Automated Cryptanalysis of Monoalphabetic Substitution Ciphers Using Stochastic Optimization Algorithms

Rod Hilton

1 Introduction

All forms of symmetric encryption take a key shared between a small group of people and encode data using this key so that only those with the key are able to decrypt it. Encryption algorithms tend to rely on problems that are computationally intractable for security, but even more generally these algorithms rely on a fundamental assumption: that the number of potential keys is so large that it cannot be searched via brute force for the correct key in a reasonable amount of time.

Certain ciphers – such as Monoalphabetic Substitution Ciphers – have extremely large keyspaces, seemingly avoiding this weakness. However, these ciphers exhibit a particular property that actually makes it feasible to search these impossibly large keyspaces efficiently, making them vulnerable to cryptanalysis via randomized search algorithms.

This work and its companion code implementation explore the property that makes certain ciphers vulnerable to this form of attack and utilizes the weakest and least efficient of these search algorithms to illustrate the severity of this vulnerability.

2 Monoalphabetic Substitution Ciphers

The substitution cipher, one of the oldest forms of encryption algorithms according to [Sin00], takes each character of a plaintext message and uses a substitution process to replace it with a new character in the ciphertext. This substitution method is deterministic and reversible, allowing the intended message recipients to reverse-substitute ciphertext characters to recover the plaintext.

One particular form of substitution cipher is the Monoalphabetic Substitution Cipher, often called a "Simple Substitution Cipher". Monoalphabetic Substitution Ciphers rely on a single key mapping function K, which consistently replaces a particular character α with a character from the mapping $K(\alpha)$. For encryption function E and decryption function D with plaintext P and ciphertext C, $|P| = |C|$ and $\forall i, 0 < i \leq |P|, C_i = E(P_i) = K(P_i), P_i = D(C_i)$. The mapping is one-to-one, so for all characters α, β in the plaintext, $\alpha = \beta \implies K(\alpha) = K(\beta)$ and $\alpha \neq \beta \implies K(\alpha) \neq K(\beta)$ ([Bis02]).

As an example, if we take the plaintext $P = \texttt{HELLO WORLD}$ and we use:

$$K(x) = \begin{cases} \texttt{M} & x = \texttt{D} \\ \texttt{V} & x = \texttt{E} \\ \texttt{I} & x = \texttt{H} \\ \texttt{F} & x = \texttt{L} \\ \texttt{J} & x = \texttt{O} \\ \texttt{K} & x = \texttt{R} \\ \texttt{R} & x = \texttt{W} \end{cases}$$

The resulting ciphertext would be $C = \texttt{IFVVJ FJKFM}$.

For simplicity, Monoalphabetic Substitution Cipher keys are typically expressed as a permutation of the 26 letters of the alphabet, such as $K = \texttt{MQLDEHNWKZOAPXVUTCYISBFRGJ}$. With this notation, each character in an lexicographic ordering of the letters of the alphabet maps to the character in K that shares its position, so $K(\texttt{A}) = \texttt{M}, K(\texttt{B}) = \texttt{Q}, K(\texttt{C}) = \texttt{L}, \ldots K(\texttt{Z}) = \texttt{J}$. Using this notation, $K = \texttt{YTUMVCLINDAFEZJBXKHPOWRQSG}$ could represent the key in the previous example.

2.1 Cryptanalysis

Since the set of possible keys is the set of all possible permutations of the alphabet, Monoalphabetic Substitution Ciphers have a keyspace of 26!, which is over 403 septillion. If someone were able to check 1,000,000 keys per second, it would still take over 12 trillion years to check all possible keys, so cryptanalysis by brute force is infeasible.

If Eve were to intercept an encrypted ciphertext C from Alice to Bob, she could rely on her knowledge of the language the message was written in and use frequency analysis. Knowing that the most common English letter, `E`, occurs 12.7% of the time would allow Eve to assume the most common letter in C is mapped to by `E`. The next most-common letters according to [Bek82] are `T` at 9.1%, `A` at 8.2%, `O` at 7.5%, `I` at 7%, `N` at 6.7%, and `S` at 6.3%. These single-letter frequencies, generally referred to as unigram frequencies, are well-known for the English language, and illustrated in Figure 1.

After `E`, unigram frequencies are too close to each other to help, but Eve could look look beyond unigram frequencies and compare pairs and triples of letters (bigrams and trigrams, respectively), which are detailed in [SBJ79]. Using these frequencies, Eve can make a series of informed hypotheses about letter substitutions and test them, looking for words or phrases that she recognizes. This process is time-consuming and involves a great deal of guesswork as outlined in [TW05], so the goal of any automated cryptanalysis tool should be to use this methodology to automate the process.

