Norwegian University of
Science and Technology

Master's thesis

2021

Karl August Høivik

Master's thesis

NTNU
Norwegian University of
Science and Technology
Faculty of Information Technology and Electrical
Engineering
Department of Computer Science

Karl August Høivik

On the Usage of Metaheuristic Optimization in Cryptanalysis

June 2021

Norwegian University of
Science and Technology

On the Usage of Metaheuristic Optimization in Cryptanalysis

Karl August Høivik

Computer Science (MTDT)
Submission date: June 2021
Supervisor: Magnus Lie Hetland

Norwegian University of Science and Technology
Department of Computer Science

On the Usage of Metaheuristic Optimization in Cryptanalysis

Karl August Høivik

Abstract

This report explores the possibility of solving cryptographic decryption problems with optimization algorithms by converting ciphertexts of a known cipher to solvable optimization problems, and solving them with a C++ framework.

Figures

Tables

Acknowledgements

I would like to use this chapter to thank my advisor, Magnus Lie Hetland, for helping me stay motivated through this project and seeing the value of my work when I had lost sight of it. I would also like to thank my family for reminding me when my work seemed overwhelming, that to move a mountain one must begin by carrying away the smaller stones. Lastly, I would like to thank my partner for keeping me on track when I would rather put off work until the next day.

Chapter 1

Introduction

Cryptography is the practice of hiding secret messages by performing operations on the messages according to a cryptographic algorithm and an encryption key. The encrypted/enciphered message can then be sent to a recipient who then uses the same algorithm and either the same key or an inverse/decryption key to reveal the hidden message. Cryptanalysis is the practice of attempting to reveal this hidden message without knowing the key beforehand. Essentially this leaves the cryptanalysts with the same problem as the intended recipient of the message, except one variable; the encryption/decryption key. The goal of this project is to determine whether this problem can be solved by leaving the unknown variable to a metaheuristic optimization algorithm.

Using metaheuristic optimization to solve cryptographic problems is by no means a new concept. A lot of research has already been done in this exact field since the early 90s, in which the scope was restricted to small number of cryptographic algorithms and optimization algorithms. The motivation for this project is to expand the scope by attacking nine different cryptographic algorithms with six different optimization algorithms, including Walton's Improved Cuckoo Search[1].

Our contributions to the field includes:

- Summarizing the last thirty years of research in the field of using metaheuristic optimization in cryptanalysis.
- Developing a modular and extendable framework for solving cryptographic problem instances as optimization problems with meta-

  heuristic algorithms.
- An implementation Walton's Improved Cuckoo Search[1] for the pagmo library.
- Achieved slightly higher key recovery for DES than earlier research.
- Showing that the Improved Cuckoo Search algorithm can be useful for ciphertext-only attacks on DES, AES and SPECK.

This report is structured as follows. The *Background* chapter explains the ciphers we have chosen to work with, as well as summarizes prior related research in the field of using metaheuristic optimization to solve cryptographic problems. The *Ciphers* and *Optimization algorithms/schemes* is reproduced from Høivik's master project report from 2020[2], with some changes to formatting. The *Method* chapter describes the framework that was developed to solve the cryptographic problems using metaheuristic optimization, in addition to outlining the experiment parameters and process. The *Results* chapter presents the data obtained from running the experiments, and how each optimization algorithm performs compared to each other. Lastly, we will discuss the results, limitations and future work in the *Discussion and Future Work* chapter.

Chapter 2

Background

The goal of cryptanalyzing a cryptosystem is to find weaknesses in the system that might lead to revealing the plaintext, partially or fully, using its corresponding ciphertext. In this project, we will only use ciphertext-only attacks, in which the optimization algorithm only has access to the cipher and a ciphertext, and is tasked with recovering the key and plaintext. Simply trying all possible keys for a cipher will eventually yield the correct key and decryption, but for many ciphers the number of possible keys, is too large to mount a computationally feasible brute-force attack such as this. Instead, we want to limit the search-space as much as possible or at least guide the search efficiently.

For the classical cryptosystems or ciphers, such as Caesar, Vigenère and monoalphabetic substitution ciphers, there exists several well-known cryptanalysis techniques which can be done by hand. One example of this is for the Caesar cipher, where one finds the character that occurs most times in the ciphertext and assumes that this character will decrypt to ’E’, the most frequent character in the English language. Then do the same for the second-to-most frequent character, and so on. This will not necessarily lead to the correct solution, but it will give the most likely key when only considering character frequencies.

Modern ciphers such as DES and AES implement several diffusion techniques, making the relative character frequency in the resulting ciphertext as flat as possible. For this reason, there is no simple way to retrieve the key from modern ciphers through ciphertext-only analytic attacks. For DES, AES and SPECK there are known-plaintext, chosen-plaintext and chosen-ciphertext attacks that can retrieve the key, but they require access to the corresponding plaintexts or an encryption oracle, as well as large amounts of resources.

In this paper, we will not be using the usual techniques for classic ciphers or the complicated resource intensive attacks for modern ciphers. Instead, we will perform ciphertext-only attacks on the ciphers by treating the decryption of the ciphertext as a black box that takes the potential key as an input and outputs how well the key decrypted the ciphertext.

2.1 Ciphers

The classical ciphers are simple cipher that were created long before the modern computers, and as such were meant to be computed and solved by hand. Due to their simplicity, they are usually very simple to break with modern technology, and have therefore fallen into disuse in modern security settings. The ciphers are often divided into *transposition ciphers* and *substitution ciphers*:

Transposition ciphers maintain the characters in the plaintext, and simply reposition them in the ciphertext according to a well-defined scheme. Common transposition ciphers include: The Columnar Cipher, and the Rail Fence Cipher.

Substitution ciphers maintain the position of the characters in the plaintext, but systematically replace the characters (or groups of characters) throughout the plaintext to produce the ciphertext. Common substitution ciphers include: The Caesar/Shift Cipher, the general monoalphabetic substitution cipher, Vigenère cipher and the general polyalphabetic substitution cipher. Classical ciphers are no longer used for encryption in modern security settings, however they are still very useful for understanding modern cryptosystems and their development.

Columnar Cipher The Columnar Cipher is a transposition cipher that uses a grid where the columns are rearranged for encryption/decryption. As an example, we will encrypt the string "Columnar Cipher" with the key "KEYS" (table 2.1a). The plaintext is written into the grid row-wise, then the columns are reordered according to the key. The ciphertext is read column-wise, resulting in the ciphertext "onCeCm hurp lair". For decryption, the recipient knows the key and therefore also knows the number of columns, and writes the ciphertext into the grid column-wise, reorders the columns according to the key and reads the plaintext row-wise.

K	E	Y	S
1	0	3	2
C	o	l	u
m	n	a	r
	C	i	p
h	e	r	

(a) Encryption: Writing the plaintext into a grid row-wise

E	K	S	Y
0	1	2	3
o	C	u	l
n	m	r	a
C		p	i
e	h		r

(b) Encryption: Reordering the columns according to the key

E	K	S	Y
0	1	2	3
o	C	u	l
n	m	r	a
C		p	i
e	h		r

(c) Decryption: Writing the ciphertext into the grid column-wise

K	E	Y	S
1	0	3	2
C	o	l	u
m	n	a	r
	C	i	p
h	e	r	

(d) Decryption: Reordering the columns according to the key

Caesar/Shift cipher In the Caesar cipher (also known as the shift cipher), each character in the plaintext is shifted forward by k places, where k is the key, and wrapping around the alphabet if necessary. Decryption works in reverse by shifting backwards the same number of places (k).

Plaintext	CAESARCIPHERXYZ
Ciphertext	FDHVDUFLSKHUABC

Figure 2.2: Caesar cipher with $k = 3$

General substitution ciphers A monoalphabetic substitution cipher is a classical cryptosystem where every occurrence of a character is substituted by another, dictated by the key. A common way of generating a key is to choose a keyword, such as "CIPHER", as the start of the key and fill the remainder with the rest of the alphabet ("CIPHERABDFGJKLMNOQS-TUVWXYZ"). However, depending on the keyword, this method will create a key that often results "non-substitutions", where characters at the end of the alphabet are substituted with themselves. In this project we will instead look at a generalized key-generation, where the key is generated randomly, instead of a keyword. The key has the same size as the alphabet, and in the example below every "A" will be replaced with an "N", all "B"s will stay the same, every "C" replaced with "A", etc.

Alphabet	ABCDEFGHIJKLMNOPQRSTUVWXYZ
Enc. key	NBAJYFOWLZMPXIKUVCDEGRQSTH

Figure 2.3: Example encryption key for a monoalphabetic substitution cipher

Plaintext	M	O	N	O	A	L	P	H	A	B	E	T	I	C
Ciphertext	X	K	I	K	N	P	U	W	N	B	Y	E	L	A

Figure 2.4: Encryption with the key in 2.3

Several monoalphabetic substitution ciphers can be combined into a polyalphabetic substitution cipher, usually by rotating which key is used for each character. This results in a much longer key length and introduces some diffusion and confusion. For polyalphabetic substitution ciphers, the strategy becomes slightly more convoluted. First, we need to determine how many alphabets the cipher uses. A common way of doing this is to split the ciphertext into n substrings where the first character is in the first substring, the second character in the second substring, and so on. Now we calculate the chi-squared (section 2.3) value for each of these substrings and average them. We perform this analysis for a few values of n and choose the value of n that gives the lowest average. The number of alphabets used in the cipher is most likely a multiple of n. Now it is simply a matter of solving m monoalphabetic substitution ciphers, where m is a multiple of n.

Vigenère The Vigenère cipher is a polyalphabetic substitution cipher that uses a word, usually of short length, as the key. The key simply dictates how much each character is shifted in the text, "A" means no shift, "B" shifted by one, etc. For example, if the key is "ABC", the first character would not be shifted, the second character shifted by one character, the third by two, the fourth not shifted, etc. The key is repeated to match the length of the plaintext. The encryption/decryption can also be expressed mathematically by enumerating each character (A=0, B=1,...) and using the following formulas:
Encryption: $c_i = p_i + k_i \pmod{26}$
Decryption: $p_i = c_i - k_i \pmod{26}$

Plaintext	V	I	G	E	N	E	R	E
Key	K	E	Y	K	E	Y	K	E
Ciphertext	F	M	E	O	R	C	B	I

Figure 2.5: Vigenère encryption

Playfair The Playfair cipher substitutes digrams according to a set of rules. First, a 5x5 cipher table is created from a chosen keyword, such as "PLAYFAIREXAMPLE", by inserting each character into the table if that character does not already exist, starting with the keyword, then the rest of the alphabet.
Then, the plaintext is split into digrams, or two-character blocks. To avoid

P	L	A	Y	F
I	R	E	X	M
B	C	D	G	H
K	N	O	Q	S
T	U	V	W	Z

Figure 2.6: Playfair key matrix using the key "PLAYFAIREXAMPLE"

digrams with the same character, an "X" is inserted. The digrams are then substituted according to a set of rules based on their position in the grid:

1. If the characters form a rectangle, replace each of the characters with the character in the same row, but opposite corner of the rectangle
2. If the characters are in the same column, replace them with the character below, wrapping around the same column if necessary
3. If the characters are in the same row, replace them with the character to the right, wrapping around the same row if necessary

DES The Data Encryption Standard is a modern block cipher which works on 64-bit blocks of plaintext to produce a 64-bit block of ciphertext. Its key is a 64-bit string, however one byte is used for parity, hence the effective key length is 56 bits. The algorithm performs two permutation functions, one before and one after applying a round function 16 times.

1. Apply the initial permutation function IP on the block
2. Split the block into two 32-bit halves, LE_0 and RE_0
3. For i to $r = 16$ rounds:
   a. $LE_i = F(RE_{i-1}, K_{k-i})$
   b. $RE_i = LE_{i-1} \oplus LE_i$
4. Apply the final permutation function FP on the block and return it

AES The Advanced Encryption Standard, also known as Rijndael, is a modern block cipher created by Vincent Rijmen and Joan Daemen and established by the National Institute of Standards and Technology (NIST) in 2001. The cipher is based on a design known as a substitution-permutation network (SPN), as opposed to the Feistel network used by DES. AES operates on a 4x4 column-major order array of bytes, called the *state* (shown in figure 2.7). Most calculations are done in a particular finite field.

The algorithm

- `KeyExpansion` - the cipher key is used to derive a round key for each round of the algorithm
- First round: `AddRoundKey` - each byte in the state is XORed with a byte of the round key

- For nine, eleven or thirteen rounds:
  - `SubBytes` - non-linearly substitute each byte with another according to a lookup table, called the Rijndael S-box.
  - `ShiftRows` - the last three rows of the state are shifted cyclically a certain number of steps.
  - `MixColumns` - multiplies each column of the state with a fixed polynomial to combine the four bytes in each column
  - `AddRoundKey`
- Final round (making 10, 12, or 14 rounds in total):
  - `SubBytes`
  - `ShiftRows`
  - `AddRoundKey`

$$\begin{bmatrix} b_0 & b_4 & b_8 & b_{12} \\ b_1 & b_5 & b_9 & b_{13} \\ b_2 & b_6 & b_{10} & b_{14} \\ b_3 & b_7 & b_{11} & b_{15} \end{bmatrix}$$

Figure 2.7: AES state representation

RSA Key generation:

1. Choose two distinct prime numbers p and q
2. Compute $n = pq$
3. Compute $\lambda(n) = lcm(p-1, q-1)$
4. Choose an integer e such that $1 < e < \lambda(n)$ and $gcd(e, \lambda(n)) = 1$
5. Determine d such that $d \equiv e^{-1} \pmod{\lambda(n)}$

The private key is d and the public key is (e, n). Miller's Theorem proves that determining d from e is as hard as factorizing n into its two prime factors.
Encryption: $m^e \equiv c \pmod{n}$
Decryption: $c^d \equiv (m^e)^d \equiv m \pmod{n}$

Speck The Simon and Speck families of block ciphers were publically released by the National Security Agency (NSA) in June 2013[3]. Simon is tuned for optimal performance in hardware, and Speck for optimal performance in software.

The algorithm

```c
#include <stdint.h>

#define ROR(x, r) ((x >> r) | (x << (64 - r)))
#define ROL(x, r) ((x << r) | (x >> (64 - r)))
#define R(x, y, k) (x=ROR(x,8), x+=y, x^=k, y=ROL(y,3), y^=x)
#define ROUNDS 32

void encrypt(uint64_t ct[2],
             uint64_t const pt[2],
             uint64_t const K[2])
{
   uint64_t y = pt[0], x = pt[1], b = K[0], a = K[1];

   R(x, y, b);
   for (int i = 0; i < ROUNDS - 1; i++) {
      R(a, b, i);
      R(x, y, b);
   }

   ct[0] = y;
   ct[1] = x;
}
```

2.2 Optimization algorithms/schemes

Genetic/evolutionary algorithms Genetic/evolutionary algorithms begin with a population of candidate solutions. Each candidate solution has a set of properties which can be mutated and altered. Each generation, a portion of the existing population is selected to breed a new generation. Solutions are typically selected through a fitness-based process, where the fitter solutions are more likely to be selected.

Simulated annealing Simulated annealing is used to probabilistically approximate the global optimum of a function. The algorithm uses a random search and accepts changes that improve the objective function, but also keeps some changes that are not ideal, with a probability p, in an attempt to avoid converging on local optima.

Particle Swarm Optimization Particle Swarm Optimization iteratively tries to improve a candidate solution by having a population of candidate solutions, called particles, which move around in the search-space according to a simple formula. Each particle's movement will be based on the current best global solution, and the particle's local best solution.

Ant algorithm Ant Colony optimization is a class of optimization algorithms based on the actions of an ant colony. Artificial "ants" (simulation agents) locate optimal solutions by moving through a parameter space representing all possible solutions. Real ants lay down pheromones directing each other to resources while exploring their environment. The simulated "ants" similarly record their positions and the quality of their solutions, so that in later simulation iterations, more ants locate better solutions.[4]

Artificial Bee Colony algorithm In the ABC algorithm[5], a colony of bees contains three groups of bees: employed bees, onlookers and scouts. A bee waiting to be assigned a food source is called an onlooker, a bee going to the food source previously visited by itself is named an employed bee, a bee carrying out a random search is called a scout. In the algorithm, the first half of the colony consists of employed artificial bees, and the second half consists of onlookers. The main steps of the algorithm are as follows:

1. Initialize
2. REPEAT UNTIL (requirements are met)
    a. Place the employed bees on the food sources in memory
    b. Place the onlookers bees on the food sources in memory
    c. Send the scouts to the search area for discovering new food sources

In other words, each cycle of the search consists of three steps: sending employed bees onto food sources (possible solutions) and then measuring their nectar amounts (quality/fitness); selecting which food sources to send onlookers to after determining nectar amounts; determining which bees become scouts and sending them to possible food sources. In the initialization stage, we randomly select a set of food source positions and determine their nectar amounts.

Cuckoo search The cuckoo search algorithm is based on the cuckoo bird's parasitic behavior, where a cuckoo bird will lay its eggs in a host bird's nests. The algorithm uses the following representations: each egg in a nest represents a solution, and a cuckoo egg represents a new solution. Each cuckoo lays one egg at a time and dumps its egg in a randomly chosen nest. The nests with the highest quality of eggs (the best fitness) will carry over to the next generation. The number of host nests is fixed, and an egg laid by a cuckoo is discovered and discarded by the host bird with a probability p_a. For this project I have chosen to implement Walton et al.'s Modified Cuckoo Search[1]:

```
1. Generate an initial population of $n$ host nests;
2. While($t < MaxGeneration$)
3.    a. Replace a fraction $p_a$ of the worse solutions by performing
         Lévy flights
      b. For $i$ in all top nests
      c.   i. Pick another nest ($j$) at random
          ii. If $i == j$, replace $i$ by performing Lévy flight
         iii. Else, move $\sqrt(|i - j|)/1.618$ from worst to best
4. end while
```

2.3 Fitness function

In cryptanalysis of classical ciphers, one might use chi-squared statistics as a measure of how close to the English language a decryption is, by comparing letter frequency in the decryption with letter frequency in the English language. The chi-squared value is calculated as follows:

$$\chi^2(C,E) = \sum_{i=A}^{i=Z} \frac{(C_i - E_i)^2}{E_i} \tag{2.1}$$

Where C_i is the number of occurrences of the letter i in the decryption and E_i is the number of expected occurrences in the decryption.

A lower value means a letter frequency close to the letter frequency in English, which in turn should be the most likely decryption of the ciphertext.
We will perform these calculations for digram and trigram frequencies as well, and weight them.

$$\chi^2(C,E) = 0.1 \cdot \sum_{i=A}^{i=Z} \frac{(C_i - E_i)^2}{E_i} + 0.1 \cdot \sum_{i=AA}^{i=ZZ} \frac{(C_i - E_i)^2}{E_i} + 0.8 \cdot \sum_{i=AAA}^{i=ZZZ} \frac{(C_i - E_i)^2}{E_i} \tag{2.2}$$

2.4 Index of Coincidence

To determine the period of the Vigenère cipher, i.e. the length of the key, we exploit that the cipher repeats the key throughout the ciphertext. We calculate the Index of Coincidence (I_c) using the equation below:

$$I_c = \frac{\sum_{i=1}^{j} f_i(f_i - 1)}{n(n-1)} \tag{2.3}$$

, where n is the length of the ciphertext and f_i is the frequency count of the ith letter. We can then approximate the key length by using the following equation[6]:

$$key\_length = \frac{0.027n}{I_c(n-1) - 0.038n + 0.065} \tag{2.4}$$

2.5 Relevant prior works

2.5.1 Classical Ciphers

Columnar Cipher

In 1993, R.A.J. Matthews used an order-based genetic algorithm to attack the columnar cipher. He used a simple position based crossover, and then applied one of two mutations; swapping two indexes or shifting the chromosome forward a random number of places. The fitness function was based on the frequency of a small list of digrams and trigrams. This attack was later re-implemented by Bethany Delman in 2004[7], with more parameter combinations.

Monoalphabetic Substitution Cipher

In 1993, Spillman et al.[8] used a simple genetic algorithm with a two-point swap mutation to cryptanalyze simple substitution ciphers, achieving full key recovery after only 1000 function evaluations. In 1998, Andrew Clark[9] used simulated annealing and was able to consistently correctly determine all but one key element. His algorithm chose a new candidate solution by swapping two random indexes in the current solution, and evaluated the new solution using a fitness function based entirely on trigram frequencies. However, he found that in terms of the number of keys recovered vs. total keys considered, the genetic algorithm vastly outperformed simulated annealing. In 2003, Gründlingh and Van Vuuren[10] introduced a custom fitness function based on ciphertext length and monogram frequencies, which they used with a genetic algorithm. In 2006, Uddin and Youssef[11] suggested using Particle Swarm optimization in which particles were more likely to change to a new permutation based on velocity, instead of adding the velocity to the particle on each dimension. Mekhaznia and Menai[12] used ant colony optimization for several classical ciphers, including monoalphabetic substitution, however they only achieved an average of 24% key recovery across ciphertext lengths. In 2016, Grari et al.[13] demonstrated that one can recover up to 100% of the key elements when using ant colony optimization with enough ciphertext and a fitness function based on only mono- and digram frequencies. Sabonchi and Akay[14] used the artificial bee colony algorithm, recovering up to 19 out of 26 key characters. In 2015, Jain et al.[15] showed that cuckoo search performs reasonably well compared to genetic algorithms and tabu search in terms of how many key elements were correctly identified. Their algorithms used only digram frequencies for the fitness function. The genetic algorithm that cuckoo search was compared with used the same crossover operator as the one proposed by Clark, and a mutation operator that switched the values at two indexes. In addition, if no improvement had occurred for a number of generations, a second mutation operator would be used; interchanging the values at three random indexes.

Vigenère

As the Vigenère cipher is nothing more than a number of Caesar ciphers used in succession, one natural way of solving them is to simply solve these ciphers in parallel. This approach was used for general polyalphabetic substitution ciphers, which the Vigenère cipher is a simplification of. Their algorithm used a number of parallel genetic algorithms, each working on a slice of the ciphertext to find the best fitting key for their slice. To be able to calculate fitness for digrams and trigrams, each process would communicate their current best key to the other processes after a number of iterations. The mutation worked as follows: If the child had a fitness greater than the median, each character was swapped with the one to its right; if the child's fitness was less than the median, each character was swapped with a randomly chosen character in the key. No messages were successfully decrypted using this method. In 2011, Omran et al.[16] suggested mating two randomly selected parents using a one-point crossover, and mutating some solutions by swapping two characters in the key string. With a mutation rate of 0.2 and population size 20, they reached upwards of 100% key recovery after 50 generations.

Brezočnik et al.[17] compared several optimization algorithms in terms of how much of the Vigenère key they were able to retrieve. They found that the artificial bee colony algorithm performed almost just as well as particle swarm optimization, but they were both outperformed by genetic algorithms on longer key lengths. Only the differential evolution algorithm managed to recover the entire key, PSO performing second best. The worst results came from their cuckoo search algorithm.

Mekhaznia and Menai[12] that ant colony optimization could recover up to 43% of the Vigenère key.

In 2015, Bhateja et al.[6] used Cuckoo Search to solve Vigenère ciphers and found that it could outperform both genetic algorithms and Particle

Swarm Optimization. Their cuckoo search algorithm used Yang and Deb's implementation with fixed step size parameter, and a fitness function based on the 20 most frequent monograms and 25 most frequent digrams, where monograms are weighted 0.23 and digrams are weighted 0.77

Playfair

In 2012, Negara showed that Playfair is a good candidate for permutation-based evolution[18]. Their fitness function uses monograms and digrams. Their GAPFC algorithm starts with random sets of distinct letters as the population and evolves with one of two crossover operators and mutation operators at random. The crossover chooses a random index in both parents and swaps either the initial or final parts of the key. One mutation operator splits the key at a random index and switches the parts. The other mutation operator swaps two indexes. The algorithm also takes a key length as a parameter, meaning that it can limit its search. For testing, the same plaintext is successively enciphered using the keys from three test sets (3, 4 and 5-letter keys, respectively). With a population size of 1000, the three-letter keys were found after 1-17 generations, the four-letters keys after 8-38 and the five-letter keys after 16-83. As a second test, the same ciphertext was enciphered with 20 related keys of length 6, obtained by appending all possible letters to the word "relat" to produce a key with distinct letters. Tests resulted in five out of twenty keys being successfully found after 1000 generations by using 2000 population. By doubling the number of generations, this number increased to nine out of twenty. For the fourth test, a different and longer plaintext was enciphered using the 7-letter key "GABRIEL". The algorithm found the key after between 64 and 174 generations. The same test was run with an eight-letter key "PLAYFIRC", which the algorithm found after between 100 and 279 generations. The conclusion is that the results and efficiency is influenced by the genetic operators used, the parameters settings, the fitness function and the enciphered text length. Generic evolutionary schemes such as this one are potentially useful tools in analyzing and solving both cryptanalysis and cryptographic problems.

Cowan shows in [19] that the most efficient way of changing a Playfair key is to randomly swap two rows or columns in the matrix. The Simulated Annealing program should include a mix of letter, column and row swapping to get the best results. He also refers to Jan Stumpel's technique of flipping the entire key matrix around an axis, but find that this is of second order advantage compared to swapping rows and columns. Cowan also explains that when solving Playfair ciphers with SA, using digrams will not solve short ciphers, and trigrams will only solve the easiest ones. To efficiently solve them, we should also introduce quadgrams. There is very high variability in his results, taking between 2.6 and 115.3 million keys to find the key for an 86-letter ciphertext, and as low as 0.27 million for a 124-letter ciphertext. As for starting temperature, Cowan suggests using the following equation, however this is likely to be customized to his data sets.

$$T = 10 + 0.087 \cdot (length - 84)$$

.

2.5.2 Modern ciphers

DES

Genetic algorithms have been used previously[20] to crack DES without much success, even when evolving the keys as bit strings instead of character strings. Mekhaznia et al. [21] managed to get up to 100% bit-recovery in SDES, 80% for 4-round DES and 40% for full DES on large ciphertexts. The same paper also attempted to use Particle Swarm Optimization by calculating fitness and velocity, and flipping a random bit in the key only if the velocity exceeds 0.5.

Nalini and Rao used Simulated Annealing against SDES, or simplified DES[22], and compared the results to those of a genetic algorithm and tabu search. Their experiments used ciphertexts of length 200, 500 and 1000, and simulated annealing was able to find on average 7.5, 8.4 and 9.2 out of 10 key bits, respectively. The genetic algorithm would find 7.4, 8.1 and 9.1 key bits respectively.

Sharma, Pathak and Sharma's 2012 paper[23] attempts to break Simplified DES using binary particle swarm optimization. Their fitness function only uses monograms and digrams. Their genetic algorithm uses a ring crossover to produce the children from the parents, meaning that the two parents are connected, forming a ring, and two children are produced by choosing a random point and splitting the ring clockwise and counter-clockwise. In the binary PSO, the particle's personal best and global best is updated as in continuous PSO. The major difference is that the velocity of the particles uses the probability that a bit takes on 0 or 1, meaning that the velocity must be restricted within the range $[0,1]$. With self-recognition parameter 2, social parameter 2, inertia weight between 0.99 and 1, population 100, 50 iterations and $r_{mute} = 0.004$, BPSO managed to find 7/10 bits with 200/400 characters of ciphertext, 8/10 with 600 characters of ciphertext, 9/10 with 800 characters of ciphertext, 10/10 bits with 1000/1200 characters of ciphertext.

Khan, Shahzad and Khan's 2010 paper[24], and Khan, Ali, and Durrani's 2013 paper[25] attempts to use Binary Ant Colony Optimization to crack four-rounded DES. In order to apply their ACO to the problem, the search space has to be formed in a directed graph-like structure that essentially represents a bit string of length 64. From a node, when an ant decides which node to move to next, which in this case is a '0' or a '1' node, it uses two parameters to calculate the probability of moving to a particular node; first, the distance to that node and second, the amount of 'pheromone' on the connecting edge. Their attack relies on known plaintext-ciphertext pairs, and they seed their population with the ciphertext XOR'd with its corresponding plaintext. For their experiments, they use 4 ants, pheromone influence factor $\alpha = 1.5$ and heuristic influence factor $\beta = 1$. Their fitness function is the number of same bits in identical positions between the original ciphertext generated using the original secret key, and the candidate ciphertext generated by key from the ant algorithm. For four-rounded DES, this attack found a maximum of 19 bits with 1000 generations and four ants and some clever parameter choices. For one-, two- and three-round DES, this attack reaches a success rate of 99-100% after 10000 generations.

AES

In 2019, Grari et al.[26] proposed using ant colony optimization to find the key of simplified-AES using known ciphertext/plaintext pairs. Assuming we know a part of plaintext P and its corresponding ciphertext C, we use ant colony optimization to decrypt the known ciphertext with a candidate key and evaluate its fitness against the known plaintext:

$$F(K_c) = \frac{\sum_{i=1}^{z} \#(GP_i \oplus P_i)}{z \cdot 16}$$

Where z is the number of known plaintext-ciphertext pairs (P_i, C_i), K_c is the candidate key, GP_i is the plaintext produced by decrypting C_i with K_c.

Their first experiment was to find the optimal number of ants that allows finding the key in a minimal search space. Results show that the algorithm could not find the correct key with less than 40 ants. The optimal result was 120 ants and 43 generations, meaning a total of 5160 keys browsed. Their approach only needs 2 plaintext-ciphertext pairs to find the correct key.

