FAST PRE-PROCESSING FOR THE SLIDING WINDOW METHOD USING GENETIC ALGORITHMS

Nadia Nedjah and Luiza de Macedo Mourelle

Department of Systems Engineering and Computation,
Faculty of Engineering, State University of Rio de Janeiro,
Rio de Janeiro, Brazil
{nadia, ldmm}@eng.uerj.br

ABSTRACT
Modular exponentiation is a cornerstone operation to several public-key cryptography systems such as the RSA. It is performed using successive modular multiplications. The latter is time consuming for large operands. Accelerating public-key cryptography software or hardware needs reducing the total number of modular multiplication needed. This paper introduces a novel idea based on genetic algorithms for evolving an optimal addition chain that allows one to perform pre-computations necessary in the window modular exponentiation methods. The obtained addition chain allows one to perform exponentiation with a minimal number of multiplication and hence implementing efficiently the exponentiation operation. We compare our results with those obtained using the algorithm of Brun.

Keywords: Cryptosystems, sliding window method, M-ary method, Modular exponentiation.

1. INTRODUCTION

Public-key cryptographic systems (such as the RSA encryption scheme (Koç, 1994), (Rivest et. al., 1978)) often involve raising large elements of some groups fields (such as GF(2^n) or elliptic curves (Menezes, 1993)) to large powers. The performance and practicality of such cryptosystems is primarily determined by the implementation efficiency of the modular exponentiation. As the operands (the plain text of a message or the cipher (possibly a partially ciphered) are usually large (i.e. 1024 bits or more), and in order to improve time requirements of the encryption/decryption operations, it is essential to attempt to minimise the number of modular multiplications performed.

A simple procedure to compute $C = T^E \bmod M$ based on the paper-and-pencil method is described in Algorithm 1. This method requires $E-1$ modular multiplications. It computes all powers of T: $T \to T^2 \to T^3 \to ... \to T^{E-1} \to T^E$.

Algorithm 1. *simpleExponentiationMethod*(T, M, E)
```
1:   C := T;
2:   for i := 1 to E-1 do
3:      C := (C × T) mod M;
4:   return C
end algorithm.
```

The paper-and-pencil method computes more multiplications than necessary. For instance, to compute T^8, it needs 7 multiplications, i.e. $T \to T^2 \to T^3 \to T^4 \to T^5 \to T^6 \to T^7 \to T^8$. However, T^8 can be computed using only 3 multiplications $T \to T^2 \to T^4 \to T^8$. The basic question is: what is the fewest number of multiplications to compute T^E, given that the only operation allowed is multiplying two already computed powers of T? Answering the above question is *NP*-hard, but there are several efficient algorithms that can find a near optimal ones. However, these

algorithms need some pre-computations that if not performed efficiently can deteriorate the algorithm overall performance. The pre-computations are themselves an ensemble of exponentiations and so it is also *NP*-hard to perform them optimally. In this paper, we concentrate on this problem and engineer a new way to do the necessary pre-computations very efficiently. We do so using evolutionary computation. We compare our results with those obtained using the Brun's algorithm (Begeron, et. al., 1989).

Evolutionary algorithms are computer-based solving systems, which use evolutionary computational models as key element in their design and implementation. A variety of evolutionary algorithms have been proposed. The most popular ones are *genetic algorithms* (DeJong and Spears, 1989), (Haupt and Haupt, 1998). They have a conceptual base of simulating the evolution of individual structures via the Darwinian natural selection process. The process depends on the adherence of the individual structures as defined by its environment to the problem pre-determined constraints. Genetic algorithms are well suited to provide an efficient solution of *NP*-hard problems (DeJong and Spears, 1989), (Haupt and Haupt, 1998).

This paper will be structured as follows: in Section 2, we present the window methods; in Section 3, we define addition chains and sequences; in Section 4, we give an overview on the concepts of genetic algorithms; in Section 5, we explain how these concepts can be used to compute a minimal addition sequences to perform efficiently necessary pre-computations in the window methods; in Section 6, we present some useful results; and finally, we draw some conclusions in Section 7.

