Generating Prime Numbers Using Genetic Algorithms

Karlo Knežević
Algebra University College
Ilica 242, 10000 Zagreb, Croatia
Email: karlo.knezevic@algebra.hr

***Abstract*—Genetic algorithms are well-known and frequently used heuristic methods for solving optimization problems. In modern cryptography, the generation of large primes has an important role in the implementation of public-key cryptosystems such as RSA. In general, the prime number generation starts from the random number. If the generated random number can pass a specified probabilistic primality test, the random number is tentatively considered as a prime number and applied to a public-key cryptosystem. This paper investigates the application of genetic algorithms for generating large prime numbers that have special significance in cryptography. An introduction to the theory of prime numbers and the methods used to check the primality of a large number are shown. Moreover, the implementation of a genetic algorithm for generating prime numbers with a convenient representation and genetic operators is presented. We compare the efficiency of the existing method for generating prime numbers and genetic algorithms techniques, and we show that GA can be a viable option to generate large prime numbers.**

I. Introduction

Any natural number n greater than 1 is prime if it has no divisor other then 1 and itself.The role of prime numbers is diverse and spans across several research domains such as communication, coding theory, and **cryptography**. Within cryptography area, prime numbers have a special importance that result in a need for techniques able to produce prime numbers of various sizes.

Cryptography can be defined as a science of secret writing to hide the meaning of a message [1]. To ensure that goal one uses cryptographic algorithms, commonly known as **ciphers**. When all parties that participate in a secure communication use the same key, we talk about symmetric-key cryptography [2]. Public key cryptography, or asymmetric cryptography, is a cryptographic system that uses pairs of keys: public keys, which may be known to others, and private keys, which may never be known by any except the owner. To encode a message (commonly known as plaintext) into a ciphertext, one uses **encryption** transformation and to revert the ciphertext back to plaintext one uses **decryption** transformation.

RSA is public-key cryptography which was proposed by Rivest, Shamir, and Adleman in 1977, and it has become the international standard of **public-key cryptography**. RSA is a special reversible modular exponentiation encryption system, and the theoretical basis of it is an important conclusion in number theory: it is easy to do the product of two **large prime numbers**, but it is difficult to decompose large prime factors combined number [3]. In addition to encryption, it is also used for digital signatures and authentication. Because the safety of the RSA algorithm is closely related to the choice of large prime numbers, finding large prime numbers have great significance to the concrete realization of the RSA algorithm and security. Therefore, a research focus has been for years to efficiently determine the primality and search large prime numbers. In general, the prime number generation starts from the random number. If the generated random number can pass a specified probabilistic primality test, the random number is tentatively considered as a prime number and applied to a public-key cryptosystem.

Some of the fastest modern primality tests for determining whether an arbitrary given number n is prime are probabilistic (or Monte Carlo) algorithms, meaning that they have a small random chance of producing an incorrect answer. A composite number that passes such a test is called a pseudoprime. Some of the most used primality tests are Solovay-Strassen test [4] and the Miller-Rabbin test [5]. These algorithms iteratively check if the number is composite or not. If the number passes the primality test and is not declared as composite, one can say that it is probably prime, and the more such iterative steps are done, the greater the chances that it is true are.

Keys in public-key cryptography, due to their unique nature, are more computationally costly than their counterparts in secret-key cryptography. Asymmetric keys must be many times longer than keys in secret-cryptography in order to boast equivalent security. Keys in asymmetric cryptography are also more vulnerable to brute force attacks than in secret-key cryptography. There exist algorithms for public-key cryptography that allow attackers to crack private keys faster than a brute force method would require [6]. One of the main strength of public-key cryptography remains in the difficulty of factorization to the large primes.

A. Contributions

In this paper, we are focused on generating large prime numbers with practical application in public-key cryptography. To evolve prime numbers, we employ evolutionary algorithms (EAs), and more precisely Genetic Algorithm (GA). As a primality test, we use Baillie–PSW primality test, which is a combination of a strong Fermat probable prime test to base 2 and a strong Lucas probable prime test. Using that technique we show practical applications of using Genetic Algorithm in generating prime numbers.

