Genetic Algorithm Stream Cipher Key Generation Using NIST Functions

By

Hunter Leggett

December, 2022

Director of Thesis: Dr. Te-Shun Chou

Major Department: Department of Technology Systems

Abstract

Stream ciphers are beneficial because of their efficiency, speed, and low resource utilization. However, stream ciphers are vulnerable to many attacks if they do not use strong keys for encryption and decryption. Thus, one way to increase the security of stream ciphers is to improve the key generation algorithm. This study sought to evaluate the keys produced by a genetic algorithm stream cipher when individual and combinations of fitness functions are used. Furthermore, this study identified which fitness function is the best for a specific scenario. For the genetic algorithm, the fitness tests are thirteen of the tests defined in NIST SP 800-22rla. The thirteen different fitness functions were inputted into the genetic algorithm stream cipher one at a time. Next, 50 total keys of varying bit sizes were generated. These keys were evaluated by using the Hamming distance between the keys and time that it took for key generation. After each individual fitness function was evaluated, two combinations of five tests were created and used as a single fitness function. The two combinations were the best performing NIST functions for Hamming distance and time for 256-bit keys. Sensitivity analysis was then performed to find the

best possible combination of the NIST functions. Based on the results, using different individual functions or a combination of functions as a fitness functions changed the Hamming distance between the keys and the time that it takes to generate a key. Furthermore, using sensitivity analysis results for the top two, three, four, and five combinations for Hamming distance and time, prediction equations were created and used to predict values for other combinations and key sizes.

Keywords: Cryptography, genetic algorithm, Hamming distance, cybersecurity, stream cipher

Genetic Algorithm Stream Cipher Key Generation Using NIST Functions

A Thesis

Presented to the Faculty of the Department of Technology Systems

East Carolina University

In Partial Fulfillment of the Requirements for the Degree

Master's of Science in Network Technology with a Concentration in Information Security

By

Hunter W. Leggett

December, 2022

Director of Thesis: Te-Shun Chou, PhD

Thesis Committee Members:

Rui Wu, PhD

Biwu Yang, PhD

Genetic Algorithm Stream Cipher Key Generation Using NIST Functions

By

Hunter Leggett

APPROVED BY:

Director of Thesis _____

Te-Shun Chou, PhD

Committee Member _____

Rui Wu, PhD

Committee Member _____

Biwu Yang, PhD

Chair of the Department of Technology Systems _____

Tijjani (TJ) Mohammed, PhD

Interim Dean of the Graduate School _____

Kathleen T Cox, PhD

Table of Contents

List of Tables

List of Figures

Chapter 1. Introduction

Cryptography is the use of techniques to secure data in rest and transit. The two main cryptographic techniques are asymmetric encryption and symmetric encryption. Symmetric encryption first appeared in the 1970s and uses a single key for the encryption and decryption of plaintext (Stallings, 2017). Symmetric encryption uses block ciphers, stream ciphers, or a combination of a block and stream cipher to perform the encryption process.

Block ciphers encrypt the plaintext one block of plaintext at a time. On the other hand, a stream cipher uses a generator to generate a keystream that the original text will XOR against (Lalar & Nahta, 2016). Stream ciphers offer many benefits over the use of block ciphers. For example, stream ciphers tend to be faster and more suited for data stream encryption than block ciphers (Stallings, 2017). Furthermore, stream ciphers tend to use fewer resources than block ciphers.

Unfortunately, many attacks are very successful at breaking stream ciphers. These attacks use various techniques to find the key used by the stream cipher and recover the plaintext whether it is at rest, in use, or in transit (Jiao et al., 2020). One proposed way to prevent these attacks is to combine stream ciphers with machine learning.

Machine learning allows computers to learn a process that they were not originally designed to do. Even though machine learning has been around since the 1960s, the use of machine learning with symmetric encryption is relatively recent. Several articles have been written to describe potential machine learning block and stream ciphers (Ali, 2013; Ding et al., 2021; Fadil et al., 2014; Goyat, 2012; Guo et al., 1999; Krishna et al., 2018; Kumar & Chatterjee, 2016; Lian, 2009; Long, 2012; Nazeer et al., 2018; Noura et al., 2015; Sakr et al., 2022; Sindhuja & Pramela, 2014; Som et al., 2011; Sudeepa et al., 2020; Tsai & Chou, 2021).

One of the primary machine learning algorithms used for these ciphers is genetic algorithms. Genetic algorithms use the theory of natural selection to select the best solution for a given problem. For example, for a stream cipher, the genetic algorithm can be used to get the best possible key to XOR with the plaintext (Krishna et al., 2018; Kumar & Chatterjee, 2016).

Stream ciphers are still predominantly used because of their low resource requirement and efficiency. Therefore, increasing the security of stream ciphers will increase data security now and in the future. A way to increase their security is to increase the security of the key generation algorithm. A more secure key generation algorithm would heighten the difficulty that it takes for an attacker to break the key. This surge in difficulty could very well act as a deterrent for attackers attempting to use attacks against the cipher.

Even though, there have been advancements in making a cryptographically strong genetic algorithm stream cipher. The vast majority of the current genetic algorithm stream ciphers use coefficients of correlation and Shannon's entropy for fitness functions. Unfortunately, flaws have been discovered in these functions (Tsai & Chou, 2021). Therefore, different fitness functions need to be tested to determine their effect on a genetic algorithm stream cipher.

The desired genetic algorithm stream cipher would use a genetic algorithm to generate a random key stream. The fitness function for this genetic algorithm would choose the key stream that had the most randomness. The most random key stream is chosen because it would be the most cryptographically secure. The chosen keystream would be used to encrypt and decrypt the data. Overall, this genetic algorithm stream cipher should optimize efficiency and security. Unfortunately, the current genetic algorithm stream ciphers tend to use fitness functions for key selection that are shown to have flaws (Tsai & Cho, 2021). Finding a better fitness function will be paramount to the success of future genetic algorithm stream ciphers.

1.1. Statement of the Problem

The problem of this research was to determine how different individual fitness functions and combinations of fitness functions affect the key generation for a genetic algorithm stream cipher as well as to predict the Hamming distance and average time for key generation for sensitivity analysis.

1.2. Research Questions and Hypotheses

- RQ: What effect do certain fitness function functions have on the performance of a genetic algorithm stream cipher?
    - HO1: Certain fitness functions will cause the genetic algorithm stream cipher to generate more random keys than other fitness functions.
    - HO2: Certain fitness functions will cause the genetic algorithm stream cipher to generate keys faster than other fitness functions.
- RQ: What effect does combining fitness functions have on the performance of a genetic algorithm stream cipher?
    - HO1: The combination of multiple functions as a single fitness function will improve the keys that are generated by the genetic algorithm stream cipher.
    - HO2: Certain combinations of functions as fitness functions will generate more random keys than other combinations of functions.
    - HO3: Certain combinations of functions as fitness functions will generate keys faster than other combinations of functions.
- RQ: Could the Hamming distance and time results for the sensitivity analysis of the top x Hamming distance and time performers be predicted?

    - HO1: The Hamming distance and average time results for different amounts of functions in a combination could be accurately predicted using regression.
- RQ: Could the Hamming distance and time results for the sensitivity analysis of the top two Hamming distance and time performers be predicted for each key size?
    - HO1: The Hamming distance and average time results for differing key sizes for the top two functions in terms of average Hamming distance and average time for 256-bit keys could be accurately predicted using regression.

1.3. Research Objectives

The objective of this study was five-fold: (a) To evaluate the keys produced by each genetic algorithm stream cipher, (b) To evaluate how combinations of fitness functions affect the genetic algorithm stream cipher, (c) To identify which fitness function or combination of fitness functions is the best for a given scenario, (d) To predict the Hamming distance and time results for the sensitivity analysis if a certain number of functions are used in the combination, and (e) To predict the Hamming distance and time results for the sensitivity analysis if a certain key size is used.

1.4. Significance of the Study

This research provided more insight into how different fitness functions affect the key generation for the genetic algorithm stream cipher to create a more cryptographically secure stream cipher.

1.5. Assumptions

The following assumptions were made when experimenting: (a) The NIST SP 800-22r1a tests would work if minimum bit recommendations in the guidelines were ignored, as long as, the mathematical requirements were met for the tests, (b) The genetic algorithm used for the

stream cipher works properly, (c) Plotting the key sizes with hamming distances and time would be linear for each fitness function, (d) Using combinations of the top functions in terms of Hamming distance and time will perform better than randomly selecting the functions for the combinations, (e) The top five functions for time for the 256-bit key size would not conflict with each other when combined, this also applies for the top five hamming distance function combination, and (f) Using the top two, three, four, and five functions would perform better then using a random group of functions.

1.6. Limitations

The limitations of this experiment were: (a) Only one computer was used to test the algorithms, (b) Parallel computing was not used for the sensitivity analysis, (c) The sensitivity analysis was only run once for each weight combination, (d) Only four weights were used for the sensitivity analysis, and (e) Only five fitness functions were used for the sensitivity analysis. All of these limitations were attributed to the resource constraints of the experiment.

1.7. Terminology

- Ciphertext: the encrypted plaintext.
- Combination: a grouping of NIST functions.
- Crossover: a process where an existing solution passes on its characteristics to a new solution.
- Elite Ratio: the number of elites in each population.
- Exclusive or: also known as XOR, a logical operator that is true when one of the operands is true and false when both operands are true or false.
- Fitness Function: an objective function that is used as the selection criteria for the best solution in the genetic algorithm.

- Genetic Algorithm: a machine learning algorithm that is based on the theory of natural selection.
- Hamming Distance: the number of bits that are different between two binary strings.
- Iterations: the number of populations generated for testing by the genetic algorithm. The populations are each generated one after another.
- Key Size: the length of the key.
- Key Stream: the continuous generation of keys to encrypt the total length of the plaintext.
- Key: the generated string that is used by the stream cipher to encrypt the plaintext.
- Machine Learning: a field of study where computers learn to do tasks that they were not originally programmed to do.
- Mutation: a process in the genetic algorithm where one bit is randomly replaced by a random value.
- NIST SP 800-22r1a: NIST Special Publication 800-22 revision 1a is a technical document that describes a statistical test suite for random and pseudorandom number generators.
- Plaintext: the original unencrypted text
- Population Size: the number of solutions in each iteration.
- Pseudorandom Number Generator: a generator that appears to be random.
- P-value: the probability that an outcome occurs by chance. This value is used to determine if the null hypothesis is rejected or not rejected.
- Random Number Generator: a generator that creates truly random strings.
- Sensitivity Analysis: a methodology that evaluates how different inputs can affect the output of an algorithm to determine with inputs can give the best output.

- Stream Cipher: a cipher that encrypts the plaintext one bit at a time through the Exclusive or process.
- Symmetric Cryptography: a form of encryption that uses a single key to encrypt and decrypt plaintext.

Chapter 2. Literature Review

The section is organized as follows. The first section details articles about machine learning. The second section details articles about stream ciphers. The third section details articles about both stream ciphers and machine learning. Finally, the fourth section will discuss the gaps in the research and how this study will fill those gaps.

2.1. Machine Learning

Several articles detail the complicated history of machine learning. Fradkov (2020) details the history of modern-day machine learning. The origin of the current form of machine learning is commonly associated with psychologist Frank Rosenblatt and his work on a machine that could recognize the letters of the alphabet (Fradkov, 2020). After Rosenblatt, machine learning started to gain steam in the 1960s with deterministic and stochastic approaches (Fradkov, 2020). However, after this, not much work was achieved in machine learning until the twenty-first century due to the limitations that computers faced.

On the other hand, Plasek (2016) details how difficult it is to write a comprehensive history of machine learning. Regardless, Alzubi et al. (2018) gave further details on the history of machine learning. According to Alzubi et al. (2018), the origin of machine learning can be traced back further to Alan Turning and the Turning Test in the early 1950s. Additionally, Alzubi et al. (2018) described the generic model of machine learning. This model has six phases which can be classified as: collection and preparation of data, feature selection, choice of algorithm, selection of models and parameters, training, and performance evaluation (Alzubi et al., 2018). From this generic model, several types of machine learning algorithms have been created to accomplish various tasks. Each of these algorithms is trained through one of the types of learning.

According to Sah (2020), the types of learning include supervised learning, unsupervised learning, semi-supervised learning, reinforcement learning, self-supervised learning, self-taught learning, multi-task learning, active learning, online learning, transfer learning, federated learning, ensemble learning, adversarial learning, meta learning, targeted learning, concept learning, Bayesian learning, analytical learning, multi-modal learning, deep learning, and curriculum learning.

Mahesh (2020) and Pandy et al. (2019) take a deeper look into the types of machine learning algorithms. Mahesh (2020) provides a review of the algorithms classified by the types of learning that they use. According to Mahesh (2020), a few algorithms that use supervised learning include the Decision tree and Naive Bayes. Moreover, some algorithms that use unsupervised learning include K-Means clustering and Principal component analysis. Mahesh (2020) continues to describe a few more algorithms before discussing neural networks. Pandey et al. (2019) describe algorithms that use supervised learning, unsupervised learning, and reinforcement learning. According to Pandey et al. (2019), algorithms that use supervised learning can be classified further into classification and regression methods. Classification methods are used to solve discrete-value problems, on the other hand, regression methods are used to make decisions (Pandey et al., 2019).

Jordan & Mitchell (2015) analyze the current trends, perspectives, and prospects in machine learning. According to the article, machine learning has become a multi-disciplinary field of study that has had many advances in recent years. The most common form of machine learning methods that have been used are supervised learning methods. In supervised learning methods, one of the areas where a lot of progress has been attained is deep learning. Even with all of the progress in machine learning, there is still a lot of room for growth and opportunity.

Sarker (2021) agrees with this in discussing the real-world applications of different types of machine learning, as well as directions for the research of machine learning. However, Sarker (2021) details that a lot of this growth depends on data that is "good" and diverse learning algorithms. Good data is defined as data that is representative, high quality, relevant, and can be sufficiently trained (Sarker, 2021).

One of the first overviews of genetic algorithms came in 1988 (De Jong, 1988). This article details many of the early advancements in genetic algorithms and refers to it as an area worth studying. The article discusses that genetic algorithms are good for specific situations; however, they do need a lot of data, between 500-1000 samples to truly be effective (De Jong, 1988). According to Katoch et al. (2021), the classic genetic algorithm was designed as an optimization algorithm inspired by natural selection. The algorithm divides a population into chromosomes randomly. The fitness of each chromosome is calculated by the fitness function. Two chromosomes are chosen, and they undergo crossover and mutation until they produce an offspring that meets the best solution given (Katoch et a., 2021). In this process, several operators are used. The four operators are selection, encoding, crossover, and mutation (Katoch et al., 2021). Genetic algorithms are classified as either binary or real depending upon the data being entered into them.

Nierhaus (2009) provides further details on the construction of genetic algorithms in the chapter "Genetic Algorithms". In this chapter, the history of genetic algorithms is covered as well as the creation of genetic algorithms. This chapter also details some of the limitations that genetic algorithms face. According to the chapter, if the algorithm is not set up carefully, then the genetic algorithm may not yield correct results because it could undertake a generation task,

instead of an evaluation task. Furthermore, if more than one fitness function is used, then the functions could end up hurting each other in trying to determine which is the best solution.

Finally, Verma & Kumar (2014) provide another analysis of genetic algorithms. The article provides a short background on genetic algorithms as well as pseudo-code for a simple genetic algorithm. Furthermore, the article provides examples of problems that can be solved with genetic algorithms along with an example of a genetic algorithm solving the Knapsack Problem (Verma & Kumar, 2014).

2.2. Stream Ciphers

In cryptography, the two main forms of algorithms are symmetric key algorithms and asymmetric key algorithms. Chandra et al. (2014b) provided an analysis of the differences between symmetric and asymmetric cryptography. In symmetric key algorithms, the sender and receiver share a single secret key (Chandra et al., 2014b). In asymmetric key algorithms, two keys are used: a public key and a private key (Chandra et al., 2014b). Chandra et al. (2014a) provide further information about symmetric algorithms. This article gives a short description of common symmetric algorithms, such as AES, RC4, and Blowfish, and proposed symmetric methods. Additionally, the book *Cryptography and Network Security* provides a more thorough analysis of asymmetric and symmetric algorithms. This book also provides examples of the code for algorithms such as RC4 and discusses the number theory behind the creation of the algorithms.

Stream ciphers are encryption algorithms that belong to the family of symmetric key algorithms. Lalar & Nahta (2016) provides an analysis of the stream cipher methodology as well as two attacks that can be used on stream ciphers. They describe stream ciphers that use a random number generator, one-time pad, and shift register. Gorbenko et al. (2017) analyze the

currently used stream ciphers. Another form of stream ciphers is lightweight stream ciphers. These stream ciphers are designed to work with embedded devices or devices that do not have a lot of resources (Manifavas et al., 2016). Manifavas et al. (2016) provide a survey of the current trends with these types of stream ciphers.

Jiao et al. (2020) provide an analysis of past and current stream cipher designs. Their article describes stream ciphers by the structure that they use along with the cryptanalysis of the ciphers (Jiao et al., 2020). The common cryptanalysis methods that are listed are exhaustive search, time-memory trade-off, algebraic, correlation and linear, guess and determine, resynchronization, differential, cube, and fault attacks (Jiao et al., 2020, p. 22). Exhaustive search, or brute force, attacks try every key to guess the correct key. Time-memory trade-off attacks are two-phase exhaustive search attacks, where a table of keys is generated by using the general structure of the stream cipher, and then the keys are used on real data (Jiao et al., 2020, p. 23). Algebraic and correlation attacks are specifically used on linear-feedback shift register stream ciphers. Linear attacks rely on linear relations in the keystream (Jiao et al., 2020, p. 23). Guess-and-determine attacks determine the state elements of the cipher by guessing some of the elements. Differential attacks use the biased distribution of keys generated by a specific initialization vector. Resynchronization attacks are similar to differential attacks. Cube attacks use a "low degree polynomial representation of a single output bit" to determine the key (Jiao et al., 2020, p. 23). Fault attacks use bit flipping faults to decrypt the message. Today, stream ciphers are much more powerful, efficient, and resistant to attacks.

Pseudo-random number generators produce a seemingly random string that can be used to generate a secret key for a stream cipher. Many pseudo-random number generators are available, as shown by Deng & Bowman (2017) and Zeng et al. (1991). However, to be

considered secure, they must meet three criteria: the keystream must be able to accommodate all of the plaintexts, the keystream must be easy to generate, and the keystream must be hard to predict (Zeng et al., 1991). To test the security of these generators, fifteen tests were developed by the National Institute of Standards and Technologies (Bassam III et al., 2010). Each of these tests evaluates certain aspects of the keystreams generated by the pseudo-random number generators. Furthermore, another test that can be used is the Hamming distance between the streams, which is the binary bits that differ between the two keystreams (Black, 2006).

2.3. Machine Learning and Stream Ciphers

Many approaches have been used to combine machine learning and symmetric encryption. One approach to combining machine learning and symmetric encryption is through the use of neural networks. One of the first neural network stream ciphers was discussed by Guo et al. (1999). They propose a symmetric cipher based using an Over-storage Hopfield Neural Network (OHNN). The proposed cipher scheme randomly selects a permutation matrix and a coding matrix. These matrices will be used as the secret key pairs and sent to the other user using public-key cryptography. Then a new synaptic matrix and attracts are generated using the permutation matrix. Next, the coding matrix and attractors will map the plaintext into a coded plaintext. Finally, a pseudorandom number generator will generate a random amount of 0s and 1s which will be the initial state for the OHNN process (Guo et al., 1999). To decrypt, the received permutation matrix would be used to generate the synaptic matrix. Then the coded plaintext would be regenerated and decoded into the plaintext.

This work led to the development of another cipher by Long (2012). This article used a Hopfield Neural Network (HNN) and works in three steps. In the first step, a randomly selected matrix and secret keys are selected by a linear feedback shift register (LFSR) random number

generator. In the second step, the LFSR output is inputted into the HNN, then the input will converge to one of the network attractors (Long, 2012). In the final step, the attractors are sorted into groups with varying amounts of 1's and 0's and one of the LFSR outputs is chosen based on the attractors (Long, 2012).

Furthermore, chaotic neural networks (CNNs) and artificial neural networks (ANNs) have been used. Lian (2009) suggests a CNN-based block cipher. To encrypt data, the plaintext is entered into the chaotic neuron layer and then transferred to the linear neuron layer. This is repeated until the data is secure. To decrypt the data, the matrices and functions are inverted with similar parameters to the encryption function (Lian, 2009). Unfortunately, this cipher is not the most efficient. Another CNN cipher was developed by Fadil et al. (2014). This cipher uses discrete cosine transform (DCT) with a CNN to encrypt. However, this cipher was only developed for use with video transmissions.

Another example of an ANN cipher was developed by Noura et al. (2015). The cipher takes in an original message and a secret key that can be 128, 256, or 512 bits and is created through dynamic key generation (Noura et al., 2015). Another cipher that was developed for image encryption was developed by Ding et al. (2021). Their cipher uses deep learning to form a key generation network to develop a key that is XORed against the image. The key for the proposed system has a strength of $(2^8)^{196608}$ (Ding et al., 2021). Furthermore, because of the randomness, the system would not generate the same key twice.

An alternate approach to combining machine learning and symmetric encryption is through the use of genetic algorithms. To help show how this could be done an article by Ali (2013) details a simplistic form of a genetic algorithm. Furthermore, Goyat (2012) provides an analysis of some of the current methods that apply genetic algorithms to symmetric encryption.

Some novel approaches to symmetric encryption have been proposed using genetic algorithms. For example, Nazeer et al. (2018) propose an encryption algorithm that works in three phases. In the first phase, key generation, random keys are entered into the function and crossovered and mutated. The best key is chosen by using Shannon entropy. In the second phase, the original text is diffused by using crossover and mutation. In the third and final phase, the key is XORed with the diffused text.

Another novel approach was proposed by Som et al. (2011). The proposed cipher would first XOR the plaintext with user input or two random numbers. Next, the new text would undergo transformation. This transformed text would be XORed again by a key created by choosing two index positions randomly from 8-bit blocks made from the transformed text. This level 1 cipher text is entered into the genetic algorithm. Once within the algorithm a single-point crossover function. The fitness is determined by which child is most different from the parents. One chromosome is chosen and reversed and added to the final cipher text until all are produced. The key is created using the crossover points, block amount, and length of blocks. A third novel approach was proposed by Sindhuja & Pramela (2014). In their approach, both the plaintext and a key are inputted into the cipher. The plaintext and key will then undergo matrix addition, substitution, and crossover before the ciphertext is outputted.

Additionally, articles have used genetic algorithms to improve current symmetric ciphers. For example, Tsai & Chou (2021) propose a new variation of the data encryption algorithm (DES) referred to as GADES. The proposed cipher would generate keys using a normalized linear programming model for the fitness function. This function will choose the key with the greatest distance from the plaintext that generates it. The cipher uses two main steps. In step one, the plaintext and seven keys are entered into the algorithm. The algorithm would run and choose

the key based on the fitness function. The chosen key and plaintext are then XORed to get the ciphertext.

Sakr et al. (2022) proposed a genetic algorithm cipher for amino acid encryption. The proposed cipher uses three phases. In the first phase, the initial population is generated by an RNG that chooses a 1 or 0. Next, the fitness function, the run test of randomness, is calculated on the initial input. If it fails the test, then it undergoes crossover and mutation until it passes the test. The selected key that passes the test is inputted into the Needleman-Wunsch (NW) algorithm. The NW algorithm calculates a key using the similarities between the chosen key and all of the keys from the last generation. Finally, the Playfair cipher is used to encrypt the data.