2. WINDOW METHODS

Generally speaking, the window methods for exponentiation (Knuth, 1981) may be thought of as partitioning in *k*-bits windows the binary representation of the exponent *E*, pre-computing the powers in each window one by one, squaring them *k* times to shift them over, and then multiplying by the power in the next window.

There are several partitioning strategies. The window size may be constant or variable. For the *m*-ary methods, the window size is constant and the windows are next to each other. On the other hand, for the sliding window methods the window size may be of variable length. It is clear that zero-windows, i.e. those that contain only zeros, do not introduce any extra computation. So a good strategy for the sliding window methods is one that attempts to maximise the number of zero-windows. The details of *m*-ary methods are exposed in Section 2.1 while those related to sliding constant-size window methods are given in Section 2.2. In Section 2.3, we introduce the adaptive variable-size window methods.

2.1. *M*-ary Methods

The *m*-ary methods (DeJong and Spears, 1989) scans the digits of *E* form the less significant to the most significant digit and groups them in partitions of equal length $\log_2 m$, where *m* is a power of two.

In general, the exponent *E* is partitioned into *p* partitions, each one containing $l = \log_2 m$ successive digits. The ordered set of the partition of *E* will be denoted by $\wp(E)$. If the last partition has less digits than $\log_2 m$, then the exponent is expanded to the left with at most $\log_2 m - 1$ zeros. The *m*-ary algorithm is described in Algorithm 2, wherein V_i denotes the decimal value of partition P_i.

Algorithm 2. *m-aryMethod*(T, M, E)

1: Partition E into p l-digits partitions;
2: for i := 2 to *m*
3: Compute T^i mod M;
4: C := $T^{V_{p-1}}$ mod M;
5: for i := p-2 downto 0

6: $C := C^{2^l} \bmod M$;
7: if $V_i \neq 0$ then $C := C \times T^{V_i} \bmod M$;
8: return C;
end algorithm.

2.2. Sliding Window Methods

For the sliding window methods the window size may be of variable length and hence the partitioning may be performed so that the number of zero-windows is as large as possible, hence reducing the number of modular multiplication necessary in the squaring and multiplication phases. Furthermore, as all possible partitions have to start (i.e. in the right side) with digit 1, the pre-processing step needs to be performed for odd values only. The sliding method algorithm is presented in Algorithm 3, wherein d denotes the number of digits in the largest possible partition and L_i the length of partition P_i.

Algorithm 3. *slidingWindowMethod*(T, M, E)
1: Partition E using the given strategy;
2: for i := 2 to 2^d-1 step 2
3: Compute $T^i \bmod M$;
4: $C := T^{P_{p-1}} \bmod M$;
5: for i := p-2 downto 0
6: $C := C \times T^{2^{L_i}} \bmod M$;
7: if $V_i \neq 0$ then $C := C \times T^{V_i} \bmod M$;
8: return C;
end algorithm.

2.3. Adaptive Window Methods

In adaptive methods (Kunihiro and Yamamoto, 2000) the computation depends on the input data, such as the exponent E. M-ary methods and window methods compute all possible partitions, knowing that the partitions of the actual exponent may or may not include all possible partitions. Thus, the number of modular multiplication in the pre-processing step can be reduced if partitions of E do not contain all possible windows.

Let $\wp(E)$ be the list of partitions obtained from the binary representation of E. Assume that the list of partition is non-redundant and ordered according to the ascending value of the partitions contained in the expansion of E. As before let p be the number of the partition of E and recall that V_i and L_i are the decimal value and the number of digits of partition P_i. The generic algorithm for describing the computation of $T^E \bmod M$ using the window methods is given in Algorithm 4.

Algorithm 4. *AdaptiveWindowMethod*(T, M, E)
1: Partition E using the given strategy;
2: for each partition in $\wp(E)$
3: Compute $T^{V_i} \bmod M$;
4: $C := T^{V_{b-1}} \bmod M$;
5: for i := p-2 downto 0
6: $C := T^{2^{L_i}} \bmod M$;
7: if $V_i \neq 0$ then $C := C \times T^{V_i} \bmod M$;
8: return C;
end algorithm.