B. Outline of the Paper

This paper is organized as follows. In Section II, we first discuss the properties of prime numbers. Furthermore, we elaborate Baillie–PSW primality test. Next, in Section III, we show the usage of primes in RSA. In Section IV, we give a short list of related work. Then, section V gives details about the GA approach we use and present the results of our experiments. Moreover, we discuss the consequences of our approach and we offer a possible roadmap for future work. Finally, in Section VI, we give a short conclusion.

II. Preliminaries

A prime number (or a prime) is any number greater than 1 and divisible only by 1 and by itself. Numbers that have multiple divisors are said to be composite. By convention, number 1 is neither composite nor prime.

*Lemma 2.1 (**Euclid's lemma**):* If p is a prime number and divides the product of $a \cdot b$, then p divides a or p divides b (or both).

Widening this lemma is a **fundamental theorem of arithmetic**, showed in equation 1, which says that any natural number can be written in a unique way as a product of prime numbers (if we ignore the order of these numbers):

$$n = p_1^{\alpha_1} p_2^{\alpha_2} \cdots p_r^{\alpha_r}, \tag{1}$$

where $p_1 < p_2 < \cdots < p_r$ are different prime factors of n, and $\alpha_1, \alpha_2, ..., \alpha_r \in \mathbb{Z}$.

If the number n is composite, it is divisible by a prime number less than $\sqrt{n}$. This fact makes it easy to check the simplicity of smaller numbers since one has to check the divisibility with numbers only to the square root of the checking number.

A. Distribution of prime numbers

The fact that there are infinitely many prime numbers was also known to the already mentioned Euclid. But, although there are infinitely many of them, their arrangement is extremely irregular. The function $\pi(x)$ tells how many primes are in the interval $[1, x]$.

*Theorem 2.2 (**Prime number theorem**):* Let $\pi(x)$ be the prime-counting function that gives the number of primes less than or equal to x, for any real number x. The limit of the quotient of the two functions $\pi(x)$ and $\frac{x}{ln(x)}$ as x increases without bound is 1:

$$\lim_{x \to \infty} \frac{\pi(x)}{x/ln(x)} = 1. \tag{2}$$

Table I shows primes density in certain intervals as well as values of equation 2.

We can observe that the ratio tends towards number 1 with increasing value. Gauss also came to the conclusion that the $\pi(x)$ function could be represented by a logarithmic integral showed in equation 3.

$$\pi(x) \approx Li(x) = \int_2^x \frac{dt}{ln\, t}. \tag{3}$$

TABLE I: Primes density and values of π function.

x	$\pi(x)$	$\frac{\pi(x)}{x/ln(x)}$
10	4	0.92103
10^2	25	1.15129
10^6	78498	1.08449
10^{10}	455052511	1.04780
10^{20}	2220602560918840	1.02273

B. Definitions and Theorems on Prime Numbers

The question arises as to whether it is possible for a number by some check to conclude very quickly whether it is prime or composite. To answer this question, mathematicians devised primality tests. We distinguish two types of tests: **deterministic** and **probabilistic**. Deterministic, as their name suggests, give the correct answer to the question of whether a number is prime or composite. The problem with such tests is a big computational complexity, which increases computation time as the tested number being increased. For this reason, probabilistic tests are more often used, which perform the check much faster, but their answer is not exact - if they conclude that the number is composite, it is certainly composite, but if they conclude that the number is prime, there is a probability (although often small) that it is composite. We give a short overview of most relevant definitions and theorems related to prime numbers [7].

*Definition 2.1 (**Greatest common divisor**):* The greatest common divisor of the numbers a and b is denoted by (a, b), and represents the largest number that divides both a and b.

*Definition 2.2 (**Congruence**):* If the number m divides the difference $a - b$, then we say that a is congruent to b modulo m and write it as:

$$a \equiv b \pmod{m} \tag{4}$$

*Definition 2.3 (**Coprime numbers**):* The numbers a and b are coprime if their greatest common divisor is 1.

*Definition 2.4 (**Euler's totient function**):* Let $n \in \mathbb{N}$. Euler's totient function $\varphi(n)$ is equal to the number of natural numbers less than n, and which are coprime with n.

For example, $\varphi(20) = 8$ because the numbers $1, 3, 7, 9, 11, 13, 17$ and 19 are coprime with the number 20. If n is a prime number, then the $\varphi(n) = n - 1$ holds true since the prime number is divisible only by the number 1 and by itself.

