Efficient Pre-Processing for Large Window-Based Modular Exponentiation Using Ant Colony

Nadia Nedjah
Department of Electronics Engineering and Telecommunications,
Faculty of Engineering, State University of Rio de Janeiro, Brazil
nadia@eng.uerj.br

Luiza de Macedo Mourelle
Department of Systems Engineering and Computation,
Faculty of Engineering, State University of Rio de Janeiro, Brazil
ldmm@eng.uerj.br

Keywords: ant colony, addition chain, cryptosystem, modular exponentiation

Received: September 30, 2004

Modular exponentiation is the main operation to RSA-based public-key cryptosystems. It is performed using successive modular multiplications. This operation is time consuming for large operands, which is always the case in cryptography. For software or hardware fast cryptosystems, one needs thus reducing the total number of modular multiplications required. Existing methods attempt to reduce this number by partitioning the exponent in constant or variable size windows. However, these window-based methods require some pre-computations, which themselves consist of modular exponentiations. It is clear that pre-processing needs to be performed efficiently also. In this paper, we exploit the ant colony strategy to finding an optimal addition sequence that allows one to perform the pre-computations in window-based methods with a minimal number of modular multiplications. Hence we improve the efficiency of modular exponentiation. We compare the yielded addition sequences with those obtained using Brun's algorithm.

Povzetek: Metoda kolonij mravelj je uporabljena za kriptografske probleme.

1 Introduction

Public-key cryptographic systems (such as the RSA encryption scheme [6], [12]) often involve raising large elements of some groups fields (such as GF(2^n) or elliptic curves [9]) 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: $T \rightarrow T^2 \rightarrow \ldots \rightarrow T^{E-1} \rightarrow T^E$.

Algorithm 1. simpleExponentiation(T, M, E)
1. $C := T$;
2. for $i := 1$ to $E - 1$ do $C := (C \times T) \bmod M$;
3. return C;
end algorithm.

The computation of exponentiations using Algorithm 1 is very inefficient. The problem of yielding the power of a number using a minimal number of multiplications is NP-hard [5], [10]. There are several efficient algorithms that perform exponentiation with a nearly minimal number of modular multiplications, such that the window-based methods. However, these methods 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 the ant colony methodology. We compare our results with those obtained using the Brun's algorithm [1].

Ant systems [2-1] are distributed multi-agent systems [3-1] that simulate real ant colony. Each agent behaves as an ant within its colony. Despite the fact that ants have very bad vision, they always are capable to find the shortest path from their nest to wherever the food is. To do so, ants deposit a trail of a chemical substance called *pheromone* on the path they use to reach the food. On intersection points, ants tend to choose a path with high amount of pheromone. Clearly, the ants that travel through the shorter path are ca-

pable to return quicker and so the pheromone deposited on that path increases relatively faster than that deposited on much longer alternative paths. Consequently, all the ants of the colony end using the shorter way.

In this paper, we exploit the ant colony methodology to obtain an optimal solution to AS-chain minimisation *NP*-complete problem. In order to clearly report the research work performed, we subdivide the rest of this paper into five important sections. In Section 2, we present the window methods; In Section 3, we present the concepts of addition chains and sequence and they can be used to improve the pre-computations of the window methods; In Section 4, we give an overview on ant colony concepts; In Section 5, we explain how these concepts can be used to compute a minimal addition chain to perform efficiently necessary pre-computations in the window methods. In Section 6, we present some useful results.

2 Window-Based Methods

Generally speaking, the window methods for exponentiation [5] may be thought of as a three major step procedure:

1. partitioning in k-bits windows the binary representation of the exponent E;
2. pre-computing the powers in each window one by one;
3. iterating the squaring of the partial result k times to shift it over, and then multiplying it by the power in the next window when if window is not 0.

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 [3] scans the digits of E form the less significant to the most significant digit and groups them into partitions of equal length $\log_2 m$, where m is a power of two. Note that 1-ary methods coincides with the square-and- multiply well-known binary exponentiation method.