In Algorithm 2 and Algorithm 3, it is clear how to perform the pre-computation indicated in lines 1 and 2. For instance, let E = 1011001101111000. The pre-processing step of the 4-ary method needs 14 modular multiplications ($T \rightarrow T \times T = T^2 \rightarrow T \times T^2 = T^3 \rightarrow \ldots \rightarrow T \times T^{14} = T^{15}$) and that of the maximum 4-digit sliding window method needs only 8 modular multiplications ($T \rightarrow T \times T = T^2 \rightarrow T \times T^2 = T^3 \rightarrow T^3 \times T^2 = T^5 \rightarrow T^5 \times T^2 = T^7 \rightarrow \ldots \rightarrow T^{13} \times T^2 = T^{15}$). However the adaptive 4-ary method would partition the exponent as E = [1011][0011][0111][1000] and hence needs to pre-compute the powers T^3, T^7, T^8 and T^{11} while the method maximum 4-digit sliding window method would partition the exponent as E = [1][0][11][00][11] [0][1111][000] and therefore needs to pre-compute the powers T^3 and T^{15}. The pre-computation of the powers needed by the adaptive 4-ary method may be done using 6 modular multiplications ($T \rightarrow T \times T = T^2 \rightarrow T \times T^2 = T^3 \rightarrow T^2 \times T^2 = T^4 \rightarrow T^3 \times T^4 = T^7 \rightarrow T^7 \times T = T^8 \rightarrow T^8 \times T^3 = T^{11}$) while the pre-computation of those powers necessary to apply the adaptive sliding window may be accomplished using 5 modular multiplications ($T \rightarrow T \times T = T^2 \rightarrow T \times T^2 = T^3 \rightarrow T^2 \times T^3 = T^5 \rightarrow T^5 \times T^5 = T^{10} \rightarrow T^5 \times T^{10} = T^{15}$). Algorithm 4 does not suggest how to compute the powers needed to use the adaptive window methods. Finding the best way to compute them is a *NP*-hard problem (Erdös, 1961), (Kunihiro and Yamamoto, 2000).

3. ADDITION CHAINS AND ADDITION SEQUENCES

An *addition chain* of length l for an positive integer N is a list of positive integers (a_0, a_1, a_2, ..., a_l) such that $a_0 = 1$, $a_l = N$ and $a_k = a_i + a_j$, $0 \leq i \leq j < k \leq l$. Finding a minimal addition chain for a given positive integer is an *NP*-hard problem. It is clear that a short addition chain for exponent E gives a fast algorithm to compute T^E mod M as we have if $a_k = a_i + a_j$ then $T^{a_k} = T^{a_i} \times T^{a_j}$. The adaptive window methods described earlier use a near optimal addition chain to compute T^E mod M. However these methods do not prescribe how to perform the pre-processing step (lines 1 and 2 of Algorithm 4). In the following we show how to perform this step with minimal number of modular multiplications.

3.1. Addition sequences

There is a generalisation of the concept of addition chains, which can be used to formalise the problem of finding a minimal sequence of powers that should be computed in the pre-processing step of the adaptive window method.

An *addition sequence* for the list of positive integers V_0, V_1, …, V_p such that $V_0 < V_1 < \ldots < V_p$ is an addition chain for integer V_p, which includes all the remaining integers V_0, V_1, …, V_p of the list. The length of an addition sequence is the numbers of integers that constitute the chain. An addition sequence for a list of positive integers V_0, V_1, …, V_p will be denoted by $\mathcal{S}(V_0, V_1, \ldots, V_p)$.

Hence, to optimise the number of modular multiplications needed in the pre-processing step of the adaptive window methods for computing T^E mod M, we need to find an addition sequence of minimal length (or simply minimal addition sequence) for the values of the partitions included in the non-redundant ordered list $\wp(E)$. This is an *NP*-hard problem and we use genetic algorithm to solve it. General principles of genetic algorithms are explained in the next section.

3.2. Brun's algorithm

Now we describe briefly, Brun's algorithm (Begeron, et. al., 1989) to compute relatively short addition sequences. The algorithm is a generalisation of the continued fraction algorithm (Begeron, et. al., 1989). Assume that we need to compute the addition sequence $\mathcal{S}(V_0, V_1, \ldots, V_p)$.