2.2 Properties of Substitution Ciphers

Substitution Ciphers differ from many other ciphers in that similar keys have similar utility. Take RSA as a point of contrast: if Eve were to guess at Alice's p and q and only be off by a

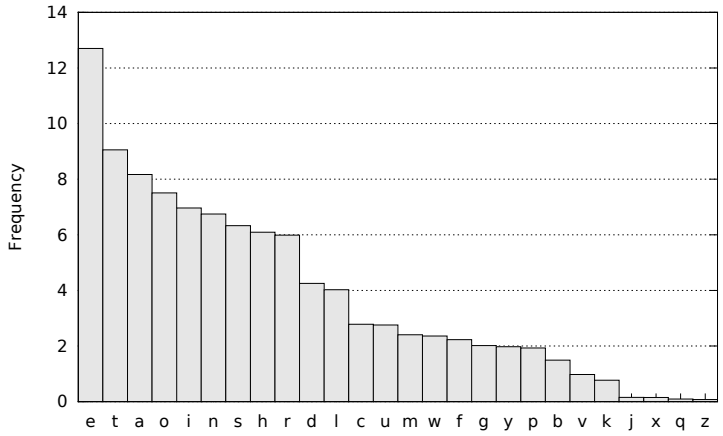

Figure 1: Unigram frequencies graphed from [Lew00]

small amount, while Evan were to guess at Alice's p and q but be off by significantly more, both Eve and Evan's attempted decryptions would be equally unintelligible, with virtually no utility. With substitution ciphers, on the other hand, this is not the case.

Imagine the message `MEETATMIDNIGHT` were encrypted with a Monoalphabetic Substitution Cipher using the key `SECURITYZXWVQPONMLKJHGFDBA`, resulting in a ciphertext of `QRRJSJQZUPZTYJ`. If Eve were to attempt to decrypt with key `YHEUNATLJICDKBMRSWOXVPFQZG`, the resulting partially decrypted text is `XPPIQIXYDVYGAI` which only matches the actual plaintext in two out of fourteen positions. Meanwhile, if Evan decrypts with a key of `HODURYTVZGCXQPLMAISJFNWEBK`, the resulting partially decrypted text is `MEETSTMIDNIGFT`, which matches twelve of fourteen positions of the original plaintext. While neither key is the correct key, Evan's key is objectively better than Eve's.

More formally, a cipher exhibits a property which this text refers to as the *Utility of Partial Solutions* if, given plaintext P, ciphertext C, decryption function D, key K, and a difference function L (such as a [Lev66] distance function for text), there exist many pairs of keys K_a and K_b such that $K_a \neq K_b$ and $L(P, D(C, K_a)) < L(P, D(C, K_b)) \wedge L(K, K_a) < L(K, K_b)$. In other words, if there are many pairs of keys in the keyspace where the decryption of the ciphertext by the key more similar to the correct key more closely resembles the plaintext than the decryption of the ciphertext by the other key, the cipher has *Utility of Partial Solutions*.

> **Conjecture:** Any cipher which exhibits the *Utility of Partial Solutions* property can be efficiently broken by searching the keyspace via optimized search algorithm.

Though left unproven by this work, this conjecture makes intuitive sense. If there is a correlation between the degree to which a key resembles the correct key and the degree to which that key's decryption of the ciphertext resembles the plaintext, it should be possible to search the keyspace efficiently by quickly discarding keys that are "worse" than whatever key

is the closest match at any moment, climbing ever closer to the optimal key without knowing it initially. More specifically, these keyspaces can be searched via Stochastic Optimization Algorithms, covered in Section 3.

3 Stochastic Optimization

Stochastic Optimization Algorithms (often referred to as "metaheuristics") employ randomization to find near-optimal solutions to computationally intractable problems. [Luk11] explains they are used to solve "I know it when I see it" problems, meaning that they work when it's difficult or impossible to come up with a solution to a problem via efficient algorithm, but if somehow given a solution, it would be possible to efficiently test it and grade it.

One of the first metaheuristics, the evolutionary algorithm, was first proposed by [Rec65] in 1965 and later adapted into the more commonly known "genetic algorithm" by [Hol75]. Though less efficient as a stochastic optimization algorithm than more recent algorithms such as Firefly, Cuckoo search, or Particle Swarm, evolutionary algorithms are interesting from a historical perspective. Additionally, they have reductive value; since most metaheuristics are improvements over simple evolutionary algorithms in terms of either space or time efficiency, it's reasonable to assume that any problem solvable by an evolutionary algorithm can also be solved by another metaheuristic. As such, the evolutionary strategy algorithm proposed by [Rec65] will be the primary focus of this work.

```
EVOLUTION-STRATEGY()
 1  μ ← number of parents selected
 2  λ ← number of children generated by the parents
 3  P ← {}
 4  for λ times
        do
 5          P ← P ∪ {new random individual}
 6  Best ← □
 7  repeat
 8          for P_i ∈ P
                do
 9                  AssessFitness(P_i)
10                  if Best = □ or Fitness(P_i) > Fitness(Best)
                      then
11                        Best ← P_i
12          Q ← the μ individuals whose fitness is greatest
13          P ← {}
14          for Q_i ∈ Q
                do
15                  for λ/μ times
                        do
16                        P ← P∪ Mutate(Copy(Q_j))
17    until Best is the ideal solution or time has run out
18  return Best
```

Figure 2: Pseudocode for an Evolutionary Strategy from [Luk11]

The evolution strategy works as outlined in Figure 2. In short, an evolutionary strategy first creates a population of random solutions to the problem, scores each of them according to how well they solve the problem, then creates a new population by applying small mutations to the best-scoring individual from the previous population, then repeats that process. On each repetition, the quality of the candidate solution improves.

Because Monoalphabetic Substitution Ciphers exhibit the *Utility of Partial Solutions*, it should be possible to use an evolutionary algorithm to climb closer and closer to the optimal key for any given ciphertext.

