Journal of Heuristics manuscript No.
(will be inserted by the editor)

Finding Short and Hardware-friendly Addition Chains with Evolutionary Algorithms

Stjepan Picek · Carlos A. Coello Coello · Domagoj Jakobovic · Nele Mentens

Received: date / Accepted: date

Abstract Finding the shortest addition chain for a given exponent is a significant problem in cryptography. In this work, we present a genetic algorithm with a novel encoding of solutions and new crossover and mutation operators to minimize the length of the addition chains corresponding to a given exponent. We also develop a repair strategy that significantly enhances the performance of our approach. The results are compared with respect to those generated by other metaheuristics for exponents of moderate size, but we also investigate values up to $2^{255} - 21$. For numbers of such size, we were unable to find any results produced by other metaheuristics which could be used for comparison purposes. So, we decided to add three three additional strategies to serve as benchmarks. Our results indicate that the proposed approach is a very promising alternative to deal with this problem. We also consider a more practical perspective by taking into account the implementation cost of the chains where we optimize the addition chains with regards to the type of operations as well as the number of instructions required for the implementation.

Keywords Addition chains · Genetic algorithms · Cryptography · Optimization · Implementation

Stjepan Picek
KU Leuven, ESAT/COSIC and imec, Kasteelpark Arenberg 10, bus 2452, B-3001 Leuven-Heverlee, Belgium E-mail: stjepan@computer.org

Carlos A. Coello Coello
CINVESTAV-IPN, Department of Computer Science, Av. IPN No. 2508, Col. San Pedro Zacatenco, Mexico, D.F. 07360, MEXICO E-mail: ccoello@cs.cinvestav.mx

Domagoj Jakobovic
University of Zagreb, Faculty of Electrical Engineering and Computing, Croatia E-mail: domagoj.jakobovic@fer.hr

Nele Mentens
KU Leuven, ESAT/COSIC and imec, Kasteelpark Arenberg 10, bus 2452, B-3001 Leuven-Heverlee, Belgium E-mail: Nele.Mentens@kuleuven.be

1 Introduction

Field or modular exponentiation has several important applications in error-correcting codes and cryptography. Well-known public-key cryptosystems such as Rivest-Shamir-Adleman (RSA) [**?**] adopt modular exponentiation. However, those operations are often the most expensive ones in cryptosystems and naturally one aims to make them as efficient as possible. In a simplified way, modular exponentiation can be defined as the problem of finding the (unique) integer $B \in [1, \ldots, p-1]$ that satisfies:

$$B = A^c \bmod p, \tag{1}$$

where A is an integer in the range $[1, \ldots, p-1]$, c is an arbitrary positive integer, and p is a large prime number. One possible way of reducing the computational load of Eq. (1) is to minimize the total number of multiplications required to compute the exponentiation.

Since the exponent in Eq. (1) is additive, the problem of computing powers of the base element A can be also formulated as an addition calculation, for which the so-called *addition chains* are used. Informally speaking, an addition chain for the exponent c of length l is a sequence V of positive integers $v_0 = 1, \ldots, v_l = c$, such that for each $i > 1$, $v_i = v_j + v_k$ for some j and k with $0 \leq j \leq k < i$. An addition chain provides the correct sequence of multiplications required for performing an exponentiation. Thus, given an addition chain V that computes the exponent c as indicated before, we can find $B = A^c$ by successively computing: $A, A^{v_1}, \ldots, A^{v_{l-1}}, A^c$.

As an example, consider A^{60}, where the naive procedure would require 59 $(c-1)$ multiplications. One simple algorithm that can be used (although, it will be often the case that it does not give optimal results) works in the following way. First, write the exponent in its binary representation. Then, replace each occurrence of the digit 1 with the letters "DA" and each occurrence of the digit 0 with the letter "D". After all digits are replaced, remove the first "DA" that appears on the left. What remains represents a rule to calculate the exponent, since the letter "A" stands for addition (multiplication) and the letter "D" for doubling (squaring). If we consider again the example A^{60}, the exponent 60 in binary representation equals "111100". After the replacement and the removal of "DA" at the left, it remains the "DADADADD" sequence. Thus, the rule is: square, multiply, square, multiply, square, multiply, square, square $(1 \to 2 \to 3 \to 6 \to 7 \to 14 \to 15 \to 30 \to 60)$.

This simple example describes the so-called *the binary* or *square-and-multiply* method. However, this method does not always result in the shortest chain (cf. with the chain given in Eq. eq2). In fact, even for the value 15, the binary method will not produce the shortest chain [**?**]. Still, it can be generalized to some more powerful methods such as those presented in Section 2. Another option is to use the addition chain $[1 \to 2 \to 4 \to 6 \to 12 \to 24 \to 30 \to 60]$,

for which we see that only seven multiplications are required:

$$A^1; A^2 = A^1A^1; A^4 = A^2A^2; A^6 = A^4A^2; A^{12} = A^6A^6;$$
$$A^{24} = A^{12}A^{12}; A^{30} = A^{24}A^6; A^{60} = A^{30}A^{30}. \quad (2)$$

Thus, the length of the addition chain defines the number of multiplications required for computing the exponentiation. The aim is to find the shortest addition chain for a given exponent c (many addition chains can be produced for the same exponent and a number of them can have the same length). Naturally, as the exponent value grows, it becomes more difficult to find a chain that forms the exponent in a minimal number of steps. Moreover, there exists an argument that finding the shortest addition chain is an **NP**-complete problem [**?**]. One possible way of tackling difficult problems is to use metaheuristics. To that end, we propose a genetic algorithm to find short addition chains for a given exponent.

This work is based on the paper "Evolutionary Algorithms for Finding Short Addition Chains: Going the Distance" [**?**]. We use the same algorithm as given in [**?**] but here we optimize it in order to be able to use it on even larger exponent values. The source code of the evolutionary algorithms used here is available as a part of the ECF framework [**?**]. Next, in this paper we present new results for a number of random values in order to test our algorithm in the case when there is no perceived structure in the exponent value. We also conduct tests for values that consist of a relatively large number of small steps which constitutes them as difficult values to find shortest addition chains. Besides the experiments for the $2^{127} - 3$ value, we add an additional real-world case – the value $2^{255} - 21$ where we again run extensive experiments. Finally, in this paper we also consider an implementation perspective and we try to evolve addition chains that are computationally inexpensive when implemented in hardware.