*Definition 2.5 (**Quadratic residue**):* The number a is the quadratic residue of modulo p if exists $x \in \mathbb{Z}_p^*$ such that is valid relation:

$$x^2 \equiv a \pmod{p} \tag{5}$$

*Theorem 2.3 (**Fermat's little theorem**):* If p is a prime number and a is a coprime with p then:

$$a^{p-1} \equiv 1 \pmod{p} \tag{6}$$

*Definition 2.6 (**Fermat's pseudoprime numbers**):* Let n be an odd number that is not prime and a is coprime with n. Then, n is Fermat's pseudo-prime number in a base a if next equation is valid:

$$a^{n-1} \equiv 1 \pmod{p} \tag{7}$$

TABLE II: Primality tests, test type and running time complexity. n is number to be tested and k is number of tests.

Primality test	Year	Type	Running time
AKS [8]	2002	Deterministic	$\mathcal{O}((\log n)^{6+\epsilon})$
Elliptic curve [9]	1986	Las Vegas	$\mathcal{O}((\log n)^{4+\epsilon})$
Baillie-PSW [10], [11]	1980	Monte Carlo	$\mathcal{O}((\log n)^{2+\epsilon})$
Miller-Rabin [12]	1980	Monte Carlo	$\mathcal{O}(k(\log n)^{2+\epsilon})$
Soovay-Strassen [12]	1977	Monte Carlo	$\mathcal{O}(k(\log n)^{2+\epsilon})$

Fermat's pseudoprime numbers are called only pseudoprimes in the rest of the paper.

*Definition 2.7 (**Legendre symbol**):* Let p be a prime number. For each integer a we define the Legendre symbol:

$$\left(\frac{a}{p}\right) = \begin{cases} 0, & \text{if } a \equiv 0 \pmod{p}, \\ 1, & \text{if } a \text{ quadratic residue modulo } p, \\ -1, & \text{if } a \text{ not quadratic residue modulo } p. \end{cases} \tag{8}$$

What should be noted is that the Legendre symbol is not defined if the number n is not prime. Jacobi symbol, which can be determined for any odd natural number, solves this problem.

*Definition 2.8 (**Jacobi symbol**):* Let n be an odd natural number that has factorization as follows:

$$n = p_1^{\alpha_1} p_2^{\alpha_2} \cdots p_k^{\alpha_k}. \tag{9}$$

For every integer a, Jacobi symbol $\left(\frac{a}{n}\right)$ is defined as:

$$\left(\frac{a}{n}\right) = \left(\frac{a}{p_1}\right)^{\alpha_1} \left(\frac{a}{p_2}\right)^{\alpha_2} \cdots \left(\frac{a}{p_k}\right)^{\alpha_k}. \tag{10}$$

Jacobi symbol is obtained by multiplying Legendre symbols by all prime factors of the number n. If the number n is prime, then Jacobi symbol becomes equal to Legendre symbol.

C. Baillie-PSW Primality Test

Some of the primality tests are the Fermat test, the Solovay-Strassen test [4], and the Miller-Rabin test [5]. Algorithms work by running in steps to check if a number is composite. If a number passes an algorithm step and is not declared composite, we can say that it is probably prime, and the more such steps pass, the greater the chances are that it is indeed true. In the Fermat and Solovay-Strassen test, the probability that a composite number passes the whole test (without declaring it composite) is $\frac{1}{2^m}$ (where m is the number of steps of the algorithm), while this number in the Miller-Rabin test $\frac{1}{4^m}$. In table II we give a list of primality tests.

In this paper, we use the **Baillie-PSW** (Pomerance, Selfridge and Wagstaff) probability test to check the primality. The test is a combination of the Miller-Rabin primality test and the Lucas probability test [11]. For both the Miller-Rabin and Lucas tests, there are lists of pseudo-prime numbers - composite numbers that pass the test, but it is not currently known whether there are numbers in both lists - there is even evidence that these numbers belong to different groups, which means if a number is pseudo-prime for both tests, it is very likely prime. No compound number less than 2^{64} (approx. $1.845 \cdot 10^{19}$) passes the test, which means that in that case the test is deterministic [10].

