An Evolutionary Algorithm for Short Addition Chains

Hazem M. Bahig[1]*, Khaled A. Alutaibi[2], Mohammed A. Mahdi[3], Amer AlGhadhban[4], Hatem M. Bahig[5]
Computer Science and Information Department, College of Computer Science and Engineering, University of Ha'il, Ha'il, KSA[1,3]
Computer Engineering Department, College of Computer Science and Engineering, University of Ha'il, Ha'il, KSA[2]
Electrical Engineering Department, College of Engineering, University of Ha'il, Ha'il, KSA[4]
Computer Science Division, Mathematics Department, Faculty of Science, Ain Shams University, Cairo, Egypt[1, 5]

***Abstract*—The encryption efficiency of the Rivest-Shamir-Adleman cryptosystem is based on decreasing the number of multiplications in the modular exponentiation (ME) operation. An addition chain (AC) is one of the strategies used to reduce the time consumed by ME through generating a shortest/short chain. Due to the non-polynomial time required for generating a shortest AC, several algorithms have been proposed to find a short AC in a faster time. In this paper, we use the evolutionary algorithm (EA) to find a short AC for a natural number. We discuss and present the role of every component of the EA, including the population, mutation operator, and survivor selection. Then we study, practically, the effectiveness of the proposed method in terms of the length of chain it generates by comparing it with three kinds of algorithms: (1) exact, (2) non-exact deterministic, and (3) non-exact non-deterministic. The experiment is conducted on all natural numbers that have 10, 11, 12, 13, and 14 bits. The results demonstrate that the proposed algorithm has good performance compared to the other three types of algorithms.**

***Keywords*—Addition chain; short chain; evolutionary algorithm; modular exponentiation; RSA**

I. Introduction

The purpose of cryptography is to secure communications over an open channel. To achieve this, two procedures are used, encryption and decryption. In simple terms, encryption is a process of transforming a given data, called plaintext, to unmeaningful data, called ciphertext, which can be reinstated to original form through the process of decryption.

One of the modern and strong encryption techniques used to encrypt data is the Rivest-Shamir-Adleman (RSA) cryptosystem. Encryption in RSA is based on generating a secret message c from the original message m using the modular exponentiation (ME) formula.

$$c \equiv m^e \bmod N$$

where the pair (e,N) is the public key and N is a composite odd number and equal to the product of two prime numbers. Similarly, for decryption, one needs to compute.

$$m \equiv c^d \bmod N$$

where d is the private key.

The main challenge associated with the encryption and decryption techniques in RSA is that the time consumed, in particular for decryption, is quite large because its performance is based on the ME mathematical operation that requires high computation resource. The time consumed in ME is due to the computation of a sequence (thousands, 1024-4096) of multiplications and squares for large-size numbers. Therefore, several techniques have been proposed to speed up the computation of multiplication and ME using sequential and parallel computation, for examples [6, 8, 19, 20, 35].

One of the strategies that can be used to decrease the number of multiplications in ME involves the use of an addition chain (AC) [27, 30]. An AC [22] for a number e is a finite increasing sequence of non-repeated natural numbers such that the start element of the sequence is 1 and any next element in the sequence is the sum of any two previous elements that are not necessarily distinct. The final element in the sequence is e. There are different types of chain besides the ACs, such as addition-subtraction chains [21,28], addition-multiplication chains [2, 9], Lucas chains and q-chains [22, 25] and addition sequence [7,17].

For a natural number e, there are many ACs of different lengths. A chain with minimal length is called a shortest AC, otherwise it is called a short AC. Therefore, if we can construct a shortest AC for e as $e_0 = 1, e_1 = 2, e_2, \dots, e_\ell = e$, then m^e can be computed in a minimal number of multiplications as m, m^2, m^{e_2},…, , m^e.

For example, for the natural number $e = 37$, we can construct different chains for e with different lengths as (1) $C_1 = (1,2,3,6,9,15,17,20,29,35,37)$ of length 10, (2) $C_2 = (1,2,3,5,10,20,30,35,37)$ of length 8, and (3) $C_3 = (1,2,4,8,16,32,36,37)$ of length 7. Therefore, the minimum number of multiplications to compute m^{37} is 7 and can be computed as m, $m^2 = m \times m$, $m^4 = m^2 \times m^2$, $m^8 = m^4 \times m^4$, $m^{16} = m^8 \times m^8$, $m^{32} = m^{16} \times m^{16}$, $m^{36} = m^{32} \times m^4$, $m^{36} = m^{32} \times m^4$, $m^{37} = m^{36} \times m^1$.

A large number of algorithms have been designed to find a solution to the AC problem. The goal of these algorithms is either to (1) find a shortest AC or (2) find a short AC. In respect of the first goal, the branch and bound method is one of the efficient strategies that has been used to find a shortest AC (see for examples, [1,3,12,34]). Another improvement strategy that has been suggested to find a shortest AC is to use high-performance computing to speed up the computation required to find the exact solution. The authors in [5] and [10] used a graphics processing unit and multicore system, respectively, to find a solution for the AC more quickly as compared to using sequential algorithms. However, while these solutions produce a chain that is shortest and therefore the solution is optimal, the running time of these algorithms is in non-polynomial time.

*Corresponding Author

On the other hand, many non-exact algorithms [4,13-16,18,23,24,26-33,36] have been proposed to find a short AC. In these algorithms, the length of the generated AC is not necessarily minimal and the algorithms run in polynomial time. Many different strategies have been proposed to find a solution to produce a short AC. We can classify these algorithms into two classes based on their behavior into deterministic algorithms and non-deterministic algorithms. The non-deterministic algorithm may generate different lengths in different runs for the same input e, while the deterministic algorithm produces the same length of AC even with different runs.

