Solving the factorization problem with P systems*

Alberto Leporati**, Claudio Zandron and Giancarlo Mauri

(Computer Science Department, University of Milan, 20126 Milano, Italy)

Abstract P systems have been used many times to face with computationally difficult problems, such as NP-complete decision problems and NP-hard optimization problems. In this paper we focus our attention on another computationally intractable problem: factorization. In particular, we first propose a simple method to encode binary numbers using multisets. Then, we describe three families of P systems: the first two allow to add and to multiply two binary encoded numbers, respectively, and the third solves the factorization problem.

Keywords: factorization, P systems, membrane systems.

Membrane systems (also known as P systems) have been introduced in Ref. [1] as a new class of distributed and parallel computing devices, inspired by the structure and functioning of living cells. The basic model consists of a hierarchical structure composed by several membranes, embedded into a main membrane called the skin. Membranes divide the Euclidean space into regions that contain some objects (represented by symbols of an alphabet) and evolution rules. Using these rules, the objects may evolve and/or move from a region to a neighboring one. The rules are applied in a non-deterministic and maximally parallel way: all the objects that may evolve are forced to evolve. A computation starts from an initial configuration of the system and terminates when no evolution rule can be applied. The result of a computation is the multiset of objects contained into an output membrane, or emitted to the environment from the skin of the system.

In what follows we assume the reader is already familiar with the basic notions and the terminology underlying P systems. For a layman-oriented introduction, see [2, 3], whereas a formal description is given in [4], and the latest information about P systems can be found in the P systems web page: http://psystems.disco.unimib.it/.

In the last few years, a large number of P systems have been proposed that solve computationally difficult problems in polynomial time. Some of them deal with non-numerical NP-complete decision problems, such as SAT[5–9]. Others deal with numerical NP-complete problems, that is, decision problems whose instances consist of sets or sequences of integer numbers, such as subset sum[10], knapsack[11], bin packing[12], and partition[13,14]. Also, some membrane algorithms inspired by P systems have been proposed that solve NP-hard optimization problems, such as TSP[15] and min storage[16]. However, there exist problems which are neither decision nor optimization problems, but are nevertheless very interesting. A notable example is factorization, which can be defined as follows.

Problem 1. Name: factorization.

Instance: a positive integer number n which is the product of two prime numbers p and q.

Output: p.

This problem is generally considered intractable, which means that no polynomial time (with respect to the instance size) algorithm is known that solves it on every instance. The natural instance size for the problem is $k = \lfloor \log_2 n \rfloor + 1 = \Theta(\log_2 n)$, that is, the number of bits which are needed to represent the number n in binary form. By the way, let us note that the trivial algorithm that tries to divide n by every number comprised between 1 and $\sqrt{n}$ requires an exponential time with respect to the instance size. The conjectured intractability of this problem is often exploited in cryptography: a famous example is the RSA cryptosystem[17].

* This work has been partially supported by the Italian Ministry of University (MIUR) under the project PRIN-04 "Systems Biology: modellazione, linguaggi e analisi (SYBILLA)", and by the European Research Training Network Segravis

** To whom correspondence should be addressed. E-mail: leporati@disco.unimib.it

In [18] we have proposed a general framework which can be applied to solve many NP-complete problems. Such framework, like many solutions of intractable problems proposed in the literature, uses membrane division to produce an exponential number of membranes in polynomial time. Each membrane contains a candidate solution, which is tested to see whether it solves the problem; the systems that contain a solution emit a specified symbol, which is interpreted as the answer to the problem. In this paper we apply some ideas underlying the above framework, and we propose a family $\{\Pi_{\text{Fact}}(k)\}_{k\in\mathbb{N}}$ of P systems which solves the factorization problem, where $\Pi_{\text{Fact}}(k)$ solves all the instances in which the number n to be decomposed is a k-bit integer number. Thus, we first propose a simple method to represent non-negative integer numbers in binary notation using multisets of objects. Then, we present two simple P systems that compute the sum and the product of two k-bit numbers, respectively. Finally we show how to generate in polynomial time, using (elementary) membrane division, all possible pairs of k-bit numbers. Given a non-negative integer number n to be decomposed, we generate in this way all possible candidate decompositions (A, B). Each subsystem contains one of those candidates, and the rules to multiply them and test whether the result is equal to n. In case a match is found, the number A is sent to a prefixed output membrane.

Let us note that the only other currently known polynomial time algorithm that solves the factorization problem requires a quantum computer to be executed[19]. This is an indication of how powerful P systems with membrane division can be.

The paper is organized as follows. In Section 1 we introduce our encoding of binary numbers using multisets, and we describe two families of simple P systems which can be used to add and multiply two k-bit integer numbers, respectively. In Section 2 we propose a family $\{\Pi_{\text{Fact}}(k)\}_{k\in\mathbb{N}}$ of P systems which solves the factorization problem. Section 3 contains the conclusions.