The steps of the Baillie-PSW test to check the primality of the number n are:

1) Using trial division, check that the number n is divisible by small prime numbers up to some suitable limit.
2) Check that n is a very pseudo-prime number in a base 2. If not, then it is composite.
3) Find the first D in the sequence $5, -7, 9, -11, 13, -15, ...$ for which Jacobi symbol $\left(\frac{D}{n}\right)$ is equal to -1. Set $P = 1$ and $Q = (1 - D)/4$.
4) Perform a Lucas primality test using the parameters D, P and Q. If n is not a Lucas pseudo-prime number, then it is composite. Otherwise, it is almost certainly prime.

A few notes related to the test [10], [13]:

- The first step is optional and is performed to speed up the algorithm.
- The second step is actually the Miller-Rabin test using the base 2. There is no particular reason why this base is used, but after a series of tests it was concluded that the best results are given by this base in combination with Lucas test.
- Baillie and Wagstaff proved that the average number of parameters D, for which the test must be performed, is about 3.147755149.
- If a number n is the square root of some number, the third step will never give such a D that $\left(\frac{D}{n}\right) = -1$. If the test does not calculate the parameter D quickly, it is necessary to suspect that it is a square root and by Newton's method it is possible to check whether it is really a square root or not.

III. Prime Numbers in Public-Key Cryptography

Prime numbers are used in various encryption algorithms due to factorization problem - there is currently no efficient algorithm that can factorize large numbers. Here, we briefly present the RSA algorithm.

Nowadays, RSA is a mostly used algorithm to transmit shared keys for symmetric key cryptography, which are then used for bulk encryption-decryption. This way of encrypting implies the existence of two keys - the public one, which is used for encryption, and the secret one, which is used for decryption. It is assumed that it is not possible to obtain the secret key in real-time by using the public key.

Let's assume there are two parties - A and B and that A wants to send a message to B. The procedure takes place in the following steps:

1) B **generates a public key** and sends it via unsecured channel to A. Such key should not be changed change and needs to be stored in order to be able to communicate with B.
2) A generates a message and encrypts it using B's public key and eventually sends it to B. The message is encrypted with a public key and is sent through a unsecured channel.
3) B uses his private key to decrypt the message and read its contents.

A. Generate Public Key

To generate a public key, one should fallow next steps:

1) **Generate two large prime numbers** p and q that are approximately equal size
2) Calculate $n = p \cdot q$ and Euler's totient function $\varphi(n) = (p-1)(q-1)$. The number n is called the RSA module.
3) The encryption exponent e is selected, which must be greater than 1 and less than $\varphi(n)$, and coprime with $\varphi(n)$.
4) The decryption exponent d is calculated, which must also be greater than 1 and less than $\varphi(n)$, and satisfy equation 11.

$$e \cdot d \equiv 1 \pmod{\varphi(n)} \tag{11}$$

5) Finally, the public key-pair (n, e) is published, and the parameters d, p, q and $\varphi(n)$ are kept secret.

B. Encryption and Decryption

We briefly show a process of encryption and decryption:

1) Use the public key (n, e) and encrypt the message m : $c \equiv m^e \pmod{n}$
2) The sender sends the encrypted message c via an unsecured channel.
3) The receiver calculates the original message m using the secret key (n, d): $m \equiv c^d \pmod{n}$

The security of this algorithm lies in the fact that a composite number n, which is the product of two large prime numbers, is impossible to factorize in a real-time. But as this number is public, its factorization would make it possible to determine all the remaining parameters needed to decrypt and read the original message would be enabled. For this reason, the prime numbers used in the RSA algorithm are of the order of 10^{100}.

IV. Related Work

As mentioned in Section I, there are many successful applications of heuristic techniques usable in cryptography [14]. In this section, we discuss works that consider the evolution of primes, and different approaches to achieve them. Generally, there are two approaches when generating primes: one evolves a **generator function** and one evolves **generates prime numbers**. We firstly show papers considering generator functions.

Mabrouk, Hernandez-Castro, and Fukushima in [15] use memetic programming to generate a formula that produces a set of distinct primes. Some of the new formulas are polynomials that can produce up to 40 distinct primes for a set of consecutive integers. Other rational functions are also generated and they can produce up to 59 distinct primes for a set of consecutive integers.

