers easily have four times the amount of resources a smartphone has, or it is also possible that a computer also would have returned gibberish over and over. But, I am not taking this as a failure. In the many hours spent trying to fit the pieces of the algorithm together, I continually found myself asking 'But how will the computer know?!' as I would work out

what I was trying to do in plain English, but of course, plain English means nothing to a computer.

While running tests on the application, the ultimate realization that I needed more computing power to truly test this application led to a proof of concept desktop application to test the validity of the algorithm. I stripped the heart of the computation out of the Android application and reduced it to the only necessary process. From there, I indefinitely iterated through the algorithm on one encryption continually comparing it to the known decryption in order to verify any progress as it happened. After several hours of searching for potential decryptions, I was able to get results. It was rather interesting to see that the decryption would fluctuate. Sometimes, the decryption would match the 'm', 'e', and 'a' correctly, while other times it would match the 't', 'h', and 'e'. This is somewhat to be expected, because just like with the first attempt, and slight changes in the key can greatly affect the results. At the same time, it continually was able to match subsets of the most common English languages, even though the first attempt never achieved any consistency on this scale. All in all, this was a mini success! The fact that I was able to teach the computer how to pick out popular letters in the English language using complete randomness is astounding. Suddenly, the size of my quarter from the first example has shrunk.

While I am not completely satisfied with my solution, I did still produce results. The main reason that the proof of concept was able to achieve this measure of success is because it has more computing power, and more time than an Android app is

allowed. When working with a mobile phone, if something does not happen near instantaneously the user almost immediately becomes bored. To make matters worse, the runtime environment of Android will try to stop the app from running if any calculation takes more than a few seconds, deeming the app as unresponsive.

Therefore, returning to the desktop seemed like the correct move because I had more computational power, and I could leave the algorithm searching for a solution over night without any system process trying to halt the search.

MAN VERSUS MACHINE: THE LINE IN THE SAND?

I set out on a mission to solve substitution ciphers with my phone, but what I discovered instead was something completely different. I have no absolute solution to substitution ciphers to show other than what I can guess at while looking at a substitution cipher in the newspaper. But that in itself is an important realization. When I see 'QSTHWLQSVV' with the clue 'bouncy and orange' in the newspaper, my logical guess for the word is 'BASKETBALL', but that is not guessed from the jumble of letters alone, that is guessed from the clue, and then when I match the letters of basketball to the letters of the cipher, I can be sure that my guess is sound and therefore correct because the letters match up. The computer does not have this luxury. A computer only knows what is 'bouncy and orange' if we explicitly tell it so, so if a computer was fed the same cipher string, the only thing it has to go off of is the statistics of the language itself.

Ultimately, we as humans possess something that computers do not. We have a self, an 'I', the ability to go back into our memories that are truly ours and pull thoughts and ideas that enrich the current calculation our minds are making (Gelernter, 7). This is the exact downfall of a computer trying to crack a substitution cipher. It has no memory that we do not give it. It cannot evoke phrases and context unless we have explicitly told it to use those memories in the given calculation. Human minds do not work like that. Our minds wander on their own, and we associate every thought we have with

thoughts from our past. This is the key difference between intelligence and artificial intelligence. At best, computers can only 'fake' at these complex human thoughts, and to do complex calculations requires a lot of processing power, which is something that I think my android smartphone was unable to deliver (Gelernter, 11). I had set out to solve a substitution cipher, but instead I learned nuances to what a machine can do, and what it cannot.

Separating it in this fashion illustrates the difference between the human mind and the digital mind we call a computer. The computer only has the computations it has been told to go off of, and in my particular case, I told it relative frequencies for every quadragram as well as a formula to compute how good the English was at the end of the decryption. The human mind is able to make connections between the clue and the ciphered text. Even without any knowledge of cryptography or letter frequencies, we can cut our guess down to a handful of words just by knowing 'bouncy and orange'. Furthermore, we have some idea of what are allowable words. Since my genetic algorithm only knows 'English' in four-letter chunks and the frequencies associated with those chunks, it is going to return things that are not words.