Let:

$$Q = \left\lfloor \frac{V_p}{V_{p-1}} \right\rfloor,$$

and let $\mathcal{B}(Q)$ be the addition chain for Q using the binary method (i.e. Algorithm 2 with $l = 1$). Let $R = V_p - Q \times V_{p-1}$. By induction we can construct an addition sequence $\mathcal{S}(V_0, V_1, \ldots, R, \ldots, V_{p-1})$. Then obtain

$$\mathcal{S}(V_0, V_1, \ldots, V_p) = \mathcal{S}(V_0, V_1, \ldots, R, \ldots, V_{p-1}) \cup V_{p-1} \times \mathcal{B}(Q) \backslash \{1\} \cup \{V_p\}$$

Example. Let us compute the addition sequence $\mathcal{S}(47, 117, 343)$ using Brun's algorithm. We proceed inductively as follows:

$$\begin{aligned}
\mathcal{S}(47, 117, 343) &= \mathcal{S}(47, 109, 117) \cup 117 \times \{2\} \cup \{343\} \\
&= \mathcal{S}(47, 109, 117) \cup \{234, 343\} \\
&= \mathcal{S}(8, 47, 109) \cup \{117, 234, 343\} \\
&= \mathcal{S}(8, 15, 47) \cup 47 \times \{2\} \cup \{109, 117, 234, 343\} \\
&= \mathcal{S}(8, 15, 47) \cup \{94, 109, 117, 234, 343\} \\
&= \mathcal{S}(8, 15, 17) \cup 15 \times \{2\} \cup \{47, 94, 109, 117, 234, 343\} \\
&= \mathcal{S}(8, 15, 17) \cup \{30, 47, 94, 109, 117, 234, 343\} \\
&= \mathcal{S}(2, 8, 15) \cup \{17, 30, 47, 94, 109, 117, 234, 343\} \\
&= \mathcal{S}(2, 7, 8) \cup \{15, 17, 30, 47, 94, 109, 117, 234, 343\} \\
&= \mathcal{S}(1, 2, 7) \cup \{8, 15, 17, 30, 47, 94, 109, 117, 234, 343\} \\
&= \mathcal{S}(1, 2) \cup 2 \times \{2, 3\} \cup \{8, 15, 17, 30, 47, 94, 109, 117, 234, 343\} \\
&= \{1, 2, 4, 6, 8, 15, 17, 30, \underline{47}, 94, 109, \underline{117}, 234, \underline{343}\}
\end{aligned}$$

So the Brun's addition sequence for 47, 117, 343 is (1, 2, 4, 6, 8, 15, 17, 30, 47, 94, 109, 117, 234, 343). This addition sequence allows us to perform the pre-computation step with 13 modular multiplications.

4. PRINCIPLES OF GENETIC ALGORITHMS

Genetic algorithms maintain a *population* of *individuals* that evolve according to *selection* rules and other *genetic operators*, such as *mutation* and *recombination*. Each individual receives a measure of *fitness*. *Selection* focuses on individuals, which shows high fitness. *Mutation* and *crossover* provide general heuristics that simulate the *reproduction* process. Those operators attempt to perturb the characteristics of the parent individuals as to generate *distinct* offspring individuals.

Genetic algorithms are implemented through the following generic algorithm described by Algorithm 5, wherein parameters *popSize*, *fit* and *genNum* are the population maximum size, the expected fitness of the expected individual and the maximum number of generation allowed respectively.

Algorithm 5. GA(popSize,fit,genNum):individual;

```
1:      generation  := 0;
2:      population  := initialPopulation();
3:      fitness := evaluate(population);
4:      do
5:          parents    := select(population);
```

```
6:          population := reproduce(parents);
7:          fitness    := evaluate(population);
8:          generation := generation + 1;
9:      while(fitness[i] < fit, ∀ i∈ population) and (generation < genNum);
10: return fittestIndividual(population);
end algorithm.
```

In Algorithm 5, function *intialPopulation* returns a valid random set of individuals that would compose the population of first generation, function *evaluate* returns the fitness of a given population storing the result into *fitness*. Function *select* chooses according to some criterion that privileges fitter individuals, the individuals that should be used to generate the population of the next generation and function *reproduction* implements the crossover and the mutation process to actually yield the new population.

