A Genetic Algorithm with Repair and Local Search Mechanisms Able to Find Minimal Length Addition Chains for Small Exponents

Luis G. Osorio-Hernández, *Student Member, IEEE*, Efrén Mezura-Montes, *Member, IEEE*, Nareli Cruz-Cortés, *Member, IEEE* and Francisco Rodríguez-Henríquez, *Member, IEEE*

***Abstract*— In this paper, we present an improved Genetic Algorithm (GA) that is able to find the shortest addition chains for a given exponent e. Two new variation operators (special two-point crossover and a local-search-like mutation) are proposed as a means to improve the GA search capabilities. Furthermore, the usage of an improved repair mechanism is applied to the process of generating the initial population of the algorithm. The proposed approach is compared on a set of test problems with two state-of-the-art evolutionary heuristic-based approaches recently published. Finally, the modified GA is used to find the optimal addition chain length for a small collection of "hard" exponents. The results obtained are competitive and even better in the more difficult instances of the exponentiation problem that were considered here.**

I. Introduction

A prime finite field, denoted as F=GF(p), is the set of p integers in the range $[0, 1, 2, \ldots, p-1]$, where p is a prime number. In a prime finite field, customary arithmetic operations such as addition, subtraction, multiplication, division by nonzero integers and exponentiation are all well defined. In order to ensure the *closure property*, which guarantees that the result of any arithmetic operation will produce an element within the field (i.e., a positive integer less than p), all arithmetic computations are performed by taking the remainder on integer division by p. Hence, the result of adding or multiplying any two arbitrary field elements will always be an element in the field. Furthermore, the usual algebraic laws, namely, commutative, associative and distributive laws, hold [1]. For example, let us consider the prime field GF($p = 23$). That field has a total of 23 elements corresponding to the integers in the range $[0, 22]$. Given the field elements $a = 10$ and $b = 18$, their field multiplication $d = a \cdot b$ is calculated as, $d = a \cdot b = 10 \cdot 18 \bmod 23 = 19$.

Field exponentiation can be defined in terms of field multiplication as follows. Let a be an arbitrary element of a finite field F =GF(p). Let also e be defined as an arbitrary positive integer. Then, field exponentiation of an element a raised to the power e is defined as the problem of finding an element $b \in F$ such that,

$$b = a^e \bmod p \tag{1}$$

Luis G. Osorio-Hernández and Efrén Mezura-Montes are with the National Laboratory on Advanced Informatics (LANIA A.C.). Rébsamen 80, Centro, Xalapa, Veracruz, 91000, MEXICO. Email: ({luis,emezura}@lania.mx).
Nareli Cruz-Cortés is with the Center for Computing Research (CIC) of National Polytechnic Institute (IPN), Mexico City, Mexico. Email: (nareli@cic.ipn.mx).
Francisco Rodríguez-Henríquez is with CINVESTAV, Department of Computing, Mexico City, Mexico. Email: (francisco@cs.cinvestav.mx).

where as it has been said, p is a large prime. For example, let us consider once again the prime field GF($p = 23$), the field element $a = 10$ and the exponent $e = 25$. Then, it follows that, $b = 10^{25} \bmod 23 = 11$.

Field exponentiation is a major building block of computational number theory, since this operation is utilized for computing many relevant problems such as integer prime testing, integer factorization, field multiplicative inverse computation, etc. Moreover, modular exponentiation is the most important building block in several major public-key cryptographic schemes such as RSA, Diffie-Hellman and the Digital Signature Algorithm (DSA) [2], [1]. For example, the DSA signature scheme generates the public/private key pair of a given user by computing $y = g^x \bmod p$, where $g \in [2, \ldots, p-1]$, x is a 160-bit positive number and p is a 512-bit prime number.

In this paper we address the problem of finding the minimum number of multiplications required for computing b in (1). We shall assume that arbitrary choices of the base element a are allowed but we will consider that the exponent e has been previously fixed.

It is worth noticing that taking advantage of the linearity property of the modular operation, (1) can be evaluated by performing a reduction modulo p at each step of the exponentiation thus guaranteeing that all the partial results will not grow larger than twice the length of the modulus p. Hence, in the rest of this paper we will consider that every multiplication operation always includes a subsequent reduction step.

The problem of computing powers of the base element a, can be directly translated to an addition calculation, which leads to the idea of using *addition chain* for computing the field exponentiation problem. An addition chain dictates the correct sequence of multiplications required for performing an exponentiation to a fixed exponent e. The first and last elements of an addition chain are always 1 and e, respectively. Moreover, an addition chain is a monotonically increasing sequence of integers such that any intermediate element can be obtained by adding two previous elements that can or cannot be different. For instance, the six-step addition chain $(1, 2, \underline{4,} 6, \underline{12,} 13, 25)$ leads to the following scheme for the computation of $10^{25} \bmod 23$,

$a^1 = 10^1 = 10;$
$a^2 = a^1 \cdot a^1 = 10^1 \cdot 10^1 = 8;$
$a^4 = a^2 \cdot a^2 = 8 \cdot 8 = 18;$
$a^6 = a^4 \cdot a^2 = 18 \cdot 8 = 6;$
$a^{12} = a^6 \cdot a^6 = 6 \cdot 6 = 13;$
$a^{13} = a^{12} \cdot a^1 = 13 \cdot 10 = 15;$
$a^{25} = a^{13} \cdot a^{12} = 15 \cdot 13 = 11 = 10^{25} \bmod 23.$

It can be shown that the shortest addition for $e = 25$ has length $l = 6$. Therefore, the above procedure for computing $10^{25} \bmod 23$, is optimal in the sense that utilizes the theoretical minimum number of field multiplications for performing the required exponentiation.