RSA

Genetic algorithms show some promise for integer factorization, according to Rutkowski and Houghten[27]. They use three genetic algorithms. Population size was arbitrarily chosen as 2000 for all algorithms. Number of generations is also 2000. Their simple genetic algorithm represents the candidate keys as bit strings where the leftmost bit is always set to 1. They use a fitness of $f(p) = N \pmod{p}$. This means that the chromosome represents one of the primes, not both. The initial population is generated randomly. The algorithm uses a two-point crossover, where two points are randomly chosen, all values between these points are copied to one child, and all other values are copied to the other child. Chromosomes are mutated by flipping a random bit. In their second genetic algorithm, they exploit the property that all primes $p > 3$ must satisfy $p = 6m \pm 1$ to reduce the search space. Each chromosome is now a bit string starting with 1 representing m in the equation. Fitness is still calculated with $f(p) = N \pmod{p}$, but both $p = 6m + 1$ and $p = 6m - 1$ is examined and the solution with the lowest fitness is returned. The initial population is generated in the same way as before, and crossover and mutation is the same. In the third algorithm, they force every chromosome to be a probable prime by regenerating the chromosome until it is a probable prime. The crossover operator is modified to run the resulting children through the primality test and potentially generating new children by choosing new points, up to a maximum number of times equal to two times the length of the chromosome. The mutation operator flips a random bit, tests for primality and potentially reverts the flip and chooses another random bit to flip, up to a maximum number of tries equal to the length of the chromosome. Each of the three genetic algorithms were run for up to 16 data sets of N, half of which were taken from earlier literature. The first simple genetic algorithm was able to factorize a 17-digit number after an average of 1564 generations, which outperformed the Yampolskiy's algorithm. The "chromosome is m" algorithm, using 100% crossover and mutation rates, took fewer generations on average to find one of the correct primes than the previous algorithm (12 vs 55 generations). This GA was also able to factor a larger N. The simple GA was able to factor up to the 38-bit number 100% of the time, while the "chromosome is m" GA was able to factor up to the 44-bit number 100% of the time. It was also the best performing GA in factoring large semi-primes, being able to factor a 19-digit semi-prime 1/30 of the time. For the last GA, the primality test GA, the best crossover rate was 50% and 95% mutation rate. Overall, this GA found a correct prime number earlier in the evolution. For most data sets, the average number of generations were less than 35. The trade-off, however, is that the GA has to repeat operations to maintain the primality criteria at each step. This also decreased the maximum length of N the GA was able to factor consistently. The GA was able to factor up to 36-bits 100% of the time, but was able to factor a 54-bit number twice.

Mishra and Chaturvedi used a firefly algorithm to attempt to factorize primes[28]. While this not an algorithm we will be using, it is similar to Particle Swarm Optimization, and does have some good points, such as suggesting that the search space should be limited to $p \in (10^{d-1}, \sqrt{N}$, where d is the number of digits in the square root of N floored to an integer value. The experiments use 1000 generations, after which the run is deemed a success or failure. The fitness function used is $f(x) = N \pmod{x}$. The algorithm was tested on 10 datasets based on the number of digits of factors, two of each kind. They conclude that the Firefly Algorithm is a very promising metaheuristic in solving the prime factorization problem, with some tuning and modifications.

SPECK/SIMON

There does not seem to be any recorded attempt at using optimization algorithms to attack SPECK/SIMON, but presumably this will yield about the same results as DES/AES.

Chapter 3

Method

This project requires a unified way of testing each combination of cipher and optimization algorithm. For this reason, a C++ framework using pagmo and Crypto++ was developed[1]. The framework is given an optimization algorithm and encryption algorithm or cipher, as well as a ciphertext produced by the chosen cipher. This framework is then used to iteratively test each optimization algorithm's efficiency on each cipher. These results can then be used to compare each optimization algorithm to each other as well as to results from earlier literature and related works.

3.1 Implementation

Pagmo[29] is a C++ library for parallel optimization which provides a unified interface to optimization algorithms and problems. It contains efficient implementations of nature-inspired and evolutionary algorithms as well as state-of-the-art optimization algorithms, which can be used to solve constrained, unconstrained, single-objective, multi-objective, continuous and integer optimization problems, as well as stochastic and deterministic problems.

For pagmo to be able to solve a problem, it must be formulated in a certain way. Pagmo problems consist of a problem size or dimension, linear constraints, a fitness function and lower and upper bounds for each variable. Each cipher that is used in this project has a corresponding generic pagmo problem, which will be explained in their own subsec-

[1]https://github.com/sm0xe/OptimizationCryptanalysis

tion below. The ciphertext is passed to this generic pagmo problem to create a specific problem that can be solved by the chosen optimization algorithm. The optimization algorithm is given this specific problem formulation and is then initialized with a population of the appropriate size. Then the population is evolved for a chosen number of generations according to the optimization algorithm, guided by the problem's fitness function, constraints and bounds. Finally, the algorithm will output the best solution in the population. Each generation is logged for each run such that we are able to calculate the average number of generations or fitness evaluations needed to reach the solution.

In the next sections, we will describe how each optimization algorithm in pagmo works.

Simple Genetic Algorithm

The Simple Genetic Algorithm provided by PaGMO (pagmo::sga) supports two different selection schemes, four crossover schemes and three mutation schemes.

The two selection methods provided are "tournament" and "truncated". Tournament selection divides the population into random groups of size $param\_s$ and selects each offspring as the one having the minimal fitness in the group. Truncated selection selects $param\_s$ of the best chromosomes from the entire population.

The four crossover schemes are "single", "exponential", "binomial" and "sbx". The single-point crossover scheme chooses a random point in the parent chromosome and inserts the partner chromosome thereafter. The exponential crossover scheme selects a random point in the parent chromosome and for each successive gene inserts the partner values with a probability cr. The binomial crossover inserts each gene from the partner with probability cr.

The three different mutation schemes are "uniform", "gaussian" and "polynomial". Uniform mutation randomly samples from the bounds. Gaussian mutation samples around each gene using a normal distribution with standard deviation proportional to $param\_m$ and the width of the bounds.

All experiments with GA use a crossover rate of 0.75, a single-point crossover, polynomial mutation with $param\_m = 1$ and tournament selection with $param\_s = 5$. We will also perform experiments with three different mutation rates; $m = 0.02$, $m = 0.2$ and $m = 1.0$.

Customized Simple Genetic Algorithm

The implementation of Simple Genetic Algorithm in pagmo does not provide a mutation operator that simply swaps two indexes, which is often used in related literature for ciphers such as the monoalphabetic substitution cipher. For the purpose of adequately comparing results with earlier literature, I implemented a customized version of SGA that provides this mutation operator, as well as optionally maintaining unique values in the chromosome, which will prove useful for monoalphabetic substitution, Playfair and columnar ciphers.

All experiments use the same parameters as the Simple Genetic Algorithm, except for the mutation operator, which is the two-index swap. In addition, for monoalphabetic substitution and Playfair ciphers, the algorithm will enforce unique values after crossover.

Self-Adaptive Differential Evolution

jDE/iDE are improvements on the original differential evolution algorithm by introducing parameter self-adaptation. Two variants are implemented in PaGMO. The first (jDE), as proposed by Brest et al., does not use DE operators to produce new values for the weight coefficient and the crossover probability, and as such uses parameter control, not parameter self-adaptation. The second variant (iDE), inspired by Elsayed et al., uses a variation of the selected DE operator to produce new CR and F parameters for each individual. By default, PaGMO's SaDE uses jDE.

All experiments will use jDE self-adaptation and the default mutation variation. The default mutation variation constructs a donor vector from three randomly selected individuals in the population and performs an exponential crossover with the individual to be mutated.

Simulated Annealing

PaGMO implements Corana's version of Simulated Annealing with adaptive neighborhood[30]. The algorithm is essentially an iterative random search procedure with adaptive moves along the coordinate directions. It is not suitable for multi-objective problems, nor for constrained or stochastic optimization. As opposed to the other optimization algorithms, Simulated Annealing does not use a fixed number of generations. It is not a population-based algorithm, and as such only operates on a single individual. If the population size is larger than 1, the algorithm will improve on the best individual in the initial population.

All experiments will use default parameters: starting temperature $Ts = 10$, final temperature $Tf = 0.1$, number of temperature adjustments $n\_T\_adj = 10$, number of adjustments of the search range at a constant temperature $n\_range\_adj = 1$, number of mutations used to compute the acceptance rate $bin\_size = 20$, and starting range for mutating the decision vector $start\_range = 1.0$.

Particle Swarm Optimization

PSO is a population-based algorithm inspired by the foraging behavior of swarms. Each particle has memory of the position where it achieved the best performance $\mathbf{x_i^l}$ (local memory) and the best decision vector $\mathbf{x^g}$ in a certain neighborhood, and uses this to update its positions according to the equation

$$\begin{aligned} \mathbf{v}_{i+1} &= \omega\left(\mathbf{v}_i + \eta_1 \mathbf{r}_1 \cdot \left(\mathbf{x}_i - \mathbf{x}_i^l\right) + \eta_2 \mathbf{r}_2 \cdot (\mathbf{x}_i - \mathbf{x}^g)\right) \\ \mathbf{x}_{i+1} &= \mathbf{x}_i + \mathbf{v}_i \end{aligned} \tag{3.1}$$

The algorithm is suitable for box-bounded single-objective unconstrained optimization, with both continuous and integer values.

All experiments will use parameters $\omega = 0.7298$, forces in direction of local and global best $eta_1 = eta_2 = 2.05$, maximum velocity $max\_vel = 0.5$, and swarm topology $lbest$ with degree 4.

Extended Ant Colony Optimization (gaco)

Ant colony optimization is modeled on the behavior of an ant colony. Artificial ants locate optimal solutions by moving through a parameter space representing all possible solutions. The ants record their positions and quality of their solutions such that in later iterations more ants locate better solutions. PaGMO implements a version of this algorithm called extended ACO, which was originally described by Schlueter et al.[4]. This version generates future generations by using a multi-kernel gaussian distribution based on three values which are computed depending on the quality of each previous solution. The solutions are then ranked through an oracle penalty method. This algorithm can be applied to box-bounded single-objective, constrained and unconstrained optimization, with both continuous and integer values.

All experiments use parameters: kernel size 20, oracle penalty 100, convergence rate $q = 0.01$.

Artificial Bee Colony

Artificial Bee Colony is an optimization algorithm based on the foraging behavior of honey bee swarms, proposed by Karaboga in 2005[5]. PaGMO's implementation of the algorithm is based on the pseudo-code provided by Mernik et al.[31]. It is suitable for box-constrained single-objective continuous optimization problems.

All experiments use 200 generations and default parameter $limit = 20$, the maximum number of trials for abandoning a source.

Cuckoo Search

Pagmo does not provide an implementation of the Cuckoo Search algorithm. Therefore, I decided to implement the algorithm in a way that pagmo can use. My implementation follows Walton's Improved Cuckoo Search implementation[1].

All experiments will use 200 generations and default parameters $p_a = 0.25$ and $A = 1.0$

3.1.1 Optimization Problems

Pagmo requires optimization problems to be formulated a certain way. This includes a problem size or dimension, optional linear constraints, bounds on each variable and a fitness function. In this section, we will explain how each cipher is formulated as an optimization problem.

Caesar/Shift Cipher The corresponding optimization problem is a single variable in the range 0 to 25, representing the shift and no other constraints. The solution's fitness is evaluated by passing the decryption to the weighted chi-squared function described in section 2.3

Columnar Cipher The corresponding optimization problem is an array of values between -1 and the length of the array. The decryption function will read the array until the first -1 is discovered, and decrypt according to the number of columns read and the relative ordering each index represents. This is done to make the optimization problem more flexible in terms of key length, by finding the number of column and ordering at the same time. The solution's fitness is evaluated by passing the decryption to the weighted chi-squared function described in section 2.3

Monoalphabetic Substitution Cipher We defined two problems, one where the problem is constrained to having unique values in the array using pagmo, and one without these constraints. However, experiments during development showed that not having these constraints in the problem definition and instead maintaining unique values during evolution was more efficient. Every candidate solution is an array of length 26 with values between 0 and 25. The solution's fitness is evaluated by passing the decryption to the weighted chi-squared function as described in section 2.3.

Vigenère The most likely key length is determined by using index of coincidence[**ioc**] and passed to the optimization problem. The decision vector is of the same length as the key and every value in the vector is between 0 and 25, each representing a shift. The solution's fitness is evaluated by passing the decryption to the weighted chi-squared function described in section 2.3

Playfair The decision vector has a length of 25, and every value is between 0 and 24. The decryption function creates a valid Playfair key out of the decision vector by removing 'J' from the key and making sure all values are unique. The solution's fitness is evaluated by passing the decryption to a weighted chi-squared function that disregards the frequency of 'X'.

DES, AES and SPECK The decision vector contains 8, 12 or 16 values between 0 and 255, for DES, SPECK and AES respectively. The decision vector is then passed to Crypto++'s implementation of DES, SPECK or AES to produce a plaintext which is then evaluated with the weighted chi-squared function from section 2.3

RSA Factorization Problem I have defined two pagmo problems for RSA Factorization. In both problems we have to store the large integers we work with in a clever way as to fit within the constraints of the double type. I have chosen to represent the integers as their prime factorization by storing only the exponents for each prime number in the decision vector and calculating the integers by taking the product of each prime number to the power of the corresponding exponent in the decision vector. This representation also makes it simpler to calculate the squares used in the fitness function $x^2 - y^2 \pmod{n}$.

Three pagmo problems were defined for RSA. In the first one, the decision vector represents the prime factorization of two integers, x and y. The first half of the array contains the exponents for prime numbers in the prime factorization of x, and the second half for y. For example, if the decision vector is of length 8 and contains the values

$$0, 1, 2, 3, 1, 3, 2, 0$$

, this would represent the two integers $x = 2^0 \cdot 3^1 \cdot 5^2 \cdot 7^3 = 25725$ and $y = 2^1 \cdot 3^3 \cdot 5^2 \cdot 7^0 = 1350$. The goal is to find two integers, x and y that satisfy the equation $x^2 + y^2 \equiv 0 \pmod{n}$ for the public key n. This problem calculates x^2 and y^2 from the decision vector and uses the fitness function $f(x^2, y^2, n) = |x^2 - y^2| \pmod{n}$. However, the solution will be rejected by using a high fitness penalty if either $x = 0$ or $y = 0$, or $x = y$ or $x + y = n$.

The second problem uses the decision vector to represent a single integer in the same way as the first problem, but uses the fitness function $f(x) = x \pmod{n}$, with a high fitness penalty if $x \notin [\sqrt{n}, 10^{\lceil \log_{10} \sqrt{n} - 1 \rceil}]$.

The last problem exploits the property that all primes $p > 3$ must satisfy $p = 6m \pm 1$ by using the decision vector to represent m as a bit string of length $\lceil \log_2 \frac{\sqrt{n}+1}{6} \rceil$ with a leading 1 bit. The fitness function is $f(m) = \min 6m + 1\%n, 6m - 1\%n$. This is the only problem that yielded any results out of the three, and the only one whose results will be presented in the next chapter.

3.2 Experiments

When constructing the experiments, we had two options: create experiments for each combination with comparable parameters to earlier research, or use the same parameters for each cipher-optimization pair such that they can be compared to each other. We chose the latter.

Each cipher has at least one sample ciphertext that we run through the program to gather results. The program will output the best fitness

Table 3.1: Ciphertexts used

Cipher	Ciphertext	Key	Plaintext fitness	Plaintext length
Columnar	JulesVerne1.txt	[1,2,0,3]	10613.59	18957
Caesar	caesar.txt	Enc: HIJKLMNOPQRSTUVWXYZABCDEFG	5428003.61	168
		Dec: TUVWXYZABCDEFGHIJKLMNOPQRS		
Caesar	caesar2.txt	Enc: HIJKLMNOPQRSTUVWXYZABCDEFG	4550880.57	216
		Dec: TUVWXYZABCDEFGHIJKLMNOPQRS		
Caesar	JulesVerne4.txt	Enc: HIJKLMNOPQRSTUVWXYZABCDEFG	32659.92	23058
		Dec: TUVWXYZABCDEFGHIJKLMNOPQRS		
Vigenère	JulesVerne1_Vigenere_key10.txt	JULESVERNE	10613.59	18957
Vigenère	JulesVerne_Vigenere_key5.txt	JULES	10613.59	18957
Vigenère	JulesVerne_Vigenere_key3.txt	KEY	10613.59	18957
Playfair	JulesVerne.txt	VERNABCDFGHIKLMOPQSTUWXYZ	10630.32	15557
Playfair	JulesVerne1.txt	PLAYFIREXMBCDGHKNOQSTUVWZ	10867.69	15421
MSub	JulesVerne1.txt	Enc: NBAJYFOWLZMPXIKUVCDEGRQSTH	26888.37	18984
		Dec: CBRSTFUZNDOIKAGLWVXYPQHMEJ		
DES	JulesVerne1_des.txt	deadbeefbabe1337	10613.59	37921
AES	JulesVerne1_aes.txt	deadbeefbabe1337feed7abe1037def0	10613.59	37921
SPECK	JulesVerne1_simon.txt	deadbeefbabe1337feed7abe	10613.59	31121

value in the population for each generation such that we can track improvements each generation. For each ciphertext, we will run the program five times with each optimization algorithm and each population size (20,30,50,70), for 200 generations. For RSA, we will run the program ten times with each optimization algorithm and each population size, for a maximum of 20000 generations. We then calculate how much of the key has been recovered and average number of generations needed to recover the key, if it was recovered in its entirety, as well as the average number of evaluations. After key recovery, the best metric for efficiency is the number of evaluations needed to reach full key recovery.

Below is a table of the datasets used in our experiments. All ciphertexts named "JulesVerne" are the two first chapters of Jules Verne's book "Twenty Thousand Leagues Under the Seas"[32], encrypted with the corresponding cipher and key in the table.

The "ciphertexts" or datasets used for RSA are different from the ones for the other ciphers. This is because the RSA factorization problem operates on the public key of the cipher, not a ciphertext. In the table below you will find the public n for each dataset, its length in digits and bits, and its factorization.

Table 3.2: Data sets used for RSA

Data set	n	Digits	Bits	p	q
rsa1.txt	10909343	8	24	2693	4051
rsa2.txt	29835457	8	25	4001	7457
rsa3.txt	392913607	9	29	17911	29137
rsa4.txt	5325280633	10	33	57731	92243
rsa5.txt	42336478013	11	36	174169	243077
rsa6.txt	272903119607	12	38	374989	727763
rsa7.txt	11683458677563	14	44	2595899	4500737
rsa8.txt	51790308404911	14	46	5581897	9278263
rsa9.txt	115137038087959	15	47	10037141	11471099
rsa10.txt	8335465900089539	16	53	90745723	91855193
rsa11.txt	10380088039872631	17	54	101858333	101907107
rsa12.txt	253422413591685001	18	58	501900991	504925111
rsa13.txt	1160633764479964633	19	61	1004922797	1154948189
rsa14.txt	31625125947164338313	20	65	3510002059	9010002107
rsa15.txt	454367322351811534933	21	69	13545006127	33545006779
rsa16.txt	4500000514520012390279	22	72	50000003993	90000003103

Chapter 4

Results

In this chapter we will present the results of our experiments. All box plots and tables can be found in the Appendix, but only the most representative or interesting tables or figures will be reproduced here.

4.1 Caesar/Shift cipher

The Simple Genetic Algorithm is able to correctly determine the key in all runs for all population sizes on all ciphertexts, in very few generations. The custom Genetic Algorithm has between 20 and 100% average key recovery, however in all runs where the key is found, it is found in the first generation. For the Custom Genetic Algorithm, $m = 1.0$ seems to give the best average key recovery out of the three. Self-Adaptive Differential Evolution, Particle Swarm Optimization, Ant Colony Optimization, Artificial Bee Colony Optimization and Cuckoo Search all found the key in all runs. Simulated Annealing missed the key in one run on one of the shorter ciphertexts.

4.2 Columnar Transposition Cipher

Our experiments had little success with using the Simple Genetic Algorithm with $m = 0.02$, only succeeding in one run, and achieving an average key recovery of 22.5% across all runs. With $m = 0.2$, the algorithm fared slightly better, succeeding in nine runs and reaching 57.5% average key recovery across all runs. Finally, $m = 1.0$ lead the algorithm

Table 4.1: Columnar cipher - Average key recovery (top 5)

Optimization Algorithm	Average key recovery
Ant Colony	96.25%
Particle Swarm	86.25%
Self-Adaptive Differential Evolution	73.75%
Artificial Bee Colony	62.5%
Simple Genetic Algorithm with $m = 0.2$	57.5%

to the key in eleven runs and 55% average key recovery.

The Custom Genetic Algorithm did not manage to find the key in any runs with $m = 0.02$ or $m = 1.0$ (26.25% and 18.75%), however it did find the key in three runs with $m = 0.2$ (33.75%).

Self-Adaptive Differential Evolution found the key in fourteen runs (73.75% key recovery), with a population of 20 and 70 individuals giving the highest success rates, but also the highest average number of generations.

Simulated Annealing did not find the key in any runs, but achieved an average of 35% key recovery, outperforming the Custom Genetic Algorithm overall.

Particle Swarm Optimization and Ant Colony Optimization had the highest success rates overall. Particle Swarm Optimization found the key in seventeen runs (86.25% key recovery), with population sizes 50 and 70 finding the key in every run. while the Ant Colony Optimization algorithm found it in nineteen (96.25% key recovery). The Artificial Bee Colony Optimization found the key in eleven runs (62.5% key recovery), and Cuckoo Search in eight (48.75% key recovery).

4.3 Monoalphabetic substitution ciphers

In our experiments, none of the algorithms in the default pagmo library was able to find the key in its entirety after 200 generations. However, our Custom Genetic Algorithm, which enforces unique values, found the key in five runs with $m = 0.02$ as shown in table B.28, and in ten runs with $m = 0.2$ and $m = 1.0$ as shown in tables B.29 and B.30. The lowest average number of evaluations were 3480, which was achieved with the

Table 4.2: (B.21) Columnar Transposition cipher - Particle Swarm Optimization

Ciphertext	Pop.	Avg. rec.	Max. rec.	100% rec.	Avg. gen.	Avg. eval.
JulesVerne1.txt	20	65.00%	100.00%	3/5	50.33	1006.67
JulesVerne1.txt	30	80.00%	100.00%	4/5	55.50	1665.00
JulesVerne1.txt	50	100.00%	100.00%	5/5	84.80	4240.00
JulesVerne1.txt	70	100.00%	100.00%	5/5	20.20	1414.00

Table 4.3: (B.22) Columnar Transposition cipher - Ant Colony Optimization

Ciphertext	Pop.	Avg. rec.	Max. rec.	100% rec.	Avg. gen.	Avg. eval.
JulesVerne1.txt	20	100.00%	100.00%	5/5	48.20	944.00
JulesVerne1.txt	30	85.00%	100.00%	4/5	35.50	1035.00
JulesVerne1.txt	50	100.00%	100.00%	5/5	36.00	1750.00
JulesVerne1.txt	70	100.00%	100.00%	5/5	16.80	1106.00

Custom Genetic Algorithm with $m = 0.2$ (average 72.5% key recovery) and a population size of 20. The highest average key recovery overall (79.04%) was achieved by the Custom Genetic Algorithm with $m = 1.0$. Even though we managed to retrieve the key in some runs, we were not able to achieve better results than the experiments in previous research papers.

4.4 Vigenère

Our Simple Genetic Algorithm performed fairly well on Vigenère, finding the entire key in 54/60 runs with $m = 0.02$, having the most difficulty with key length 10. With $m = 0.2$, the algorithm found all keys of all three length in every run. Using $m = 1.0$ gave the worst results, only succeeding in nine runs, with 81.67% average key recovery overall. With a population of 20 and $m = 0.2$, the genetic algorithm found the key of length 3 in 13.4 generations, length 5 in 29.4 and length 10 in 114.8. In comparison, Omran et al. reached upwards of 100% key recovery after

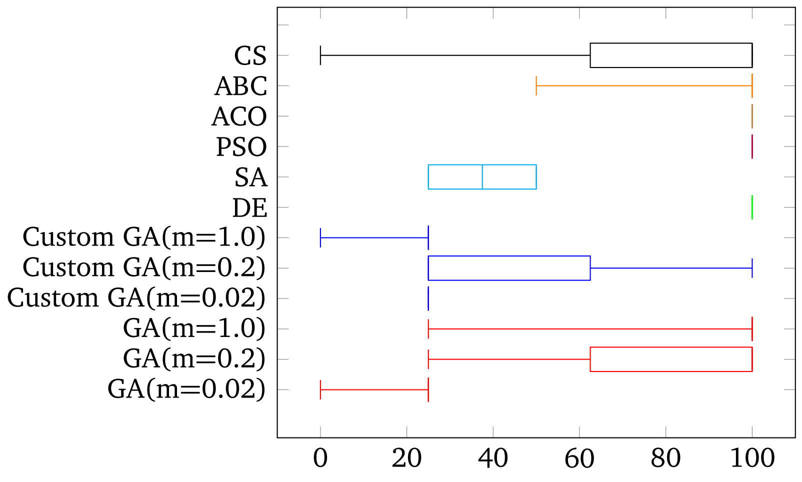

Figure 4.1: (A.8) Columnar Cipher - Population size = 70

Table 4.4: Monoalphabetic substitution - Average key recovery (top 5)

Optimization Algorithm	Average key recovery
Custom Genetic Algorithm with $m = 1$	79.04%
Custom Genetic Algorithm with $m = 0.2$	72.5%
Custom Genetic Algorithm with $m = 0.02$	71.54%
Self-Adaptive Differential Evolution	6.54%
Artificial Bee Colony	5%

50 generations with the same parameters.

The Custom Genetic Algorithm did not perform as well, succeeding in seven runs with $m = 0.02$, and in five runs with $m = 0.2$ and $m = 1.0$. It was only able to recover keys of length 3 and 5, but did so with relatively few fitness evaluations and generations.

Self-Adaptive Differential Evolution found the key in all 60 runs, with relatively few fitness evaluations. With the same type of algorithm, Brezočnik achieved about 95% average key recovery for keys of length 8 and about 93% for keys of length 12 after 50000 fitness evaluations, whereas our experiments achieved 100% average key recovery with far less fitness evaluations.

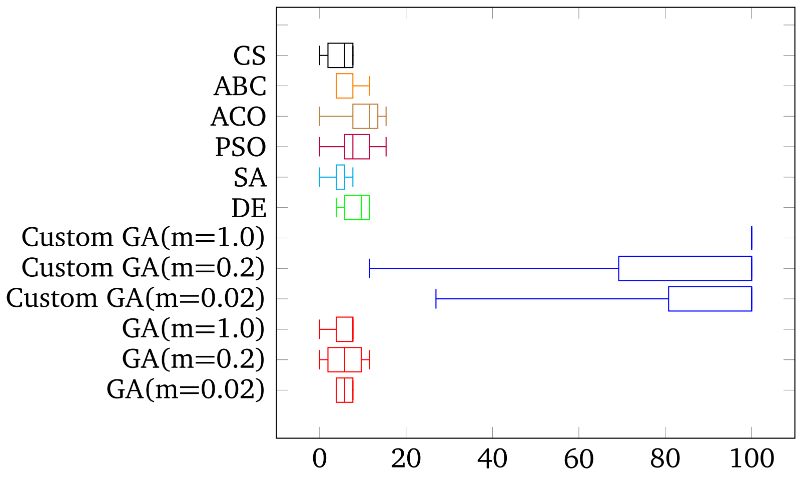

Figure 4.2: (A.12) Monoalphabetic Substitution cipher - Population size = 70

Simulated Annealing found the key with length 3 in all runs, but only reached 80% and 90% max key recovery for length 5 and 10 respectively.

Particle Swarm Optimization found the key with length 3 in all runs and the key of length 5 in 15/20 runs. It did not find the key of length 10 in any of the runs. Overall, the algorithm achieved 87.17% average key recovery. In comparison, Brezočnik recovered 3.8 out of 4 key elements (95% key recovery), 5.4 out of 6 key elements (90% key recovery), 6 out of 8 key elements (75% key recovery), and 7.8 out of 12 (65% key recovery) with the same algorithm and 50000 fitness evaluations.

The Ant Colony Optimization algorithm found the key of length 3 in 17/20 runs, and the key of length 5 in 12/20 runs. Overall, the algorithm had an average key recovery of 76.17%.

Artificial Bee Colony found the key of length 3 in 19/20 runs (98.33% average key recovery), the key of length 5 in 15/20 runs (94% average key recovery) and the longest key twice (76% average key recovery). To compare, Brezočnik achieved an average of 87% key recovery with keys of length 6, 80% with length 8 and 58% with length 12.

Cuckoo Search found the shortest key in only five runs (73.33% average key recovery), and the key of length 5 twice (57% key recovery). It was not able to find the key of length 10 in any run (44.5% average key recovery). In comparison, Brezočnik reached 65% key recovery with

Table 4.5: Vigenère - Average key recovery (top 5)

Optimization Algorithm	Average key recovery
Simple Genetic Algorithm with $m = 0.2$	100%
Self-Adaptive Differential Evolution	100%
Simple Genetic Algorithm with $m = 0.02$	98.28%
Artificial Bee Colony	89.44%
Particle Swarm Optimization	87.17%

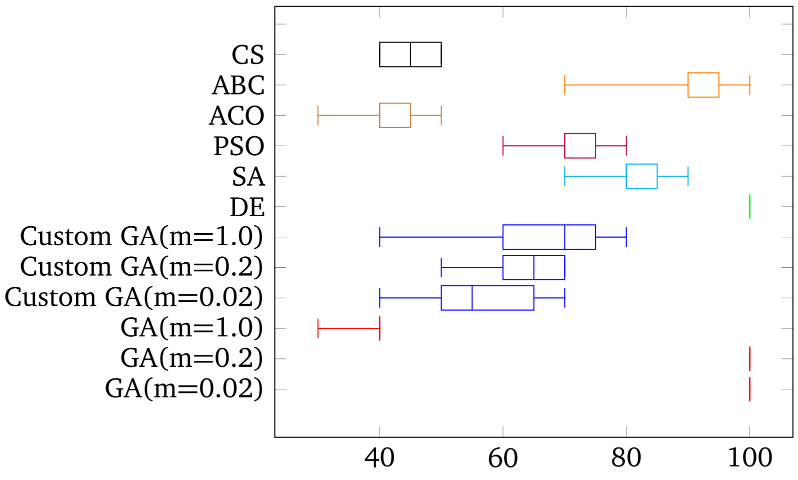

Figure 4.3: (A.24) Vigènere with key length 10 - Population size = 70

keys of length 4, 50% with length 6, 47.5% with length 8 and 33% with length 12. Bhateja found that CS could consistently find keys up to length 6 on short ciphertexts, and up to 18 on longer ciphertexts, however we were not able to achieve comparable results.