Furthermore, genetic algorithms have been applied to key generators and stream ciphers. For key generation, one approach by Krishna et al. (2018) uses a mutated Huffman tree coding algorithm. The key is generated using Non-Dominated Sorting Genetic Algorithm II (NSGA-II) in the bi-objective optimization framework and Improved Modified Harmony Search + Differential Evolution (IMHS+DE) in the single objective framework. The mutated Huffman tree coding algorithm is then used to encode the plaintext. Another approach to key generation is proposed by Sudeepa et al. (2020). The focus of their work was to increase the length of linear feedback shift registers (LSFRs) by using a genetic algorithm.

Moving forward, a stream cipher using a genetic algorithm was proposed by Kumar & Chatterjee (2016). For this algorithm, random keys are generated. The random keys are put into the genetic algorithm to start creating populations. Two parents are selected from the random keys; they are crossed over and then the threshold is checked. After the crossover, the child is changed by the mutation function. The fitness of the mutated key is found using Shannon entropy, chi-square, and the coefficient of autocorrelation.

Chapter 3. Methodology

To determine how different fitness functions affect a genetic algorithm stream cipher, a group of fitness functions had to be selected. The selected functions are the statistical testing functions defined in NIST SP 800-22r1a. These functions were chosen because they evaluate the randomness of a given binary sequence (Bassam et al., 2010). The given binary sequence is evaluated on a scale of 0 to 1, where 0 is not random and 1 is perfect randomness. This value between 0 and 1 is referred to as the P-value, or tail probability.

3.1. Fitness Functions

Thirteen of the fifteen tests were used in the experiment. The two excluded tests were the linear complexity test and the overlapping template matching test. The linear complexity test was not used because it only applies to linear feedback shift registers. The overlapping template matching test was not used because all of the mathematical requirements could not be met for smaller input sizes without getting very small P-values. The thirteen used tests are frequency (monobit) test, frequency test within a block, runs test, test for the longest run of ones in a block, binary matrix rank test, discrete Fourier transform (spectral) test, non-overlapping template matching test, Maurer's "Universal Statistical" test, serial test, approximate entropy test, cumulative sums (Cusum) test, random excursions test, and random excursions variant test. Bassam et al. (2010) define these thirteen tests as follows:

- The frequency (monobit) test evaluates the proportion of zeros and ones in the given binary string. If the number of zeros and ones is about the same, then the binary string would pass the test.

- The frequency test within a block test evaluates the frequency of ones in a block of a certain length. If the numbers of ones are approximately the length of the block divided by two, then the binary string would pass the test.
- The runs test evaluates the runs of ones and zeros. To pass the test, the binary string cannot have too little and/or too many bits before it switches between ones and zeros.
- The test for the longest run of ones in a block determines if the longest run of ones in a block is what should be expected in a random sequence. If there are too many clusters of ones in the blocks, then the binary string would fail the test.
- The binary matrix rank test evaluates the rank of disjoint sub-matrices of a sequence. If the linear dependences of the matrices deviate too much from the theoretical value, then the binary string fails the test.
- The discrete Fourier transform (spectral) test determines if the repetitive patterns that are close to each other deviate from an assumption of randomness. This is tested by seeing if the number of peaks that exceed the 95% threshold is more or less than 5%. If the number is different from 5%, then the binary string would fail the test.
- The non-overlapping template matching test determines if there are too many appearances of a certain pattern. If there are too many appearances, then the binary string fails the test.
- Maurer's "Universal Statistical" test evaluates the number of bits between matching patterns. The number of bits between matching patterns relates to the length of a compressed sequence. If the sequence is too compressible, then the binary string fails the test.

- The serial test determines if the number of occurrences of the 2^m m-bit overlapping patterns is as expected in uniformity. If the overlapping patterns do not appear uniformly, then the binary string fails the test.
- The approximate entropy test evaluates the frequency of all overlapping m-bit patterns. If the frequency of consecutive overlapping m-bit blocks deviates too much from what is expected, then the binary string fails the test.
- The cumulative sums (Cusum) test determines if the cumulative sum of partial sequences in a test sequence is too large or small relative to what is expected. If the number of excursions from the cumulative sum is not close to zero, then the binary string fails the test.
- The random excursions test evaluates the number of cycles having a certain number of visits to a state in a cumulative sum walk to see if it is as expected. If the number of visits to a state deviate from the expected value, then the binary string fails the test.
- The random excursions variant test determines the number of times that a state occurs in a cumulative sum walk. Similar to the random excursions test, if the number of visits to a state deviate from the expected value, then the binary string fails the test. The difference between this test and random excursions is that this test consists of 18 tests rather than 8 tests.

3.2. Experimental Environment

The Python genetic algorithm that was used for this experiment was developed by Solgi (2020). Unfortunately, the original NIST tests were written in the programming language C. For the tests to work best with the genetic algorithm, they had to be converted to the programming language Python. The tests were each converted to Python by using the original C code. The

code developed by Pasqualini (2021) was used as a reference to ensure that the tests were correctly translated. The tests were modeled as closely to the original C tests as possible. The constants for each test were chosen by using the mathematical constraints and formulas given in NIST SP 800-22r1a. Each of the thirteen tests used for this experiment can be viewed in Appendix A.

Additionally, a Windows 10 desktop computer was used to perform the tests. This computer had 16 gigabytes of RAM, a 2 terabyte hybrid hard drive, a 3.70 gigahertz i7-8700k Intel processor, and a NIVIDA GeForce GTX 1080Ti graphics card. The tests were run using an Ubuntu Linux terminal through the Windows Subsystems for Linux (WSL). While the tests were running, only the Excel file used to record the data was open.

3.3. Individual Fitness Function Testing

Once the tests were converted into Python, they were inserted into the Python genetic algorithm one at a time as the fitness function. For the tests, the genetic algorithm ran for 500 iterations and used a population size of 17. 500 iterations were determined to be enough time for each of the tests to converge. Convergence can be visualized on a graph as shown in Figure 1.

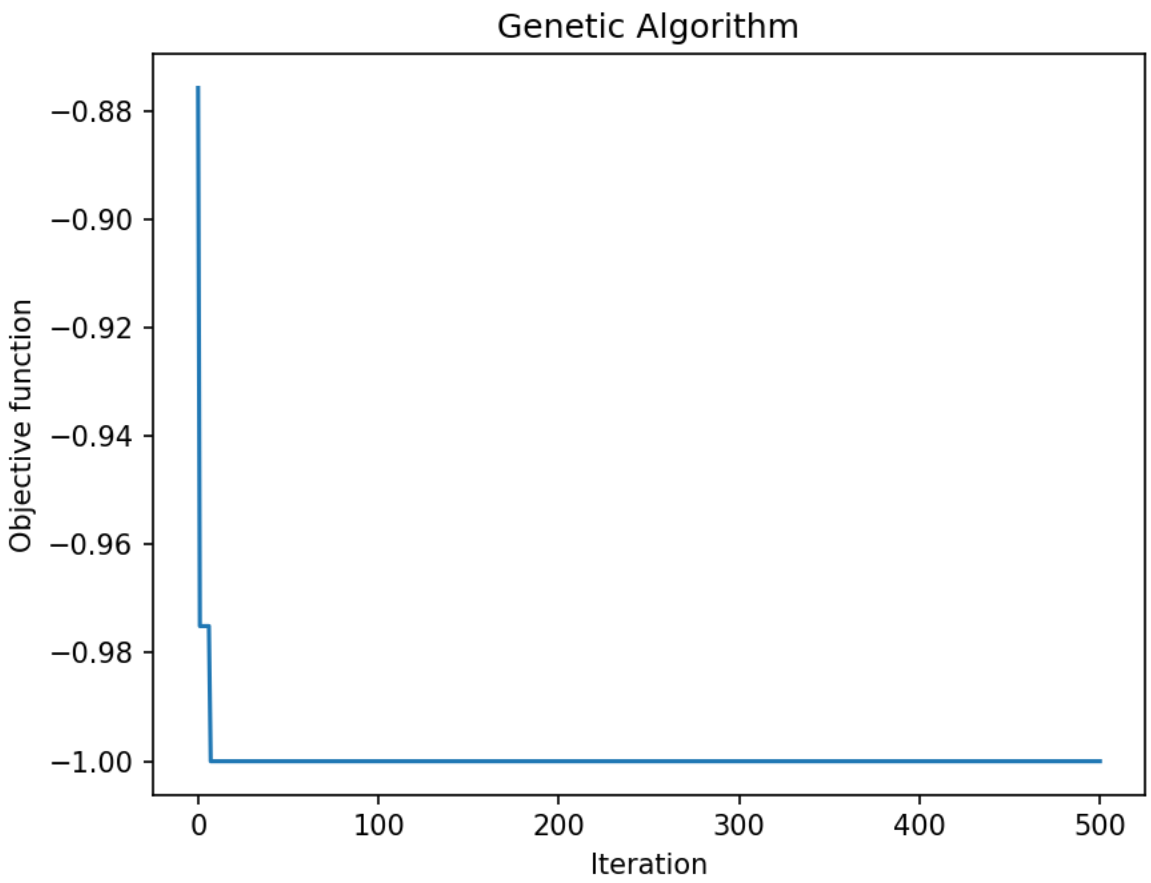

Figure 1. Frequency (Monobit) Graph for Convergence

In Figure 1, convergence can be seen be the line no longer decreasing after 0. This means that the test was could not find a better solution over the next several iterations with the fitness function. Since the suggested population size for the algorithm was 1/30th of the suggested number of iterations, the same logic was applied for the testing, therefore 1/30th of 500 is about 17. Each member of the population was a randomly generated string of either 1 or 0s. The length of each member of the population was determined by the key size. For example, if the key size was 256-bits, then each member of the population would be 256 1s and 0s.

For the population, the parents portion parameter was set to 3/17 (17.647%), the mutation probability parameter was set to 0.05 (5%), the elite ratio parameter was set to 1/17 (5.882%), and the crossover probability parameter was set to 0.25 (25%). The mutation probability and

crossover probability are half of the default values because a smaller population size was used. While the elite ratio was set to 1/17 to allow for one elite in each population. Furthermore, the parent's portion was set to 3/17 so that three members of the next generation are from the previous generation. Three parents were chosen so that the percentage of parents in the population was close to half of the default value. The type of crossover used for the tests was a two-point crossover. In a two-point crossover, two points of the parents, existing solutions are chosen, then the bits around these points would be exchanged to form the children, new solutions.

According to Solgi (2020), the parents portion parameter controls how much of the next generation is from the previous generation. For the parent's portion, the members that move to the next generation are randomly chosen. The mutation probability parameter controls the probability that any of the individuals in a population could have a random bit change within it. The elite ratio parameter controls the percent of elites in each population. The crossover probability parameter controls the probability that a parent, or existing solution, in the population would pass on its characteristics to a child, or new solution.

The genetic algorithm was run 10 times with these settings to produce ten different keys of five different sizes. Running the genetic algorithm 10 times for each key size ensured that more accurate results were achieved. This resulted in 50 total keys generated for each of the thirteen tests. The key sizes that were used for the tests are 256-bit, 512-bit, 1024-bit, 2048-bit, and 4096-bit. These key sizes were chosen for two main reasons: (a) so that smaller and larger keys were tested, and (b) these are common key sizes that are used by stream ciphers. The process that was used to generate the keys can be seen in Figure 2.

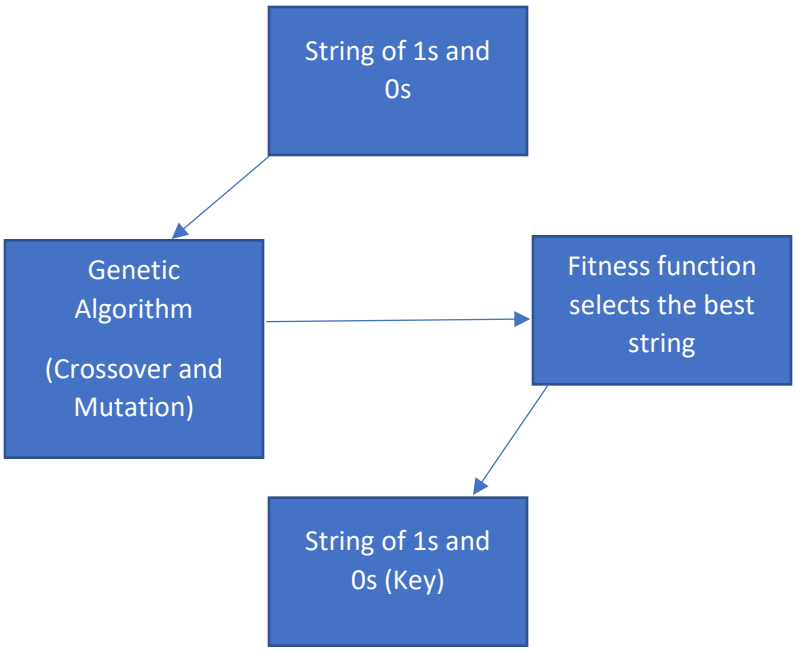

Figure 2. Key Generation Process

Figure 2 shows the key generation process. This process begins with the first generation of 17 random strings of 1s and 0s being entered into the genetic algorithm. These strings are made using Booleans so that each value can only be a one or a zero. The length of the strings depends on the key size entered into the genetic algorithm. Once the strings are entered into the algorithm each string is inputted into the fitness function to get its fitness value. Next, the string has a 25% chance to undergo crossover with another string and a 5% chance to be mutated to make new string for the next generation. Once, the crossovers and mutations complete, the next iteration will contain 13 new strings in addition to three strings from the previous generation and one elite string. This creates a total of 17 strings. This process continuous for 500 iterations. Once all of the iterations are finished running, the best string is chosen from the last iteration. The chosen string is the string with the highest fitness value.

After the keys were generated, the key size in bits, the average p-value, the average Hamming distance between each key in bits, and the time that it took to generate each key in

seconds, were inputted into an Excel file for each algorithm. As previously stated, the Hamming distance between each key is the number of differing bits between the keys. Therefore, if each key has a lot of differing bits from the next key, a high Hamming distance, then the keys would be more random. The average Hamming distance was found by calculating the Hamming distance between each of the keys, adding these values together, and dividing by one hundred. One hundred was used because that was the total number of Hamming distances found between each of the keys. Furthermore, the amount of time key generation takes dictates how quickly the genetic algorithm can encrypt or decrypt data. This value was found by adding the times for the generation of each of the ten keys and then dividing by ten. For each test, two separate graphs were created to compare the performance of the algorithms. The first graph plots the key size and the average Hamming distance between each key. The second graph plots the key size and the time that it took to generate the keys.

3.4. Sensitivity Analysis

After the individual tests were completed, two combinations of five tests were created to each serve as a single fitness function for the genetic algorithm. The reason that only five fitness functions were used is due to the amount of time that the sensitivity analysis would take. Table 1 shows the amount of time that it takes for the sensitivity analysis for all thirteen fitness functions with various amounts of weights. For example, if the thirteen functions are used with 4 weights, then it would take over one hundred years since it takes an average of ninety seconds to generate two keys. Whereas, as shown in Table 2, if five functions are used with four weights, it would only take around five hours to complete the sensitivity analysis since it only takes an average of eighteen seconds to generate two keys. The average of eighteen seconds was based upon a random group of five functions being used.

Table 1. Estimated Sensitivity Analysis Times for 13 NIST Functions and Various Weights

Amount of Weights	Number of NIST Functions	Average Time to Generate Two Keys (Seconds)	Total Seconds	Total Minutes	Total Hours	Total Days	Total Years
2	13	90	7.37E+05	1.23E+04	2.05E+02	8.53E+00	
3	13	90	1.43E+08	2.39E+06	3.99E+04	1.66E+03	4.55E+00
4	13	90	6.04E+09	1.01E+08	1.68E+06	6.99E+04	1.92E+02
5	13	90	1.10E+11	1.83E+09	3.05E+07	1.27E+06	3.48E+03
6	13	90	1.18E+12	1.96E+10	3.27E+08	1.36E+07	3.73E+04
7	13	90	8.72E+12	1.45E+11	2.42E+09	1.01E+08	2.77E+05
8	13	90	4.95E+13	8.25E+11	1.37E+10	5.73E+08	1.57E+06
9	13	90	2.29E+14	3.81E+12	6.35E+10	2.65E+09	7.25E+06
10	13	90	9.00E+14	1.50E+13	2.50E+11	1.04E+10	2.85E+07
11	13	90	3.11E+15	5.18E+13	8.63E+11	3.60E+10	9.85E+07

Table 2. Estimated Sensitivity Analysis Times for 5 NIST Functions and Various Weights

Amount of Weights	Number of NIST Functions	Average Time to Generate Two Keys (Seconds)	Total Seconds	Total Minutes	Total Hours	Total Days	Total Years
2	5	18	5.76E+02	9.60E+00	1.60E-01	6.67E-04	
3	5	18	4.37E+03	7.29E+01	1.22E+00	5.06E-02	1.39E-04
4	5	18	1.84E+04	3.07E+02	5.12E+00	2.13E-01	5.84E-04
5	5	18	5.63E+04	9.38E+02	1.56E+01	6.51E-01	1.78E-03
6	5	18	1.40E+05	2.33E+03	3.89E+01	1.62E+00	4.44E-03
7	5	18	3.03E+05	5.04E+03	8.40E+01	3.50E+00	9.59E-03
8	5	18	5.90E+05	9.83E+03	1.64E+02	6.83E+00	1.87E-02
9	5	18	1.06E+06	1.77E+04	2.95E+02	1.23E+01	3.37E-02
10	5	18	1.80E+06	3.00E+04	5.00E+02	2.08E+01	5.71E-02
11	5	18	2.90E+06	4.83E+04	8.05E+02	3.36E+01	9.19E-02

The combinations were created by using the five NIST functions that had the fastest average time and the five functions that had the largest average Hamming distance. The purpose of the combinations is to evaluate how the best performers from each evaluation function perform in a group as the fitness function. Sensitivity analysis was then performed on this

combination to determine how the functions impacted the average time and average Hamming distance of the generated key.

To conduct the sensitivity analysis, weights were applied to each fitness function. The weights that were applied started at 0.0 and were incremented by 1/3. This resulted in four total weights: 0.0, 1/3, 2/3, and 1.0. The weight of 0.0 was applied to test whether the combination would perform better if one or more of the functions did not have an impact on the generated key. The reason that four weights were used with the two combinations was due to time constraints. Table 2 shows how different weight amounts affect the time that it takes to complete the sensitivity analysis with various amounts of weights. For example, the difference in time that the sensitivity analysis takes for four and five weights is over ten hours.

Figure 3 shows a flow chart that described the sensitivity analysis:

1. The first combination was made of the top five NIST functions that had the highest Hamming distances for 256-bit keys. While the second combination was made of the top five fastest NIST functions for 256-bit keys.
2. For each combination the four weights, 0.0, 1/3, 2/3, and 1.0, were applied to each function using five for loops. Each iteration of the for loops would give a set of five weights.
3. The weights and NIST functions were inputted into the fitness function for the genetic algorithm as weights 1 through 5 and functions 1 through 5. The weights and functions were used to create an equation were weight 1 was multiplied to function 1, weight 2 was multiplied to function 2, weight 3 was multiplied to function 3, weight 4 was multiplied to function 4, and weight 5 was multiplied to function 5. The products were then added

together to produce a single value. The key that made this value the highest was then chosen as the best key.

4. The genetic algorithm generated two 256-bit keys. This key size was chosen because it required the shortest amount of time to create the key. These two keys were the keys that resulted in the highest fitness function value.
5. The Hamming distance between the two keys and the average time that it took to generate the two keys were recorded.
6. Once the keys were generated, the evaluation metric, or formula used to determine which chosen key had the best combination of Hamming distance and time, was calculated using the formula:

$$Evaluation\ metric = Hamming\ Distance + \frac{1}{Time} \quad (1)$$

The reason that $\frac{1}{Time}$ is used is to allow a smaller time value to result in a larger value. For example, 1/2 , or 0.5, is larger than 1/3, or 0.33.

7. The resulting evaluation metric was compared to the previous best combination of average Hamming distance and time.
    a. If the new evaluation metric was higher than the previous, then the weights, average Hamming distance, average time, and average p-value were stored.
    b. If the new evaluation metric was not higher than the previous, then the weights, average Hamming distance, average time, and average p-value were not stored.

Steps 2 through 7 were repeated until all NIST functions in the combination were used with each weight. At the end, the weights, Hamming distance, time, and p-value that resulted in the highest evaluation metric were inputted into the Excel file that was used for the individual tests.

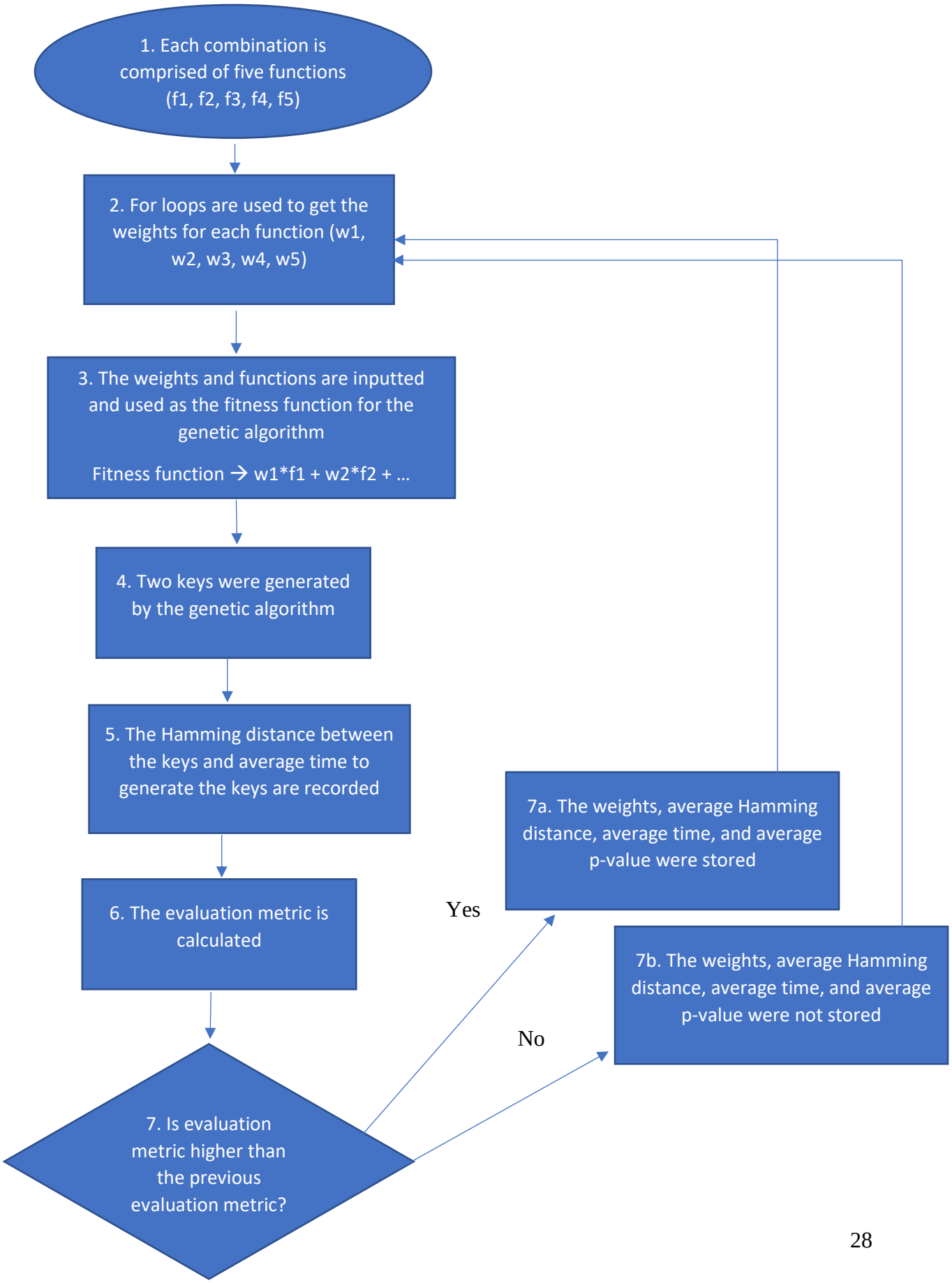
1. Each combination is comprised of five functions (f1, f2, f3, f4, f5)
2. For loops are used to get the weights for each function (w1, w2, w3, w4, w5)
3. The weights and functions are inputted and used as the fitness function for the genetic algorithm
Fitness function → w1*f1 + w2*f2 + …
4. Two keys were generated by the genetic algorithm
5. The Hamming distance between the keys and average time to generate the keys are recorded
6. The evaluation metric is calculated
7. Is evaluation metric higher than the previous evaluation metric?
Yes
No
7a. The weights, average Hamming distance, average time, and average p-value were stored
7b. The weights, average Hamming distance, average time, and average p-value were not stored

Figure 3. Sensitivity Analysis Flow Chart

3.5 Prediction

After the initial sensitivity analysis was run for the top five functions, the sensitivity analysis was run for the top two, three, and four combinations for Hamming distance and time. The results from these sensitivity analyses were documented in the Excel file and used with the top 5 to generate four prediction equations. Three of the equations were generated through linear regression and one was generated through polynomial regression. Both the linear regressions and polynomial regression were conducted by using data analysis in Excel.

