Converting Integer Numbers from Binary to Unary Notation with P Systems

Miguel A. GUTIÉRREZ NARANJO[1], Alberto LEPORATI[2] and Claudio ZANDRON[2]

[1] Research Group on Natural Computing
Department of Computer Science and Artificial Intelligence
Sevilla University
Avda Reina Mercedes s/n, 41012 Sevilla, Spain
E-mail: magutier@us.es

[2] Dipartimento di Informatica, Sistemistica e Comunicazione
Università degli Studi di Milano – Bicocca
Via Bicocca degli Arcimboldi 8, 20126 Milano, Italy
E-mail: {leporati,zandron}@disco.unimib.it

Abstract. Current P systems which solve **NP**–complete numerical problems represent instances in unary notation. In classical complexity theory, based upon Turing machines, switching from binary to unary encoded instances generally corresponds to simplify the problem. In this paper we show that this does not occur when working with P systems. Namely, we propose a simple method to encode binary numbers using multisets, and a family of P systems which transforms such multisets into the usual unary notation.

1 Introduction

P systems (also called *membrane systems*) were introduced in [**?**] 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 nondeterministic 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[1].

[1] A layman-oriented introduction can be found in [**?**]; a formal description in [**?**] and the latest information about P systems can be found on [**?**].

Many P systems which solve **NP**–complete decision problems have appeared in the literature during the last few years. Both in the field of *numerical* problems, that is, problems whose instances consist of sets or sequences of integer numbers (see for example Subset Sum [**?**], Knapsack [**?**], Bin Packing [**?**] or Partition [**?**] problems) or non-numerical problems as SAT [**?**, **?**] or QSAT [**?**].

It is well known [**?**, **?**] that the difficulty of such numerical problems is tied to the magnitude of the numbers which appear into the instance. For example, let us consider the PARTITION problem, which can be stated as follows:

Problem 1.1 NAME*:* PARTITION*.*

- INSTANCE*: a set* $A = \{a_1, a_2, \ldots, a_n\}$ *of positive integer numbers*
- QUESTION*: is there a subset* $A' \subseteq A$ *such that* $\sum_{a' \in A'} a' = \sum_{a \in A \setminus A'} a$*?*

The following algorithm solves the problem using the well known Dynamic Programming technique [**?**]. In particular, the algorithm returns 1 on positive instances, and 0 on negative instances.

```
PARTITION({a_1, a_2, ..., a_n})
s ← Σ_{i=1}^{n} a_i
if s mod 2 = 1 then return 0
for j ← 1 to s/2
    do M[1, j] ← 0
M[1, 0] ← M[1, a_1] ← 1
for i ← 2 to n
     do for j ← 0 to s/2
              do M[i, j] ← M[i − 1, j]
                 if j ≥ a_i and M[i − 1, j − a_i] > M[i, j]
                    then M[i, j] ← M[i − 1, j − a_i]
return M[n, s/2]
```

First of all, the algorithm computes the sum s of all elements in the instance. If s is odd then the instance is certainly negative, and thus the algorithm returns 0. If s is even then the algorithm checks for the existence of a subset $A' \subseteq A$ such that $\sum_{a' \in A'} a' = \frac{s}{2}$. In order to look for A', the algorithm uses a $n \times (\frac{s}{2} + 1)$ matrix M whose entries are from $\{0, 1\}$. It fills the matrix by rows, starting from the first row. Each row is filled from left to right. The entry $M[i, j]$ is filled with 1 if and only if there exists a subset of $\{a_1, a_2, \ldots, a_i\}$ whose elements sum up to j. The given instance of PARTITION is thus a *positive* instance if and only if $M[n, \frac{s}{2}] = 1$ at the end of the execution.

Since each entry is considered exactly once to determine its value, the time complexity of the algorithm is proportional to $n(\frac{s}{2} + 1) = \Theta(ns)$. This means that the difficulty of the problem depends on the value of s, that is, on the magnitude of the values in A. In fact, let us denote by K the maximum element of A. If K is polynomially bounded w.r.t. n then also $s = \sum_{i=1}^{n} a_i \leq Kn$ is polynomially bounded w.r.t. n, and thus the above algorithm works in polynomial time. On the other hand, if K is exponential w.r.t. n, say $K = 2^n$, then also s is exponential and the above algorithm works in exponential time and space. This behavior is usually referred to in the literature by telling that the PARTITION problem is a *pseudo–polynomial* **NP**–complete problem.