The remainder of this paper is organized as follows. Section 2 provides some background information on addition chains, as well as on possible chain elements, and different types of chains. Furthermore, we discuss several techniques for exponentiation, relevant from a cryptographic perspective. In Section 3 we provide an overview of related work in which heuristics have been used to find short chains. Section 4 presents our design goals as well as the algorithm that we propose. In Section 5, we report extensive results for various test cases and exponent sizes. Following that, in Section 6, we present two important modifications of the problem where we do not only consider finding the shortest chains, but also finding chains that are "cheap" for implementation in hardware. In Section 7, we give a discussion about the results we obtained as well as some possible future research directions. Finally, in Section 8, we conclude the paper. An example of the code listing all necessary instructions for a chain of interest is given in Appendix A.

2 On Addition Chains

We start this section with some basic notions about addition chains. Afterwards, we give several important results that we use when designing our evolutionary algorithm. Next, we briefly discuss algorithms that are commonly used to compute exponentiations. In this work, we follow the notation and theoretical results presented in "The Art of Computer Programming, Volume 2: Seminumerical Algorithms" [**?**]. For more detailed information about addition chains, we refer readers to Chapter 4.6.3. "Evaluation of Powers" [**?**].

Let n be the exponent value and $\nu(n)$ be the number of ones in the binary representation of that exponent, i.e., $\nu(n)$ represents the Hamming weight of a number n. The number of bits necessary to represent the exponent (integer) value n is denoted as $\lambda(n) + 1$, where $\lambda(n) = \lfloor log_2(n) \rfloor$.

2.1 Theoretical Background

Definition 1 *An addition chain is a sequence* $a_0 = 1, a_1, ..., a_r = n$ *where:*

$$a_i = a_j + a_k, \text{ for some } k \leq j < i. \tag{3}$$

Definition 2 *An addition chain is called ascending if:*

$$1 = a_0 < a_1 < a_2 < ... < a_r = n. \tag{4}$$

In this work, we focus only on ascending chains. From this point on, when we talk about addition chains, we consider ascending addition chains. The shortest length of any valid addition chain for a value n is denoted as $l(n)$. In the length of a chain, the initial step that has the value one is not counted.

Next, it is possible to define types of steps in the addition chain based on Eq. (3):

- *Doubling step*; when $j = k = i - 1$. This step always gives the maximal possible value at the position i.
- *Star step*; when j but not necessarily k equals $i - 1$.
- *Small step*; when $\lambda(a_i) = \lambda(a_{i-1})$.
- *Standard step*; when $a_i = a_j + a_k$ where $i > j > k$.

On the basis of the aforementioned steps, it is easy to infer the following conclusions [**?**]:

- The first step is always a doubling step.
- A doubling step is always a star step and never a small step.
- A doubling step must be followed by a star step.
- If step i is not a small step, then step $i + 1$ is either a small step or a star step, or both.

Now, we focus on the shortest addition chains. Trivially, the shortest chain for any number n must have at least $log_2(n)$ steps. To be more precise, any chain length is equal to $log_2(n)$ plus the number of small steps [**?**].

When $\nu(n) \geq 9$ then there are at least four small steps in any chain for exponent length n [**?**]. That statement can be also generalized with the following definition [**?**]:

Definition 3 *If* $\nu(n) \geq 2^{4 \cdot m - 1} + 1$*, then* $l(n) \geq log_2(n) + m + 3$ *where* m *is a nonnegative value.*

A star chain is a chain that involves only star operations. The minimal length of a star chain is denoted as $l^*(n)$ and the following holds [**?**]:

$$l(n) \leq l^*(n). \tag{5}$$

Although it seems intuitive that the shortest addition chain is also a star chain, in 1958, Walter Hansen proved that for certain large exponents n, the value of $l(n)$ is smaller than $l^*(n)$ [**?**]. The smallest of such exponent values n equals 12 509.

Albeit counterintuitive, there also exist values of n for which $l(n) = l(2n)$ with the smallest example being $n = 191$. Here, both n and $2n$ have length l equal to 11. Furthermore, there exist values of n where $l(n) > l(2n)$ [**?**]. The smallest of such values of n is 375 494 703 [**?**].

Finally, the length seems to be difficult to compute for a specific class of numbers: let $c(r)$ be the smallest value of n such that $l(n) = r$ [**?**]. Therefore, $c(r)$ is the first integer value requiring r steps in the shortest addition chain [**?**]. To obtain such shortest addition chains is regarded more difficult than to obtain the shortest addition chain for some other greater value.

2.2 Techniques for Exponentiation

A number of techniques that are useful for cryptography, and that apply to both exponentiation in a multiplicative group and elliptic curve point multiplication, are explained in [**?**] and [**?**] and can be divided into three categories:

1. techniques for general exponentiation,
2. techniques for fixed-base exponentiation, and
3. techniques for fixed-exponent exponentiation.

In the following paragraphs, we use the term exponentiation, but all principles hold for both exponentiation and elliptic curve point multiplication. In the first category, the most straightforward ways to perform an exponentiation or a point multiplication, are the left-to-right and right-to-left binary methods. With the aforementioned method, the length of a chain n is upper bounded by $\nu(n) + \lambda(n) - 1$. In the worst case scenario, the binary method needs $2\lambda(n)$ multiplications and $3\lambda(n)/2$ on average [**?**].

An option for speeding up these algorithms consists of evaluating more than one bit of the exponent at a time after precomputing a number of multiples of the base. An example is the window or m-ary method that evaluates

m bits of the exponent at a time. The precomputation of base multiples maximizes the speed by minimizing the number of multiplications. However, the optimizations require a larger memory usage for the storage of the precomputed values. When the base is fixed, the precomputed multiples of the base can be prestored.

The m-ary method can be further generalized into sliding window methods and adaptive methods [**?**]. Another way of minimizing the number of multiplications without storing precomputed multiples of the base is by exponent recoding, which uses a representation of the exponent that is different from the binary representation. The recoding of the exponent requires additional resources on a chip (logic gates) or a microprocessor (program memory).

For elliptic curve cryptography, further speed optimizations are possible by considering elliptic curves with special properties, like the Gallant-Lambert-Vanstone (GLV) curve [**?**], the Galbraith-Lin-Scott (GLS) curve [**?**] or the FourQ curve [**?**]. In [**?**], side-channel security is taken into account in the derivation of efficient algorithms for scalar multiplication on GLS-GLV curves.

In this paper, we focus on addition chains for fixed-exponent exponentiations or fixed-scalar point multiplication without taking into account optimizations using specific fields or curves. We do not consider side-channel analysis, but we believe this does not undermine our results, since a number of side-channel countermeasures can be applied on top of the proposed addition chains. Examples are point blinding or randomized projective coordinates [**?**].

3 Related Work

In 1990, Bos and Coster presented the Makesequence algorithm that produces an addition sequence of a set of numbers [**?**]. The proposed method is able to find chains of large dimensions, and the authors conclude that their method is relatively more effective than the binary method. The heuristics in the algorithm choose, on the basis of a weight function, which method will be used to produce the sequence (the authors experimented with four methods). However, the authors report that their current weight function does not give satisfactory results and then, they decided to experiment with simulated annealing, but without success.