4 Fitness and Frequency Analysis

To use a Stochastic Optimzation attack on Substitution Ciphers, we will need a way to define the effectiveness of a key, generally referred to as a fitness function. Nearly all fitness functions used to evaluate how well a particular key performs rely on some form of Frequency Analysis. As with the manual trial-and-error process given in Section 2.1, the fundamental question when relying on Frequency Analysis is "how much does this decryption resemble English?" A formal process of Frequency Analysis allows us to quantify exactly how much a particular text looks like the language in which the plaintext was written.

Frequency Analysis is done by scanning a corpus of text and, for each unigram i, bigram ij, and trigram ijk incrementing a counter Ct^u_i, Ct^b_{ij}, and Ct^t_{ijk}, respectively. Once these counts are tabulated, one can sum the total number of all unigrams, bigrams, and trigrams $S_u = \Sigma_i(Ct^u_i)$, $S_b = \Sigma_{ij}(Ct^b_{ij})$, $S_t = \Sigma_{ijk}(Ct^t_{ijk})$. Finally, we can derive fractional **reference** frequencies for each unigram, bigram, and trigram using $R^u_i = Ct^u_i \div S_u$, $R^b_{ij} = Ct^b_{ij} \div S_b$, and $R^t_{ijk} = Ct^t_{ijk} \div S_t$. Scanning a text for the counts can be done in $O(n)$ time by using 3 separate hash tables and walking through the text exactly once. Then summing and calculating the frequencies takes, at worst, $O(26 + 26^2 + 26^3)$, which is a constant (though a large one), so the total runtime for analysis is $O(n)$.

In order to score the fitness of a candidate key K_c, we can utilize the reference frequencies and compare them to the frequencies of unigrams, bigrams, and trigrams within the text resulting from the decryption of the ciphertext using key K_c, which we can calculate using the same $O(n)$ process. If we call these partial decryption frequencies for unigrams, bigrams, and trigrams P^u, P^b, and P^t respectively, a simple fitness function for a key K_c might be:

$$f(K_c) = \Sigma(R^u_i - P^u_i) + \Sigma(R^b_{ij} - P^b_{ij}) + \Sigma(R^t_{ijk} - P^t_{ijk})$$

In other words, the sums of all the differences in unigram, bigram, and trigram frequences, themselves summed together. Of course, this function dictates that higher values indicate that K_c is a key whose decryption results in text *less* like English, not more. This can be easily rectified by modifying the Stochastic Optimization Algorithm to attempt to minimize this value rather than maximize it, or alternatively simply take the negative of $f(K_c)$.

For Frequency Analysis to work, the text being cryptanalyzed must be sufficiently long. To illustrate why, we can examine an artificially short plaintext, $P =$ `MEET ME`. As shown in Figure 1, we should expect the letter `E` to occur about 12.7% of the time which, for this

6-character message, would be $\lceil .127 \times 6 \rceil = \lceil .762 \rceil = 1$ time. Performing Frequency Analysis on such a short message would result in an inaccurately low fitness value; in essence, `MEET ME`, though English, doesn't actually resemble English from a frequency standpoint. Longer messages level out these kinds of outliers, so only relatively long messages can be accurately analyzed in this manner.

4.1 Local Maxima

One potential problem when using frequency analysis to score the fitness of candidate keys is the chance to encounter a local maximum. This can happen when a particular key's decryption of a particular ciphertext resembles English to a certain extent, and every small mutation that would bring the key closer to the correct key would result in analysis-based fitness scores that are actually worse.

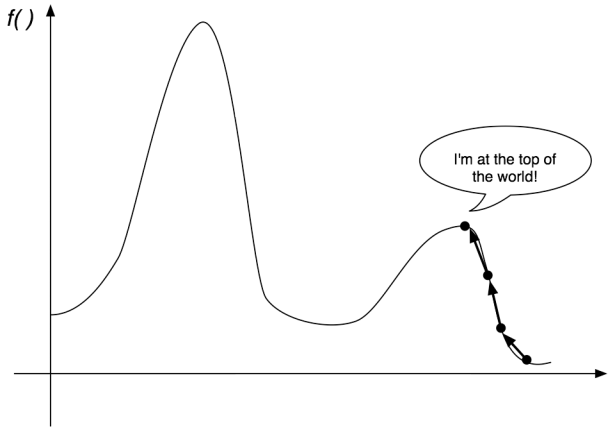

Figure 3: Local Maximum illustration from [Luk11]

This phenomenon is illustrated in Figure 3. If all of the possible fitness values were graphed on a curve, once a fitness function reaches the top of any peak, if there is a valley between the zenith of this peak and the optimal value, many forms of Stochastic Optimization Algorithm have no way of getting to the optimal value. Doing so would first require selecting a series of solutions that are so much worse than the local maxima, the algorithm would effectively decide that it has selected the optimal value and stay there.

The problem of local maxima plagues much research with Stochastic Optimization Algorithms, and using them for cryptanalysis on Monoalphabetic Substitution Ciphers is no exception. A great deal of the work in the field of metaheursitics is an attempt to reduce or remove the problem of local maxima from the search process.

5 Prior Work

Much research has already been done to investigate the efficacy of performing automated cryptanalysis against substitution ciphers using Stochastic Optimization Algorithms. Though by no means exhaustive, below is a sampling of some of this work.