Examples of deterministic algorithms [4,11,18,20,22,30] for a short AC include the binary method, window method, factor method, and continued fraction method. The difference between these methods lies in the strategy used to find the solution, the length of the generated short AC, and the running time. In general, the running time of these algorithms is very fast when compared to that of exact algorithms. Also, the window and continued fraction methods give a better output than other methods.

Examples of non-deterministic algorithms [13-16,23,26-29,31-33] for a short AC include the genetic algorithm (GA), evolutionary algorithm (EA), ant colony algorithm, swarm intelligence algorithms, and artificial immune algorithm. All these algorithms are based on many factors such as the size of the population, the maximum number of iterations, and the strategies of different operators such as crossover and/or mutation. In general, non-deterministic algorithms can produce a short AC with a length that is better than that generated by the deterministic algorithms, see [26, 32].

From a review of the previous works related to the generation of a short AC, we can make several of observations. First, experimental studies have been conducted on (1) certain numbers only or (2) a small set of numbers within a data range such as 30 random numbers in the range [1,2048]. Second, in some research works, the value of some parameters, such as the total number of generations, is large. This leads to an increase in the running time of artificial intelligence (AI) algorithms. Third, in some previous algorithms, the number of independent runs for each natural number is 30 or 50, which is a large number and leads to increase running time.

In this paper, we are interested in using the EA to find an approximate solution for the AC problem. The proposed algorithm is based on modifying one of the EAs [16] for a short AC in relation to three aspects (1) the process of generating the elements of chain, (2) mutation operator, and (3) survivor selection. The developed algorithm exhibits good performance in terms of the generated length of the AC as compared to previous methods. To prove the effectiveness of the developed algorithm, we conducted an experimental study on all integers with 10, 11, 12, 13, and 14 bits. We used two types of algorithms for the comparison: (1) an exact algorithm that gives a minimal length AC; and (2) non-exact algorithms (deterministic and non-deterministic) that give a short chain.

The rest of the paper consists of five sections. In Section II, we describe the background to the AC problem and the EA. Next, in Section III, we briefly review the related works on the AC problem that use AI strategies to find a short chain. Then, in Section IV, we present the proposed algorithm and the details of the EA for a short AC. This is followed by Section V, in which we provide the results of our measurements to determine the effectiveness of the proposed algorithm on different ranges of bits for the exponent e, as well as the results of a comparison with the outputs of exact and approximate solutions in the literature. Finally, in Section VI, we conclude the work and highlight our future works.

II. Background

In this section, first, we provide a brief background on the definition of the AC problem, the types of chain and their elements [22]. Then, we give an overview of the EA and its components.

A. The Addition Chain Concept

An AC for a natural number e is a sequence of natural numbers e_0,e_1,e_2,…,e_m such that.

- $e_0 = 1$,
- $e_i = e_j+e_k$, $0\leq k \leq j < i$, and
- $e_m = e$.

The length of the AC for e is the number of steps needed to compute e, which is equal to the number of elements in the chain minus one, m. The second rule for generating a chain for e leads to the possibility of constructing a large number of different chains for e. This means that the lengths of the different chains for the same e may be different in general. Therefore, there are two types of chain based on the length of the chain as follows:

- If the length, m, of an AC for e is minimal, the length of the chain is denoted as $\ell(e)$ and the chain is termed a shortest AC.
- If the length, m, of the AC for e is greater than $\ell(e)$, the chain is termed a short AC.

For the natural number e, we define two functions as follows:

- $\lambda(e)$ is the length of the binary representation for e minus one and equal to $\lfloor log_2 e \rfloor$.
- $v(e)$ is the number of 1's in the binary representation of e.

There are different types of steps involved in the generation of an AC. The important steps are defined as follows:

- The ith step in an AC is termed star, if $e_i = e_{i-1}+e_k$, $0\leq k < i$.
- The ith step in an AC is termed doubling, if $e_i = e_{i-1}+e_{i-1}$.
- The ith step in an AC is termed plus-one, if $e_i = e_{i-1}+e_0$.

B. The Evolutionary Algorithm

An EA is a generic population-based metaheuristic optimization algorithm. An EA uses mechanisms inspired by

biological evolution, such as reproduction, mutation, recombination, and selection.

Given a quality function to be maximized/minimized, a set of candidate solutions or population, called parents (i.e., elements of the function's domain), can be randomly generated. Then, a quality function can be applied to these candidates to evaluate their fitness values, where the higher the fitness the better. Based on these fitness values, the best candidates are chosen to seed the next generation. This is done using two main strategies:

- A variation operator that generates the necessary variety within the population to be used in the next generation. Examples of variation operators are recombination and mutation.
- Selection that acts as a force that increases the mean quality of the solutions in the population.

Recombination is an operator that is applied to two or more selected candidates to produce one or more new candidates (children). One the other hand, the mutation operator uses one candidate, which results in one new candidate. Hence, these operations on candidates to create a set of new candidates (offspring). The candidates' fitness levels are evaluated before they compete – based on their fitness (and sometimes age) – with the old leads for a place in the next generation. This process is repeated until either a candidate with sufficient quality is found or a previously set computational limit is reached. Note that a combination of variation and selection leads to improved fitness values of consecutive populations.

III. Related Works

In this section, we briefly review different AI techniques that have been proposed to find a short AC.

A number of researchers have presented solutions to the AC problem based on the GA. Nedjah and Mourelle [27,28] used binary encoding to represent the solution, where 1 indicates that the number is present in the AC and 0 otherwise. They used four standard crossover operations: single-point, double-point, uniform, and arithmetic. Mutation is done by randomly changing some genes from 0 to 1 and vice-versa. The fitness function is based on the validity of the AC and its length. Cruz-Cortés et al. [13] adopted an integer encoding approach using variable-length chromosomes. They use a one-point crossover operator and the fitness function is based on the length of the AC.