5. APPLICATION TO ADDITION SEQUENCE MINIMISATION PROBLEM

It is perfectly clear that the shorter the addition sequence is, the faster Algorithm 4. We propose a novel idea based on genetic algorithm to solve this minimisation problem. The addition sequence minimisation problem consists of finding a sequence of numbers that constitutes an addition sequence for a given ordered list of n positive integers, say V_i for $0 \leq i \leq n-1$. The addition sequence should be of a minimal length.

5.1. Individual Encoding

Encoding of individuals is one of the implementation decisions one has to take in order to use genetic algorithms. It very depends on the nature of the problem to solve. There are several representations that have been used with success: *binary encoding* which is the most common mainly because it was used in the first works on genetic algorithms, represents an individual as a string of bits; *permutation encoding* mainly used in ordering problem, encodes an individual as a sequence of integer; *value encoding* represents an individual as a sequence of values that are some evaluation of some aspect of the problem (Michalewics, 1996), (Neves et. al., 1999).

In our implementation, an individual represents an addition sequence. We use the binary encoding wherein 1 implies that the entry number is a member of the addition sequence and 0 otherwise. Let V_1 = 3, V_2 = 7, V_3 = 9 and V_4 = 11, be the exponent, the encoding of Fig. 1 represents the addition sequence for sequence (3, 7, 9, 11):

1	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*
1	1	1	1	0	0	1	0	1	0	1

Figure 1. Addition sequence encoding

5.2. The genetic algorithm

Consider Algorithm 5. Besides the parameters *popSize*, *fit* and *genNum* which represent the population maximum size, the fitness of the expected result and the maximum number of generation allowed, the genetic algorithm has several other parameters, which can be adjust by the user so that the result is up to his or her expectation. The selection is performed using some *selection probabilities* and the reproduction, as it is subdivided into crossover and mutation processes, depends on the kind of crossover and the mutation rate and degree to be used.

Selection. The selection function as described in Algorithm 6, returns two populations: one represents the population of first parents, which is *parents*[1][] and the other consists of the

population of second parents, which is *parents*[2][].

Algorithm 6. select(population pop):population[]
```
1:     population[] parents[2];
2:     for i := 1 to popSize
3:         n1 := random(0,1);
4:         n2 := random(0,1);
5:         for j := 1 to popSize do
6:             parents[1][i] := parents[popSize];
7:             parents[2][i]= parents[popSize];
8:             if SelectionProbabilities[j] ≥ n1 then
9:                 parents[1][i] = pop[j];
10:        else if SelectProbabilities[j] ≥ n2 then
11:                parents[2][i]=pop[j];
12: return parents;
end algorithm.
```

The selection proceeds like this: whenever no individual that attends to the selection criteria is encountered, one of the last individuals of the population is then chosen, i.e. one of the fitter individuals of population. Note that the population from which the parents are selected is sorted in decreasing order with respect to the fitness of individuals, which will be described later on. The array *selectionProbabilities* is set up at initialisation step and privileges fitter individuals.

Reproduction. Given the parents populations, the reproduction proceeds using replacement as a reproduction scheme, i.e. offspring replace their parents in the next generation. Obtaining offspring that share some traits with their corresponding parents is performed by the *crossover* function. There are several *types* of crossover schemes. These will be presented shortly. The newly obtained population can then suffer some mutation, i.e. some of the individuals (addition chains) of some of the genes (power numbers). The crossover type, the number of individuals that should mutated and how far these individuals should be altered are set up during the initialisation process of the genetic algorithm.

Crossover. There are many ways how to perform crossover and these may depend on the individual encoding used (Michalewics, 1996). We present crossover techniques used with binary, permutation and value representations.

Single-point crossover consists of choosing randomly one *crossover point*, then, the part of the integer sequence from beginning of offspring till the crossover point is copied from one parent, the rest is copied from the second parent as depicted in Fig. 2(a).