The problem to find the optimal addition chain has been treated as an optimization problem and it has been solved mostly by using deterministic approaches. However, in the recent years, some heuristic-based approaches have been proposed to tackle this problem. In this paper we further explore the use of one of them, a Genetic Algorithm, in order to improve its capabilities in this type of search space. An improved and more extensively used repair mechanism besides two novel variation operators are proposed.

The rest of this paper is organized as follows: In Section II the addition chain problem is formally stated. Section III presents a brief summary of previous techniques proposed to solve this problem. After that, in Section IV we detail the proposed approach. Section V includes the experimental design, the results obtained and their discussion. Finally, the conclusions of this work and the future paths of research are summarized in Section VI.

II. Problem Statement

The problem we want to solve is to find the shortest addition chain for a given exponent e, that is, the addition chain with the lowest length l.

An addition chain U with length l is defined as a sequence of positive integers $U = u_1, u_2, \ldots, u_i, \ldots, u_l$, with $u_1 = 1$, $u_2 = 2$ and $u_l = e$, and $u_{i-1} < u_i < u_{i+1} < u_l$, where each u_i is obtained by adding two previous elements $u_i = u_j + u_k$ with $j, k < i$ for $i > 2$. Note that j and k are not necessarily different.

III. Related Work

A large number of field exponentiation algorithms have been reported. Known strategies include: binary, m-ary, adaptive m-ary, power tree and the factor method, to mention just a few [3], [4], [5], [6], [7], [8], [9], [10], [11], [12], [13], [14], [15]. In the Binary Method, the exponent e is expanded into its binary representation, then it is scanned from left to right or from right to left by executing a predetermined algorithm. This algorithm computes field squarings and multiplications operation depending of the binary value of the scanned bits.

This Binary Method can be generalized by scanning more than one bit at a time. Hence, the Window Strategy [4] scans k bits at a time. It is based on a k-ary expansion of the exponent, where the bits of e are divided into k-bit words. The resulting words are scanned by executing consecutive squarings and multiplications according to a predefined algorithm.

The Adaptive Window Strategy (different versions can be found in [11], [13], [3], [4], [16], [17]) adjusts its method according to the specific form of the given exponent e. It divides the binary input exponent into a series of variable-length zero and nonzero digits, called windows, which are processed. In this way, the algorithm consists of two phases: exponent partitioning and the exponentiation itself. This algorithm is very useful for exponentiations with large exponents (i.e., exponents with bit length greater than 128 bits).

Most of the known methods to find short addition chains are deterministic. However, very recently two probabilistic heuristic approaches were proposed in [18], [19]. In [18], a Genetic Algorithm (GA) where addition chains are directly encoded within a chromosome was presented. One-point crossover, and a repair function were considered in that approach. It was applied to a small set of exponents, obtaining competitive results with respect to those reported by deterministic methods. The second one is an algorithm based in an Artificial Immune System [19], where only feasible addition chains are considered (based on a repair mechanism). This is done by emulating the Clonal Selection Principle where the best individuals are cloned and these clones are also mutated. This algorithm was tested in a large set of exponents, obtaining better results that all the known methods in that time.

IV. The Genetic Algorithm with Repair and Local Search mechanisms (REPLS-GA) to Find Shortest Addition Chains

The motivation of this work is two-fold:

a. The research regarding heuristic-based approaches to find optimal addition chains in the specialized literature is still scarce
b. The GA-based approach proposed in [18] was not further explored as to get more information about the performance of GAs in this specific problem.

Therefore, we present an improved GA to find the shortest addition chain, given an exponent e.

Based on the previous heuristic-based approaches [18], [19], we noted the following:

a. No repair mechanism in the initial population generation was used in the GA proposed in [18].
b. The mutation operator used in the previous GA [18] can be further used as a local-search-type operator.
c. The repair process used in [19] to generate the initial population and also in the mutation of clones is subject to improvement.
d. The use of other crossover operators has not been explored.

This analysis of the previous heuristic-based approaches led us to propose an improved GA based on three modifications: (1) An improved and more extensively used repair mechanism, (2) a new crossover operator and (3) a local-search-like mutation operator. In this Section we describe all the elements (solution encoding, fitness function design, population initialization and variation operators used) of our repair-and-local-search-based GA (REPLS-GA) by highlighting those modifications proposed in this approach.

A. Solution Encoding

In our REPLS-GA, we work only with a set of valid addition chains (invalid chains are discarded) unlike the GA proposed in [18]. Each valid chain is an individual in the population. Each individual is represented, as in [18] and [19] by the values from the addition chain directly i.e. there is a direct mapping between the addition chain and the chromosome [18], [19]. In other words, each number from the addition chain corresponds to a gene in the chromosome e.g. for the exponent $e = 79$ one possible valid addition chain could be:
$1 \rightarrow 2 \rightarrow 4 \rightarrow 6 \rightarrow 12 \rightarrow 24 \rightarrow 48 \rightarrow 50 \rightarrow 62 \rightarrow 68 \rightarrow 74 \rightarrow 78 \rightarrow 79$. This addition chain was obtained by random selection of valid elements and it is not the shortest one for $e = 79$. Therefore, its corresponding individual representation would be a chromosome with 13 genes whose alleles correspond to the addition chain values (same order). See Figure 1 with another example, now for $e = 29$. Note however that chromosomes might have different lengths.