5.1 Genetic Algorithm Implementations

Genetic Algorithms, introduced by [Hol75], modify the concept of the Evolutionary Algorithm by adding a step called crossover. With crossover, the best-fit individuals have their solution "chromosomes" recombined in some way to result in a new chromosome, the offspring of its parents. These offspring are then randomly mutated in the same way as in Evolutionary Strategy and used for a new population. Typically this crossover process picks a chunk from each chromosome and swaps them.

At first glance, crossover with Monoalphabetic Substitution Ciphers seems impossible. With two keys $K_a =$ `EGOQILKVTXJRUHWACFZNBMPYSD` and $K_b =$ `OPJDVCEQRAHIFXMLWBSTGKYUZN`, if the random crossover process were to select swapping the first 3 characters of each key, the result would be $K'_a =$ *`OPJ`*`QILKVTXJRUHWACFZNBMPYSD` and $K'_b =$ *`EGO`*`DVCEQRAHIFXMLWBSTGKYUZN`. K'_a now has two `J`'s and two `P`'s and lacks a `G` and an `E`, while K'_b now has two `G`'s and two `E`'s, lacking a `J` or `P`. Neither of these keys are valid keys for a Monoalphabetic Substitution Cipher, which means neither can be legitimately used for decryption.

[SJNK93] implemented a Genetic Algorithm attack on Monoalphabetic Substitution Ciphers in 1993. The fitness function chosen for this implementation, using the same notation as outlined in Section 4, is:

$$f(K_c) = \left(1 - \frac{\Sigma(R_i^u - P_i^u) + \Sigma(R_{ij}^b - P_{ij}^b)}{4}\right)^8$$

This fitness function is meant to normalize the error value so that larger fitness scores correlate to smaller errors, and this fitness value raised to the 8^{th} power to "amplify small differences"

[SJNK93] also had to solve the crossover problem. In that work, when creating offspring from two parents, the crossover operation scans the two parent keys and, for each position, selects the character which occurs most often in the ciphertext to pass on to the offspring key. As an example, if one parent mapped `A` to `C` and the other parent mapped `A` to F, but `F` occurs more frequently in the ciphertext, the child would map `A` to `F`. As it scans, if the more frequent character already appears in the child key, the less-frequent character is chosen instead. This avoids the problem of duplicate or missing letters.

Results were very positive - written in Turbo Pascal and run on a 25Mhz machine, the implementation generally found a key with a fitness value of 0.9 (high enough to read the text) within 100 generations, in under a minute, using a modest population size of 10.

[OAKAS10] also implemented a Genetic Algorithm attack against Monoalphabetic Substitution Ciphers in 2010. That work uses a fitness function of:

$$f(K_c) = \alpha\Sigma(R_i^u - P_i^u) + \beta\Sigma(R_{ij}^b - P_{ij}^b) + \gamma\Sigma(R_{ijk}^t - P_{ijk}^t)$$

This fitness function allows different weights to be assigned to the different n-gram types. This was based on work by [CD97], who experimented with different weight values and discovered that messages are decrypted most quickly with $\alpha = 0.1$ and $\gamma = 0.8$.

For crossover, a random crossover point is selected along the keys and the letters after that point in each parent as swapped to get intermediate offspring keys. Any characters that appear in the left half of the offspring are not copied over, and the original character from the parent key is retained. This crossover process is less "informed" since it does not utilize the letter frequencies in the ciphertext when making decisions the way [SJNK93] does. In that way, this crossover function is more similar to a natural biological process.

Results were mixed here, with a population of 20 keys finding acceptable keys after 400 generations. It's no surprise that the number of generations here is larger, since the crossover process is less optimized.

5.2 Simulated Annealing Implementation

In 1993, [FSN93] performed automated cryptanalysis against Monoalphabetic Substitution Ciphers using a Simulated Annealing algorithm. Simulated Annealing is another form of metaheuristic first proposed in 1983 by [KGV83], which is a modification of a simple Hill-Climing procedure. In simple Hill-Climbing, a single candidate solution is generated and scored for fitness. Then, the candidate is tweaked (mutated) in some small way and re-scored. If the score is better, it replaces the previous candidate, otherwise it does not, and the process repeats.

With Simulated Annealing, worse solutions actually *are* chosen over better ones, according to a probability. As a result, the algorithm sometimes travels down hills rather than up them, which is an attempt to avoid the local maxima problem outlined in Section 4.1. The proability decreases as the algorithm runs, causing selection to favor climbing over descending as it iterates.

[FSN93] had excellent results, with 5000-character texts being decrypted to the optimal key with an average time of 10 seconds on a modest computer. Shorter texts were less likely to result in full decryption, but when decryption did occur it also took around 10-12 seconds.

5.3 Tabu Search Implementation

[Glo86] proposed Tabu Search as another metaheuristic in 1986. Tabu search keeps a list of recently-considered candidates called the "tabu list" and will not re-consider any of those solutions until the difference between the solution's generation and the current generation is sufficiently large. As a result, if a local maxima is reached, the algorithm has no choice but to go back down a new hill because it cannot keep selecting values at the top of the local maxima.

In 2007, [VDJ07] implemented an attack against Monoalphabetic Substitution Ciphers using the Tabu Search as well as an implementation using a Genetic Algorithm. Though the published work is vague about the exact implementation of a fitness function, the function was identical in both implementations, allowing a direct comparison of Tabu Search's performance to the Genetic Algorithm's.