Walker and Miller in [16] predict prime numbers using Cartesian genetic programming (CGP). In this paper, the authors presented an approach for evolving non-consecutive prime generating polynomials and also two different approaches using CGP for evolving a function $f(i)$, which produces consecutive prime numbers $p(i)$, for consecutive input values i. The digital circuit approach using multi-chromosome CGP evolved multiple functions for consecutive sequences of prime numbers with increasing length, the longest of which produced 208 consecutive prime numbers, for input value i, where $1 \leqslant i \leqslant 208$.

Next, we give an overview of papers trying to generate as large prime as possible using a genetic algorithm. Qing and Zhihua in [17] use a genetic algorithm to generate large primes. Authors analyze safety factors restricting the algorithm to produce large primes and propose a method for determining large primes. They also experiment with several different crossover and mutation operators. In the paper, they show the largest number containing 512 bits.

Hu and Zang in [18] use a genetic algorithm and propose a new strong prime number generator algorithm. The authors claim the method described in the paper is simple and easy to implement to meet the needs of the RSA algorithm security.

V. Experiments and Results

In this section we discuss on experimental settings and obtained results. Moreover, we describe in detail the algorithmic modification we make to make our results practical.

A. Algorithms and Representations

The problem of generating prime numbers is not a classic optimization problem and therefore the approach to implementation of a genetic algorithm has to be somewhat different than usual. The implementation of the algorithm in this paper, during the evolution, generates as many as possible different prime numbers and store their values that can be accessed after execution. Instead of using a generational or steady-state version where the population size is constant, we increase the population by each new prime individual (see Algorithm 1). The reason for that modification is that each prime individual is equally good and represents a viable genetic material that could further create new prime individuals.

Algorithm 1 Increasing population genetic algorithm.

```
population ← initialize with N primes
while termination criteria not satisfied do
  parents ← randomly select 2 individuals
  child ← crossover (parents)
  perform mutation on child, with probability p_m
  if child is prime and child ∉ population then
    population ← population ∪ child
  end if
end while
```

The evolutionary process begins by generating an initial population consisting of the first N prime numbers. It is worth noticing that N is a small number and can be generated by using a simple prime sieve or primes can be cached. In our experiments, we simply use first N primes (2, 3, 5, ...). We experiment with 2 termination criteria: number of evaluations and time interval. As long as termination criteria have not been satisfied, 2 individuals have randomly been selected. After the child has been created by a crossover operator, the mutation is applied with probability p_m. Finally, using the Baillie-PWS

primality test we check if the child is prime and does it exist in the population.

In this work, we experiment with two different encodings and compare their efficiency: **bitstring** and **integer** representation. In bitstring encoding, we use concatenating crossover operator and simple mutation, where a single bit is inverted. Concatenating crossover operator randomly chooses in which order will merge two individuals. For example, crossover of 101 and 1101 can produce individuals 1011101 and 1101101.

In integer encoding, we use crossover operator defined in [19]. The main goal of the crossover operator (see Algorithm 2) is to generate a candidate with high probability to be prime. For mutation, we use the same operator as for crossover but with constant value $p_2 = 2$. Notice that p_2 could be any other prime number.

Algorithm 2 Integer crossover operator.

Inputs:
p_1, p_2 – primes from the population
Output:
c – prime number candidate
if $p_2 > p_1$ **then**
  $high$, $low \leftarrow p_2$, p_1
else
  $high$, $low \leftarrow p_1$, p_2
end if
$c_{(low)} \leftarrow$ convert base $high_{(10)}$
reverse the digits of $c_{(low)}$
$c_{(10)} \leftarrow$ convert base $c_{(low)}$
return c

To compare results obtained by genetic algorithm, we use naïve prime finding algorithm, which is widely used in practice. We use 128 iterations in Miller-Rabin primality test.

Algorithm 3 Naïve prime finding algorithm.

Inputs:
n – prime bit length
Output:
p – prime number candidate
$p \leftarrow$ random even number $\in \mathbb{N}$
while p not passed Miller-Rabin test **do**
  $p \leftarrow$ generate random bits of length n
  $p \leftarrow$ set MSB and LSB to 1
end while
return p

1) Algorithm Parameters: In all the experiments the number of independent trials T for each configuration is 30 and the stopping criteria for equals 200 000 evaluations and time limit. Following the creation, the new individual immediately undergoes mutation, which depends on the mutation rate parameter. After tuning, in our experiments this parameter equals 0.05, which results in five out of one hundred new individuals being mutated on average (see Algorithm 1). In Table III we give all parameter details. Moreover, to make our experiments more realistic, we set the execution time limit short enough to have a practical significance (5, 10 and 20 seconds).