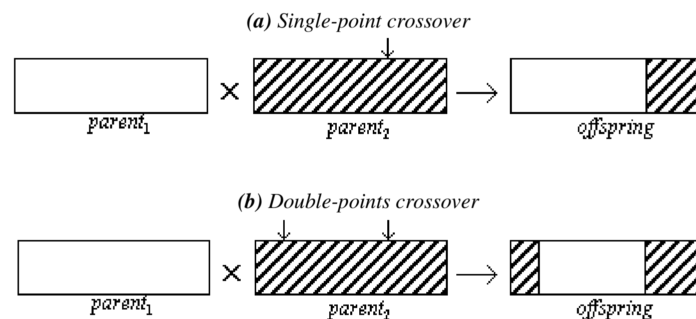

Figure 2. Single-point vs. double-points crossover

Double-points crossover consists of selecting randomly two *crossover points*, the part of the integer sequence from beginning of offspring to the first crossover point is copied from one parent, the part from the first to the second crossover point is copied from the second parent and the rest is copied from the first parent as depicted in Fig. 2(b). *Uniform crossover* copies integers randomly from the first or from the second parent. Finally, *arithmetic crossover* consists of applying some arithmetic operation to yield a new offspring.

The single point and two points crossover use randomly selected crossover points to allow variation in the generated offspring and to avoid premature convergence on a local optimum (DeJong and Spears, 1990), (Michalewics, 1996). In our implementation, we tested all four-crossover strategies.

Mutation. Mutation consists of changing some genes of some individuals of the current population. The number of individuals that should be mutated is given by the parameter *mutationRate* while the parameter *mutationDegree* states how many genes of a selected individual should be altered.

The mutation parameters have to be chosen carefully as if mutation occurs very often then the genetic algorithm would in fact change to *random search* (DeJong and Spears, 1990). Algorithm 7 describes the mutation procedure used in our genetic algorithm. When either of *mutationRate* or *mutationDegree* is null, the population is then kept unchanged, i.e. the population obtained from the crossover procedure represents actually the next generation population.

When mutation takes place, a number of genes are randomised and mutated: when the gene is 1 then it becomes 0 and vice-versa. The parameter *mutationDegree* indicates the number of gene to be mutated.

Algorithm 7. mutate(population pop, int mutationDegree,
int mutationRate):population;

```
1:    if (mutationRate ≠ 0)and(mutationDegree ≠ 0) then
2:       for a := 1 to PopSize do
3:             n := random(0,1);
4:             if n ≤ mutationRate then
5:                for I := 1 to mutationDegree do
6:                     gene := random(2, n - 1);
7:                     pop[a][gene] := (pop[a][gene] + 1) mod 2;
8:    return pop;
end algorithm.
```

Fitness. This step of the genetic algorithm allows us to classify the population so that fitter individuals are selected more often to contribute in the constitution of a new population.

The fitness evaluation of addition chain is done with respect to two aspects: *(i)* how much a given addition chain adheres to the Definition 1, i.e. how many members of the addition chain cannot be obtained summing up two previous members of the chain; *(ii)* how far the addition chain is reduced, i.e. what is the length of the addition chain. Algorithm 8 describes the evaluation of fitness used in our genetic algorithm.

For a valid addition sequence, the fitness function returns its length, which is smaller than the last integer V_n. The evolutionary process attempts to minimise the number of ones in a valid addition sequence and so minimise the corresponding length. Individuals with fitness larger or equal to V_n are invalid addition chains. The constant *largePenalty* should be larger than V_n. With well-chosen parameters, the genetic algorithm deals only with valid addition sequences.

Algorithm 8. evaluate(individual s): int;

```
1:      int fitness = 0;
2:      for i := 2 to n-1 do
3:          if s[i] = 1 then
4:                  fitness := fitness + 1;
5:          if i = V_i & s[i] ≠ 1 then
6:              fitness = fitness + largePenalty;
7:          if ∄j, k| 1 ≤ j ≤ i & 1 ≤ k ≤ i & i = j+k & s[i] = s[k] = 1 then
8:              fitness := fitness + largePenalty;
9:      return f;
end algorithm.
```

6. IMPLEMENTATION RESULTS

In applications of genetic algorithms to a practical problem, it is difficult to predict a priori what combination of settings will produce the best result for the problem in a relatively short time. The settings consist of the population size, the crossover type, the mutation rate and the mutation degree. We investigated the impact of different values of these parameters in order to choose the more adequate ones to use. We found out that the ideal parameters are: a population of at most 50 individuals; the double-points crossover; a mutation rate between 0.4 and 0.7 and a mutation degree of about 1% of the value of the last value in sequence V_p.