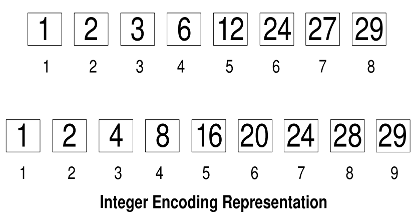

Fig. 1. Two individuals in our GA. Lengths may be different.

B. Fitness Function

The fitness value for each individual is calculated, as in previous approaches [18], [19], by the chromosome's length l minus one, e.g. for the following individual 1-2-4-6-12-24-48-50-62-68-74-78-79 its corresponding fitness value is equal to $l = 12$.

The aim of this fitness function is to favor those individuals with shorter chromosomes in the selection process and to discard those with larger encodings. In this way, regions with even shorter valid chains will be explored. Based on this definition of fitness function, we have a minimization problem.

C. Initial Population

The initial population is created in such a way that the individuals are feasible solutions, i. e., valid addition chains. In fact, our GA only works with feasible (valid) solutions.

In [19], solutions are repaired as indicated in Algorithm 1 and explained as follows:

The first two elements within any valid addition chain U are the number 1 followed by number 2, that is, $u_1 = 1$ and $u_2 = 2$. In this way, each next element u_i (from $i = 3$ to $i = l$ with $u_l = e$) is computed sequentially, being e the exponent we want to reach. Each u_i step is computed in three different ways: The first one is by applying the double stepping, that is $u_i = 2u_{i-1}$. The second way consists on adding the two previous elements $u_i = u_{i-1} + u_{i-2}$, and the third option just adds the last element plus a randomly selected element $u_i = u_{i-1} + rnd(0, i-1)$ where $rnd(A, B)$ returns an integer number into the range (A, B), generated with a uniform distribution. Note that $u_{i-1} < u_i \leq e$. Each of these three strategies are selected based on two parameters called f and g. It is worth noticing that the following situations are avoided because they generate invalid individuals: (1) $u_i > e$ and (2) $u_i \leq u_{i-1}$.

In this work, we propose to modify the third option, which in its original version consists on a loop where random values are generated until a valid value is found. Instead, we propose to just generate one random value. If this value is invalid, then, starting from this invalid random value we perform a sequential search down to the first element until the next value is valid. The expected behavior is to keep the algorithm from wasting too much time looking for a valid value generated by random values (especially complicated for large exponents). An example of this modification is presented in Figure 2.

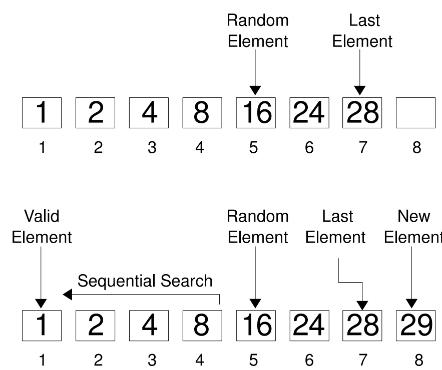

Fig. 2. Sequential Search (No loop)

Moreover, we propose, as suggested in [19], to use this modified repair method in the generation of the initial population. The pseudocode of this process is depicted in Algorithm 2.

D. Crossover Operator

In this work we extend the one-point crossover used in [18] to a two-point and uniform crossover operators. We tested both operators. Based on several experiments performed (not detailed due to space restrictions) we observed that the two-point crossover performed better with respect to the uniform crossover. Therefore, we focus on detailing two-point crossover in the remaining of the document.

In this operator, the interchanging of genes between the two parents is performed in same manner as in the standard two-point crossover. The applied modifications are

Function 1 $Fill(U, i, e)$

Input: An Incomplete Addition Chain $U = u_1, u_2, ..., u_l = e$, where i represents the next position to be filled.
Output: A feasible addition chain for Exponent e, with length l

Set $m = i - 1$
{/*U_m defines the last element of the addition chain*/}
while $U_m \neq e$ **do**
  if $flip(f)$ **then**
    {/*Applying Double Stepping(DS)*/}
    $U_m = 2U_{m-1}$
  else if $flip(g)$ **then**
    {/*Selecting two last elements*/}
    $U_m = U_{m-1} + U_{m-2}$
  else
    {/*Selecting a random element*/}
    $U_m = U_{m-1} + rnd(U_0, U_{m-1})$
  end if
  while $U_m > e$ **do**
    Look for a new feasible element, starting a sequential search from last random element selected to first element if necessary.
  end while
  $m = m + 1$
end while

Function 2 $InitialPopulation$

Input: Exponent e
Output: A valid addition chain $U = u_1, u_2, ..., u_l = e$ with length l

Set $u_1 = 1$ and $u_2 = 2$
Assign $u_3 = rnd(u_1, u_2)$ where rnd() returns a random integer in the interval
Generate a complete Addition Chain, $(U, l) = fill(U, 4, e)$

the following: The values into the parents' chromosome segments *before* the first crossover point p are copied into the corresponding offspring segments.

For the chromosome segments *after* the first (p) and second (q) crossover points, the operator copies the rules (instead of the values) from the parents' segments into the offspring chromosomes (as illustrated in Figure 3). If it is not possible to apply the aforementioned rules because the result of the addition could be greater than the exponent e (this of course would create an invalid addition chain), then the fill function is used to complete the chromosome. The last segments of the offspring are copied in the same way of the segments before the first crossover point. A detailed pseudocode of the modified crossover operator used is presented in Algorithm 3.