4.5 Playfair

The Simple Genetic algorithm recovered the key in four out of the 40 runs with $m = 0.2$. SGA with $m = 0.02$ and $m = 1.0$ failed to recover the key in any runs. The Custom Genetic Algorithm recovered the key once, using $m = 0.02$ and a population size of 70. Self-Adaptive Differential

Table 4.6: Playfair - Average key recovery (top 5)

Optimization Algorithm	Average key recovery
Particle Swarm	48.2%
Ant Colony	46%
Simple Genetic Algorithm with $m = 0.2$	44.4%
Self-Adaptive Differential Evolution	43.7%
Simple Genetic Algorithm with $m = 1.0$	40.40%

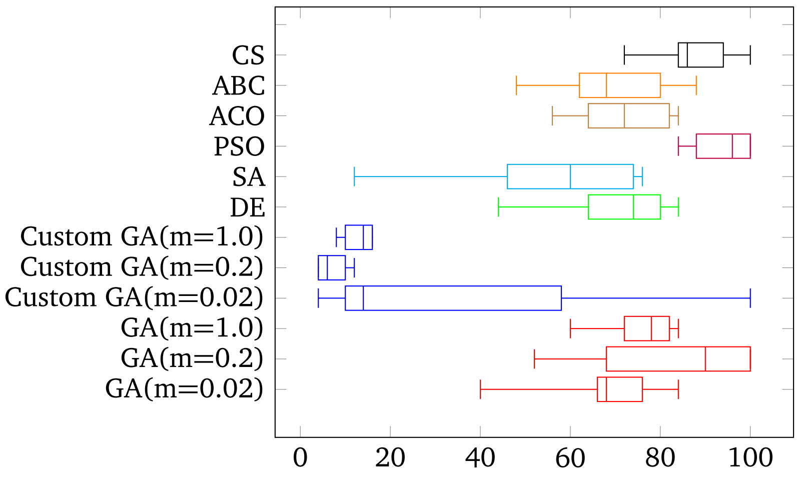

Figure 4.4: (A.28) Playfair cipher - Population size = 70

Evolution also recovered the key in one run, using a population size of 20. Simulated Annealing and Artificial Bee Colony Optimization did not recover the key at all. Particle Swarm Optimization recovered the entire key in only two runs with population size 70, but had the highest average key recovery out of all algorithms. Ant Colony Optimization recovered the key in two runs, with the second-highest average key recovery of 46&. Cuckoo Search recovered the key in one run with population size 70.

Table 4.7: DES - Average key recovery (top 5)

Optimization Algorithm	Average key recovery
Simple Genetic Algorithm with $m = 0.2$	52.66%
Self-Adaptive Differential Evolution	52.42%
Cuckoo Search	51.72%
Custom Genetic Algorithm with $m = 0.2$	51.41%
Particle Swarm	51.41%

Table 4.8: DES - Maximum key recovery (top 5)

Optimization Algorithm	Maximum key recovery
Custom Genetic Algorithm with $m = 0.2$	67.19%
Simple Genetic Algorithm with $m = 1.0$	65.62%
Particle Swarm	65.62%
Cuckoo Search	65.62%
Self-Adaptive Differential Evolution	64.06%

4.6 DES

Not a single run was able to recover the key from a full DES ciphertext. The best result we were able to get was 65.62% key recovery, which was achieved by the Custom Genetic Algorithm ($m = 0.2$) with a population of 20. In comparison, Mekhaznia et al., achieved about 45% key recovery after 1600 generations of DE, and about 60% key recovery after 1600 generations of PSO.

4.7 AES

As with DES, not a single run was able to recover the entire key from a full AES ciphertext. The best result was a 60.16% key recovery, which was achieved by the Simple Genetic Algorithm with $m = 1.0$, Ant Colony Optimization with population size 30 and Cuckoo Search with population size 70.

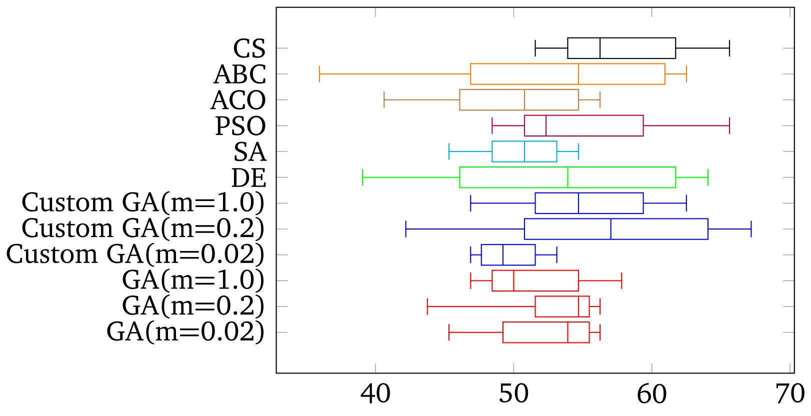

Figure 4.5: (A.32) DES - Population size = 70

Table 4.9: AES - Average key recovery (top 5)

Optimization Algorithm	Average key recovery
Particle Swarm	52.35%
Custom Genetic Algorithm with $m = 0.2$	51.92%
Artificial Bee Colony	50.86%
Cuckoo Search	50.86%
Ant Colony	50.20%

4.8 SPECK

Not a single run was able to recover the entire SPECK key. The best result was 67.71% key recovery, achieved by Cuckoo Search with population size 30.

4.9 RSA

Three different problem formulations were experimented with, but only the formulation proposed by Rutkowski and Houghten[27] yielded results. Therefore, we will only present the results from this problem formulation.

The Simple Genetic algorithm with found most consistency with $m =$

Table 4.10: AES - Maximum key recovery (top 5)

Optimization Algorithm	Maximum key recovery
Simple Genetic Algorithm with $m = 1.0$	60.16%
Ant Colony	60.16%
Cuckoo Search	60.16%
Simple Genetic Algorithm with $m = 0.2$	59.38%
Custom Genetic Algorithm with $m = 0.2$	59.38%

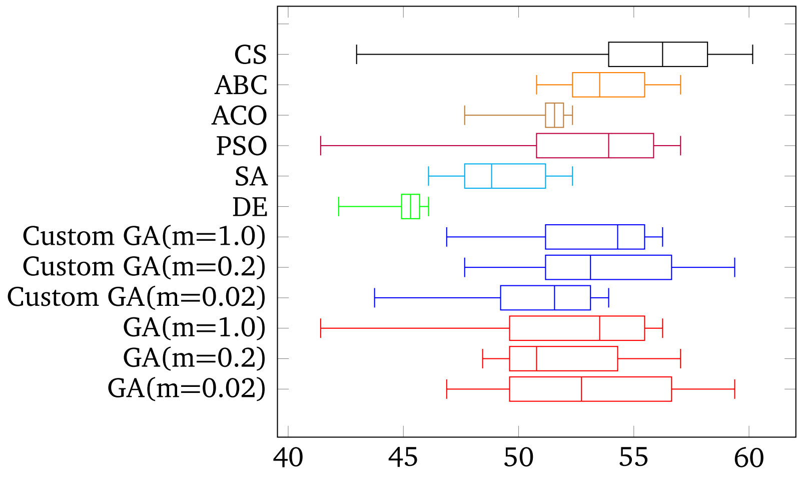

Figure 4.6: (A.36) AES - Population size = 70

1.0, with which it consistently factorized integers up to the 44-bit integer and up to the 61-bit integer with some luck, as shown in A.43 and B.99. The Custom Genetic Algorithm had a much lower success rate, but managed to factorize up to the 46-bit integer with some luck.

Self-Adaptive Differential Evolution factorized integers up to 33 bits with a high rate of success, and managed to factorize up to 54-bit integers.

Simulated Annealing was the least successful algorithm overall, only factorizing the 24-bit integer three times and the 25-bit integer once.

Particle Swarm Optimization performed somewhat well up to the 33-bit integer, factorizing the 47-bit integer once with a population of 50.

The Ant Colony Optimization algorithm factorized up to the 47-bit in-

Table 4.11: Speck - Average key recovery (top 5)

Optimization Algorithm	Average key recovery
Custom Genetic Algorithm with $m = 0.2$	52%
Simple Genetic Algorithm with $m = 1.0$	51.72%
Particle Swarm Optimization	51.67%
Cuckoo Search	51.62%
Custom Genetic Algorithm with $m = 1$	51.46%

Table 4.12: Speck - Maximum key recovery (top 5)

Optimization Algorithm	Maximum key recovery
Cuckoo Search	67.71%
Artificial Bee Colony	63.54%
Ant Colony	61.46%
Custom Genetic Algorithm with $m = 0.2$	61.46%
Simple Genetic Algorithm with $m = 0.02$	61.46%

teger with high success rate and the 61-bit integer once with a population of 20, as shown in figure A.50

The Artificial Bee Colony algorithm consistently factorized up to the 38-bit integer, and up to the 47-bit integer with somewhat high success, as shown in figure A.51

The success rate of Cuckoo Search rapidly declined after the 24-bit integer.

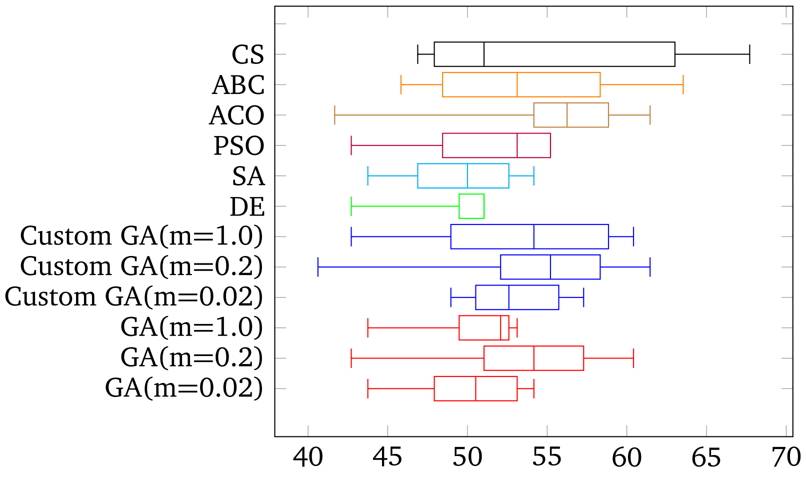

Figure 4.7: (A.38) Speck - Population 30

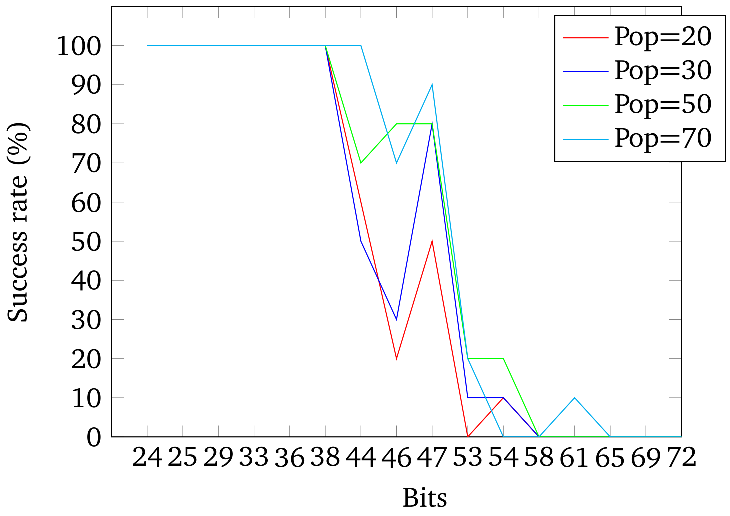

Figure 4.8: (A.43) RSA key recovery - Simple Genetic Algorithm (m=1.0)

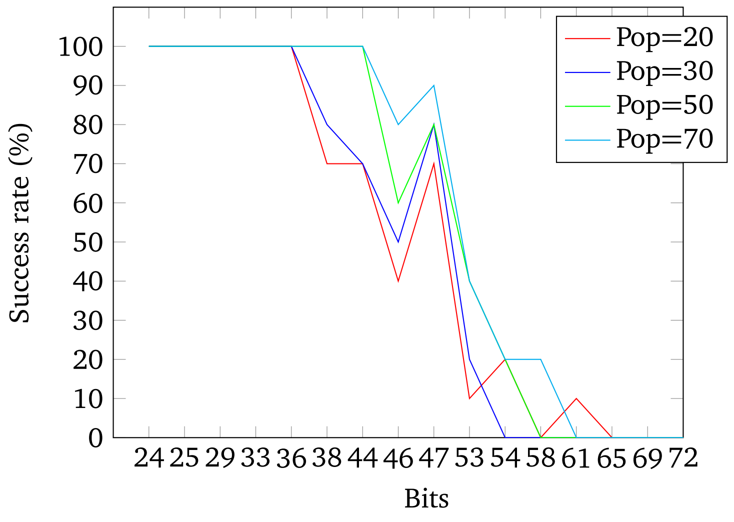

Figure 4.9: (A.50) RSA key recovery - Ant Colony Optimization

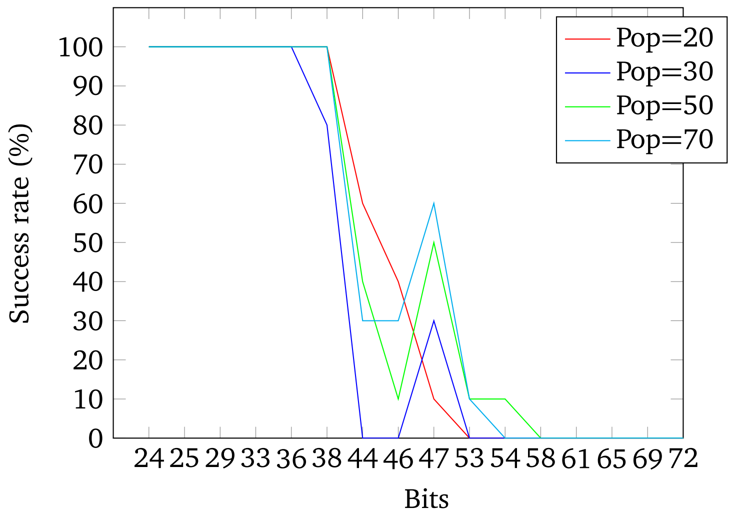

Figure 4.10: (A.51) RSA key recovery - Artificial Bee Colony Optimization

Chapter 5

Discussion

Our experiments have shown that given enough ciphertext to work with, out-of-the-box optimization algorithms perform reasonably well on decrypting several classical ciphers, and on small integers used in RSA. On classical ciphers such as the monoalphabetic substitution cipher, the out-of-the-box optimization algorithms from pagmo is heavily outperformed by the Genetic Algorithm customized for these exact ciphers. Our implementation of Walton's[1] Improved Cuckoo Search algorithm performed reasonably well on classical ciphers, but never outperformed any of the other algorithms overall on classic ciphers.

As one would expect, none of the optimization algorithms were able to correctly determine the entire key for DES/AES/SPECK. The highest key recovery that was achieved was 65.62% for DES, 60.16& for AES and 67.71 for SPECK. All algorithms averaged between 47% and 53% on all ciphers. The way key recovery is measured for DES, AES and SPECK is to count the number of correct bits in the key, meaning each bit has a 50% chance of being correct. Therefore, one could argue that the best key found by the algorithm is not much better than a randomly chosen bit string. However, this slightly outperforms previous research done by Mekhaznia et al., who measured key recovery in exactly the same way for DES.

There was not one optimization algorithm that performed well on all ciphers. For the Columnar cipher, Ant Colony Optimization and Particle Swarm Optimization were the clear winners. For the Monoalphabetic substitution cipher, only the Custom Genetic Algorithm was able to achieve any useful results. The Vigenère cipher was consistently broken by the

Simple Genetic Algorithm and Self-Adaptive Differential Evolution. Cuckoo Search achieved the highest maximum key recovery for AES and SPECK, and was outperformed by the Custom Genetic Algorithm on DES. For the RSA factorization problem, the Ant and Bee Colony Optimization algorithms were the most consistent overall.

Chapter 6

Limitations and Future work

One of the limitations of the experiments is the low number of runs for each ciphertext, optimization and parameter combination. Another limitation is that we did not have time to implement a simplified version of DES, or the so-called SDES algorithm, which would have allowed us to compare our results with more previous research. In our experiments, the fitness function only takes printable characters between A and Z in the decryption into consideration when calculating the chi-squared statistic. A DES/AES/SPECK decryption with an incorrect key usually produces unprintable characters which will only contribute to the total string length in the chi-squared statistic, which can be a limitation in our implementation.

For future work, one can implement SDES and Simplified AES, as well as run each experiment more times for a larger sample size. There is also a possibility that one can achieve even better results with DES/AES/SPECK by slightly changing the fitness function used. The Improved Cuckoo Search algorithm seems to show promise for ciphertext-only attacks on DES, AES and SPECK, and one might be able to achieve even higher key recovery with other parameters than the ones used in our experiments.

Bibliography

[1] S. Walton, O. Hassan, K. Morgan and M. Brown, 'Modified cuckoo search: A new gradient free optimisation algorithm,' *Chaos, Solitons & Fractals*, vol. 44, no. 9, pp. 710–718, 2011, ISSN: 0960-0779. DOI: `https://doi.org/10.1016/j.chaos.2011.06.004`. [Online]. Available: `https://www.sciencedirect.com/science/article/pii/S096007791100107X`.

[2] K. A. Høivik, 'Optimization in cryptanalysis,' Unpublished paper, Norwegian University of Science and Technology, Nov. 2020.

[3] R. Beaulieu, D. Shors, J. Smith, S. Treatman-Clark, B. Weeks and L. Wingers, *The simon and speck families of lightweight block ciphers*, Cryptology ePrint Archive, Report 2013/404, `https://eprint.iacr.org/2013/404`, 2013.

[4] M. Schlueter, J. A. Egea and J. Banga, 'Extended ant colony optimization for non-convex mixed integer nonlinear programming,' *Computers & Operations Research*, vol. 36, pp. 2217–2229, Jul. 2009. DOI: `10.1016/j.cor.2008.08.015`.

[5] D. Karaboga and B. Bahriye, 'A powerful and efficient algorithm for numerical function optimization: Artificial bee colony (abc) algorithm,' *Journal of Global Optimization*, vol. 39, no. 3, pp. 459–471, 2007. DOI: `http://dx.doi.org/10.1007/s10898-007-9149-x`.

[6] A. K. Bhateja, A. Bhateja, S. Chaudhury and P. Saxena, 'Cryptanalysis of vigenere cipher using cuckoo search,' *Applied Soft Computing*, vol. 26, pp. 315–324, 2015, ISSN: 1568-4946. DOI: `https://doi.org/10.1016/j.asoc.2014.10.004`. [Online]. Available: `http://www.sciencedirect.com/science/article/pii/S1568494614005031`.

[7] B. Delman, 'Genetic algorithms in cryptography,' 2004.

[8] R. Spillman, M. Janssen, B. Nelson and M. Kepner, 'Use of a genetic algorithm in the cryptanalysis of simple substitution ciphers,' *Cryptologia*, vol. 17, no. 1, pp. 31–44, 1993. DOI: `10.1080/0161-119391867746`. eprint: `https://doi.org/10.1080/0161-119391867746`. [Online]. Available: `https://doi.org/10.1080/0161-119391867746`.

[9] A. J. Clark, 'Optimisation heuristics for cryptology,' Ph.D. dissertation, Queensland University of Technology, 1998.

[10] W. Grundlingh and J. H. Van Vuuren, 'Using genetic algorithms to break a simple cryptographic cipher,' *Retrieved March*, vol. 31, 2003.

[11] M. F. Uddin and A. M. Youssef, 'Cryptanalysis of simple substitution ciphers using particle swarm optimization,' in *2006 IEEE International Conference on Evolutionary Computation*, IEEE, 2006, pp. 677–680.

[12] T. Mekhaznia and M. E. B. Menai, 'Cryptanalysis of classical ciphers with ant algorithms,' *International Journal of Metaheuristics*, vol. 3, no. 3, pp. 175–198, 2014.

[13] H. Grari, A. Azouaoui and K. Zine-Dine, 'A novel ant colony optimization based cryptanalysis of substitution cipher,' in *International Afro-European Conference for Industrial Advancement*, Springer, 2016, pp. 180–187.

[14] A. K. S. Sabonchi and B. Akay, 'Cryptanalysis using artificial bee colony algorithm guided by frequency based fitness value,'

[15] A. Jain and N. S. Chaudhari, 'A new heuristic based on the cuckoo search for cryptanalysis of substitution ciphers,' in *Neural Information Processing*, S. Arik, T. Huang, W. K. Lai and Q. Liu, Eds., Cham: Springer International Publishing, 2015, pp. 206–215, ISBN: 978-3-319-26535-3.

[16] S. Omran, A. Al-Khalid and D. Al-Saady, 'A cryptanalytic attack on vigenère cipher using genetic algorithm,' in *2011 IEEE Conference on Open Systems*, IEEE, 2011, pp. 59–64.

[17] L. Brezočnik, I. Fister and V. Podgorelec, 'Nature-inspired cryptoanalysis methods for breaking vigenère cipher,' in *New Technologies, Development and Application III*, I. Karabegović, Ed., Cham: Springer International Publishing, 2020, pp. 446–453, ISBN: 978-3-030-46817-0.

[18] G. Negara, 'An evolutionary approach for the playfair cipher cryptanalysis,' in *Proceedings of the International Conference on Security and Management (SAM)*, The Steering Committee of The World Congress in Computer Science, Computer …, 2012, p. 1.

[19] M. J. Cowan, 'Breaking short playfair ciphers with the simulated annealing algorithm,' *Cryptologia*, vol. 32, no. 1, pp. 71–83, 2008. DOI: `10.1080/01611190701743658`. eprint: `https://doi.org/10.1080/01611190701743658`. [Online]. Available: `https://doi.org/10.1080/01611190701743658`.

[20] K. P. Bergmann, 'Cryptanalysis using nature-inspired optimization algorithms,' Ph.D. dissertation, University of Calgary, 2007.

[21] T. Mekhaznia, M. Menai and A. Zidani, 'Nature inspired heuristics for cryptanalysis of feistel ciphers,' *ICIST13*, pp. 48–55, 2013.

[22] N. Nalini and G. R. Rao, 'Cryptanalysis of simplified data encryption standard via optimization heuristics,' in *2005 3rd International Conference on Intelligent Sensing and Information Processing*, IEEE, 2005, pp. 74–79.

[23] L. Sharma, B. K. Pathak and N. Sharma, 'Breaking of simplified data encryption standard using binary particle swarm optimization,' 2012.

[24] S. Khan, W. Shahzad and F. A. Khan, 'Cryptanalysis of four-rounded des using ant colony optimization,' in *2010 International Conference on Information Science and Applications*, 2010, pp. 1–7. DOI: `10.1109/ICISA.2010.5480260`.

[25] S. Khan, A. Ali and M. Y. Durrani, 'Ant-crypto, a cryptographer for data encryption standard,' *International Journal of Computer Science Issues (IJCSI)*, vol. 10, no. 1, p. 400, 2013.

[26] H. Grari, A. Azouaoui and K. Zine-Dine, 'A cryptanalytic attack of simplified-aes using ant colony optimization,' English, *International Journal of Electrical and Computer Engineering*, vol. 9, no. 5, pp. 4287–4295, Oct. 2019, Copyright - Copyright IAES Institute of Advanced Engineering and Science Oct 2019; Last updated - 2020-04-19. [Online]. Available: `https://www.proquest.com/docview/2391254121?accountid=12870`.

[27] E. Rutkowski and S. Houghten, 'Cryptanalysis of rsa: Integer prime factorization using genetic algorithms,' in *2020 IEEE Congress on Evolutionary Computation (CEC)*, IEEE, 2020, pp. 1–8.

[28] M. Mishra, U. Chaturvedi and S. K. Pal, 'A multithreaded bound varying chaotic firefly algorithm for prime factorization,' in *2014 IEEE International Advance Computing Conference (IACC)*, 2014, pp. 1322–1325. DOI: `10.1109/IAdCC.2014.6779518`.

[29] F. Biscani and D. Izzo, 'A parallel global multiobjective framework for optimization: Pagmo,' *Journal of Open Source Software*, vol. 5, no. 53, p. 2338, 2020. DOI: `10.21105/joss.02338`. [Online]. Available: `https://doi.org/10.21105/joss.02338`.

[30] A. Corana, M. Marchesi, C. Martini and S. Ridella, 'Minimizing multimodal functions of continuous variables with the "simulated annealing" algorithm—corrigenda for this article is available here,' *ACM Transactions on Mathematical Software (TOMS)*, vol. 13, no. 3, pp. 262–280, 1987.

[31] M. Mernik, S.-H. Liu, D. Karaboga and M. Črepinšek, 'On clarifying misconceptions when comparing variants of the artificial bee colony algorithm by offering a new implementation,' *Information Sciences*, vol. 291, pp. 115–127, 2015, ISSN: 0020-0255. DOI: `https://doi.org/10.1016/j.ins.2014.08.040`. [Online]. Available: `https://www.sciencedirect.com/science/article/pii/S0020025514008378`.

[32] J. Verne, *Twenty thousand leagues under the sea*. Oxford University Press, 1998.

Appendix A

Key Recovery Plots

Below you will find box plots showing the key recovery of each optimization algorithm on each cipher, except RSA. We have chosen to draw the box plots with the upper and lower whiskers respectively representing maximum and minimum values in the data, and not plotting outlying values. As usual, the boxes represent the 25% and 75% quartile of the data, with the median value between. For some ciphers, we have used more than one ciphertext in our experiments. For brevity, we have chosen to only include box plots for some of these ciphertexts:

- **Columnar cipher**: JulesVerne1.txt
- **Caesar cipher**: JulesVerne4.txt
- **Monoalphabetic substitution cipher**: JulesVerne1.txt
- **Vigènere cipher**: JulesVerne1_key3.txt, JulesVerne1_key5.txt and JulesVerne1_key10.txt
- **Playfair cipher**: JulesVerne1.txt
- **DES**: JulesVerne1_des.txt
- **AES**: JulesVerne1_aes.txt
- **SPECK**: JulesVerne1_simon.txt

For RSA, we have instead chosen to plot key recovery as the percentage of runs that successfully factorized *n* in a line chart. Results from all 16 datasets are given. The x-axis shows the amount of bits in the *n* to be factorized.