Nedjah and de Macedo Mourelle experimented with a genetic algorithm (GA) in order to find minimal addition chains [**?**]. They used binary encoding where value 1 means that the entry number is in the chain, and 0 means the opposite. This representation is not suitable for large numbers and the authors experimented with values of only up to 250. We note that the chromosome is of length 250 for that value, and for any value of practical interest the chromosome would amount to more than the memory of all computers in the world. The same authors focused on optimizing addition-subtraction chains with GAs [**?**]. They used the same representation and exponent values as in [**?**], which makes their work also far from applicable. They also experimented with addition-subtraction chains with a maximal value of 343 [**?**].

Nedjah and de Macedo Mourelle used Ant Colony Optimization to find minimal addition chains working with exponent sizes of up to 128 bits [?]. However, since they do not provide the numbers themselves, but only their sizes, it is impossible to assess the quality of this approach besides the fact that they report that it is better than the binary, quaternary, and octal method. The same authors extended their work for exponent sizes up to 1 024 bits resulting in better results for the Ant Colony Optimization algorithm than in cases when binary, quaternary, octal, and GA methods are used [?].

Cortés et al. proposed a genetic algorithm approach for which the encoding is the chain itself [?]. Besides that, the authors also proposed dedicated mutation and crossover operators. Using this approach, they report to successfully find minimal addition chains for numbers up to 14 143 037.

Cortés, Rodríguez-Henríquez, and Coello presented an Artificial Immune System for generating short addition chains of sizes up to 14 143 037 [?]. With that approach, the authors were successful in finding almost all optimal addition chains for exponents $e < 4\,096$.

Osorio et al. [?] proposed a genetic algorithm coupled with a local search algorithm and repair mechanism in order to find minimal short addition chains. This work is of high relevance since it clearly discusses the need for a repair mechanism when using heuristics for the addition chains problem.

León-Javier et al. [?] experimented with the Particle Swarm Optimization algorithm in order to find optimal short addition chains.

Nedjah and de Macedo Mourelle [?] implemented the Ant Colony Optimization algorithm on a SoC in order to speed up the modular exponentiation in cryptographic applications.

Sarkar and Mandal [?] used Particle Swarm Optimization to obtain faster modular multiplication in cryptographic applications for wireless communications.

Rodriguez-Cristerna and Torres-Jimenez [?] used a GA to find minimal Brauer chains where a Brauer chain is an addition chain in which each member uses the previous member as a summand.

Domínguez-Isidro et al. [?, ?] investigated the usage of evolutionary programming for minimizing the length of addition chains.

Finally, Picek et al. used genetic algorithms with customized operators to evolve short addition chains for values up to $2^{127} - 3$. This work also discusses several drawbacks appearing in related work as well as some of their possible solutions [?].

4 The Design of the Proposed Algorithm

Before discussing the choice of the algorithm, we briefly enumerate some basic rules our chains need to fulfill:

1. Every chain (solution) needs to be an ascending chain.
2. Every chain needs to be non-redundant, i.e., there should not be two identical numbers in a chain.

3. Every chain needs to be valid, i.e., every number in a chain needs to be a sum of two previously appearing numbers.
4. Every chain needs to start with the value 1 and finish with the desired exponent value.

When choosing the appropriate algorithm for the evolution of chains, we start with the considerations about the representation. If we disregard the approach where one encodes individuals in a binary way (i.e., for each possible value, we use either 0 if it is not a part of the chain, or 1 when it is a part of the chain), up to now there is not much of a choice. Indeed, encoding solutions as integer values where each value represents the number that occurs in the chain seems rather natural. Accordingly, we also use that representation, which we denote as encoding with *chain values*.

However, internally, our algorithm works with one more representation where we represent each value n as a pair of positions i_1 and i_2 that hold the previous values n_1 and n_2 forming the value n, which is denoted as encoding with *summand positions*.

Although such position-based encoding gives longer chromosomes, for large exponents the encoded values are much smaller and the memory requirements for storing an individual are consequently smaller. Furthermore, it is possible to use operators that work on the positions and to give an algorithm more options to combine solutions (since we have two positions for every number, the length of a chain encoded with positions is always twice as long as the one encoded with chain values).

For both representations, a GA seems a natural choice, but there is one important difference in both approaches. When using the representation based on chain values for large numbers, the chromosome encoding needs to support large numbers, while in the representation based on summand positions we only need to support large numbers for calculating the chain elements, but not for storing them.

However, one cannot aim to fulfill the aforementioned rules and use a standard GA. Therefore, we need to design a custom initialization procedure, mutation, and crossover operators. In fact, only the selection algorithm can be used as in the standard GA. In all our experiments, we work with k-tournament selection where $k = 3$. In each tournament, the worst of k randomly selected individuals is replaced by the offspring of the best two from the same tournament. This selection scheme not only eliminates the need for crossover probability, but has produced good results in different applications, in our experience.

Since initialization and variation operators are expected to produce many invalid solutions (in fact, for larger chains our experiments showed that it is highly unlikely that genetic operators will produce valid solutions) we also need to design a repair strategy. The repair strategy can be incorporated in each of the previous parts or to be considered as a special kind of operator, which is the approach we opted to follow. Next, we present the operators we use in our GA.

4.1 Initialization Algorithm

We designed the initialization algorithm aiming to maintain as much diversity as possible. We accomplished this by analyzing a number of known optimal chains (both star and standard chains) and checking the necessary steps to obtain them. Here, we note that if the initialization can produce only star chains and the mutation can generate only star steps, the whole algorithm will be able to produce only star chains. Naturally, one could circumvent this by adding additional steps in the repair mechanism. In that case, the model would not follow the intuition, since one expects that the repair mechanism only repairs the chains and it should not possess additional mechanisms for the generation of new values.

The initial population is generated via a set of hardcoded values that are positioned at the beginning of the chain together with randomly generated chain sequences as presented below. The probability values are selected on the basis of a set of tuning experiments.

- Set the zeroth element to 1 and the first element to 2.
- Uniformly at random select between all minimal subchains consisting of three elements (i.e., the second, third, and fourth positions in the chain) and a random choice of the second element (according to the rules, either the value 3 or 4).
- With a probability equal to 3/5, double the elements until they reach half of the exponent size.
- Check whether the current element and any previous element sum up to the exponent value.
- Uniformly at random, choose from among the following mechanisms to obtain the next value in the chain, under the constraint that it needs to be smaller than the exponent value:
  1. Sum two preceding elements of the chain.
  2. Sum the previous element and a random element.
  3. Sum two random elements. One random element is chosen between the zeroth position and the element in the middle of the chain and the second one is chosen between the middle element and the final (exponent) value.
  4. Loop from the element on the position $i - 1$ until the largest element that can be summed up with the last element is found.