1 Adding and multiplying two k-bit integer numbers

First of all let us show how a given non-negative integer number x can be represented in binary notation using a multiset. Let $x_{k-1}, \cdots, x_1, x_0$ (with $k \geqslant 1$) be the binary representation of $x \geqslant 0$, so that $x = \sum_{i=0}^{k-1} x_i 2^i$. We use the objects from the following alphabet $\mathcal{A}_k$, for $k \geqslant 1$:

$$\mathcal{A}_k = \{\langle b, j\rangle \mid b \in \{0,1\},\ j \in \{0,1,\cdots,k-1\}\} \quad (1)$$

Object $\langle b, j\rangle$ is used to represent bit b into position j in the binary encoding of an integer number. Hence, to represent the above number x we will use the following multiset (actually, a set) of objects:

$$\langle x_{k-1}, k-1\rangle, \cdots, \langle x_1, 1\rangle, \langle x_0, 0\rangle$$

Let us remark that the alphabet $\mathcal{A}_k$ depends on the length of the binary representation of the number x. Moreover, it is clear that with $\mathcal{A}_k$ we can represent all integer numbers in the range $0, 1, \cdots, 2^k - 1$.

Now, let $a_{k-1}, \cdots, a_1, a_0$ and $b_{k-1}, \cdots, b_1, b_0$ be the binary representations of the k-bit integer numbers $A = \sum_{i=0}^{k-1} a_i 2^i$ and $B = \sum_{i=0}^{k-1} b_i 2^i$, respectively. The sum $A + B$ can thus be expressed as $A + B = \sum_{i=0}^{k-1} (a_i + b_i) 2^i$, and we can define a family $\{\Pi_{\text{Add}}(k)\}_{k\in\mathbb{N}}$ of P systems which performs the addition as follows. For the moment, let us assume that we do not need to differentiate the objects of the two input numbers A and B, and of the output number C. The system $\Pi_{\text{Add}}(k)$, which performs the addition of two k-bit integer numbers, is formally defined as:

$$\Pi_{\text{Add}}(k) = (\mathcal{A}_k, \mu, w, R(k), i_{\text{in}}, i_{\text{out}})$$

where: $\mathcal{A}_k$ is the above (see equation (1)) alphabet of objects; $\mu = [\ ]_{skin}$ the membrane structure consisting of the skin only; w the multiset of objects initially present in the region enclosed by the skin. Such objects represent (in binary form, as described above) the k-bit integer numbers A and B which have to be added; $R(k)$ is the following set of evolution rules associated with the skin region: for all $i \in \{0, 1, \cdots, k-1\}$,

$$\langle 0, i\rangle\langle 0, i\rangle \to \langle 0, i\rangle$$
$$\langle 0, i\rangle\langle 1, i\rangle \to \langle 1, i\rangle$$
$$\langle 1, i\rangle\langle 1, i\rangle \to \langle 0, i\rangle\langle 1, i+1\rangle;$$

$i_{\text{in}} = skin$ specifies the input membrane of $\Pi_{\text{Add}}(k)$; and $i_{out} = skin$ specifies the output membrane of $\Pi_{\text{Add}}(k)$.

The semantics of the rules is the usual for evolution rules: they are all applied in a maximally parallel way. The computation halts when no more rules can

be applied. When this happens, the multiset which occurs in the output membrane (in this case, the skin) is the output of the computation.

Computations are performed as follows. The objects which denote the bits of the binary representation of A and B are initially put into the region enclosed by the skin. Then the computation starts, and the rules from $R(k)$ are applied. It is easily seen that these rules mimic the usual addition algorithm that considers the bits of A and B occurring in the same positions, and for each of these pairs produces the corresponding result and (eventually) a carry bit. Hence at the end of the computation, when no more rules from $R(k)$ can be applied, the skin will contain the multiset which corresponds to the binary representation of $C = A + B$. The number of cellular steps of the P system, as well as the cardinality of its alphabet, are clearly linear with respect to k.

Now, let us describe a family $\{\Pi_{\text{Prod}}(k)\}_{k\in\mathbb{N}}$ of P systems such that $\Pi_{\text{Prod}}(k)$ computes the product of two k-bit integer numbers $A = \sum_{i=0}^{k-1} a_i 2^i$ and $B = \sum_{i=0}^{k-1} b_i 2^i$. It is immediately seen that such a product can be written as $A \cdot B = \sum_{i=0}^{k-1}\sum_{j=0}^{k-1} a_i b_j 2^{i+j}$. Hence we will first compute the contribution of each pair of bits a_i and b_j, and then we will sum all these contributions. However, in order to realize this strategy we need to distinguish between the objects that represent the bits of A and B, and those that represent the bits of the result C. To this aim, we mark each element in the *input* multiset with a leading label A, B or C, thus obtaining the following modified alphabet $\mathscr{A}_k'$:

$$\mathscr{A}_k' = \{\langle l, b, j\rangle \mid l \in \{A, B\}, b \in \{0,1\}, j \in \{0,1,\cdots,k-1\}\} \cup \{\langle C, b, j\rangle \mid b \in \{0,1\}, j \in \{0,1,\cdots,k\}\}$$