TABLE III: Experiments parameters.

Parameter	Values
Encoding	{binary, integer}
Selection	random
p_m	{0.05, 0.1, 0.3}
Evaluations	200 000
Time limit	{5 s, 10 s, 20 s}
Initial population size	{3, 30, 100}

B. Fitness Function

We consider a binary fitness functon in this work:

$$fitness(n) = \begin{cases} 0, & n \nrightarrow \text{prime or } n \in \text{population}, \\ 1, & n \rightarrow \text{prime and } n \notin \text{population}. \end{cases} \quad (12)$$

An individual whose value is not prime must not be part of the population and therefore the fitness of such an individual is 0. Also, there must be no duplicates in the population, and individuals who are prime but whose values are already in the population are punished and do not enter the population.

Furthermore, an individual whose value is prime, but which is not in the population, has a fitness equal to 1, and this is true for all individuals who meet the given conditions. Any individual that meets criteria defined by fitness function is equally good, and since individuals that are prime and unique do not have to be compared to each other, they all have the same fitness value.

C. Results

Due to the fact, we experiment with large prime numbers, and our motivation in a practical application, all results are given in a bit length used to represent a prime. The recommended RSA modulus ($p \cdot q$) size for most settings is 2048 bits to 4096 bits. Thus, the primes, p and q, to be generated need to be 1024 bit to 2048 bit long [20]. For such reason, we bold all values higher than 1024. In Tables IV and V, we give results for bitstring and integer encodings, respectively.

TABLE IV: Results for the binary representation. N represents initial population size, time limit is time given to the algorithm execution. Results show prime bit length.

N	Time Limit [s]	Max n	Average n	Std dev
3	5	**1035**	683.35	113.66
	10	**1145**	831.32	130.31
	20	**1895**	1045.42	248.17
30	5	460	326.77	67.10
	10	755	446.90	92.70
	20	726	542.42	75.15
100	5	279	219.61	35.05
	10	433	318.52	53.27
	20	540	408.06	55.30

When considering bitstring encoding (see Table IV), we can construct prime numbers larger than 1024 bits only for the smallest size of the initial population we investigate ($N = 3$).

TABLE V: Results for the integer representation. N represents initial population size, time limit is time given to the algorithm execution. Results show prime bit length.

N	Time Limit [s]	Max n	Average n	Std dev
3	5	232	155.71	34.77
	10	312	220.48	34.85
	20	382	253.71	37.74
30	5	211	138.23	26.07
	10	235	177.06	20.46
	20	305	226.97	36.25
100	5	162	114.52	22.14
	10	185	144.00	18.13
	20	275	187.52	28.44

Naturally, the more time algorithm is given, the larger prime is generated. However, the larger the initial population size is, the maximal prime bits length gets smaller. We find interesting the fact the smaller the initial population is, the lower number of large primes is generated. In Table IV, one can notice how standard deviation drops by increasing prime generation time.

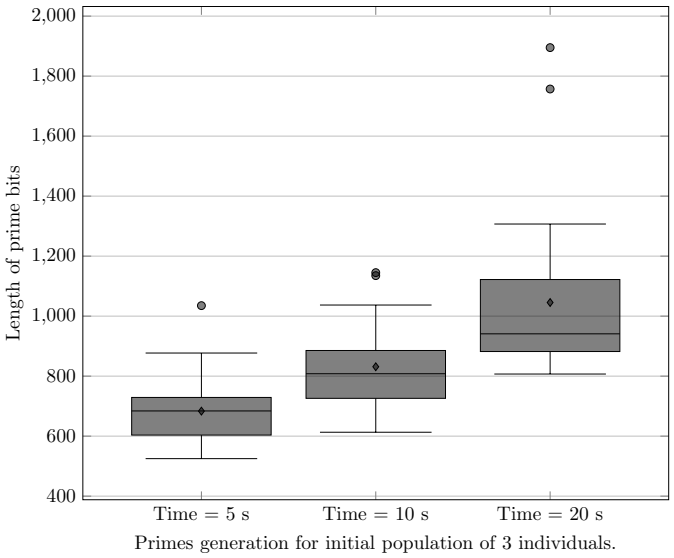

Fig. 1: Boxplot results for 3 different mutation operators in optimization of balanced Boolean function with 8 inputs.