Figure 3 shows an example that combines two individuals representation for $e = 29$. As said, $C1$ will copy the values of $P1$, starting from first element $i = 1$ until $i = p - 1$ is reached. For the next element ($i = 4$), $C1$ will consider just the rules from $P2$, in this case, $P2_4$ is generated by a Double Stepping applied to last element ($2{*}P2_3 = 6$), therefore, $C1_4$ will be conceived as $2{*}C1_3 = 8$, the same process is adopted to generate next elements of the addition chain, but changing

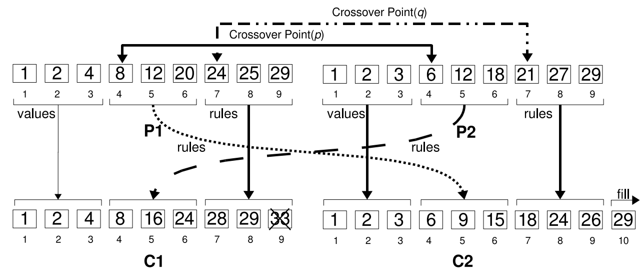

Fig. 3. Two-points crossover operator

the inheritance rules from $P2$ to $P1$ when q is found. Notice that for $C1_8$ the exponent is completely reached, obtaining a shorter addition chain than the ones represented by both parents. In this case, the generation process is stopped and a new valid individual is considered. For $C2$, the same process takes place (as done with $C1$), but applying *fill* function to repair individual due to invalid state ending $C2_9 = 26$, which is not reaching the given exponent. Final representation of $C2$, shows that its length is greater than $P1$, $P2$ and even $C1$.

E. Mutation Operator

The mutation applied in REPLS-GA is a modification of that proposed in [18] with the aim to use it as a local search operator. The main difference is that we propose to generate N mutant individuals instead of just one [18]. It is implemented as follows: One mutation point is randomly selected in each offspring and the segment before the mutation point is copied N times. Then, the function *fill* (defined in Algorithm 1) is utilized to complete the N individuals. Only the best new mutated child will be allowed to survive, replacing the original child.

An example is illustrated in Figure 4 for the exponent $e = 79$, where a mutation point (the position with value 28) is selected into the original child (a), then all the steps from 1 to 24 are copied $N = 4$ times. The remaining steps are computed by using the function *fill* (b). The best resulting mutated child is the first one, meanwhile the worst is the third one. Only the best child will be allowed to survive by replacing the original one. The pseudocode of the mutation operator is shown in Algorithm 4.

F. Complete REPLS-GA

The complete REPLS-GA pseudocode is shown in Algorithm 5. It receives an exponent *e* and returns a quasi optimal addition chain.

V. Experiments

The experiments to test our REPLS-GA were designed with the aim to: (1) show that its performance is competitive (or even better) with those provided by other heuristic-based approaches and (2) to solve even more complex instances of the problem. In this way, two experiments were performed. The difference between experiments is the value of the exponent used, which determines the complexity of the problem.

Function 3 $crossover(P1, P2)$

Input: Parents $P1$ and $P2$ to be reproduced
Output: Two new children, $C1$ and $C2$

```
if flip(P_c) then
  Set a common length l for both Parents (P1, P2)
  {/*Choose first crossover point*/}
  Set p = rnd(2, l/2)
  {/*Choose second crossover point*/}
  Set q = rnd((l/2) + 1, l - 1)
  {/*Generate C1 elements by using rules from P1*/}
  for k = 1 to p do
    C1_k = P1_k
  end for
  {/*Generate C1 elements by using rules from P2*/}
  Set k = p + 1
  while k <= q do
    Look for positions that define the rule for actual element,
    such that P2_k = P2_a + P2_b
    C1_k = C1_a + C1_b
  end while
  {/*Generate C1 elements by using rules from P1*/}
  Set k = q + 1
  {/*C1 will attempt to have the same length as P1*/}
  while k <= l(P1) do
    Look for positions that define the rule for actual element,
    such that P1_k = P1_a + P1_b
    C1_k = C1_a + C1_b
  end while
  If C1 is not a valid addition chain, apply repair process to C1
  {/*Generate C2 elements by using rules from P2*/}
  for k = 1 to p do
    C2_k = P2_k
  end for
  {/*Generate C2 elements by using rules from P1*/}
  Set k = p + 1
  while k <= q do
    Look for positions that define the rule for actual element,
    such that P1_k = P1_a + P1_b
    C2_k = C2_a + C2_b
  end while
  {/*Generate C2 elements by using rules from P2*/}
  Set k = q + 1
  {/*C1 will attempt to have the same length as P2*/}
  while k <= l(P2) do
    Look for positions that define the rule for actual element,
    such that P2_k = P2_a + P2_b
    C2_k = C2_a + C2_b
  end while
  If C2 is not a valid addition chain, apply repair process to C2
else
  {/*C1 will become a copy of P1*/}
  for k = 1 to l(P1) do
    C1_k = P1_k
  end for
  {/*C2 will become a copy of P2*/}
  for k = 1 to l(P2) do
    C2_k = P2_k
  end for
end if
```