In this way, the i-th bit of A (that is, a_i) and the j-th bit of B (that is, b_j) are represented by the objects $\langle A, a_i, i\rangle$ and $\langle B, b_j, j\rangle$, respectively. By adopting this alphabet, also the rules $R(k)$ to perform the addition $A + B$ have to be modified as follows: for all $i \in \{0,1,\cdots,k-1\}$,

$\langle A,0,i\rangle\langle B,0,i\rangle \to \langle C,0,i\rangle$
$\langle A,0,i\rangle\langle B,1,i\rangle \to \langle C,1,i\rangle$
$\langle A,1,i\rangle\langle B,0,i\rangle \to \langle C,1,i\rangle$
$\langle A,1,i\rangle\langle B,1,i\rangle \to \langle C,0,i\rangle\langle C,1,i+1\rangle$
$\langle C,0,i\rangle\langle C,0,i\rangle \to \langle C,0,i\rangle$
$\langle C,0,i\rangle\langle C,1,i\rangle \to \langle C,1,i\rangle$
$\langle C,1,i\rangle\langle C,1,i\rangle \to \langle C,0,i\rangle\langle C,1,i+1\rangle$

Also in this case the rules are applied in a maximally parallel way, and the computation halts when no rule can be applied. In particular, the first step of the computation produces some bits of the result C as well as the carry bits; the subsequent steps involve only the objects $\langle C, b, j\rangle$, and compute the final result by adding all carry bits. In all cases, the computation halts after a linear (with respect to k) number of steps.

We can now formally define the P system $\Pi_{\text{Prod}}(k)$ that computes the product of any two k-bit numbers A and B:

$$\Pi_{\text{Prod}}(k) = (\mathscr{A}_k'', \mu, w, R(k), i_{\text{in}}, i_{\text{out}})$$

where $\mathscr{A}_k''$ is the following alphabet of objects:

$$\mathscr{A}_k'' = \{\langle l, b, j\rangle \mid l \in \{A, B\}, b \in \{0,1\}, j \in \{0,1,\cdots,k-1\}\} \cup \{\langle C, b, j\rangle \mid b \in \{0,1\}, j \in \{0,1,\cdots,2k-1\}\} \cup \{\langle i, j\rangle \mid i, j \in \{0,1,\cdots,k-1\}\};$$

$\mu = [\ ]_{skin}$ is the membrane structure consisting of the skin only;

w is the multiset of objects initially present in the region enclosed by the skin. Such objects represent the k-bit numbers A and B which have to be multiplied; moreover, for all $i, j \in \{0,1,\cdots,k-1\}$, w contains exactly one copy of the object $\langle i, j\rangle$;

$R(k)$ is the following set of evolution rules associated with the skin region: for all $i, j \in \{0,1,\cdots,k-1\}$, and for all $l \in \{0,1,\cdots,2k-1\}$,

$\langle A,0,i\rangle\langle B,0,j\rangle\langle i,j\rangle \to \langle C,0,i+j\rangle\langle A,0,i\rangle\langle B,0,j\rangle$
$\langle A,0,i\rangle\langle B,1,j\rangle\langle i,j\rangle \to \langle C,0,i+j\rangle\langle A,0,i\rangle\langle B,1,j\rangle$
$\langle A,1,i\rangle\langle B,0,j\rangle\langle i,j\rangle \to \langle C,0,i+j\rangle\langle A,1,i\rangle\langle B,0,j\rangle$
$\langle A,1,i\rangle\langle B,1,j\rangle\langle i,j\rangle \to \langle C,1,i+j\rangle\langle A,1,i\rangle\langle B,1,j\rangle$
$\langle C,0,l\rangle\langle C,0,l\rangle \to \langle C,0,l\rangle$
$\langle C,0,l\rangle\langle C,1,l\rangle \to \langle C,1,l\rangle$
$\langle C,1,l\rangle\langle C,1,l\rangle \to \langle C,0,l\rangle\langle C,1,l+1\rangle$;

$i_{\text{in}} = skin$ specifies the input membrane of $\Pi_{\text{Prod}}(k)$; and $i_{\text{out}} = skin$ specifies the output membrane of $\Pi_{\text{Prod}}(k)$.

As told above, the first four rules of $R(k)$ compute all the contributions $a_i b_j 2^{i+j}$ given by the single bits of A and B to the product $C = A \cdot B = \sum_{i=0}^{k-1} \sum_{j=0}^{k-1} a_i b_j 2^{i+j}$. The objects that represent such bits, namely $\langle A, a_i, i \rangle$ and $\langle B, b_j, j \rangle$, are regenerated each time these rules are applied, so that they can be reused for different combinations of i and j. The possibility to reuse the objects for the same combination of i and j is excluded by the fact that objects $\langle i, j \rangle$, for all $i, j \in \{0, 1, \cdots, k-1\}$, initially present in the input multiset w, are consumed when the rules are applied. In other words, the presence of a given object $\langle i, j \rangle$ denotes the fact that the contribution given by the bits a_i and b_j has not been taken into account yet. The remaining rules of $R(k)$ sum all the contributions, that is, all the bits of C produced in the previous steps of computation. Note the similarity with the rules of the first P system that computes the addition of A and B proposed above. It is easily seen that the total number of cellular steps is polynomially bounded with respect to k, the instance size.