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 $\mathbb{P}(E)$. 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 Compute $T^i \bmod M$;
3. $C := T^{V_p} \bmod M$;
4. for $i := p - 2$ downto 0
5. $\quad C := C^{2^l} \bmod M$;
6. $\quad$ if $V_i \neq 0$ then $C := C\times \bmod M$;
7. 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, thus 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 do
3. $\quad$ Compute $T^i \bmod M$;
4. $C := T^{V_{p-1}} \bmod M$;
5. for $i := p - 2$ downto 0 do
6. $\quad C := C^{L_i} \bmod M$;
7. $\quad$ if $V_i \neq 0$ then $C := C \times T^{V_i} \bmod M$;
8. return C;

end algorithm.

In adaptive methods [7] the computation depends on the input data, such as the exponent E. M-ary methods and window methods pre-compute powers of all possible partitions, not taking into account that the partitions of the actual exponent may or may not include all possible partitions. Thus, the number of modular multiplications in the pre-processing step can be reduced if partitions of E do not contain all possible ones.

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 decimal value of the partitions contained in the expansion of E. 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.

In Algorithm 2 and Algorithm 3, it is clear how to perform the pre-computation indicated in lines 2–3. For instance, let E = 1011001101111000. The pre-processing

step of the 4-ary method needs 14 modular multiplications ($T \rightarrow T \times T = T^2 \rightarrow T \times T^2 = T^3 \rightarrow \ \rightarrow T \times T^{14} = T^{15}$) and that of the maximum 4-digit sliding window method needs only 8 modular multiplications ($T \rightarrow T \times T = T^2 \rightarrow T \times T^2 = T^3 \rightarrow T^3 \times T^2 = T^5 \rightarrow T^5 \times T^2 = T^7 \rightarrow \ \rightarrow T^{13} \times T^2 = T^{15}$). 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-digit sliding window 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 $T \rightarrow T \times T = T^2 \rightarrow T \times T^2 = T^3 \rightarrow T^2 \times T^3 = T^5 \rightarrow T^5 \times T^5 = T^{10} \rightarrow T^5 \times T^{10} = T^{15}$. Note that Algorithm 4 does not suggest how to compute the powers (lines 2–3) needed to use the adaptive window methods. Finding the best way to compute them is a NP-hard problem [4], [7].

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

end algorithm.

3 Addition Chains and Sequences

An *addition chain* of length l for an positive integer N is a list of positive integers $(E_1, E_2, \ldots, E_l)$ such that $E_1 = 1$, $E_l = N$ and $E_k = E_i + E_j$, $0 \leq i \leq j < k \leq l$. 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 $T^E \bmod M$ as we have if $E_k = E_i + E_j$ then $T^{E_k} = T^{E_i} \times T^{E_j}$. The adaptive window methods described earlier use a near optimal addition chain to compute $T^E \bmod M$. However these methods do not prescribe how to perform the pre-processing step (Line 3 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 $V_1, V_2, \ldots, V_p$ such that $V_1 < V_2 < \ldots < V_p$ is an addition chain for integer V_p that includes all the integers $V_1, V_2, \ldots, V_p$. The length of an addition sequence is the numbers of integers that constitute the chain. An addition sequence for a list of positive integers $V_1, V_2, \ldots, V_p$ will be denoted by $\xi(V_1, V_2, \ldots, V_p)$.

Hence, to optimise the number of modular required multiplications in the pre-processing step of the adaptive window methods for computing $T^E \bmod M$, 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. Our method showed to be very effective for large window size. General principles of genetic algorithms are explained in the next section.

3.2 Brun's algorithm

Now we describe briefly, Brun's algorithm [1] to compute addition sequences. The algorithm is a generalisation of the continued fraction algorithm [1]. Assume that we need to compute the addition sequence $\xi(V_1, V_2, \ldots, V_p)$. Let $Q = \lfloor \frac{V_p}{V_{p-1}} \rfloor$ and let $\chi(Q)$ be the addition chain for Q using the binary method (i.e. that used in Algorithm 2 with l = 1). Let $R = V_p - Q \times V_{p-1}$. By induction we can construct an addition sequence $\xi(V_1, V_2, \ldots, R, \ldots, V_{p-1})$, then obtain:

$$\begin{aligned} \xi(S) = \quad & \xi(V_1, V_2, \ldots, R, \ldots, V_{p-1}) \cup \\ & V_{p-1} \times \chi(Q) \setminus \{1\} \cup \{V_p\}, \\ S = \quad & V_1, V_2, \ldots, V_p \end{aligned} \tag{1}$$

4 Ant Systems and Algorithms

Ant systems can be viewed as multi-agent systems [3] that use a shared memory through which they communicate and a local memory to bookkeep the locally reached problem solution. Fig. 1. depicts the overall structure of an system, wherein A_i and LM_i represent the $i^{th.}$ agent of the ant system and its local memory respectively. Mainly, the shared memory (*SM*) holds the pheromone information while the local memory LM_i keeps the solution (possibly partial) that agent A_i reached so far.