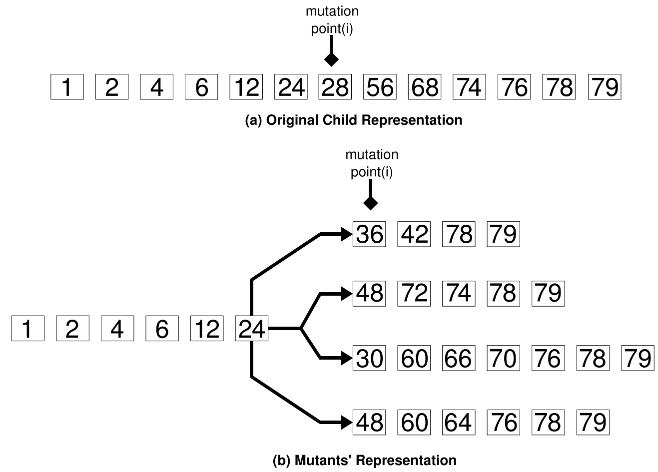

Fig. 4. Mutation Operator with Local Search

Function 4 $mutation(C)$

Input: A Child Individual $C = (U, l)$.
Output: The Best Mutated Child Version $C_m = (U_m, l)$

```
if flip(P_m) then
  Generate N mutants of the Child Individual
  {/*Select a common limit to generate next mutation point*/}
  i = rnd(3, l(C))
  for k = 1 to N do
    Select a random position C_{k_j} where j = rnd(0, i - 1) is a
    random element of the actual clone.
    Generate a new element for C_{k_{i+1}} (mutation point), such
    that C_{k_{i+1}} = C_{k_i} + C_{k_j}
    Repair addition chain starting from new generated element
    C_k = fill(C_k, i + 1, e)
  end for
  Choose the best mutation version.
end if
```

Quality (the best solution reached so far) and consistency (better mean, standard, deviation and/or worst values) of the obtained results are considered in the discussion.

All the experiments performed in this work, were obtained by using the following parameter values:

- Population Size ($POPSIZE$) = 200
- Maximum number of generations ($MAXGEN$) = 300
- Two-point crossover (as explained in Section IV-D).
- Local-search mutation (as explained in Section IV-E)
- Crossover Rate (P_c) = 0.4%
- Mutation Rate (P_m) = 1.0%
- Double Stepping Rate ($U_i = 2U_{i-1}$ used in *fill*) (f) = 0.7
- Probability of selecting $U_i = U_{i-1} + U_{i-2}$ in *fill* function, (g) = 0.2
- (N) utilized in mutation operator = 4
- Binary Tournament Selection

In the experiments with the following exponents: $e = 573, 734, 749$, and 764 it was necessary to change the value of $g = 0.7$.

All these parameters values were obtained by a trial-and-

Function 5 $GA\_Optimal\_Addition\_Chain(e)$

```
Require: POPSIZE is odd
  Input: An exponent e
  Output: A quasi optimal addition chain U = u_1, u_2, ..., u_l = e

  {/*Setting up initial population*/}
  for i = 1 to POPSIZE do
    Parents_i = InitialPopulation(e)
  end for
  for i = 1 to MAXGEN do
    Randomize parents population
    k = 0
    while k < (POPSIZE - 1) do
      {/*Select a couple of parents (P_1, P_2)*/}
      P_1 = selection(T)
      P_2 = selection(T)
      {/*Generate two new children (C_k, C_{k+1}), by applying
      crossover operator*/}
      crossover(P_1, P_2)
      {/*Apply the mutation operator to each new created child
      */}
      mutation(C_k)
      mutation(C_{k+1})
      Set k = k + 2
    end while
    Replace parents population with children population
  end for
  Report fittest individual
```

error process by favoring the best overall performance.

The first set of experiments consisted on computing the total accumulated addition chains for a fixed set of exponents. An accumulated addition chain for a maximum value N, represents the sum of all addition chains obtained for all the exponents $1, 2, \dots, N$, as stated in 2.

An accumulated addition chain for a given exponent e, represents the sum of all addition chains obtained for each number in the sequence defined by e, as stated in 2.

$$AAC(N) = \sum_{i=1}^{N} GA\_Optimal\_Addition\_Chain(i) \quad (2)$$

The smaller the addition chain for each exponent in the sequence is, the shorter the total accumulated addition chain is. Hence, while obtaining values closer to the optimal ones, the algorithm will be demonstrating its search capability.

A set of accumulated addition chains for all exponents less than: 512 ($e \in [1, 512]$), 1000 ($e \in [1, 1000]$), 2000 ($e \in [1, 2000]$), 2048 ($e \in [1, 2048]$) and 4096 ($e \in [1, 4096]$), is presented in order to compare our results with respect to those obtained by the Artificial Immune System presented in [19], and a previous GA presented in [18].

Regarding this first experiment, in Table I, the optimal value and best results obtained by each heuristic are reported, including our REPLS-GA.

In Table II, we present the statistical results obtained by REPLS-GA in 30 independent runs for all exponent sets considered in the first experiment.

The second experiment comprised a family of exponents that are considered relatively hard to optimize [20]. In general, a given exponent is hard or complex to optimize when its shortest addition chain cannot be obtained by traditional deterministic methods, such as the binary method or the adaptive window method, or other non-deterministic heuristics. The main objective of this experiment was to assess the search capability of our REPLS-GA over other deterministic and heuristic-based approaches when dealing with difficult exponents.

TABLE I

COMPARISON OF BEST RESULTS OBTAINED BY THE GA [18], AIS [19] AND THIS REPLS-GA.