The curve of Fig. 3 shows the progress made in the first 500 generations of an execution to obtain the addition sequence for the list of numbers (47, 117, 343). The settings used are: a 50 individual per population, double-points crossover, a mutation rate of 0.645 and a degree of 50.

The Brun's algorithm yields (1, 2, 4, 6, 8, 15, 17, 30, 47, 94, 109, 117, 234, 343) and the genetic algorithm yield the same addition sequence as well as (1, 2, 4, 8, 11, 18, 36, 47, 55, 91, 109, 117, 226, 343). Both addition sequences have the same length. They allow performing the pre-computation step with 13 modular multiplications.

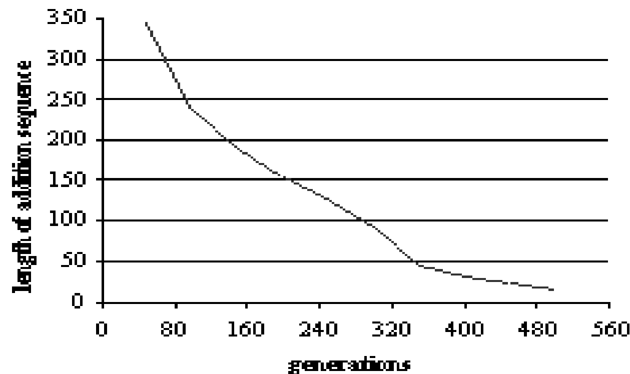

Figure 3. The genetic algorithm result curve for the parameters given above.

Finding the best addition sequence is impractical. However, we can find near-optimal ones. Our genetic algorithm always finds addition sequences far shorter than those used by the *m*-ary method independently of the value of *m* and by the sliding windows independently of the partition strategy used, and as short as the addition sequence yield by the Brun's algorithm. Table 1 shows some examples for lists (5, 9, 23), (9, 27, 55) and (5, 7, 95).

Table 1. The addition sequences yield by the exposed methods (*m*-ary, window, and Brun's method) vs. the genetic algorithm for lists (5, 9, 23), (9, 27, 55) and (5, 7, 95) together with the number of necessary modular multiplications.

V_i	*Method*	*Addition sequence*	*#Mult*
	5-ary	(1, 2, 3, 4, 5, 6, 7, 8, 9, …, 22, 23, …, 30, 31)	30
5,	5-window	(1, 2, 3, 5, 7, 9, 11, …, 31)	16
9,	Brun's	(1, 2, 4, 5, 9, 18, 23)	6
23	Genetic algorithm	(1, 2, 4, 5, 9, 18, 23)	6
		(1, 2, 4, 5, 9, 14, 23)	6
9,	6-ary	(1, 2, 3, …, 8, 9, …, 26, 27, …, 54, 55, …, 63)	62
27,	6-window	(1, 2, 3, …, 7, 9, …, 25, 27, …, 53, 55, …, 63)	31
55	Brun's	(1, 2, 3, 6, 9, 18, 27, 54, 55)	8
	Genetic algorithm	(1, 2, 4, 8, 9, 18, 27, 28, 55)	8
		(1, 2, 3, 6, 9, 18, 27, 54, 55)	8
5,	7-ary	(1, 2, 3, 4, 5, 6, 7, …, 95)	94
7,	7-window	(1, 2, 3, 5, 7, …, 95)	43
95	Brun's	(1, 2, 4, 5, 7, 14, 21, 42, 84, 91, 95)	10
	Genetic algorithm	(1, 2, 3, 5, 7, 10, 20, 30, 35, 65, 95)	10
		(1, 2, 4, 5, 7, 14, 21, 42, 84, 91, 95)	10

7. CONCLUSIONS

In this paper, we presented an application of genetic algorithms to minimisation of addition sequences. We first explained how individuals are encoded. Then we described the necessary algorithmic solution. Then we presented some empirical observations about the performance of the genetic algorithm implementation.