Next, sensitivity analysis was conducted on the combinations of the top two NIST functions for Hamming distance and time for 256-bit keys. This time the variable that was changed was the key size. The key sizes that were used were 512-bit, 1024-bit, and 2048-bit. The results from these sensitivity analyses were recorded in the Excel file and used to generate four prediction equations through linear regression. As with the regressions from the other prediction, these linear regressions were conducted by using data analysis in Excel.

Chapter 4. Results and Discussion

4.1. Individual Fitness Functions

To evaluate how the thirteen NIST functions each performed individually for each of the five key sizes the average hamming distance, time, and p-value for the keys generated were recorded. Table 3 displayed these statistics for each fitness function.

Table 3. Statistics for Fitness Functions

NIST Functions	Key Size (bits)	Average Hamming Distance (bits)	Average Time (seconds)	Average P-value
Frequency (Monobit)	256	114.880	2.286	1.000
	512	233.800	3.601	1.000
	1024	457.560	6.169	1.000
	2048	922.800	11.261	1.000
	4096	1838.760	21.612	1.000
Frequency Test Within a Block	256	113.740	2.048	1.000
	512	233.000	3.194	1.000
	1024	457.820	5.552	0.999
	2048	923.180	10.253	0.999
	4096	1840.700	21.924	1.000
Runs Test	256	116.840	2.319	1.000
	512	233.840	3.624	1.000
	1024	456.080	6.241	1.000
	2048	921.960	11.660	1.000
	4096	1840.400	21.623	0.999
Test for the Longest Run of Ones in a Block	256	114.260	2.450	0.999
	512	228.520	3.916	0.999
	1024	460.020	6.761	0.999
	2048	912.420	12.552	1.000
	4096	1844.020	23.611	1.000
Binary Matrix Rank Test	256	114.960	6.948	1.000
	512	229.520	10.777	0.970
	1024	464.000	20.018	0.999
	2048	923.020	41.244	0.984
	4096	1844.540	94.655	0.981
Discrete Fourier Transform (Spectral) Test	256	114.640	3.207	0.819
	512	229.380	5.317	0.935
	1024	461.540	9.641	0.909
	2048	922.280	17.769	0.968
	4096	1841.340	34.885	0.954

Non-overlapping Template Matching Test	256	114.200	7.993	0.999
	512	231.680	17.067	0.999
	1024	461.180	35.202	0.986
	2048	922.640	72.609	0.868
	4096	1842.340	144.852	0.999
Maurer's "Universal Statistical" Test	256	115.920	2.839	0.999
	512	229.440	4.867	0.999
	1024	459.740	8.794	0.999
	2048	920.500	16.527	0.999
	4096	1840.540	32.110	0.999
Serial Test	256	115.240	4.075	2.000
	512	232.480	7.340	2.000
	1024	458.720	13.507	2.000
	2048	919.480	26.534	2.000
	4096	1840.600	53.912	2.000
Approximate Entropy Test	256	116.240	4.452	1.000
	512	227.740	8.008	0.999
	1024	459.100	14.907	0.999
	2048	920.440	29.131	0.999
	4096	1837.040	59.146	0.999
Cumulative Sums (Cusum) Test	256	116.640	2.836	1.999
	512	228.720	4.614	1.999
	1024	463.660	7.999	1.999
	2048	925.420	14.400	1.999
	4096	1842.800	27.124	1.999
Random Excursions Test	256	115.180	3.978	7.528
	512	230.540	6.022	7.473
	1024	460.420	9.338	7.500
	2048	921.780	16.522	7.550
	4096	1844.300	31.544	7.495
Random Excursions Variant Test	256	115.180	3.371	16.871
	512	230.600	6.143	16.867
	1024	461.260	11.244	16.568
	2048	919.000	21.836	16.249
	4096	1847.400	43.070	16.314

Based on the results shown on Table 3, the average time between the tests varied greater than the average Hamming distance. For example, the average Hamming distance for 256-bit keys varied by roughly two bits, while the time varied by roughly five seconds. Additionally, for the majority of the NIST functions, the average p-value cannot be above 1. However, for the

serial, cumulative sums (cusum), random excursions, and random excursions variant tests the average p-values are above one. This is due to these tests using multiple p-values to determine if a string passes the test. This can be seen in the functions code in Appendix A.

When the thirteen NIST functions were used as individual fitness functions, patterns started to emerge in the keys. One pattern is that the average Hamming distance for each key size was almost half of the key size. For example, for 256-bit keys, the average Hamming distance was between 113.740 and 116.840. The reason that this occurs is most likely due to the randomness of the keys. An average Hamming distance of half the key size meant that, on average, each key was only about fifty percent similar to the next. Which, in turn, means that, on average, fifty percent of each key had no relationship with the next key.

Furthermore, the average Hamming distance for a higher key size was closer to double the average Hamming distance for the previous key size. For example, the average Hamming distance for the runs test for 256-bit keys was 116.840. For 512-bit keys, the average Hamming distance for the runs test was 233.840, which is 2.001 times larger than 116.840. This supports the idea that the key size and Hamming distance variables were in a linear relationship.

Additionally, a similar pattern appeared in the average time that it took to generate a key where the times were almost doubled when the key size increased. This pattern was not as apparent between 256-bit and 512-bit keys, however, as the key size got above 1024-bits, this became more evident. For example, for the frequency test within a block, the time needed for 512-bit key generation is 1.560 times than the time needed for 256-bit key generation. However, the time needed for 2048-bit key generation is 1.848 times larger than the time needed for 1024-bit key generation. Similar to Hamming distance, this doubling pattern was most likely due to the

linear relationship between key size and time. For example, 512 is double the value of 256, 1024 is double the value of 512, and so on.

Furthermore, to better understand how each of the thirteen individual NIST functions compared to each other two figures were created. Figure 4 is a graph of average hamming distance for each key size for the thirteen NIST functions. Figure 5 is a graph of the average time that it took to generate each key for each key size for the thirteen NIST functions.

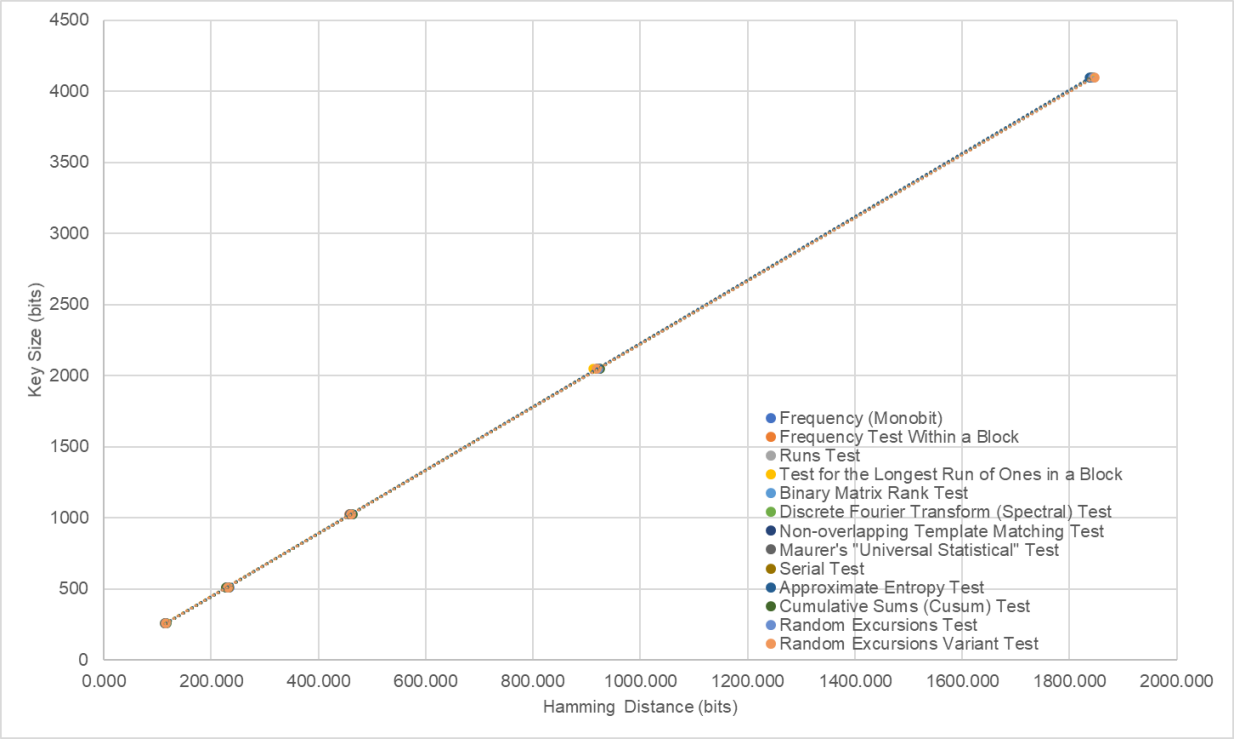

Figure 4. Key Size vs. Hamming Distance

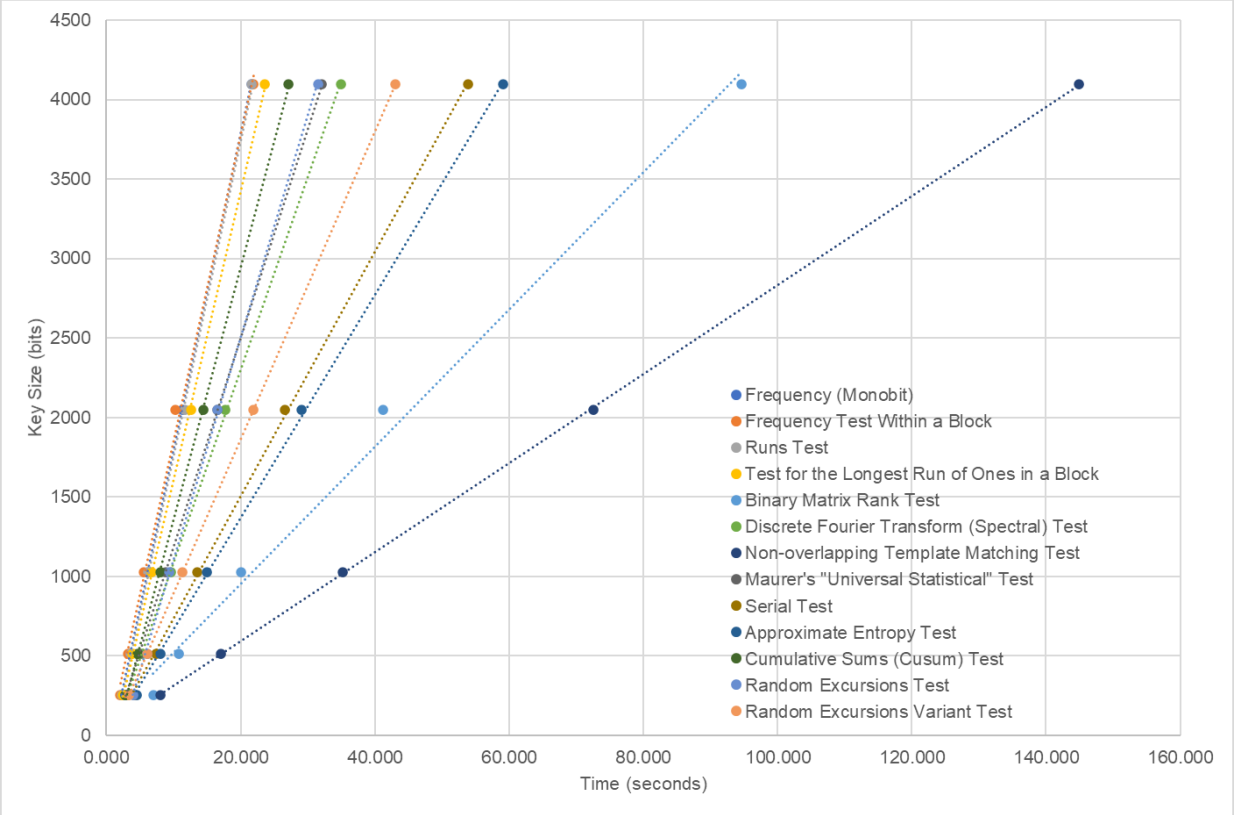

Figure 5. Key Size vs. Time

The results in Figure 4 show that each of the NIST functions had little variance in terms of average hamming distance. This is conveyed by the graph appearing to only have one or two lines on it. This is different from Figure 5 where all of the lines are viewed easily. The reason for that the lines can be viewed in Figure 5 is because there was a higher variance in the average time values. Therefore, speed would be considered as more of a deciding factor when it comes to choosing which NIST function to use individually as a fitness function.

4.2. Best Performing Individual Fitness Functions

To conduct the sensitivity analysis, the average Hamming distances and time of the thirteen NIST functions were ranked in terms of the key sizes. Table 4 showed the comparison in terms of average hamming distance between keys. The best average hamming distance represented the NIST function that had the highest average hamming distance. Table 5 showed

the rankings in terms of average time to generate a key for each key size. The best average time was the NIST function that took the least amount of time to generate a key.

Table 4. Average Hamming Distance Rankings

NIST Function	256 bits	Hamming Distance (bits)	512 bits	Hamming Distance (bits)	1024 bits	Hamming Distance (bits)	2048 bits	Hamming Distance (bits)	4096 bits	Hamming Distance (bits)
Frequency (Monobit)	9	114.880	2	233.800	12	457.560	4	922.800	12	1838.760
Frequency Test within a Block	13	113.740	3	233.000	11	457.820	2	923.180	8	1840.700
Runs Test	**1**	**116.840**	**1**	**233.840**	13	456.080	7	921.960	11	1840.400
Test for the Longest Run of Ones in a Block	11	114.260	12	228.520	7	460.020	13	912.420	4	1844.020
Binary Matrix Rank Test	8	114.960	8	229.520	**1**	**464.000**	3	923.020	2	1844.540
Discrete Fourier Transform (Spectral) Test	10	114.640	10	229.380	3	461.540	6	922.280	7	1841.340
Non-overlapping Template Matching Test	12	114.200	5	231.680	5	461.180	5	922.640	6	1842.340
Maurer's "Universal Statistical" Test	4	115.920	9	229.440	8	459.740	9	920.500	10	1840.540
Serial Test	5	115.240	4	232.480	10	458.720	11	919.480	9	1840.600
Approximate Entropy Test	3	116.240	13	227.740	9	459.100	10	920.440	13	1837.040
Cumulative Sums (Cusum) Test	2	116.640	11	228.720	2	463.660	**1**	**925.420**	5	1842.800
Random Excursions Test	6	115.180	7	230.540	6	460.420	8	921.780	3	1844.300
Random Excursions Variant Test	7	115.180	6	230.600	4	461.260	12	919.000	**1**	**1847.400**

From the results on Table 4, the best, number one, performing test for average Hamming distance is not the same for each key size. The best performing function for average Hamming

distance for 256-bit and 512-bit keys was the Runs Test. Whereas, the best performing tests for 1024-bit, 2048-bit, and 4096-bit keys were the binary matrix rank test, cumulative sums (cusum) test, and random excursions variant test. Furthermore, this lack of consistency could be seen throughout the table by many functions drastically changing ranking.

Table 5. Average Time Rankings

NIST Function	256 bits	Time (seconds)	512 bits	Time (seconds)	1024 bits	Time (seconds)	2048 bits	Time (seconds)	4096 bits	Time (seconds)
Frequency (Monobit)	**1**	**2.286**	**1**	**3.601**	**1**	**6.169**	**1**	**11.261**	**1**	**21.612**
Frequency Test within a Block	2	2.048	2	3.194	2	5.552	2	10.253	3	21.924
Runs Test	3	2.319	3	3.624	3	6.241	3	11.660	2	21.623
Test for the Longest Run of Ones in a Block	4	2.450	4	3.916	4	6.761	4	12.552	4	23.611
Binary Matrix Rank Test	12	6.948	12	10.777	12	20.018	12	41.244	12	94.655
Discrete Fourier Transform (Spectral) Test	7	3.207	7	5.317	8	9.641	8	17.769	8	34.885
Non-overlapping Template Matching Test	13	7.993	13	17.067	13	35.202	13	72.609	13	144.852
Maurer's "Universal Statistical" Test	6	2.839	6	4.867	6	8.794	7	16.527	7	32.110
Serial Test	10	4.075	10	7.340	10	13.507	10	26.534	10	53.912
Approximate Entropy Test	11	4.452	11	8.008	11	14.907	11	29.131	11	59.146
Cumulative Sums (Cusum) Test	5	2.836	5	4.614	5	7.999	5	14.400	5	27.124
Random Excursions Test	9	3.978	8	6.022	7	9.338	6	16.522	6	31.544
Random Excursions Variant Test	8	3.371	9	6.143	9	11.244	9	21.836	9	43.070

Based on the results on Table 5, the performance of the NIST functions for average time was much more consistent. In fact, there was only one change in the top five performing functions for each key size. This change occurred when the runs test outperformed the frequency test within a block for 4096-bit keys. Additionally, outside of the top five, there was few NIST functions that changed position as the key size changed.

Moving forward, Tables 4 and 5 further exhibited that there was more variance between time then Hamming distance. For example, for 256-bit keys, the average Hamming distance ranged from 113.740 to 116.840, while the average times ranged from 2.048 to 7.993. Additionally, these tables show that the best performer for either evaluation function for each key size was not always the same. For example, the runs test had the highest Hamming distance for 256-bit and 512-bit keys, however the binary matrix rank test, cumulative sums (cusum) test, and the random excursions variant test had the highest average Hamming distance for 1024-bit, 2048-bit, and 4096-bit keys respectively.

Furthermore, the complexity of the NIST functions had an impact on the time that the genetic algorithm stream cipher took to generate a key. For example, simpler NIST functions, like frequency monobit and frequency test within a block, had a shorter time to generate keys, than more complex tests, like the binary matrix rank test. Furthermore, the design of the NIST tests potentially had an impact on the average Hamming distance. For example, some of the NIST tests were designed for large bit sizes, like the random excursion test and random excursion variant tests. These two tests had two of the top three Hamming distances values for the largest key size.

Regardless, the average Hamming distance and time values were different for each NIST function. Furthermore, the order of the best performing NIST functions for average Hamming

distance for each key size was different than the order of the best performing NIST functions for average time. Thus, the following hypotheses are not rejected:

- Certain fitness functions will cause the genetic algorithm stream cipher to generate more random keys than other fitness functions.
- Certain fitness functions will cause the genetic algorithm stream cipher to generate keys faster than other fitness functions.

4.3. Combinations of Fitness Functions

To evaluate how a combination of NIST functions perform as a single fitness function, the top five performs in terms of average hamming distance and average time for 256-bit key generation were put into two combinations. This key size was chosen because it required the shortest amount of time to create the key. The first combination contained the top five average hamming distance performers, and the second combination contained the top five average time performers. Sensitivity analysis was then conducted on each combination and the results are recorded in Tables 6 and 7. These tables include the weights that were applied to each fitness function as well as the average hamming distance, time, and p-value.

Table 6. Top Five Hamming Distance Performers for 256-bit Key Generation

Fitness Function	Weight	Evaluation Function	Value
Runs Test	1.000	Hamming Distance (bits)	154.000
Cumulative Sums (Cusum) Test	0.000	Average Time (seconds)	8.520
Approximate Entropy Test	0.333	Average P-value	2.667
Maurer's "Universal Statistical" Test	0.000		
Serial Test	0.667		

Table 7. Top Five Time Performers for 256-bit Key Generation

Fitness Function	Weight	Evaluation Function	Value
Frequency Test within a Block	0.000	Hamming Distance (bits)	152.000
Frequency (Monobit)	0.333	Average Time (seconds)	4.134
Runs Test	1.000	Average P-value	2.332
Test for the Longest Run of Ones in a Block	0.333		

Cumulative Sums (Cusum) Test	0.333

Based on the results in Tables 6 and 7, the best performing combination canceled out at least one of the functions. In the first combination, the top five Hamming distance performers, the cumulative sums (cusum) and Maurer's "Universal Statistical" test were canceled out. While in the second combination, the top five time performers, the frequency test within a block was canceled out. As expected, the top five Hamming distance performers combination had a higher Hamming distance than the top five time performers combination. Whereas the top five time performers had a quicker time than the top five Hamming distance performers. However, the Hamming distance difference was only two bits, while the difference in average time between the two combinations was about four seconds. Additionally, the average p-value for each of the combinations was between 2 and 3.

In regard to the individual tests, both combinations had a much higher Hamming distance for 256-bit keys. However, neither had a better time than the individual best performing NIST function for 256-bit keys. Although, if the same evaluation metric, Equation 1, that was used to select the best combination in the sensitivity analysis was applied to the individual NIST tests, then both the combinations would have a higher evaluation metric than any of the individual NIST tests. When looking at the average Hamming distance and time for the two combinations, the average time still has a bigger difference then the average Hamming distance. Since the evaluation metric is higher for the combinations, and both the combinations have different average Hamming distance and time values the following hypotheses are not rejected:

- The combination of multiple functions as a single fitness function will improve the keys that are generated by the genetic algorithm stream cipher.

- Certain combinations of functions as fitness functions will generate more random keys than other combinations of functions.
- Certain combinations of functions as fitness functions will generate keys faster than other combinations of functions.