Returning to the initial criteria of when to use a genetic algorithm, the very last rule specified that genetic algorithms were best suited when a 'global optimum' does not need to be found. The one downfall to attempting this algorithm on a smartphone is that the processing power hinders countermeasures to the fact that we are seeking a global optimum, because there is but one key that will turn the cipher text back into

plain English. The way to counter-act this is to encase the entire body of the genetic algorithm in an infinite while loop. Doing this would make the computer re-calculate the key from infinitely many starting points, which would make our key end up at infinitely many local maximums. Computationally speaking, this is an intense, time-consuming operation which does not fit well with an Android application, where users expect the operation to be nearly instantaneous. If the genetic algorithm were looping indefinitely, we can assume that one of those local maximums is also the absolute maximum which would be the key to our substitution cipher.

If I were to test this theory in full, it would depend on processing power, and to get the answer in a meaningful amount of time would rely on parallel processing. The same theories that I speculate will change modern cryptography would also enhance my own solution to crack a substitution cipher with a genetic algorithm. If a quantum computer type machine existed, then I could take any given cipher, and instantaneously gather every local maximum that the fitness function will produce on the cipher, and from there it would be a matter of sorting the maximums to find the absolute maximum which should be the expected decryption.

There are still flaws with this approach though, since the fitness function relies entirely on the encoded message to follow the statistical frequencies of the English language, such as expecting 'e' to be the most common letter, 'the' to be the most common trigraph, and 'tion' to be the most common quadragram. Since the program is actively scoring combinations of letters, it can compensate for slight statistical deviation,

but if the message was all about 'zebras', the chances of this type of statistical attack working are slim, since 'z' is not very high up in any letter frequency counts.

Overall, this is an immensely complex problem, and the ability for a computer to solve it efficiently will come down to parallel computing, whether it is done in the traditional and modern state, or if one day there are quantum computers capable of perfect parallel computation. When a human looks at a substitution cipher with a clue, many different areas of our brain come together to create the most logical answer, so in a similar way the computer needs many processing threads to solve a hard problem fast (Gelernter, 11). The human mind will rely on past experiences and memories of what fits the criteria, the computer will only have its numbers. I think that finding all the local maximums of a substitution cipher is much more computationally intensive for a computer than it is for a human to reach into their memories for possible solutions, and that is the beauty of the problem. Artificial life was and still is about studying natural life patterns and finding ways to effectively implement them into a computational machine.

The current state of cryptography is in good hands. There is some incredible math behind it all, and I would argue that modern cryptosystems play off the notion that there are some things that are easy for a human mind to do that are difficult for a computer to replicate or 'fake'. All of today's modern cryptosystems rely on being based off of computationally intensive problems, problems that, without some un-invented super-computer will remain unsolved problems. Cryptography has the job of

protecting our secrets, digital or otherwise, and computer science has elevated cryptography to a level that before was only dreamed of. We have come a long way from the original Caesar cipher and slightly more complex substitution cipher. I think that our understanding of cryptography will continue to outpace our ability to compute solutions, simply because of that line between organic thought and machine processing. Therefore, I think that our digital secrets, for better or for worse, are safe for centuries to come.

The substitution cipher defeated my Android app. Yet it is simple enough to be included in newspaper puzzles, and assigned to undergrad's as homework (without a genetic algorithm). Because machines grow more intelligent as computing power rises, it is easy to see how a smartphone, even though it is powerful in its own right, is not 'intelligent' enough to run an attack on even simple substitution ciphers. To make matters more complicated, modern cryptosystems are specifically designed to eliminate the possibility of any fitness functions to rate decryptions. I struggled creating an adequate fitness function on a system that has been around longer than the formal science of cryptography, knowing that fitness functions exist. The issue was finding the right combination of various theories for measuring substitution ciphers. Undergrads use pure frequency analysis, yet the gap when moving this to the computer is that we have the English skills to fill in the blanks when the frequency analysis fails. Computers do not. Merging the finesse of the human brain and the raw computational power of computers has been the issue since the inception of computers. Computers will

continue to gain more power, and we will continue to find better ways to mimic the finesse that is the human mind and translate it to code. There is a line in the sand, there have been many lines in the sand, and we continue to cross it.