4.2 Variation Operators

Next, we present the mutation and crossover operators we use. They are very similar to the operators provided, for instance, in [**?**, **?**]. For such a specific problem as the one we study here, the task of devising new operators is difficult. Furthermore, many operators reduce to the ones described here. For instance, we present here something that is analogous to a single-point mutation, but since the change in a single position will invalidate the chain, after the repair

mechanism, the mutation can also be regarded as a mixed mutation. Therefore, the number of mutation points is irrelevant since a single point change brings changes in every position until the end of the chain.

Since we have several branches in the mutation operator, one can say that those branches could be separated into different mutation operators. We note that there are more possibilities on how to combine two values to form a new value in a sequence and there could be possibilities for additional mutation operators. On the other hand, we implemented two crossover operators and we consider advantageous to use both of them, since this promotes diversity. However, identifying which of them is better than the other is hard, since this depends on the exponent value that we aim to reach.

Crossover

We implemented two versions of the crossover operator: one-point crossover and two-point crossover. We provide the pseudocode for one-point crossover in Algorithm 1 and the two-point version is analogous. The selection of which crossover is used is done uniformly at random for each call of the crossover operator. Here, the function $FindLowestPair(P, i, pair_1, pair_2)$ determines the pair of elements with lowest indexes $(pair_1, pair_2)$ which give the target element i in a chain P. The dominant difference between the mutation operator and the crossover operator lies in the fact that in the crossover, we have defined the rules on how to build elements while in the mutation we do not have such strict rules. However, since both require the usage of the repair mechanism, that difference can become rather blurred.

Algorithm 1 Crossover operator.

Require: Exponent $exp > 0$, *Parent addition chains* P_1, P_2
  $rand = random(3, exp - 1)$
  for all i such that $0 \leq i \leq rand$ **do**
    $e_i = P_{1\,i}$
  end for
  for all i such that $rand \leq i + 1 \leq n$ **do**
    $FindLowestPair(P_2, i, pair_1, pair_2)$
    $e_i = e_{pair_1} + e_{pair_2}$
  end for
  $RepairChain(e, exp)$
  return $e = e_0, e_1, ..., e_n$

Mutation

The mutation operator is again similar to those presented in the related literature, but we allow more diversity in the generation process as presented in Algorithm 2. As already stated, since the mutation invalidates the chain, it

is impossible to expect small changes (except when the mutation point is at the end of the chain) and therefore, this is actually a macromutation operator.

Algorithm 2 Mutation operator.

Require: Exponent $exp > 0, e = e_0, e_1, ..., e_n$
  $rand = random(2, exp - 1)$
  $rand_2 = random(0, 1)$
  if $rand_2 == 1$ **then**
    $e_{rand} = e_{rand-1} + e_{rand-2}$
  else
    $rand_3 = random(2, rand - 1)$
    $e_{rand} = e_{rand-1} + e_{rand_3}$
  end if
  $RepairChain(e, exp)$
  return $e = e_0, e_1, ..., e_n$

4.3 The Repair Algorithm

Function $RepairChain(e, exp)$ takes the chain e and repairs it in the following way:

1. Delete duplicate elements in the chain.
2. Delete elements greater than the exp value.
3. Check that all elements are in ascending order, if not, sort them.
4. Ensure that the chain finishes with the exp value by repeating operations in the following order:
   (a) Try to find two elements in the chain that result in exp.
   (b) Uniformly at random apply:
      i. Double the last element of the chain while it is smaller than exp.
      ii. Add the last element and a random element.
      iii. Add two random elements.

This function is in many ways similar to the Initialization procedure, but in this case, the primary goal is removing redundant chain elements, rather than maximizing diversity as is the case in the Initialization.

There are several places in our algorithm where we choose what branch to enter based on random values. We decided to use uniform random values where each branch has the same probability to be chosen. We believe this mechanism can be further improved. One trivial modification would be with regards to whether one wants to obtain a star chain or not. In the case when only star chains are wanted, then the branches that cannot result in a star step can be set either to a zero or some small value, analogous for the case when we want to have a larger number of standard steps.

The number of independent runs for each experiment is 50. For the stopping criterion we use *stagnation*, which we set to 100 generations without improvement. We set the total number of generations to 1 500. The population size

is set to 300 in all experiments. We note that larger population sizes perform even better thanks to increased diversity from the initialization mechanism, but for large exponent values the evolution takes a long time. With the current setting, even for relatively large exponent size, one evolutionary run finishes in less than one hour. We note that all listed parameters are selected based on a tuning phase, whose results we do not give here due to the lack of space. For all the experiments, we use the Evolutionary Computation Framework (ECF) [**?**].

5 Finding Short Addition Chains

In this section, we concentrate on a number of scenarios where the goal is to find the shortest addition chains.

5.1 The Fitness Function

In all the experiments in this section, we use a simple fitness function where the goal is *minimization*. The number of elements in the chain (i.e., the length *len* of an addition chain *chain* for exponent value n) is minimized as given by the equation:

$$fitness(chain) = len(chain). \tag{6}$$

5.2 Tests Based on a Comparison with Previous Work

For the first category, we use a set of exponent values that are also used in previous work. Namely, those are the exponents belonging to the class that is difficult to calculate according to [**?**]. Recall, those values are the minimal integers that form an addition chain of a certain length i. Up to now, experiments had been done for values of i up to 30 [**?**, **?**]. However, in an effort to evaluate the performance of our algorithm with even higher values we experimented with values up to $i = 40$. Furthermore, for each of those values we give statistical indicators in order to understand better the performance of our algorithm as well as to serve as a reference for future work.

We note that any comparison with previous work is difficult since other authors only report the value (and the chain) that presents the best obtained solution. From the reproducibility and the efficiency side, we find those approaches somewhat incomplete since it makes a big difference if the algorithm found the best possible value in one instance out of 100 runs or in 90 instances out of 100 runs.