In tests, The Tabu Search recovered less of the key than the Genetic Algorithm when the ciphertext had fewer than 800 characters, but recovered more of it with longer ciphertexts. Overall, Tabu Search did very well, recovering a readable plaintext in $4/10^{th}$s of the time of a comparable Genetic Algorithm.

5.4 Particle Swarm Implementation

Particle Swarm Optimization, proposed in 1995 by [KE95], is similar to evolutionary algorithms, but it is modeled not after evolution, but instead swarm and flocking behaviors in organisms. Particle Swarm has no selection method at all, and instead maintains a single population whose members are mutated in response to discoveries about the problem space being searched.

A random swarm of particles (candidates) are generated, each at a random location within the problem space, and each with a velocity and a direction for traveling through the problem space. Particles are evaluated for fitness, then all of the particles are modified to have their velocity vector point a bit more toward the most fit particle, after which the particles are mutated by moving along their velocity vectors.

[UY06] implemented a Particle Swarm attack against Monoalphabetic Substitution Ciphers in 2006. The fitness function used was:

$$f(K_c) = \alpha\Sigma(R_i^u - P_i^u) + \beta\Sigma(R_{ij}^b - P_{ij}^b)$$

This fitness function ignores trigram statistics but applies weights to the unigram and bigram frequency differences. Two different pairs for these weights were tried, $(\alpha, \beta) = (1, 0)$ and $(\alpha, \beta) = (0, 1)$, so both functions ignore one of the two types of statistics. Particle swarms had a size of 500 particles, and the algorithm was limited to run for no more than 200 iterations.

Results were good, with a fitness function ignoring unigrams far outperforming one ignoring bigrams. With a ciphertext of 500 characters or more, the Particle Swarm Optimization algorithm was able to recover the optimal or near-optimal key in fewer than 200 iterations. At 300 characters or less, the algorithm struggled, discovering only 21 of the 26 correct key characters in the case of lengths of 300 and only 11 correct characters with lengths under 200.

5.5 Genetic Algorithms on Polyalphabetic Substitution Ciphers

In 2011, [OAKAS11] applied Genetic Algorithms to a Polyalphabetic Substitution Cipher, the Vigenère Cipher. The Vigenère cipher is an attempt to make a more secure cipher than

the Monoalphabetic Substitution Cipher, which takes a key K and, for each letter of the the plaintext P at position i (P_i), shifts that letter by the value of $K_{(i \pmod{|K|})}$.

This attack was quite successful, with the correct key generally being discovered after fewer than 100 generations. Unfortunately, other weaknesses in the Vigenère cipher make it easy to perform cryptanalysis on sufficiently long texts with far less effort.

The problem, discovered by [Kas63], is that it's possible to use repetitions in the ciphertext to figure out the key length. With knowledge of the key length, an attacker can break the ciphertext into strips, one for each letter in the key, representing all and only the characters shifted by that letter. These strips can then be checked for all 26 possible shifts and have unigram-based frequency analysis performed on them, piecing together the key.

For a key length of n, an attacker can simply break the message into n strips and check 26 possible shift keys for each of them, making the effective keyspace that must be searched $26n$. Unless $n = 25!$, this effective search space is actually smaller than that of a Monoalphabetic Substitution Cipher, making it easier to attack rather than more difficult. While a metaheuristic-based attack on such a cipher is valuable from an academic standpoint, it is ultimately overkill for such a breakable cipher.

6 Geneticrypt

Geneticrypt is the name given to the implementation component of this work. It is an Open Source project licensed under the GNU General Public License version 3, and it can be cloned or forked on GitHub at `https://github.com/rodhilton/Geneticrypt` .

Geneticrypt consists of three primary components, **Core**, **CLI**, and **Frontend**. **Core** is the main library, which does frequency analysis, simulation, fitness calculation, and automated cryptanalysis. **CLI**, whose binary is referred to as `cryptools`, is a command-line interface for interacting with Core, allowing key generation, substitution cipher encryption and decryption, and automated cryptanalysis via the library. **Frontend** is a GUI for Core which allows live, automated cryptanalysis. A more detailed `README` file with instructions for building and running these tools is located at the project's GitHub page.

6.1 Decisions and Trade-offs

Geneticrypt does all frequency analysis by first removing all spacing and punctuation, looking only at the streams of letters in both the reference text as well as the partially decrypted ciphertext. This is to emulate the most difficult scenario for cryptanalysis, where there are no punctuation-based hints to help. Thus, while much research relies on, for example, parsing "won't" into four bigrams of [wo], [on], [n'], and ['t] and three trigrams of [won], [on'], and [n't], Geneticrypt's analyzer only creates entries for three bigrams [wo], [on], [nt] and two trigrams [won] and [ont]. Spaces are handled similarly, where a phrase like "meet me" would usually get bigrams including [t_] and [_m] and trigrams including [t_m], Geneticrypt would store bigrams including [tm].

The fitness function for a candidate key K_c, using the same notation as in Section 4, is:

$$f(K_c) = \alpha(1 - \Sigma(R_i^u - P_i^u)) + \beta(1 - \Sigma(R_{ij}^b - P_{ij}^b)) + \gamma(1 - \Sigma(R_{ijk}^t - P_{ijk}^t))$$

This is a weighted function that uses similarities rather than differences, and scales the values of the unigram, bigram, and trigram frequencies. The weights used are $\alpha = 0.2$, $\beta = 0.3$, and $\gamma = 0.5$. The effect of these weights is that, when the algorithm is first starting out and matching letters are infrequent, the similarities for bigrams and trigrams will be near 0, so the weights have no effect and the unigram matches define the progress of the algorithm. In later stages of execution, unigram similarity has gotten large enough that the bigrams and trigrams start matching more often, and the larger weight factors result in bigram and eventually trigram similarity taking over the function, dwarfing the importance of unigram similarity. The reference text frequencies are calculated by scanning English books published under Project Gutenberg[1].