The GA proposed by Osorio-Hernández et al. [31] works only on valid ACs (invalid chains are discarded) and represents each number from the AC corresponding to a gene in the chromosome. They use a repair process to generate valid solutions in the initial population and mutation operator. Also, they use a two-point crossover operator which applies value and rule copying operations. The fitness function is again based on the length of the AC.

Domínguez-Isidro et al. [16] proposed using an EA, which is based on a mutation operator that is able to produce as set of valid solutions to the AC problem from a single solution. In addition, the proposed algorithm includes a replacement technique based on stochastic elements to introduce diversity into the population. The numbers in the AC are represented directly in the solution and their fitness is the length of the AC.

Picek et al. [32] presented a GA with a repair strategy to enhance the performance of AC generation. The solution is encoded as a set of tuples of the form (v_k,i,j), where v_k is the value of the kth number in the AC while i and j are the positions of the previous numbers v_i and v_j, respectively, in the AC forming v_k. The algorithm implements crossover and mutation operators similar to the ones used in the previous literature [27,28]. However, these operators are followed by a repair operation to guarantee the validity of the solution.

Other researchers have used optimization techniques to find short ACs. Nedjah and Mourelle [29] proposed an ant colony optimization (ACO) approach based on a multi-agent schema with two types of memory: shared and local. The same authors implemented the ACO algorithm on a system-on-chip (SoC) to improve the computations [30].

On the other hand, Léon-Javier et al. [23] proposed algorithm based on particle swarm optimization (PSO). Mullai and Mani [26] used PSO and simplified swarm optimization to generate ACs for RSA for the purpose of using ACs to optimize computations in encryption/decryption processes to reduce processing time and power consumption in mobile devices. Cruz-Cortes et al. [14] introduced an artificial immune system for finding short ACs for moderate-sized exponents (i.e., less than 20 bits) and large exponents (i.e., up to 2048 bits).

IV. Proposed Algorithm

In this section, we describe the main steps of the proposed algorithm that is aimed at solving the AC problem by using an EA. The input of the algorithm is a natural number $e \geq 2$ and the output is a short AC $e_0,e_1,e_2,\ldots,e_m$.

In order to describe the proposed algorithm using EA, first, we describe the main components of EA involved in solving the AC problem, namely, representation, initial population, fitness function, variant operators, and survivor selection. Then, we present the steps of the proposed algorithm.

A. Representation

Since an AC for a natural number e is a sequence of natural numbers, $e_0,e_1,e_2,\ldots,e_m$, the individual AC in the proposed EA is an array of dynamic length. The length of each individual is not fixed during the computation for two reasons. The first reason is the length of each individual in the search space may be different from the others. The second reason is the length of an individual AC may change during the different mutation operations.

In order to represent the population of the problem, we assume that the number of elements in the population for the AC problem is n. The population represents as a 2-dimensional array, E. The first dimension represents the number of individuals in the search space, n, while the second dimension is the length of each individual which is variable.

For example, let us assume that e = 37 and that we have four ACs for e, (1,2,4,8,16,32,36,37),

(1,2,3,6,9,15,17,20,29,35,37), (1,2,4,8,16,18, 36,37), and (1,2,3,5,10,20,30,35,37). Then the four chains can be represented as four arrays of different lengths, i.e., 8, 11, 7, and 8, respectively, as in Fig. 1.

1	1	2	4	8	16	32	36	37			
2	1	2	3	6	9	15	17	20	29	35	37
3	1	2	4	8	16	18	36	37			
4	1	2	3	5	10	20	30	35	37		

Fig. 1. Four Chains for *e*=37.

B. Evaluation Function

Each individual AC needs to be evaluated using a fitness function to decide the best set of solutions. Since the goal of solving the AC problem is to find a sequence of natural numbers with minimal length, the value of the fitness function for each individual is the length of chain. Formally, if we have a chain $E_k = (e_{k,0}, e_{k,1}, \dots, e_{k,m})$, then $f(E_k) = m$, because the first element, e_0, in the chain is not counted. Note that, in some times, we write $E_k = (e_0, e_1, \dots, e_m)$ for simplicity.

C. Initial Population

The first step in any EA is to generate a random initial population that consists of *n* individuals, i.e., ACs. Each individual should satisfy the following general conditions to be a valid AC:

- The first two elements in any chain are 1 and 2.
- Any element, e_i, in the chain can be constructed from the addition of any two previous elements, e_j and e_k, where $j, k < i$.
- No elements in the chain are repeated.
- The last element in the chain is *e*.
- The elements of the chain are in increasing order of size.

All elements in the AC, except the first two elements, are generated randomly. In order to generate the elements, we use the following variables:

- r_1, which is a real random number between 0 and 1. The variable is generated using a user-defined function *RandomReal*(r_1).
- r_2, which is an integer random number between 0 and an integer α-1. The variable is generated using a user-defined function *RandomInt*(r_2,α).
- *diff*, which is the difference between *e* and the current element, e_i, in the chain.

The previous proposed strategy [16] used to generate an element in a chain is based on applying one of the following rules based on a random number:

R1. Double the last element: $e_{i+1} = e_i + e_i$.

R2. Add the last two elements: $e_{i+1} = e_i + e_{i-1}$

R3. Add the last element with a random element from 0 to *i*-1: $e_{i+1} = e_i + e_j$, $0 \le j < i$.

In our proposed algorithm, we modify this strategy by measuring, first, the difference, *diff*, between the target number, *e*, and the last element generated in the chain, e_i. Therefore, we have three cases for the value of *diff* as follows:

Case 1: When *diff* is greater than or equal to e_i, the algorithm executes one of the rules, R1, R2 or R3, based on the value of a random number.

Case 2: When *diff* is greater than or equal to e_{i-1}, the algorithm executes one of the rules, R2 or R3, based on the value of a random number.