We note that for exponent values $n < 2^{27}$ one can find optimal chains online [**?**], while values up to $n = 2^{31}$ can be downloaded from the same web page. Besides our algorithm, we implemented the binary algorithm as well as two variants of the window method. In the first m-window method (called Window method in tables), we set the value of k to four in the expression

Table 1: c(r) family of the exponent values.

r	c(r)	Binary	Window	Opt. window	GA		
					Min	Avg	Stdev
30	14 143 037	38	40	34	30	30.92	0.60
31	25 450 463	38	42	35	31	32.62	0.66
32	46 444 543	42	43	36	32	33.50	0.54
33	89 209 343	42	44	38	33	34.46	0.81
34	155 691 199	42	45	39	34	35.44	1.03
35	298 695 487	46	47	41	35	35.67	0.74
36	550 040 063	45	47	41	36	37.96	0.83
37	994 660 991	46	48	42	37	38.76	1.47
38	1 886 023 151	48	48	42	38	40.28	1.21
39	3 502 562 143	48	49	43	39	41.36	1.19
40	6 490 123 999	52	52	45	41	41.77	0.63

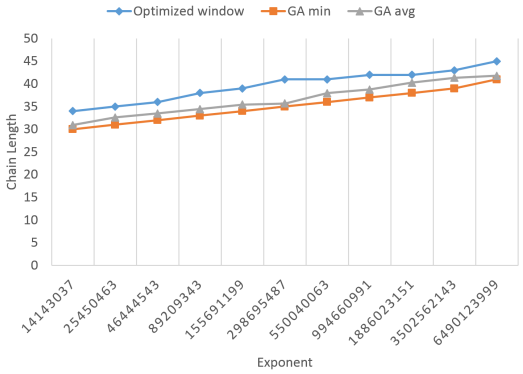


Fig. 1: Efficiency comparison, GA and Optimized window approach, c(r) values.

$m = 2^k$. It has been shown [**?**] that with this method the length of the chain is:

$$l(n) \leq log_2(n) + 2^{k-1} - (k-1) + \lfloor log_2(n)/k \rfloor, \forall k. \tag{7}$$

The second version of the window method (called Opt. window method in tables) tries to optimize Eq. (7) by choosing the value k that minimizes $2^{k-1} - (k-1) + \lfloor log_2(n)/k \rfloor$. We emphasize that none of the aforementioned methods should be regarded as the state-of-the-art, but only as methods that give good results and should serve as the baseline cases.

The results are given in Table 1 where it is easy to observe that the GA performs better than the binary, window, and optimized window methods. In Figure 1 we depict a comparison between the GA and the Optimized window method for $c(r)$ values.

Table 2: Testing random values.

n	l(n)	Binary	Window	Opt. window	GA		
					Min	Avg	Stdev
488 705	23	26	33	27	23	23.53	0.51
1 273 909	25	29	36	30	25	25.87	0.63
3 399 779	25	31	37	31	25	26.87	0.87
5 425 679	27	32	38	32	28	28.23	0.50
9 264 263	28	34	40	34	28	29.63	0.67
20 279 147	29	39	42	36	30	31.07	0.52
51 950 083	30	34	42	36	30	31.20	0.55
115 216 741	31	39	44	38	32	33.60	1.01
159 963 579	N/A	41	45	39	34	35.20	0.85
310 469 637	N/A	36	46	39	34	34.23	0.43
1 073 740 801	N/A	49	47	41	35	35.26	0.45

5.3 Testing Random Values

Up to now, we investigated a number of values of various sizes where we observe that the GA approach performs very well. However, the investigated values have a certain structure, i.e., they are not randomly chosen. Our goal in this set of experiments is to check how the GA performs when we look for the shortest addition chains for random values of various sizes. In order to obtain such values, we use the infrastructure from RANDOM.ORG [**?**] where the only constraint we enforce is to use odd values. Furthermore, we experiment with values between 2^{20} and 2^{31} in order to be able to compare with the experimentally validated shortest addition chains [**?**]. The results are given in Table 2 while in Figure 2 we display the comparison between the GA and the Optimized window approaches. Note that for the last three values we write N/A in the $l(n)$ column since those values are too large to be obtained from [**?**]. As in the previous scenario, we see that the GA approach is by far the best out of those tested here.

5.4 Testing "Difficult" Values

In this section, we test several values that can be regarded as difficult. That difficulty stems from the fact that all experiments done up to now indicate those numbers have a small number of optimal addition chains (i.e., there are only a few options on how to build optimal addition chains). Furthermore, those numbers have a relatively large number of small steps (cf. with the value $n = 2^k$ that has only one optimal addition chain but is simple due to the lack of small steps). The results are given in Table 3 and Figure 3. The values in the table are experimentally shown to have 7 small steps and in total a length of 41 steps. Note that although the GA outperforms the other tested methods, it is still not able to reach optimal addition chains (except in one case). Furthermore, here we can observe a relatively small difference in the performance between the GA and the Optimized window method.

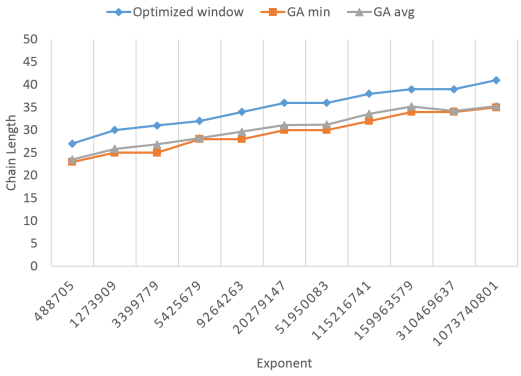


Fig. 2: Efficiency comparison, GA and Optimized window approach, random values.

Table 3: "Difficult" values.

n	Binary	Window	Opt. window	GA		
				Min	Avg	Stdev
17 180 843 711	50	51	45	42	43.93	0.96
17 181 535 967	49	52	46	42	45.16	1.45
17 181 824 999	50	52	46	42	44.40	1.08
17 181 857 663	50	52	46	41	44.46	1.24
17 181 878 143	50	52	46	42	44.45	1.07
17 181 921 023	51	52	46	42	44.16	1.27
17 181 425 531	51	52	46	43	44.06	0.78
17 181 433 703	50	52	46	42	43.80	0.69
17 181 750 911	49	52	46	42	44.35	1.27
17 181 793 151	50	52	46	42	44.96	1.21
17 181 963 167	51	52	46	42	43.99	1.02
17 182 209 983	50	52	46	42	44.83	1.36
17 182 210 751	49	52	46	42	44.65	1.07
17 182 215 157	48	52	46	42	44.48	1.14
17 182 219 767	50	52	46	43	44.55	1.03
17 182 226 303	51	52	46	42	44.49	1.01
17 182 318 319	49	52	46	42	44.77	1.20

5.5 Real-world benchmark tests

Finally, as a real-world benchmark, we investigate two values that are used in practice: $2^{127} - 3$ and $2^{255} - 21$. The first value has applications in certain high-speed Diffie-Hellman implementations [**?**] while the latter one is used in the inversion part in the 25519 curve [**?**]. To provide additional experiments for a comparison, we start with much smaller values and we gradually progress by increasing the exponent in steps of ten, i.e., the value following $2^{37} - 3$ equals $2^{47} - 3$. We finish the experiments with the exponent values $2^{127} - 3$ and $2^{255} - 21$. The results are given in Tables 4 and 5. Similarly as in the previous

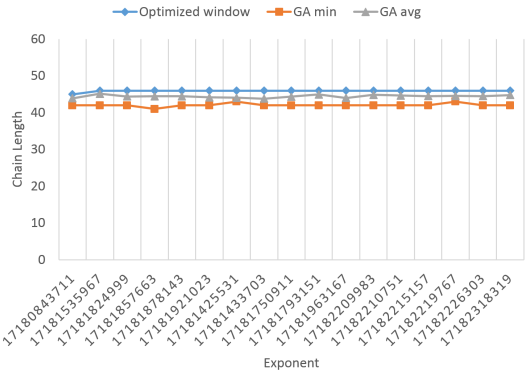


Fig. 3: Efficiency comparison, GA and Optimized window approach, "difficult" values.