4.4 Prediction

4.4.1. Prediction based on number of functions.

The prediction equations to predict the Hamming distance and average time according to the amount of NIST functions were created using the results from the top two through five combinations of NIST functions. The NIST functions in the combinations were chosen in respect to their individual average time performance for 256-bit keys in Table 5. Equation 2 was created through linear regression and Equation 3 was created through polynomial regression.

$$y = 0.5815x + 1.039 \quad (2)$$

$$y = 2.8333x^3 - 31x^2 + 110.17x + 22 \quad (3)$$

The reason that polynomial regression was used for Equation 5 instead of linear regression was because polynomial regression resulted in an R^2 of 1. This means that the equation perfectly fits the actual data. Furthermore, linear regression was used for Equation 2 because the R^2 value was above 0.9. These prediction equations were used to predict the Hamming distance and average time values for the top 2 functions to all 13 functions. The predicted Hamming distance and average time values for the top 2 to 5 combinations were then compared with the actual values in Table 8 and Figures 6 and 7.

Table 8. Prediction Using Top 5 Average Time Fitness Functions

Number of functions	Actual Hamming Distance (bits)	Predicted Hamming Distance (bits)	Actual Average Time (seconds)	Predicted Average Time (seconds)
2	141.000	141.006	2.353	2.202
3	150.000	150.009	2.669	2.784

4	148.000	148.011	3.141	3.365
5	152.000	152.013	4.134	3.947
6		179.013		4.528
7		246.012		5.110
8		370.010		5.691
9		568.006		6.273
10		857.000		6.854
11		1253.992		7.436
12		1775.982		8.017
13		2439.970		8.599

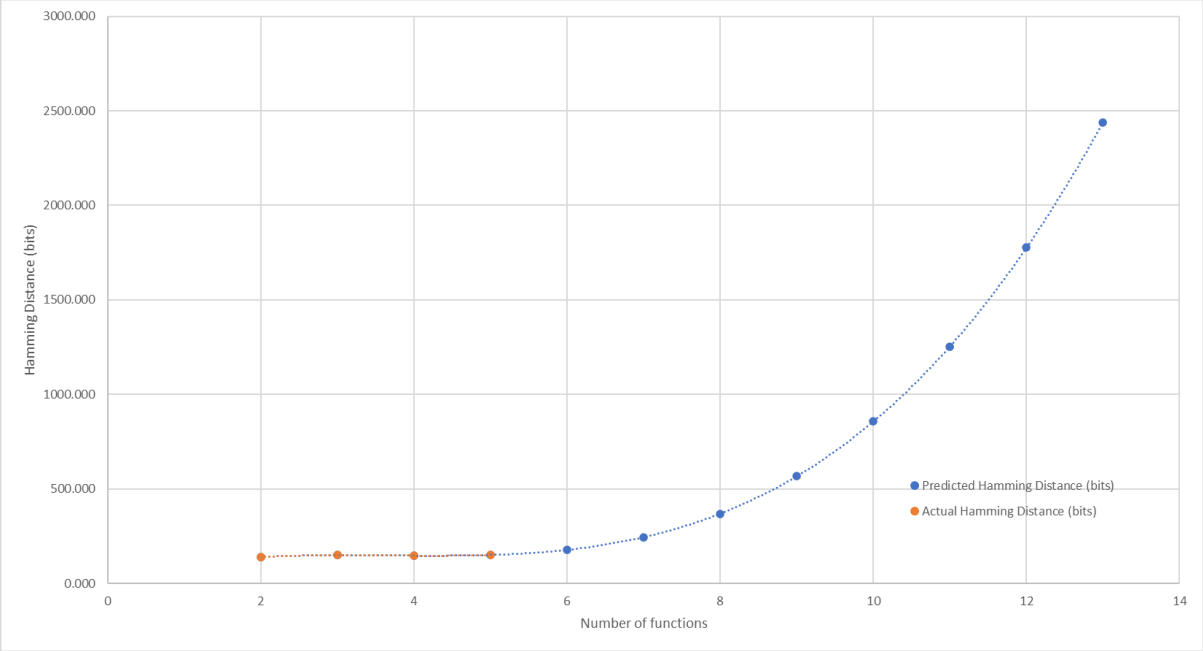

Figure 6. Predicted vs. Actual Hamming Distance for Top 5 Average Time Fitness Functions

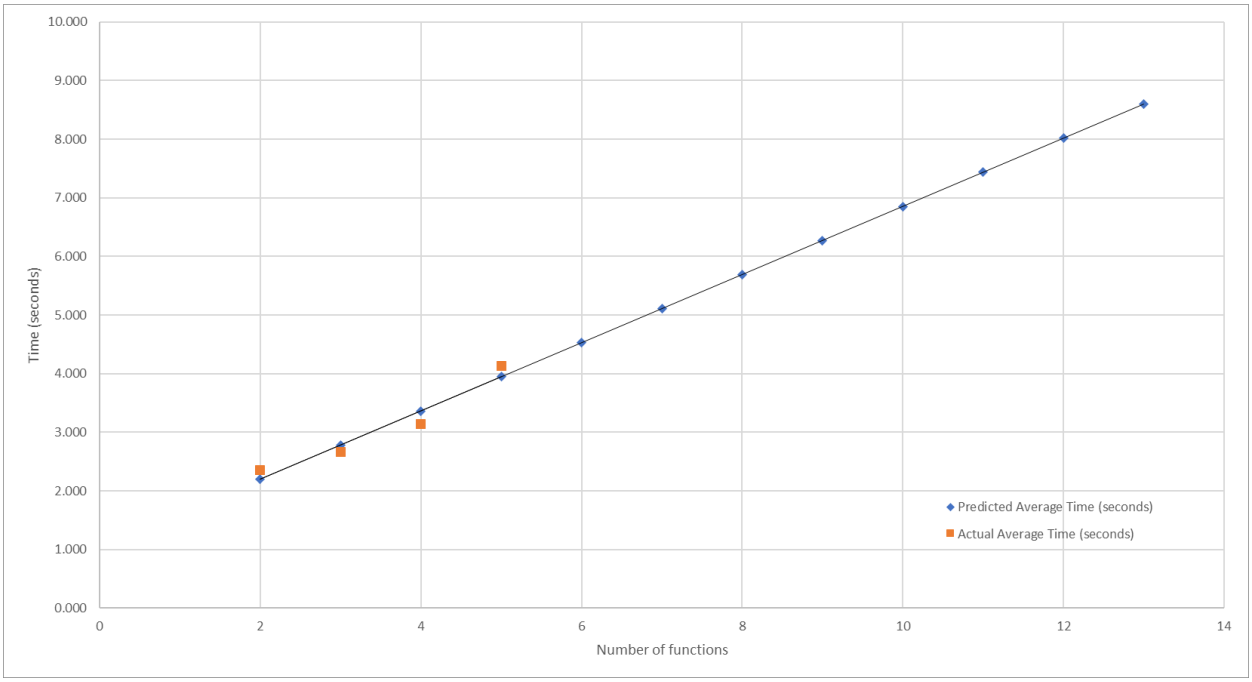

Figure 7. Predicted vs. Actual Average Time for Top 5 Average Time Fitness Functions

Based on the results, the predicted values for Hamming distance on Table 8 were very close to the actual values. This was most likely due to the R^2 value for the polynomial regression being 1, which means that the equation, Equation 3, perfectly fits the actual data. However, due to the way that Hamming distance is calculated, the Hamming distance cannot be above the key size. This means that at the tested key length 256-bits, the Hamming distance cannot exceed 256 bits. Therefore, it is believed that the predicted Hamming distance values that are above 256 bits mean that the keys would have a Hamming distance of 256 bit indicating the keys are completely dissimilar from all other keys produced. Further research can be done to support this theory. The line of best fit represented by Equation 3 further support the close fit of the equation to the data as shown in Figure 6. Furthermore, the predicted average time values were within two tenths of the actual values, which was within the standard error of 0.40. Again, this was most likely due to the R^2 of Equation 2 being above 0.93.

The prediction equations to predict the Hamming distance and average time according to the amount of NIST functions were created using the results from the top two through five combinations of NIST functions. The NIST functions in the combinations were chosen in respect to their individual Hamming distance performance for 256-bit keys in Table 4. The prediction equations, Equations 4 and 5, were created through linear regression.

$$y = 1.668x + 0.1595 \quad (4)$$

$$y = 5.8x + 126.2 \quad (5)$$

Linear regression was used because it resulted in R^2 values above 0.99. The prediction equations were used to predict the Hamming distance and average time values for the top 2 functions to all 13 functions. The predicted Hamming distance and average time values for the top 2 to 5 combinations were then compared with the actual values in Table 9 and Figures 8 and 9.

Table 9. Prediction Using Top 5 Hamming Distance Fitness Functions

Number of functions	Actual Hamming Distance (bits)	Predicted Hamming Distance (bits)	Actual Average Time (seconds)	Predicted Average Time (seconds)
2	136.000	137.800	3.231	3.496
3	146.000	143.600	5.713	5.164
4	150.000	149.400	6.526	6.832
5	154.000	155.200	8.520	8.500
6		161.000		10.168
7		166.800		11.836
8		172.600		13.504
9		178.400		15.172
10		184.200		16.840
11		190.000		18.508
12		195.800		20.176
13		201.600		21.844

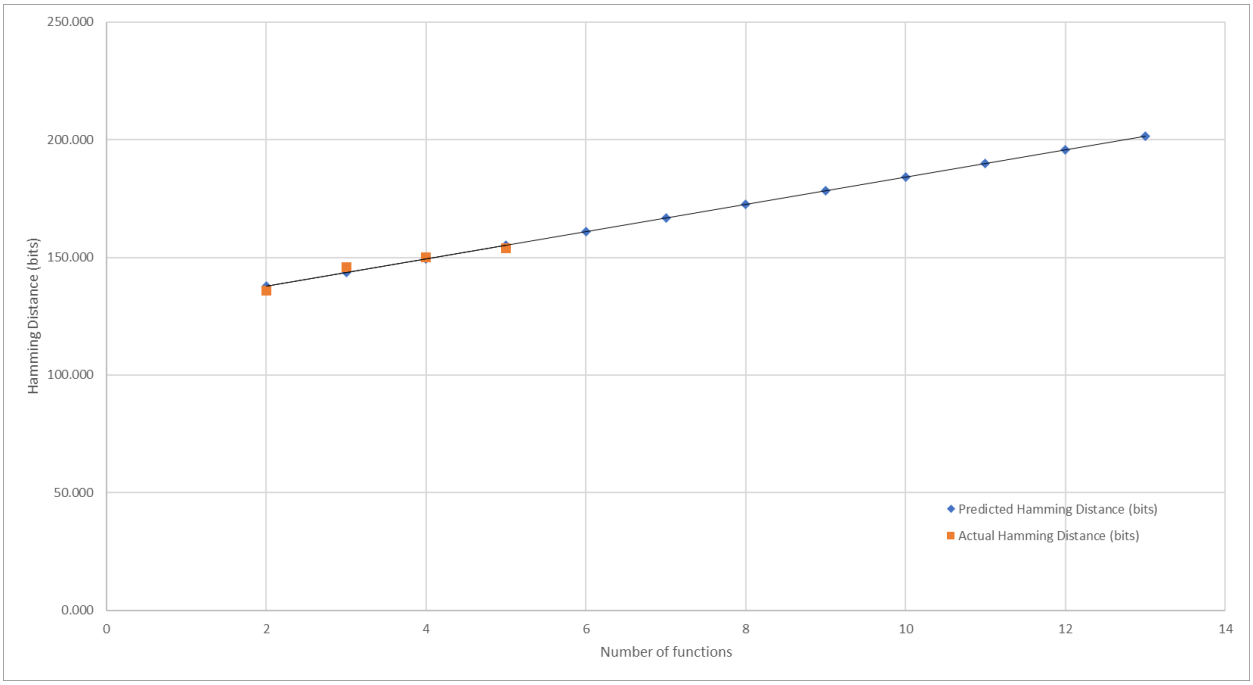

Figure 8. Predicted vs. Actual Hamming Distance for Top 5 Hamming Distance Fitness Functions

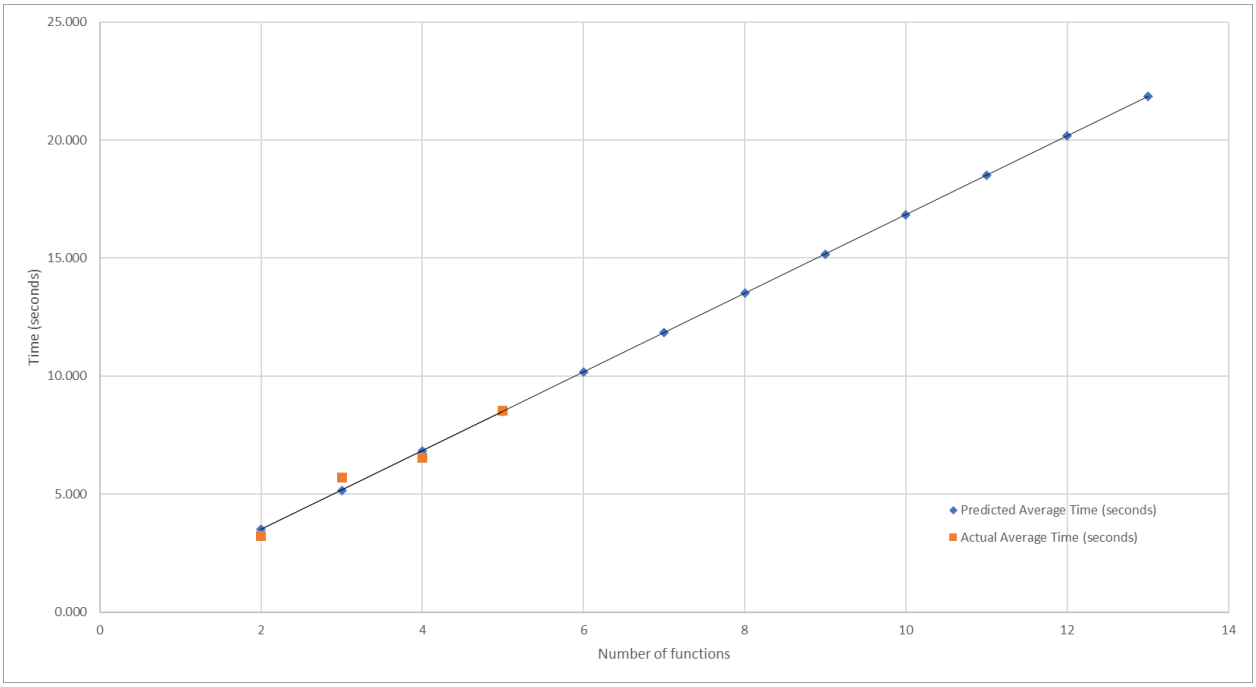

Figure 9. Predicted vs. Actual Average Time for Top 5 Hamming Distance Fitness Functions

Based on the results, the predictions made for Hamming distance in Table 9 were not as close to the actual values as the predicted Hamming distance values in Table 8. The higher inaccuracy of the predictions is due to the lower R^2 value for Equation 5. The R^2 for equation 5 was 0.94, whereas the R^2 for Equation 3 was 1. The average time predictions in Table 9 became closer to the actual values as the number of functions increased. Regardless, all of the predicted values were within the standard error of 0.79 when compared to the actual values.

4.4.2. Prediction based on key size.

The first two prediction equations to predict the Hamming distance and average time according to the size of the key were created using the results from sensitivity analyses conducted on the combination of the top two performing NIST functions in terms of average time for 256-bit keys. The equations were created through linear regression.

$$y = 0.0059x + 0.7593 \quad (6)$$

$$y = 0.5216x + 13.478 \quad (7)$$

Equations 6, for time, and 7, for Hamming distance, were used to predict the values for 256-bit, 512-bit, 1024-bit, 2048-bit, and 4096-bit keys. The predicted Hamming distance and time values for 256-bit, 512-bit, 1024-bit, and 2048-bit keys were compared with the actual values in Table 10 and Figures 10 and 11.

Table 10. Prediction Using Key Size for Top Two Average Time Fitness Functions

Key Size (bits)	Actual Hamming Distance (bits)	Predicted Hamming Distance (bits)	Actual Average Time (seconds)	Predicted Average Time (seconds)
256	141.000	147.008	2.353	2.270
512	282.000	280.537	3.776	3.780
1024	556.000	547.596	6.716	6.801
2048	1078.000	1081.715	12.942	12.843
4096		2149.952		24.926

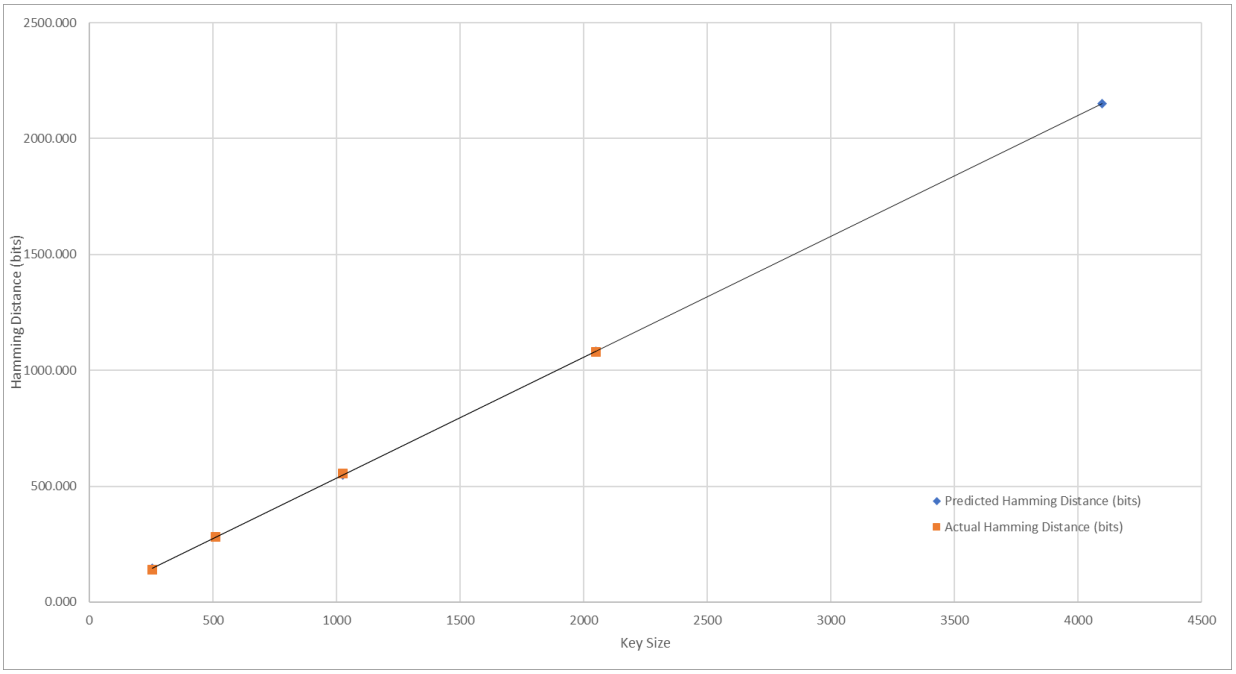

Figure 10. Predicted vs. Actual Hamming Distance for Key Size for Top 2 Average Time Fitness Functions

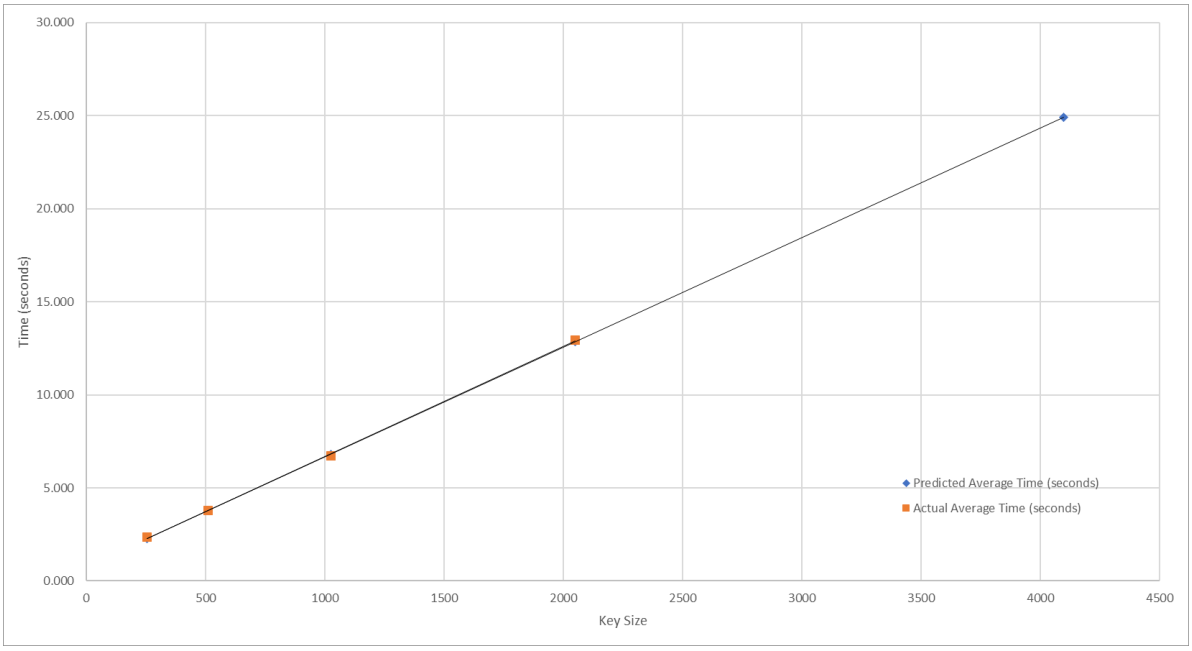

Figure 11. Predicted vs. Actual Average Time for Key Size for Top 2 Average Time Fitness Functions

From the results, the predicted values on Table 10 were relatively close to the actual values for Hamming distance and average time. All of the predicted values for Hamming distance fell within the standard error of 8.08 except for the predicted value for 1024-bit keys. For the average time predictions, all of the predicted values fell within the standard error of 0.17. Figures 10 and 11 exhibit this by how much overlap occurs between the predicted values line and actual values line.

The other two prediction equations to predict the Hamming distance and average time according to the size of the key were created using the results from sensitivity analyses conducted on the combination of the top two performing NIST functions in terms of Hamming distance for 256-bit keys. The equations were created through linear regression.

$$y = 0.0072x + 1.5585 \quad (8)$$

$$y = 0.531x - 1.5217 \quad (9)$$

Equations 8, for time, and 9, for Hamming distance, were used to predict the values for 256-bit, 512-bit, 1024-bit, 2048-bit, and 4096-bit keys. The predicted Hamming distance and time values for 256-bit, 512-bit, 1024-bit, and 2048-bit keys were compared with the actual values in Table 11 and Figures 12 and 13.