Case 3: When *diff* is less than e_{i-1}, the algorithm executes R3 based on the value of a random number.

One of the advantages of using *diff*, in the first two cases, compared to other previous strategies is that it prevents the occurrence of the following cases: (1) generation of repeated elements, (2) generation of an element greater than the target element, and (3) generation of a non-increasing sequence without making a repair to the chain. For the third case, we need to repair the generated elements such that the new element is less than or equal to *e*. Note that we can add more cases for the variable *diff*, but this leads to an increase in the running time in general.

The two subroutines InitialPopulation and GenerateSubChain are used to create the initial population.

Subroutine InitialPopulation(e, n)
Begin
1. For $k = 1$ to n do
2. $e_{k,0} = 1$, $e_{k,1} = 2$
3. *RandomReal*(r_1)
4. If ($r_1 \ge 0.5$)
5. $e_{k,2} = 4$
6. Else
7. $e_{k,2} = 3$
8. $i = 2$
9. *GenerateSubChain*(e,E_k, i)
10. Return *E*

End

Subroutine GenerateSubChain(e, E_k, i)
Begin
1. Repeat
2. $diff = e - e_{k,i}$
3. If (*diff* $\geq e_{k,i}$) then
4. $RandomReal(r_1)$
5. If ($r_1 \geq 0.5$)
6. $e_{k,i+1} = e_{k,i} + e_{k,i}$
7. Else
8. $RandomReal(r_1)$
9. If ($r_1 \geq 0.5$)
10. $e_{k,i+1} = e_{k,i} + e_{k,i-1}$
11. Else
12. $RandomInt(r_2, i-1)$
13. $e_{k,i+1} = e_{k,i} + e_{k,r_2}$
14. Else
15. If (*diff* $\geq e_{i-1}$) then
16. $RandomReal(r_1)$
17. If ($r_1 \geq 0.5$)
18. $e_{k,i+1} = e_{k,i} + e_{k,i-1}$
19. Else
20. $RandomInt(r_2, i-1)$
21. $e_{k,i+1} = e_{k,i} + e_{k,r_2}$
22. Else
23. $RandomInt(r_2, i-1)$
24. $e_{k,i+1} = e_{k,i} + e_{k,r_2}$
25. *j*=*i*-2
26. While ($e_{k,i+1} > e$ do
27. $e_{k,i+1} = e_{k,i} + e_{k,j}$
28. *j*=*j*-1
29. $i = i + 1$
30. Until $e_{k,i} = e$

End

D. Mutation

In the EA, random strategy is used to mutate an individual element, AC, which means that for a given chain, E_k, we mutate E_k from a random position in the chain. This strategy to mutate an AC of length m is based on the following idea [16]: First, the algorithm picks, randomly, a position, *j*, in the AC between 3 and m. Second, the algorithm eliminates the elements of the AC from j to m and generates new random elements in the AC using the same strategy, i.e., the *GenerateSubChain* subroutine. Third, the algorithm repeats the second step *t* times and then the algorithm selects the best AC that has the smallest length.

In our algorithm, we modify the above method to mutate the AC so that the selected random position that is used to mutate the chain is different, if possible, in each iteration, where we have *t* iterations. In the above-described mutated strategy, the position of mutation is fixed during all *t* iterations. Also, when we generate a random position in the chain E_k, in our method, we select it from a range of 3 to the number of bits in e, say $\lambda(e)$.

In order to generate *t* mutated chains from the chain, E_k, we use two auxiliary arrays. The first auxiliary array, Aux_1, is used to save the best mutated chain from *e*, while the second auxiliary array, Aux_2, is used to generate the mutated chain from the chain, E_k. At the end of *t* iterations, the best mutated AC for *e* is selected as offspring.

Subroutine Mutation($\boldsymbol{e, E, n, t}$)
Begin
1. For $k = 1$ to n do
2. $RandomInt(r_2, \lambda(e) - 2)$
3. $r_2 = r_2 + 2$
4. $Aux_1(0, \dots, r_2) = E_k(e_0, \dots, e_{r_2})$
5. $GenerateSubChain(e, Aux_1, r_2)$
6. For $j = 2$ to t do
7. $RandomInt(r_2, \lambda(e) - 2)$
8. $r_2 = r_2 + 2$
9. $Aux_2(0, \dots, r_2) = E_k(e_0, \dots, e_{r_2})$
10. $GenerateSubChain(e, Aux_2, r_2)$
11. If ($|Aux_2| < |Aux_1|$) then
12. $Aux_1 = Aux_2$
13. $M_k = Aux_1$
14. Return *M*

End

Fig. 2 shows an example of applying the mutation to the chain E_k = (1,2,3,6,9,15,17,20,29,35,37), where the four random numbers used in the t = 4 iterations are 4, 5, 3, and 3, respectively. The results of the mutation are four chains with lengths 8, 7, 10, and 9, respectively. The subroutine *Mutation* returns the offspring (1, 2, 3, 6, 9, 18, 36, 37) of length 7 instead of 10.

e	1	2	3	6	9	15	17	20	29	35	37
$r_2 = 4$	1	2	3	6	12	24	30	36	37		
$r_2 = 5$	1	2	3	6	9	18	36	37			
$r_2 = 3$	1	2	3	5	10	13	18	20	33	36	37
$r_2 = 3$	1	2	3	4	7	14	28	32	35	37	

Fig. 2. Four Mutated Chains for *e*=(1,2,3,6,9,15,17,20,29,35,37).

E. Survivor Selection

Survivor selection is the process of generating the next population from two population sets, where the first set is the current population and the second is the set of offspring that is generated from the mutation of the current population.