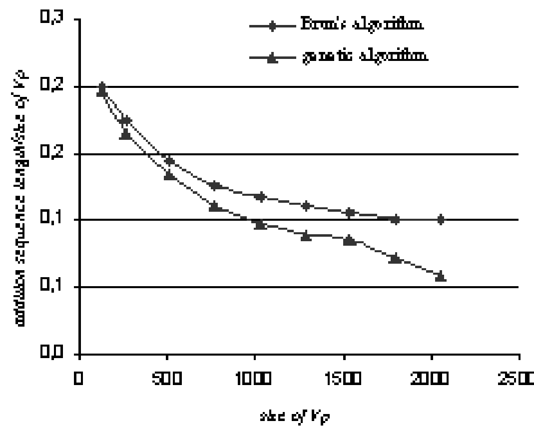

Figure 4. Ratio for the addition sequences yield by the GA vs. *m*-ary, sliding window and Brun's methods.

This application of genetic algorithms to the minimisation problem proved to be very useful and effective technique. Shorter addition sequences compared with those obtained by the *m*-

ary methods, those obtained for the sliding window methods as well as those obtained using Brun's algorithm (see Table 1 of the previous section) can be obtained with a little computational effort. A comparison of the performance of the *m*-ary, sliding window and the Brun's method vs. the genetic algorithm is shown in Fig. 4. A satisfactory addition sequence can be obtained in a 7 seconds to 4 minutes using a Pentium III with a 256 MB of RAM.

REFERENCES

Begeron, R. Berstel, J, Brlek, S. and Duboc, C. (1989), Addition chains using continued fractions, *Journal of Algorithms*, no. 10, pp. 403-412.

DeJong, K. and Spears, W.M. (1990), An analysis of the interacting roles of the population size and crossover type in genetic algorithms, In *Parallel problem solving from nature*, Springer-Verlag, pp. 38-47.

DeJong, K. and Spears, W.M. (1989), Using genetic algorithms to solve NP-complete problems, In Proceedings of *the Third International Conference on Genetic Algorithms*, Morgan Kaufmann, pp. 124-132.

Erdös, P. (1960), Remarks on number theory III: On addition chain, *Acta Arithmetica*, pp 77-81.

Haupt, R.L. and Haupt, S.E. (1998), *Practical genetic algorithms*, John Wiley and Sons, New York.

Knuth, D.E. (1981), *The Art of Programming: Seminumerical Algorithms*, vol. 2. Reading, MA: Addison_Wesley, Second edition.

Koç, Ç.K. (1994), High-speed RSA Implementation, *Technical report*, RSA Laboratories, Redwood City, califirnia, USA.

Kunihiro, N. and Yamamoto, H. (2000), New methods for generating short addition chain, *IEICE Transactions*, vol. E83-A, no. 1, pp. 60-67.

Michalewics, Z. (1996), *Genetic algorithms + data structures = evolution program*, Springer-Verlag, USA, third edition.

Menezes, A.J. (1993), *Elliptic curve public key cryptosystems*, Kluwer Academic.

Neves, J., Rocha, M., Rodrigues, Biscaia, M. and Alves, J. (1999), Adaptive strategies and the design evolutionary applications, In Proceedings of *the Genetic and the Design of Evolutionary Computation Conference*, Orlando, Florida, USA.

Rivest, R.L., Shamir, A. and Adleman, L. (1978), A method for obtaining digital signature and public-key cryptosystems, *Communication of ACM*, vol. 21, no.2, pp. 120-126.

Received: 30 May 2003
Accepted in final form: 30 March 2004 after one revision

About the authors:

Nadia Nedjah and *Luiza de Macedo Mourelle* are researchers at the Department of Systems Engineering and Computation, Faculty of Engineering, State University of Rio de Janeiro, Rio de Janeiro, Brazil. They can be contacted by email at {nadia, ldmm}@eng.uerj.br