The behaviour of an artificial ant colony is summarised in Algorithm 5, wherein N, C, SM are the number of artificial ant that form the colony, the characteristics of the expected solution and the shared memory used by the artificial ants to store pheromone information respectively.

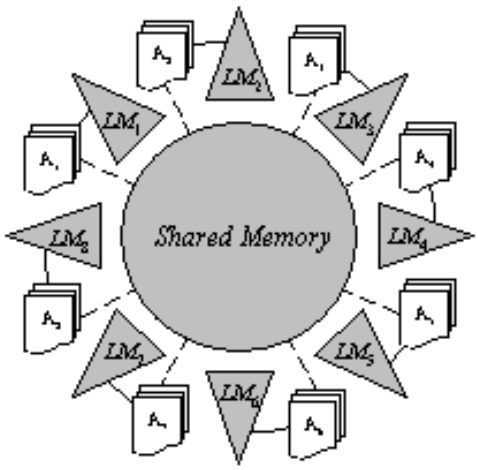

Figure 1: Multi-agent system architecture

The first step consists of activating N distinct artificial ants that should work in simultaneously. Every time an ant conclude its search, the shared memory is updated with an amount of pheromone, which should be proportional to the quality of the reached solution. This called *global* pheromone update. When the solution yield by an ant's work is suitable (i.e. fits characteristics C) then all the active ants are stopped. Otherwise, the process is iterated until an adequate solution is encountered.

Algorithm 5. ArtificialAntColony(N, C)
1. Initialise SM with initial pheromone;
2. do
3. for $i := 1$ to N
4. Start ArtificialAnt(A_i, LM_i);
5. $Active := Active \cup \{A_i\}$;
6. do
7. Update SM (pheromone evaporation);
8. when an ant (say A_i) halts do
9. $Active := Active \setminus \{A_i\}$;
10. Φ := Pheromone(LM_i);
11. Update SM (global pheromone Φ);
12. S := ExtractSolution(LM_i);
13. until Fitness$(S) = C$ or $Active = \emptyset$;
14. while $Active \neq \emptyset$ do
15. Stop ant $A_i \mid A_i \in Active$;
16. $Active := Active \setminus \{A_i\}$;
17. until Fitness$(S) = C$;
18. return S;
end.

The behaviour of an artificial ant is described in Algorithm 6, wherein A_i and LM_i represent the ant identifier and the ant local memory, in which it stores the solution computed so far. First, the ant computes the probabilities that it uses to select the next state to move to. The computation depends on the solution built so far, the problem constraints as well as some heuristics [2], [6]. Thereafter, the ant updates the solution stored in its local memory, deposits some *local* pheromone into the shared memory then moves to the chosen state. This process is iterated until complete problem solution is yielded.

Algorithm 6. ArtificialAnt(A_i, LM_i)
1. Initialise LM_i;
2. do
3. P := TransitionProbabilities(LM_i);
4. $NextState$:= StateDecision(LM_i, P);
5. Update LM_i;
6. Update SM with local pheromone;
7. $CurrentState := NextState$);
8. until $CurrentState := TargetState$;
9. Halt A_i;
end.

5 Chain Sequence Minimisation Using Ant System

In this section, we concentrate on the specialisation of the ant system of Algorithm 4 and Algorithm 5 to the addition sequence minimisation problem. For this purpose, we describe how the shared and local memories are represented. We then detail the function that yields the solution (possibly partial) characteristics. Thereafter, we define the amount of pheromone to be deposited with respect to the solution obtained so far. Finally, we show how to compute the necessary probabilities and make the adequate decision towards a shorter addition sequence for the considered the sequence $(V_1, V_2, \ldots, V_p)$.