The method of selecting the survivors is done by combining the two sets, current and offspring, into a set of chains and then sorting the combined set based on the length of the chain in increasing order of length. After the sorting

process has been completed, the first n smallest chains are selected based on length, such that the selected chains contain different elements. The two chains, E_i and E_j, are different if (1) the length of the two chains is different or (2) there exists at least one position, *pos*, such that the element at *pos* in the chain E_i is not equal to the element at *pos* in the chain E_j.

Our algorithm for survivor selection is different than that proposed in the previous work [16] in that our algorithm eliminates the step of calculating the fitness of a chain from q randomly selected chains. Also, our modified survivor selection algorithm involves deleting duplicated chains, i.e., two chains of the same length and containing the same elements are deleted. To do this, we use a different method to identify duplicated chains. The complete steps of the process of selecting the next generation are given in Subroutine SurvivorSel.

Subroutine SurvivorSel(E, O)

Begin

1. $Temp = E \cup O$
2. Sort(*Temp*)
3. $Aux_1 = Temp_1$
4. $i = 1, k = 1$
5. Repeat
6. $\quad i = i + 1$, *flag=false*, $j = i - 1$
7. $\quad$ While ($j > 0$) and (*flag=false*) do
8. $\quad\quad$ If ($|Temp_i| \neq |Temp_j|$) then
9. $\quad\quad\quad$ *flag=true*, $k = k + 1$
10. $\quad\quad\quad Aux_k = Temp_i$
11. $\quad\quad$ Else If (Equal($Temp_i, Temp_j$))
12. $\quad\quad\quad$ *flag=true*
13. $\quad\quad j = j - 1$
14. $\quad$ If (*flag=false*) then
15. $\quad\quad k = k + 1$
16. $\quad\quad Aux_k = Temp_i$
17. Until ($k \geq n$) or ($i \geq |Temp|$)
18. Return Aux

End

F. Complete Proposed Algorithm

The proposed algorithm starts by generating an initial population, *E*, consisting of *n* ACs. Each AC is a valid chain, meaning that it satisfies the conditions for an AC. Then the algorithm repeats a sequence of steps consisting of *MaxNoIter* iterations, where *MaxNoIter* is one of the parameters used in the EA and represents the maximum number of iterations. The first step in the repetition loop is the application of the mutation operator on the population set, *E*. The output of this step is a set of mutated ACs of size *n*. The second step in the repetition loop is the application of the survivor selection subroutine on the combination of the current population and mutated sets. The output of this step is the next generation of the population of ACs of size *n*. The complete pseudocode of the proposed algorithm is shown in Algorithm Evolutionary Addition Chain, EAC.

Algorithm EAC (Evolutionary Addition Chain)

Input: a natural number $e \geq 2$.

Output: a short chain $(e_0, \ldots, e_l)$.

Begin

1. *InitialParameters*(*MaxNoIter*, n, t)
2. E =*InitialPopulation*(e, n)
3. For $i = 1$ to *MaxNoIter* do
4. $\quad O$ =*Mutation*(e, E, n, t)
5. $\quad E$= *SurvivorSel* (E, O)
6. Return E_1

End

V. PERFORMANCE EVALUATION OF THE EAC

In this section, we study the performance of the proposed algorithm (EAC) in terms of the length of the generated AC. In order to verify performance, we compare the output of the EAC, i.e., the length of AC, with that of the following types of algorithm:

- An exact algorithm, denoted as ExA.
- Two non-exact deterministic algorithms, namely, the binary algorithm (BA) and the continued fraction algorithm (CFA).
- Two non-exact non-deterministic algorithms, one based on the GA and one based on the EA.

The following subsections describe the experimental setup, including the machine and software used in the coding of the algorithms, as well as the initialization parameters and the data required to execute the EA. Then, in the second, third, and fourth subsections we introduce and explain the results of the experimental study for the first, second, and third type of comparison, respectively.

A. Experimental Setup

Several parameters can be used in an experimental study to assess their effects on the performance of AC algorithms that are based on AI techniques:

- The first parameter, $\lambda(e)$, is the size of the natural number e, which is equal to the number of bits, *b*. In our experiment we used b = 10, 11, 12, 13, and 14 for all studied algorithms.
- The second parameter, $\varphi(b)$, is the set of all numbers of size (number of bits), *b*, which is equal to 2^b. This means that $\varphi(b) = \{2^b, 2^b + 1, 2^b + 2, \ldots, 2^{b+1} - 1\}$. This parameter was used for all studied algorithms in our experiment.
- The third parameter, *n*, is the size of the population. In our study, *n* = 100, in line with previous works. This parameter was used for all studied EA.
- The fourth parameter, *MaxNoIter*, is the maximum number of generations used in the computation. In our

experiment, we set *MaxNoIter* as 300 for all studied EA.

- The fifth parameter, *t*, is the number of mutated sequences. In our study, *t* = 4, similar to previous works [16]. This parameter was used for all studied EA.

All algorithms were implemented using C++ language on a computer running a Windows operating system, which had a 2.4-GHz processor and a 32-GB memory.

For a comparison between two algorithms, A and B, we first determined the number of bits b and generated all natural numbers of b bits, $\varphi(b)$. Second, we applied the two algorithms on the number $e = 2^b$ and measured the length of the output of both algorithms. Third, we increased the value of e by one. Then we repeated the second and third steps until the last element, $e = 2^{b+1} - 1$, was reached.

In our comparisons of two algorithms, we measured the following criteria that are related to the length of the chain:

- $\#(A = B)$, which is the number of cases where the length of the AC generated by both algorithms is equal.
- $\#(A < B)$, which is the number of cases where the length of the AC generated by algorithm A is less than that generated by algorithm B.
- $\#(A > B)$, which is the number of cases where the length of the AC generated by algorithm A is greater than that generated by algorithm B.
- $MaxDiff(A, B)$, which is the maximum difference in the length between the lengths of the ACs generated by algorithm A compared to those generated by algorithm B for a fixed number of bits, b. Note that $MaxDiff(A, B)$ is not necessarily equal to $MaxDiff(B, A)$.