	$e \in [1, 512]$	$e \in [1, 1000]$	$e \in [1, 2000]$	$e \in [1, 2048]$	$e \in [1, 4096]$
Optimal	4924	10808	24063	24731	54425
GA	4925	10818	24124	-	-
AIS	4924	10813	24108	24778	54617
REPLS-GA	4924	10809	24076	24748	54487

TABLE II

REPLS-GA STATISTICAL RESULTS

	Best	Average	Median	Worst	Std. Dev.
$e \in$ [1,512]	4924	4924.03	4924	4925	0.180
$e \in$ [1,1000]	10809	10810.88	10811	10815	1.431
$e \in$ [1,2000]	24076	24083.29	24084	24091	3.415
$e \in$ [1,2048]	24748	24753.02	24753	24761	2.966
$e \in$ [1,4096]	54487	54499	54499	54510	6.215

For all these exponents, their shortest addition chain have a length equal to 27. In Table III and IV, we show these exponents with the addition chain found by our REPLS-GA.

A. Discussion of Results

The first experiment provided the following findings: Based on the results presented in Table I, REPLS-GA clearly outperforms both heuristics (GA and AIS), mostly in more complex exponents (columns 5 and 6 in Table I). Our REPLS-GA obtains values very close to the optimal solutions for all the sets of exponents. Notice that, REPLS-GA is close to compute optimal values in most cases, having a percentage error of 0.009% for ($e \in [1, 512]$) and 0.11% for ($e \in [1, 4096]$), minimizing the values reported in [19] by 0.061% and 0.29% for 512 and 4096, respectively.

From the statistical results shown in Table II we observe the following: The worst accumulated addition chains found by REPLS-GA for ($e \in [1, 2000]$), ($e \in [1, 2048]$) and ($e \in [1, 4096]$) (Column 5) are better than the best ones obtained by the AIS [19] for each one of these exponent sets (row 4, columns 3, 4 and 5 in Table I).

The results of the second experiment ("hard" exponents) whose results were presented in Tables III and IV, showed the REPLS-GA is capable of finding all the optimal results on this instances of the problem. No results on these problems were found in [18], [19] as to perform a comparison. It is also important to mention that both compared approaches, AIS [19]and GA [18], obtained better results than any deterministic approaches such as binary and window strategies (see Section III and [18], [19] for details).

The overall findings suggest that the modified *fill* function (not used in the previous GA [18] to generate the initial population), and the modified variation operators (two-point crossover and local-search mutation) help to improve the quality of the results obtained.

VI. Conclusions and Future Work

In this paper we present three modifications to a GA-based approach to find the shortest addition chain for a given exponent e: (1) A repair mechanism usually adopted in the variation operators (crossover and mutation) was improved and its use was extended to the generation of the initial population, (2) A two-point crossover operator was considered and (3) a local-search-like mutation operator was used to further explore the neighborhood of solutions generated by the crossover operator. REPLS-GA was compared against two state-of-the-art heuristic-based approaches in simple instances of the problem. REPLS-GA outperformed both techniques mostly in large exponent values ($e \in [1, 2000]$), ($e \in [1, 2048]$) and ($e \in [1, 4096]$). Furthermore, REPLS-GA was able to reach the optimal chain length on exponent values where no other heuristic-based approach has reported results in the specialized literature. Our future work includes striving with other evolutionary algorithms and to design some control mechanisms for the parameters used in the *fill* function. Finally, we will test REPLS-GA with larger exponents (e.g. 128 bits).

Acknowledgment

The authors would like to thank Neill Clift for given us the set of exponents used in the second experiment. The second author acknowledges support from CONACyT through project 79809. The third author acknowledges support from SIP-IPN through project 20091712. The fourth author acknowledges support from CONACyT through project 60240.

References

[1] IEEE P1363, *Standard specifications for public-key cryptography, Draft Version D18*. "http://grouper.ieee.org/groups/1363/": IEEE standards documents, November 2004.

[2] ANSI X9.17 (Revised), National Standards for financial institution key management (wholesale), American Bankers Association, 1986.

[3] D. M. Gordon, "A survey of fast exponentiation methods," *Journal of Algorithms*, vol. 27, no. 1, pp. 129–146, April 1998.

[4] D. E. Knuth, *Art of Computer Programming, Seminumerical Algorithms*. Addison-Wesley Professional, November 1997, vol. 2.

[5] A. J. Menezes, P. C. van Oorschot, and S. A.Vanstone, *Handbook of Applied Cryptography*. Boca Raton, Florida: CRC Press, 1996.

[6] N. Kunihiro and H. Yamamoto, "New methods for generating short addition chains," *IEICE Trans. Fundamentals*, vol. E83-A(1), pp. 60–67, January 2000.

[7] Ç. K. Koç, "Analysis of sliding window techniques for exponentiation," *Computer and Mathematics with Applications*, vol. 30(10), pp. 17–24, October 1995.

[8] F. Bergeron, J. Berstel, and S. Brlek, "Efficient computation of addition chains," *Journal de thorie des nombres de Bordeaux*, vol. 6, pp. 21–38, 1994.

[9] Ç. K. Koç, "High-Speed RSA Implementation," RSA Laboratories, Redwood City, CA, Tech. Rep. TR 201, 71 pages, 1994.

[10] N. Kunihiro and H. Yamamoto, "Window and extended window methods for addition chain and addition-subtraction chain," *IEICE Trans. Fundamentals*, vol. E81-A(1), pp. 72–81, January 1998.

[11] J. Bos and M. Coster, "Addition chain heuristics," *In G. Brassard, (editor)* Advances in Cryptology —CRYPTO 89 *Lecture Notes in Computer Science*, vol. 435, pp. 400–407, 1989.

[12] Y. Yacobi, "Exponentiating faster with addition chains," *In I. B. Damgard, (editor)* Advances in Cryptology —EUROCRYPT 90 *Lecture Notes in Computer Science*, vol. 473, pp. 222–229, 1990.