5.1 The Ant System Shared Memory

The ant system shared memory is a two-dimension array. If the last exponent in the sequence is V_p then the array should V_p rows. The number of columns depends on the row. It can be computed as in Eq. 2, wherein NC_i denotes the number of columns in row i.

$$NC_i = \begin{cases} 2^{i-1} - i + 1 & \text{if } 2^{i-1} < V_p \\ 1 & \text{if } i = V_p \\ V_p - i + 3 & \text{otherwise} \end{cases} \quad (2)$$

An entry $SM_{i,j}$ of the shared memory holds the pheromone deposited by ants that used exponent $i+j$ as the i th. member in the built addition sequence. Note that $1 \leq i \leq V_p$ and for row i, $0 \leq j \leq NC_i$. Fig. 2 gives an example of the shared memory for exponent 17. In this example, a table entry is set to show the exponent corresponding to it. The exponent $E_{i,j}$ corresponding to entry $SM_{i,j}$ should be obtainable from exponents of previous rows. Eq. 3 formalises such a requirement.

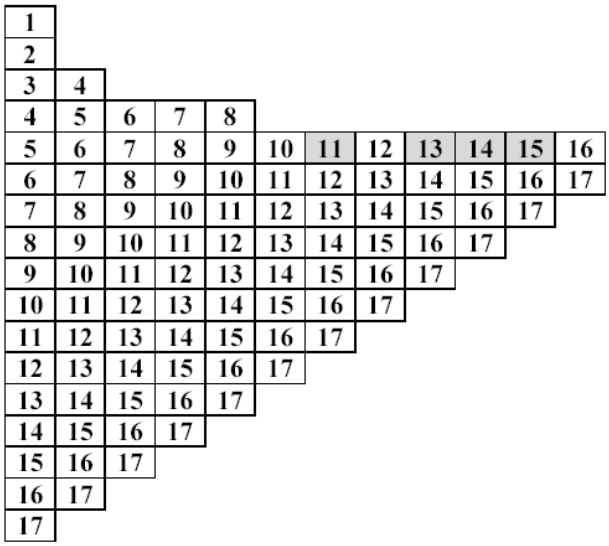

Figure 2: Example of shared memory content for $V_p = 17$

$$E_{i,j} = E_{k_1,l_1} + E_{k_2,k_2} | \quad \begin{array}{l} 1 \leq k_1, k_2 < i, \\ 0 \leq l_1, l_2 \leq j, \\ k_1 = k_2 \Longleftrightarrow l_1 = l_2 \end{array} \tag{3}$$

Note that, in Fig. 2, the exponents in the shaded entries are not valid exponents as for instance exponent 7 of row 4 can is not obtainable from the sum of two previous different stages, as described in Eq. 3. The computational process that allows us to avoid these exponents is of very high cost. In order to avoid using these few exponents, we will penalise those ants that use them and hopefully, the solutions built by the ants will be almost all valid addition chains. Furthermore, note that for a valid solution need also to contain all the exponents of the sequence i.e., $V_1, V_2, \ldots, V_{p-1}, V_p$.

5.2 The Ant Local Memory

In an ant system, each ant is endowed a local memory that allows it to store the solution or the part of it that was built so far. This local memory is divided into two parts: the first part represents the (partial) addition sequence found by the ant so far and consists of a one-dimension array of V_p entries; the second part holds the *characteristic* of the solution. It represents the solution fitness i.e., its length. The details of how to compute the fitness of a possibly partial addition sequence are given in the next section. Fig. 3 shows six different examples of an ant local memory for sequence (5, 7, 11). Fig. 3(a) represents addition sequence (1, 2, 4, 5, 7, 11), which is a valid and complete solution of fitness 5. Fig. 3(b) depicts addition sequence (1, 2, 3, 5, 7, 10, 11), which is also a valid and complete solution but of fitness 6. Fig. 3(c) represents partial addition sequence (1, 2, 4, 5), which is a valid and but incomplete solution as it does not include exponent 7 and 11 and the last exponent