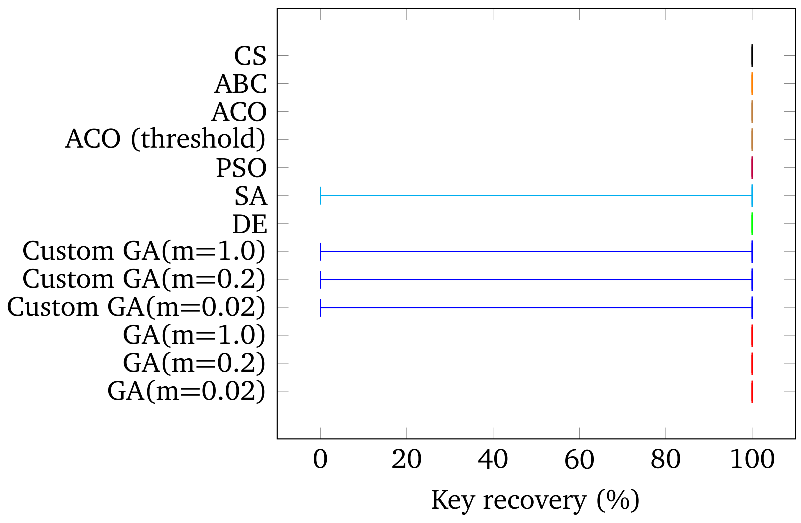

Figure A.1: Caesar cipher - Population size = 20

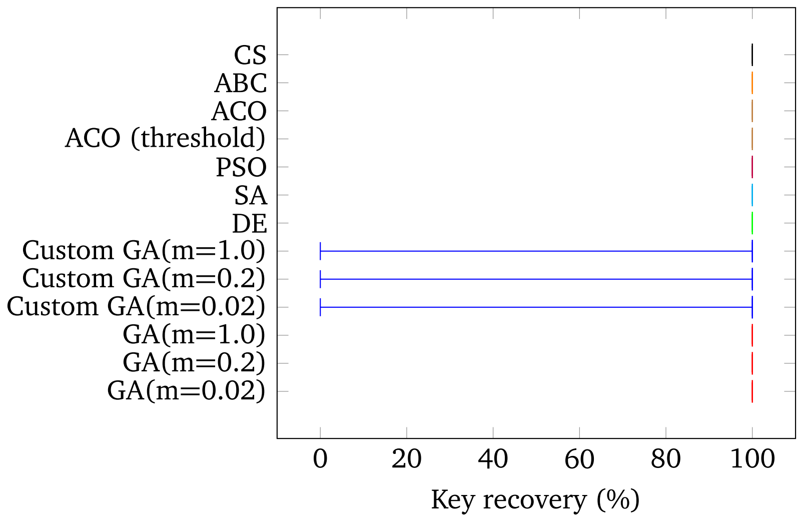

Figure A.2: Caesar cipher - Population size = 30

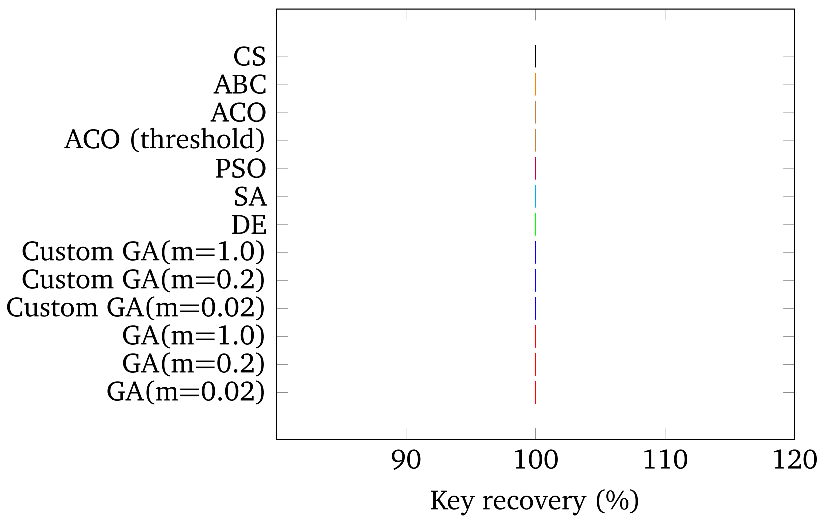

Figure A.3: Caesar cipher - Population size = 50

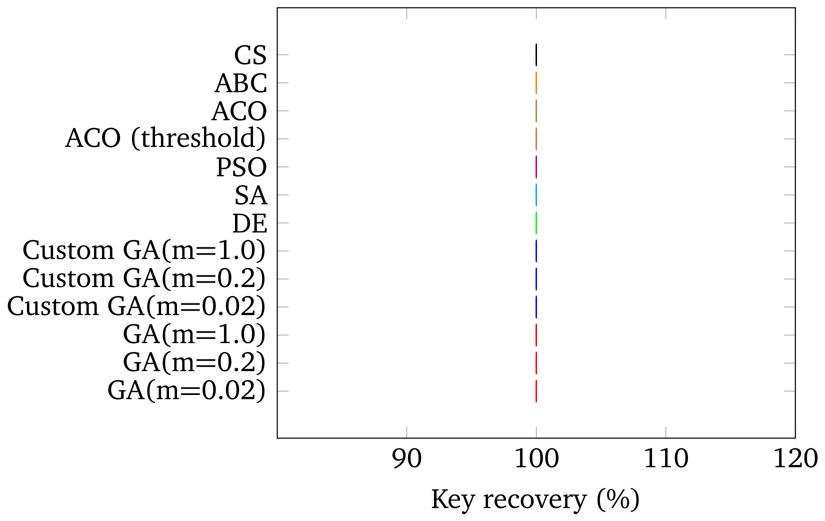

Figure A.4: Caesar cipher - Population size = 70

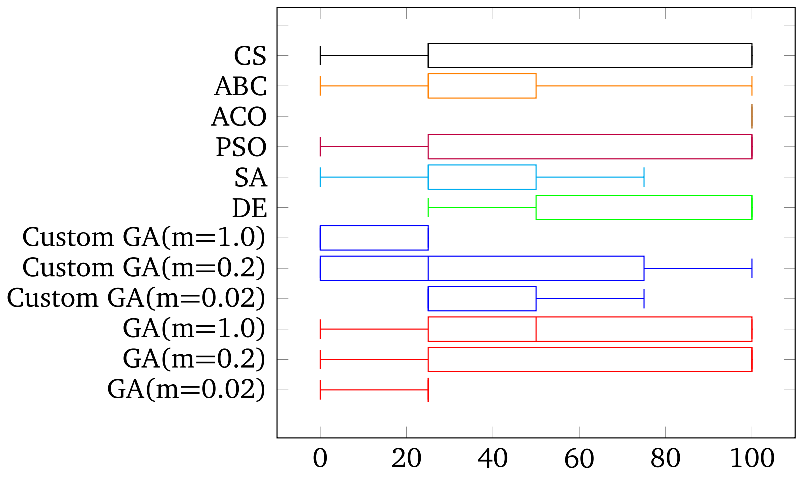

Figure A.5: Columnar Cipher - Population size = 20

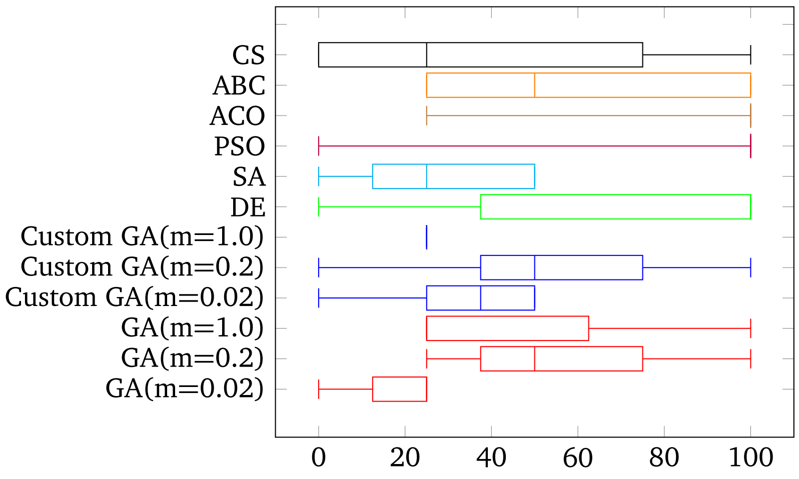

Figure A.6: Columnar Cipher - Population size = 30

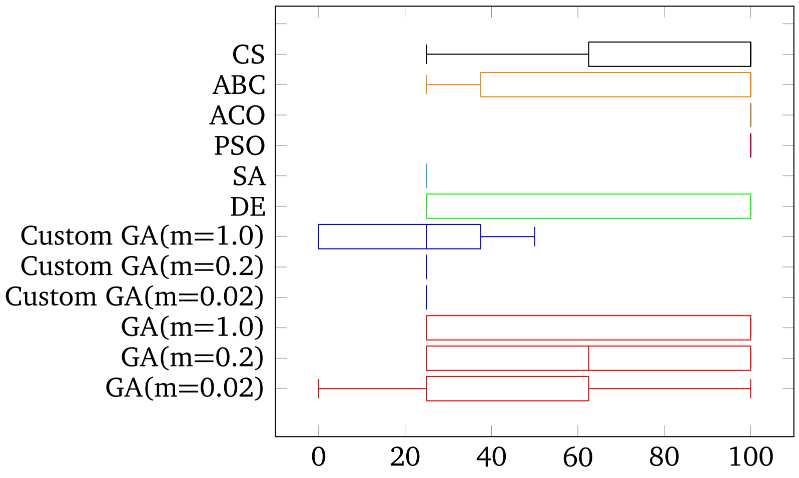

Figure A.7: Columnar Cipher - Population size = 50

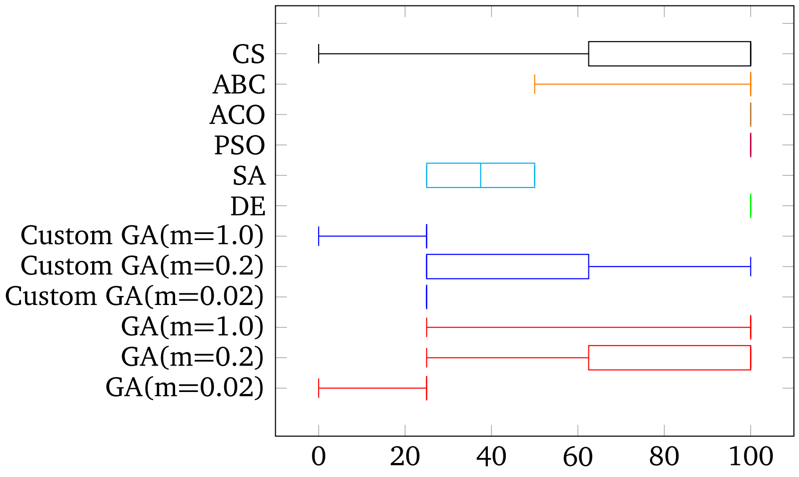

Figure A.8: Columnar Cipher - Population size = 70

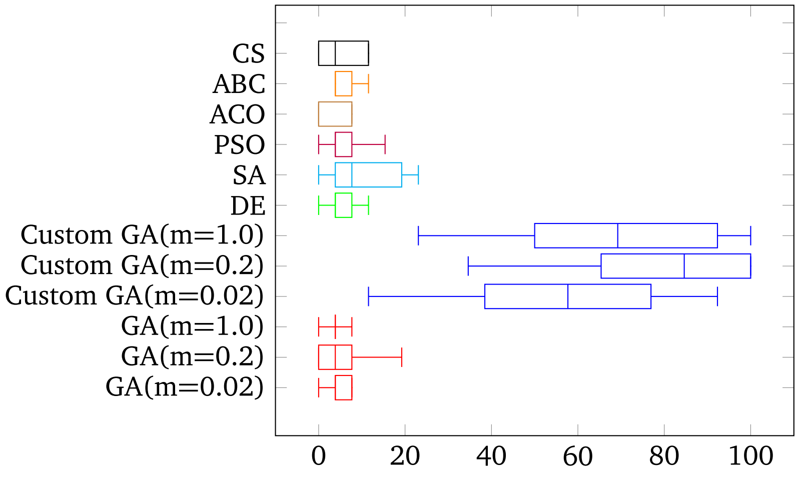

Figure A.9: Monoalphabetic Substitution cipher - Population size = 20

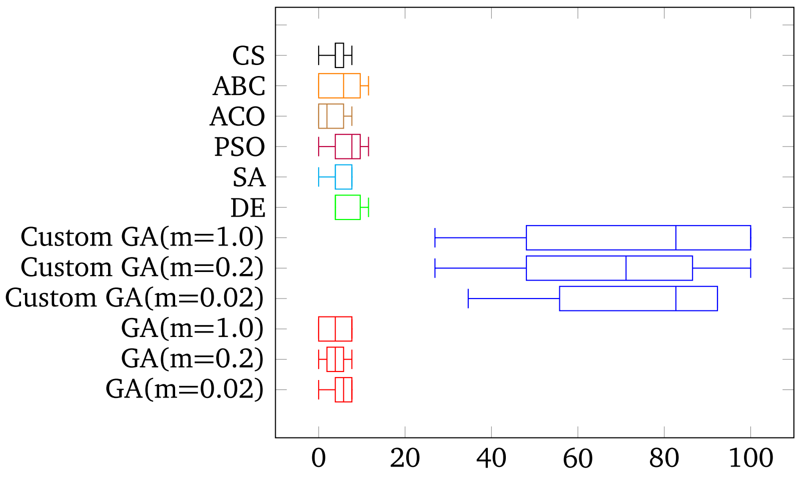

Figure A.10: Monoalphabetic Substitution cipher - Population size = 30

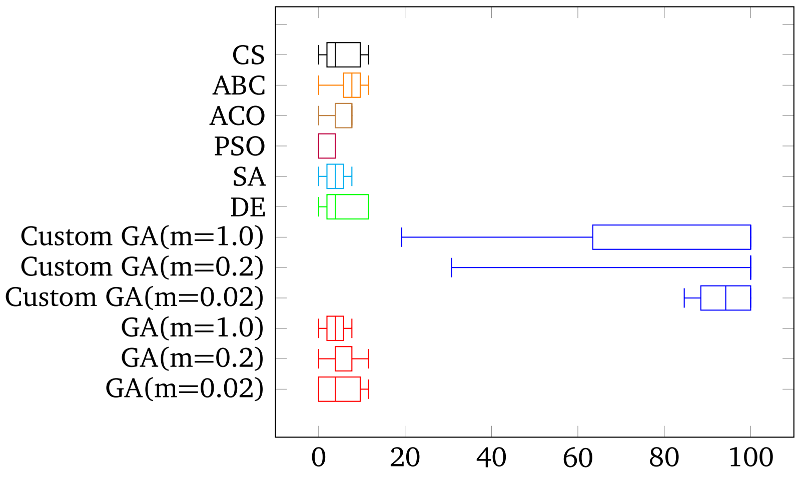

Figure A.11: Monoalphabetic Substitution cipher - Population size = 50

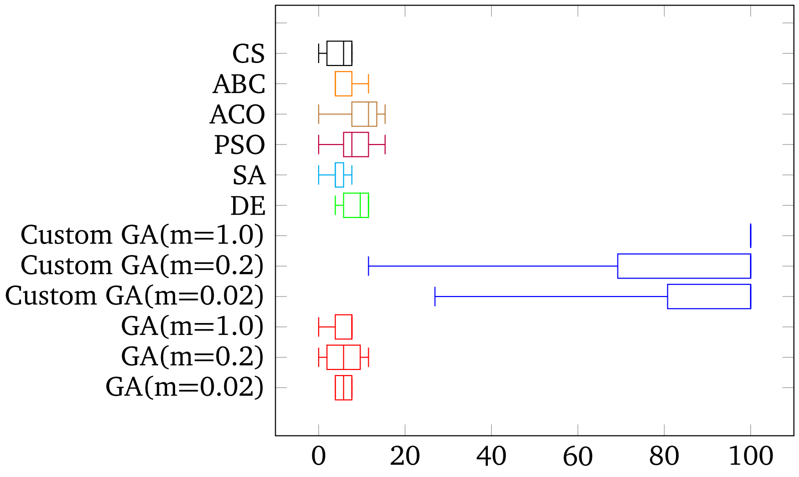

Figure A.12: Monoalphabetic Substitution cipher - Population size = 70

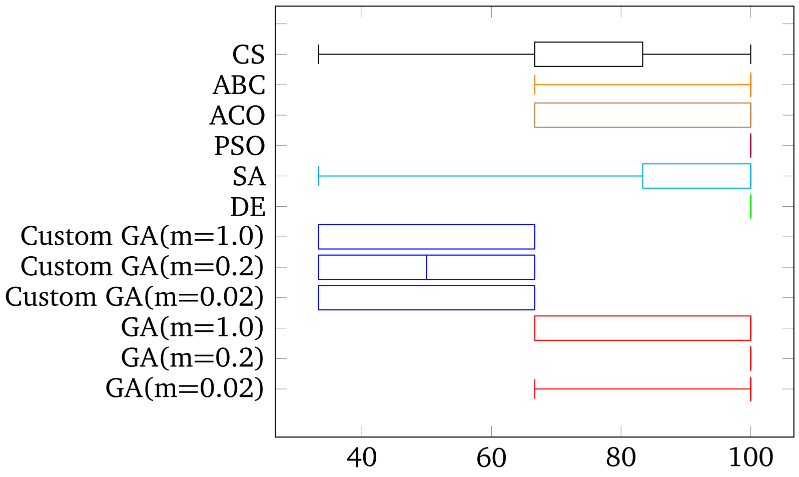

Figure A.13: Vigènere with key length 3 - Population size = 20

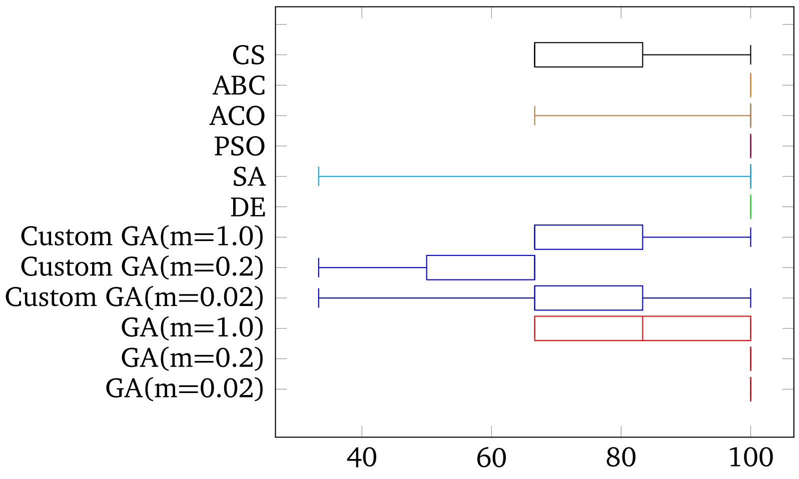

Figure A.14: Vigènere with key length 3 - Population size = 30

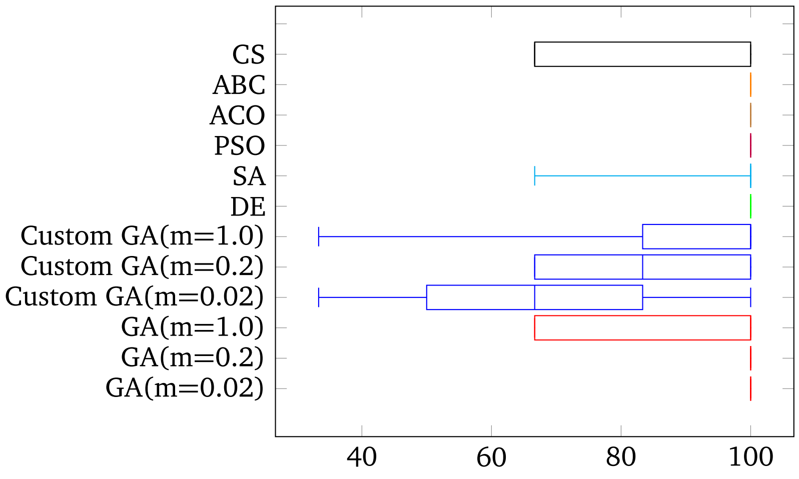

Figure A.15: Vigènere with key length 3 - Population size = 50

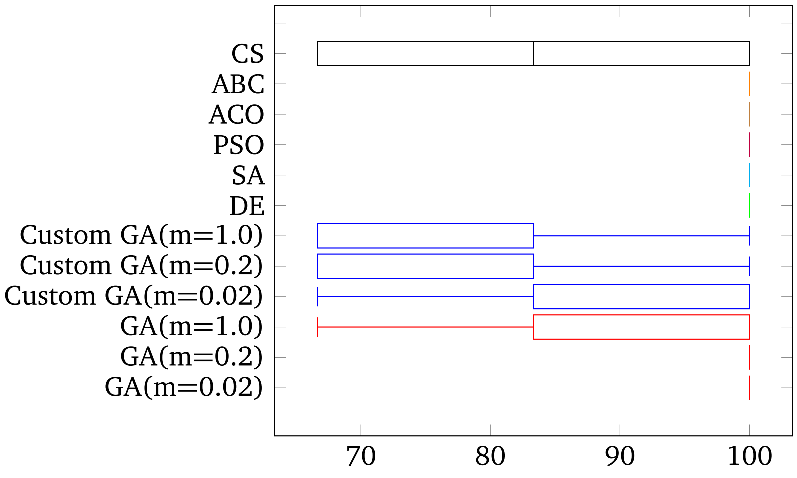

Figure A.16: Vigènere with key length 3 - Population size = 70

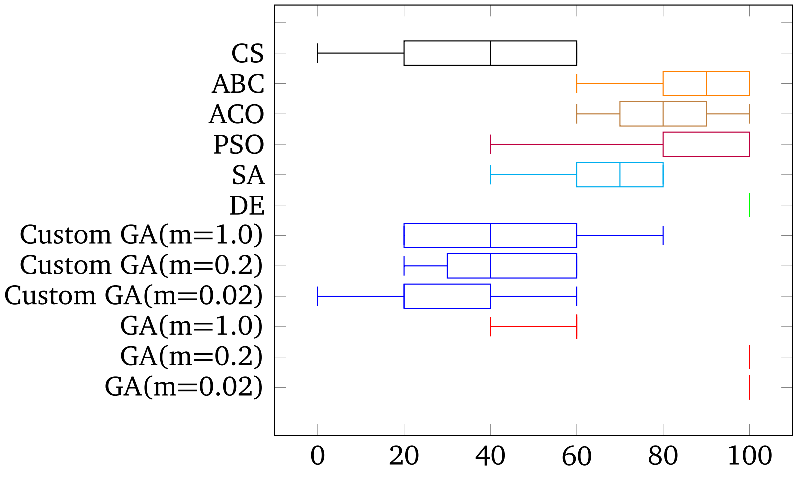

Figure A.17: Vigènere with key length 5 - Population size = 20

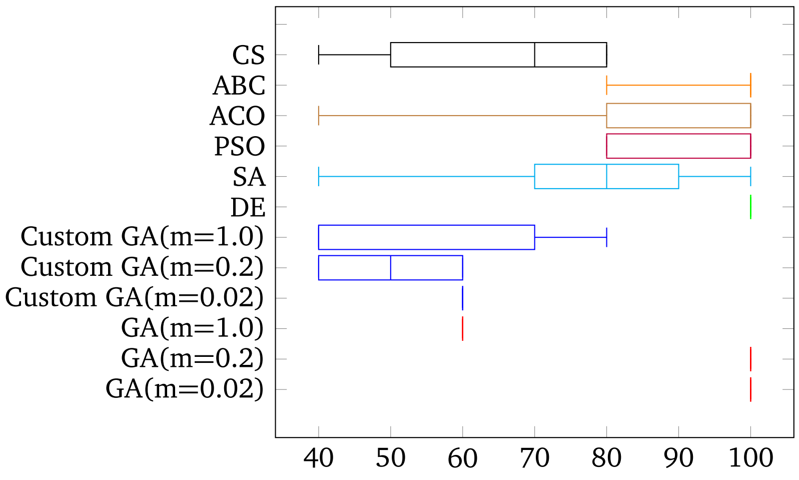

Figure A.18: Vigènere with key length 5 - Population size = 30

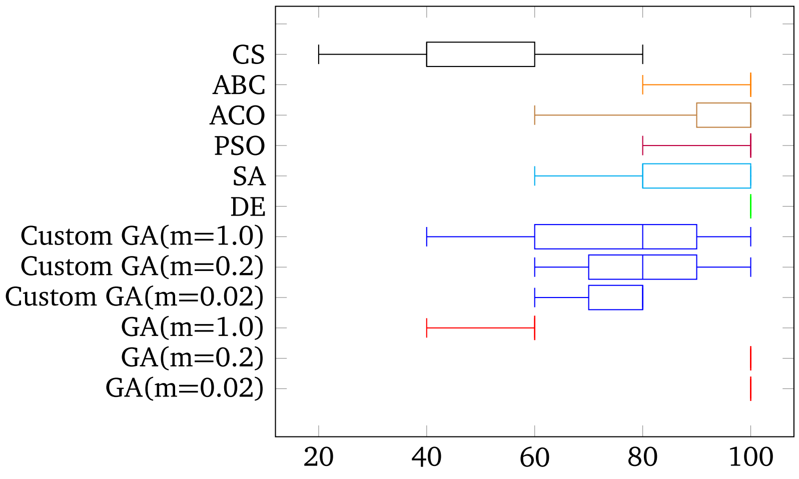

Figure A.19: Vigènere with key length 5 - Population size = 50

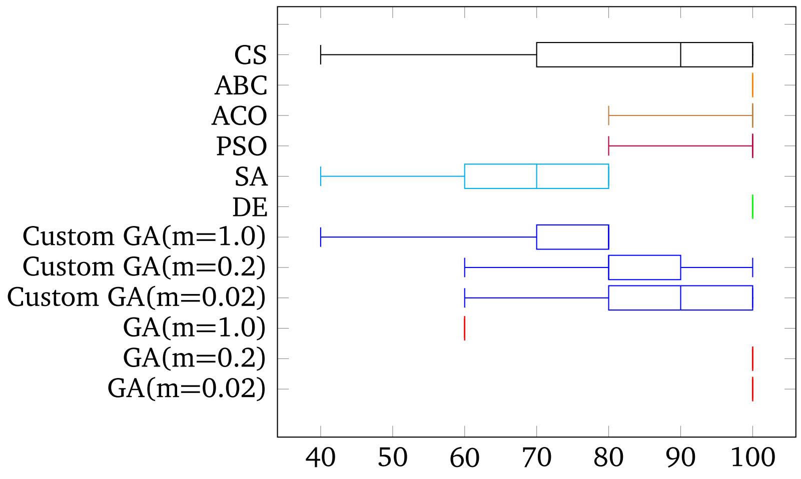

Figure A.20: Vigènere with key length 5 - Population size = 70

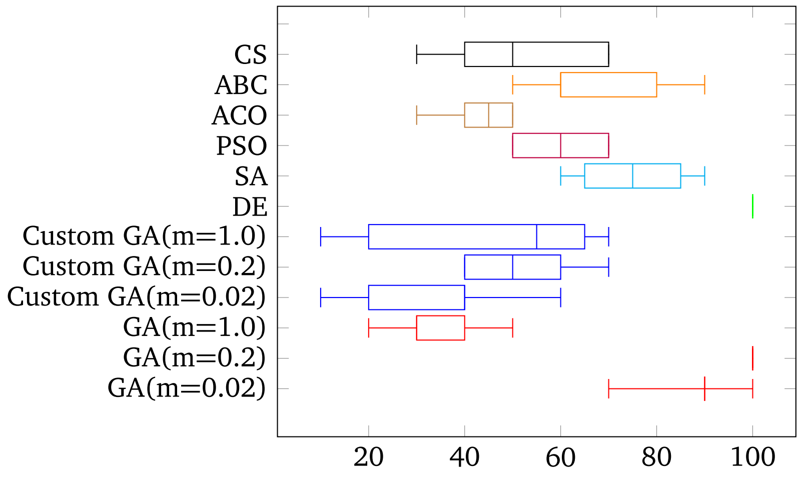

Figure A.21: Vigènere with key length 10 - Population size = 20

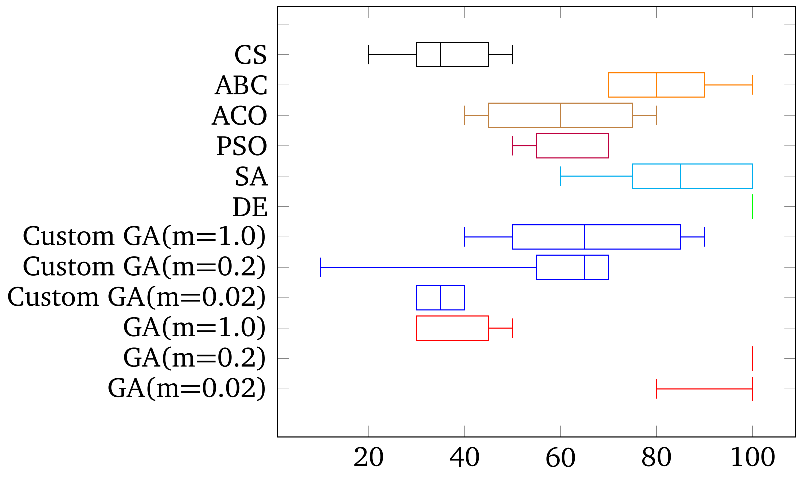

Figure A.22: Vigènere with key length 10 - Population size = 30