Table 4: Exponents up to $2^{127} - 3$.

Exponent	Binary	Window	Opt. window	GA		
				Min	Avg	Stdev
$2^{37} - 3$	71	57	51	43	45.32	0.99
$2^{47} - 3$	91	69	63	54	56.25	1.11
$2^{57} - 3$	111	82	76	64	64.90	0.87
$2^{67} - 3$	131	94	88	73	73.22	0.43
$2^{77} - 3$	151	107	101	85	85.44	0.51
$2^{87} - 3$	171	119	113	97	104.36	3.56
$2^{97} - 3$	191	132	126	106	107.27	0.91
$2^{107} - 3$	211	144	138	115	115.71	0.75
$2^{117} - 3$	231	157	151	126	126.68	0.89
$2^{127} - 3$	251	169	163	136	136.83	0.83

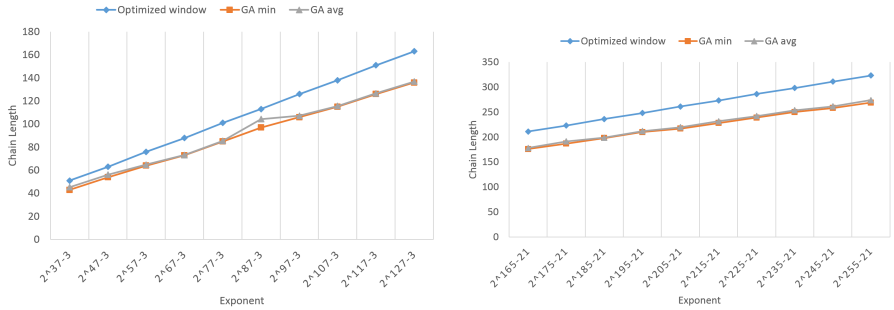
cases, the GA approach is again superior while the differences between the results are even more striking than before. We note that for the $2^{127} - 3$ value, the GA found a chain of the same length as the currently shortest known. On the other hand, for the value $2^{255} - 21$ our best results equals 269 steps while the best known result is only 265 steps. In Figures 4a and 4b we give a comparison between the GA and the Optimized window approaches for values from Tables 4 and 5, respectively.

6 On the Implementation Perspective

Up to now, our experiments investigated only the evolution of the shortest addition chains. However, in realistic scenarios, addition chains also have an important perspective that concerns implementation details. Accordingly, here we concentrate on two such implementation scenarios where we start with the motivation for each problem and then we present the obtained results. Note

Table 5: Exponents up to $2^{255} - 21$.

Exponent	Binary	Window	Opt. window	GA		
				Min	Avg	Stdev
$2^{165} - 21$	326	217	211	176	178.21	1.81
$2^{175} - 21$	346	229	223	187	191.28	2.97
$2^{185} - 21$	366	242	236	198	198.90	0.88
$2^{195} - 21$	386	254	248	210	211.94	1.85
$2^{205} - 21$	406	267	261	217	219.84	1.87
$2^{215} - 21$	426	279	273	228	231.72	2.54
$2^{225} - 21$	446	292	286	239	242.03	2.67
$2^{235} - 21$	466	304	298	250	253.52	2.27
$2^{245} - 21$	486	317	311	258	261.55	2.41
$2^{255} - 21$	506	329	323	269	273.81	2.57



(a) Values up to $2^{127} - 3$.

(b) Values up to $2^{255} - 21$.

Fig. 4: Efficiency comparison, GA and the Optimized window approaches.

that we disregard certain aspects of the problem and we concentrate only on the addition chains perspective.

6.1 Adding the Weights of Operations

As already said, finding the shortest addition chains can be an extremely difficult problem. However, one can also consider how many shortest addition chains are there for a certain value and whether all those chains are equivalent. The number of the shortest chains for a given value depends on the specific value while from the theoretical perspective all chains of the same size are equally good/optimal. However, since those chains often need to be implemented in hardware, we need to consider the implementation costs where the multiplication operation is more expensive than the squaring operation.

To elaborate this further, we start with a small example where we consider the value 511. By checking the online repository of the shortest addition chains [**?**] we see that the length of that addition chain equals 12. Furthermore, we run GA for 30 times where we obtained 30 optimal chains of length 12.

However, when inspecting those solutions we see that there are 20 unique solutions, all of them reaching the value 511 in 12 steps. Out of those 20 solutions, we obtain 3 solutions with 4 multiplications, 12 solutions with 5 multiplications, and 5 solutions with 6 multiplications. Next, we give examples of each of the categories discussed:
$1 \rightarrow 2 \rightarrow 3 \rightarrow 6 \rightarrow 12 \rightarrow 15 \rightarrow 30 \rightarrow 60 \rightarrow 120 \rightarrow 240 \rightarrow 480 \rightarrow 510 \rightarrow 511$.
$1 \rightarrow 2 \rightarrow 4 \rightarrow 6 \rightarrow 10 \rightarrow 20 \rightarrow 30 \rightarrow 60 \rightarrow 120 \rightarrow 240 \rightarrow 480 \rightarrow 510 \rightarrow 511$.
$1 \rightarrow 2 \rightarrow 3 \rightarrow 6 \rightarrow 12 \rightarrow 18 \rightarrow 30 \rightarrow 31 \rightarrow 60 \rightarrow 120 \rightarrow 240 \rightarrow 480 \rightarrow 511$.

All three previous solutions represent the shortest addition chains for a value 511, but from the implementation perspective, the first solution is the cheapest, while the last one is the most expensive. In this section, our goal is to find the shortest, but also as "cheap" as possible chains for the value $2^{127} - 3$. However, first we need to answer how much more expensive is the multiplication when compared with the squaring. In general, the multiplication operation is more expensive than the squaring operation where the exact cost ratio depends on several factors. For instance, in [**?**], the author writes that general multiplication costs 243 floating-point operations and squaring costs 162 floating point operations which gives a ratio of 0.67. On the other hand, L. Duc-Phong estimates that the squaring costs 0.8 multiplications [**?**]. In this set of experiments, we follow the latter estimate.