The fact that in general the above algorithm is not a polynomial time algorithm for PARTITION can be immediately understood by comparing its time complexity with the instance size. The usual size for the instances of PARTITION is $\Theta(n \log K)$ (also $O(n \log s)$ in [?, page 91]), since for conciseness every "reasonable" encoding is assumed to represent each element of A using a string whose length is $O(\log K)$. Here all logarithms are taken with base 2. Stated differently, the size of the instance is usually considered to be the number of bits which must be used to represent in binary all the integer numbers which occur in A. If we would represent such numbers using the unary notation, then the size of the instance would be $\Theta(nK)$. But in this case we could write a program which first converts the instance in binary form and then uses the above algorithm to solve the problem in polynomial time with respect to the new instance size. We can thus conclude that the difficulty of a numerical **NP**–complete problem depends also on the measure of the instance size we adopt.

The fact that the difficulty of a problem generally depends upon how we measure the instance size is even more apparent if we consider the FACTORIZATION problem:

Problem 1.2 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 algorithm is known that solves it on every instance. The conjectured intractability of this problem is often exploited in Cryptography: a notable example is the RSA cryptosystem [?]. Here the natural instance size for the problem is $\Theta(\log n)$, the number of bits which are needed to represent n in binary form. Also for this problem, if we let the instance size be $\Theta(n)$ then the trivial algorithm which tries to divide n by every number comprised between 1 and $\sqrt{n}$ is a polynomial time algorithm which solves the FACTORIZATION problem.

For these reasons we believe that it is important to show that P systems which solve **NP**–complete numerical problems do not take their power from the fact that the instances are represented in unary notation. Hence in this paper we first propose a simple method to represent positive integer numbers in binary notation using multisets of objects. Then, we propose a family of P systems which transforms this binary encoding into the unary notation used in [?, ?, ?, ?].

The paper is organized as follows. In section 2 we introduce our encoding of binary numbers using multisets. In section 3 we propose a family of simple P systems which can be used to transform a given positive integer number from such encoding to unary notation. Section 4 concludes the paper and gives some directions for future research.

2 Encoding binary numbers using multisets

First of all let us show how a given positive integer number x can be represented in binary notation using a multiset. Let $x_n, x_{n-1}, \ldots, x_1$ be the binary representation of x, so that $x = \sum_{i=1}^{n} x_i 2^{i-1}$. We use the objects from the following alphabet:

$$\mathcal{A}_n = \{\langle b, j \rangle \mid b \in \{0, 1\}, j \in \{1, 2, \ldots, n\}\} \tag{1}$$

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:

$$\langle x_n, n\rangle, \langle x_{n-1}, n-1\rangle, \ldots, \langle x_1, 1\rangle$$

Let us remark that the alphabet $\mathcal{A}$ depends on the length of the binary representation of the number x, i.e., with the alphabet $\mathcal{A}_n$ we can represent from 1 to $2^n - 1$.

On the other hand, the unary representation of x is obtained by choosing a symbol from an alphabet, say the symbol a from alphabet $\mathcal{A}'$, and putting into the multiset x copies of such symbol: a^x. Hence, unary notation is exponentially longer than binary notation. Our transformation thus solves another problem raised by the solutions exposed in [**?**, **?**, **?**, **?**]: in order to provide the input values to the P systems, we should insert into such systems an exponential (with respect to the instance size) number of objects. This means that an exponential amount of work to prepare the system is required.

Working with binary encoded numbers, instead, allows one to prepare the system by inserting a polynomially bounded number of objects.

3 Converting from binary to unary notation

In this section we propose a family of simple P systems which allows to convert a given positive integer number x, expressed in binary notation as exposed in the previous section, to the usual unary notation.

The objects used by the P systems form a subset of alphabet $\mathcal{A}$ of equation (**??**). Namely, in order to represent x in binary notation we will use only the objects which correspond to the bits of x which are equal to 1. For example, if $x = 25$ then its binary representation is 11001, and we will use the objects $\langle 1, 5\rangle$, $\langle 1, 4\rangle$, and $\langle 1, 1\rangle$ to represent it. Since the first element in the pairs of $\mathcal{A}$ used is always equal to 1, we can be more concise by omitting it. Once omitted the first element of the pair, also angular parenthesis are superflous.