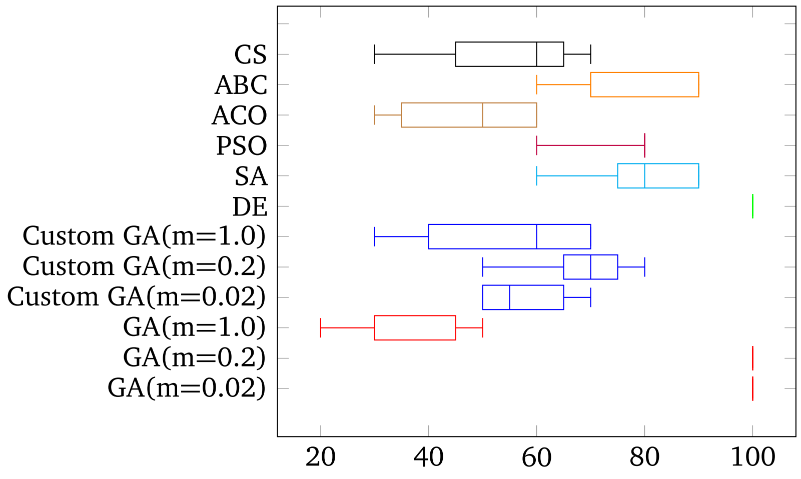

Figure A.23: Vigènere with key length 10 - Population size = 50

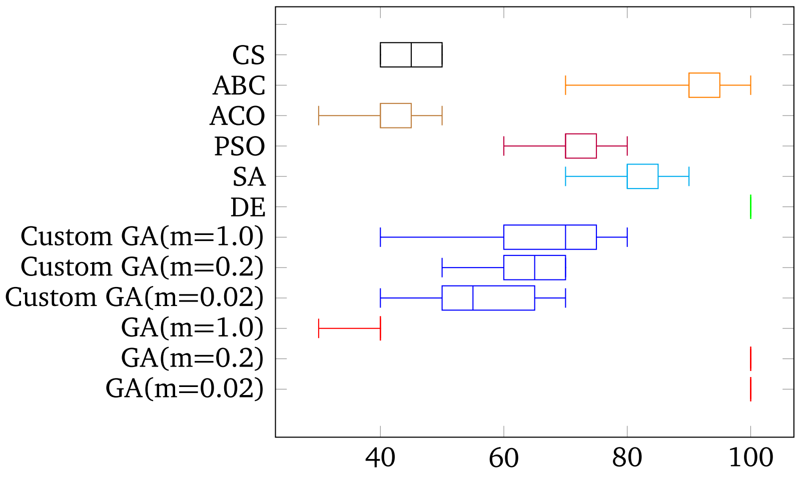

Figure A.24: Vigènere with key length 10 - Population size = 70

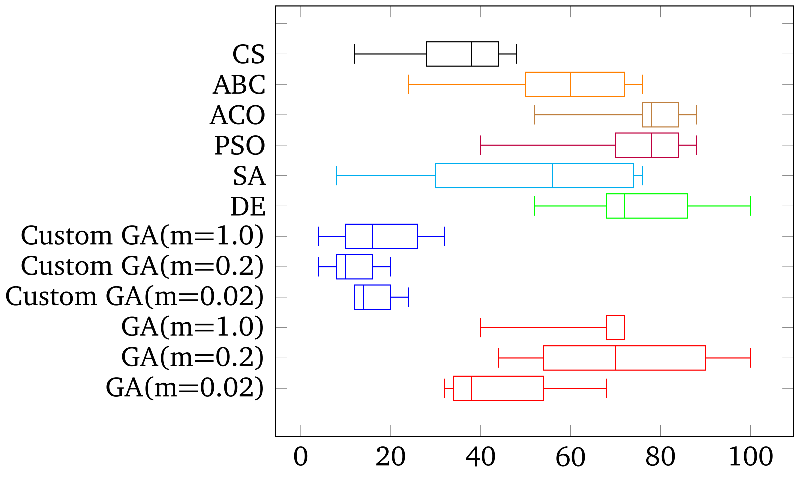

Figure A.25: Playfair cipher - Population size = 20

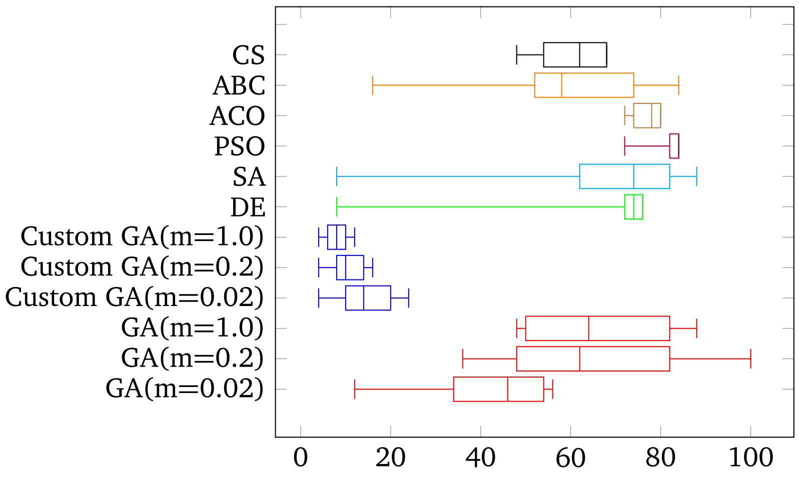

Figure A.26: Playfair cipher - Population size = 30

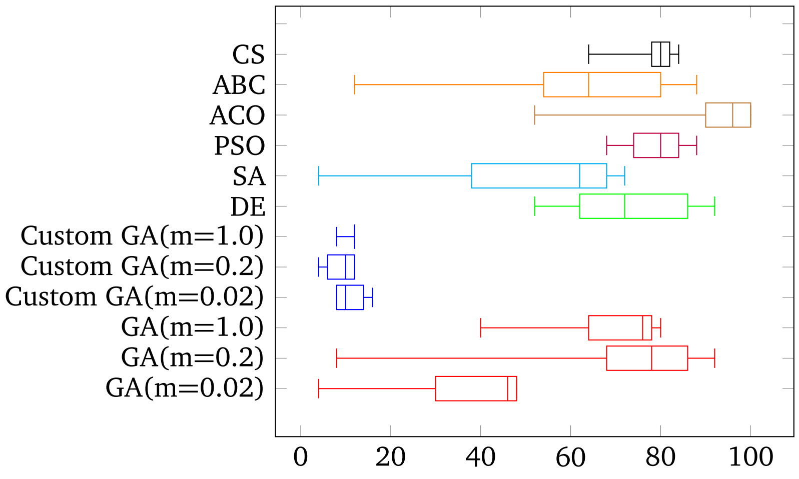

Figure A.27: Playfair cipher - Population size = 50

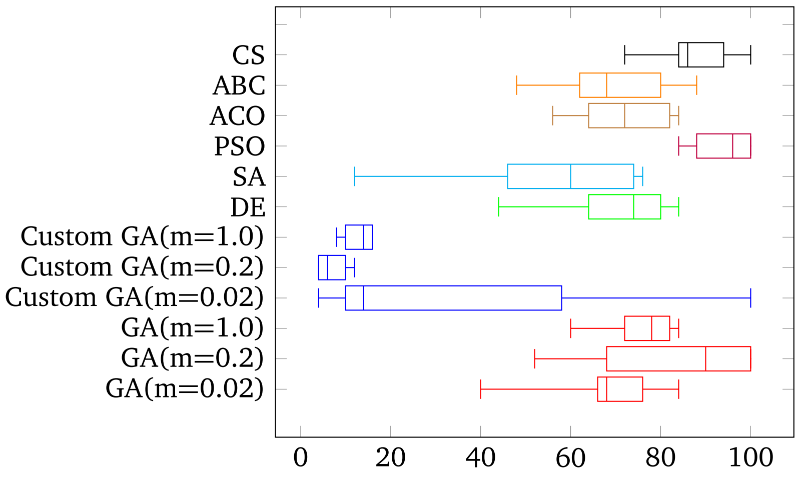

Figure A.28: Playfair cipher - Population size = 70

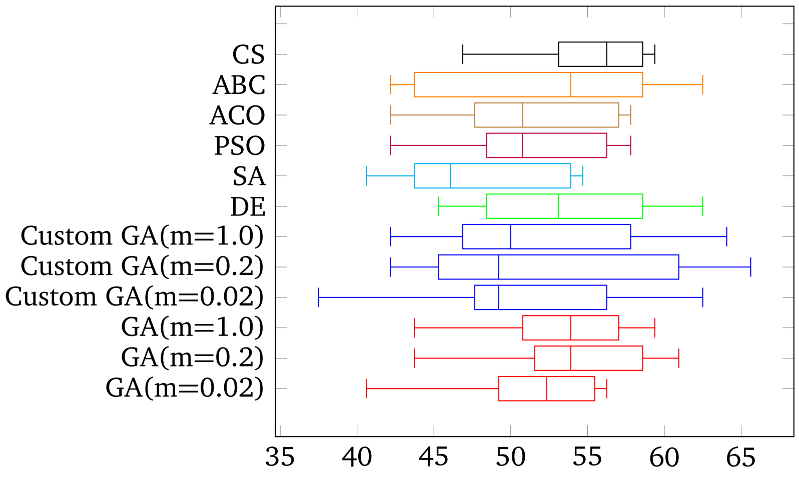

Figure A.29: DES - Population size = 20

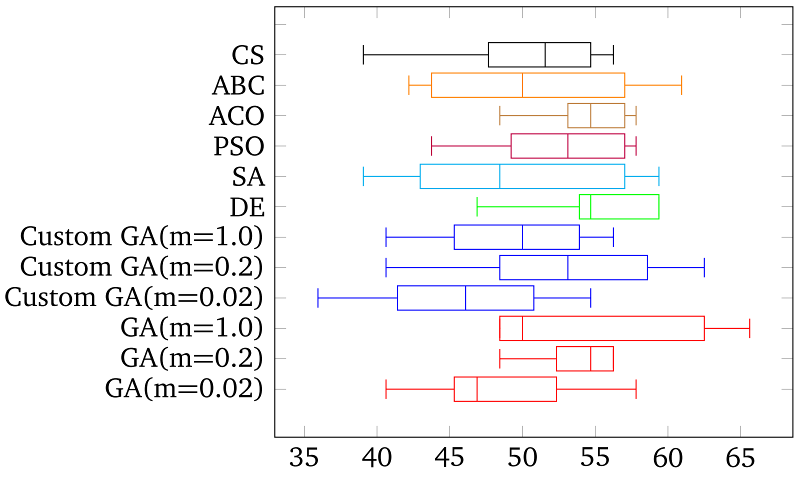

Figure A.30: DES - Population size = 30

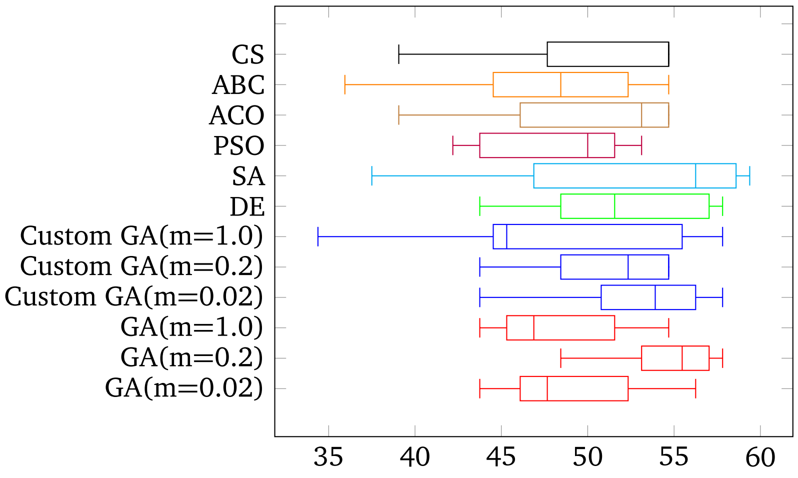

Figure A.31: DES - Population size = 50

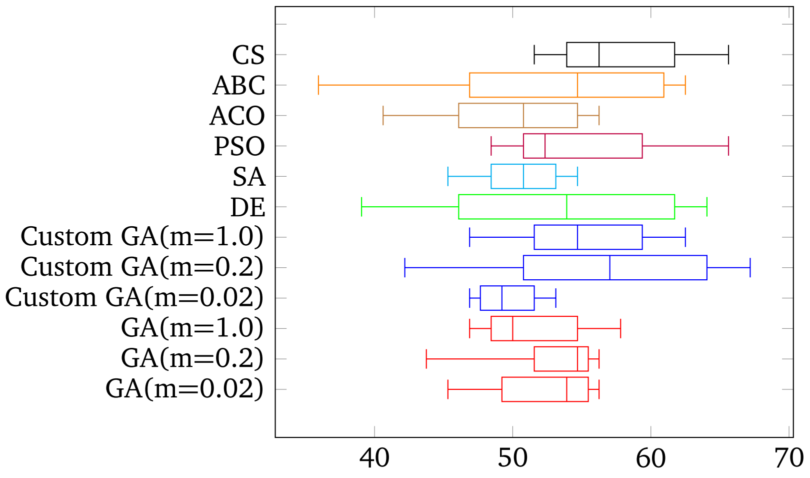

Figure A.32: DES - Population size = 70

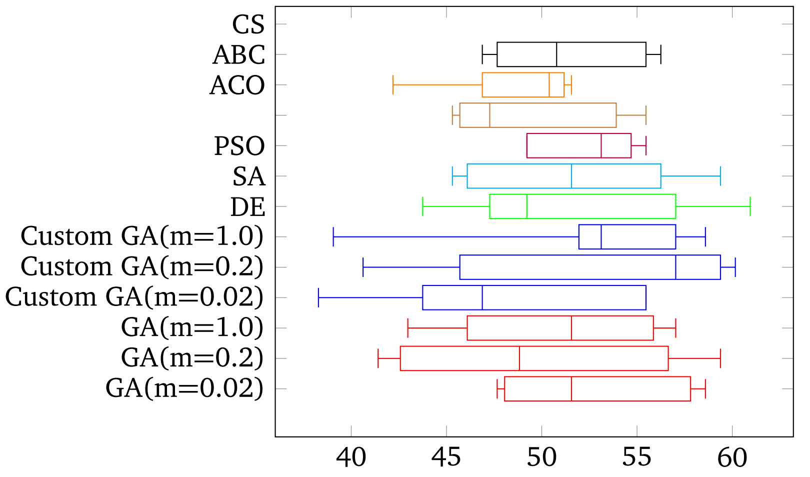

Figure A.33: AES - Population size = 20

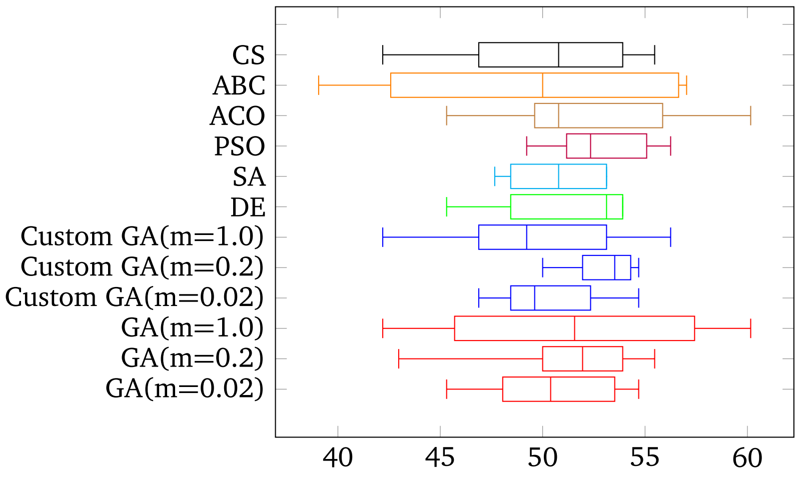

Figure A.34: AES - Population size = 30

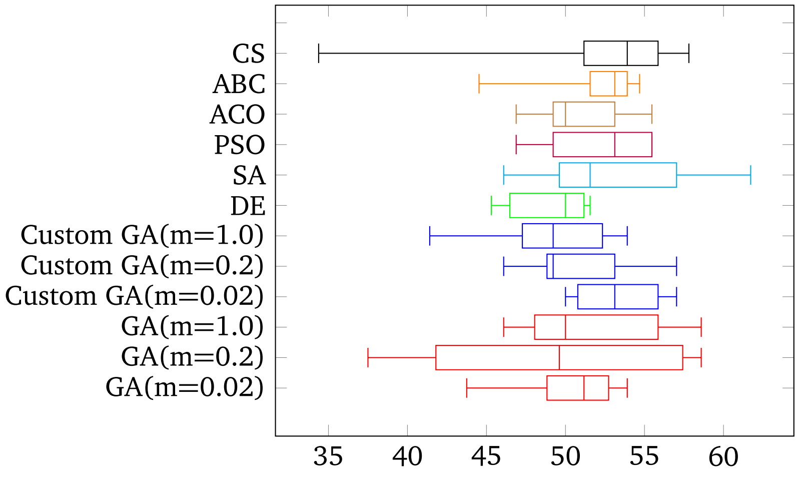

Figure A.35: AES - Population size = 50

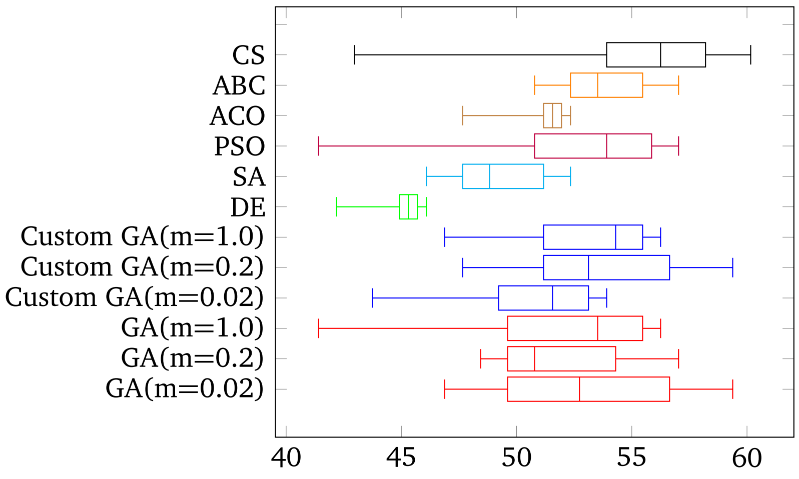

Figure A.36: AES - Population size = 70

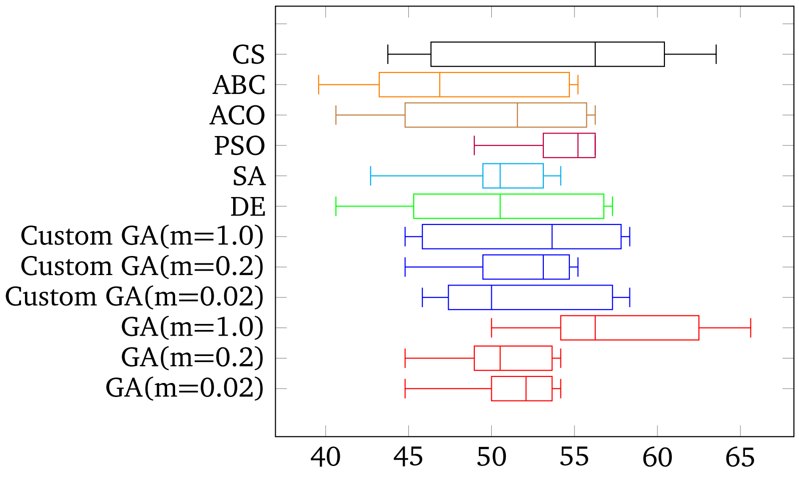

Figure A.37: Speck - Population 20

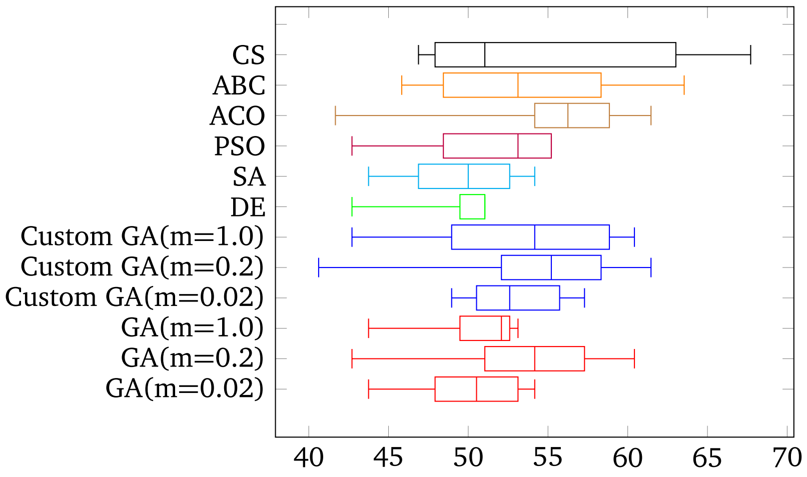

Figure A.38: Speck - Population 30

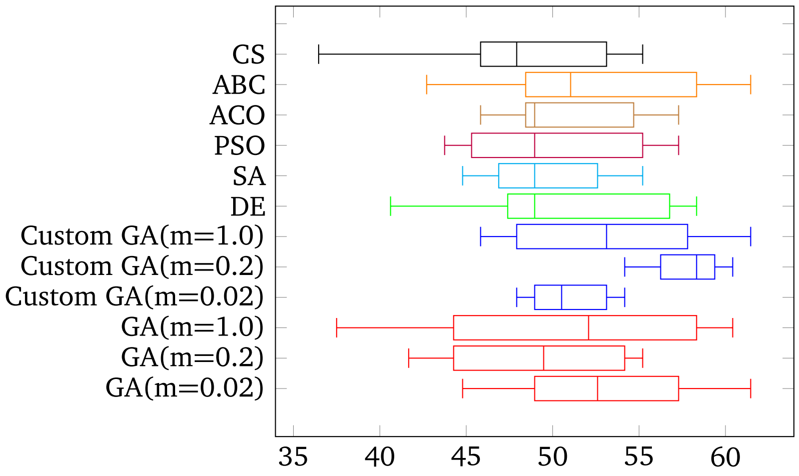

Figure A.39: Speck - Population 50

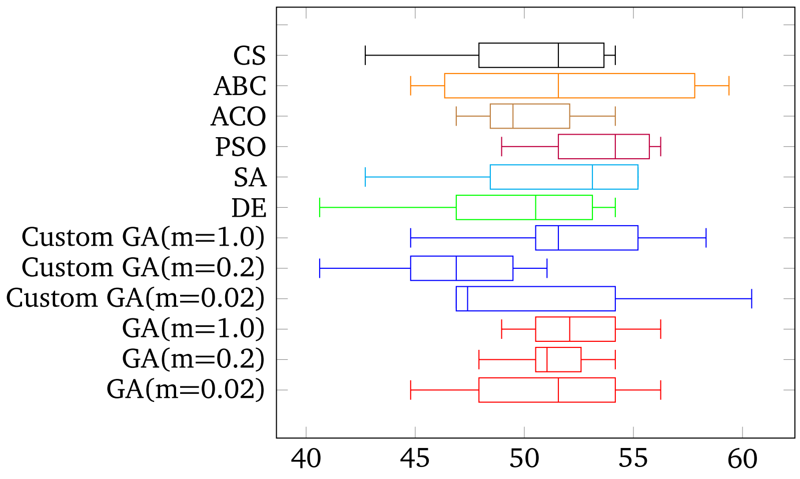

Figure A.40: Speck - Population 70

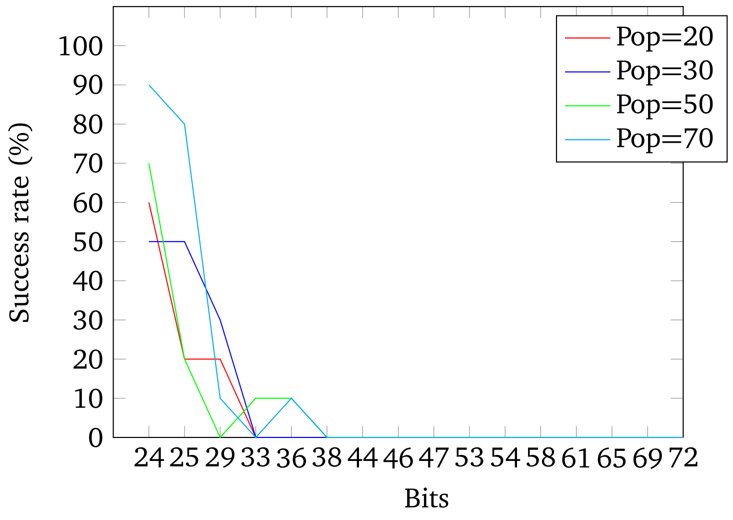

Figure A.41: RSA key recovery - Simple Genetic Algorithm (m=0.02)

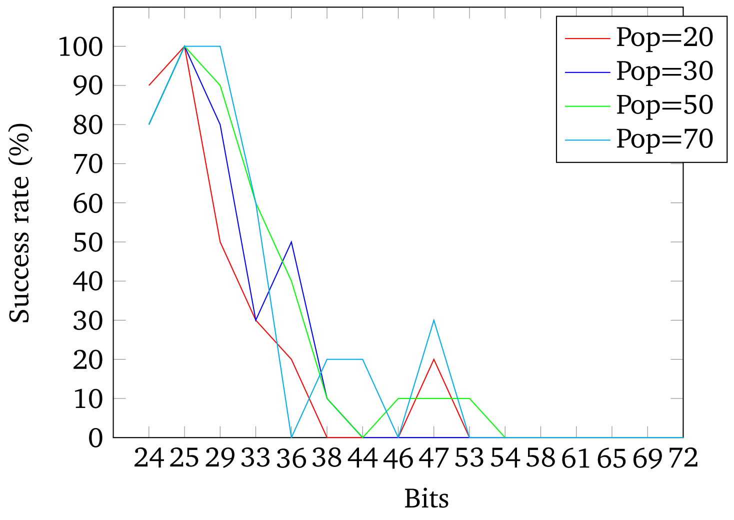

Figure A.42: RSA key recovery - Simple Genetic Algorithm (m=0.2)

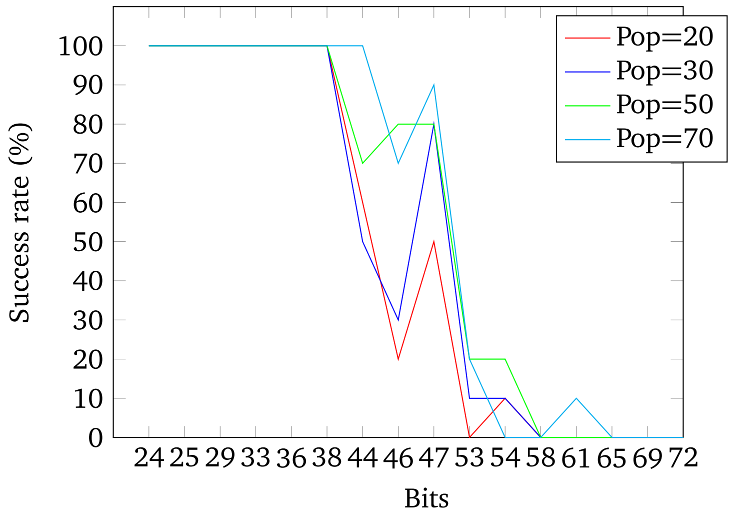

Figure A.43: RSA key recovery - Simple Genetic Algorithm (m=1.0)

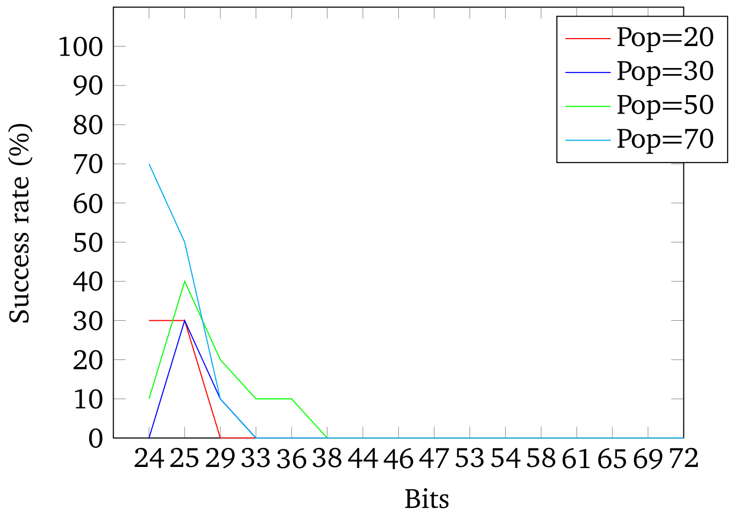

Figure A.44: RSA key recovery - Custom Genetic Algorithm (m=0.02)

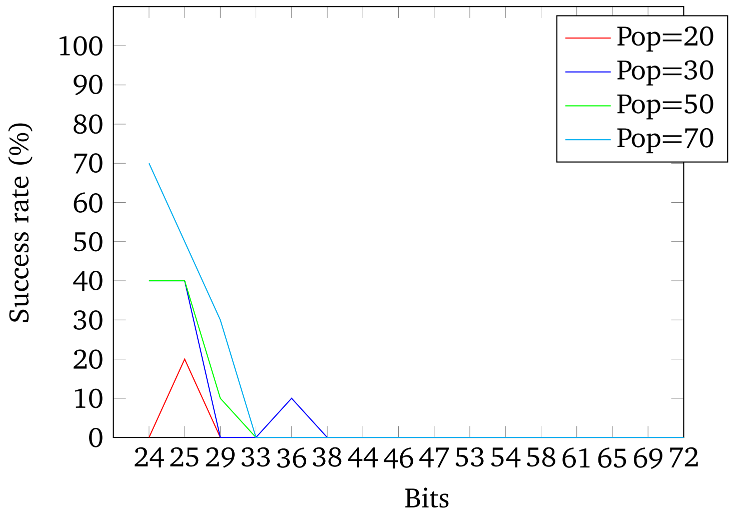

Figure A.45: RSA key recovery - Custom Genetic Algorithm (m=0.2)

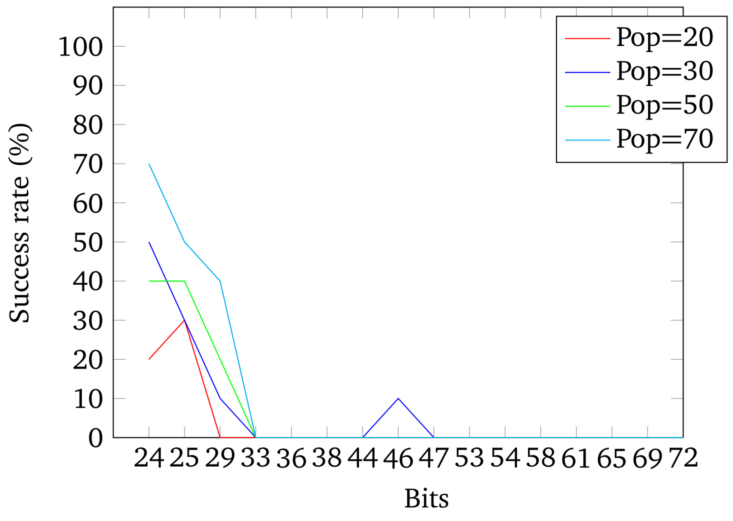

Figure A.46: RSA key recovery - Custom Genetic Algorithm (m=1.0)