The Fitness Function

In this set of experiments, our fitness function aims to *minimize* the total cost of instructions:

$$fitness(chain) = a \times \sum squaring + b \times \sum multiplication, \qquad (8)$$

where $a = 0.8$ and $b = 1$.

We note that instead of immediately trying to find chains that are as short as possible and having as small number of multiplications as possible, we could had first aimed to find the shortest chains and then try to improve on the type of operations while maintaining the chain length. However, we considered this option to be much harder for that GA so we did not pursue it further.

Results

Due to the size of the obtained solutions, we do not list the whole addition chains here, but instead, we discuss their lengths and the number of the each type of operations. We obtained 12 different chains with the total length of 136 steps (therefore, with the shortest known length). Out of those 12 chains, 10 chains consist of 125 squaring operations and 11 multiplication operations while 2 chains consist of **126** squaring operations and only **10** multiplications. Therefore, we succeeded in obtaining two chains that are cheaper when implemented in hardware when compared with the other evolved chains of length 136. Finally, we note that we did not find any chain of length 136 that has more than 11 multiplications.

6.2 Extending the Operations Set

In the second implementation scenario, we consider the case when a certain addition chain is to be implemented in hardware. We use here the example of inversion in $GF(2^{127})$ [**?**]. The optimal chain for the value 127 is trivial to find and it equals [**?**]:
$1 \rightarrow 2 \rightarrow 3 \rightarrow 6 \rightarrow 12 \rightarrow 15 \rightarrow 30 \rightarrow 60 \rightarrow 63 \rightarrow 126 \rightarrow 127$.
Let us consider how such a chain would be implemented with an example from Sage [**?**]:

```
def Inversion (din):        r0 = r1*r0          r0 = r1*r0
r0 = din                    r1 = r0^(2^6)       r1 = r0^(2^3)
r1 = r0^(2^1)               r0 = r1*r0          r0 = r1*r3
r0 = r1*r0                  r1 = r0^(2^3)       r1 = r0^(2^63)
r1 = r0^(2^1)               r0 = r1*r3          r0 = r1*r0
r0 = r1*din                 r1 = r0^(2^15)      r0 = r0^(2^1)
r3 = r0                     r0 = r1*r0          return r0
r1 = r0^(2^3)               r1 = r0^(2^30)
```

We see there are in total 9 multiplications and 10 squaring operations. However, for instance to calculate $r_1 = r_0^{2^{126}}$ it would require that we either have the value $r_1 = r_0^{2^{63}}$ stored in the memory or to find it on-the-fly. An obvious technique to circumvent this problem is to use a number of operations that can reach the desired value faster than the multiplication or squaring operations. Here, we concentrate on an example where such operations are implemented in an FPGA core. As already said, besides the multiplication and squaring operations we can implement also a small number of additional operations. Since squaring operations are much cheaper (the exact ratio depends on the implementation) than the multiplications, we ideally want those additional operations to be the powers of the squaring operation, i.e., squaring equals x^{2^1} and additional operations are of the form $x^{2^2}, x^{2^3}, x^{2^4}, \ldots, x^{2^z}$ where z cannot be too large so we limit it to the values smaller than 10. Note that besides the constraint that z cannot be too large, we also need to limit the number of additional operations we have on our disposal due to the hardware constraints. In accordance with that, we select our squaring operations set size to the maximal value of 4 (note that x^{2^1} must be used which means we have only up to three more possible squaring operations to choose). One rather standard choice for the squaring operations is to use the powers of two, i.e., $x^{2^1}, x^{2^2}, x^{2^4}, x^{2^8}$. Now, we consider how to calculate $r1 = r0^{2^{63}}$ with the aforesaid operations:

```
r1 = r0^(2^8)       r1 = r1^(2^8)       r1 = r1^(2^2)
r1 = r1^(2^8)       r1 = r1^(2^8)       r1 = r1^(2^1)
r1 = r1^(2^8)       r1 = r1^(2^8)
r1 = r1^(2^8)       r1 = r1^(2^4)
```

Note that we need 10 instructions to calculate the value $r_1 = r_0^{2^{63}}$ and, in total, we need 30 instructions to calculate all squaring operations in the *Inversion* function given above. Besides that, it becomes evident from the above example that we additionally require 9 multiplications to calculate the chain. As already said, squaring operations are cheaper than multiplication operations but the exact ratio depends on the implementation scenario. We work here with an assumption where the multiplication has a cost which is the double as that of squaring. Therefore, if we set the cost of squaring to 1

and multiplication to 2, it means that the above chain has the total cost of **48** instructions. We formulate the problem in two possible scenarios:

- Find a different addition chain that uses operations $x^{2^1}, x^{2^2}, x^{2^4}, x^{2^8}$ and results in a smaller number of operations.
- Use default addition chain but select different squaring operations that will result in a smaller number of instructions.

Finding Different Addition Chains

When finding different addition chains that use the predefined set of squaring operations, we can use the fitness function where the goal is to *minimize* the number of instructions necessary to build a chain:

$$fitness(chain) = \sum instructions\_in\_squaring + 2 \times \sum multiplication. \quad (9)$$

Note that we multiply the multiplication instructions by 2 since we said they are twice as expensive as the squaring instructions. To test which number of operations is necessary for each squaring value we simply run repeated division processes with all the values in the operations set (from the largest to the smallest, i.e., 8, 4, 2, 1) while the squaring value is larger than 0.

With this approach we are able to find a number of chains that require 20 instructions for all squaring values. However, all those chains require 2 multiplications more than the original chain. We give an example of such evolved chain: $1 \rightarrow 2 \rightarrow 4 \rightarrow 8 \rightarrow 16 \rightarrow 32 \rightarrow 64 \rightarrow 88 \rightarrow 120 \rightarrow 124 \rightarrow 126 \rightarrow 127$.

Note that although the number of multiplications is larger and the chain is longer than the shortest chain possible, still this chain requires less operations to implement – 11 multiplication operations and 20 squaring operations, which equals in total **42** instructions. Note that this chain requires a smaller number of operations than the default chain even if the multiplication operation is 4 times more expensive than the squaring operation. We believe this scenario represents an interesting example on how sometimes even larger chains can be optimal from the implementation perspective when compared to the shortest addition chains.

Finding Different Squaring Operations

In this scenario, we use the default (i.e., the shortest) addition chain and we investigate which squaring operations are to be used to minimize the cost of the whole chain when considering the number of instructions. Recall that we limit the number of squaring operations to 4 and the power of the largest squaring operation to 9. However, this represents only one example from practice and we note that further investigation with a different number of squaring operations and their dimensions would constitute an interesting research direction.