The implementation forgoes any form of crossover mutation, implementing a straightforward Evolution Strategy like the one explained in Figure 2 on Page 4. The first generation of λ random individuals is formed by creating random permutations of the letters of the alphabet. Subsequent generations are formed by mutating the μ best fit individuals according to f from that current generation λ/μ times, where each mutation consists of picking two random characters of the individual and swapping them. This process continues until the user intercedes; since the cryptanalysis library doesn't receive the original plaintext or the correct key as input, it has no way of knowing when it has found the optimal key, so the user must declare he or she is able to read the decipherment well enough for the algorithm to stop.

Appropriate values for λ and μ were arrived at via experimentation. The population size λ was tested at 10, 20, 50, 100, and 200. The selection size μ was tested at 1, 2, 5, and 10, all factors of the population sizes. For each combinatorial pair (λ, μ), 5 trials were run with random, 1000-character ciphertexts from the sample text library, then evaluated according to how many characters of the key the pair recovered after 100 generations. The results of these trials are shown in Figure 4, which includes the average number of characters of the original key recovered as well as the maximum number during the trials.

The Evolutionary Strategy algorithm appears to be only hindered by different values for selection size, $\mu = 1$ performed best for nearly every population size. Additionally, little was gained by moving the population size from 100 to 200, which doubles the length of time it takes to iterate over 100 generations, but only marginally helps with key recovery. These results indicate that a population size of 100 and a selection size of 1 are ideal for this algorithm, at least when the ciphertext is relatively lengthy.

6.2 Performance

Geneticrypt was tested for how well it performed with ciphertexts of sizes 100, 200, 300, 400, 500, 750, 100, 1250, 1500, and 2000. Ten trials were run for each length, with random

[1] `http://www.gutenberg.org/`

λ	μ	Avg.	Max.
10	1	10.6	14
10	2	11.6	16
10	5	5.4	11
10	10	1.6	3
20	1	19.4	21
20	2	17.6	19
20	5	10.8	15
20	10	7.2	10
50	1	23.2	26
50	2	22.2	24
50	5	12.6	20
50	10	16.0	17
100	1	24.8	26
100	2	19.8	26
100	5	23.6	26
100	10	21.0	24
200	1	20.2	26
200	2	24.4	26
200	5	25.4	26
200	10	19.6	26

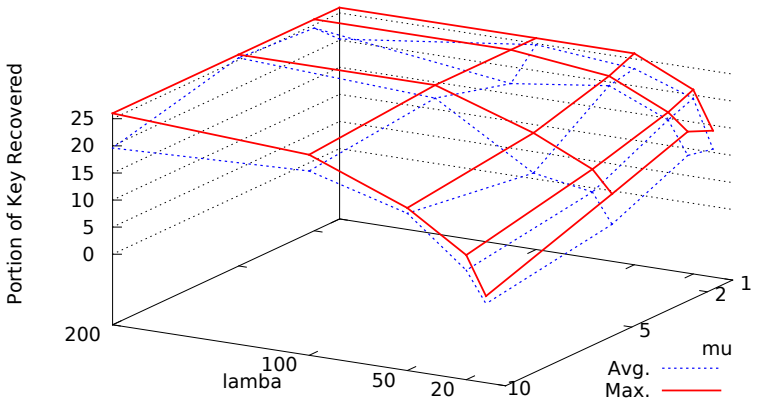

Figure 4: Effect of population size and selection size on key recovery

ciphertexts from the sample text library, each encrypted with a random key, and allowed to run for 100 generations. The results of this benchmark are in Figure 5.

Length	Median	Avg.	Min	Max	Std. Dev
100	5.5	6.2	1.0	16.0	4.63
200	11.5	10.3	2.0	17.0	5.85
300	17.5	15.9	4.0	22.0	6.20
400	19.0	17.4	5.0	26.0	6.99
500	20.0	17.6	3.0	24.0	7.26
750	24.0	19.7	3.0	26.0	8.93
1000	24.0	22.1	4.0	26.0	6.45
1250	26.0	24.1	14.0	26.0	3.84
1500	26.0	25.5	23.0	26.0	1.08
2000	26.0	25.2	22.0	26.0	1.39

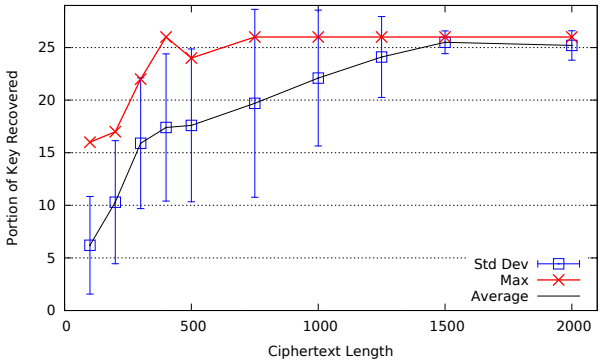

Figure 5: Effect of population size and selection size on key recovery

Based on the medians, once the ciphertext is 1,250 characters long, the optimal key is found at least half the time. Even at 750, 24 characters are consistently found, which is the highest possible number short of the optimal key (since it means two characters must be swapped). Unsurprisingly, the standard deviation for the amount of the key recovered gets smaller and smaller with longer ciphertexts, indicating that Geneticrypt is more consistently able to avoid local maxima in a short period of time with longer texts.