[13] E. F. Brickell, D. M. Gordon, K. S. McCurley, and D. B. Wilson, "Fast exponentiation with precomputation," *In R. A. Rueppel, (editor)* Advances in Cryptology —EUROCRYPT 92 *Lecture Notes in Computer Science*, vol. 658, pp. 200–207, 1992.

[14] J. von zur Gathen and M. Nöcker, "Computing special powers in finite fields: extended abstract," in *Proceedings of the 1999 international symposium on Symbolic and algebraic computation*. ACM Press, 1999, pp. 83–90.

[15] N. Cruz-Cortes, F. Rodriguez-Henriquez, and C. C. Coello, "On the optimal computation of finite field exponentiation," *In C Lematre, C. Reyes, J. Gonzlez, (editors)* Advances in Artificial Intelligence - IBERAMIA 2004: 9th Ibero-American Conference on AI *Lecture Notes in Computer Science*, vol. 3315, pp. 747–756, November 2004.

[16] C. K. Koc, "Analysis of sliding window techniques for exponentiation," *Computer and Mathematics with Applications*, vol. 30, no. 10, pp. 17–24, October 1995.

[17] N. Kunihiro and H. Yamamoto, "New methods for generating short addition chains," *IEICE Trans. Fundamentals*, vol. E83-A, no. 1, pp. 60–67, January 2000.

[18] N. Cruz-Cortés, F. Rodríguez-Henríquez, R. Juárez-Morales, and C. A. C. Coello, "Finding Optimal Addition Chains Using a Genetic Algorithm Approach," in *Computational Intelligence and Security, International Conference, CIS 2005, Proceedings*, ser. Lecture Notes in Computer Science, Y. Hao, J. Liu, Y. Wang, Y. Cheung, H. Yin, L. Jiao, J. Ma, and Y. Jiao, Eds., vol. 3801. Springer, 2005, pp. 208–215.

[19] N. Cruz-Cortés, F. Rodríguez-Henríquez, and C. A. C. Coello, "An Artificial Immune System Heuristic for Generating Short Addition Chains," *IEEE Transactions on Evolutionary Computation*, vol. 12, no. 1, pp. 1–24, February 2008.

[20] N. Clift, *Shortest Addition Chains*, 2009, available at: http://wwwhomes.uni-bielefeld.de/achim/addition_chain.html. Last update: 2009-02-02.

TABLE III

SHORTEST ADDITION CHAINS FOR A SPECIAL CLASS OF EXPONENTS (TABLE 1 OF 2)

Exponent	Addition Chain	Length
3243679	$1 \to 2 \to 4 \to 5 \to 7 \to 14 \to 28 \to 56 \to 112 \to 224 \to 448 \to 453 \to 906 \to 1359 \to 1583 \to 3166 \to 6332 \to 12664 \to 25328 \to 50656 \to 101312 \to 202624 \to 405248 \to 810496 \to 1620992 \to 3241984 \to 3243567 \to 3243679$	27
3493799	$1 \to 2 \to 3 \to 5 \to 8 \to 13 \to 26 \to 52 \to 104 \to 106 \to 212 \to 424 \to 848 \to 1696 \to 3392 \to 6784 \to 13568 \to 27136 \to 54272 \to 108544 \to 217088 \to 434176 \to 868352 \to 875136 \to 875149 \to 1750298 \to 2625447 \to 3493799$	27
3459835	$1 \to 2 \to 3 \to 5 \to 10 \to 20 \to 40 \to 60 \to 70 \to 140 \to 280 \to 283 \to 563 \to 1126 \to 2252 \to 4504 \to 9008 \to 18016 \to 36032 \to 72064 \to 72074 \to 144148 \to 288296 \to 576592 \to 1153184 \to 2306368 \to 3459552 \to 3459835$	27
3235007	$1 \to 2 \to 4 \to 5 \to 9 \to 18 \to 27 \to 54 \to 81 \to 162 \to 324 \to 648 \to 972 \to 1944 \to 3888 \to 7776 \to 15552 \to 31104 \to 62208 \to 124416 \to 248832 \to 248859 \to 497691 \to 995382 \to 1990764 \to 2986146 \to 3235005 \to 3235007$	27
3230591	$1 \to 2 \to 3 \to 6 \to 12 \to 24 \to 48 \to 96 \to 192 \to 194 \to 388 \to 776 \to 1552 \to 3104 \to 6208 \to 12416 \to 12419 \to 24838 \to 49676 \to 49679 \to 99355 \to 198710 \to 397420 \to 794840 \to 1589680 \to 3179360 \to 3229039 \to 3230591$	27
3182555	$1 \to 2 \to 4 \to 5 \to 10 \to 15 \to 30 \to 60 \to 120 \to 240 \to 480 \to 960 \to 975 \to 1950 \to 3900 \to 7800 \to 15600 \to 31200 \to 62400 \to 62405 \to 124805 \to 187210 \to 312015 \to 624030 \to 1248060 \to 2496120 \to 3120150 \to 3182555$	27
3440623	$1 \to 2 \to 3 \to 6 \to 12 \to 13 \to 26 \to 52 \to 104 \to 208 \to 416 \to 419 \to 835 \to 1670 \to 3340 \to 3366 \to 6706 \to 13412 \to 26824 \to 53648 \to 107296 \to 214592 \to 429184 \to 858368 \to 1716736 \to 1720102 \to 3440204 \to 3440623$	27
3926651	$1 \to 2 \to 4 \to 5 \to 9 \to 18 \to 36 \to 72 \to 144 \to 288 \to 576 \to 1152 \to 2304 \to 4608 \to 9216 \to 18432 \to 18437 \to 36869 \to 73738 \to 92175 \to 184350 \to 368700 \to 737400 \to 811138 \to 1548538 \to 3097076 \to 3908214 \to 3926651$	27
3234263	$1 \to 2 \to 3 \to 4 \to 8 \to 16 \to 32 \to 64 \to 128 \to 131 \to 262 \to 263 \to 525 \to 1050 \to 2100 \to 4200 \to 8400 \to 16800 \to 33600 \to 67200 \to 100800 \to 201600 \to 202125 \to 404250 \to 808500 \to 1617000 \to 3234000 \to 3234263$	27
3352927	$1 \to 2 \to 3 \to 6 \to 12 \to 24 \to 48 \to 96 \to 97 \to 194 \to 291 \to 582 \to 1164 \to 1746 \to 2910 \to 5820 \to 11640 \to 23280 \to 46560 \to 69840 \to 139680 \to 279360 \to 558720 \to 1117440 \to 2234880 \to 3352320 \to 3352902 \to 3352926 \to 3352927$	27