Since here we already have an addition chain that we need to use and we are looking for a set of values representing power operations, we do not use

our custom-made GA. Instead, we use a standard GA that has a permutations encoding where we limit the number of operations that can be used to 4 out of 9 possible. To state it differently, our encoding will contain 4 values that represent the optimal choice of the power values. All the other GA parameters are kept the same as in the previous experiments. The fitness function aims to *minimize* the number of instructions necessary to build all squaring operations. Here, we can disregard the multiplication part since it is fixed (i.e., our chain consists of 9 multiplications):

$$fitness(chain) = \sum instructions\_in\_squaring. \tag{10}$$

The results point that the optimal set of operations is $x^{2^1}, x^{2^3}, x^{2^6}, x^{2^9}$, which results in a total of 20 squaring instructions and 9 multiplications. Therefore, our chain built with those instructions requires in total **38** instructions.

7 Discussion

In this paper, we conduct an extensive analysis on the efficiency of the GA approach when finding shortest addition chains or addition chains that are cheap to implement in hardware. When comparing our approach with previous work as well as several deterministic algorithms, we see that the GA performs extremely well. From the results obtained we see that the $c(r)$ family of numbers although usually perceived as very difficult to calculate does not provide much difficulty for the GA. The motivation behind Random Values testing stems from the fact that we want to check whether our approach favors some structure (regardless of how complex that structure may be), and would it have difficulties with random values that presumably do not possess any specific structure. However, our experiments show that yet again the GA is easily able to reach optimal solutions. Finally, we tested a new set of numbers for which it should be difficult to find shortest chains because it is believed that those numbers have only a few optimal chains as well as that they have relatively many small steps. This is the first test suite where our approach could not find optimal solutions, but was usually off by one step. Therefore, we believe these numbers should represent the future reference point when investigating the performance of metaheuristic techniques in the evolution of the shortest addition chains.

When discussing real-world cases ($2^{127}-3$ and $2^{255}-21$), we can notice that those numbers are much longer than those usually tested with metaheuristics. Our experiments show that despite the (extreme) size of the numbers, the GA is again performing very well when compared with deterministic algorithms. For the value $2^{127} - 3$ the shortest known chain has 136 elements, which is the same value our algorithm reached. The question is whether this should be regarded as a success or a failure. In a sense, it depends on the perspective; if one knows that the value 136 was obtained (somewhat surprising) by a pen and

paper approach in a matter of a few hours by an expert, then our result does not seem impressive. However, recall Definition 3 where it is easy to calculate that $n = (2^{127} - 3)$ has a chain of a length at least equal to 130 since this exponent has 125 ones in its binary representation. This means that even if our solution does not have the optimal length, it is quite close to that value. For the value $2^{255} - 21$ our shortest chain has length 269, which is a huge improvement over all three tested deterministic methods. However, again, by pen and paper method the shortest obtained chain for that value has length of 265. Therefore, our algorithm for this test case obviously cannot compete with the knowledge of an expert. Still, we note that our results are competitive due to the relatively high speed of the evolution process as well as the fact that we are able to obtain multiple chains of size 269. Furthermore, we note that the chains obtained by pen and paper utilize expert knowledge of the numbers' structure that we do not use in our black-box optimization.

As the main future research challenge, we see the need to increase the speed of the evolution process in order to be able to offer our GA as an on-the-fly generation mechanism. One option would be to write a custom implementation of large number arithmetic that could utilize full support of modern processors. The second option would be to use some faster evolutionary algorithm like Evolution Strategy (ES). Our preliminary experiments with ES show potential since this algorithm is able to reach optimal values for many of the tested numbers. Finally, it should be possible to use a smarter seeding technique where the initial population would be obtained by various deterministic methods and possibly small mutations in order to increase the diversity.

Besides the experiments dealing with the evolution of the shortest addition chains, we introduced here a scenario where we try to optimize the chain from the implementation perspective, i.e., when considering instructions one needs to use in hardware to really implement addition chains. We experimented with two scenarios where in the first one we fixed the addition chain and tried to find a set of additional instructions to make the implementation cheaper. On the other hand, in the second scenario we fixed a small set of additional operations and then tried to find a chain that has a smaller number of instructions. Both scenarios yielded good results which constitutes heuristics a good choice for realistic settings. We especially note the interesting case where we managed to find addition chain consisting of more operations than the shortest addition chain, but one that still has a smaller number of operations than the shortest addition chain.

8 Conclusions

In this work, we showed that GAs can be used to find the shortest addition chains for a wide set of exponent sizes. However, we note this problem is not as easy as could be perceived from a number of related publications. Indeed, the first step is the design of a custom Genetic Algorithm and then one needs to carefully tune the parameters. Here, we managed to find chains that are

either optimal (where it was possible to confirm based on related work) or as short as possible for a number of values.

From that perspective, we also see this work as a reference work against which new heuristics should be tested, since it is undoubtedly possible to compare the results. Furthermore, we present here a set of numbers that seem to be especially difficult for heuristic search techniques which will make an interesting future benchmark suite. As far as we know, we are the first to investigate these kind of heuristics for exponent values that have a real world usage. Besides the evolution of the shortest addition chains, we were also able to find addition chains that are extremely cheap when implemented in hardware which opens a complete new research perspective for metaheuristics and addition chains.

Acknowledgements

This work has been supported in part by Croatian Science Foundation under the project IP-2014-09-4882. The second author acknowledges support from CONACyT project no. 221551.

A Sage Example With the Optimal Set of Squaring Instructions

Here we give an example of the inversion built with $x^{2^1}, x^{2^3}, x^{2^6}, x^{2^9}$ instructions that has in total 9 multiplications operations (18 multiplication instructions) and 20 squaring instructions. The lines beginning with # denote comments.

```
def Inversion (din):
r0 = din
r1 = r0^(2^1)
r0 = r1*r0
r1 = r0^(2^1)
r0 = r1*din
r3 = r0
r1 = r0^(2^3)
r0 = r1*r0
r1 = r0^(2^6)
r0 = r1*r0
r1 = r0^(2^3)
r0 = r1*r3
#  r1 = r0^(2^15)
r1 = r0^(2^9)
r1 = r1^(2^6)
r0 = r1*r0
#  r1 = r0^(2^30)
r1 = r0^(2^9)
r1 = r0^(2^9)
r1 = r0^(2^9)
r1 = r0^(2^3)
r0 = r1*r0
r1 = r0^(2^3)
r0 = r1*r3
#  r1 = r0^(2^63)
r1 = r0^(2^9)
r1 = r0^(2^9)
r1 = r0^(2^9)
r1 = r0^(2^9)
r1 = r0^(2^9)
r1 = r0^(2^9)
r0 = r1*r0
r0 = r0^(2^1)
return r0
```