Table 11. Prediction Using Key Size for Top Two Hamming Distance Fitness Functions

Key Size (bits)	Actual Hamming Distance (bits)	Predicted Hamming Distance (bits)	Actual Average Time (seconds)	Predicted Average Time (seconds)
256	136.000	134.409	3.231	3.402
512	276.000	270.345	5.335	5.245
1024	531.000	542.217	9.111	8.931
2048	1090.000	1085.961	16.233	16.304
4096		2173.449		31.050

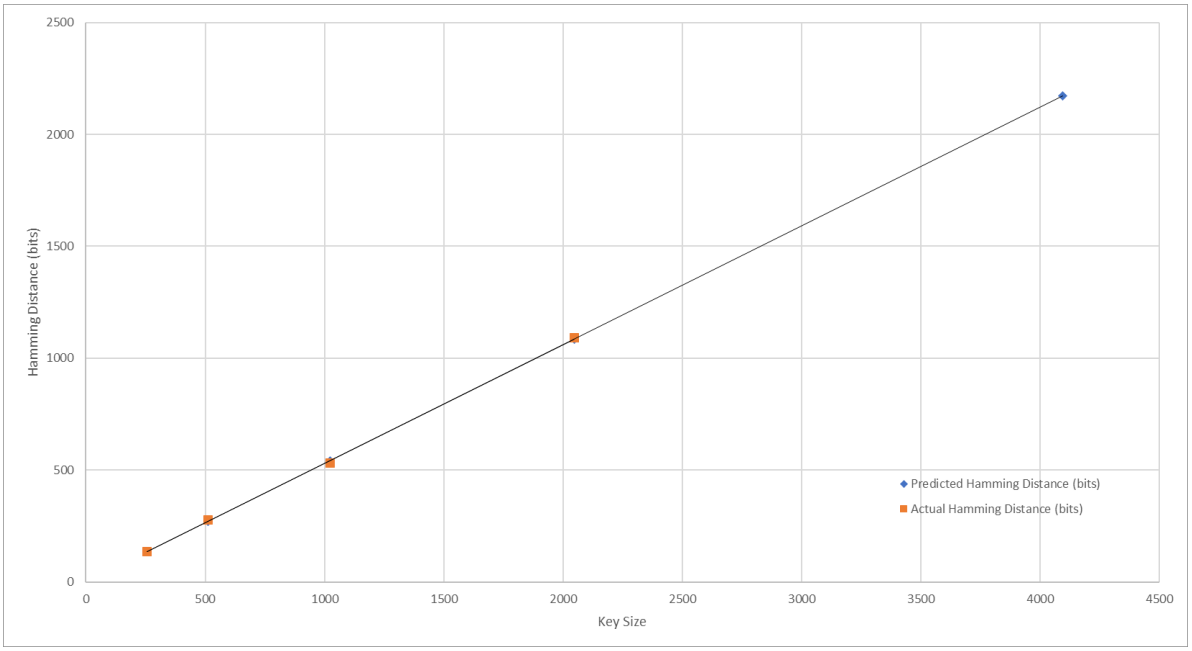

Figure 12. Predicted vs. Actual Hamming Distance for Key Size for Top 2 Hamming Distance Fitness Functions

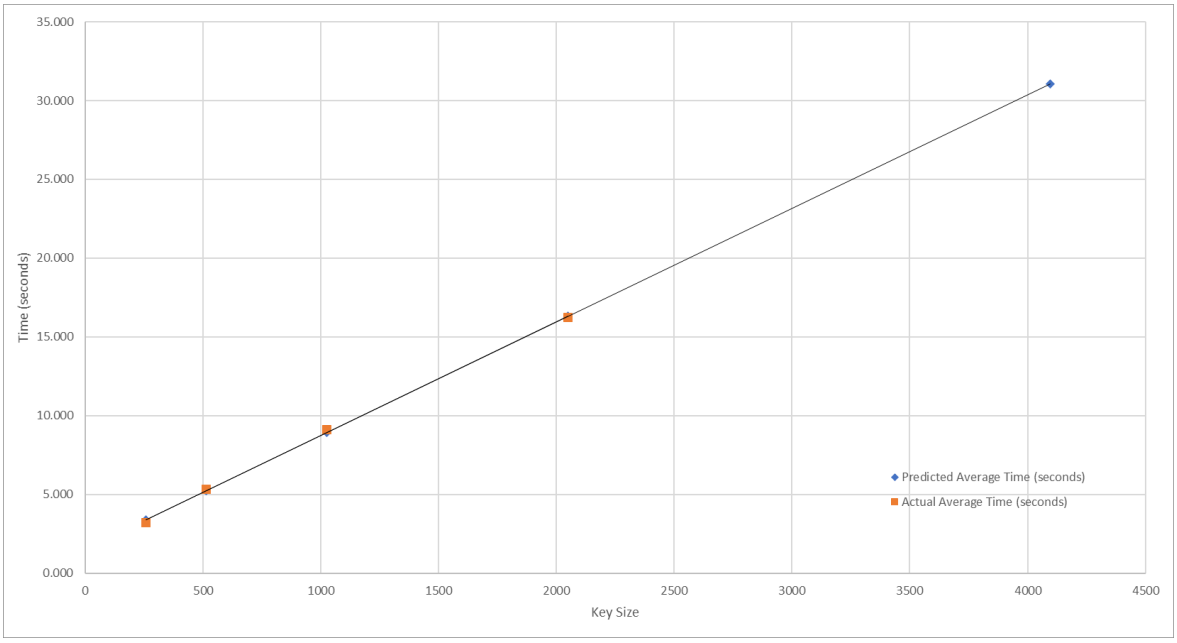

Figure 13. Predicted vs. Actual Average Time for Key Size for Top 2 Hamming Distance Fitness Functions

The predicted values in Table 11 were also relatively close to the actual values for Hamming distance and time. Similar to the predicted Hamming distance values in Table 10, the predicted value for 1024-bit keys fell outside of the standard error of 6.73 for Equation 9. In regard to the average time predictions in Table 11, all of the values fell outside the standard error of 0.08 for Equation 8. This means that these values fall more than one standard deviation from the mean. However, the values do fall within two standard deviations from the mean. Figures 12 and 13 further showed that the values were still close by the overlap between the two lines. Additionally, the R^2 values for Equations 6 through 9 were above 0.99, which explains the closeness of the predicted values.

Due to the closeness of the predicted results in Tables 8 to 11 and the R^2 values for the equations, the following hypotheses are not rejected:

- The Hamming distance and time results could be accurately predicted using regression.
- The Hamming distance and time results for 256-bit keys for the top two functions could be accurately predicted using regression.

Chapter 5. Conclusion

The goal of this research was to identify the effect that fitness functions, either used individually or in a combination, have on a genetic algorithm stream cipher and to predict the Hamming distance and average time through the results of the sensitivity analyses. To conduct this experiment thirteen individual NIST functions and two combinations of NIST functions were used as the fitness function for the genetic algorithm. The reason that only two combinations were used was due to time constraints. However, the two combinations that were tested produced differing results. Furthermore, each of the individual tests performed differently in terms of Hamming distance and time to generate a key. Additionally, prediction equations were found to closely predict the Hamming distance and average time values depending on the number of functions used or key size. Overall, this research showed that the randomness of the keys and the time that it takes to generate a key depend upon the individual fitness function or combination of fitness functions used.

Due to resource limitations, the evaluation of how combinations of fitness functions affect the genetic algorithm stream cipher and the identification of which stream cipher is the best for a given scenario were not completed. In future research, researchers should apply parallel computing to test all possible combinations of the thirteen NIST functions. Examples of parallel computing include CUDA and virtual machine clusters. Parallel computing will reduce the amount of time that it takes to generate the keys for each combination. This reduction of time will allow for each combination to be tested faster.

Additionally, this will allow for researchers to use each combination to generate more keys which will help account for randomness in the calculation of the Equation 1 to evaluate the combinations of functions. The analysis of all combinations will show how each combination

affects the genetic algorithm stream cipher. Furthermore, since all combinations will be tested the combinations can be compared with the individual tests to identify when stream cipher is the best for a given scenario.

Future researchers could perform additional sensitivity analysis with more data points and different key sizes to produce more accurate prediction formulas that address a variety of possible function and key size combinations. Furthermore, multivariable analysis would allow researches to investigate the relationship and potential influence of multiple factors on Hamming distance and/or average time. These factors can include key size, weights, number of functions, combination of functions, or factors not addressed in this research.

References

Ali, A. M. (2013). Randomly encryption using genetic algorithm. *Int. J. Appl. Innov. Eng. Manage.*, *2*(8), 242-246.

Alzubi, J., Nayyar, A., & Kumar, A. (2018, November). Machine learning from theory to algorithms: an overview. *Journal of physics: conference series* 1142(1). IOP Publishing. https://doi.org/10.1088/1742-6596/1142/1/012012

Bassham III, L. E., Rukhin, A. L., Soto, J., Nechvatal, J. R., Smid, M. E., Barker, E. B., ... & Vo, S. (2010). SP 800-22 rev. 1a. a statistical test suite for random and pseudorandom number generators for cryptographic applications. NIST. https://nvlpubs.nist.gov/nistpubs/legacy/sp/nistspecialpublication800-22r1a.pdf

Black, P. E. (2006, May 31). *Hamming distance*. Dictionary of Algorithms and Data Structures. https://www.nist.gov/dads/HTML/HammingDistance.html

Chandra, S., Bhattacharyya, S., Paira, S., & Alam, S. S. (2014a). A study and analysis on symmetric cryptography. Paper presented at the 2*014 International Conference on Science Engineering and Management Research (ICSEMR)* (pp. 1-8). https://doi.org/10.1109/ICSEMR.2014.7043664

Chandra, S., Paira, S., Alam, S. S., & Sanyal, G. (2014b). A comparative survey of symmetric and asymmetric key cryptography. Paper presented at the *2014 international conference on electronics, communication and computational engineering (ICECCE)* (pp. 83-93). https://doi.org/10.1109/ICECCE.2014.7086640

De Jong, K. (1988). Learning with genetic algorithms: An overview. *Machine learning*, *3*(2), 121-138. https://doi.org/10.1007/BF00113894

Deng, L. Y., & Bowman, D. (2017). Developments in pseudo-random number generators. *Wiley Interdisciplinary Reviews: Computational Statistics*, 9:e1404. https://doi.org/10.1002/wics.1404

Ding, Y., Tan, F., Qin, Z., Cao, M., Choo, K.-K. R., & Qin, Z. (2021). DeepKeyGen: A Deep Learning-Based Stream Cipher Generator for Medical Image Encryption and Decryption. *IEEE Transactions on Neural Networks and Learning Systems*, 1-15. doi:10.1109/TNNLS.2021.3062754

Fadil, T. A., Yaakob, S. N., & Ahmad, B. (2014). A hybrid chaos and neural network cipher encryption algorithm for compressed video signal transmission over wireless channel. *2014 2nd International Conference on Electronic Design (ICED).* Penang, Malaysia. doi:10.1109/ICED.2014.7015772

Fradkov, A. L. (2020). Early history of machine learning. *IFAC-PapersOnLine*, *53*(2), 1385-1390. https://doi.org/10.1016/j.ifacol.2020.12.1888

Gorbenko, I., Kuznetsov, A., Lutsenko, M., & Ivanenko, D. (2017). The research of modern stream ciphers. Paper presented at the *2017 4th International Scientific-Practical Conference Problems of Infocommunications. Science and Technology (PIC S&T)* (pp. 207-210). https://doi.org/10.1109/INFOCOMMST.2017.8246381

Goyat, S. (2012). Cryptography Using Genetic Algorithms (GAs). *OSR Journal of Computer Engineering (IOSRJCE)*, *1*(5), 06–08. www.iosrjournals.org

Guo, D., Cheng, L. M., & Cheng, L. L. (1999). A New Symmetric Probabilistic Encryption Scheme Based on Chaotic Attractors of Neural Networks. *Applied Intelligence, 10,* 71-84. doi:10.1023/A:1008337631906

Hjelle, G. A. (2022, March 21). Python Timer Functions: Three Ways to Monitor Your Code. Real Python. https://realpython.com/python-timer/

Jiao, L., Hao, Y., & Feng, D. (2020). Stream cipher designs: a review. *Science China Information Sciences*, *63*(3), 1-25. https://doi.org/10.1007/s11432-018-9929-x

Jordan, M. I., & Mitchell, T. M. (2015). Machine learning: Trends, perspectives, and prospects. *Science*, *349*(6245), 255-260. https://doi.org/0.1126/science.aac4520

Katoch, S., Chauhan, S. S., & Kumar, V. (2021). A review on genetic algorithm: past, present, and future. *Multimedia Tools and Applications*, *80*(5), 8091-8126. https://doi.org/10.1007/s11042-020-10139-6

Krishna, G. J., Ravi, V., & Bhattu, S. N. (2018). Key generation for plain text in stream cipher via bi-objective evolutionary computing. *Applied Soft Computing*, *70*, 301–317. https://doi.org/10.1016/j.asoc.2018.05.025

Kumar, A., & Chatterjee, K. (2016). An efficient stream cipher using genetic algorithm. In *2016 International Conference on Wireless Communications, Signal Processing and Networking (WiSPNET)* (pp. 2322-2326). IEEE. https://doi.org/ 10.1109/WiSPNET.2016.7566557

Lalar, D., & Nahta, R. (2016). Stream cipher. *International Journal of Advanced Research in Computer Science*, *7*(7). https://www.proquest.com/scholarly-journals/stream-cipher/docview/1871594708/se-2

Lian, S. (2009). A block cipher based on chaotic neural networks. *Neurocomputing*, *72*(4-6), 1296-1301. doi:10.1016/j.neucom.2008.11.005

Long, H. (2012). Stream Cipher Method Based on Neural Network. *National Conference on Information Technology and Computer Science (CITCS 2021)*, (pp. 414-417). Dongguan, China. Retrieved from https://www.atlantis-press.com/article/3111.pdf

Mahesh, B. (2020). Machine learning algorithms: A Review. *International Journal of Science and Research (IJSR). 9*, 381-386.

Manifavas, C., Hatzivasilis, G., Fysarakis, K., & Papaefstathiou, Y. (2016). A survey of lightweight stream ciphers for embedded systems. *Security and Communication Networks*, *9*(10), 1226-1246. https://doi.org/10.1002/sec.1399

Nazeer, M. I., Mallah, G. A., Shaikh, N. A., Bhatra, R., Memon, R. A., & Mangrio, M. I. (2018). Implication of genetic algorithm in cryptography to enhance security. *International Journal of Advanced Computer Science and Applications*, *9*(6), 375-379.

Nierhaus, G. (2009) Genetic Algorithms. *Algorithmic Composition*. Springer, Vienna. https://doi.org/10.1007/978-3-211-75540-2_7

Noura, H., Samhat, A. E., Harkouss, Y., & Yahiya, T. A. (2015). Design and realization of a new neural block cipher. *2015 International Conference on Applied Research in Computer Science and Engineering (ICAR)*, (pp. 1-6). Beiriut, Lebanon. doi:10.1109/ARCSE.2015.7338131

Pandey, D., Niwaria, K., & Chourasia, B. (2019). Machine Learning Algorithms: A Review. *International Research Journal of Engineering and Technology (IRJET)* , *6*(2), 916–922. www.irjet.net

Plasek, A. (2016). On the cruelty of really writing a history of machine learning. *IEEE Annals of the History of Computing*, *38*(4), 6-8. https://doi.org/10.1109/MAHC.2016.43

Pasqualini, L. (2021). *Nistrng* (Version 1.2.3). PyPI. https://pypi.org/project/nistrng/

S. Sakr, A., Y. Shams, M., Mahmoud, A., & Zidan, M. (2022). Amino acid encryption method using genetic algorithm for key generation. *Computers, Materials & Continua, 70*(1), 123-134. https://doi.org/10.32604/cmc.2022.019455

Sah, S. (2020). Machine learning: A review of learning types. *Preprints*. https://doi.org/10.20944/preprints202007.0230.v1

Sarker, I. H. (2021). Machine learning: Algorithms, real-world applications and research directions. *SN Computer Science, 2*(3), 1-21. https://doi.org/10.1007/s42979-021-00592-x

Sindhuja, K., & Devi, S. P. (2014). A symmetric key encryption technique using genetic algorithm. *International journal of computer science and information technologies, 5*(1), 414-416.

Solgi, M. (2020). *geneticalgorithm* (Version 1.0.2**).** PyPI. https://pypi.org/project/geneticalgorithm/

Som, S., Chatterjee, N. S., & Mandal, J. K. (2011). Key based bit level genetic cryptographic technique (KBGCT). Paper presented at the 240-245. https://doi.org/10.1109/ISIAS.2011.6122826

Stallings, W. (2017). *Cryptography and Network Security*. Pearson Education, Inc.

Sudeepa, K. B., Aithal, G., Rajinikanth, V., & Satapathy, S. C. (2020). Genetic algorithm based key sequence generation for cipher system. *Pattern Recognition Letters, 133,* 341–348. https://doi.org/10.1016/j.patrec.2020.03.015

Tsai, M., & Cho, H. (2021). A high security symmetric key generation by using genetic algorithm based on a novel similarity model. *Mobile Networks and Applications, 26*(3), 1386-1396. https://doi.org/10.1007/s11036-021-01753-1

Verma, V. K., & Kumar, B. (2014). Genetic algorithm: an overview and its application. *International Journal of Advanced Studies in Computers, Science and Engineering*, *3*(2), 21. www.ijascse.org

Zeng, K., Yang, C. H., Wei, D. Y., & Rao, T. R. N. (1991). Pseudorandom bit generators in stream-cipher cryptography. *Computer*, *24*(2), 8-17. https://doi.org/10.1109/2.67207

Appendix A – NIST SP 800-22r1a Tests

This appendix contains the python code for the thirteen tests. This code was structured to be as similar as possible to the original C code developed by NIST in NIST SP 800-22r1a.

A.1. Frequency (Monobit)

```
import math
def freq_monobit(ga_array):
    n = len(ga_array)
    sum = 0.0
    for i in range(n):
        sum += 2*int(ga_array[i])-1
    s_obs = abs(sum) / math.sqrt(n)
    f = s_obs / math.sqrt(2)
    p_value = math.erfc(f)
    return p_value
```

A.2. Frequency Test Within a Block

```
import scipy.special
def block_freq(ga_array):
    block_len = 64
    num_block = len(ga_array) // block_len
    sum = 0
    for i in range(num_block):
        block_sum = 0
        for j in range(num_block):
            block_sum += int(ga_array[j+i*num_block])
        pi = block_sum / num_block
        v = pi - 0.5
        sum += v * v
    chi_squared = 4.0 * block_len * sum
    p_value = scipy.special.gammaincc((num_block/2.0), (chi_squared/2.0))
    return p_value
```

A.3. Runs Test

```
import math
def runs(ga_array):
    n = len(ga_array)
    array = [int(x) for x in ga_array]
    S = array.count(1)
    pi = float(S) / n
    if abs(pi - 0.5) > (2.0 / math.sqrt(n)):
        p_value = 0.0
    else:
        V = 1
        for k in range(1, n):
            if array[k] != array[k-1]:
                V += 1
        erfc_arg = abs(V - 2.0 * n * pi * (1 - pi)) / (2.0 * pi * (1 - pi) *
      math.sqrt(2 * n))
        p_value = math.erfc(erfc_arg)
    return p_value
```

A.4. Test for the Longest Run of Ones in a Block

```
import scipy.special
```

```
def longestrunofones(ga_array):
    n = len(ga_array)
    assert n >= 128
    if n < 6272:
        K = 3
        M = 8
        V = [1, 2, 3, 4]
        pi = [0.21484375, 0.3671875, 0.23046875, 0.1875]
    elif 6272 < n < 750000:
        K = 5
        M = 128
        V = [4, 5, 6, 7, 8, 9]
        pi = [0.1174035788, 0.242955959, 0.249363483, 0.17517706,
      0.102701071, 0.112398847]
    else:
        K = 6
        M = 10000
        V = [10, 11, 12, 13, 14, 15, 16]
        pi = [0.0882, 0.2092, 0.2483, 0.1933, 0.1208, 0.0675, 0.0727]

    N = n // M
    nu = [0] * (K + 1)
    for i in range(N):
        v_n_obs = 0
        run = 0
        for j in range(M):
            if int(ga_array[i*M+j]) == 1:
                run += 1
                if run > v_n_obs:
                    v_n_obs = run
            else:
                run = 0
        if v_n_obs < V[0]:
            nu[0] += 1
        for j in range(K+1):
            if v_n_obs == V[j]:
                nu[j] += 1
        if v_n_obs > V[K]:
            nu[K] += 1

    chi2 = 0.0
    for i in range(K+1):
        chi2 += ((nu[i] - N * pi[i]) * (nu[i] - N * pi[i])) / (N * pi[i])

    p_value = scipy.special.gammaincc(float(K/2.0), (chi2 / 2.0))
    return p_value
```

A.5. Binary Matrix Rank Test

```
import math
import numpy
import nistmatrixdefinitions as nmatrix
def binary_matrix_rank(ga_array):
    n = len(ga_array)
    M = int(math.sqrt(n / 38))
    Q = int(math.sqrt(n / 38))
    N = n // (M * Q)

    if N == 0:
        p_value = 0.0
    else:
        r = M
        product = 1
        for i in range(r):
```

```
            product *= float(((1.0 - (2 ** (i - Q))) * (1.0 - (2 ** (i -
        M)))) / (1.0 - (2 ** (i - r))))
          p_32 = float(2 ** (r * (Q + M - r) - (M * Q))) * product


          r = M - 1
          product = 1
          for i in range(r):
              product *= float(((1.0 - (2 ** (i - Q))) * (1.0 - (2 ** (i -
        M)))) / (1.0 - (2 ** (i - r))))
          p_31 = float(2 ** (r * (Q + M - r) - (M * Q))) * product

          p_30 = 1 - (p_32 + p_31)

          F_32 = 0
          F_31 = 0
          for k in range(N):
              matrix = nmatrix.BinaryMatrix(ga_array, M, Q, k)
              R = matrix.compute_rank()
              if R == M:
                  F_32 += 1
              if R == M - 1:
                  F_31 += 1
          F_30 = (N - (F_32 + F_31))
          chi_squared_F_32 = ((F_32 - (N * p_32)) ** 2) / (N * p_32)
          chi_squared_F_31 = ((F_31 - (N * p_31)) ** 2) / (N * p_31)
          chi_squared_F_30 = ((F_30 - (N * p_30)) ** 2) / (N * p_30)
          chi_squared = chi_squared_F_32 + chi_squared_F_31 + chi_squared_F_30
          arg1 = -chi_squared/2.0
          p_value = math.e ** arg1
      return p_value
```