In Figure 1, we present results for the length of the prime number bits and 3 individuals in the initial population, respectively. We show differences among 3 different time limits for primes generation. Note that the values given are average values over all experimental runs. One can observe the more time is given to GA, the larger prime would be generated.

Table V gives results for the integer encoding. As it can be seen, for all initial population sizes and time limits we consider, we are not able to find a large enough prime number to satisfy security criteria. Moreover, we observe a similar correlation between initial population size and time limit as with bitstring. Note that integer encoding does not vary in the best prime bit length. The reason for such behavior could be in the genetic operators' implementations. Comparing the sizes of generated primes in the populations, integer encoding

TABLE VI: Results for the naïve prime finding algorithm.

n	Limit [s]	Succ [%]	Min n [s]	Avg n [s]	Std dev
382	20	100.00	0.04	0.43	0.42
1035	5	48.39	0.09	2.60	1.54
1145	10	96.77	0.06	2.69	2.26
1895	20	58.06	0.93	9.03	5.29

generates 20% more primes than bitstring encoding. Therefore, we can conclude that genetic operators defined for integer encoding rather choose to generate closer primes than further ones.

Next, in Table VI we show a success rate in generating primes of certain bit length with naïve prime finding algorithm (see Algorithm 3). We took the best-obtained results with bitstring and integer encoding. Moreover, we set the time limit equal to the time GA had when the founding best certain solution. When the prime number of bits is very low, naïve prime search always find prime of a certain length. Increasing prime number of bits, naïve algorithm has more difficulties, and the success rate depends on the time limit given. Nevertheless, naïve prime search time is not comparable to the GA's because it is several times faster.

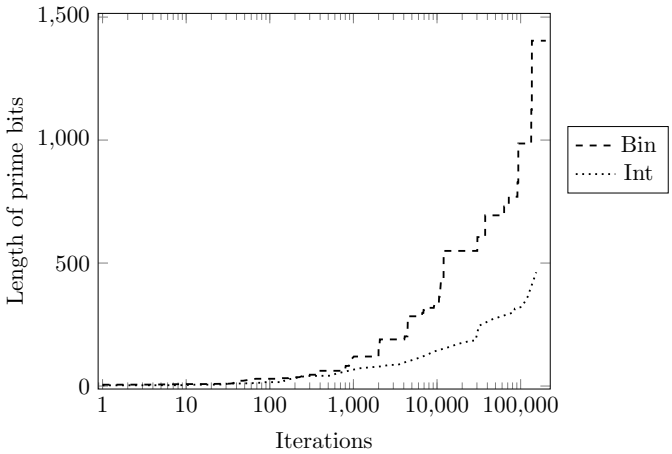

Fig. 2: Dynamics of finding a large primes through iterations, for bitstring and integer representation.

Finally, in Figure 2, we show the dynamics of finding a large primes through iterations. As previously discussed, bitstring GA outperforms integer GA in finding larger prime.

D. Future Work

Based on obtained results, there are several possible future directions to explore. First, the results show that a small initial size population finds large primes faster than a larger initial population. It would be interesting to investigate how the distribution of primes in the initial population affects the speed of finding large primes.

Next, we use a modified genetic algorithm that increases in each iteration population size. Instead of increasing, one possibility to explore would be to keep the population constant size and how that affects the speed of evolving primes and the size in bit length.

On the other hand, this work shows the integer encoding can evolve large prime numbers but not applicable in cryptography. One avenue of research would be a reconstruction of evolutionary operators (crossover or mutation). Moreover, it would be interesting to examine the performances of other encodings with GA.

Finally, in this paper, we experimented only with GA. There are a plethora of evolutionary algorithms with different properties and advantages comparing to GA. One way of future experiments would include different evolutionary algorithms or combining multiple algorithms with hybridization methods.

VI. Conclusion

In this paper, we addressed the construction of large prime numbers through an evolutionary algorithm. To resist factorization attacks, it is essential in public-key cryptography to construct large primes. One of the approaches used in practice is naïve prime finding algorithm.

Our results suggest that genetic algorithms can be used to evolve primes larger than 2^{1024} whereas the best performing encoding we consider bitstring. That success stems from the representation rather than using some specific selection strategy.