B. EAC and ExA

In this subsection, we compare the proposed algorithm, EAC, with the exact solution, ExA. The minimal length of ACs for e can be obtained from [37].

Table I displays the results of implementing ExA and EAC on different values of b as described in the experimental setup subsection. The results in the table lead to the following observations:

- For a fixed value of b, the output, i.e., the length of the AC, of both algorithms is the same in most of the cases, in that the percentage of $\#(EAC = ExA)$ is greater than 75%.
- The percentage of $\#(EAC > ExA)$ increases with an increase in the value of b. This means that with an increase in the value of b, the parameter *MaxNoIter* should be increased to obtain AC with short length near to the shortest length.
- When there is a difference in the lengths, the length of the AC generated by the EAC is near to the minimal length for the studied cases, because the maximum difference between the short and the shortest chains is 2.
- Even where the difference in the length of the AC generated by both algorithms is greater than 1, the percentage of such cases is small, in that it is less than 2%. For example, for b = 13 and 14, the percentage of cases that have a length greater than the minimal length of 2 is 0.5% and 1.9%, respectively.

C. EAC, BA and CFA

In this subsection, we compare the proposed algorithm, EAC, with two deterministic non-exact algorithms, BA and CFA that are based on the Fermat strategy [11].

Table II displays the results of comparing the performance of EAC and BA. These results lead to the following observations:

- The EAC performs better than the BA for all values of b, because the length of the AC generated by the EAC is less than or equal to that obtained by BA.
- The length of the AC generated by EAC is less than that obtained by BA in most of the studied cases with a percentage more than 85%.
- The maximum difference between the output of both algorithms increases with an increase in the number of bits, b. On the other hand, the percentage of the number of equal cases decreases with an increase in b. For example, the maximum difference between the two algorithms is 7, 8, and 9 for b = 12, 13, and 14, respectively.
- Most of the differences in the length of the AC generated by EAC and BA occur when *MaxDiff* = 1, 2, 3, and 4, as illustrated in Fig. 3.

TABLE I. COMPARISON OF EAC AND EXA

Criteria	b				
	10	11	12	13	14
$\#(EAC = ExA)$	1017 (99.3%)	1961 (95.7%)	3706 (90.5%)	6897 (84.2%)	14042 (76.5%)
$\#(EAC > ExA)$	7 (0.7%)	87 (4.3%)	390 (9.5%)	1295 (15.8%)	2342 (23.5%)
$MaxDiff(ExA, EAC)$	1	1	1	2 (0.5%)	2 (1.9%)

TABLE II. COMPARISON OF EAC AND BA

Criteria	b				
	10	11	12	13	14
$\#(EAC = BA)$	145 (14.2%)	207 (10.1%)	296 (7.2%)	455 (5.6%)	677 (4.1%)
$\#(EAC < BA)$	879 (85.8%)	1841 (89.9%)	3800 (92.8%)	7737 (94.4%)	15707 (95.9%)
$\#(EAC > BA)$	0	0	0	0	0
$MaxDiff(EAC, BA)$	5	7	7	8	9
$MaxDiff(BA, EAC)$	0	0	0	0	0

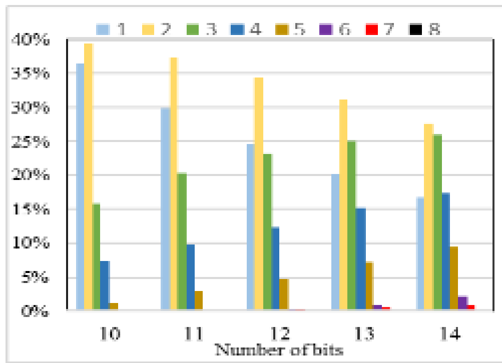

Fig. 3. Distribution of difference in Lengths for EAC< BA.

Table III displays the results of comparing EAC and CFA. These results lead to the following observations:

- The EAC shows better performance than the CFA for all values of b with percentage more than 25%. Also, the length of the AC generated by both algorithms is equal in more than 65% of all cases.
- In a few cases (less than 5%), the length of the AC generated by the CFA is less than that obtained by the EAC.
- The maximum difference between the two algorithms is limited to 2.
- When $\#(EAC < CFA)$, most cases occur when *MaxDiff* = 1, as illustrated in Fig. 4.

D. EAC and GA

In this subsection, we compare the proposed algorithm, EAC, with the GA. The details of the GA and its pseudocode can be found in [38].

Table IV displays the results of running the EAC and GA on different values of b. The results lead to the following observations:

- The EAC outperforms GA for all values of b, because the length of the AC generated by the EAC is less than or equal to that obtained by GA in 99% of cases.
- The percentage of $\#(EAC < GA)$ increases with an increase in the value of b. For example, the percentage of cases that have $\#(EAC < GA)$ for b = 11 and 12 are equal to 65.87% and 71.8%, respectively.
- When the GA performs better than the EAC, the maximum difference in the length of ACs is 1. On the other hand, when the EAC performs better than the GA, the maximum difference in the length of the AC increases with an increase in b.
- Most of the differences in the AC lengths generated by the EAC and the GA occur when $MaxDiff(EAC, GA)$ =1, 2 and 3, as illustrated in Fig. 5.

Remark: We compared EAC with the EA that is proposed by [16], we found that there is no significant difference between the output of both algorithms, i.e., minimal length of ACs. The main difference between them is in the running time, where our algorithm is faster than the EA in [16] with almost 25%. The reasons for reducing the running time come from (1) using the variable diff during generation the elements of chains, and (2) our proposed method for survivor selection.