2 Solving the factorization problem

In this section, starting from the framework proposed in [18] to solve NP-complete problems by membrane division, we design a family $\{\Pi_{\text{Fact}}(k)\}_{k \in \mathbb{N}}$ of P systems which solves the factorization problem. The P system $\Pi_{\text{Fact}}(k)$ solves the problem over all the instances of size k, and is defined as follows:

$\Pi_{\text{Fact}}(k) = (O_k, \mu, w_1, w_2, R_1(k), R_2(k), i_{\text{in}}, i_{\text{out}})$

where

$$\begin{aligned} O_k = & \{\langle l, b, j \rangle \mid l \in \{A, B\}, b \in \{0,1\}, \\ & j \in \{0, 1, \cdots, k-2\}\} \\ & \cup \{\langle l, b, j \rangle \mid l \in \{C, X, n\}, b \in \{0,1\}, \\ & j \in \{0, 1, \cdots, 2k-2\}\} \\ & \cup \{\langle i, j \rangle \mid i, j \in \{0, 1, \cdots, k-2\}\} \\ & \cup \{s_0, s_1, \cdots, s_{k-1}\} \\ & \cup \{\langle d, i \rangle \mid i \in \{-1, 0, 1, 2, \cdots, k-2\}\} \\ & \cup \{\langle e, i \rangle \mid i \in \{0, 1, \cdots, k-1\}\} \end{aligned}$$

is the set of objects. Objects $\langle l, b, j \rangle$, with $l \in \{A, B\}$, and $\langle C, b, j \rangle$ are used to represent candidate solutions and their products, respectively, as described above. Similarly, objects $\langle n, b, j \rangle$ and $\langle X, b, j \rangle$ are used to represent the instance n and the result of the bitwise comparison between n and C, respectively. Objects s_j are used to drive the first phase of computation, in which all possible candidate solutions are generated. Objects $\langle i, j \rangle$ are used instead to correctly multiply the candidate solutions, whereas objects $\langle d, i \rangle$ and $\langle e, i \rangle$ drive the comparison between C and n and the (possible) expulsion of A to region 1, respectively. As we can see, the number of objects in O_k is polynomial with respect to k, the instance size;

$\mu = [[\ ]_2]_1$ is the initial membrane structure. At the beginning of the computation, the system is composed by a skin membrane which encloses a single subsystem, as shown in Fig. 1;

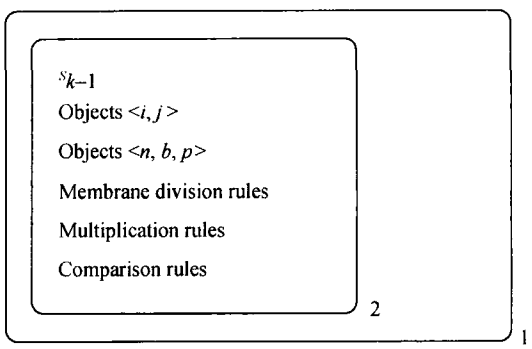

Fig. 1. The initial state of system $\Pi_{\text{Fact}}(k)$.

$w_1 = \emptyset$ is the multiset of objects initially present into the skin membrane;

w_2 is the multiset of objects initially present into the subsystem. Such objects represent the instance n of the problem (a k-bit integer number which is the product of two unknown prime numbers p and q). Moreover, there is one copy of the symbol s_{k-1} which is used to start the generation phase, during which all possible candidates for p and q are generated through membrane division. Further, there is also one copy of each object $\langle i, j \rangle$, for all $i, j \in \{0, 1, \cdots, k-2\}$; these objects are duplicated during the generation phase, and are used to correctly multiply the candidate solutions;

$R_1(k) = \emptyset$ is a set of rules associated with region 1 (the skin);

$R_2(k)$ is the set of rules associated with region 2 (the subsystem). There are three kinds of rules: rules which generate all possible candidate solutions, using membrane division:

$$\begin{aligned} [s_j]_2 \rightarrow & [\langle A,0,j\rangle \langle B,0,j\rangle s_{j-1}]_2 \\ & \cdot [\langle A,0,j\rangle \langle B,1,j\rangle s_{j-1}]_2 \\ & [\langle A,1,j\rangle \langle B,0,j\rangle s_{j-1}]_2 \\ & \cdot [\langle A,1,j\rangle \langle B,1,j\rangle s_{j-1}]_2 \end{aligned}$$

for all $j \in \{1, 2, \cdots, k-1\}$, (2)

rules which compute the product $C = A \cdot B$ of the two candidates, and rules which compare C with n, and expel the first candidate (A) to region 1 if such comparison is successful. Unlike what is usually found in the literature, the above division rules do not duplicate the number of membranes at each step, but rather they quadruplicate them. Let us note, however, that this operation can be performed even if we assume that we are only able to duplicate membranes: we first duplicate the membrane with the two possible values of bit a_j, and then for each of the membranes obtained we further duplicate, producing the two possible values of bit b_j;

$i_{in} = 2$ is the input membrane; and $i_{out} = 1$ (the skin) is the output membrane.