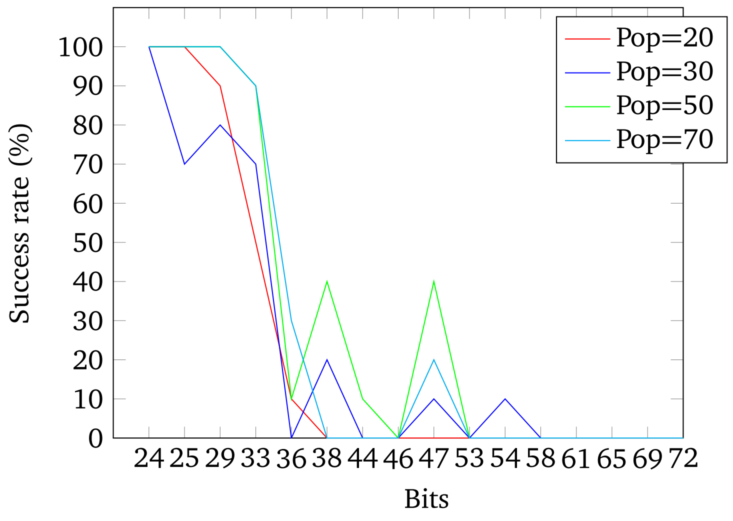

Figure A.47: RSA key recovery - Self-Adaptive Differential Evolution

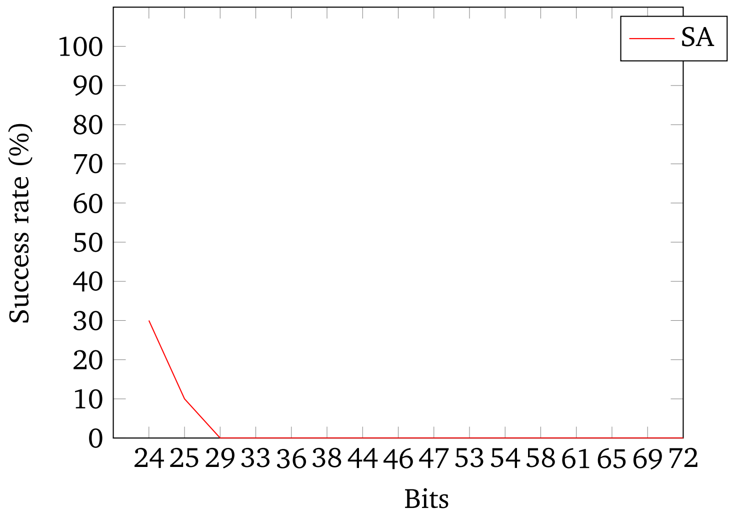

Figure A.48: RSA key recovery - Simulated Annealing

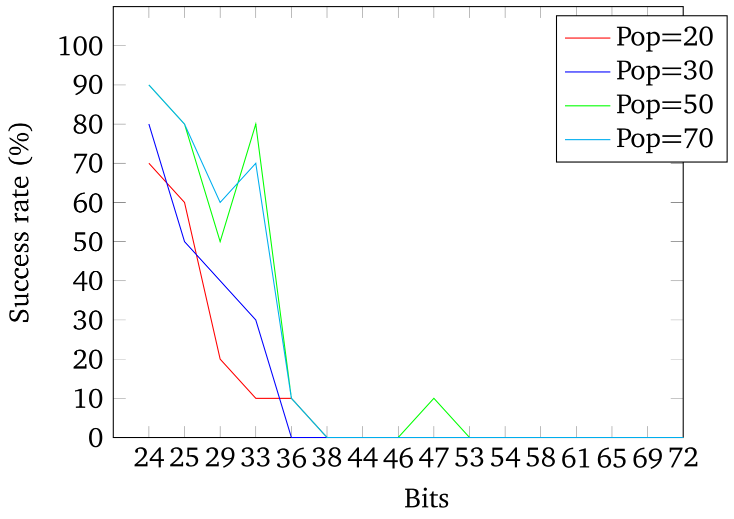

Figure A.49: RSA key recovery - Particle Swarm Optimization

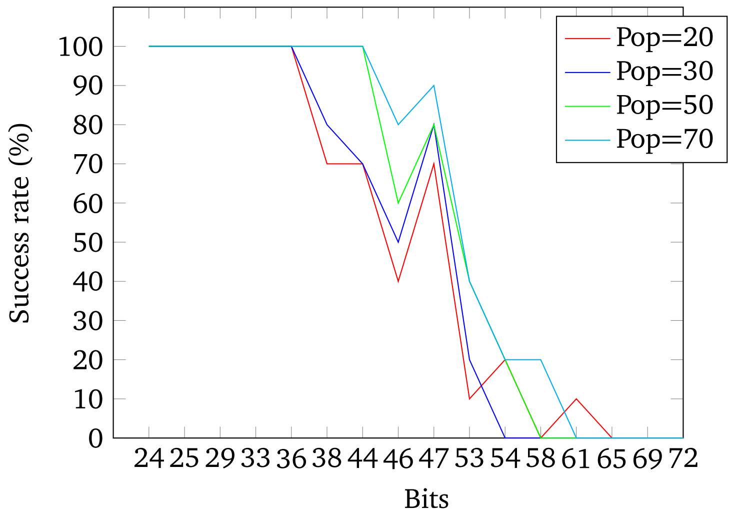

Figure A.50: RSA key recovery - Ant Colony Optimization

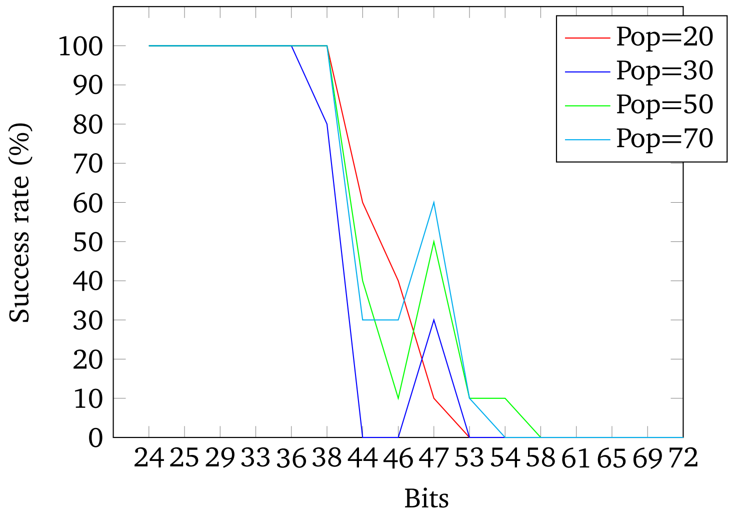

Figure A.51: RSA key recovery - Artificial Bee Colony Optimization

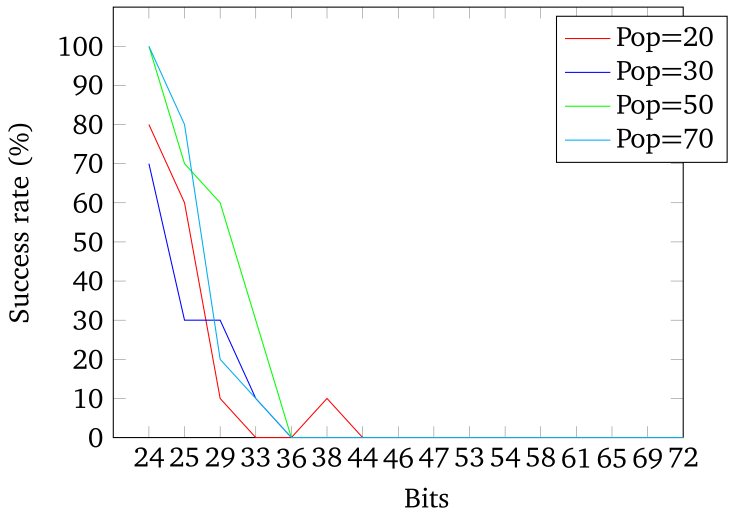

Figure A.52: RSA key recovery - Cuckoo Search

Appendix B

Result Tables

Below you will find tables with an overview of each experiment performed with each cipher and optimization algorithm. Each optimization algorithm is run five times, or ten times in the case of RSA, for a maximum of 200 generations on each ciphertext for each population size in 20, 30, 50 and 70. Key recovery is recorded for each run, and an average and maximum recovery is given. The number of runs that achieved a complete key recovery is given in the table, as well as the average number of generations and fitness evaluations it took for these runs to achieve complete key recovery. As Simulated Annealing is not a population-based optimization algorithm, the population size parameter will only increase the likelihood of a good solution in the initial population, and not affect the algorithm in any other way. For this reason, only the results from the five runs using population size 20 is given. For the RSA factorization problem, there is no partial key recovery, which makes the columns for average and maximum key recovery unnecessary. Instead, we will only give the number of successful runs out of ten. For brevity, all experiments that yielded no successful runs have been excluded from the table.

Table B.1: Caesar cipher - Simple Genetic Algorithm (m=0.02)

Ciphertext	Pop.	Avg. rec.	Max. rec.	100% rec.	Avg. gen.	Avg. eval.
JulesVerne4.txt	20	100.00%	100.00%	5/5	1.00	20.00
JulesVerne4.txt	30	100.00%	100.00%	5/5	12.80	384.00
JulesVerne4.txt	50	100.00%	100.00%	5/5	1.00	50.00
JulesVerne4.txt	70	100.00%	100.00%	5/5	13.00	910.00
caesar.txt	20	100.00%	100.00%	5/5	27.80	556.00
caesar.txt	30	100.00%	100.00%	5/5	10.40	312.00
caesar.txt	50	100.00%	100.00%	5/5	1.00	50.00
caesar.txt	70	100.00%	100.00%	5/5	2.20	154.00
caesar2.txt	20	100.00%	100.00%	5/5	33.20	664.00
caesar2.txt	30	100.00%	100.00%	5/5	18.80	564.00
caesar2.txt	50	100.00%	100.00%	5/5	10.80	540.00
caesar2.txt	70	100.00%	100.00%	5/5	18.40	1288.00

Table B.2: Caesar cipher - Simple Genetic Algorithm (m=0.2)

Ciphertext	Pop.	Avg. rec.	Max. rec.	100% rec.	Avg. gen.	Avg. eval.
JulesVerne4.txt	20	100.00%	100.00%	5/5	1.60	32.00
JulesVerne4.txt	30	100.00%	100.00%	5/5	8.20	246.00
JulesVerne4.txt	50	100.00%	100.00%	5/5	1.60	80.00
JulesVerne4.txt	70	100.00%	100.00%	5/5	1.40	98.00
caesar.txt	20	100.00%	100.00%	5/5	8.40	168.00
caesar.txt	30	100.00%	100.00%	5/5	3.00	90.00
caesar.txt	50	100.00%	100.00%	5/5	2.60	130.00
caesar.txt	70	100.00%	100.00%	5/5	1.00	70.00
caesar2.txt	20	100.00%	100.00%	5/5	4.00	80.00
caesar2.txt	30	100.00%	100.00%	5/5	3.80	114.00
caesar2.txt	50	100.00%	100.00%	5/5	1.20	60.00
caesar2.txt	70	100.00%	100.00%	5/5	1.00	70.00

Table B.3: Caesar cipher - Simple Genetic Algorithm (m=1.0)

Ciphertext	Pop.	Avg. rec.	Max. rec.	100% rec.	Avg. gen.	Avg. eval.
JulesVerne4.txt	20	100.00%	100.00%	5/5	1.00	20.00
JulesVerne4.txt	30	100.00%	100.00%	5/5	2.20	66.00
JulesVerne4.txt	50	100.00%	100.00%	5/5	1.20	60.00
JulesVerne4.txt	70	100.00%	100.00%	5/5	1.00	70.00
caesar.txt	20	100.00%	100.00%	5/5	1.60	32.00
caesar.txt	30	100.00%	100.00%	5/5	1.80	54.00
caesar.txt	50	100.00%	100.00%	5/5	1.00	50.00
caesar.txt	70	100.00%	100.00%	5/5	1.00	70.00
caesar2.txt	20	100.00%	100.00%	5/5	3.40	68.00
caesar2.txt	30	100.00%	100.00%	5/5	1.00	30.00
caesar2.txt	50	100.00%	100.00%	5/5	1.00	50.00
caesar2.txt	70	100.00%	100.00%	5/5	1.00	70.00

Table B.4: Caesar cipher - Custom Genetic Algorithm (m=0.02)

Ciphertext	Pop.	Avg. rec.	Max. rec.	100% rec.	Avg. gen.	Avg. eval.
JulesVerne4.txt	20	80.00%	100.00%	4/5	1.00	20.00
JulesVerne4.txt	30	80.00%	100.00%	4/5	1.00	30.00
JulesVerne4.txt	50	100.00%	100.00%	5/5	1.00	50.00
JulesVerne4.txt	70	100.00%	100.00%	5/5	1.00	70.00
caesar.txt	20	20.00%	100.00%	1/5	1.00	20.00
caesar.txt	30	80.00%	100.00%	4/5	1.00	30.00
caesar.txt	50	40.00%	100.00%	2/5	1.00	50.00
caesar.txt	70	100.00%	100.00%	5/5	1.00	70.00
caesar2.txt	20	40.00%	100.00%	2/5	1.00	20.00
caesar2.txt	30	60.00%	100.00%	3/5	1.00	30.00
caesar2.txt	50	80.00%	100.00%	4/5	1.00	50.00
caesar2.txt	70	100.00%	100.00%	5/5	1.00	70.00

Table B.5: Caesar cipher - Custom Genetic Algorithm (m=0.2)

Ciphertext	Pop.	Avg. rec.	Max. rec.	100% rec.	Avg. gen.	Avg. eval.
JulesVerne4.txt	20	60.00%	100.00%	3/5	1.00	20.00
JulesVerne4.txt	30	80.00%	100.00%	4/5	1.00	30.00
JulesVerne4.txt	50	100.00%	100.00%	5/5	1.00	50.00
JulesVerne4.txt	70	100.00%	100.00%	5/5	1.00	70.00
caesar.txt	20	40.00%	100.00%	2/5	1.00	20.00
caesar.txt	30	60.00%	100.00%	3/5	1.00	30.00
caesar.txt	50	80.00%	100.00%	4/5	1.00	50.00
caesar.txt	70	80.00%	100.00%	4/5	1.00	70.00
caesar2.txt	20	40.00%	100.00%	2/5	1.00	20.00
caesar2.txt	30	40.00%	100.00%	2/5	1.00	30.00
caesar2.txt	50	100.00%	100.00%	5/5	1.00	50.00
caesar2.txt	70	100.00%	100.00%	5/5	1.00	70.00

Table B.6: Caesar cipher - Custom Genetic Algorithm (m=1.0)

Ciphertext	Pop.	Avg. rec.	Max. rec.	100% rec.	Avg. gen.	Avg. eval.
JulesVerne4.txt	20	80.00%	100.00%	4/5	1.00	20.00
JulesVerne4.txt	30	80.00%	100.00%	4/5	1.00	30.00
JulesVerne4.txt	50	100.00%	100.00%	5/5	1.00	50.00
JulesVerne4.txt	70	100.00%	100.00%	5/5	1.00	70.00
caesar.txt	20	100.00%	100.00%	5/5	1.00	20.00
caesar.txt	30	60.00%	100.00%	3/5	1.00	30.00
caesar.txt	50	100.00%	100.00%	5/5	1.00	50.00
caesar.txt	70	80.00%	100.00%	4/5	1.00	70.00
caesar2.txt	20	100.00%	100.00%	5/5	1.00	20.00
caesar2.txt	30	100.00%	100.00%	5/5	1.00	30.00
caesar2.txt	50	100.00%	100.00%	5/5	1.00	50.00
caesar2.txt	70	80.00%	100.00%	4/5	1.00	70.00

Table B.7: Caesar cipher - Self-Adaptive Differential Evolution

Ciphertext	Pop.	Avg. rec.	Max. rec.	100% rec.	Avg. gen.	Avg. eval.
JulesVerne4.txt	20	100.00%	100.00%	5/5	1.60	32.00
JulesVerne4.txt	30	100.00%	100.00%	5/5	1.00	30.00
JulesVerne4.txt	50	100.00%	100.00%	5/5	1.00	50.00
JulesVerne4.txt	70	100.00%	100.00%	5/5	1.00	70.00
caesar.txt	20	100.00%	100.00%	5/5	1.00	20.00
caesar.txt	30	100.00%	100.00%	5/5	1.00	30.00
caesar.txt	50	100.00%	100.00%	5/5	1.00	50.00
caesar.txt	70	100.00%	100.00%	5/5	1.00	70.00
caesar2.txt	20	100.00%	100.00%	5/5	1.20	24.00
caesar2.txt	30	100.00%	100.00%	5/5	1.60	48.00
caesar2.txt	50	100.00%	100.00%	5/5	1.00	50.00
caesar2.txt	70	100.00%	100.00%	5/5	1.00	70.00

Table B.8: Caesar cipher - Simulated Annealing

Ciphertext	Avg. rec.	Max. rec.	100% rec.	Avg. eval.
JulesVerne4.txt	100.00%	100.00%	5/5	1.00
caesar.txt	100.00%	100.00%	5/5	1.00
caesar2.txt	80.00%	100.00%	4/5	1.00

Table B.9: Caesar cipher - Particle Swarm Optimization

Ciphertext	Pop.	Avg. rec.	Max. rec.	100% rec.	Avg. gen.	Avg. eval.
JulesVerne4.txt	20	100.00%	100.00%	5/5	1.00	20.00
JulesVerne4.txt	30	100.00%	100.00%	5/5	1.00	30.00
JulesVerne4.txt	50	100.00%	100.00%	5/5	1.20	60.00
JulesVerne4.txt	70	100.00%	100.00%	5/5	1.00	70.00
caesar.txt	20	100.00%	100.00%	5/5	2.00	40.00
caesar.txt	30	100.00%	100.00%	5/5	1.00	30.00
caesar.txt	50	100.00%	100.00%	5/5	1.00	50.00
caesar.txt	70	100.00%	100.00%	5/5	1.00	70.00
caesar2.txt	20	100.00%	100.00%	5/5	1.00	20.00
caesar2.txt	30	100.00%	100.00%	5/5	1.00	30.00
caesar2.txt	50	100.00%	100.00%	5/5	1.00	50.00
caesar2.txt	70	100.00%	100.00%	5/5	1.00	70.00

Table B.10: Caesar cipher - Ant Colony Optimization

Ciphertext	Pop.	Avg. rec.	Max. rec.	100% rec.	Avg. gen.	Avg. eval.
JulesVerne4.txt	20	100.00%	100.00%	5/5	1.20	4.00
JulesVerne4.txt	30	100.00%	100.00%	5/5	1.40	12.00
JulesVerne4.txt	50	100.00%	100.00%	5/5	1.20	10.00
JulesVerne4.txt	70	100.00%	100.00%	5/5	1.00	0.00
caesar.txt	20	100.00%	100.00%	5/5	2.00	20.00
caesar.txt	30	100.00%	100.00%	5/5	1.80	24.00
caesar.txt	50	100.00%	100.00%	5/5	1.00	0.00
caesar.txt	70	100.00%	100.00%	5/5	1.20	14.00
caesar2.txt	20	100.00%	100.00%	5/5	1.40	8.00
caesar2.txt	30	100.00%	100.00%	5/5	1.20	6.00
caesar2.txt	50	100.00%	100.00%	5/5	1.00	0.00
caesar2.txt	70	100.00%	100.00%	5/5	1.00	0.00

Table B.11: Caesar cipher - Artificial Bee Colony Optimization

Ciphertext	Pop.	Avg. rec.	Max. rec.	100% rec.	Avg. gen.	Avg. eval.
JulesVerne4.txt	20	100.00%	100.00%	5/5	1.00	40.00
JulesVerne4.txt	30	100.00%	100.00%	5/5	1.00	60.00
JulesVerne4.txt	50	100.00%	100.00%	5/5	1.00	100.00
JulesVerne4.txt	70	100.00%	100.00%	5/5	1.00	140.00
caesar.txt	20	100.00%	100.00%	5/5	1.00	40.00
caesar.txt	30	100.00%	100.00%	5/5	1.00	60.00
caesar.txt	50	100.00%	100.00%	5/5	1.00	100.00
caesar.txt	70	100.00%	100.00%	5/5	1.00	140.00
caesar2.txt	20	100.00%	100.00%	5/5	1.00	40.00
caesar2.txt	30	100.00%	100.00%	5/5	1.00	60.00
caesar2.txt	50	100.00%	100.00%	5/5	1.00	100.00
caesar2.txt	70	100.00%	100.00%	5/5	1.00	140.00

Table B.12: Caesar cipher - Cuckoo Search

Ciphertext	Pop.	Avg. rec.	Max. rec.	100% rec.	Avg. gen.	Avg. eval.
JulesVerne4.txt	20	100.00%	100.00%	5/5	1.00	20.00
JulesVerne4.txt	30	100.00%	100.00%	5/5	1.00	30.00
JulesVerne4.txt	50	100.00%	100.00%	5/5	1.00	50.00
JulesVerne4.txt	70	100.00%	100.00%	5/5	1.00	70.00
caesar.txt	20	100.00%	100.00%	5/5	1.40	28.00
caesar.txt	30	100.00%	100.00%	5/5	1.00	30.00
caesar.txt	50	100.00%	100.00%	5/5	1.00	50.00
caesar.txt	70	100.00%	100.00%	5/5	1.80	126.00
caesar2.txt	20	100.00%	100.00%	5/5	1.40	28.00
caesar2.txt	30	100.00%	100.00%	5/5	1.20	36.00
caesar2.txt	50	100.00%	100.00%	5/5	1.20	60.00
caesar2.txt	70	100.00%	100.00%	5/5	1.00	70.00

Table B.13: Columnar Transposition cipher - Simple Genetic Algorithm (m=0.02)

Ciphertext	Pop.	Avg. rec.	Max. rec.	100% rec.	Avg. gen.	Avg. eval.
JulesVerne1.txt	20	20.00%	25.00%	0/5		
JulesVerne1.txt	30	15.00%	25.00%	0/5		
JulesVerne1.txt	50	35.00%	100.00%	1/5	18.00	900.00
JulesVerne1.txt	70	20.00%	25.00%	0/5		

Table B.14: Columnar Transposition cipher - Simple Genetic Algorithm (m=0.2)

Ciphertext	Pop.	Avg. rec.	Max. rec.	100% rec.	Avg. gen.	Avg. eval.
JulesVerne1.txt	20	55.00%	100.00%	2/5	65.00	1300.00
JulesVerne1.txt	30	50.00%	100.00%	1/5	7.00	210.00
JulesVerne1.txt	50	55.00%	100.00%	3/5	47.00	2350.00
JulesVerne1.txt	70	70.00%	100.00%	3/5	93.67	6556.67

Table B.15: Columnar Transposition cipher - Simple Genetic Algorithm (m=1.0)

Ciphertext	Pop.	Avg. rec.	Max. rec.	100% rec.	Avg. gen.	Avg. eval.
JulesVerne1.txt	20	40.00%	100.00%	0/5		
JulesVerne1.txt	30	40.00%	100.00%	3/5	95.33	2860.00
JulesVerne1.txt	50	55.00%	100.00%	4/5	117.00	5850.00
JulesVerne1.txt	70	85.00%	100.00%	4/5	48.00	3360.00

Table B.16: Columnar Transposition cipher - Custom Genetic Algorithm (m=0.02)

Ciphertext	Pop.	Avg. rec.	Max. rec.	100% rec.	Avg. gen.	Avg. eval.
JulesVerne1.txt	20	25.00%	25.00%	0/5		
JulesVerne1.txt	30	30.00%	50.00%	0/5		
JulesVerne1.txt	50	25.00%	25.00%	0/5		
JulesVerne1.txt	70	25.00%	25.00%	0/5		

Table B.17: Columnar Transposition cipher - Custom Genetic Algorithm (m=0.2)

Ciphertext	Pop.	Avg. rec.	Max. rec.	100% rec.	Avg. gen.	Avg. eval.
JulesVerne1.txt	20	25.00%	75.00%	0/5		
JulesVerne1.txt	30	45.00%	100.00%	2/5	5.50	165.00
JulesVerne1.txt	50	25.00%	25.00%	0/5		
JulesVerne1.txt	70	40.00%	100.00%	1/5	17.00	1190.00

Table B.18: Columnar Transposition cipher - Custom Genetic Algorithm (m=1.0)

Ciphertext	Pop.	Avg. rec.	Max. rec.	100% rec.	Avg. gen.	Avg. eval.
JulesVerne1.txt	20	10.00%	25.00%	0/5		
JulesVerne1.txt	30	25.00%	25.00%	0/5		
JulesVerne1.txt	50	20.00%	50.00%	0/5		
JulesVerne1.txt	70	20.00%	25.00%	0/5		

Table B.19: Columnar Transposition cipher - Self-Adaptive Differential Evolution

Ciphertext	Pop.	Avg. rec.	Max. rec.	100% rec.	Avg. gen.	Avg. eval.
JulesVerne1.txt	20	85.00%	100.00%	4/5	68.50	1370.00
JulesVerne1.txt	30	70.00%	100.00%	3/5	50.33	1510.00
JulesVerne1.txt	50	40.00%	100.00%	2/5	54.50	2725.00
JulesVerne1.txt	70	100.00%	100.00%	5/5	63.60	4452.00

Table B.20: Columnar Transposition cipher - Simulated Annealing

Ciphertext	Avg. rec.	Max. rec.	100% rec.	Avg. eval.
JulesVerne1.txt	35.00%	50.00%	0/5	

Table B.21: Columnar Transposition cipher - Particle Swarm Optimization

Ciphertext	Pop.	Avg. rec.	Max. rec.	100% rec.	Avg. gen.	Avg. eval.
JulesVerne1.txt	20	65.00%	100.00%	3/5	50.33	1006.67
JulesVerne1.txt	30	80.00%	100.00%	4/5	55.50	1665.00
JulesVerne1.txt	50	100.00%	100.00%	5/5	84.80	4240.00
JulesVerne1.txt	70	100.00%	100.00%	5/5	20.20	1414.00

Table B.22: Columnar Transposition cipher - Ant Colony Optimization

Ciphertext	Pop.	Avg. rec.	Max. rec.	100% rec.	Avg. gen.	Avg. eval.
JulesVerne1.txt	20	100.00%	100.00%	5/5	48.20	944.00
JulesVerne1.txt	30	85.00%	100.00%	4/5	35.50	1035.00
JulesVerne1.txt	50	100.00%	100.00%	5/5	36.00	1750.00
JulesVerne1.txt	70	100.00%	100.00%	5/5	16.80	1106.00

Table B.23: Columnar Transposition cipher - Artificial Bee Colony Optimization

Ciphertext	Pop.	Avg. rec.	Max. rec.	100% rec.	Avg. gen.	Avg. eval.
JulesVerne1.txt	20	35.00%	100.00%	2/5	89.50	3580.00
JulesVerne1.txt	30	50.00%	100.00%	1/5	19.00	1140.00
JulesVerne1.txt	50	75.00%	100.00%	4/5	52.25	5225.00
JulesVerne1.txt	70	90.00%	100.00%	4/5	49.25	6895.00

Table B.24: Columnar Transposition cipher - Cuckoo Search

Ciphertext	Pop.	Avg. rec.	Max. rec.	100% rec.	Avg. gen.	Avg. eval.
JulesVerne1.txt	20	35.00%	100.00%	1/5	14.00	280.00
JulesVerne1.txt	30	25.00%	100.00%	1/5	15.00	450.00
JulesVerne1.txt	50	70.00%	100.00%	3/5	42.33	2116.67
JulesVerne1.txt	70	65.00%	100.00%	3/5	87.67	6136.67

Table B.25: Monoalphabetic Substitution cipher - Simple Genetic Algorithm (m=0.02)

Ciphertext	Pop.	Avg. rec.	Max. rec.	100% rec.	Avg. gen.	Avg. eval.
JulesVerne1.txt	20	4.62%	7.69%	0/5		
JulesVerne1.txt	30	4.62%	7.69%	0/5		
JulesVerne1.txt	50	3.85%	11.54%	0/5		
JulesVerne1.txt	70	5.38%	7.69%	0/5		

Table B.26: Monoalphabetic Substitution cipher - Simple Genetic Algorithm (m=0.2)

Ciphertext	Pop.	Avg. rec.	Max. rec.	100% rec.	Avg. gen.	Avg. eval.
JulesVerne1.txt	20	5.38%	19.23%	0/5		
JulesVerne1.txt	30	3.08%	7.69%	0/5		
JulesVerne1.txt	50	4.62%	11.54%	0/5		
JulesVerne1.txt	70	4.62%	11.54%	0/5		

Table B.27: Monoalphabetic Substitution cipher - Simple Genetic Algorithm (m=1.0)

Ciphertext	Pop.	Avg. rec.	Max. rec.	100% rec.	Avg. gen.	Avg. eval.
JulesVerne1.txt	20	3.85%	7.69%	0/5		
JulesVerne1.txt	30	3.08%	7.69%	0/5		
JulesVerne1.txt	50	3.85%	7.69%	0/5		
JulesVerne1.txt	70	4.62%	7.69%	0/5		

Table B.28: Monoalphabetic Substitution cipher - Custom Genetic Algorithm (m=0.02)

Ciphertext	Pop.	Avg. rec.	Max. rec.	100% rec.	Avg. gen.	Avg. eval.
JulesVerne1.txt	20	50.00%	76.92%	0/5		
JulesVerne1.txt	30	66.15%	92.31%	0/5		
JulesVerne1.txt	50	92.31%	100.00%	2/5	154.50	7725.00
JulesVerne1.txt	70	77.69%	100.00%	3/5	152.67	10686.67

Table B.29: Monoalphabetic Substitution cipher - Custom Genetic Algorithm (m=0.2)

Ciphertext	Pop.	Avg. rec.	Max. rec.	100% rec.	Avg. gen.	Avg. eval.
JulesVerne1.txt	20	74.62%	100.00%	2/5	174.00	3480.00
JulesVerne1.txt	30	59.23%	100.00%	1/5	190.00	5700.00
JulesVerne1.txt	50	86.15%	100.00%	4/5	156.50	7825.00
JulesVerne1.txt	70	70.00%	100.00%	3/5	136.67	9566.67

Table B.30: Monoalphabetic Substitution cipher - Custom Genetic Algorithm (m=1.0)