Bibliography

- Anthes, Gary. "French-Team Invents Faster Code Breaking Algorithm." *Communications of the ACM* 57.1 (2014): 21-23. Print.
- Buchmann, Johannes, et al. "Post-Quantum Cryptography: Lattice Signatures." *Computing* 85.1/2 (2009): 105-125. Business Source Complete. Web. 16 Nov. 2014.
- Damgard, Ivan. "Towards Practical Public Key Systems Secure Against Chosen Ciphertext Attacks." *Advances in Cryptology* (1992): 445-55. Print.
- Gelernter, David. "How hard is chess?." *Time* 19 May 1997: 72. Academic Search Premier. Web. 1 Mar. 2015.
- Goldberg, David E. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Crawfordsville: Addison Wesley Longman, 1989. Print.
- Landau, Susan. "Designing Cryptography for the New Century." *Communications of the ACM* 43.5 (2000): 115-120. Print.
- Lyons, James. "Cryptanalysis of the Simple Substitution Cipher." *Practical Graphy*. 1 Jan. 2012. Web. 27 Mar. 2015.
- Mitchell, Melanie. *An introduction to genetic algorithms*. Cambridge, Mass.: MIT Press, 1998-1996. Print.
- Mone, Gregory. "Future-Proof Encryption." *Communications of the ACM* 56.11 (2013): 12-14. Print.
- Rieffel, Eleanor, and Wolfgang Polak. "An Introduction to Quantum Computing for Non-

physicists." ACM Computing Surveys 32.3 (2000): 300-35. Print.

Trappe, Wade, and Lawrence C. Washington. *Introduction to cryptography: with coding theory*. 2nd ed. Upper Saddle River, N.J.: Pearson Prentice Hall, 2006. Print.

Appendix A: SOURCE CODE FROM FIRST GENETIC ALGORITHM

This class, PopulationManager, is the main Java class responsible for the calculation of the original genetic algorithm. The gene is based on the original concept of letter pairs used to swap letters within the decryption. New genes are produced by splicing two parent genes and merging the halves from the two different genes.

```
import domain.*;
import java.util.*;

/**
 *
 * @author BrandonWard
 */

public class PopulationManager {

    public Population calculate(Population
        population) {
        //This is the main method and this controls
        the
        //subsequent method calls per calculation
        population = calculateMutations(
            population);
        population = calculateFitnessAverage(
            population);
        population = calculateMatingPool(
            population);
        population = calculatePairing(population
            );
        population = calculateCrossover(
            population);
        return population;
    }

    private Population calculateMutations(
        Population population) {
        Iterator iter = population.getGenes().
```

```

        iterator();
Random mValue = new Random();
Random randomC = new Random();
int randomChar;
int m;
while (iter.hasNext()) {
    Gene gene = (Gene) iter.next();
    char [][] geneString = gene.getGene()
        ;
    for (int i = 0; i < gene.getSize();
        i++) {
        for (int j = 0; j < 2; j++) {
            m = mValue.nextInt(10000);
            if (m == 50) {
                randomChar = randomC.
                    nextInt(26) + 97;
                geneString[i][j] = (char
                    ) randomChar;
            }
        }
    }
}
return population;
}

```

```

private Population calculateFitnessAverage(
Population population) {
    double fitnessAvg = 0;
    double i = 0;
    String cipher = population.getCipherText
        ();
    Iterator iter = population.getGenes().
        iterator();
    while (iter.hasNext()) {
        Gene gene = (Gene) iter.next();
        gene.setDecryption(decode(gene,
            cipher));
        double fitness =
            calculateGeneFitness(gene);
        gene.setFitness(fitness);
        fitnessAvg += fitness * gene.
            getCount();
        i += gene.getCount();
    }
}