The computation is composed of two phases:

(i) Generation phase: using membrane division, the subsystem initially present into the skin generates 4^{k-1} subsystems. Each subsystem contains a pair of candidates (A, B) for the factors p and q of n.

(ii) Verification and output phase: each of the 4^{k-1} subsystems created during the generation phase performs the multiplication $C = A \cdot B$ of its two candidate factors. Then it checks (by a bitwise comparison) whether $C = n$ and, in case of positive answer, it expels to region 1 the first candidate (A).

The next subsections give a detailed description of each computation phase.

2.1 Generation phase

As stated above, this first phase uses membrane division to generate 4^{k-1} subsystems, each one containing two candidate solutions A and B (which are two $(k-1)$-bit integer numbers, expressed in binary form as described above), as well as the objects $\langle i, j \rangle$, for all $i, j \in \{0, 1, \cdots, k-2\}$, which are needed to multiply them. The values generated in this way for A and B range from 0 to $\lceil n/2 \rceil$. We point out that, as a matter of fact, it would suffice to consider numbers ranging from 1 to $\lceil \sqrt{n} \rceil$. However, this would complicate in a non trivial way the definition of the system $\Pi_{Fact}(k)$, without giving a true advantage (at least, from a theoretical point of view). In fact, if the instance n of the problem is correctly chosen, then there exists only one possible factorization having two factors p and q also in the range from 0 to $\lceil n/2 \rceil$. A number of subsystems of $\Pi_{Fact}(k)$ will certainly make useless computations, hence wasting computational resources, but this a major concern only if we should really implement the algorithm in a true, biological (or whatever) membrane system.

The division process is driven by the presence of symbols $s_{k-1}, \cdots, s_1, s_0$ into the subsystems generated so far. The subsystem (membrane 2) initially present into $\Pi_{Fact}(k)$ contains the symbol s_{k-1}, which activates the corresponding rule from (2). As a result of the application of this rule, membrane 2 is divided in four copies with all its content (objects and rules). The symbol s_{k-1} is replaced with s_{k-2} in all the generated subsystems. Moreover, each subsystem contains one possible configuration for the $(k-1)$-th bit of the two candidate solutions A and B. In other words, the first subsystem contains the objects $\langle A, 0, k-1 \rangle$ and $\langle B, 0, k-1 \rangle$, corresponding to the fact that bits a_{k-1} and b_{k-1} are both equal to zero. The second subsystem contains the objects $\langle A, 0, k-1 \rangle$ and $\langle B, 1, k-1 \rangle$, corresponding to $a_{k-1} = 0$ and $b_{k-1} = 1$, respectively. Similarly, the third subsystem contains the objects $\langle A, 1, k-1 \rangle$ and $\langle B, 0, k-1 \rangle$, while the fourth one contains $\langle A, 1, k-1 \rangle$ and $\langle B, 1, k-1 \rangle$. Each of these subsystems also contains one copy of symbol s_{k-2}, as well as the rules (2). The rule

$$[s_{k-2}]_2 \rightarrow [\langle A, 0, k-2 \rangle \langle B, 0, k-2 \rangle s_{k-3}]_2$$
$$[\langle A, 0, k-2 \rangle \langle B, 1, k-2 \rangle s_{k-3}]_2$$
$$[\langle A, 1, k-2 \rangle \langle B, 0, k-2 \rangle s_{k-3}]_2$$
$$[\langle A, 1, k-2 \rangle \langle B, 1, k-2 \rangle s_{k-3}]_2$$

can thus be applied in each subsystem. As a result, we obtain four subsystems from each initial subsystem, which contain all previous rules as well as the objects which represent the corresponding assignments to the $(k-2)$-th bit for both candidate solutions A and B. Each copy of object s_{k-2} is replaced with a copy of object s_{k-3}, and so on. This process, which is depicted in Fig. 2 for $k = 3$, terminates when 4^{k-1} subsystems have been obtained, each containing the symbol s_0. Clearly, this phase is performed in linear time with respect to k. The system $\Pi_{Fact}(k)$ is now ready to execute the verification phase.