The family of P systems which performs the transformation is formally defined as follows:

$$\Pi(n) = (A(n), \mu, w, R(n), i_{\text{in}}, i_{\text{out}})$$

where:

- $A(n) = \{1, 2, \ldots, n\} \cup \{a\}$ is the alphabet;
- $\mu = [\,]_{skin}$ is the membrane structure consisting of the skin only;
- $w = \emptyset$ is the multiset of objects initially present in region 1;
- $R(n)$ is the following set of evolution rules associated with region 1:

$$[j \to (j-1)^2]_{skin} \qquad \text{for all } j \in \{2, 3, \ldots, n\}$$
$$[1 \to a]_{skin}$$

- $i_{\text{in}} = skin$ specifies the input membrane of Π;
- $i_{\text{out}} = skin$ specifies the output membrane of Π.

The semantics of the rules is the usual for evolution rules. All they are applied in a maximal parallel mode. The number of cellular steps of the P system is bounded by n and the computation halts when no more rules can be applied. When this happens, the multiset placed in the output membrane (the only one membrane) is the *output* of the computation.

Computations proceed as follows. The objects which denote the positions of 1's in the binary representation of x are initially put into the region enclosed by the skin. Then the computation starts, and the rules from R are applied. It is easily seen that the presence of object j, with $j \in \{1, 2, \ldots, n\}$, will produce 2^{j-1} copies of object a. Hence at the end of the computation, when no more rules from R can be applied, the skin will contain x copies of object a, that is, the unary representation of x.

We conclude this section with an example of computation of the above P systems.

Let us consider again the value $x = 25$; as previously said, it will be represented by means of objects 5, 4, and 1 (each in a unique copy). At the first step of computation, we apply in parallel the rules $1 \to a$, $4 \to 3, 3$ and $5 \to 4, 4$, obtaining the multiset $a, 3, 3, 4, 4$.

Then, we apply in parallel the rule $3 \to 2, 2$ on each copy of the symbol 3, thus obtaining four copies of the symbol 2, and the rule $4 \to 3, 3$ on each copy of the symbol 4, thus obtaining four copies of the symbol 3. The multiset we obtain after the second step of computation will be $a, 2, 2, 2, 2, 3, 3, 3, 3$.

Hence, we apply the rules $2 \to 1, 1$ and $3 \to 2, 2$ obtaining $a, 1^8, 2^8$. By means of the rules $1 \to a$ and $2 \to 1, 1$ we then obtain the multiset $a^9, 1^{16}$ and finally, applying again $1 \to a$ we obtain the multiset a^{25} which is exactly the unary codification of the initially binary coded number.

From the previous definition and example, it is easy to see that the cardinality of the alphabet and the number of computation steps are linear with respect to the input size.

4 Conclusions and directions for future work

When solving numerical **NP**–complete problems using P systems, integer numbers are usually represented in unary notation. However, in classical complexity theory such numbers are assumed to be represented in binary notation, which is an exponentially more compact encoding with respect to unary notation.

Switching from binary to unary notation simplifies **NP**–complete numerical problems, because it modifies the way me measure the size of instances, as well as the relation between instance size and the running time of algorithms which solve the problem.

The eventual composition between our systems and the ones exposed in literature allows to solve **NP**–complete numerical problems working on instances whose numbers are encoded in binary form. Moreover, since the instances must be injected into the systems before starting computations, working with binary notation allows to prepare such systems with a polynomially bounded effort. The preparation of these systems requires instead an exponential amount of work when dealing with instances whose numbers are encoded in unary form.

However, this paper does not fully conclude the work on P systems which solve **NP**–complete numerical problems. In [**?**, **?**, **?**, **?**], a *uniform* family of P systems is designed to solve the problem and the *same* P system of the family solves every instance of the problem with the same *size*. As we have seen, the P systems which are able to transform an integer from binary to unary representation depends on the length of the number in binary representation. In this way, if we want to solve an instance of the Partition problem

with n integers, the P system that we need do not depend only on n, but on the concrete numbers of the set, which can be arbitrarily large.

The work presented in this paper opens a research line in the field of complexity of P systems which should be deeper studied in the future.

Acknowledgments

The present paper has been inspired by joint work with Seville research group during the Third Brainstorming Week held in Seville from January 31^{st} to February 4^{th}, 2005. The first author acknowledges the support for this research through the project TIC2002-04220-C03-01 of the Ministerio de Ciencia y Tecnología of Spain, cofinanced by FEDER funds.

References

[1] G. Ausiello, P. Crescenzi, G. Gambosi, V. Kann, A. Marchetti–Spaccamela, M. Protasi. *Complexity and Approximation. Combinatorial Optimization Problems and Their Approximability Properties*. Springer–Verlag, Berlin, 1999.