```

```

    }
    fitnessAvg = fitnessAvg / i;
    population.setFitnessAvg(fitnessAvg);
    return population;
}

private Population calculateMatingPool(
    Population population) {
    List<Gene> sortGene = new LinkedList<
        Gene>();
    sortGene = population.getGenes();
    Collections.sort(sortGene, new
        Comparator<Gene>() {
            @Override
            public int compare(final Gene gene1,
                final Gene gene2) {
                return compareTo(gene1.
                    getFitness(), gene2.getFitness
                    ());
            }

            private int compareTo(double
                fitness1, double fitness2) {
                return java.lang.Double.compare(
                    fitness1, fitness2);
            }
        });
    population.setGenes(sortGene);
    List<Gene> genes = new LinkedList<Gene
        >();
    Iterator<Gene> iter = population.
        getGenes().iterator();
    int i = 0;
    while (iter.hasNext() && i < 100) {
        Gene gene = iter.next();
        //System.out.println(gene.getFitness
            ());
        genes.add(gene);
        i++;
    }
    population.setGenes(genes);
    return population;
}

```

```

private Population calculatePairing(
    Population population) {
    List<Gene> allGenes = population.
        getGenes();
    List<Pairing> pairings = new LinkedList
        ();
    int size = allGenes.size() / 2;
    //create the pairings of genes
    for (int i = 0; i < size; i++) {
        Pairing p = new Pairing();
        p.setGeneA(allGenes.get(i));
        p.setGeneB(allGenes.get(i + size));
        if (!(p.getGeneA().equals(p.getGeneB
            ()))) {
            pairings.add(p);
        }//end if the pairings do not have
        identical genes
    }//end for int i < size
    population.setPairings(pairings);
    return population;
}

```

```

private Population calculateCrossover(
    Population population) {
    Random lValue = new Random();
    Iterator iter = population.getPairings()
        .iterator();
    List<Gene> genes = population.getGenes()
        ;
    int size = population.getGenes().get(0).
        getSize();
    while (iter.hasNext()) {
        Pairing pair = (Pairing) iter.next()
            ;
        char [][] geneA = pair.getGeneA().
            getGene();
        char [][] geneB = pair.getGeneB().
            getGene();
        int l = lValue.nextInt(size - 1) +
            1;
        char [][] geneC = new char[size][2];
        char [][] geneD = new char[size][2];
    }
}

```

```

    for (int i = 0; i < size; i++) {
        if (i < 1) {
            geneC[i] = geneA[i];
            geneD[i] = geneB[i];
        } else {
            geneC[i] = geneB[i];
            geneD[i] = geneA[i];
        }
    }
    Gene g1 = new Gene(size);
    g1.setGene(geneC);
    Gene g2 = new Gene(size);
    g2.setGene(geneD);
    if (genes.contains(g1)) {
        Gene thisGene = genes.get(genes.
            indexOf(g1));
        thisGene.setCount(thisGene.
            getCount() + 1);
    } else {
        g1.setCount(1);
        g1.setDecryption(decode(g1,
            population.getCipherText()));
        g1.setFitness(
            calculateGeneFitness(g1));
        genes.add(g1);
    }
    if (genes.contains(g2)) {
        Gene thisGene = genes.get(genes.
            indexOf(g2));
        thisGene.setCount(thisGene.
            getCount() + 1);
    } else {
        g2.setCount(1);
        g2.setDecryption(decode(g2,
            population.getCipherText()));
        g2.setFitness(
            calculateGeneFitness(g2));
        genes.add(g2);
    }
}
List<Gene> sortGene = new LinkedList<
Gene>();