TABLE IV

SHORTEST ADDITION CHAINS FOR A SPECIAL CLASS OF EXPONENTS (TABLE 2 OF 2)

Exponent	Addition Chain	Length
3704431	$1 \to 2 \to 4 \to 5 \to 9 \to 18 \to 36 \to 72 \to 144 \to 288 \to 576 \to 581 \to 1157 \to 2314 \to 4628 \to 9256 \to 18512 \to 37024 \to 74048 \to 148096 \to 296192 \to 592384 \to 1184768 \to 1185349 \to 2370698 \to 3556047 \to 3704143 \to 3704431$	27
3922763	$1 \to 2 \to 3 \to 6 \to 12 \to 24 \to 26 \to 52 \to 104 \to 208 \to 416 \to 832 \to 1664 \to 3328 \to 3331 \to 6659 \to 9990 \to 19980 \to 39960 \to 79920 \to 159840 \to 163171 \to 326342 \to 652684 \to 1305368 \to 1958052 \to 3916104 \to 3922763$	27
2948207	$1 \to 2 \to 3 \to 4 \to 7 \to 14 \to 28 \to 29 \to 58 \to 116 \to 232 \to 239 \to 478 \to 956 \to 1912 \to 3824 \to 3853 \to 7677 \to 15354 \to 30708 \to 61416 \to 122832 \to 245664 \to 491328 \to 982656 \to 1965312 \to 2947968 \to 2948207$	27
3093839	$1 \to 2 \to 3 \to 5 \to 10 \to 20 \to 30 \to 60 \to 120 \to 150 \to 151 \to 302 \to 604 \to 1208 \to 2416 \to 4832 \to 9664 \to 19328 \to 38656 \to 77312 \to 154624 \to 309248 \to 618496 \to 1236992 \to 2473984 \to 3092480 \to 3093688 \to 3093839$	27
3243931	$1 \to 2 \to 4 \to 8 \to 16 \to 32 \to 64 \to 128 \to 256 \to 258 \to 514 \to 515 \to 1029 \to 2058 \to 4116 \to 8232 \to 16464 \to 32928 \to 65856 \to 66371 \to 132227 \to 198083 \to 396166 \to 792332 \to 1584664 \to 3169328 \to 3235699 \to 3243931$	27
3325439	$1 \to 2 \to 4 \to 8 \to 16 \to 17 \to 33 \to 66 \to 132 \to 264 \to 528 \to 1056 \to 2112 \to 4224 \to 4241 \to 8482 \to 16964 \to 33928 \to 67856 \to 135712 \to 271424 \to 271457 \to 542914 \to 1085828 \to 1085861 \to 2171722 \to 3257583 \to 3325439$	27
3190511	$1 \to 2 \to 3 \to 6 \to 12 \to 24 \to 48 \to 96 \to 192 \to 384 \to 768 \to 1536 \to 3072 \to 6144 \to 6147 \to 12294 \to 24588 \to 24684 \to 24685 \to 49370 \to 98740 \to 197480 \to 394960 \to 789920 \to 1579840 \to 3159680 \to 3184364 \to 3190511$	27
3287999	$1 \to 2 \to 3 \to 6 \to 12 \to 24 \to 48 \to 96 \to 192 \to 193 \to 386 \to 772 \to 1544 \to 3088 \to 6176 \to 12352 \to 24704 \to 24897 \to 49794 \to 99588 \to 99591 \to 199179 \to 398358 \to 796716 \to 1593432 \to 3186864 \to 3286455 \to 3287999$	27
3266239	$1 \to 2 \to 4 \to 8 \to 16 \to 32 \to 64 \to 128 \to 256 \to 512 \to 1024 \to 2048 \to 4096 \to 4098 \to 8196 \to 16392 \to 16393 \to 32785 \to 49178 \to 98356 \to 196712 \to 393424 \to 786848 \to 1573696 \to 3147392 \to 3245748 \to 3262141 \to 3266239$	27
3167711	$1 \to 2 \to 3 \to 6 \to 12 \to 24 \to 48 \to 96 \to 192 \to 384 \to 768 \to 1536 \to 1539 \to 3078 \to 6156 \to 12312 \to 12313 \to 24626 \to 49252 \to 98504 \to 197008 \to 394016 \to 788032 \to 1576064 \to 3152128 \to 3164441 \to 3167519 \to 3167711$	27