Ciphertext	Pop.	Avg. rec.	Max. rec.	100% rec.	Avg. gen.	Avg. eval.
JulesVerne1.txt	20	74.62%	92.31%	0/5		
JulesVerne1.txt	30	72.31%	100.00%	2/5	140.50	4215.00
JulesVerne1.txt	50	69.23%	100.00%	3/5	149.33	7466.67
JulesVerne1.txt	70	100.00%	100.00%	5/5	139.00	9730.00

Table B.31: Monoalphabetic Substitution cipher - Self-Adaptive Differential Evolution

Ciphertext	Pop.	Avg. rec.	Max. rec.	100% rec.	Avg. gen.	Avg. eval.
JulesVerne1.txt	20	6.15%	11.54%	0/5		
JulesVerne1.txt	30	6.15%	11.54%	0/5		
JulesVerne1.txt	50	6.15%	11.54%	0/5		
JulesVerne1.txt	70	7.69%	11.54%	0/5		

Table B.32: Monoalphabetic Substitution cipher - Simulated Annealing

Ciphertext	Avg. rec.	Max. rec.	100% rec.	Avg. eval.
JulesVerne1.txt	3.85%	7.69%	0/5	

Table B.33: Monoalphabetic Substitution cipher - Particle Swarm Optimization

Ciphertext	Pop.	Avg. rec.	Max. rec.	100% rec.	Avg. gen.	Avg. eval.
JulesVerne1.txt	20	3.85%	7.69%	0/5		
JulesVerne1.txt	30	6.15%	11.54%	0/5		
JulesVerne1.txt	50	1.54%	3.85%	0/5		
JulesVerne1.txt	70	6.92%	15.38%	0/5		

Table B.34: Monoalphabetic Substitution cipher - Ant Colony Optimization

Ciphertext	Pop.	Avg. rec.	Max. rec.	100% rec.	Avg. gen.	Avg. eval.
JulesVerne1.txt	20	1.54%	7.69%	0/5		
JulesVerne1.txt	30	3.08%	7.69%	0/5		
JulesVerne1.txt	50	5.38%	7.69%	0/5		
JulesVerne1.txt	70	8.46%	15.38%	0/5		

Table B.35: Monoalphabetic Substitution cipher - Artificial Bee Colony Optimization

Ciphertext	Pop.	Avg. rec.	Max. rec.	100% rec.	Avg. gen.	Avg. eval.
JulesVerne1.txt	20	4.62%	7.69%	0/5		
JulesVerne1.txt	30	3.85%	11.54%	0/5		
JulesVerne1.txt	50	6.15%	11.54%	0/5		
JulesVerne1.txt	70	5.38%	11.54%	0/5		

Table B.36: Monoalphabetic Substitution cipher - Cuckoo Search

Ciphertext	Pop.	Avg. rec.	Max. rec.	100% rec.	Avg. gen.	Avg. eval.
JulesVerne1.txt	20	3.08%	7.69%	0/5		
JulesVerne1.txt	30	3.85%	7.69%	0/5		
JulesVerne1.txt	50	4.62%	11.54%	0/5		
JulesVerne1.txt	70	3.85%	7.69%	0/5		

Table B.37: Vigenère cipher - Simple Genetic Algorithm (m=0.02)

Ciphertext	Pop.	Avg. rec.	Max. rec.	100% rec.	Avg. gen.	Avg. eval.
JulesVerne1_key3.txt	20	93.33%	100.00%	4/5	55.75	1115.00
JulesVerne1_key3.txt	30	100.00%	100.00%	5/5	39.20	1176.00
JulesVerne1_key3.txt	50	100.00%	100.00%	5/5	3.60	180.00
JulesVerne1_key3.txt	70	100.00%	100.00%	5/5	5.40	378.00
JulesVerne1_key5.txt	20	100.00%	100.00%	5/5	98.60	1972.00
JulesVerne1_key5.txt	30	100.00%	100.00%	5/5	64.00	1920.00
JulesVerne1_key5.txt	50	100.00%	100.00%	5/5	40.00	2000.00
JulesVerne1_key5.txt	70	100.00%	100.00%	5/5	40.00	2800.00
JulesVerne1_key10.txt	20	90.00%	100.00%	1/5	155.00	3100.00
JulesVerne1_key10.txt	30	96.00%	100.00%	4/5	136.50	4095.00
JulesVerne1_key10.txt	50	100.00%	100.00%	5/5	103.80	5190.00
JulesVerne1_key10.txt	70	100.00%	100.00%	5/5	56.00	3920.00

Table B.38: Vigenère cipher - Simple Genetic Algorithm (m=0.2)

Ciphertext	Pop.	Avg. rec.	Max. rec.	100% rec.	Avg. gen.	Avg. eval.
JulesVerne1_key3.txt	20	100.00%	100.00%	5/5	13.40	268.00
JulesVerne1_key3.txt	30	100.00%	100.00%	5/5	8.00	240.00
JulesVerne1_key3.txt	50	100.00%	100.00%	5/5	6.00	300.00
JulesVerne1_key3.txt	70	100.00%	100.00%	5/5	4.20	294.00
JulesVerne1_key5.txt	20	100.00%	100.00%	5/5	29.40	588.00
JulesVerne1_key5.txt	30	100.00%	100.00%	5/5	23.60	708.00
JulesVerne1_key5.txt	50	100.00%	100.00%	5/5	13.80	690.00
JulesVerne1_key5.txt	70	100.00%	100.00%	5/5	11.00	770.00
JulesVerne1_key10.txt	20	100.00%	100.00%	5/5	114.80	2296.00
JulesVerne1_key10.txt	30	100.00%	100.00%	5/5	77.40	2322.00
JulesVerne1_key10.txt	50	100.00%	100.00%	5/5	52.40	2620.00
JulesVerne1_key10.txt	70	100.00%	100.00%	5/5	34.80	2436.00

Table B.39: Vigenère cipher - Simple Genetic Algorithm (m=1.0)

Ciphertext	Pop.	Avg. rec.	Max. rec.	100% rec.	Avg. gen.	Avg. eval.
JulesVerne1_key3.txt	20	80.00%	100.00%	2/5	85.00	1700.00
JulesVerne1_key3.txt	30	80.00%	100.00%	2/5	63.00	1890.00
JulesVerne1_key3.txt	50	80.00%	100.00%	2/5	44.50	2225.00
JulesVerne1_key3.txt	70	86.67%	100.00%	3/5	95.67	6696.67
JulesVerne1_key5.txt	20	56.00%	60.00%	0/5		
JulesVerne1_key5.txt	30	60.00%	60.00%	0/5		
JulesVerne1_key5.txt	50	56.00%	60.00%	0/5		
JulesVerne1_key5.txt	70	60.00%	60.00%	0/5		
JulesVerne1_key10.txt	20	32.00%	40.00%	0/5		
JulesVerne1_key10.txt	30	36.00%	50.00%	0/5		
JulesVerne1_key10.txt	50	34.00%	50.00%	0/5		
JulesVerne1_key10.txt	70	38.00%	40.00%	0/5		

Table B.40: Vigenère cipher - Custom Genetic Algorithm (m=0.02)

Ciphertext	Pop.	Avg. rec.	Max. rec.	100% rec.	Avg. gen.	Avg. eval.
JulesVerne1_key3.txt	20	46.67%	66.67%	0/5		
JulesVerne1_key3.txt	30	66.67%	100.00%	1/5	6.00	180.00
JulesVerne1_key3.txt	50	60.00%	100.00%	1/5	7.00	350.00
JulesVerne1_key3.txt	70	86.67%	100.00%	3/5	5.00	350.00
JulesVerne1_key5.txt	20	28.00%	60.00%	0/5		
JulesVerne1_key5.txt	30	60.00%	60.00%	0/5		
JulesVerne1_key5.txt	50	72.00%	80.00%	0/5		
JulesVerne1_key5.txt	70	84.00%	100.00%	2/5	8.50	595.00
JulesVerne1_key10.txt	20	26.00%	40.00%	0/5		
JulesVerne1_key10.txt	30	34.00%	40.00%	0/5		
JulesVerne1_key10.txt	50	56.00%	70.00%	0/5		
JulesVerne1_key10.txt	70	54.00%	70.00%	0/5		

Table B.41: Vigenère cipher - Custom Genetic Algorithm (m=0.2)

Ciphertext	Pop.	Avg. rec.	Max. rec.	100% rec.	Avg. gen.	Avg. eval.
JulesVerne1_key3.txt	20	53.33%	66.67%	0/5		
JulesVerne1_key3.txt	30	53.33%	66.67%	0/5		
JulesVerne1_key3.txt	50	80.00%	100.00%	2/5	5.00	250.00
JulesVerne1_key3.txt	70	73.33%	100.00%	1/5	7.00	490.00
JulesVerne1_key5.txt	20	40.00%	60.00%	0/5		
JulesVerne1_key5.txt	30	48.00%	60.00%	0/5		
JulesVerne1_key5.txt	50	76.00%	100.00%	1/5	8.00	400.00
JulesVerne1_key5.txt	70	80.00%	100.00%	1/5	7.00	490.00
JulesVerne1_key10.txt	20	44.00%	50.00%	0/5		
JulesVerne1_key10.txt	30	52.00%	70.00%	0/5		
JulesVerne1_key10.txt	50	66.00%	80.00%	0/5		
JulesVerne1_key10.txt	70	62.00%	70.00%	0/5		

Table B.42: Vigenère cipher - Custom Genetic Algorithm (m=1.0)

Ciphertext	Pop.	Avg. rec.	Max. rec.	100% rec.	Avg. gen.	Avg. eval.
JulesVerne1_key3.txt	20	53.33%	66.67%	0/5		
JulesVerne1_key3.txt	30	66.67%	66.67%	0/5		
JulesVerne1_key3.txt	50	80.00%	100.00%	3/5	4.67	233.33
JulesVerne1_key3.txt	70	73.33%	100.00%	1/5	4.00	280.00
JulesVerne1_key5.txt	20	44.00%	80.00%	0/5		
JulesVerne1_key5.txt	30	40.00%	40.00%	0/5		
JulesVerne1_key5.txt	50	68.00%	100.00%	1/5	6.00	300.00
JulesVerne1_key5.txt	70	68.00%	80.00%	0/5		
JulesVerne1_key10.txt	20	42.00%	70.00%	0/5		
JulesVerne1_key10.txt	30	64.00%	90.00%	0/5		
JulesVerne1_key10.txt	50	48.00%	70.00%	0/5		
JulesVerne1_key10.txt	70	62.00%	80.00%	0/5		

Table B.43: Vigenère cipher - Self-Adaptive Differential Evolution

Ciphertext	Pop.	Avg. rec.	Max. rec.	100% rec.	Avg. gen.	Avg. eval.
JulesVerne1_key3.txt	20	100.00%	100.00%	5/5	27.80	556.00
JulesVerne1_key3.txt	30	100.00%	100.00%	5/5	11.80	354.00
JulesVerne1_key3.txt	50	100.00%	100.00%	5/5	14.40	720.00
JulesVerne1_key3.txt	70	100.00%	100.00%	5/5	16.20	1134.00
JulesVerne1_key5.txt	20	100.00%	100.00%	5/5	52.80	1056.00
JulesVerne1_key5.txt	30	100.00%	100.00%	5/5	42.20	1266.00
JulesVerne1_key5.txt	50	100.00%	100.00%	5/5	44.60	2230.00
JulesVerne1_key5.txt	70	100.00%	100.00%	5/5	41.20	2884.00
JulesVerne1_key10.txt	20	100.00%	100.00%	5/5	153.80	3076.00
JulesVerne1_key10.txt	30	100.00%	100.00%	5/5	116.00	3480.00
JulesVerne1_key10.txt	50	100.00%	100.00%	5/5	129.60	6480.00
JulesVerne1_key10.txt	70	100.00%	100.00%	5/5	118.60	8302.00

Table B.44: Vigenère cipher - Simulated Annealing

Ciphertext	Avg. rec.	Max. rec.	100% rec.	Avg. eval.
JulesVerne1_key3.txt	100.00%	100.00%	5/5	30.20
JulesVerne1_key5.txt	64.00%	80.00%	0/5	
JulesVerne1_key10.txt	80.00%	90.00%	0/5	

Table B.45: Vigenère cipher - Particle Swarm Optimization

Ciphertext	Pop.	Avg. rec.	Max. rec.	100% rec.	Avg. gen.	Avg. eval.
JulesVerne1_key3.txt	20	100.00%	100.00%	5/5	42.00	840.00
JulesVerne1_key3.txt	30	100.00%	100.00%	5/5	42.00	1260.00
JulesVerne1_key3.txt	50	100.00%	100.00%	5/5	17.40	870.00
JulesVerne1_key3.txt	70	100.00%	100.00%	5/5	25.20	1764.00
JulesVerne1_key5.txt	20	96.00%	100.00%	4/5	134.75	2695.00
JulesVerne1_key5.txt	30	92.00%	100.00%	3/5	105.33	3160.00
JulesVerne1_key5.txt	50	96.00%	100.00%	4/5	127.75	6387.50
JulesVerne1_key5.txt	70	96.00%	100.00%	4/5	94.75	6632.50
JulesVerne1_key10.txt	20	60.00%	70.00%	0/5		
JulesVerne1_key10.txt	30	60.00%	70.00%	0/5		
JulesVerne1_key10.txt	50	76.00%	80.00%	0/5		
JulesVerne1_key10.txt	70	70.00%	80.00%	0/5		

Table B.46: Vigenère cipher - Ant Colony Optimization

Ciphertext	Pop.	Avg. rec.	Max. rec.	100% rec.	Avg. gen.	Avg. eval.
JulesVerne1_key3.txt	20	80.00%	100.00%	2/5	77.00	1520.00
JulesVerne1_key3.txt	30	100.00%	100.00%	5/5	58.80	1734.00
JulesVerne1_key3.txt	50	100.00%	100.00%	5/5	31.40	1520.00
JulesVerne1_key3.txt	70	100.00%	100.00%	5/5	12.40	798.00
JulesVerne1_key5.txt	20	80.00%	100.00%	1/5	195.00	3880.00
JulesVerne1_key5.txt	30	84.00%	100.00%	3/5	144.33	4300.00
JulesVerne1_key5.txt	50	92.00%	100.00%	4/5	109.00	5400.00
JulesVerne1_key5.txt	70	96.00%	100.00%	4/5	106.25	7367.50
JulesVerne1_key10.txt	20	40.00%	50.00%	0/5		
JulesVerne1_key10.txt	30	58.00%	80.00%	0/5		
JulesVerne1_key10.txt	50	44.00%	60.00%	0/5		
JulesVerne1_key10.txt	70	40.00%	50.00%	0/5		

Table B.47: Vigenère cipher - Artificial Bee Colony Optimization

Ciphertext	Pop.	Avg. rec.	Max. rec.	100% rec.	Avg. gen.	Avg. eval.
JulesVerne1_key3.txt	20	93.33%	100.00%	4/5	35.50	1420.00
JulesVerne1_key3.txt	30	100.00%	100.00%	5/5	27.40	1644.00
JulesVerne1_key3.txt	50	100.00%	100.00%	5/5	16.60	1660.00
JulesVerne1_key3.txt	70	100.00%	100.00%	5/5	9.00	1260.00
JulesVerne1_key5.txt	20	84.00%	100.00%	2/5	65.50	2620.00
JulesVerne1_key5.txt	30	96.00%	100.00%	4/5	91.25	5475.00
JulesVerne1_key5.txt	50	96.00%	100.00%	4/5	54.25	5425.00
JulesVerne1_key5.txt	70	100.00%	100.00%	5/5	55.40	7756.00
JulesVerne1_key10.txt	20	60.00%	70.00%	0/5		
JulesVerne1_key10.txt	30	80.00%	100.00%	1/5	155.00	9300.00
JulesVerne1_key10.txt	50	76.00%	90.00%	0/5		
JulesVerne1_key10.txt	70	88.00%	100.00%	1/5	114.00	15960.00

Table B.48: Vigenère cipher - Cuckoo Search

Ciphertext	Pop.	Avg. rec.	Max. rec.	100% rec.	Avg. gen.	Avg. eval.
JulesVerne1_key3.txt	20	66.67%	100.00%	1/5	24.00	480.00
JulesVerne1_key3.txt	30	73.33%	100.00%	1/5	11.00	330.00
JulesVerne1_key3.txt	50	73.33%	100.00%	1/5	13.00	650.00
JulesVerne1_key3.txt	70	80.00%	100.00%	2/5	30.50	2135.00
JulesVerne1_key5.txt	20	44.00%	60.00%	0/5		
JulesVerne1_key5.txt	30	64.00%	80.00%	0/5		
JulesVerne1_key5.txt	50	44.00%	80.00%	0/5		
JulesVerne1_key5.txt	70	76.00%	100.00%	2/5	125.50	8785.00
JulesVerne1_key10.txt	20	46.00%	70.00%	0/5		
JulesVerne1_key10.txt	30	32.00%	50.00%	0/5		
JulesVerne1_key10.txt	50	56.00%	70.00%	0/5		
JulesVerne1_key10.txt	70	44.00%	50.00%	0/5		

Table B.49: Playfair cipher - Simple Genetic Algorithm (m=0.02)

Ciphertext	Pop.	Avg. rec.	Max. rec.	100% rec.	Avg. gen.	Avg. eval.
JulesVerne.txt	20	41.60%	68.00%	0/5		
JulesVerne.txt	30	37.60%	56.00%	0/5		
JulesVerne.txt	50	32.00%	48.00%	0/5		
JulesVerne.txt	70	64.80%	84.00%	0/5		
JulesVerne1.txt	20	7.20%	12.00%	0/5		
JulesVerne1.txt	30	12.00%	20.00%	0/5		
JulesVerne1.txt	50	14.40%	36.00%	0/5		
JulesVerne1.txt	70	13.60%	24.00%	0/5		

Table B.50: Playfair cipher - Simple Genetic Algorithm (m=0.2)

Ciphertext	Pop.	Avg. rec.	Max. rec.	100% rec.	Avg. gen.	Avg. eval.
JulesVerne.txt	20	66.40%	100.00%	1/5	122.00	2440.00
JulesVerne.txt	30	59.20%	100.00%	1/5	191.00	5730.00
JulesVerne.txt	50	63.20%	92.00%	0/5		
JulesVerne.txt	70	77.60%	100.00%	2/5	39.00	2730.00
JulesVerne1.txt	20	21.60%	52.00%	0/5		
JulesVerne1.txt	30	13.60%	28.00%	0/5		
JulesVerne1.txt	50	28.80%	56.00%	0/5		
JulesVerne1.txt	70	24.80%	52.00%	0/5		

Table B.51: Playfair cipher - Simple Genetic Algorithm (m=1.0)

Ciphertext	Pop.	Avg. rec.	Max. rec.	100% rec.	Avg. gen.	Avg. eval.
JulesVerne.txt	20	64.00%	72.00%	0/5		
JulesVerne.txt	30	62.40%	88.00%	0/5		
JulesVerne.txt	50	64.80%	80.00%	0/5		
JulesVerne.txt	70	73.60%	84.00%	0/5		
JulesVerne1.txt	20	5.60%	12.00%	0/5		
JulesVerne1.txt	30	12.80%	24.00%	0/5		
JulesVerne1.txt	50	12.00%	24.00%	0/5		
JulesVerne1.txt	70	25.60%	52.00%	0/5		

Table B.52: Playfair cipher - Custom Genetic Algorithm (m=0.02)

Ciphertext	Pop.	Avg. rec.	Max. rec.	100% rec.	Avg. gen.	Avg. eval.
JulesVerne.txt	20	15.20%	24.00%	0/5		
JulesVerne.txt	30	12.80%	24.00%	0/5		
JulesVerne.txt	50	10.40%	16.00%	0/5		
JulesVerne.txt	70	28.00%	100.00%	1/5	33.00	2310.00
JulesVerne1.txt	20	9.60%	24.00%	0/5		
JulesVerne1.txt	30	14.40%	28.00%	0/5		
JulesVerne1.txt	50	9.60%	12.00%	0/5		
JulesVerne1.txt	70	6.40%	8.00%	0/5		

Table B.53: Playfair cipher - Custom Genetic Algorithm (m=0.2)

Ciphertext	Pop.	Avg. rec.	Max. rec.	100% rec.	Avg. gen.	Avg. eval.
JulesVerne.txt	20	10.40%	20.00%	0/5		
JulesVerne.txt	30	9.60%	16.00%	0/5		
JulesVerne.txt	50	8.00%	12.00%	0/5		
JulesVerne.txt	70	6.40%	12.00%	0/5		
JulesVerne1.txt	20	6.40%	12.00%	0/5		
JulesVerne1.txt	30	13.60%	16.00%	0/5		
JulesVerne1.txt	50	8.80%	12.00%	0/5		
JulesVerne1.txt	70	10.40%	16.00%	0/5		

Table B.54: Playfair cipher - Custom Genetic Algorithm (m=1.0)

Ciphertext	Pop.	Avg. rec.	Max. rec.	100% rec.	Avg. gen.	Avg. eval.
JulesVerne.txt	20	15.20%	32.00%	0/5		
JulesVerne.txt	30	7.20%	12.00%	0/5		
JulesVerne.txt	50	11.20%	12.00%	0/5		
JulesVerne.txt	70	12.00%	16.00%	0/5		
JulesVerne1.txt	20	7.20%	16.00%	0/5		
JulesVerne1.txt	30	13.60%	20.00%	0/5		
JulesVerne1.txt	50	11.20%	24.00%	0/5		
JulesVerne1.txt	70	10.40%	16.00%	0/5		

Table B.55: Playfair cipher - Self-Adaptive Differential Evolution

Ciphertext	Pop.	Avg. rec.	Max. rec.	100% rec.	Avg. gen.	Avg. eval.
JulesVerne.txt	20	72.00%	100.00%	1/5	29.00	580.00
JulesVerne.txt	30	60.80%	76.00%	0/5		
JulesVerne.txt	50	69.60%	92.00%	0/5		
JulesVerne.txt	70	66.40%	84.00%	0/5		
JulesVerne1.txt	20	18.40%	28.00%	0/5		
JulesVerne1.txt	30	18.40%	40.00%	0/5		
JulesVerne1.txt	50	32.00%	56.00%	0/5		
JulesVerne1.txt	70	12.00%	24.00%	0/5		

Table B.56: Playfair cipher - Simulated Annealing

Ciphertext	Avg. rec.	Max. rec.	100% rec.	Avg. eval.
JulesVerne.txt	43.20%	76.00%	0/5	
JulesVerne1.txt	14.40%	52.00%	0/5	

Table B.57: Playfair cipher - Particle Swarm Optimization

Ciphertext	Pop.	Avg. rec.	Max. rec.	100% rec.	Avg. gen.	Avg. eval.
JulesVerne.txt	20	69.60%	88.00%	0/5		
JulesVerne.txt	30	80.80%	84.00%	0/5		
JulesVerne.txt	50	76.80%	88.00%	0/5		
JulesVerne.txt	70	92.00%	100.00%	2/5	115.00	8050.00
JulesVerne1.txt	20	17.60%	28.00%	0/5		
JulesVerne1.txt	30	9.60%	20.00%	0/5		
JulesVerne1.txt	50	17.60%	44.00%	0/5		
JulesVerne1.txt	70	21.60%	60.00%	0/5		

Table B.58: Playfair cipher - Ant Colony Optimization

Ciphertext	Pop.	Avg. rec.	Max. rec.	100% rec.	Avg. gen.	Avg. eval.
JulesVerne.txt	20	74.40%	88.00%	0/5		
JulesVerne.txt	30	76.00%	80.00%	0/5		
JulesVerne.txt	50	86.40%	100.00%	2/5	154.00	7650.00
JulesVerne.txt	70	69.60%	84.00%	0/5		
JulesVerne1.txt	20	13.60%	40.00%	0/5		
JulesVerne1.txt	30	18.40%	48.00%	0/5		
JulesVerne1.txt	50	9.60%	20.00%	0/5		
JulesVerne1.txt	70	20.00%	56.00%	0/5		

Table B.59: Playfair cipher - Artificial Bee Colony Optimization

Ciphertext	Pop.	Avg. rec.	Max. rec.	100% rec.	Avg. gen.	Avg. eval.
JulesVerne.txt	20	53.60%	76.00%	0/5		
JulesVerne.txt	30	53.60%	84.00%	0/5		
JulesVerne.txt	50	56.00%	88.00%	0/5		
JulesVerne.txt	70	66.40%	88.00%	0/5		
JulesVerne1.txt	20	18.40%	40.00%	0/5		
JulesVerne1.txt	30	11.20%	24.00%	0/5		
JulesVerne1.txt	50	12.80%	24.00%	0/5		
JulesVerne1.txt	70	20.80%	32.00%	0/5		

Table B.60: Playfair cipher - Cuckoo Search

Ciphertext	Pop.	Avg. rec.	Max. rec.	100% rec.	Avg. gen.	Avg. eval.
JulesVerne.txt	20	31.20%	48.00%	0/5		
JulesVerne.txt	30	58.40%	68.00%	0/5		
JulesVerne.txt	50	76.80%	84.00%	0/5		
JulesVerne.txt	70	85.60%	100.00%	1/5	162.00	11340.00
JulesVerne1.txt	20	7.20%	12.00%	0/5		
JulesVerne1.txt	30	15.20%	36.00%	0/5		
JulesVerne1.txt	50	8.00%	24.00%	0/5		
JulesVerne1.txt	70	20.80%	56.00%	0/5		

Table B.61: DES - Simple Genetic Algorithm (m=0.02)

Ciphertext	Pop.	Avg. rec.	Max. rec.	100% rec.	Avg. gen.	Avg. eval.
JulesVerne1_des.txt	20	49.38%	56.25%	0/5		
JulesVerne1_des.txt	30	47.19%	57.81%	0/5		
JulesVerne1_des.txt	50	48.12%	56.25%	0/5		
JulesVerne1_des.txt	70	50.94%	56.25%	0/5		

Table B.62: DES - Simple Genetic Algorithm (m=0.2)

Ciphertext	Pop.	Avg. rec.	Max. rec.	100% rec.	Avg. gen.	Avg. eval.
JulesVerne1_des.txt	20	52.19%	60.94%	0/5		
JulesVerne1_des.txt	30	53.12%	56.25%	0/5		
JulesVerne1_des.txt	50	53.75%	57.81%	0/5		
JulesVerne1_des.txt	70	51.56%	56.25%	0/5		

Table B.63: DES - Simple Genetic Algorithm (m=1.0)

Ciphertext	Pop.	Avg. rec.	Max. rec.	100% rec.	Avg. gen.	Avg. eval.
JulesVerne1_des.txt	20	50.94%	54.69%	0/5		
JulesVerne1_des.txt	30	52.19%	65.62%	0/5		
JulesVerne1_des.txt	50	48.12%	54.69%	0/5		
JulesVerne1_des.txt	70	50.62%	57.81%	0/5		

Table B.64: DES - Custom Genetic Algorithm (m=0.02)

Ciphertext	Pop.	Avg. rec.	Max. rec.	100% rec.	Avg. gen.	Avg. eval.
JulesVerne1_des.txt	20	51.25%	62.50%	0/5		
JulesVerne1_des.txt	30	44.06%	54.69%	0/5		
JulesVerne1_des.txt	50	51.56%	57.81%	0/5		
JulesVerne1_des.txt	70	49.06%	53.12%	0/5		

Table B.65: DES - Custom Genetic Algorithm (m=0.2)

Ciphertext	Pop.	Avg. rec.	Max. rec.	100% rec.	Avg. gen.	Avg. eval.
JulesVerne1_des.txt	20	50.31%	65.62%	0/5		
JulesVerne1_des.txt	30	50.94%	62.50%	0/5		
JulesVerne1_des.txt	50	50.00%	54.69%	0/5		
JulesVerne1_des.txt	70	54.38%	67.19%	0/5		

Table B.66: DES - Custom Genetic Algorithm (m=1.0)

Ciphertext	Pop.	Avg. rec.	Max. rec.	100% rec.	Avg. gen.	Avg. eval.
JulesVerne1_des.txt	20	47.50%	51.56%	0/5		
JulesVerne1_des.txt	30	48.75%	56.25%	0/5		
JulesVerne1_des.txt	50	44.38%	53.12%	0/5		
JulesVerne1_des.txt	70	53.75%	62.50%	0/5		

Table B.67: DES - Self-Adaptive Differential Evolution

Ciphertext	Pop.	Avg. rec.	Max. rec.	100% rec.	Avg. gen.	Avg. eval.
JulesVerne1_des.txt	20	52.81%	62.50%	0/5		
JulesVerne1_des.txt	30	53.75%	59.38%	0/5		
JulesVerne1_des.txt	50	52.19%	57.81%	0/5		
JulesVerne1_des.txt	70	50.94%	64.06%	0/5		

Table B.68: DES - Simulated Annealing

Ciphertext	Avg. rec.	Max. rec.	100% rec.	Avg. eval.
JulesVerne1_des.txt	49.69%	54.69%	0/5	

Table B.69: DES - Particle Swarm Optimization

Ciphertext	Pop.	Avg. rec.	Max. rec.	100% rec.	Avg. gen.	Avg. eval.
JulesVerne1_des.txt	20	51.88%	57.81%	0/5		
JulesVerne1_des.txt	30	52.19%	57.81%	0/5		
JulesVerne1_des.txt	50	47.81%	53.12%	0/5		
JulesVerne1_des.txt	70	53.75%	65.62%	0/5		

Table B.70: DES - Ant Colony Optimization

Ciphertext	Pop.	Avg. rec.	Max. rec.	100% rec.	Avg. gen.	Avg. eval.
JulesVerne1_des.txt	20	50.94%	57.81%	0/5		
JulesVerne1_des.txt	30	53.75%	57.81%	0/5		
JulesVerne1_des.txt	50	49.69%	54.69%	0/5		
JulesVerne1_des.txt	70	48.44%	56.25%	0/5		