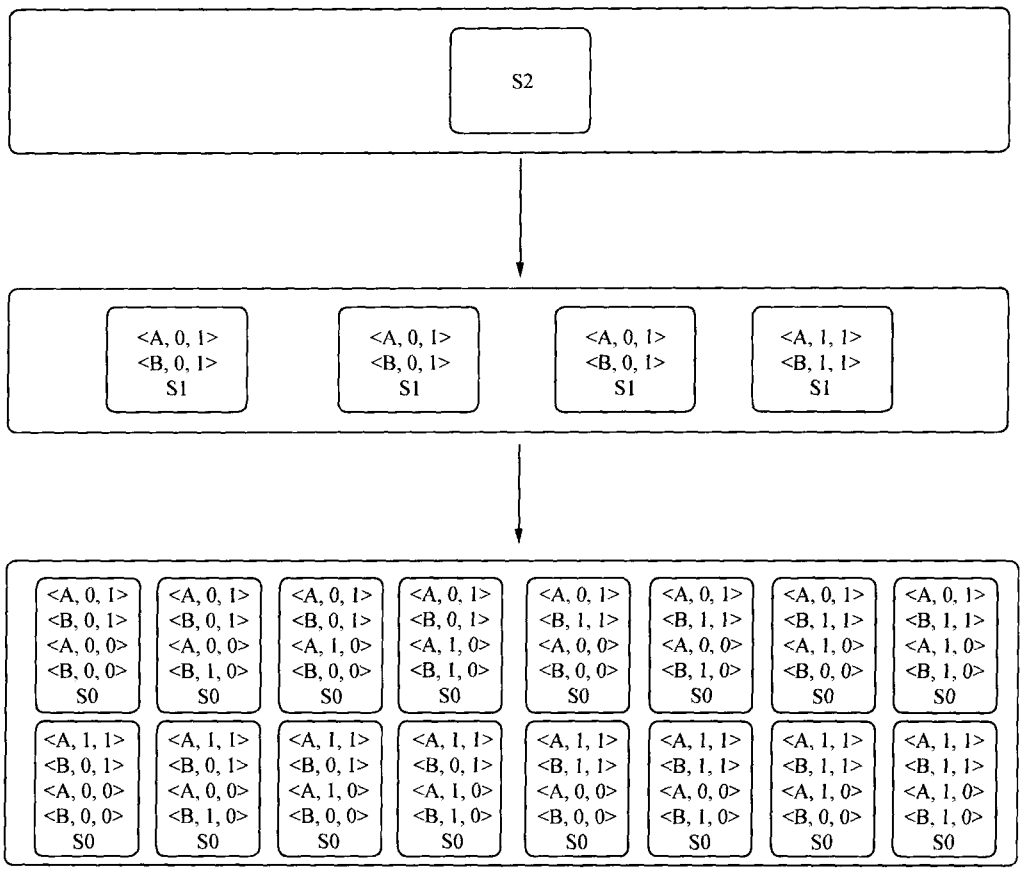

Fig. 2. An example of generation phase, for $k = 3$.

2.2 Verification and output phase

During this phase, each of the 4^{k-1} subsystems generated during the first phase computes the product $C = A \cdot B$ between its candidate solutions A and B, and checks whether the result C is equal to n. If so, it outputs A to region 1.

Let us focus our attention to a single subsystem. To compute the product $C = A \cdot B$, we use the following cooperative rules, which are similar to those of system $\Pi_{Prod}(k)$ described above: for all $i, j \in \{0, 1, \cdots, k-2\}$, and for all $l \in \{0, 1, \cdots, 2k-1\}$,

$$\langle A,0,i\rangle\langle B,0,j\rangle\langle i,j\rangle \rightarrow \langle C,0,i+j\rangle\langle A,0,i\rangle\langle B,0,j\rangle$$
$$\langle A,0,i\rangle\langle B,1,j\rangle\langle i,j\rangle \rightarrow \langle C,0,i+j\rangle\langle A,0,i\rangle\langle B,1,j\rangle$$
$$\langle A,1,i\rangle\langle B,0,j\rangle\langle i,j\rangle \rightarrow \langle C,0,i+j\rangle\langle A,1,i\rangle\langle B,0,j\rangle$$
$$\langle A,1,i\rangle\langle B,1,j\rangle\langle i,j\rangle \rightarrow \langle C,1,i+j\rangle\langle A,1,i\rangle\langle B,1,j\rangle$$
$$\langle C,0,l\rangle\langle C,0,l\rangle \rightarrow \langle C,0,l\rangle$$
$$\langle C,0,l\rangle\langle C,1,l\rangle \rightarrow \langle C,1,l\rangle$$
$$\langle C,1,l\rangle\langle C,1,l\rangle \rightarrow \langle C,0,l\rangle\langle C,1,l+1\rangle$$

These rules are present in region 2 in the initial setting of $\Pi_{Fact}(k)$, and are copied to each generated subsystem during the generation phase.

After the product $C = A \cdot B$ has been computed, the system uses the following rules to compare it with n:

$$\langle C,0,i\rangle\langle n,0,i\rangle \rightarrow \langle X,0,i\rangle$$
$$\langle C,0,i\rangle\langle n,1,i\rangle \rightarrow \langle X,1,i\rangle$$
$$\langle C,1,i\rangle\langle n,0,i\rangle \rightarrow \langle X,1,i\rangle$$
$$\langle C,1,i\rangle\langle n,1,i\rangle \rightarrow \langle X,0,i\rangle$$
$$\langle X,0,k-1\rangle \rightarrow \langle d,k-2\rangle$$
$$\langle d,i\rangle\langle X,0,i\rangle \rightarrow \langle d,i-1\rangle$$

for all $i \in \{0, 1, \cdots, k-2\}$

$$\langle d,-1\rangle \rightarrow \langle e,k-1\rangle\cdots\langle e,1\rangle\langle e,0\rangle$$
$$\langle e,i\rangle\langle A,0,i\rangle \rightarrow \langle A,0,i\rangle_{out}$$
$$\langle e,i\rangle\langle A,1,i\rangle \rightarrow \langle A,1,i\rangle_{out}$$