Geneticrypt was also run on 1000-character ciphertexts with no generation restriction, to see how many generations are required, on average, to recover enough of the key to read the decryption. The results of these trials are in Figure 6. Each of these runs was independent,

meaning that a trial would start with a generation number from the set {10, 20, 30, 40, 50, 75, 100, 125, 150, 200} and evaluate key recovery for a generation limit of that number, so the same set of trials running for 30 generations are not the same as the ones running for 20 generations.

Gen.	Median	Avg.	Max	Std. Dev
10	6.0	6.8	11.0	2.77
20	13.0	10.8	15.0	4.91
30	26.0	24.6	26.0	2.19
40	24.0	21.4	26.0	7.60
50	26.0	24.8	26.0	1.78
75	24.0	24.6	26.0	1.34
100	24.0	19.4	24.0	9.20
125	26.0	25.2	26.0	1.09
150	23.0	20.6	26.0	8.35
200	26.0	25.2	26.0	1.09

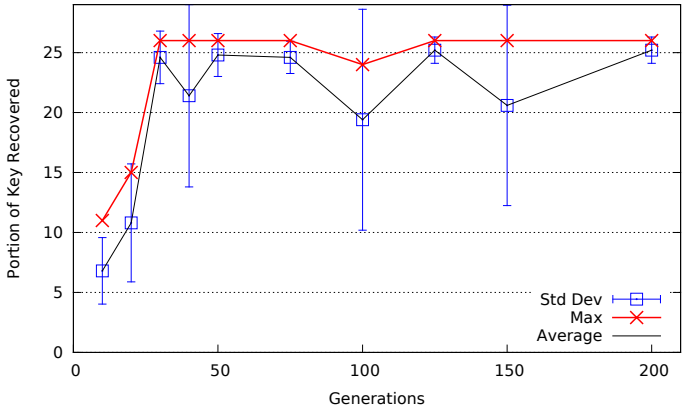

Figure 6: Effect of number of generations on key recovery

These results indicate that there is a point of diminishing returns after 30 generations. At 30 generations, at least half of the decryption attempts will recover 24 characters of the key, which is enough for a human to read the partially decrypted ciphertext. The standard deviation values indicate that local maxima skewed the numbers for some trials, so this algorithm suffers substantially from that possibility.

Geneticrypt utilizes the `Future` and `Executors` classes from the `java.util.concurrent` package to evenly distribute the $O(n)$ fitness calculation work across x different threads, where x is the number of processors available to the operating system. On a 4-core Intel i7 2.3GHz machine[2], iterating over the required 30 generations typically takes under 1 minute.

6.3 Future Directions

Geneticrypt was built to be extended. The class representing a candidate, `Candidate`, is a simple interface that allows any type of problem-solving candidates to be created and dropped into a simulator with ease. The Simulator interface, `Simulator` is similarly designed to be extended so different Stochastic Optimization Algorithms can be tested. As such, there are two primary ways that Geneticrypt lends itself to future work: performing cryptanalysis on Monoalphabetic Substitution Ciphers using different Stochastic Optimization Algorithms, and attempting cryptanalysis on other ciphers which exhibit the *Utility of Partial Solutions* property.

It would be worthwhile to implement the work of [SJNK93], [OAKAS10], [VDJ07], [FSN93], [LP11], and [UY06], who performed cryptanalysis on Monoalphabetic Substitu-

[2]The i7 utilizes Hyper-Threading Technology, so for each core, the operating system addresses two virtual cores, making the effective number of cores 8.

tion Ciphers using different, more efficient metaheuristics discussed in Section 5. It would also be valuable to implement some of the more recent proposals in Stochastic Optimization Algorithms, such as [YD09]'s Cuckoo Search, [SH11]'s Galaxy-based Search, and [TY11]'s Spiral Optimization Algorithm.

Another area for additional research is using Geneticrypt's optimization algorithms to perform automated cryptanalysis against Polyalphabetic Substitution Ciphers such as the Autokey, Hill, or Alberti cipher, or perhaps even the Enigma machine. Implementing attacks against these ciphers using a tool like Geneticrypt can help provide evidence that a particular cipher exhibits the *Utility of Partial Solutions* property.

7 Conclusions

Monoalphabetic Substitution Ciphers are different from many other ciphers, in that partial keys are useful, which makes them vulnerable to an optimized attack via Stochastic Optimization Algorithms. The simplest and least efficient of these algorithms, the simple Evolutionary Strategy algorithm, can perform automated cryptanalysis on sufficiently long ciphertexts in a few minutes on a modest computer, quickly finding a near-optimal key despite a keyspace in the septillions.