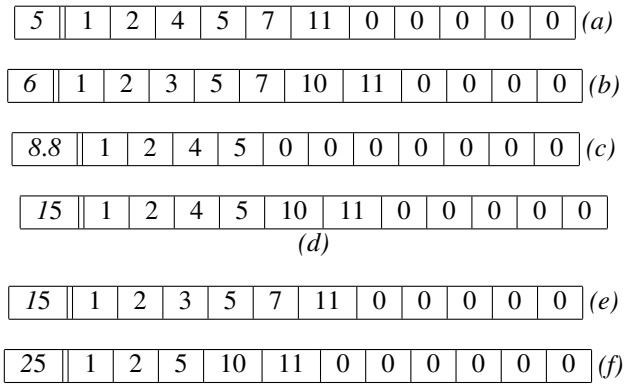

Figure 3: Example of an ant local memory

is smaller than both 7 and 11. The corresponding fitness is 8.8. Fig. 3(d) consists of non-valid addition sequence (1, 2, 4, 5, 10, 11) as 7 is not included. The corresponding fitness is 15. Fig. 3(e) represents also non-valid addition sequence (1, 2, 3, 5, 7, 11) as 11 is not a sum two previous exponents in the sequence. Its fitness is also 15. Finally, Fig. 3(f) represents also non-valid addition sequence (1, 2, 5, 10, 11) as 5 is not a sum two previous and mandatory exponent 7 is not in the addition sequence. exponents in the sequence. Its fitness is also 25. In next section, we explain how the fitness of a solution is computed.

5.3 Addition Sequence Characteristics

The fitness evaluation of an addition sequence is performed with respect to three aspects: *(a)* how much it adheres to the definition (see Section 3), i.e. how many of its members cannot be obtained summing up two previous members of the sequence; *(b)* how far the it is reduced, i.e. what is the length of the chain; *(c)* how many of the mandatory exponents do not appear in the sequence. Eq. 4 shows how to compute the fitness f of solution $\alpha = (E_1, E_2, \ldots, E_n, 0, \ldots, 0)$ regarding mandatory exponents $\sigma = V_1, V_2, \ldots, V_p$.

$$\begin{array}{ll} f(S,A) = & \frac{V_p \times (n-1)}{E_n} + \psi \times (\eta_1 + \eta_2) \\ \sigma = & V_1, V_2, \ldots, V_p \\ \alpha = & V_1, V_2, \ldots, V_p \end{array} \tag{4}$$

wherein ψ is a penalty, η_1 represents the number of E_i, $3 \leq i \leq n$ in the addition sequence that verify the predicate below:

$$\forall j, k \mid \ 1 \leq j, k < i, E_i \neq E_j + E_k \tag{5}$$

and η_2 represents the number of mandatory exponents V_i, $1 \leq i \leq p$ that verify the predicate below:

$$V_i \leq E_n \Longrightarrow \forall j \mid \ 1 \leq j \leq n, E_j \neq V_i \tag{6}$$

For a valid complete addition sequence, the fitness coincides with its length, which is the number of multiplica-

tions that are required to compute the exponentiation using the sequence. For a valid but incomplete addition sequence, the fitness consists of its *relative* length. It takes into account the distance between last mandatory exponent V_p and the last exponent in the partial addition sequence. Furthermore, for every mandatory exponent that is smaller than the last member of the sequence which is not part of it, a penalty is added to the sequence fitness. Note that valid incomplete sequences may have the same fitness of some other valid and complete ones. For instance, addition sequence (1, 2, 3, 6, 8) and (1, 2, 3, 6) for exponent mandatory exponents (3, 6, 8) have the same fitness 4.

For an invalid addition sequences, a penalty, which should be larger than V_p, is introduced into the fitness value for each exponent for which one cannot find two (may be equal) members of the sequence whose sum is equal to the exponent in question or two distinct previous members of the chain whose difference is equal to the considered exponent. Furthermore, a penalty is added to the fitness of a addition sequence whenever the a mandatory exponent is not part of it. The penalty used in the examples of Fig. 3 is 10.