[2] T. H. Cormen, C. H. Leiserson, R. L. Rivest. *Introduction to Algorithms*. MIT Press, 1990.

[3] M. R. Garey, D. S. Johnson. *Computers and Intractability. A Guide to the Theory on NP–Completeness*. W. H. Freeman and Company, 1979.

[4] Gutiérrez-Naranjo, M.A.; Pérez-Jiménez, M.J.; Romero-Campero, F.J.: Solving SAT with Membrane Creation. Accepted paper for CiE 2005.

[5] Gutiérrez-Naranjo, M.A.; Pérez-Jiménez, M.J.; Romero-Campero, F.J.: A linear solution for QSAT with Membrane Creation. Submitted, 2005.

[6] Gutiérrez-Naranjo, M.A.; Pérez-Jiménez, M.J.; Riscos-Núñez, A.: A fast P system for finding a balanced 2-partition, DOI: 10.1007/s00500-004-0397-0 *Soft Computing*, in press. See also M. A. Gutiérrez-Naranjo, M. J. Pérez-Jiménez, A. Riscos-Núñez. An Efficient Cellular Solution for the Partition Problem. In *Proceedings of the Second Brainstorming Week on Membrane Computing*, University of Seville, February 2–7, 2004, pp. 237–246. Available at: *http://www.gcn.us.es/Brain/bravolpdf/AGPART.pdf*

[7] G. Păun. Computing with membranes. *Journal of Computer and System Sciences*, 1(61):108–143, 2000. See also Turku Centre for Computer Science — TUCS Report No. 208, 1998. Available at: *http://www.tucs.fi/Publications/techreports/TR208.php*

[8] G. Păun. Computing with Membranes. An Introduction. *Bulletin of the EATCS*, 67:139–152, February 1999.

[9] G. Păun. Computing with Membranes. A variant: P Systems with Polarized Membranes. *International Journal on Foundations of Computer Science*, 11(1):167–182, 2000. See also CDMTCS Technical Report 098, University of Auckland, 1999. Available at: *http://www.cs.auckland.ac.nz/CDMTCS*

[10] G. Păun. *Membrane Computing. An Introduction*. Springer–Verlag, Berlin, 2002.

[11] Păun, Gh.; Pérez-Jiménez, M.J.: Recent computing models inspired from biology: DNA and membrane computing, *Theoria*, **18**, 46 (2003), 72–84.

[12] G. Păun, G. Rozenberg. A Guide to Membrane Computing. *Theoretical Computer Science*, 287(1):73–100, 2002.

[13] M. J. Pérez-Jiménez, A. Riscos-Núñez. A linear solution for the Knapsack problem using active membranes. In C. Martín-Vide, G. Mauri, G. Păun, G. Rozenberg and A. Salomaa (eds.), *Membrane Computing*, Lecture Notes in Computer Science, vol. 2933, Springer-Verlag, Berlin, 2004, pp. 250–268.

[14] M. J. Pérez-Jiménez, A. Riscos-Núñez. Solving the Subset-Sum problem by active membranes. *New Generation Computing*, to appear.

[15] Pérez-Jiménez, M.J.; Romero-Campero, F.J.: Solving the BIN PACKING problem by recognizer P systems with active membranes, *Proceedings of the Second Brainstorming Week on Membrane Computing*, Gh. Păun, A. Riscos, A. Romero and F. Sancho (eds.), Report RGNC 01/04, University of Seville, 2004, 414–430.

[16] Pérez-Jiménez, M.J.; Romero-Jiménez, A.; Sancho-Caparrini, F.: A polynomial complexity class in P systems using membrane division, *Proceedings of the 5th Workshop on Descriptional Complexity of Formal Systems, DCFS 2003*, E. Csuhaj-Varjú, C. Kintala, D. Wotschke and Gy. Vaszyl (eds.), 2003, 284-294.

[17] A. Riscos-Núñez. *Cellular programming: efficient resolution of NP–complete numerical problems*. Ph. D. Thesis, University of Seville, Department of Computer Science and Artificial Intelligence, 2004.

[18] R. L. Rivest, A. Shamir, L. M. Adleman. A Method for Obtaining Digital Signatures and Public–Key Cryptosystems. *Communications of the ACM*, 21(2):120–126, February 1978.

[19] P systems web page `http://psystems.disco.unimib.it/`