```

```

sortGene = genes;
Collections.sort(sortGene, new
    Comparator<Gene>() {
        @Override
        public int compare(final Gene gene1,
            final Gene gene2) {
            return(compareTo(gene2.
                getFitness(), gene1.getFitness
                    ()));
        }

        private int compareTo(double
            fitness1, double fitness2) {
            return java.lang.Double.compare(
                fitness1, fitness2);
        }
    });
population.setGenes(sortGene);
return population;
}
private String[] englishLetters = {"a", "b",
    "c", "d", "e", "f", "g", "h", "i", "j", "
k", "l", "m", "n", "o", "p", "q", "r", "s",
    "t", "u", "v", "w", "u", "x", "y", "z",
    "th", "he", "an", "in", "er", "on", "re",
    "ed", "nd", "ha", "at", "en", "es", "of",
    "nt", "ea", "ti", "to", "io", "le", "is",
    "ou", "ar", "as", "de", "rt", "ve", "the",
    "and", "tha", "ent", "ion", "tio", "for",
    "nde", "has", "nce", "tis", "oft", "men",
    "ing", "edt", "sth", "ss", "ee", "tt", "
ff", "ll", "mm", "oo"};
private final double[] englishFrequencies =
{.082, .015, .028, .043, .127, .022, .020,
    .061, .070, .002, .008, .040, .024, .067,
    .075, .019, .001, .060, .063, .091, .028,
    .010, .023, .001, .020, .001, /* end
single letter count*/
6.5, 6.25, 6, 5.75, 5.5, 5.25, 5, 4.75, 4.5, 4.25, 4, 3.75, 3.5, 3.25, 3, 2
    /* end digraph letter count*/
8, 7.5, 7, 6.5, 6, 5.5, 5, 4.5, 4, 3.5, 3, 2.5, 2, 1.5, 1, .5,
    /* end trigraph letter count*/

```

```

        2.33,2,1.66,1.33,1,.66,.33 /* end double
        letter count"*/};
private double[] tempFreq = new double[
    englishLetters.length];

public String decode(Gene gene, String
    cipher) {
    char[] cipherChars = cipher.toCharArray
        ();
    char[][] geneCycle = gene.getGene();
    for (int i = gene.getSize() - 1; i >= 0;
        i--) {
        for (int k = 0; k < cipherChars.
            length; k++) {
            if (cipherChars[k] == geneCycle[
                i][0]) {
                cipherChars[k] = geneCycle[i
                    ][1];
            } else if (cipherChars[k] ==
                geneCycle[i][1]) {
                cipherChars[k] = geneCycle[i
                    ][0];
            }
        }
    }
    String plainText = "";
    for (int l = 0; l < cipherChars.length;
        l++) {
        plainText += cipherChars[l];
    }
    return plainText;
}

public double calculateGeneFitness(Gene gene
) {
    double[] letterCount =
        calculateFrequencies(gene.
            getDecryption());
    double fitness = 0;
    for (int i = 0; i < 26; i++) {
        fitness += letterCount[i] *

```

```

        getEnglishFrequencies()[i];
    }
    return fitness;
}

private double[] calculateFrequencies(String
plainText) {
    char[] plain = plainText.toCharArray();
    for (int i = 0; i < englishLetters.
length; i++) {
        double sCount = 0;
        double diCount = 0;
        double triCount = 0;
        double doCount = 0;
        int m = i + 26;
        int n = i + 52;
        int o = i + 68;
        for (int j = 0; j < plain.length; j
++) {
            if (i < 26) {
                String cipher = String.
copyValueOf(plain, i, 1);
                if (cipher.equals(
englishLetters[i])) {
                    sCount++;
                }
            }
            if (m >= 26 && m < 52) {
                String cipher = String.
copyValueOf(plain, m, 2);
                if (cipher.equals(
englishLetters[m])) {
                    diCount++;
                }
            }
            if (n >= 52 && n < 68) {
                String cipher = String.
copyValueOf(plain, n, 3);
                if (cipher.equals(
englishLetters[n])) {
                    triCount++;
                }
            }
        }
    }
}