5.4 Pheromone Trail and State Transition Function

There are three situations wherein the pheromone trail is updated: *(a)* when an ant chooses to use exponent $F = i+j$ as the *i*th. member in its solution, the shared memory cell $SM_{i,j}$ is incremented with a constant value of pheromone $\Delta\phi$, as in the first assignment of Eq. 7; *(b)* when an ant halts because it reached a complete solution, say $\alpha = (E_1, E_2, \ldots, E_n)$ for mandatory exponent sequence σ, all the shared memory cells $SM_{i,j}$ such that $i+j = E_i$ are incremented with pheromone value of $1/Fitness(\sigma, \alpha)$, as in the second Eq. 7. Note that the better is the reached solution, the higher is the amount of pheromone deposited in the shared memory cells that correspond to the addition sequence members. *(iii)* The pheromone deposited should evaporate. Periodically, the pheromone amount stored in $SM_{i,j}$ is decremented in an exponential manner [6] as in the third assignment of Eq. 7.

$$\begin{aligned} SM_{i,j} &:= SM_{i,j} + \Delta\phi, \text{ whenever } E_i = i + j \\ SM_{i,j} &:= SM_{i,j} + 1/f(\sigma, \alpha), \forall i, j \mid i + j = E_i \\ SM_{i,j} &:= (1 - \rho)SM_{i,j} \mid \rho \in (0, 1], \text{ periodically} \end{aligned} \quad (7)$$

An ant, say A that has constructed partial addition sequence $(E_1, E_2, \ldots, E_i, 0, \ldots, 0)$ for exponent sequence $(V_1, V_2, \ldots, V_p)$, is said to be in *step i*. In step $i + 1$, it may choose exponent E_{i+1} $E_i + 1$, $E_i + 2$, …, $2E_i$, if $2E_i \leq V_p$. That is, ant A may choose one of the exponents that are associated with the shared memory cells SM_{i+1,E_i-i}, SM_{i+1,E_i-i+1}, …, $SM_{i+1,2E_i-i-1}$. Otherwise (i.e. if $2E_i > V_p$), it may only select from exponents $E_i + 1$, $E_i + 2$, …, $E + 2$. In this case, ant A may choose one of the exponent associated with $SM_{i+1,E_i-i}, SM_{i+1,E_i-i+1}, \ldots, SM_{i+1,E-i+1}$. Furthermore, ant A chooses the new exponent E_{i+1} with the probability expressed through Eq. 8 below.

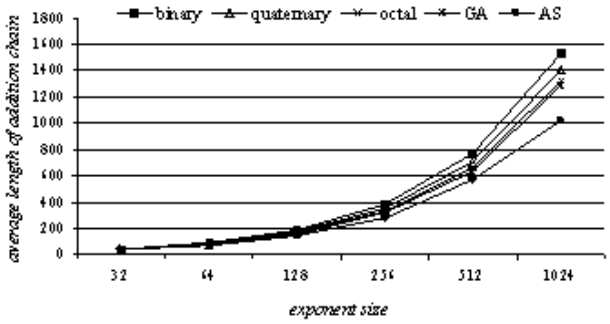

Figure 4: Comparison of the average length of the addition chains

$$P_{i,j} = \begin{cases} \dfrac{SM_{i+1,j}}{\max\limits_{k=E_i-i}^{2E_i-i-1} SM_{i+1,k}} & \text{if } 2E_i \leq E \;\&\; j \in [E_i - i, 2E_i - i - 1] \\ \dfrac{SM_{i+1,j}}{\max\limits_{k=E_i-i}^{E-i-1} SM_{i+1,k}} & \text{if } 2E_i > E \;\&\; j \in [E_i - i, E - i - 1] \\ 0 & \text{otherwise} \end{cases} \quad (8)$$

6 Performance Comparison

The ant system described in Algorithm 5 and Algorithm 6 was implemented using Java as a multi-threaded ant system. Each ant was simulated by a thread that implements the artificial ant computation of Algorithm 4. A Pentium IV-HT™ of a operation frequency of 1GH and RAM size of 2GB was used to run the ant system and obtain the performance results.

We compared the performance of m-ary methods, the Brun's algorithm, genetic algorithms and ant system-based methods. The obtained addition chains are given in Table 1.

The average lengths of the addition sequences for different exponent sequences obtained using these methods are given in Table 2. The exponent size is that of its binary representation (i.e. number of bits). The ant system-based method always outperforms all the others, including the genetic algorithm-based method [7]. The chart of Fig. 4 shows the relation between the average length of the obtained addition sequences.