The first four rules compute a bitwise XOR between every bit of C and the corresponding bit of n. The resulting sequence X of k bits is represented by a multiset whose objects $\langle X, b, i\rangle$ (with $b \in \{0, 1\}$ and $j \in \{0, 1, \cdots, k-1\}$) contain the label X in their first component. The next three rules check whether all the bits of X are equal to 0, that is, they check

whether $C = n$. This check is performed in linear time, one bit at a time, starting from the most significant bit. If such bit is 0, then the first rule produces the object $\langle d, k-1\rangle$, and the check proceeds with the next bit. On the other hand, if the most significant bit of X is 1, then the object $\langle d, k-1\rangle$ is not produced, and the computation of the entire subsystem halts. If every bit of X is zero, then the object $\langle d, -1\rangle$ is produced at the end of the check, which in turn produces the objects $\langle e, k-1\rangle, \cdots, \langle e, 1\rangle, \langle e, 0\rangle$. These objects are used by the last two rules, that in one step expel the multiset which represents the binary number A to region 1.

Let us note that, up to now, we have neglected two details in the construction of $\Pi_{\text{Fact}}(k)$: (i) how to start the verification phase when the generation phase has ended, and (ii) how to start, in each subsystem, the comparison between C and n only after the computation of $C = A \cdot B$ has been completed. These are two synchronization problems. The first can be easily solved by using active membranes: it suffices to set the polarization of membranes to different values during the various phases of the computation. Precisely, rules (2) of the generation phase should be made applicable only when the polarization of membrane 2 is zero; at the end of the phase, through the rule $[s_0]_2^0 \rightarrow [\ ]_2^1$ (notice that symbol s_0 is consumed) we can modify the polarization of the membrane, so that the verification phase can start. Of course, all the rules concerning the multiplication of A and B should be applied only if the polarization of membrane 2 is 1. The second synchronization problem, instead, can be solved by putting a step counter into each subsystem, which is decremented at each step during the computation of $C = A \cdot B$. When the counter reaches the value zero, the comparison between C and n starts. Since at each step of multiplication all the contributions $a_i b_j 2^{i+j}$ (with $i \neq j$) are taken into account, after at most a linear number of steps all k^2 contributions, as well as the carry bits generated during the additions, will be produced. In the worst case there will be a linear number of some contributions, each one requiring a logarithmic number of steps to be summed. Hence, a conservative value which can be used to initialize the step counter is $2k \log k$. Operatively, we add the rules:

$$[p_i]_2^1 \rightarrow [p_{i-1}]_2^1 \quad \forall i \in \{1, 2, \cdots, 2k\log k\}$$

to the region enclosed by membrane 2, and the objects $\{p_0, p_1, \cdots, p_{2k\log k}\}$ to the alphabet O_k. All these rules are copied during the generation phase, and become active only when the polarization of membrane 2 becomes 1 (that is, when the generation phase ends). A further rule:

$$[p_0]_2^1 \rightarrow [p_0]_2^0$$

switches the polarization back to 0, so that the comparison between C and n can start. Of course, this requires that the rules which perform such comparison can be made applicable only when the polarization of membrane 2 is 0. Note that interference with the rules concerning the generation phase does not occur, since the two phases of computation work on different kinds of objects.

Concerning the time complexity of P system $\Pi_{\text{Fact}}(k)$, it is clear that the generation phase is executed in linear time with respect to k, since at each cellular step a different symbol from $\{s_0, s_1, \cdots, s_{k-1}\}$ is used. As told above, in each of the subsystems generated during this phase, the multiplication between the two candidates A and B is executed in polynomial time (recall that we use a counter for the computation steps, which is initialized to $2k\log k$). The bitwise XOR between every bit of the result C and the corresponding bit of n is executed in one cellular step; on the other hand, checking whether all the bits obtained from the bitwise XOR are equal to zero requires a linear number of steps in the worst case. Finally, expelling the first candidate solution (A) to region 1 requires just one additional cellular step. We can thus conclude that the total number of cellular steps performed by $\Pi_{\text{Fact}}(k)$ is polynomial with respect to k, the instance size.

3 Conclusions

Starting from the framework which was introduced in [18] to solve NP-complete decision problems, we have proposed a family $\{\Pi_{\text{Fact}}(k)\}_{k\in\mathbb{N}}$ of P systems that solves a very famous and interesting problem: factorization. This problem, which is neither a decision nor an optimization problem, is currently considered intractable. That is, no deterministic (or even probabilistic) polynomial time algorithm is known, which can be executed on Turing machines, that solves the problem for every possible instance. For this reason, factorization is used in a large number of cryptographic applications, the most famous of which is certainly the public key cryptosystem RSA[17].