In this work, we experiment with two constraints: initial population size and execution time. The largest prime we construct using bitstring encoding has $1\,895$ bits, which is approximately 10^{571}, in the time limit of 20 seconds. To make finding primes easier, we modified the genetic algorithm introducing variable population size.

Finally, even though we did not outperform naïve prime finding algorithm in speed, we showed the possibility of evolving large enough prime numbers using bitstring encoding to have practical application in public-key cryptography.

References

[1] C. Paar and J. Pelzl, *Understanding Cryptography - A Textbook for Students and Practitioners*. Springer, 2010.

[2] L. R. Knudsen and M. Robshaw, *The Block Cipher Companion*, ser. Information Security and Cryptography. Springer, 2011. [Online]. Available: https://doi.org/10.1007/978-3-642-17342-4

[3] R. L. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Commun. ACM*, vol. 21, no. 2, p. 120–126, Feb. 1978. [Online]. Available: https://doi.org/10.1145/359340.359342

[4] R. M. Solovay and V. Strassen, "A fast Monte-Carlo test for primality," *SIAM Journal on Computing*, vol. 6, pp. 84–85, 1977.

[5] G. L. Miller, "Riemann's hypothesis and tests for primality," ser. STOC '75. New York, NY, USA: Association for Computing Machinery, 1975, p. 234–239. [Online]. Available: https://doi.org/10.1145/800116.803773

[6] T. Kleinjung, K. Aoki, J. Franke, A. K. Lenstra, E. Thomé, J. W. Bos, P. Gaudry, A. Kruppa, P. L. Montgomery, D. A. Osvik, H. te Riele, A. Timofeev, and P. Zimmermann, "Factorization of a 768-bit rsa modulus," in *Advances in Cryptology – CRYPTO 2010*, T. Rabin, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 333–350.

[7] A. J. Menezes, P. C. Van Oorschot, and S. A. Vanstone, *Handbook of applied cryptography*. CRC press, 2018.

[8] M. Agrawal, N. Kayal, and N. Saxena, "Primes is in p," *Annals of mathematics*, pp. 781–793, 2004.

[9] F. Morain, "Implementing the asymptotically fast version of the elliptic curve primality proving algorithm," *Math. Comput.*, vol. 76, pp. 493–505, 2007.

[10] C. Pomerance, J. L. Selfridge, and S. S. Wagstaff, "The pseudoprimes to $25 \cdot 10^9$," *Mathematics of Computation*, vol. 35, no. 151, pp. 1003–1026, 1980. [Online]. Available: http://www.jstor.org/stable/2006210

[11] R. Baillie and S. S. Wagstaff, "Lucas pseudoprimes," *Mathematics of Computation*, vol. 35, no. 152, pp. 1391–1417, 1980. [Online]. Available: http://www.jstor.org/stable/2006406

[12] L. Monier, "Evaluation and comparison of two efficient probabilistic primality testing algorithms," *Theoretical Computer Science*, vol. 12, no. 1, pp. 97–108, 1980. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0304397580900079

[13] R. Baillie, A. Fiori, and S. S. W. J. au2, "Strengthening the baillie-psw primality test," 2020.

[14] K. Knežević, "Combinatorial optimization in cryptography," in *2017 40th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, 2017, pp. 1324–1330.

[15] E. Mabrouk, J. Hernandez-Castro, and M. Fukushima, "Prime number generation using memetic programming," *Artificial Life and Robotics*, vol. 16, pp. 53–56, 06 2011.

[16] J. Walker and J. Miller, "Predicting prime numbers using cartesian genetic programming," vol. 4445, 04 2007, pp. 205–216.

[17] Z. Qing and H. Zhihua, "The large prime numbers generation of rsa algorithm based on genetic algorithm," in *2011 International Conference on Intelligence Science and Information Engineering*, 2011, pp. 434–437.

[18] Z. Hu and Q. Zhang, "The strong prime numbers generation algorithm based on genetic algorithm," *Applied Mechanics and Materials*, vol. 220-223, pp. 2963–2967, 11 2012.

[19] K. Mishra, P. Bhatia, S. Tiwari, and A. Misra, "A novel genetic algorithm for generating prime numbers," 2013.

[20] E. Barker and Q. Dang, "Recommendation for key management," 2015. [Online]. Available: https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-57Pt3r1.pdf