Table 1: The addition sequences yield for $\xi(5,9,23)$, $\xi(9,27,55)$ and $\xi(5,7,95)$ respectively

Method	Addition sequence	$\#\times$
5-ary	(1,2,3,. . .,30,31)	30
5-window	(1,2,3,5,7,9,11,. . .,31)	16
Brun's	(1,2,4,5,9,18,23)	6
GAs	(1,2,4,5,9,18,23)	6
Ant system	(1,2,4,5,9,14,23)	6
6-ary	(1,2,3,. . .,. . .,63)	62
6-window	(1,2,3,5,7,. . .,63)	31
Brun's	(1,2,3,6,9,18,27,54,55)	8
GAs	(1,2,4,8,9,18,27,28,55)	8
Ant system	(1,2,4,5,9,18,27,54,55)	8
7-ary	(1,2,3,4,5,6,7,. . .,95)	94
7-window	(1,2,3,5,7,. . .,95)	43
Brun's	(1,2,4,5,7,14,21,42,84,91,95)	10
GAs	(1,2,3,5,7,10,20,30,35,65,95)	10
Ant system	(1,2,4,5,7,14,19,38,76,95)	9
7-ary	(1,2,3,4,5,6,7,. . .,95)	94
7-window	(1,2,3,5,7,. . .,95)	43
Brun's	(1,2,4,5,7,14,21,42,84,91,95)	10
GAs	(1,2,3,5,7,10,20,30,35,65,95)	10
Ant system	(1,2,4,5,7,14,19,38,76,95)	9

Table 2: Average length of addition sequence for Brun's algorithm (BA), genetic algorithms (GA) and ant system (AS)

$\lvert V_p\rvert$	BA	GA	AS
32	41	42	45
64	84	85	86
128	169	170	168
256	340	341	331
512	681	682	658
1024	1364	1365	1313

7 Conclusion

In this paper we applied the methodology of ant colony to the addition chain minimisation problem. Namely, we described how the shared and local memories are represented. We detailed the function that computes the solution fitness. We defined the amount of pheromone to be deposited with respect to the solution obtained by an ant. We showed how to compute the necessary probabilities and make the adequate decision towards a good addition chain for the considered exponent.

Furthermore, we implemented the ant system described using multi-threading (each ant of the system was implemented by a thread). We compared the results obtained by the ant system to those of *m*-ary methods (binary, quaternary and octal methods). Taking advantage of the a previous work on evolving minimal addition chains with genetic algorithm, we also compared the obtained results to those obtained by the genetic algorithm. The ant system always finds a shorter addition chain and gain increases with the size of the exponents.

References

[1] Rivest, R., Shamir, A. and Adleman, L., A method for Obtaining Digital Signature and Public-Key Cryptosystems, Communications of the ACM, 21:120-126, 1978.

[2] Dorigo, M. and Gambardella, L.M., Ant Colony: a Cooperative Learning Approach to the Travelling Salesman Problem, IEEE Transaction on Evolutionary Computation, Vol. 1, No. 1, pp. 53-66, 1997.

[3] Feber, J., Multi-Agent Systems: an Introduction to Distributed Artificial Intelligence, Addison-Wesley, 1995.

[4] Downing, P. Leong B. and Sthi, R., Computing Sequences with Addition Chains, SIAM Journal on Computing, vol. 10, No. 3, pp. 638-646, 1981.

[5] Nedjah, N., Mourelle, L.M., Efficient Parallel Modular Exponentiation Algorithm, Second International Conference on Information systems, ADVIS'2002, Izmir, Turkey, Lecture Notes in Computer Science, Springer-Verlag, vol. 2457, pp. 405-414, 2002.

[6] Stutzle, T. and Dorigo, M., ACO Algorithms for the Travelling Salesman Problems, Evolutionary Algorithms in Engineering and Computer Science, John-Wiley & Sons, 1999.

[7] Nedjah, N. and Mourelle, L.M., Minimal addition-subtraction chains using genetic algorithms, Proceedings of the Second International Conference on Information Systems, Izmir, Turkey, Lecture Notes in Computer Science, Springer-Verlag, vol. 2457, pp. 303-313, 2002.