TABLE III. COMPARISON OF EAC AND CFA

Criteria	***b***				
	10	**11**	**12**	**13**	**14**
$\#(EAC = CFA)$	772 (75.4%)	1433 (70.0%)	2805 (68.5%)	5507 (67.2%)	10856 (66.3%)
$\#(EAC < CFA)$	252 (24.6%)	610 (29.8%)	1262 (30.8%)	2536 (31.0%)	4934 (30.1%)
$\#(EAC > CFA)$	0	5 (0.2%)	29 (0.7%)	149 (1.8%)	594 (3.6%)
MaxDiff (EAC, CFA)	2	2	2	2	2
MaxDiff (CFA, EAC)	0	1	1	1	2

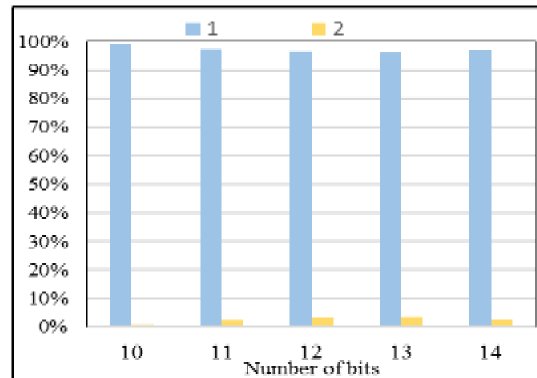

Fig. 4. Distribution of difference in Lengths for EAC< CFA.

TABLE IV. COMPARISON OF EAC AND GA

Criteria	***b***				
	10	**11**	**12**	**13**	**14**
$\#(EAC = GA)$	435 (42.5%)	697 (34.0%)	1148 (28.0%)	1920 (23.4%)	2281 (13.9%)
$\#(EAC < GA)$	589 (57.5%)	1349 (65.9%)	2941 (71.8%)	6255 (76.4%)	14103 (86.1%)
$\#(EAC > GA)$	0	2 (0.1%)	7 (0.2%)	17 (0.2%)	0
MaxDiff(EAC, GA)	4	4	5	6	7
MaxDiff(GA, EAC)	0	1	1	1	0

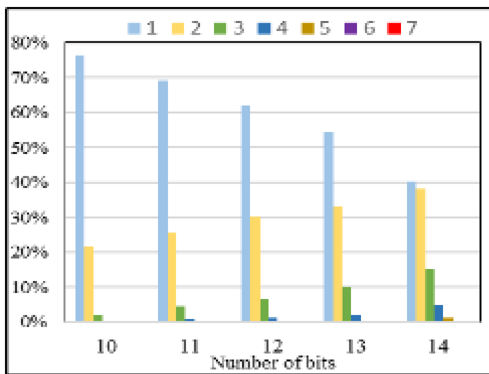

Fig. 5. Distribution of difference in Lengths for EAC< GA.

VI. Conclusion and Future Works

The AC can be used to decrease the number of multiplications in the encryption procedure of the RSA cryptosystem. In order to use the AC effectively, it is necessary to develop a method to find the shortest AC or a short AC. In this work, we discussed how we modified the EA to find a short AC in an efficient and simple way by focusing on the main components of EA such as representation, population, mutation and survivor selection. The proposed algorithm, EAC, was then implemented and compared with three types of algorithm (exact, non-exact deterministic, and non-exact non-deterministic) to assess its effectiveness. The experimental results indicated that the EAC showed good performance in comparison with the other types of algorithm when applied to natural numbers of 10, 11, 12, 13, and 14 bits.

In future work, we will extend our proposed algorithm to deal with large numbers of bits. We will also compare the performance of our algorithm with a wider range of deterministic and non-deterministic algorithms. Furthermore, we will consider running time as another criterion in the performance comparison.

Acknowledgment

This work has been funded by Scientific Research Deanship at the University of Ha'il – Saudi Arabia through project number RG-191309.

References

[1] H. Bahig, "Improved generation of minimal addition chains," Computing, vol. 78, pp. 161–172, 2006.

[2] H. Bahig, "On a generalization of addition chains: Addition–multiplication chains," Discrete mathematics, vol. 308, no. 4, pp. 611-616, 2008.

[3] H. Bahig, "Star reduction among minimal length addition chains," Computing, vol. 91, pp. 335–352, 2011.

[4] H. Bahig, "A fast optimal parallel algorithm for a short addition chain," J Supercomputing, vol. 74, no. 1, pp. 324-333, 2018.

[5] H. Bahig, and K. AbdElbari, "A fast GPU-based hybrid algorithm for addition chains," Cluster Computing, vol. 21, pp. 2001–2011, 2018.

[6] H. M. Bahig, A. Alghadhban, M. Mahdi, K. Alutaibi, and H. Bahig, "Speeding up the multiplication algorithm for large integers", Engineering, Technology & Applied Science Research, vol 10, no. 6, pp 6533-6541, 2020.

[7] H. Bahig, and H. Bahig, "A new strategy for generating shortest addition sequences," Computing, vol. 91, no. 3, 285-306, 2011.

[8] H. Bahig, H. Bahig, and K. Fathy, "Fast and scalable algorithm for product large data on multicore system," Concurrency and Computation: Practice and Experience, online published 2019, https://doi.org/10.1002/cpe.5259.

[9] H Bahig, and A Mahran, "Efficient generation of shortest addition-multiplication chains," Journal of the Egyptian Mathematical Society, vol. 26, no. 3, pp. 509-521, 2018.

[10] H. Bahig, and Y. Kotb, "An efficient multicore algorithm for minimal length addition chains," Computers, vol. 8, no. 1, pp. 1-23, 2019.

[11] F. Bergeron, J. Berstel, and S. Brlek, "Efficient computation of addition chains," J. de Theorie Nombres de Bordeaux, vol. 6, pp. 21-38, 1994.