Matrix Creation (nistmatrixdefinitions) Code:

```
import numpy

class BinaryMatrix():
    def __init__(self, bitstring, M, Q, k):
        self.rows = M
        self.columns = Q
        self.matrix = self.def_matrix(bitstring, k)
        self.base_rank = min(M, Q)

    def perform_row_operations(self, i, forward_elimination):
        if forward_elimination == 1:
            j = i + 1
            while j < self.rows:
                if self.matrix[j, i] == 1:
                    for k in range(self.columns):
                        self.matrix[j, k] = (self.matrix[j, k] +
      self.matrix[i, k]) % 2
                j += 1

          else:
            j = i - 1
            while j >= 0:
                if self.matrix[j, i] == 1:
                    for k in range(self.columns):
                        self.matrix[j, k] = (self.matrix[j, k] +
      self.matrix[i, k]) % 2
                j -= 1

    def find_unit_element_and_swap(self, i, forward_elimination):
        row_op = 0
```

```
    if forward_elimination == 1:
        index = i + 1
        while index < self.rows and self.matrix[index, i] == 0:
            index += 1
        if index < self.rows:
            row_op = self.swap_rows(i, index)
    else:

        index = i - 1
        while index >= 0 and self.matrix[index, i] == 0:
            index -= 1
        if index >= 0:
            row_op = self.swap_rows(i, index)

    return row_op

def swap_rows(self, i, index):
    temp = copy.copy(self.matrix[i, :])
    self.matrix[i, :] = self.matrix[index, :]
    self.matrix[index, :] = temp
    return 1

def determine_rank(self):
    rank = self.base_rank
    for i in range(self.rows):
        allzeroes = 1
        for j in range(self.columns):
            if self.matrix[i, j] == 1:
                allzeroes = 0
                break

        if allzeroes == 1:
            rank -= 1
    return rank

def compute_rank(self):
    i = 0
    while i < (self.base_rank - 1):
        if self.matrix[i, i] == 1:
            self.perform_row_operations(i, 1)
        else:
            if self.find_unit_element_and_swap(i, 1) == 1:
                self.perform_row_operations(i, 1)
        i += 1

    i = self.base_rank - 1
    while i > 0:
        if self.matrix[i, i] == 1:
            self.perform_row_operations(i, 0)
        else:
            if self.find_unit_element_and_swap(i, 0) == 1:
                self.perform_row_operations(i, 0)
        i -= 1

    return self.determine_rank()

def def_matrix(self, bitstring, k):
    matrix = numpy.ndarray(shape=(self.rows, self.columns), dtype=int)
    for i in range(self.rows):
        for j in range(self.columns):
            matrix[i, j] = int(bitstring[k * (self.rows * self.columns) +
  j + i * self.rows])
    return matrix
```

A.6. Discrete Fourier Transform (Spectral) Test

```
import math
import scipy.fftpack
def discrete_fourier_transform(ga_array):
    n = len(ga_array)
    X = [0] * n
    m = [0] * (n // 2)
    for i in range(n):
        X[i] = 2 * int(ga_array[i]) - 1

    dft = scipy.fftpack.rfft(X, n=n)

    m[0] = math.sqrt(dft[0] * dft[0])

    i = 0
    while i < (n // 2) - 1:
        m[i + 1] = math.sqrt((dft[2*i+1] ** 2) + (dft[2*i+2] ** 2))
        i += 1

    count = 0
    upperbound = math.sqrt(math.log(1.0/0.05) * n)
    for i in range(n // 2):
        if m[i] < upperbound:
            count += 1

    percentile = count / (n / 2) * 100
    N_1 = count
    N_0 = 0.95 * n / 2.0
    d = (N_1 - N_0) / math.sqrt(n / 4.0 * 0.95 * 0.05)
    p_value = math.erfc(abs(d) / math.sqrt(2.0))
    return p_value
```

A.7. Non-overlapping Template Matching Test

```
import scipy.special
from random import choice
def nonOverlappingTemplateMatchings(ga_array):
    n = len(ga_array)
    maxnumberOfTemplates = 148
    K = 5
    m = 9

    templates_m9 = [[0, 0, 0, 0, 0, 0, 0, 0, 1], [0, 0, 0, 0, 0, 0, 0, 1, 1],
      [0, 0, 0, 0, 0, 0, 1, 0, 1],
                    [0, 0, 0, 0, 0, 0, 1, 1, 1], [0, 0, 0, 0, 0, 1, 0, 0, 1],
      [0, 0, 0, 0, 0, 1, 0, 1, 1],
                    [0, 0, 0, 0, 0, 1, 1, 0, 1], [0, 0, 0, 0, 0, 1, 1, 1, 1],
      [0, 0, 0, 0, 1, 0, 0, 0, 1],
                    [0, 0, 0, 0, 1, 0, 0, 1, 1], [0, 0, 0, 0, 1, 0, 1, 0, 1],
      [0, 0, 0, 0, 1, 0, 1, 1, 1],
                    [0, 0, 0, 0, 1, 1, 0, 0, 1], [0, 0, 0, 0, 1, 1, 0, 1, 1],
      [0, 0, 0, 0, 1, 1, 1, 0, 1],
                    [0, 0, 0, 0, 1, 1, 1, 1, 1], [0, 0, 0, 1, 0, 0, 0, 1, 1],
      [0, 0, 0, 1, 0, 0, 1, 0, 1],
                    [0, 0, 0, 1, 0, 0, 1, 1, 1], [0, 0, 0, 1, 0, 1, 0, 0, 1],
      [0, 0, 0, 1, 0, 1, 0, 1, 1],
                    [0, 0, 0, 1, 0, 1, 1, 0, 1], [0, 0, 0, 1, 0, 1, 1, 1, 1],
      [0, 0, 0, 1, 1, 0, 0, 1, 1],
                    [0, 0, 0, 1, 1, 0, 1, 0, 1], [0, 0, 0, 1, 1, 0, 1, 1, 1],
      [0, 0, 0, 1, 1, 1, 0, 0, 1],
                    [0, 0, 0, 1, 1, 1, 0, 1, 1], [0, 0, 0, 1, 1, 1, 1, 0, 1],
      [0, 0, 0, 1, 1, 1, 1, 1, 1],
```

```
                [0, 0, 1, 0, 0, 0, 0, 1, 1], [0, 0, 1, 0, 0, 0, 1, 0, 1],
[0, 0, 1, 0, 0, 0, 1, 1, 1],
                [0, 0, 1, 0, 0, 1, 0, 1, 1], [0, 0, 1, 0, 0, 1, 1, 0, 1],
[0, 0, 1, 0, 0, 1, 1, 1, 1],
                [0, 0, 1, 0, 1, 0, 0, 1, 1], [0, 0, 1, 0, 1, 0, 1, 0, 1],
[0, 0, 1, 0, 1, 0, 1, 1, 1],
                [0, 0, 1, 0, 1, 1, 0, 1, 1], [0, 0, 1, 0, 1, 1, 1, 0, 1],
[0, 0, 1, 0, 1, 1, 1, 1, 1],
                [0, 0, 1, 1, 0, 0, 1, 0, 1], [0, 0, 1, 1, 0, 0, 1, 1, 1],
[0, 0, 1, 1, 0, 1, 0, 1, 1],
                [0, 0, 1, 1, 0, 1, 1, 0, 1], [0, 0, 1, 1, 0, 1, 1, 1, 1],
[0, 0, 1, 1, 1, 0, 1, 0, 1],
                [0, 0, 1, 1, 1, 0, 1, 1, 1], [0, 0, 1, 1, 1, 1, 0, 1, 1],
[0, 0, 1, 1, 1, 1, 1, 0, 1],
                [0, 0, 1, 1, 1, 1, 1, 1, 1], [0, 1, 0, 0, 0, 0, 0, 1, 1],
[0, 1, 0, 0, 0, 0, 1, 1, 1],
                [0, 1, 0, 0, 0, 1, 0, 1, 1], [0, 1, 0, 0, 0, 1, 1, 1, 1],
[0, 1, 0, 0, 1, 0, 0, 1, 1],
                [0, 1, 0, 0, 1, 0, 1, 1, 1], [0, 1, 0, 0, 1, 1, 0, 1, 1],
[0, 1, 0, 0, 1, 1, 1, 1, 1],
                [0, 1, 0, 1, 0, 0, 0, 1, 1], [0, 1, 0, 1, 0, 0, 1, 1, 1],
[0, 1, 0, 1, 0, 1, 0, 1, 1],
                [0, 1, 0, 1, 0, 1, 1, 1, 1], [0, 1, 0, 1, 1, 0, 0, 1, 1],
[0, 1, 0, 1, 1, 0, 1, 1, 1],
                [0, 1, 0, 1, 1, 1, 0, 1, 1], [0, 1, 0, 1, 1, 1, 1, 1, 1],
[0, 1, 1, 0, 0, 0, 1, 1, 1],
                [0, 1, 1, 0, 0, 1, 1, 1, 1], [0, 1, 1, 0, 1, 0, 1, 1, 1],
[0, 1, 1, 0, 1, 1, 1, 1, 1],
                [0, 1, 1, 1, 0, 1, 1, 1, 1], [0, 1, 1, 1, 1, 1, 1, 1, 1],
[1, 0, 0, 0, 0, 0, 0, 0, 0],
                [1, 0, 0, 0, 1, 0, 0, 0, 0], [1, 0, 0, 1, 0, 0, 0, 0, 0],
[1, 0, 0, 1, 0, 1, 0, 0, 0],
                [1, 0, 0, 1, 1, 0, 0, 0, 0], [1, 0, 0, 1, 1, 1, 0, 0, 0],
[1, 0, 1, 0, 0, 0, 0, 0, 0],
                [1, 0, 1, 0, 0, 0, 1, 0, 0], [1, 0, 1, 0, 0, 1, 0, 0, 0],
[1, 0, 1, 0, 0, 1, 1, 0, 0],
                [1, 0, 1, 0, 1, 0, 0, 0, 0], [1, 0, 1, 0, 1, 0, 1, 0, 0],
[1, 0, 1, 0, 1, 1, 0, 0, 0],
                [1, 0, 1, 0, 1, 1, 1, 0, 0], [1, 0, 1, 1, 0, 0, 0, 0, 0],
[1, 0, 1, 1, 0, 0, 1, 0, 0],
                [1, 0, 1, 1, 0, 1, 0, 0, 0], [1, 0, 1, 1, 0, 1, 1, 0, 0],
[1, 0, 1, 1, 1, 0, 0, 0, 0],
                [1, 0, 1, 1, 1, 0, 1, 0, 0], [1, 0, 1, 1, 1, 1, 0, 0, 0],
[1, 0, 1, 1, 1, 1, 1, 0, 0],
                [1, 1, 0, 0, 0, 0, 0, 0, 0], [1, 1, 0, 0, 0, 0, 0, 1, 0],
[1, 1, 0, 0, 0, 0, 1, 0, 0],
                [1, 1, 0, 0, 0, 1, 0, 0, 0], [1, 1, 0, 0, 0, 1, 0, 1, 0],
[1, 1, 0, 0, 1, 0, 0, 0, 0],
                [1, 1, 0, 0, 1, 0, 0, 1, 0], [1, 1, 0, 0, 1, 0, 1, 0, 0],
[1, 1, 0, 0, 1, 1, 0, 0, 0],
                [1, 1, 0, 0, 1, 1, 0, 1, 0], [1, 1, 0, 1, 0, 0, 0, 0, 0],
[1, 1, 0, 1, 0, 0, 0, 1, 0],
                [1, 1, 0, 1, 0, 0, 1, 0, 0], [1, 1, 0, 1, 0, 1, 0, 0, 0],
[1, 1, 0, 1, 0, 1, 0, 1, 0],
                [1, 1, 0, 1, 0, 1, 1, 0, 0], [1, 1, 0, 1, 1, 0, 0, 0, 0],
[1, 1, 0, 1, 1, 0, 0, 1, 0],
                [1, 1, 0, 1, 1, 0, 1, 0, 0], [1, 1, 0, 1, 1, 1, 0, 0, 0],
[1, 1, 0, 1, 1, 1, 0, 1, 0],
                [1, 1, 0, 1, 1, 1, 1, 0, 0], [1, 1, 1, 0, 0, 0, 0, 0, 0],
[1, 1, 1, 0, 0, 0, 0, 1, 0],
                [1, 1, 1, 0, 0, 0, 1, 0, 0], [1, 1, 1, 0, 0, 0, 1, 1, 0],
[1, 1, 1, 0, 0, 1, 0, 0, 0],
                [1, 1, 1, 0, 0, 1, 0, 1, 0], [1, 1, 1, 0, 0, 1, 1, 0, 0],
[1, 1, 1, 0, 1, 0, 0, 0, 0],
```

```
                    [1, 1, 1, 0, 1, 0, 0, 1, 0], [1, 1, 1, 0, 1, 0, 1, 0, 0],
      [1, 1, 1, 0, 1, 0, 1, 1, 0],
                    [1, 1, 1, 0, 1, 1, 0, 0, 0], [1, 1, 1, 0, 1, 1, 0, 1, 0],
      [1, 1, 1, 0, 1, 1, 1, 0, 0],
                    [1, 1, 1, 1, 0, 0, 0, 0, 0], [1, 1, 1, 1, 0, 0, 0, 1, 0],
      [1, 1, 1, 1, 0, 0, 1, 0, 0],
                    [1, 1, 1, 1, 0, 0, 1, 1, 0], [1, 1, 1, 1, 0, 1, 0, 0, 0],
      [1, 1, 1, 1, 0, 1, 0, 1, 0],
                    [1, 1, 1, 1, 0, 1, 1, 0, 0], [1, 1, 1, 1, 0, 1, 1, 1, 0],
      [1, 1, 1, 1, 1, 0, 0, 0, 0],
                    [1, 1, 1, 1, 1, 0, 0, 1, 0], [1, 1, 1, 1, 1, 0, 1, 0, 0],
      [1, 1, 1, 1, 1, 0, 1, 1, 0],
                    [1, 1, 1, 1, 1, 1, 0, 0, 0], [1, 1, 1, 1, 1, 1, 0, 1, 0],
      [1, 1, 1, 1, 1, 1, 1, 0, 0],
                    [1, 1, 1, 1, 1, 1, 1, 1, 0]]

    N = 8
    M = n // N

    mu = (M - m + 1) / (2 ** m)
    varWj = M * ((1.0 / (2 ** m)) - ((2.0 * m - 1.0) / (2.0 ** (2.0 * m))))

    sequence = choice(templates_m9)
    for jj in range(min(maxnumberOfTemplates, len(sequence))):
        j = 0
        Wj = [0] * N
        nu = [0] * (K + 1)
        for k in range(K + 1):
            nu[k] = 0
        for i in range(N):
            W_obs = 0
            for j in range(M - m + 1):
                match = 1
                for k in range(m):
                    if sequence[k] != int(ga_array[i * M + j + k]):
                        match = 0
                        break

                if match == 1:
                    W_obs += 1

            Wj[i] = W_obs

        j += m - 1

    chi2 = 0.0
    for i in range(N):
        chi2 += ((Wj[i] - mu) / (varWj ** 0.5)) ** 2
    p_value = scipy.special.gammaincc(N / 2.0, chi2 / 2.0)

    return p_value
```

A.8. Maurer's "Universal Statistical" Test

```
import math
def universal(ga_array):
    n = len(ga_array)
    L = 2
    # Q = 4
    Q = 10 * 2 ** L
    K = n // L - Q
```

```
    expected_value = [0, 0.73264948, 1.5374383, 2.40160681, 3.31122472,
      4.25342659, 5.2177052, 6.1962507, 7.1836656,
                8.1764248, 9.1723243, 10.170032, 11.168765, 2.168070,
      13.167693, 14.167488, 15.167379]
    variance = [0, 0.690, 1.338, 1.901, 2.358, 2.705, 2.954, 3.125, 3.238,
      3.311, 3.356, 3.384, 3.401,
                3.410, 3.416, 3.419, 3.421]

    p = 2 ** L
    c = 0.7 - 0.8 / L + (4 + 32 / L) * K ** (-3 / L) / 15
    sigma = c * math.sqrt(variance[L]/K)
    sqrt2 = math.sqrt(2)
    phi_sum = 0.0
    T = [0] * (Q)
    for i in range(p):
        T[i] = 0
    for i in range(Q + 1):
        decRep = 0
        for j in range(L):
            decRep += int(ga_array[(i - 1) * L + j]) * 2 ** (L - 1 - j)
        T[decRep] = i


    i = Q + 1
    while i <= (Q + K):
        decRep = 0
        for j in range(L):
            decRep += int(ga_array[(i - 1) * L + j]) * 2 ** (L - 1 - j)
        phi_sum += math.log(i - T[decRep]) / math.log(2)
        T[decRep] = i
        i += 1

    phi = phi_sum / K
    arg = abs(phi - expected_value[L]) / (sqrt2 * sigma)
    p_value = math.erfc(arg)
    return p_value
```

A.9. Serial Test

```
import scipy.special
def serial(ga_array):
    m = 2
    n = len(ga_array)

    def psi2(m, n):
        if m == 0 or m == -1:
            return 0.0
        nblocks = n
        powLen = 2 ** (m+1) - 1
        P = [0] * powLen

        i = 1
        while i < (powLen - 1):
            P[i] = 0
            i += 1


        for i in range(nblocks):
            k = 1
            for j in range(m):
                if int(ga_array[(i+j)%n]) == 0:
                    k *= 2
                elif int(ga_array[(i+j)%n]) == 1:
                    k = 2*k+1
```

```
            P[k-1] += 1

        psi2_sum = 0
        i = 2 ** m - 1
        while i < (2 ** (m+1) - 1):
            psi2_sum += P[i] ** 2
            i += 1
        psi2_sum = ((psi2_sum * 2 ** m) / n) - n
        return psi2_sum

    psim0 = psi2(m,n)
    psim1 = psi2(m-1,n)
    psim2 = psi2(m-2, n)
    del1 = psim0 - psim1
    del2 = psim0 - 2.0*psim1 + psim2
    p_value1 = scipy.special.gammaincc(2 ** (m-1)/2, del1/2.0)
    p_value2 = scipy.special.gammaincc(2 ** (m-2)/2, del2/2.0)
    return p_value1, p_value2
```

A.10. Approximate Entropy Test

```
import math
import scipy.special
def approximate_entrophy(ga_array):
    #m = 1 for 64 and 128, 2 otherwise
    m = 2
    n = len(ga_array)

    ApEn = [0] * 2
    r = 0
    blocksize = m
    while blocksize <= (m+1):
        if blocksize == 0:
            ApEn[0] = 0.00
            r += 1
        else:
            nblocks = n
            powLen = 2 ** (blocksize + 1) - 1
            P = [0] * powLen
            i = 1
            while i < (powLen - 1):
                P[i] = 0
                i += 1
            for i in range(nblocks):
                k = 1
                for j in range(blocksize):
                    k <<= 1
                    if int(ga_array[(i+j)%n]) == 1:
                        k += 1
                P[k-1] += 1

            apen_sum = 0
            index = 2 ** blocksize - 1
            for i in range(2 ** blocksize):
                if P[index] > 0:
                    apen_sum += P[index] * math.log(P[index]/nblocks)
                index += 1

            apen_sum /= nblocks
            ApEn[r] = apen_sum
            r += 1
        blocksize += 1

    apen = ApEn[0] - ApEn[1]
```

```
    chi_2 = 2.0 * n * (math.log(2) - apen)
    p_value = scipy.special.gammaincc(2 ** (m-1), chi_2/2.0)
    return p_value
```

A.11. Cumulative Sums (Cusum) Test

```
import math
def cumulative_sums(ga_array):
    n = len(ga_array)
    S = 0
    sup = 0
    inf = 0

    def cephes_normal(x):
        if x > 0:
            arg = x / math.sqrt(2)
            result = 0.5 * (1 + math.erf(arg))
        else:
            arg = -x / math.sqrt(2)
            result = 0.5 * (1 - math.erf(arg))
        return result

    for k in range(n):
        S = S + 1 if int(ga_array[k]) else S - 1
        if S > sup:
            sup += 1
        if S < inf:
            inf -= 1
        z = sup if sup > -inf else -inf
        zrev = sup - S if sup - S > S - inf else S - inf


    sumf1 = 0.0
    k = (-n / z + 1) // 4
    while k <= ((n / z) - 1) // 4:
        sumf1 += cephes_normal(((4 * k + 1)*z)/math.sqrt(n))
        sumf1 -= cephes_normal(((4 * k - 1)*z)/math.sqrt(n))
        k += 1

    sumf2 = 0.0
    k  = (-n / z - 3) // 4
    while k <= ((n/z)-1) // 4:
        sumf2 += cephes_normal(((4 * k+ 3)* z)/math.sqrt(n))
        sumf2 -= cephes_normal(((4 * k + 1)* z)/math.sqrt(n))
        k += 1

    sumb1 = 0.0
    k = (-n / zrev + 1) // 4
    while k <= (n / zrev - 1) // 4:
        sumb1 += cephes_normal(((4 * k + 1) * zrev) / math.sqrt(n))
        sumb1 -= cephes_normal(((4 * k - 1) * zrev) / math.sqrt(n))
        k += 1

    sumb2 = 0.0
    k = (-n / zrev - 3) // 4
    while k <= (n / zrev - 1) // 4:
        sumb2 += cephes_normal(((4 * k + 3) * zrev) / math.sqrt(n))
        sumb2 -= cephes_normal(((4 * k + 1) * zrev) / math.sqrt(n))
        k += 1

    p_valuef = 1.0 - sumf1 + sumf2
    p_valueb = 1.0 - sumb1 + sumb2
    return p_valuef, p_valueb
```

A.12. Random Excursions Test

```
import numpy
import scipy.special
def random_excursions(ga_array):
    n = len(ga_array)
    J = 0
    S_k = [0] * n
    cycle = [0] * n
    nu = numpy.ndarray(shape=(6, 8), dtype=int)
    pi = [[0.0000000000, 0.00000000000, 0.00000000000, 0.00000000000,
      0.00000000000, 0.0000000000],
          [0.5000000000, 0.25000000000, 0.12500000000, 0.06250000000,
      0.03125000000, 0.0312500000],
          [0.7500000000, 0.06250000000, 0.04687500000, 0.03515625000,
      0.02636718750, 0.0791015625],
          [0.8333333333, 0.02777777778, 0.02314814815, 0.01929012346,
      0.01607510288, 0.0803755143],
          [0.8750000000, 0.01562500000, 0.01367187500, 0.01196289063,
      0.01046752930, 0.0732727051]]
    stateX = [-4, -3, -2, -1, 1, 2, 3, 4]
    counter = [0, 0, 0, 0, 0, 0, 0, 0]

    S_k[0] = 2 * int(ga_array[0]) - 1
    i = 1
    while i < n:
        S_k[i] = S_k[i - 1] + 2 * int(ga_array[i]) - 1
        if S_k[i] == 0:
            J += 1
            cycle[J] = i
        i += 1

    if S_k[n - 1] != 0:
        J += 1
    cycle[J] = n

    cycleStart = 0
    cycleStop = cycle[1]
    for k in range(6):
        for i in range(8):
            nu[k, i] = 0
    j = 1
    while j <= J:
        for i in range(8):
            counter[i] = 0
        i = cycleStart
        while i < cycleStop:
            if (S_k[i] >= 1 and S_k[i] <= 4) or (S_k[i] >= -4 and S_k[i] <= -
      1):
                if S_k[i] < 0:
                    b = 4
                else:
                    b = 3
                counter[S_k[i] + b] += 1
            i += 1

        cycleStart = cycle[j] + 1
        if j < J:
            cycleStop = cycle[j + 1]

        for i in range(8):
            if counter[i] >= 0 and counter[i] <= 4:
                nu[counter[i], i] += 1
            elif counter[i] >= 5:
```

```
                nu[5, i] += 1
        j += 1

    p_list = [0.0] * 8
    for i in range(8):
        x = stateX[i]
        chi2 = 0.0
        for k in range(6):
            chi2 += (nu[k, i] - J * pi[abs(x)][k]) ** 2 / (J * pi[abs(x)][k])
        p_list[i] = scipy.special.gammaincc(2.5, chi2/2.0)

    p_value_sum = 0.0
    for x in p_list:
        if x < 0.01:
            p_value_sum = 0.0
            break
        else:
            p_value_sum += x

    return p_value_sum
```

A.13. Random Excursions Variant Test

```
import math
def random_excursions_variant(ga_array):
    n = len(ga_array)
    stateX = [-9, -8, -7, -6, -5, -4, -3, -2, -1, 1, 2, 3, 4, 5, 6, 7, 8, 9]

    J = 0
    S_k = [0] * n
    S_k[0] = 2 * int(ga_array[0]) - 1
    i = 1
    while i < n:
        S_k[i] = S_k[i - 1] + 2 * int(ga_array[i]) - 1
        if S_k[i] == 0:
            J += 1
        i += 1

    if S_k[n - 1] != 0:
        J += 1

    p_list = [0] * 18
    for p in range(18):
        x = stateX[p]
        count = 0
        for i in range(n):
            if S_k[i] == x:
                count += 1
        p_list[p] = math.erfc(abs(count-J)/(math.sqrt(2.0 * J * (4.0 * abs(x)
      - 2))))

    p_value_sum = 0.0
    for x in p_list:
        if x < 0.01:
            p_value_sum = 0.0
            break
        else:
            p_value_sum += x


    return p_value_sum
```

Appendix B – Genetic Algorithms

This is the code for the genetic algorithms. The code creates the keys and then stores the best-found key as well as the time it takes to run the genetic algorithm in a file.