The structure of the P system $\Pi_{\mathrm{Fact}}(k)$ is very simple. The system is initialized with a multiset which expresses the integer number n to be factorized, as well as some additional work objects. At the beginning of the computation, an exponential number of subsystems is generated by membrane division, each containing one pair (A, B) of candidate solutions for the factors of n. Subsequently, each subsystem computes the product $C = A \cdot B$, and compares the result C with n. If the two numbers are found equal, then the subsystem has found the factors of n, and it outputs one of them (A) to the region enclosed by the skin membrane. The number of objects and rules in our family of P systems grows polynomially with respect to k, the instance size.

For the sake of clarity, our solution scheme uses cooperative rules. However, cooperation always occurs between no more than three objects. Such kind of cooperation can be easily removed by computing the product $C = A \cdot B$, and the subsequent comparison between C and n, through and/or/not boolean circuits, and simulating these circuits using mobile catalysts, with or without the use of promoters and of weak priorities between rules, as shown in [20]. Another possibility is to transform all and/or/not circuits into reversible Fredkin circuits and simulate them using energy-based P systems, as shown in [21].

References

1 Păun Gh. Computing with membranes. Journal of Computer and System Sciences, 2000, 61(1): 108—143

2 Păun Gh and Pérez-Jiménez MJ. Recent computing models inspired from biology: DNA and membrane computing. Theoria, 2003, 18: 72—84

3 Păun Gh and Rozenberg G. A guide to membrane computing. Theoretical Computer Science, 2002, 287(1): 73—100

4 Păun Gh. Membrane Computing. An Introduction. Berlin: Springer Verlag, 2002

5 Păun Gh. P systems with active membranes: Attacking NP-complete problems. Journal on Automata Languages and Combinatorics, 2001, 6(1): 75—90

6 Pérez-Jiménez MJ, Romero-Jiménez A and Sancho-Caparrini F. A polynomial complexity class in P systems using membrane division. In: Proceedings of the 5th Workshop on Descriptional Complexity of Formal Systems (DCFS 2003). Budapest, July 12—14, 2003, 284—294

7 Gutiérrez-Naranjo MA, Pérez-Jiménez MJ and Romero-Campero FJ. Solving SAT with membrane creation. In: Proceedings of Computability in Europe 2005 (CiE2005): New Computational Paradigms. Amsterdam, The Netherlands, June 8—12, 2005, 82—91

8 Manca V. Membrane algorithms for propositional satisfiability. In: Pre-Proceedings of International Workshop on Membrane Computing, Curtea de Argeş, Romania, 2001, GRLMC TR 17, Rovira i Virgili Univ., Tarragona, 2001, 181—192

9 Zandron C, Ferretti C and Mauri G. Solving NP-complete problems using P systems with active membranes. In: Unconventional Models of Computation. London: Springer-Verlag, 2000, 289—301

10 Pérez-Jiménez MJ and Riscos-Núñez A, Solving the subset-sum problem by active membranes. New Generation Computing, 2005, 23(4): 367—384

11 Pérez-Jiménez MJ and Riscos-Núñez A. A linear solution for the Knapsack problem using active membranes. In: Membrane Computing, International Workshop (WMC 2003). Berlin: Springer Verlag, 2004, LNCS 2933, 250—268

12 Pérez-Jiménez MJ and Romero-Campero FJ. Solving the BIN PACKING problem by recognizer P systems with active membranes. In: Proceedings of the Second Brainstorming Week on Membrane Computing. University of Seville, Report RGNC 01/04, 2004, 414—430

13 Gutiérrez-Naranjo MA, Pérez-Jiménez MJ and Riscos-Núñez A. A fast P system for finding a balanced 2-partition. Soft Computing, 2005, 9(9): 53—58

14 Gutiérrez-Naranjo MA, Pérez-Jiménez MJ and Riscos-Núñez A. An efficient cellular solution for the partition problem. In: Proceedings of the Second Brainstorming Week on Membrane Computing, University of Seville, Report RGNC 01/04, 2004, 237—246

15 Nishida TY. Membrane algorithms. In: Membrane Computing: 6th International Workshop (WMC 2005). Berlin: Springer Verlag, 2006, LNCS 3850, 55—66

16 Leporati A and Pagani D. A membrane algorithm for the min storage problem. In: Membrane Computing: 7th International Workshop (WMC 2006). Berlin: Springer Verlag, 2006, LNCS 4361, 443—462

17 Rivest RL, Shamir A and Adleman LM. A method for obtaining digital signatures and public-key cryptosystems. Communications of the ACM, 1978, 21(2): 120—126

18 Leporati A, Zandron C and Gutiérrez-Naranjo MA. P systems with input in binary form. International Journal of Foundations of Computer Science, 2006, 17(1): 127—146

19 Shor P. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. SIAM Journal on Computing, 1997, 26(5): 1484—1509

20 Ceterchi R and Sburlan D. Simulating boolean circuits with P systems. In: Membrane Computing, International Workshop (WMC 2003). Berlin: Springer Verlag, 2004, LNCS 2933, 104—122

21 Leporati A, Zandron C and Mauri G. Universal families of reversible P systems. In: Machines, Computations and Universality (MCU 2004). Berlin: Springer Verlag, 2005, LNCS 2254, 257—268