[12] N. Clift, "Calculating optimal addition chains," Computing, vol. 91, pp. 265–284, 2011.

[13] N. Cruz-Cortés, F. Rodríguez-Henríquez, R. Juárez-Morales, C. Coello, "Finding optimal addition chains using a genetic algorithm approach," Lecture Notes in Computer Science, vol. 3801, pp. 208-215, 2005.

[14] N. Cruz-Cortes, F. Rodriguez-Henriquez and C. Coello, "An artificial immune system heuristic for generating short addition chains," IEEE Transactions on Evolutionary Computation, vol. 12, no. 1, pp. 1-24, 2008.

[15] Cruz-Cort´es, Nareli,c Rodr´ıguez-Henr´ıquez, Francisco, Ju´arez-Morales, Ra´ul and Carlos A. Coello Coello (2005). Finding optimal addition chains using a genetic algorithm approach. Y. Hao et al. (Eds.): CIS 2005, Part I, LNAI 3801, 208–215.

[16] S. Dominguez-Isidro, E. Mezura-Montes, and L. Osorio-Hernandez, "Evolutionary programming for the length minimization of addition chains," Engineering Applications of Artificial Intelligence, vol. 37, pp. 125–134, 2015.

[17] P. Downey, B. Leong, and R. Sethi, "Computing sequences with addition chains," SIAM Journal on Computing, vol. 10, no. 3, pp. 638-646, 1981.

[18] K. Fathy, H. Bahig, H., Bahig, and A. Ragb, "Binary addition chain on EREW PRAM," Lecture Notes of Computer Science, vol. 7017, pp. 321-330, 2011.

[19] K. Fathy, H. Bahig, and A. Ragab, "A fast parallel modular exponentiation algorithm," Arabian Journal for Science and Engineering, vol. 43, pp. 903–911, 2018.

[20] D. Gordon, "A survey of fast exponentiation methods," Journal of Algorithms, vol. 27, no. 1, pp. 129-146, 1998.

[21] R. Goundar, K. Shiota, and M. Toyonaga, "New strategy for doubling-free short addition-subtraction chain," Applied Mathematics & Information Sciences, vol. 2, no. 2, pp. 123–133, 2008.

[22] D. Knuth, "The Art of Computer Programming: Seminumerical Algorithms," vol. 2, 1973, Addison-Wesley.

[23] A. León-Javier, N. Cruz-Cortés, M. Moreno-Armendáriz, and S. Orantes-Jiménez, "Finding minimal addition chains with a particle swarm optimization algorithm," Lecture Notes in Computer Science, vol. 5845, pp. 680-691, 2009.

[24] A. Jayaram, and S. Deb, "A hybrid addition chaining based light weight security mechanism for enhancing quality of service in IoT," Wireless Personal Communications, vol. 113, pp. 1073–1095, 2020.

[25] K. Jrvinen, V. Dimitrov, and R. Azarderakhsh, "Generalization of addition chains and fast inversions in binary fields," IEEE Trans Computers, vol. 64, no. 9, pp. 2421–2432, 2015.

[26] A. Mullai, and K. Mani, "Enhancing the security in RSA and elliptic curve cryptography based on addition chain using simplified swarm optimization and particle swarm optimization for mobile devices," International Journal of Information Technology, online published 2020, https://doi.org/10.1007/s41870-019-00413-8.

[27] N. Nedjah, and L. de Macedo Mourelle, "Minimal addition chain for efficient modular exponentiation using genetic algorithms," Lecture Notes in Computer Science, vol. 2358, pp. 88-98, 2002.

[28] N. Nedjah, and L. de Macedo Mourelle, "Minimal addition-subtraction chains using genetic algorithms," Lecture Notes in Computer Science, vol. 2457, pp. 303-313, 2002.

[29] N. Nedjah, and L. de Macedo Mourelle, "Finding minimal addition chains using ant colony," Lecture Notes in Computer Science, vol. 3177, pp. 642-647 , 2004.

[30] N. Nedjah, and L. de Macedo Mourelle, "High-performance SoC-based implementation of modular exponentiation using evolutionary addition chains for efficient cryptography," Applied Soft Computing, vol. 11, no. 7, pp. 4302-4311, 2011.

[31] L. Osorio-Hernandez, E. Mezura-Montes, N. Cruz-Cortes, and F. Rodriguez-Henriquez, "A genetic algorithm with repair and local search mechanisms able to find minimal length addition chains for small exponents," 2009 IEEE Congress on Evolutionary Computation, Evolutionary Computation, Trondheim, Norway, 18-21 May 2009, pp. 1422–1429.

[32] S. Picek, C. Coello, Jakobovic, D. Jakobovic, and N. Mentens, "Finding short and implementation-friendly addition chains with evolutionary algorithms. J Heuristics vol. 24, pp. 457–481, 2018.

[33] S. Sanchez, J. Osorno, and E. Camarillo, "Simulated annealing meta-heuristic for addition chain optimization," European Journal of Electrical and Computer Engineering, vol. 3, no. 6, pp. 1-4, 2019.

[34] E. Thurber, "Efficient generation of minimal length addition chains," SIAM J Computing, vol. 28, pp. 1247–1263, 1999.

[35] J. Yang and C. Chang, "Efficient residue number system iterative modular multiplication algorithm for fast modular exponentiation," IET Computers & Digital Techniques, vol. 2, no. 1, 1-5, 2008.

[36] Yen, S.-M, "Cryptanalysis of secure addition chain for SASC applications. Electronics Letters, vol. 31, no. 3, pp. 175–176, 1995.

[37] Shortest Addition Chain: http://wwwhomes.uni-bielefeld.de/achim/addition _chain.html

[38] Gentic Addition Chain Algorithm: https://github.com/josip-u/add-chain-solver.