B.1. Frequency (Monobit) Genetic Algorithm

```
from geneticalgorithm import geneticalgorithm as ga
import pickle
import timer
from nistsp80022tests import freq_monobit

def f(ga_array):
    freq_monobit_p_value = freq_monobit(ga_array)
    return -freq_monobit_p_value


algorithm_param = {
    'max_num_iteration': 500,
    'population_size': 17,
    'mutation_probability': 0.05,
    'elit_ratio': 1/17,
    'crossover_probability': 0.25,
    'parents_portion': 0.15,
    'crossover_type': 'two_point',
    'max_iteration_without_improv': None}


model=ga(
    function=f,
    dimension=256,
    variable_type='bool',
    algorithm_parameters=algorithm_param,
    convergence_curve=False
)

output_dict_key = {}
output_dict_p_value= {}
time_lst = []
t = timer.Timer()
for x in range(1, 11):
    t.start()
    model.run()
    time_lst.append(t.stop())
    output_key = "".join(str(x) for x in
model.output_dict["variable"].astype("int"))
    output_dict_key[x] = output_key
    output_dict_p_value[x] = model.output_dict["function"]

with open("freq_monobit_bin.txt", "wb") as f1:
    pickle.dump([output_dict_key, output_dict_p_value, time_lst], f1)
```

B.2. Frequency Test Within a Block Genetic Algorithm

```
from geneticalgorithm import geneticalgorithm as ga
import pickle
import timer
from nistsp80022tests import block_freq

def f(ga_array):
```

```
    block_freq_p_value = block_freq(ga_array)
    return -block_freq_p_value


algorithm_param = {
    'max_num_iteration': 500,
    'population_size': 17,
    'mutation_probability': 0.05,
    'elit_ratio': 1/17,
    'crossover_probability': 0.25,
    'parents_portion': 0.15,
    'crossover_type': 'two_point',
    'max_iteration_without_improv': None}

model=ga(
    function=f,
    dimension=256,
    variable_type='bool',
    algorithm_parameters=algorithm_param,
    convergence_curve=False
)

output_dict_key = {}
output_dict_p_value= {}
time_lst = []
t = timer.Timer()
for x in range(1, 11):
    t.start()
    model.run()
    time_lst.append(t.stop())
    output_key = "".join(str(x) for x in
model.output_dict["variable"].astype("int"))
    output_dict_key[x] = output_key
    output_dict_p_value[x] = model.output_dict["function"]

with open("freq_block_bin.txt", "wb") as f1:
    pickle.dump([output_dict_key, output_dict_p_value, time_lst], f1)
```

B.3. Runs Test Genetic Algorithm

```
from geneticalgorithm import geneticalgorithm as ga
import pickle
import timer
from nistsp80022tests import runs

def f(ga_array):
    runs_p_value = runs(ga_array)
    return -runs_p_value


algorithm_param = {
    'max_num_iteration': 500,
    'population_size': 17,
    'mutation_probability': 0.05,
    'elit_ratio': 1/17,
    'crossover_probability': 0.25,
    'parents_portion': 3/17,
    'crossover_type': 'two_point',
    'max_iteration_without_improv': None}


model=ga(
    function=f,
    dimension=256,
```

```
    variable_type='bool',
    algorithm_parameters=algorithm_param,
    convergence_curve=False
)

# model.run()
output_dict_key = {}
output_dict_p_value= {}
time_lst = []
t = timer.Timer()
for x in range(1, 11):
    t.start()
    model.run()
    time_lst.append(t.stop())
    output_key = "".join(str(x) for x in
model.output_dict["variable"].astype("int"))
    output_dict_key[x] = output_key
    output_dict_p_value[x] = model.output_dict["function"]

with open("runs_bin.txt", "wb") as f1:
    pickle.dump([output_dict_key, output_dict_p_value, time_lst], f1)
```

B.4. Test for the Longest Run of Ones in a Block Genetic Algorithm

```
from geneticalgorithm import geneticalgorithm as ga
import pickle
import timer
from nistsp80022tests import longestrunofones

def f(ga_array):
    lroo_p_value = longestrunofones(ga_array)
    return -lroo_p_value


algorithm_param = {
    'max_num_iteration': 500,
    'population_size': 17,
    'mutation_probability': 0.05,
    'elit_ratio': 1/17,
    'crossover_probability': 0.25,
    'parents_portion': 3/17,
    'crossover_type': 'two_point',
    'max_iteration_without_improv': None}


model=ga(
    function=f,
    dimension=256,
    variable_type='bool',
    algorithm_parameters=algorithm_param,
    convergence_curve=False
)

# model.run()
output_dict_key = {}
output_dict_p_value= {}
time_lst = []
t = timer.Timer()
for x in range(1, 11):
    t.start()
    model.run()
    time_lst.append(t.stop())
    output_key = "".join(str(x) for x in
model.output_dict["variable"].astype("int"))
```

```
    output_dict_key[x] = output_key
    output_dict_p_value[x] = model.output_dict["function"]

with open("lroo_bin.txt", "wb") as f1:
    pickle.dump([output_dict_key, output_dict_p_value, time_lst], f1)
```

B.5. Binary Matrix Rank Test Genetic Algorithm

```
from geneticalgorithm import geneticalgorithm as ga
from nistsp80022tests import binary_matrix_rank
import pickle
import timer

def f(ga_array):
    binary_matrix_rank_p_value = binary_matrix_rank(ga_array)
    return -binary_matrix_rank_p_value


algorithm_param = {
    'max_num_iteration': 500,
    'population_size': 17,
    'mutation_probability': 0.05,
    'elit_ratio': 1/17,
    'crossover_probability': 0.25,
    'parents_portion': 3/17,
    'crossover_type': 'two_point',
    'max_iteration_without_improv': None}


model=ga(
    function=f,
    dimension=256,
    variable_type='bool',
    algorithm_parameters=algorithm_param,
    convergence_curve=False
)

# model.run()
output_dict_key = {}
output_dict_p_value= {}
time_lst = []
t = timer.Timer()
for x in range(1, 11):
    t.start()
    model.run()
    time_lst.append(t.stop())
    output_key = "".join(str(x) for x in
model.output_dict["variable"].astype("int"))
    output_dict_key[x] = output_key
    output_dict_p_value[x] = model.output_dict["function"]

with open("bmr_bin.txt", "wb") as f1:
    pickle.dump([output_dict_key, output_dict_p_value, time_lst], f1)
```

B.6. Discrete Fourier Transform (Spectral) Test Genetic Algorithm

```
from geneticalgorithm import geneticalgorithm as ga
from nistsp80022tests import discrete_fourier_transform
import pickle
import timer

def f(ga_array):
    discrete_fourier_transform_p_value = discrete_fourier_transform(ga_array)
    return -discrete_fourier_transform_p_value
```

```
algorithm_param = {
    'max_num_iteration': 500,
    'population_size': 17,
    'mutation_probability': 0.05,
    'elit_ratio': 1/17,
    'crossover_probability': 0.25,
    'parents_portion': 3/17,
    'crossover_type': 'two_point',
    'max_iteration_without_improv': None}


model=ga(
    function=f,
    dimension=256,
    variable_type='bool',
    algorithm_parameters=algorithm_param,
    convergence_curve=False
)

# model.run()
output_dict_key = {}
output_dict_p_value= {}
time_lst = []
t = timer.Timer()
for x in range(1, 11):
    t.start()
    model.run()
    time_lst.append(t.stop())
    output_key = "".join(str(x) for x in
model.output_dict["variable"].astype("int"))
    output_dict_key[x] = output_key
    output_dict_p_value[x] = model.output_dict["function"]

with open("dft_bin.txt", "wb") as f1:
    pickle.dump([output_dict_key, output_dict_p_value, time_lst], f1)
```

B.7. Non-overlapping Template Matching Test Genetic Algorithm

```
from geneticalgorithm import geneticalgorithm as ga
from nistsp80022tests import nonOverlappingTemplateMatchings
import pickle
import timer

def f(ga_array):
    nonOverlappingTemplateMatchings_p_value =
nonOverlappingTemplateMatchings(ga_array)
    return -nonOverlappingTemplateMatchings_p_value


algorithm_param = {
    'max_num_iteration': 500,
    'population_size': 17,
    'mutation_probability': 0.05,
    'elit_ratio': 1/17,
    'crossover_probability': 0.25,
    'parents_portion': 3/17,
    'crossover_type': 'two_point',
    'max_iteration_without_improv': None}


model=ga(
    function=f,
```

```
    dimension=256,
    variable_type='bool',
    algorithm_parameters=algorithm_param,
    convergence_curve=False
)

# model.run()
output_dict_key = {}
output_dict_p_value= {}
time_lst = []
t = timer.Timer()
for x in range(1, 11):
    t.start()
    model.run()
    time_lst.append(t.stop())
    output_key = "".join(str(x) for x in
model.output_dict["variable"].astype("int"))
    output_dict_key[x] = output_key
    output_dict_p_value[x] = model.output_dict["function"]

with open("notm_bin.txt", "wb") as f1:
    pickle.dump([output_dict_key, output_dict_p_value, time_lst], f1)
```

B.8. Maurer's "Universal Statistical" Test Genetic Algorithm

```
from geneticalgorithm import geneticalgorithm as ga
from nistsp800_22tests import universal
import pickle
import timer

def f(ga_array):
    universal_p_value = universal(ga_array)
    return -universal_p_value


algorithm_param = {
    'max_num_iteration': 500,
    'population_size': 17,
    'mutation_probability': 0.05,
    'elit_ratio': 1/17,
    'crossover_probability': 0.25,
    'parents_portion': 3/17,
    'crossover_type': 'two_point',
    'max_iteration_without_improv': None}


model=ga(
    function=f,
    dimension=256,
    variable_type='bool',
    algorithm_parameters=algorithm_param,
    convergence_curve=False
)

# model.run()
output_dict_key = {}
output_dict_p_value= {}
time_lst = []
t = timer.Timer()
for x in range(1, 11):
    t.start()
    model.run()
    time_lst.append(t.stop())
```

```
    output_key = "".join(str(x) for x in
model.output_dict["variable"].astype("int"))
    output_dict_key[x] = output_key
    output_dict_p_value[x] = model.output_dict["function"]

with open("universal_bin.txt", "wb") as f1:
    pickle.dump([output_dict_key, output_dict_p_value, time_lst], f1)
```

B.9. Serial Test Genetic Algorithm

```
from geneticalgorithm import geneticalgorithm as ga
from nistsp80022tests import serial
import pickle
import timer

def f(ga_array):
    serial_p_value1, serial_p_value2 = serial(ga_array)
    return -(serial_p_value1 + serial_p_value2)


algorithm_param = {
    'max_num_iteration': 500,
    'population_size': 17,
    'mutation_probability': 0.05,
    'elit_ratio': 1/17,
    'crossover_probability': 0.25,
    'parents_portion': 3/17,
    'crossover_type': 'two_point',
    'max_iteration_without_improv': None}


model=ga(
    function=f,
    dimension=256,
    variable_type='bool',
    algorithm_parameters=algorithm_param,
    convergence_curve=False
)

# model.run()
output_dict_key = {}
output_dict_p_value= {}
time_lst = []
t = timer.Timer()
for x in range(1, 11):
    t.start()
    model.run()
    time_lst.append(t.stop())
    output_key = "".join(str(x) for x in
model.output_dict["variable"].astype("int"))
    output_dict_key[x] = output_key
    output_dict_p_value[x] = model.output_dict["function"]

with open("serial_bin.txt", "wb") as f1:
    pickle.dump([output_dict_key, output_dict_p_value, time_lst], f1)
```

B.10. Approximate Entropy Test Genetic Algorithm

```
from geneticalgorithm import geneticalgorithm as ga
from nistsp80022tests import approximate_entrophy
import pickle
import timer

def f(ga_array):
```

```
    approximate_entrophy_p_value = approximate_entrophy(ga_array)
    return -approximate_entrophy_p_value


algorithm_param = {
    'max_num_iteration': 500,
    'population_size': 17,
    'mutation_probability': 0.05,
    'elit_ratio': 1/17,
    'crossover_probability': 0.25,
    'parents_portion': 3/17,
    'crossover_type': 'two_point',
    'max_iteration_without_improv': None}


model=ga(
    function=f,
    dimension=256,
    variable_type='bool',
    algorithm_parameters=algorithm_param,
    convergence_curve=False
)

# model.run()
output_dict_key = {}
output_dict_p_value= {}
time_lst = []
t = timer.Timer()
for x in range(1, 11):
    t.start()
    model.run()
    time_lst.append(t.stop())
    output_key = "".join(str(x) for x in
model.output_dict["variable"].astype("int"))
    output_dict_key[x] = output_key
    output_dict_p_value[x] = model.output_dict["function"]

with open("aent_bin.txt", "wb") as f1:
    pickle.dump([output_dict_key, output_dict_p_value, time_lst], f1)
```

B.11. Cumulative Sums (Cusum) Test Genetic Algorithm

```
from geneticalgorithm import geneticalgorithm as ga
from nistsp80022tests import cumulative_sums
import pickle
import timer

def f(ga_array):
    cusum_p_value1, cusum_p_value2 = cumulative_sums(ga_array)
    return -(cusum_p_value1 + cusum_p_value2)


algorithm_param = {
    'max_num_iteration': 500,
    'population_size': 17,
    'mutation_probability': 0.05,
    'elit_ratio': 1/17,
    'crossover_probability': 0.25,
    'parents_portion': 3/17,
    'crossover_type': 'two_point',
    'max_iteration_without_improv': None}


model=ga(
```

```
    function=f,
    dimension=256,
    variable_type='bool',
    algorithm_parameters=algorithm_param,
    convergence_curve=False
)

# model.run()
output_dict_key = {}
output_dict_p_value= {}
time_lst = []
t = timer.Timer()
for x in range(1, 11):
    t.start()
    model.run()
    time_lst.append(t.stop())
    output_key = "".join(str(x) for x in
model.output_dict["variable"].astype("int"))
    output_dict_key[x] = output_key
    output_dict_p_value[x] = model.output_dict["function"]

with open("cusum_bin.txt", "wb") as f1:
    pickle.dump([output_dict_key, output_dict_p_value, time_lst], f1)
```

B.12. Random Excursions Test Genetic Algorithm

```
from geneticalgorithm import geneticalgorithm as ga
from nistsp80022tests import random_excursions
import pickle
import timer

def f(ga_array):
    random_excursions_p_value = random_excursions(ga_array)
    return -random_excursions_p_value


algorithm_param = {
    'max_num_iteration': 500,
    'population_size': 17,
    'mutation_probability': 0.05,
    'elit_ratio': 1/17,
    'crossover_probability': 0.25,
    'parents_portion': 3/17,
    'crossover_type': 'two_point',
    'max_iteration_without_improv': None}


model=ga(
    function=f,
    dimension=256,
    variable_type='bool',
    algorithm_parameters=algorithm_param,
    convergence_curve=False
)

# model.run()
output_dict_key = {}
output_dict_p_value= {}
time_lst = []
t = timer.Timer()
for x in range(1, 11):
    t.start()
    model.run()
    time_lst.append(t.stop())
```

```
    output_key = "".join(str(x) for x in
model.output_dict["variable"].astype("int"))
    output_dict_key[x] = output_key
    output_dict_p_value[x] = model.output_dict["function"]

with open("ranexc_bin.txt", "wb") as f1:
    pickle.dump([output_dict_key, output_dict_p_value, time_lst], f1)
```

B.13. Random Excursions Variant Test Genetic Algorithm

```
from geneticalgorithm import geneticalgorithm as ga
from nistsp80022tests import random_excursions_variant
import pickle
import timer

def f(ga_array):
    random_excursions_variant_p_value = random_excursions_variant(ga_array)
    return -random_excursions_variant_p_value

algorithm_param = {
    'max_num_iteration': 500,
    'population_size': 17,
    'mutation_probability': 0.05,
    'elit_ratio': 1/17,
    'crossover_probability': 0.25,
    'parents_portion': 3/17,
    'crossover_type': 'two_point',
    'max_iteration_without_improv': None}


model=ga(
    function=f,
    dimension=256,
    variable_type='bool',
    algorithm_parameters=algorithm_param,
    convergence_curve=False
)

# model.run()
output_dict_key = {}
output_dict_p_value= {}
time_lst = []
t = timer.Timer()
for x in range(1, 11):
    t.start()
    model.run()
    time_lst.append(t.stop())
    output_key = "".join(str(x) for x in
model.output_dict["variable"].astype("int"))
    output_dict_key[x] = output_key
    output_dict_p_value[x] = model.output_dict["function"]

with open("ranexcvar_bin.txt", "wb") as f1:
    pickle.dump([output_dict_key, output_dict_p_value, time_lst], f1)
```

Appendix C – Evaluation Tests

This appendix contains the code for the timer and the functions to find the Hamming distance between keys and the average p-value of the keys.

C.1. Timer

```
import time

class Timer():
    def __init__(self):
        self.start_time = None

    def start(self):
        assert self.start_time == None
        self.start_time = time.perf_counter()

    def stop(self):
        assert self.start_time != None
        tot_time = time.perf_counter() - self.start_time
        self.start_time = None
        return (f"{tot_time:0.3f}")
```

C.2. Hamming Distance

```
def hamming_dist(x, y):
    assert len(x) == len(y)

    hamming = 0
    for i in range(len(x)):
        char_xor = ord(x[i]) ^ ord(y[i])
        hamming += bin(char_xor).count("1")

    return hamming
def hamming_test_binary(input):
    ham_list_1 = []
    ham_list_2 = []
    ham_list_3 = []
    ham_list_4 = []
    ham_list_5 = []
    ham_list_6 = []
    ham_list_7 = []
    ham_list_8 = []
    ham_list_9 = []
    ham_list_10 = []

    for x in range(1, 11):
        if x == 1:
            for i in range(1, 11):
                ham_list_1.append(hamming_dist(input.get(x), input.get(i)))

        if x == 2:
            for i in range(1, 11):
                ham_list_2.append(hamming_dist(input.get(x), input.get(i)))

        if x == 3:
            for i in range(1, 11):
                ham_list_3.append(hamming_dist(input.get(x), input.get(i)))

        if x == 4:
```

```
            for i in range(1, 11):
                ham_list_4.append(hamming_dist(input.get(x), input.get(i)))

        if x == 5:
            for i in range(1, 11):
                ham_list_5.append(hamming_dist(input.get(x), input.get(i)))

        if x == 6:
            for i in range(1, 11):
                ham_list_6.append(hamming_dist(input.get(x), input.get(i)))

        if x == 7:
            for i in range(1, 11):
                ham_list_7.append(hamming_dist(input.get(x), input.get(i)))

        if x == 8:
            for i in range(1, 11):
                ham_list_8.append(hamming_dist(input.get(x), input.get(i)))

        if x == 9:
            for i in range(1, 11):
                ham_list_9.append(hamming_dist(input.get(x), input.get(i)))

        if x == 10:
            for i in range(1, 11):
                ham_list_10.append(hamming_dist(input.get(x), input.get(i)))

    sum_list = [ham_list_1, ham_list_2, ham_list_3, ham_list_4, ham_list_5,
ham_list_6,
                ham_list_7, ham_list_8, ham_list_9, ham_list_10]

    avg_list = []
    for x in sum_list:
        average = sum(i for i in x)
        avg_list.append(average)

    return sum(avg_list) / 100
```

C.3. Average P-value

```
def avg_pvalue(x):
    return sum(x.values()) / len(x)
```

Appendix D – Sensitive Analysis

This appendix contains the scripts that were used to conduct the sensitive analyses.