The Monoalphabetic Substitution Cipher, because it has a quantifiable solution size and an optimal solution that cannot be known ahead of time, makes for a great benchmark of new metaheuristics. Furthermore, if Evolutionary Strategy can find the key quickly, it stands to reason that more efficient Stochastic Optimization Algorithms would fare better. Indeed, a great deal of research has shown that this is the case, with more modern metaheuristics quickly finding cipher keys as well. This has also been extended to Polyalphabetic Substitution Ciphers, and it stands to reason it would work for any cipher that exhibits the *Utility of Partial Solutions* property.

Though classical encryption schemes like these substitution ciphers are no longer widely used today, it's noteworthy that they are so weak that modern computers can perform automated cryptanalysis attacks against them incredibly easily despite enormous keyspaces, underscoring the importance of utilizing modern ciphers for encrypting long messages.

References

[Bek82] Henry Beker, *Cipher systems*, Chapman & Hall, 1982.

[Bis02] David Bishop, *Introduction to cryptography with java applets*, Jones and Bartlett Publishers, Inc., 2002.

[CD97] Andrew Clark and Ed Dawson, *A parallel genetic algorithm for cryptanalysis of the polyalphabetic substitution cipher*, Cryptologia **21** (1997), no. 2, 129–138.

[FSN93] W. S. Forsyth and R. Safavi-Naini, *Automated cryptanalysis of substitution ciphers*, Cryptologia **17** (1993), no. 4, 407–418.

[Glo86] Fred Glover, *Future paths for integer programming and links to artificial intelligence*, Computers & operations research **13** (1986), no. 5, 533.

[Hol75] John H. Holland, *Adaptation in natural and artificial systems: An introductory analysis with applications to biology, control and artificial intelligence*, The University of Michigan Press, 1975.

[Kas63] F. W. Kasiski, *Die geheimschriften und die dechiffrir-kunst*, E. S. Mittler und Sohn, 1863.

[KE95] J. Kennedy and R. Eberhart, *Particle swarm optimization*, Neural Networks, 1995. Proceedings., IEEE International Conference on, vol. 4, nov/dec 1995, pp. 1942 –1948 vol.4.

[KGV83] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, *Optimization by simulated annealing*, Science **220** (1983), no. 4598, pp. 671–680 (English).

[Lev66] V I Levenshtein, *Binary codes capable of correcting deletions, insertions, and reversals*, Soviet Physics Doklady **10** (1966), no. 8, 707–710.

[Lew00] Robert Edward Lewand, *Cryptological mathematics (mathematical association of america textbooks)*, The Mathematical Association of America, 2000.

[LP11] J. Luthra and S.K. Pal, *A hybrid firefly algorithm using genetic operators for the cryptanalysis of a monoalphabetic substitution cipher*, Information and Communication Technologies (WICT), 2011 World Congress on, 11 2011, pp. 202–206.

[Luk11] Sean Luke, *Essentials of metaheuristics*, lulu.com, 2011.

[OAKAS10] S.S. Omran, A.S. Al-Khalid, and D.M. Al-Saady, *Using genetic algorithm to break a mono - alphabetic substitution cipher*, Open Systems (ICOS), 2010 IEEE Conference on, dec. 2010, pp. 63 –67.

[OAKAS11] ———, *A cryptanalytic attack on vigenere cipher using genetic algorithm*, Open Systems (ICOS), 2011 IEEE Conference on, sept. 2011, pp. 59 –64.

[Rec65] I. Rechenberg, *Cybernetic solution path of an experimental problem*, Royal Aircraft Establishment Translation No. 1122, B. F. Toms, Trans., Ministry of Aviation, Royal Aircraft Establishment, Farnborough Hants, August 1965.

[SBJ79] Robert Solso, Paul Barbuto, and Connie Juel, *Bigram and trigram frequencies and versatilities in the english language*, Behavior Research Methods **11** (1979), 475–484, 10.3758/BF03201360.

[SH11] Hamed Shah-Hosseini, *Principal components analysis by the galaxy-based search algorithm: a novel metaheuristic for continuous optimisation*, International Journal of Computational Science and Engineering **6** (2011), no. 1, 132–140.

[Sin00] Simon Singh, *The code book: The science of secrecy from ancient egypt to quantum cryptography*, Anchor, 2000.

[SJNK93] Richard Spillman, Mark Janssen, Bob Nelson, and Martin Kepner, *Use of a genetic algorithm in the cryptanalysis of simple substitution ciphers*, Cryptologia **17** (1993), no. 1, 31–44.

[TW05] Wade Trappe and Lawrence C. Washington, *Introduction to cryptography with coding theory (2nd edition)*, Prentice Hall, 2005.

[TY11] Kenichi Tamura and Keiichiro Yasuda, *Primary study of spiral dynamics inspired optimization*, IEEJ Transactions on Electrical and Electronic Engineering **6** (2011), no. S1, S98–S100.

[UY06] M.F. Uddin and A.M. Youssef, *Cryptanalysis of simple substitution ciphers using particle swarm optimization*, Evolutionary Computation, 2006. CEC 2006. IEEE Congress on, 0-0 2006, pp. 677 –680.

[VDJ07] A. K. Verma, Mayank Dave, and R. C. Joshi, *Genetic algorithm and tabu search attack on the mono-alphabetic substitution cipher i adhoc networks*, Journal of Computer Science **3** (2007), 134–137.

[YD09] Xin-She Yang and S. Deb, *Cuckoo search via levy flights*, Nature Biologically Inspired Computing, 2009. NaBIC 2009. World Congress on, dec. 2009, pp. 210 –214.