```



```

    }
    if (o >= 68 && o <
        englishLetters.length) {
        String cipher = String.
            copyValueOf(plain, o, 2);
        if (cipher.equals(
            englishLetters[o])) {
            doCount++;
        }
    }
}
if (i < 26) {
    tempFreq[i] = sCount;
}
if (m >= 26 && m < 52) {
    tempFreq[m] = diCount;
}
if (n >= 52 && n < 68) {
    tempFreq[n] = triCount;
}
if (o >= 68 && o < englishLetters.
    length) {
    tempFreq[o] = doCount;
}
}
return tempFreq;
}

```

```

public Gene calculateStrongest(Population
    population) {
    Iterator iter = population.getGenes().
        iterator();
    Gene fittestGene = new Gene(population.
        getGenes().get(0).getSize());
    double bestFitness = 0;
    while (iter.hasNext()) {
        Gene gene = (Gene) iter.next();
        if (gene.getFitness() > bestFitness)
        {
            fittestGene = gene;
            bestFitness = gene.getFitness();
        }
    }
}

```

```
    }  
    return fittestGene;  
}  
  
/**  
 * @return the englishFrequencies  
 */  
public double[] getEnglishFrequencies() {  
    return englishFrequencies;  
}  
}
```

Appendix B: SOURCE CODE FROM SECOND GENETIC ALGORITHM

This is the second attempt at the genetic algorithm, and this attempt returned more fruitful results even if it was still not the completely correct answer. This class was in charge of loading the text file containing the quadragrams with their frequencies, loading the Map of quadragrams with their scores, and then finally iteratively computing the genetic algorithm, but this time it used the hill climbing technique to slowly work its way to the correct answer.

```
import android.content.Context;
import android.util.Log;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.math.BigDecimal;
import java.math.RoundingMode;
import java.util.ArrayList;
import java.util.Collections;
import java.util.HashMap;
import java.util.Iterator;
import java.util.List;
import java.util.Map;
import java.util.Random;

import regishonorsthesis.brandonward.
    cryptogeneticalgorithmthesis.R;
import regishonorsthesis.brandonward.
    cryptogeneticalgorithmthesis.domain.Quadragram
    ;

/**
 * Created by BrandonWard on 1/29/2015.
 */
public class DecryptionManagerQuadragram
    implements IDecryptionMgr {
```

```

private List<Quadragram> quadragrams;
private Map<Integer , Double>
    quadragrams_scored;
private Context context;
private List<Character> parentKey;
private double parentScore;
private String encryption;
private double maxScore = -99e9;
private List<Character> maxKey;
private double notFoundScore = 0;

public DecryptionManagerQuadragram(Context
    context) {
    this.context = context;
    init();
}

private void init() {
    quadragrams = new ArrayList<Quadragram
        >();
    quadragrams_scored = new HashMap<Integer
        , Double>();
    LoadFile(R.raw.englishquadgrams);
}

private void LoadFile(int resId) {
    // The InputStream opens the resourceId
    // and sends it to the buffer
    InputStream is = context.getResources().
        openRawResource(resId);
    BufferedReader br = new BufferedReader(
        new InputStreamReader(is));
    String readLine = null;

    try {
        // While the BufferedReader readLine
        // is not null
        while ((readLine = br.readLine()) !=
            null) {
            readLine = readLine.toUpperCase
                ();
            String[] result = readLine.split
                ("_");
        }
    }
}

```

```

        int score = Integer.parseInt(
            result [1]);
        Quadragram quadragram = new
            Quadragram(result [0], score);
        quadragrams.add(quadragram);
        //Log.d("TEXT", readLine);
    }
    // Close the InputStream and
    // BufferedReader
    is.close();
    br.close();

} catch (IOException e) {
    e.printStackTrace();
}
score_init();
}

@Override
public String decrypt(String encryption) {
    encryption = encryption.toUpperCase();
    char [] temp = encryption.toCharArray();
    char [] midTemp = new char [temp.length];
    int j = 0;
    for (int i = 0; i < temp.length; i++) {
        if (temp[i] >= 'A' && temp[i] <= 'Z'
            ) {
            midTemp[j] = temp[i];
            j++;
        }
    }
    char [] endTemp = new char [j];
    for (int k = 0; k < j; k++) {
        endTemp[k] = midTemp[k];
    }
    encryption = String.valueOf(endTemp)
        ;
    this.encryption = encryption;
    List<Character> parentKey = new
        ArrayList<Character>(26); //Generate
        parentKey and shuffle it before
        setting it to the class