D.1. Top Two Hamming Distance Performers for 256-bit Key Generation Sensitive Analysis

```
from nistsp80022tests import runs, cumulative_sums
from geneticalgorithm import geneticalgorithm as ga
from hammingtest import hamming_dist
from numpy import arange
import timer

class Sensitivity_Analysis():
    def __init__(self, fitness_functions, w1, w2):
        self.fitness_functions = fitness_functions
        self.w1 = w1
        self.w2 = w2


    def f(self, ga_array):
        p_value_list = []
        for fitness_function in self.fitness_functions:
            p_value_list.append(fitness_function(ga_array))

        total_p_value = (self.w1 * p_value_list[0]) + (self.w2 *
(p_value_list[1][0] + p_value_list[1][1]))
        return -total_p_value

    def genetic_algorithm(self):
        algorithm_param = {
            'max_num_iteration': 500,
            'population_size': 17,
            'mutation_probability': 0.05,
            'elit_ratio': 1 / 17,
            'crossover_probability': 0.25,
            'parents_portion': 3 / 17,
            'crossover_type': 'two_point',
            'max_iteration_without_improv': None}

        model = ga(
            function=self.f,
            dimension=2048,
            variable_type='bool',
            algorithm_parameters=algorithm_param,
            convergence_curve=False
        )

        t = timer.Timer()

        t.start()
        model.run()
        output_time = t.stop()

        output_key = "".join(str(x) for x in
model.output_dict["variable"].astype("int"))

        output_p_value = model.output_dict["function"]
```

```
        return {"key": output_key, "time": output_time, "p_value":
output_p_value}

    def evaluation(self):
        keys = []
        times = []
        p_values = []
        for i in range(2):
            ga_output = self.genetic_algorithm()
            keys.append(ga_output["key"])
            times.append(ga_output["time"])
            p_values.append(ga_output["p_value"])

        hamming_distance = hamming_dist(keys[0], keys[1])
        avg_time = sum([float(x) for x in times]) / 2
        avg_p_value = sum(p_values) / 2

        eval_value = hamming_distance + (1/avg_time)

        return {"eval_value": eval_value, "hamming_distance":
hamming_distance,
                "avg_time": avg_time, "avg_p_value": avg_p_value}

if __name__ == "__main__":

    best_eval_value = 0.0
    for w1 in arange(0.0, 4/3, 1/3):
        for w2 in arange(0.0, 4/3, 1/3):
            sa = Sensitivity_Analysis([runs, cumulative_sums],
                                      w1, w2)
            eval_output = sa.evaluation()
            if eval_output["eval_value"] > best_eval_value:
                best_weights = {"w1": w1, "w2": w2}
                best_hamming_distance = eval_output["hamming_distance"]
                best_avg_time = eval_output["avg_time"]
                best_avg_p_value = eval_output["avg_p_value"]
                best_eval_value = eval_output["eval_value"]

    print(f"\n\nBest Values:\nWeights: {best_weights}\nHamming Distance:
{best_hamming_distance}\n"
          f"Time: {best_avg_time}\nP_value: {best_avg_p_value}")
```

D.2. Top Two Time Performers for 256-bit Key Generation Sensitive Analysis

```
from nistsp80022tests import block_freq, freq_monobit
from geneticalgorithm import geneticalgorithm as ga
from hammingtest import hamming_dist
from numpy import arange
import timer

class Sensitivity_Analysis():
    def __init__(self, fitness_functions, w1, w2):
        self.fitness_functions = fitness_functions
        self.w1 = w1
        self.w2 = w2

    def f(self, ga_array):
        p_value_list = []
        for fitness_function in self.fitness_functions:
            p_value_list.append(fitness_function(ga_array))

        total_p_value = (self.w1 * p_value_list[0]) + (self.w2 *
p_value_list[1])
        return -total_p_value
```

```
    def genetic_algorithm(self):
        algorithm_param = {
            'max_num_iteration': 500,
            'population_size': 17,
            'mutation_probability': 0.05,
            'elit_ratio': 1 / 17,
            'crossover_probability': 0.25,
            'parents_portion': 3 / 17,
            'crossover_type': 'two_point',
            'max_iteration_without_improv': None}

        model = ga(
            function=self.f,
            dimension=2048,
            variable_type='bool',
            algorithm_parameters=algorithm_param,
            convergence_curve=False
        )

        t = timer.Timer()

        t.start()
        model.run()
        output_time = t.stop()

        output_key = "".join(str(x) for x in
model.output_dict["variable"].astype("int"))

        output_p_value = model.output_dict["function"]

        return {"key": output_key, "time": output_time, "p_value":
output_p_value}

    def evaluation(self):
        keys = []
        times = []
        p_values = []
        for i in range(2):
            ga_output = self.genetic_algorithm()
            keys.append(ga_output["key"])
            times.append(ga_output["time"])
            p_values.append(ga_output["p_value"])

        hamming_distance = hamming_dist(keys[0], keys[1])
        avg_time = sum([float(x) for x in times]) / 2
        avg_p_value = sum(p_values) / 2

        eval_value = hamming_distance + (1/avg_time)

        return {"eval_value": eval_value, "hamming_distance":
hamming_distance,
                "avg_time": avg_time, "avg_p_value": avg_p_value}

if __name__ == "__main__":

    best_eval_value = 0.0
    for w1 in arange(0.0, 4/3, 1/3):
        for w2 in arange(0.0, 4/3, 1/3):
            sa = Sensitivity_Analysis([block_freq, freq_monobit],
                                      w1, w2)
            eval_output = sa.evaluation()
            if eval_output["eval_value"] > best_eval_value:
                best_weights = {"w1": w1, "w2": w2}
```

```
                best_hamming_distance = eval_output["hamming_distance"]
                best_avg_time = eval_output["avg_time"]
                best_avg_p_value = eval_output["avg_p_value"]
                best_eval_value = eval_output["eval_value"]

    print(f"\n\nBest Values:\nWeights: {best_weights}\nHamming Distance:
{best_hamming_distance}\n"
          f"Time: {best_avg_time}\nP_value: {best_avg_p_value}")
```

D.3. Top Three Hamming Distance Performers for 256-bit Key Generation Sensitive Analysis

```
from nistsp80022tests import runs, cumulative_sums, approximate_entrophy
from geneticalgorithm import geneticalgorithm as ga
from hammingtest import hamming_dist
from numpy import arange
import timer

class Sensitivity_Analysis():
    def __init__(self, fitness_functions, w1, w2, w3):
        self.fitness_functions = fitness_functions
        self.w1 = w1
        self.w2 = w2
        self.w3 = w3

    def f(self, ga_array):
        p_value_list = []
        for fitness_function in self.fitness_functions:
            p_value_list.append(fitness_function(ga_array))

        total_p_value = (self.w1 * p_value_list[0]) + (self.w2 *
(p_value_list[1][0] + p_value_list[1][1])) + \
                        (self.w3 * p_value_list[2])
        return -total_p_value

    def genetic_algorithm(self):
        algorithm_param = {
            'max_num_iteration': 500,
            'population_size': 17,
            'mutation_probability': 0.05,
            'elit_ratio': 1 / 17,
            'crossover_probability': 0.25,
            'parents_portion': 3 / 17,
            'crossover_type': 'two_point',
            'max_iteration_without_improv': None}

        model = ga(
            function=self.f,
            dimension=256,
            variable_type='bool',
            algorithm_parameters=algorithm_param,
            convergence_curve=False
        )

        t = timer.Timer()

        t.start()
        model.run()
        output_time = t.stop()

        output_key = "".join(str(x) for x in
model.output_dict["variable"].astype("int"))
```

```
        output_p_value = model.output_dict["function"]

        return {"key": output_key, "time": output_time, "p_value":
output_p_value}

    def evaluation(self):
        keys = []
        times = []
        p_values = []
        for i in range(2):
            ga_output = self.genetic_algorithm()
            keys.append(ga_output["key"])
            times.append(ga_output["time"])
            p_values.append(ga_output["p_value"])

        hamming_distance = hamming_dist(keys[0], keys[1])
        avg_time = sum([float(x) for x in times]) / 2
        avg_p_value = sum(p_values) / 2

        eval_value = hamming_distance + (1/avg_time)

        return {"eval_value": eval_value, "hamming_distance":
hamming_distance,
                "avg_time": avg_time, "avg_p_value": avg_p_value}

if __name__ == "__main__":

    best_eval_value = 0.0
    for w1 in arange(0.0, 4/3, 1/3):
        for w2 in arange(0.0, 4/3, 1/3):
            for w3 in arange(0.0, 4/3, 1/3):
                sa = Sensitivity_Analysis([runs, cumulative_sums,
approximate_entrophy],
                                          w1, w2, w3)
                eval_output = sa.evaluation()
                if eval_output["eval_value"] > best_eval_value:
                    best_weights = {"w1": w1, "w2": w2, "w3": w3}
                    best_hamming_distance = eval_output["hamming_distance"]
                    best_avg_time = eval_output["avg_time"]
                    best_avg_p_value = eval_output["avg_p_value"]
                    best_eval_value = eval_output["eval_value"]

    print(f"\n\nBest Values:\nWeights: {best_weights}\nHamming Distance:
{best_hamming_distance}\n"
          f"Time: {best_avg_time}\nP_value: {best_avg_p_value}")
```

D.4. Top Three Time Performers for 256-bit Key Generation Sensitive Analysis

```
from nistsp80022tests import block_freq, freq_monobit, runs
from geneticalgorithm import geneticalgorithm as ga
from hammingtest import hamming_dist
from numpy import arange
import timer

class Sensitivity_Analysis():
    def __init__(self, fitness_functions, w1, w2, w3):
        self.fitness_functions = fitness_functions
        self.w1 = w1
        self.w2 = w2
        self.w3 = w3

    def f(self, ga_array):
        p_value_list = []
```

```
        for fitness_function in self.fitness_functions:
            p_value_list.append(fitness_function(ga_array))

        total_p_value = (self.w1 * p_value_list[0]) + (self.w2 *
p_value_list[1]) + (self.w3 * p_value_list[2])

        return -total_p_value

    def genetic_algorithm(self):
        algorithm_param = {
            'max_num_iteration': 500,
            'population_size': 17,
            'mutation_probability': 0.05,
            'elit_ratio': 1 / 17,
            'crossover_probability': 0.25,
            'parents_portion': 3 / 17,
            'crossover_type': 'two_point',
            'max_iteration_without_improv': None}

        model = ga(
            function=self.f,
            dimension=256,
            variable_type='bool',
            algorithm_parameters=algorithm_param,
            convergence_curve=False
        )

        t = timer.Timer()

        t.start()
        model.run()
        output_time = t.stop()

        output_key = "".join(str(x) for x in
model.output_dict["variable"].astype("int"))

        output_p_value = model.output_dict["function"]

        return {"key": output_key, "time": output_time, "p_value":
output_p_value}

    def evaluation(self):
        keys = []
        times = []
        p_values = []
        for i in range(2):
            ga_output = self.genetic_algorithm()
            keys.append(ga_output["key"])
            times.append(ga_output["time"])
            p_values.append(ga_output["p_value"])

        hamming_distance = hamming_dist(keys[0], keys[1])
        avg_time = sum([float(x) for x in times]) / 2
        avg_p_value = sum(p_values) / 2

        eval_value = hamming_distance + (1/avg_time)

        return {"eval_value": eval_value, "hamming_distance":
hamming_distance,
                "avg_time": avg_time, "avg_p_value": avg_p_value}

if __name__ == "__main__":

    best_eval_value = 0.0
```

```
    for w1 in arange(0.0, 4/3, 1/3):
        for w2 in arange(0.0, 4/3, 1/3):
            for w3 in arange(0.0, 4/3, 1/3):
                sa = Sensitivity_Analysis([block_freq, freq_monobit, runs],
                                          w1, w2, w3)
                eval_output = sa.evaluation()
                if eval_output["eval_value"] > best_eval_value:
                    best_weights = {"w1": w1, "w2": w2, "w3": w3}
                    best_hamming_distance = eval_output["hamming_distance"]
                    best_avg_time = eval_output["avg_time"]
                    best_avg_p_value = eval_output["avg_p_value"]
                    best_eval_value = eval_output["eval_value"]

    print(f"\n\nBest Values:\nWeights: {best_weights}\nHamming Distance:
{best_hamming_distance}\n"
          f"Time: {best_avg_time}\nP_value: {best_avg_p_value}")
```

D.5. Top Four Hamming Distance Performers for 256-bit Key Generation Sensitive Analysis

```
from nistsp80022tests import runs, cumulative_sums, approximate_entrophy,
universal
from geneticalgorithm import geneticalgorithm as ga
from hammingtest import hamming_dist
from numpy import arange
import timer

class Sensitivity_Analysis():
    def __init__(self, fitness_functions, w1, w2, w3, w4):
        self.fitness_functions = fitness_functions
        self.w1 = w1
        self.w2 = w2
        self.w3 = w3
        self.w4 = w4

    def f(self, ga_array):
        p_value_list = []
        for fitness_function in self.fitness_functions:
            p_value_list.append(fitness_function(ga_array))

        total_p_value = (self.w1 * p_value_list[0]) + (self.w2 *
(p_value_list[1][0] + p_value_list[1][1])) + \
                        (self.w3 * p_value_list[2]) + (self.w4 *
p_value_list[3])

        return -total_p_value

    def genetic_algorithm(self):
        algorithm_param = {
            'max_num_iteration': 500,
            'population_size': 17,
            'mutation_probability': 0.05,
            'elit_ratio': 1 / 17,
            'crossover_probability': 0.25,
            'parents_portion': 3 / 17,
            'crossover_type': 'two_point',
            'max_iteration_without_improv': None}

        model = ga(
            function=self.f,
            dimension=256,
            variable_type='bool',
```

```
            algorithm_parameters=algorithm_param,
            convergence_curve=False
        )

        t = timer.Timer()

        t.start()
        model.run()
        output_time = t.stop()

        output_key = "".join(str(x) for x in
model.output_dict["variable"].astype("int"))

        output_p_value = model.output_dict["function"]

        return {"key": output_key, "time": output_time, "p_value":
output_p_value}

    def evaluation(self):
        keys = []
        times = []
        p_values = []
        for i in range(2):
            ga_output = self.genetic_algorithm()
            keys.append(ga_output["key"])
            times.append(ga_output["time"])
            p_values.append(ga_output["p_value"])

        hamming_distance = hamming_dist(keys[0], keys[1])
        avg_time = sum([float(x) for x in times]) / 2
        avg_p_value = sum(p_values) / 2

        eval_value = hamming_distance + (1/avg_time)

        return {"eval_value": eval_value, "hamming_distance":
hamming_distance,
                "avg_time": avg_time, "avg_p_value": avg_p_value}

if __name__ == "__main__":

    best_eval_value = 0.0
    for w1 in arange(0.0, 4/3, 1/3):
        for w2 in arange(0.0, 4/3, 1/3):
            for w3 in arange(0.0, 4/3, 1/3):
                for w4 in arange(0.0, 4/3, 1/3):
                    sa = Sensitivity_Analysis([runs, cumulative_sums,
approximate_entrophy, universal],
                                              w1, w2, w3, w4)
                    eval_output = sa.evaluation()
                    if eval_output["eval_value"] > best_eval_value:
                        best_weights = {"w1": w1, "w2": w2, "w3": w3, "w4":
w4}
                        best_hamming_distance =
eval_output["hamming_distance"]
                        best_avg_time = eval_output["avg_time"]
                        best_avg_p_value = eval_output["avg_p_value"]
                        best_eval_value = eval_output["eval_value"]

    print(f"\n\nBest Values:\nWeights: {best_weights}\nHamming Distance:
{best_hamming_distance}\n"
          f"Time: {best_avg_time}\nP_value: {best_avg_p_value}"
```

D.6. Top Four Time Performers for 256-bit Key Generation Sensitive Analysis

```
from nistsp80022tests import block_freq, freq_monobit, runs, longestrunofones
from geneticalgorithm import geneticalgorithm as ga
from hammingtest import hamming_dist
from numpy import arange
import timer

class Sensitivity_Analysis():
    def __init__(self, fitness_functions, w1, w2, w3, w4):
        self.fitness_functions = fitness_functions
        self.w1 = w1
        self.w2 = w2
        self.w3 = w3
        self.w4 = w4

    def f(self, ga_array):
        p_value_list = []
        for fitness_function in self.fitness_functions:
            p_value_list.append(fitness_function(ga_array))

        total_p_value = (self.w1 * p_value_list[0]) + (self.w2 *
p_value_list[1]) + (self.w3 * p_value_list[2]) +\
                        (self.w4 * p_value_list[3])
        return -total_p_value

    def genetic_algorithm(self):
        algorithm_param = {
            'max_num_iteration': 500,
            'population_size': 17,
            'mutation_probability': 0.05,
            'elit_ratio': 1 / 17,
            'crossover_probability': 0.25,
            'parents_portion': 3 / 17,
            'crossover_type': 'two_point',
            'max_iteration_without_improv': None}

        model = ga(
            function=self.f,
            dimension=256,
            variable_type='bool',
            algorithm_parameters=algorithm_param,
            convergence_curve=False
        )

        t = timer.Timer()

        t.start()
        model.run()
        output_time = t.stop()

        output_key = "".join(str(x) for x in
model.output_dict["variable"].astype("int"))

        output_p_value = model.output_dict["function"]

        return {"key": output_key, "time": output_time, "p_value":
output_p_value}

    def evaluation(self):
        keys = []
        times = []
        p_values = []
        for i in range(2):
            ga_output = self.genetic_algorithm()
            keys.append(ga_output["key"])
```

```
            times.append(ga_output["time"])
            p_values.append(ga_output["p_value"])

        hamming_distance = hamming_dist(keys[0], keys[1])
        avg_time = sum([float(x) for x in times]) / 2
        avg_p_value = sum(p_values) / 2

        eval_value = hamming_distance + (1/avg_time)

        return {"eval_value": eval_value, "hamming_distance":
hamming_distance,
                "avg_time": avg_time, "avg_p_value": avg_p_value}

if __name__ == "__main__":

    best_eval_value = 0.0
    for w1 in arange(0.0, 4/3, 1/3):
        for w2 in arange(0.0, 4/3, 1/3):
            for w3 in arange(0.0, 4/3, 1/3):
                for w4 in arange(0.0, 4/3, 1/3):
                    sa = Sensitivity_Analysis([block_freq, freq_monobit,
runs, longestrunofones],
                                              w1, w2, w3, w4)
                    eval_output = sa.evaluation()
                    if eval_output["eval_value"] > best_eval_value:
                        best_weights = {"w1": w1, "w2": w2, "w3": w3, "w4":
w4}
                        best_hamming_distance =
eval_output["hamming_distance"]
                        best_avg_time = eval_output["avg_time"]
                        best_avg_p_value = eval_output["avg_p_value"]
                        best_eval_value = eval_output["eval_value"]

    print(f"\n\nBest Values:\nWeights: {best_weights}\nHamming Distance:
{best_hamming_distance}\n"
          f"Time: {best_avg_time}\nP_value: {best_avg_p_value}")
```

D.7. Top Five Hamming Distance Performers for 256-bit Key Generation Sensitive Analysis

```
from nistsp80022tests import runs, cumulative_sums, approximate_entrophy,
universal, serial
from geneticalgorithm import geneticalgorithm as ga
from hammingtest import hamming_dist
from numpy import arange
import timer

class Sensitivity_Analysis():
    def __init__(self, fitness_functions, w1, w2, w3, w4, w5):
        self.fitness_functions = fitness_functions
        self.w1 = w1
        self.w2 = w2
        self.w3 = w3
        self.w4 = w4
        self.w5 = w5

    def f(self, ga_array):
        p_value_list = []
        for fitness_function in self.fitness_functions:
            p_value_list.append(fitness_function(ga_array))
```

```
        total_p_value = (self.w1 * p_value_list[0]) + (self.w2 *
(p_value_list[1][0] + p_value_list[1][1])) + \
                        (self.w3 * p_value_list[2]) + (self.w4 *
p_value_list[3]) + \
                        (self.w5 * (p_value_list[4][0] + p_value_list[4][1]))
        return -total_p_value

    def genetic_algorithm(self):
        algorithm_param = {
            'max_num_iteration': 500,
            'population_size': 17,
            'mutation_probability': 0.05,
            'elit_ratio': 1 / 17,
            'crossover_probability': 0.25,
            'parents_portion': 3 / 17,
            'crossover_type': 'two_point',
            'max_iteration_without_improv': None}

        model = ga(
            function=self.f,
            dimension=256,
            variable_type='bool',
            algorithm_parameters=algorithm_param,
            convergence_curve=False
        )

        t = timer.Timer()

        t.start()
        model.run()
        output_time = t.stop()

        output_key = "".join(str(x) for x in
model.output_dict["variable"].astype("int"))

        output_p_value = model.output_dict["function"]

        return {"key": output_key, "time": output_time, "p_value":
output_p_value}

    def evaluation(self):
        keys = []
        times = []
        p_values = []
        for i in range(2):
            ga_output = self.genetic_algorithm()
            keys.append(ga_output["key"])
            times.append(ga_output["time"])
            p_values.append(ga_output["p_value"])

        hamming_distance = hamming_dist(keys[0], keys[1])
        avg_time = sum([float(x) for x in times]) / 2
        avg_p_value = sum(p_values) / 2

        eval_value = hamming_distance + (1/avg_time)

        return {"eval_value": eval_value, "hamming_distance":
hamming_distance,
                "avg_time": avg_time, "avg_p_value": avg_p_value}

if __name__ == "__main__":

    best_eval_value = 0.0
    for w1 in arange(0.0, 4/3, 1/3):
```

```
        for w2 in arange(0.0, 4/3, 1/3):
            for w3 in arange(0.0, 4/3, 1/3):
                for w4 in arange(0.0, 4/3, 1/3):
                    for w5 in arange(0.0, 4/3, 1/3):
                        sa = Sensitivity_Analysis([runs, cumulative_sums,
approximate_entrophy, universal, serial],
                                                  w1, w2, w3, w4, w5)
                        eval_output = sa.evaluation()
                        if eval_output["eval_value"] > best_eval_value:
                            best_weights = {"w1": w1, "w2": w2, "w3": w3,
"w4": w4, "w5": w5}
                            best_hamming_distance =
eval_output["hamming_distance"]
                            best_avg_time = eval_output["avg_time"]
                            best_avg_p_value = eval_output["avg_p_value"]
                            best_eval_value = eval_output["eval_value"]

    print(f"\n\nBest Values:\nWeights: {best_weights}\nHamming Distance:
{best_hamming_distance}\n"
          f"Time: {best_avg_time}\nP_value: {best_avg_p_value}")
```

D.8. Top Five Time Performers for 256-bit Key Generation Sensitive Analysis

```
from nistsp80022tests import block_freq, freq_monobit, runs,
longestrunofones, cumulative_sums
from geneticalgorithm import geneticalgorithm as ga
from hammingtest import hamming_dist
from numpy import arange
import timer

class Sensitivity_Analysis():
    def __init__(self, fitness_functions, w1, w2, w3, w4, w5):
        self.fitness_functions = fitness_functions
        self.w1 = w1
        self.w2 = w2
        self.w3 = w3
        self.w4 = w4
        self.w5 = w5

    def f(self, ga_array):
        p_value_list = []
        for fitness_function in self.fitness_functions:
            p_value_list.append(fitness_function(ga_array))

        total_p_value = (self.w1 * p_value_list[0]) + (self.w2 *
p_value_list[1]) + (self.w3 * p_value_list[2]) +\
                        (self.w4 * p_value_list[3]) + (self.w5 *
(p_value_list[4][0] + p_value_list[4][1]))
        return -total_p_value

    def genetic_algorithm(self):
        algorithm_param = {
            'max_num_iteration': 500,
            'population_size': 17,
            'mutation_probability': 0.05,
            'elit_ratio': 1 / 17,
            'crossover_probability': 0.25,
            'parents_portion': 3 / 17,
            'crossover_type': 'two_point',
            'max_iteration_without_improv': None}

        model = ga(
            function=self.f,
            dimension=256,
```

```
            variable_type='bool',
            algorithm_parameters=algorithm_param,
            convergence_curve=False
        )

        t = timer.Timer()

        t.start()
        model.run()
        output_time = t.stop()

        output_key = "".join(str(x) for x in
model.output_dict["variable"].astype("int"))

        output_p_value = model.output_dict["function"]

        return {"key": output_key, "time": output_time, "p_value":
output_p_value}

    def evaluation(self):
        keys = []
        times = []
        p_values = []
        for i in range(2):
            ga_output = self.genetic_algorithm()
            keys.append(ga_output["key"])
            times.append(ga_output["time"])
            p_values.append(ga_output["p_value"])

        hamming_distance = hamming_dist(keys[0], keys[1])
        avg_time = sum([float(x) for x in times]) / 2
        avg_p_value = sum(p_values) / 2

        eval_value = hamming_distance + (1/avg_time)

        return {"eval_value": eval_value, "hamming_distance":
hamming_distance,
                "avg_time": avg_time, "avg_p_value": avg_p_value}

if __name__ == "__main__":

    best_eval_value = 0.0
    for w1 in arange(0.0, 4/3, 1/3):
        for w2 in arange(0.0, 4/3, 1/3):
            for w3 in arange(0.0, 4/3, 1/3):
                for w4 in arange(0.0, 4/3, 1/3):
                    for w5 in arange(0.0, 4/3, 1/3):
                        sa = Sensitivity_Analysis([block_freq, freq_monobit,
runs, longestrunofones, cumulative_sums],
                                                  w1, w2, w3, w4, w5)
                        eval_output = sa.evaluation()
                        if eval_output["eval_value"] > best_eval_value:
                            best_weights = {"w1": w1, "w2": w2, "w3": w3,
"w4": w4, "w5": w5}
                            best_hamming_distance =
eval_output["hamming_distance"]
                            best_avg_time = eval_output["avg_time"]
                            best_avg_p_value = eval_output["avg_p_value"]
                            best_eval_value = eval_output["eval_value"]

    print(f"\n\nBest Values:\nWeights: {best_weights}\nHamming Distance:
{best_hamming_distance}\n"
          f"Time: {best_avg_time}\nP_value: {best_avg_p_value}")
```