Table B.71: DES - Artificial Bee Colony Optimization

Ciphertext	Pop.	Avg. rec.	Max. rec.	100% rec.	Avg. gen.	Avg. eval.
JulesVerne1_des.txt	20	49.38%	62.50%	0/5		
JulesVerne1_des.txt	30	48.12%	60.94%	0/5		
JulesVerne1_des.txt	50	45.94%	54.69%	0/5		
JulesVerne1_des.txt	70	50.31%	62.50%	0/5		

Table B.72: DES - Cuckoo Search

Ciphertext	Pop.	Avg. rec.	Max. rec.	100% rec.	Avg. gen.	Avg. eval.
JulesVerne1_des.txt	20	53.12%	57.81%	0/5		
JulesVerne1_des.txt	30	48.44%	56.25%	0/5		
JulesVerne1_des.txt	50	48.75%	54.69%	0/5		
JulesVerne1_des.txt	70	56.56%	65.62%	0/5		

Table B.73: AES - Simple Genetic Algorithm (m=0.02)

Ciphertext	Pop.	Avg. rec.	Max. rec.	100% rec.	Avg. gen.	Avg. eval.
JulesVerne1_aes.txt	20	49.38%	52.34%	0/5		
JulesVerne1_aes.txt	30	49.69%	54.69%	0/5		
JulesVerne1_aes.txt	50	49.38%	53.91%	0/5		
JulesVerne1_aes.txt	70	51.88%	59.38%	0/5		

Table B.74: AES - Simple Genetic Algorithm (m=0.2)

Ciphertext	Pop.	Avg. rec.	Max. rec.	100% rec.	Avg. gen.	Avg. eval.
JulesVerne1_aes.txt	20	50.62%	59.38%	0/5		
JulesVerne1_aes.txt	30	50.16%	55.47%	0/5		
JulesVerne1_aes.txt	50	47.19%	58.59%	0/5		
JulesVerne1_aes.txt	70	51.25%	57.03%	0/5		

Table B.75: AES - Simple Genetic Algorithm (m=1.0)

Ciphertext	Pop.	Avg. rec.	Max. rec.	100% rec.	Avg. gen.	Avg. eval.
JulesVerne1_aes.txt	20	49.69%	57.03%	0/5		
JulesVerne1_aes.txt	30	49.69%	60.16%	0/5		
JulesVerne1_aes.txt	50	50.78%	58.59%	0/5		
JulesVerne1_aes.txt	70	50.31%	56.25%	0/5		

Table B.76: AES - Custom Genetic Algorithm (m=0.02)

Ciphertext	Pop.	Avg. rec.	Max. rec.	100% rec.	Avg. gen.	Avg. eval.
JulesVerne1_aes.txt	20	47.34%	55.47%	0/5		
JulesVerne1_aes.txt	30	49.69%	54.69%	0/5		
JulesVerne1_aes.txt	50	52.66%	57.03%	0/5		
JulesVerne1_aes.txt	70	49.69%	53.91%	0/5		

Table B.77: AES - Custom Genetic Algorithm (m=0.2)

Ciphertext	Pop.	Avg. rec.	Max. rec.	100% rec.	Avg. gen.	Avg. eval.
JulesVerne1_aes.txt	20	52.50%	58.59%	0/5		
JulesVerne1_aes.txt	30	52.50%	54.69%	0/5		
JulesVerne1_aes.txt	50	50.00%	57.03%	0/5		
JulesVerne1_aes.txt	70	52.66%	59.38%	0/5		

Table B.78: AES - Custom Genetic Algorithm (m=1.0)

Ciphertext	Pop.	Avg. rec.	Max. rec.	100% rec.	Avg. gen.	Avg. eval.
JulesVerne1_aes.txt	20	50.94%	58.59%	0/5		
JulesVerne1_aes.txt	30	48.44%	56.25%	0/5		
JulesVerne1_aes.txt	50	47.19%	50.78%	0/5		
JulesVerne1_aes.txt	70	52.03%	56.25%	0/5		

Table B.79: AES - Self-Adaptive Differential Evolution

Ciphertext	Pop.	Avg. rec.	Max. rec.	100% rec.	Avg. gen.	Avg. eval.
JulesVerne1_aes.txt	20	47.97%	53.12%	0/5		
JulesVerne1_aes.txt	30	49.84%	53.91%	0/5		
JulesVerne1_aes.txt	50	47.97%	51.56%	0/5		
JulesVerne1_aes.txt	70	44.69%	46.09%	0/5		

Table B.80: AES - Simulated Annealing

Ciphertext	Avg. rec.	Max. rec.	100% rec.	Avg. eval.
JulesVerne1_aes.txt	48.75%	52.34%	0/5	

Table B.81: AES - Particle Swarm Optimization

Ciphertext	Pop.	Avg. rec.	Max. rec.	100% rec.	Avg. gen.	Avg. eval.
JulesVerne1_aes.txt	20	52.97%	55.47%	0/5		
JulesVerne1_aes.txt	30	52.97%	56.25%	0/5		
JulesVerne1_aes.txt	50	52.50%	55.47%	0/5		
JulesVerne1_aes.txt	70	50.94%	57.03%	0/5		

Table B.82: AES - Ant Colony Optimization

Ciphertext	Pop.	Avg. rec.	Max. rec.	100% rec.	Avg. gen.	Avg. eval.
JulesVerne1_aes.txt	20	48.28%	55.47%	0/5		
JulesVerne1_aes.txt	30	51.56%	60.16%	0/5		
JulesVerne1_aes.txt	50	50.16%	55.47%	0/5		
JulesVerne1_aes.txt	70	50.78%	52.34%	0/5		

Table B.83: AES - Artificial Bee Colony Optimization

Ciphertext	Pop.	Avg. rec.	Max. rec.	100% rec.	Avg. gen.	Avg. eval.
JulesVerne1_aes.txt	20	49.84%	51.56%	0/5		
JulesVerne1_aes.txt	30	47.50%	57.03%	0/5		
JulesVerne1_aes.txt	50	52.81%	54.69%	0/5		
JulesVerne1_aes.txt	70	53.28%	57.03%	0/5		

Table B.84: AES - Cuckoo Search

Ciphertext	Pop.	Avg. rec.	Max. rec.	100% rec.	Avg. gen.	Avg. eval.
JulesVerne1_aes.txt	20	50.31%	56.25%	0/5		
JulesVerne1_aes.txt	30	49.69%	55.47%	0/5		
JulesVerne1_aes.txt	50	50.00%	57.81%	0/5		
JulesVerne1_aes.txt	70	53.44%	60.16%	0/5		

Table B.85: SPECK - Simple Genetic Algorithm (m=0.02)

Ciphertext	Pop.	Avg. rec.	Max. rec.	100% rec.	Avg. gen.	Avg. eval.
JulesVerne1_simon.txt	20	50.21%	54.17%	0/5		
JulesVerne1_simon.txt	30	49.17%	54.17%	0/5		
JulesVerne1_simon.txt	50	51.46%	61.46%	0/5		
JulesVerne1_simon.txt	70	49.79%	56.25%	0/5		

Table B.86: SPECK - Simple Genetic Algorithm (m=0.2)

Ciphertext	Pop.	Avg. rec.	Max. rec.	100% rec.	Avg. gen.	Avg. eval.
JulesVerne1_simon.txt	20	50.62%	54.17%	0/5		
JulesVerne1_simon.txt	30	51.88%	60.42%	0/5		
JulesVerne1_simon.txt	50	47.71%	55.21%	0/5		
JulesVerne1_simon.txt	70	50.83%	54.17%	0/5		

Table B.87: SPECK - Simple Genetic Algorithm (m=1.0)

Ciphertext	Pop.	Avg. rec.	Max. rec.	100% rec.	Avg. gen.	Avg. eval.
JulesVerne1_simon.txt	20	55.00%	59.38%	0/5		
JulesVerne1_simon.txt	30	49.38%	52.08%	0/5		
JulesVerne1_simon.txt	50	50.83%	60.42%	0/5		
JulesVerne1_simon.txt	70	51.67%	56.25%	0/5		

Table B.88: SPECK - Custom Genetic Algorithm (m=0.02)

Ciphertext	Pop.	Avg. rec.	Max. rec.	100% rec.	Avg. gen.	Avg. eval.
JulesVerne1_simon.txt	20	52.08%	58.33%	0/5		
JulesVerne1_simon.txt	30	52.29%	57.29%	0/5		
JulesVerne1_simon.txt	50	50.42%	54.17%	0/5		
JulesVerne1_simon.txt	70	49.79%	60.42%	0/5		

Table B.89: SPECK - Custom Genetic Algorithm (m=0.2)

Ciphertext	Pop.	Avg. rec.	Max. rec.	100% rec.	Avg. gen.	Avg. eval.
JulesVerne1_simon.txt	20	52.71%	55.21%	0/5		
JulesVerne1_simon.txt	30	52.29%	61.46%	0/5		
JulesVerne1_simon.txt	50	57.08%	60.42%	0/5		
JulesVerne1_simon.txt	70	45.83%	51.04%	0/5		

Table B.90: SPECK - Custom Genetic Algorithm (m=1.0)

Ciphertext	Pop.	Avg. rec.	Max. rec.	100% rec.	Avg. gen.	Avg. eval.
JulesVerne1_simon.txt	20	51.25%	58.33%	0/5		
JulesVerne1_simon.txt	30	52.08%	60.42%	0/5		
JulesVerne1_simon.txt	50	51.25%	61.46%	0/5		
JulesVerne1_simon.txt	70	51.25%	58.33%	0/5		

Table B.91: SPECK - Self-Adaptive Differential Evolution

Ciphertext	Pop.	Avg. rec.	Max. rec.	100% rec.	Avg. gen.	Avg. eval.
JulesVerne1_simon.txt	20	51.04%	57.29%	0/5		
JulesVerne1_simon.txt	30	48.75%	51.04%	0/5		
JulesVerne1_simon.txt	50	47.92%	55.21%	0/5		
JulesVerne1_simon.txt	70	48.12%	54.17%	0/5		

Table B.92: SPECK - Simulated Annealing

Ciphertext	Avg. rec.	Max. rec.	100% rec.	Avg. eval.
JulesVerne1_simon.txt	50.00%	55.21%	0/5	

Table B.93: SPECK - Particle Swarm Optimization

Ciphertext	Pop.	Avg. rec.	Max. rec.	100% rec.	Avg. gen.	Avg. eval.
JulesVerne1_simon.txt	20	53.75%	56.25%	0/5		
JulesVerne1_simon.txt	30	51.25%	55.21%	0/5		
JulesVerne1_simon.txt	50	48.96%	57.29%	0/5		
JulesVerne1_simon.txt	70	52.71%	56.25%	0/5		

Table B.94: SPECK - Ant Colony Optimization

Ciphertext	Pop.	Avg. rec.	Max. rec.	100% rec.	Avg. gen.	Avg. eval.
JulesVerne1_simon.txt	20	47.92%	56.25%	0/5		
JulesVerne1_simon.txt	30	54.17%	61.46%	0/5		
JulesVerne1_simon.txt	50	48.75%	52.08%	0/5		
JulesVerne1_simon.txt	70	49.58%	54.17%	0/5		

Table B.95: SPECK - Artificial Bee Colony Optimization

Ciphertext	Pop.	Avg. rec.	Max. rec.	100% rec.	Avg. gen.	Avg. eval.
JulesVerne1_simon.txt	20	47.08%	55.21%	0/5		
JulesVerne1_simon.txt	30	53.33%	63.54%	0/5		
JulesVerne1_simon.txt	50	52.92%	61.46%	0/5		
JulesVerne1_simon.txt	70	50.62%	59.38%	0/5		

Table B.96: SPECK - Cuckoo Search

Ciphertext	Pop.	Avg. rec.	Max. rec.	100% rec.	Avg. gen.	Avg. eval.
JulesVerne1_simon.txt	20	56.04%	63.54%	0/5		
JulesVerne1_simon.txt	30	54.38%	67.71%	0/5		
JulesVerne1_simon.txt	50	46.88%	55.21%	0/5		
JulesVerne1_simon.txt	70	49.17%	54.17%	0/5		

Table B.97: RSA Factorization - Simple Genetic Algorithm (m=0.02)

Ciphertext	Pop.	Successful runs	Avg. gen.	Avg. eval.
rsa1.txt	20	6/10	1175.50	23510.00
rsa1.txt	30	5/10	186.60	5598.00
rsa1.txt	50	7/10	81.43	4071.43
rsa1.txt	70	9/10	33.11	2317.78
rsa2.txt	20	2/10	5495.50	109910.00
rsa2.txt	30	5/10	5925.40	177762.00
rsa2.txt	50	2/10	1572.50	78625.00
rsa2.txt	70	8/10	3188.50	223195.00
rsa3.txt	20	2/10	142.50	2850.00
rsa3.txt	30	3/10	6389.67	191690.00
rsa3.txt	70	1/10	2.00	140.00
rsa4.txt	50	1/10	218.00	10900.00
rsa5.txt	50	1/10	2.00	100.00
rsa5.txt	70	1/10	626.00	43820.00

Table B.98: RSA Factorization - Simple Genetic Algorithm (m=0.2)

Ciphertext	Pop.	Successful runs	Avg. gen.	Avg. eval.
rsa1.txt	20	9/10	25.33	506.67
rsa1.txt	30	8/10	20.38	611.25
rsa1.txt	50	8/10	6.25	312.50
rsa1.txt	70	8/10	4.00	280.00
rsa2.txt	20	10/10	3905.50	78110.00
rsa2.txt	30	10/10	1676.20	50286.00
rsa2.txt	50	10/10	866.70	43335.00
rsa2.txt	70	10/10	1648.00	115360.00
rsa3.txt	20	5/10	5257.40	105148.00
rsa3.txt	30	8/10	3319.38	99581.25
rsa3.txt	50	9/10	2812.89	140644.44
rsa3.txt	70	10/10	3281.70	229719.00
rsa4.txt	20	3/10	1003.00	20060.00
rsa4.txt	30	3/10	1568.00	47040.00
rsa4.txt	50	6/10	167.33	8366.67
rsa4.txt	70	6/10	499.17	34941.67
rsa5.txt	20	2/10	2007.00	40140.00
rsa5.txt	30	5/10	6027.80	180834.00
rsa5.txt	50	4/10	6881.75	344087.50
rsa6.txt	30	1/10	16926.00	507780.00
rsa6.txt	50	1/10	488.00	24400.00
rsa6.txt	70	2/10	7400.00	518000.00
rsa7.txt	70	2/10	6112.00	427840.00
rsa8.txt	50	1/10	78.00	3900.00
rsa9.txt	20	2/10	295.00	5900.00
rsa9.txt	50	1/10	13390.00	669500.00
rsa9.txt	70	3/10	1414.33	99003.33
rsa10.txt	50	1/10	1192.00	59600.00

Table B.99: RSA Factorization - Simple Genetic Algorithm (m=1.0)

Ciphertext	Pop.	Successful runs	Avg. gen.	Avg. eval.
rsa1.txt	20	10/10	16.90	338.00
rsa1.txt	30	10/10	14.40	432.00
rsa1.txt	50	10/10	9.60	480.00
rsa1.txt	70	10/10	7.10	497.00
rsa2.txt	20	10/10	18.00	360.00
rsa2.txt	30	10/10	21.50	645.00
rsa2.txt	50	10/10	17.30	865.00
rsa2.txt	70	10/10	5.90	413.00
rsa3.txt	20	10/10	47.50	950.00
rsa3.txt	30	10/10	60.80	1824.00
rsa3.txt	50	10/10	13.40	670.00
rsa3.txt	70	10/10	16.60	1162.00
rsa4.txt	20	10/10	240.30	4806.00
rsa4.txt	30	10/10	124.30	3729.00
rsa4.txt	50	10/10	68.70	3435.00
rsa4.txt	70	10/10	26.10	1827.00
rsa5.txt	20	10/10	2158.20	43164.00
rsa5.txt	30	10/10	933.70	28011.00
rsa5.txt	50	10/10	421.80	21090.00
rsa5.txt	70	10/10	272.10	19047.00
rsa6.txt	20	10/10	2445.70	48914.00
rsa6.txt	30	10/10	2639.20	79176.00
rsa6.txt	50	10/10	1886.60	94330.00
rsa6.txt	70	10/10	573.90	40173.00
rsa7.txt	20	6/10	9566.50	191330.00
rsa7.txt	30	5/10	7221.00	216630.00
rsa7.txt	50	7/10	3910.14	195507.14
rsa7.txt	70	10/10	6349.50	444465.00

Table B.100: RSA Factorization - Simple Genetic Algorithm (m=1.0) (cont.)

Ciphertext	Pop.	Successful runs	Avg. gen.	Avg. eval.
rsa8.txt	20	2/10	4784.00	95680.00
rsa8.txt	30	3/10	14622.67	438680.00
rsa8.txt	50	8/10	6073.00	303650.00
rsa8.txt	70	7/10	9975.57	698290.00
rsa9.txt	20	5/10	7993.20	159864.00
rsa9.txt	30	8/10	4962.50	148875.00
rsa9.txt	50	8/10	5769.88	288493.75
rsa9.txt	70	9/10	6676.11	467327.78
rsa10.txt	30	1/10	18808.00	564240.00
rsa10.txt	50	2/10	10351.00	517550.00
rsa10.txt	70	2/10	8624.50	603715.00
rsa11.txt	20	1/10	8196.00	163920.00
rsa11.txt	30	1/10	12448.00	373440.00
rsa11.txt	50	2/10	8974.00	448700.00
rsa13.txt	70	1/10	726.00	50820.00

Table B.101: RSA Factorization - Custom Genetic Algorithm (m=0.02)

Ciphertext	Pop.	Successful runs	Avg. gen.	Avg. eval.
rsa1.txt	20	3/10	4.33	86.67
rsa1.txt	50	1/10	1.00	50.00
rsa1.txt	70	7/10	4.86	340.00
rsa2.txt	20	3/10	4.00	80.00
rsa2.txt	30	3/10	5.00	150.00
rsa2.txt	50	4/10	2.50	125.00
rsa2.txt	70	5/10	29.40	2058.00
rsa3.txt	30	1/10	1.00	30.00
rsa3.txt	50	2/10	1.50	75.00
rsa3.txt	70	1/10	2.00	140.00
rsa4.txt	50	1/10	1.00	50.00
rsa5.txt	50	1/10	1.00	50.00

Table B.102: RSA Factorization - Custom Genetic Algorithm (m=0.2)

Ciphertext	Pop.	Successful runs	Avg. gen.	Avg. eval.
rsa1.txt	30	4/10	4.25	127.50
rsa1.txt	50	4/10	2.00	100.00
rsa1.txt	70	7/10	2.29	160.00
rsa2.txt	20	2/10	3.50	70.00
rsa2.txt	30	4/10	92.25	2767.50
rsa2.txt	50	4/10	4.25	212.50
rsa2.txt	70	5/10	4.40	308.00
rsa3.txt	50	1/10	4.00	200.00
rsa3.txt	70	3/10	1.67	116.67
rsa5.txt	30	1/10	3.00	90.00

Table B.103: RSA Factorization - Custom Genetic Algorithm (m=1.0)

Ciphertext	Pop.	Successful runs	Avg. gen.	Avg. eval.
rsa1.txt	20	2/10	3.00	60.00
rsa1.txt	30	5/10	5.00	150.00
rsa1.txt	50	4/10	4.75	237.50
rsa1.txt	70	7/10	4.43	310.00
rsa2.txt	20	3/10	1.67	33.33
rsa2.txt	30	3/10	3.00	90.00
rsa2.txt	50	4/10	38.00	1900.00
rsa2.txt	70	5/10	14.60	1022.00
rsa3.txt	30	1/10	4.00	120.00
rsa3.txt	50	2/10	3.00	150.00
rsa3.txt	70	4/10	1.75	122.50
rsa8.txt	30	1/10	5.00	150.00

Table B.104: RSA Factorization - Self-Adaptive Differential Evolution

Ciphertext	Pop.	Successful runs	Avg. gen.	Avg. eval.
rsa1.txt	20	10/10	29.70	594.00
rsa1.txt	30	10/10	27.90	837.00
rsa1.txt	50	10/10	15.30	765.00
rsa1.txt	70	10/10	9.40	658.00
rsa2.txt	20	10/10	57.80	1156.00
rsa2.txt	30	7/10	47.86	1435.71
rsa2.txt	50	10/10	34.40	1720.00
rsa2.txt	70	10/10	47.60	3332.00
rsa3.txt	20	9/10	87.56	1751.11
rsa3.txt	30	8/10	56.12	1683.75
rsa3.txt	50	10/10	48.90	2445.00
rsa3.txt	70	10/10	48.80	3416.00
rsa4.txt	20	5/10	241.60	4832.00
rsa4.txt	30	7/10	382.57	11477.14
rsa4.txt	50	9/10	152.89	7644.44
rsa4.txt	70	9/10	352.00	24640.00
rsa5.txt	20	1/10	197.00	3940.00
rsa5.txt	50	1/10	2417.00	120850.00
rsa5.txt	70	3/10	404.67	28326.67
rsa6.txt	30	2/10	493.50	14805.00
rsa6.txt	50	4/10	1075.00	53750.00
rsa7.txt	50	1/10	76.00	3800.00
rsa9.txt	30	1/10	148.00	4440.00
rsa9.txt	50	4/10	2207.25	110362.50
rsa9.txt	70	2/10	2717.00	190190.00
rsa11.txt	30	1/10	7531.00	225930.00

Table B.105: RSA Factorization - Simulated Annealing

Ciphertext	Successful runs	Avg. eval.
rsa1.txt	3/10	10.33
rsa2.txt	1/10	1.00
rsa3.txt	0/10	
rsa4.txt	0/10	
rsa5.txt	0/10	
rsa6.txt	0/10	
rsa7.txt	0/10	
rsa8.txt	0/10	
rsa9.txt	0/10	
rsa10.txt	0/10	
rsa11.txt	0/10	
rsa12.txt	0/10	
rsa13.txt	0/10	
rsa14.txt	0/10	
rsa15.txt	0/10	
rsa16.txt	0/10	

Table B.106: RSA Factorization - Particle Swarm Optimization

Ciphertext	Pop.	Successful runs	Avg. gen.	Avg. eval.
rsa1.txt	20	7/10	29.29	585.71
rsa1.txt	30	8/10	7.88	236.25
rsa1.txt	50	9/10	10.11	505.56
rsa1.txt	70	9/10	6.67	466.67
rsa2.txt	20	6/10	201.83	4036.67
rsa2.txt	30	5/10	1081.40	32442.00
rsa2.txt	50	8/10	108.00	5400.00
rsa2.txt	70	8/10	89.88	6291.25
rsa3.txt	20	2/10	166.50	3330.00
rsa3.txt	30	4/10	90.00	2700.00
rsa3.txt	50	5/10	129.40	6470.00
rsa3.txt	70	6/10	25.00	1750.00
rsa4.txt	20	1/10	1250.00	25000.00
rsa4.txt	30	3/10	495.00	14850.00
rsa4.txt	50	8/10	1080.25	54012.50
rsa4.txt	70	7/10	541.57	37910.00
rsa5.txt	20	1/10	34.00	680.00
rsa5.txt	50	1/10	28.00	1400.00
rsa5.txt	70	1/10	86.00	6020.00
rsa9.txt	50	1/10	1379.00	68950.00

Table B.107: RSA Factorization - Ant Colony Optimization

Ciphertext	Pop.	Successful runs	Avg. gen.	Avg. eval.
rsa1.txt	20	10/10	50.50	990.00
rsa1.txt	30	10/10	25.70	741.00
rsa1.txt	50	10/10	11.50	525.00
rsa1.txt	70	10/10	16.60	1092.00
rsa2.txt	20	10/10	26.70	514.00
rsa2.txt	30	10/10	14.30	399.00
rsa2.txt	50	10/10	10.10	455.00
rsa2.txt	70	10/10	7.20	434.00
rsa3.txt	20	10/10	50.20	984.00
rsa3.txt	30	10/10	31.80	924.00
rsa3.txt	50	10/10	25.70	1235.00
rsa3.txt	70	10/10	14.90	973.00
rsa4.txt	20	10/10	201.80	4016.00
rsa4.txt	30	10/10	331.70	9921.00
rsa4.txt	50	10/10	100.30	4965.00
rsa4.txt	70	10/10	98.60	6832.00
rsa5.txt	20	10/10	1618.50	32350.00
rsa5.txt	30	10/10	946.60	28368.00
rsa5.txt	50	10/10	491.60	24530.00
rsa5.txt	70	10/10	765.60	53522.00
rsa6.txt	20	7/10	2948.14	58942.86
rsa6.txt	30	8/10	3625.38	108731.25
rsa6.txt	50	10/10	2904.50	145175.00
rsa6.txt	70	10/10	4618.30	323211.00
rsa7.txt	20	7/10	8268.57	165351.43
rsa7.txt	30	7/10	12164.71	364911.43
rsa7.txt	50	10/10	6274.30	313665.00
rsa7.txt	70	10/10	5108.00	357490.00

Table B.108: RSA Factorization - Ant Colony Optimization (cont.)

Ciphertext	Pop.	Successful runs	Avg. gen.	Avg. eval.
rsa8.txt	20	4/10	10345.75	206895.00
rsa8.txt	30	5/10	7086.40	212562.00
rsa8.txt	50	6/10	8196.00	409750.00
rsa8.txt	70	8/10	7374.62	516153.75
rsa9.txt	20	7/10	6528.14	130542.86
rsa9.txt	30	8/10	7420.50	222585.00
rsa9.txt	50	8/10	5984.00	299150.00
rsa9.txt	70	9/10	3740.11	261737.78
rsa10.txt	20	1/10	2313.00	46240.00
rsa10.txt	30	2/10	9977.00	299280.00
rsa10.txt	50	4/10	10147.50	507325.00
rsa10.txt	70	4/10	5629.75	394012.50
rsa11.txt	20	2/10	3911.50	78210.00
rsa11.txt	50	2/10	10059.50	502925.00
rsa11.txt	70	2/10	7398.50	517825.00
rsa12.txt	70	2/10	6157.00	430920.00
rsa13.txt	20	1/10	8261.00	165200.00

Table B.109: RSA Factorization - Artificial Bee Colony Optimization

Ciphertext	Pop.	Successful runs	Avg. gen.	Avg. eval.
rsa1.txt	20	10/10	54.50	2180.00
rsa1.txt	30	10/10	52.00	3120.00
rsa1.txt	50	10/10	51.20	5120.00
rsa1.txt	70	10/10	23.60	3304.00
rsa2.txt	20	10/10	163.20	6528.00
rsa2.txt	30	10/10	98.40	5904.00
rsa2.txt	50	10/10	33.40	3340.00
rsa2.txt	70	10/10	67.10	9394.00
rsa3.txt	20	10/10	199.20	7968.00
rsa3.txt	30	10/10	219.70	13182.00
rsa3.txt	50	10/10	78.20	7820.00
rsa3.txt	70	10/10	58.50	8190.00
rsa4.txt	20	10/10	512.20	20488.00
rsa4.txt	30	10/10	245.90	14754.00
rsa4.txt	50	10/10	565.50	56550.00
rsa4.txt	70	10/10	260.00	36400.00
rsa5.txt	20	10/10	4926.00	197040.00
rsa5.txt	30	10/10	1998.40	119904.00
rsa5.txt	50	10/10	5607.00	560700.00
rsa5.txt	70	10/10	2068.20	289548.00
rsa6.txt	20	10/10	5906.40	236256.00
rsa6.txt	30	8/10	5155.12	309307.50
rsa6.txt	50	10/10	2652.70	265270.00
rsa6.txt	70	10/10	4312.10	603694.00
rsa7.txt	20	6/10	8903.00	356120.00
rsa7.txt	50	4/10	12813.50	1281350.00
rsa7.txt	70	3/10	9742.33	1363926.67
rsa8.txt	20	4/10	11256.75	450270.00
rsa8.txt	50	1/10	17644.00	1764400.00
rsa8.txt	70	3/10	7309.00	1023260.00

Table B.110: RSA Factorization - Artificial Bee Colony Optimization (cont.)

Ciphertext	Pop.	Successful runs	Avg. gen.	Avg. eval.
rsa9.txt	20	1/10	16737.00	669480.00
rsa9.txt	30	3/10	7145.67	428740.00
rsa9.txt	50	5/10	13029.20	1302920.00
rsa9.txt	70	6/10	7256.00	1015840.00
rsa10.txt	50	1/10	6717.00	671700.00
rsa10.txt	70	1/10	10333.00	1446620.00
rsa11.txt	50	1/10	12720.00	1272000.00

Table B.111: RSA Factorization - Cuckoo Search

Ciphertext	Pop.	Successful runs	Avg. gen.	Avg. eval.
rsa1.txt	20	8/10	22.62	452.50
rsa1.txt	30	7/10	23.71	711.43
rsa1.txt	50	10/10	34.20	1710.00
rsa1.txt	70	10/10	16.70	1169.00
rsa2.txt	20	6/10	72.50	1450.00
rsa2.txt	30	3/10	441.00	13230.00
rsa2.txt	50	7/10	6.29	314.29
rsa2.txt	70	8/10	9.12	638.75
rsa3.txt	20	1/10	40.00	800.00
rsa3.txt	30	3/10	141.67	4250.00
rsa3.txt	50	6/10	110.17	5508.33
rsa3.txt	70	2/10	17.50	1225.00
rsa4.txt	30	1/10	194.00	5820.00
rsa4.txt	50	3/10	104.67	5233.33
rsa4.txt	70	1/10	1.00	70.00
rsa6.txt	20	1/10	1.00	20.00