```

```

for (int i = 'A'; i <= 'Z'; i++) {
    parentKey.add((char) i);
}
this.parentKey = parentKey;
String deciphered;
int iterations = 0;
Random r = new Random();
while (iterations < 50) {//This method
    is no longer the piece that bogs down
    the system entirely, that largely
    happens in score_init()
    Collections.shuffle(this.parentKey);
    deciphered = decipher(this.parentKey
    );
    parentScore = score(deciphered);
    if (parentScore > maxScore) {
        maxScore = parentScore;
        maxKey = this.parentKey;
    }
    int count;
    for (count = 0; count < 1000; count
    ++) {
        List<Character> childKey = this.
        parentKey;
        int a = (r.nextInt(26));
        int b = (r.nextInt(26));
        Character swap;
        swap = childKey.get(a);
        childKey.set(a, childKey.get(b))
        ;
        childKey.set(b, swap);
        String childDecipher = decipher(
        childKey);
        double childScore = score(
        childDecipher);
        if (childScore > parentScore) {
            this.parentKey = childKey;
            parentScore = childScore;
            count = 0;
            if (parentScore > maxScore)
            {
                maxKey = this.parentKey;
                maxScore = parentScore;
            }
        }
    }
}

```

```

        Log.i("Iteration_" +
            iterations, "maxScore_
            Increased" + maxScore);
    }
}
}
iterations++;
}
Log.i("Final_Score", "=_" + maxScore);
return decipher(maxKey);
}

private String decipher(List<Character> key)
{
    char[] toDecrypt = encryption.
        toCharArray();
    for (int i = 0; i < toDecrypt.length; i
        ++){
        if (toDecrypt[i] >= 'A' && toDecrypt
            [i] <= 'Z') {
            toDecrypt[i] = key.get((
                toDecrypt[i] - 'A'));
        }
    }
    return String.valueOf(toDecrypt);
}

private double score(String deciphered) {
    double score = 0;
    char[] decipherChar = deciphered.
        toCharArray();
    for (int i = 0; i < deciphered.length()
        - 3; i++) {
        String string = String.valueOf(
            decipherChar, i, 4);
        if (quadrgrams_scored.containsKey(
            string.hashCode())) {
            score += quadrgrams_scored.get(
                string.hashCode());
        } else {
            score += notFoundScore;
        }
    }
}

```

```

        return score;
    }

    private void score_init() {
        notFoundScore = Math.log10
            (0.00000000000000001);
        Iterator<Quadragram> quadragramIterator
            = quadragrams.iterator();
        BigDecimal total = BigDecimal.ZERO;
        while (quadragramIterator.hasNext()) {
            Quadragram quadragram =
                quadragramIterator.next();
            total = total.add(BigDecimal.valueOf
                (quadragram.getCount()));
        }
        //total = total.divide(BigDecimal.
            valueOf(quadragrams.size()), 10,
            RoundingMode.HALF_UP);
        quadragramIterator = quadragrams.
            iterator();
        while (quadragramIterator.hasNext()) {
            Quadragram quadragram =
                quadragramIterator.next();
            BigDecimal division = BigDecimal.
                valueOf(quadragram.getCount());
            division = division.divide(total,
                15, RoundingMode.HALF_UP);
            double value = division.doubleValue
                ();
            value = Math.log10(value);
            if (!(value > 0)) {
                value = notFoundScore;
            }
            quadragrams_scored.put(quadragram.
                toString().hashCode(), value);
        }
    }
}

```