# Solving Numerical NP-complete Problems by Spiking Neural P Systems with Pre–computed Resources

Miguel A. Gutiérrez-Naranjo[1], Alberto Leporati[2]

[1] Research Group on Natural Computing
Department of Computer Science and Artificial Intelligence
University of Sevilla
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
Viale Sarca 336/14, 20126 Milano, Italy
E-mail: alberto.leporati@unimib.it

**Summary.** Recently we have considered the possibility of using spiking neural P systems for solving computationally hard problems, under the assumption that some (possibly exponentially large) pre-computed resources are given in advance. In this paper we continue this research line, and we investigate the possibility of solving numerical **NP**-complete problems such as Subset Sum. In particular, we first propose a semi–uniform family of spiking neural P systems in which every system solves a specified instance of Subset Sum. Then, we exploit a technique used to calculate Iterated Addition with boolean circuits to obtain a uniform family of spiking neural P systems in which every system is able to solve all the instances of Subset Sum of a fixed size. All the systems here considered are deterministic, but their size generally grows exponentially with respect to the instance size.

## 1 Introduction

Spiking neural P systems (SN P systems, for short) have been introduced in [11] as a new class of distributed and parallel computing devices, inspired by the neurophysiological behavior of neurons sending electrical impulses (*spikes*) along axons to other neurons. SN P systems can also be viewed as an evolution of P systems [24, 25, 27, 28] (the latest information can be found in [34]) corresponding to a shift from *cell-like* to *neural-like* architectures. We recall that this biological background has already led to several models in the area of neural computation, e.g., see [19, 20, 8].

In SN P systems the cells (also called *neurons*) are placed in the nodes of a directed graph, called the *synapse graph*. The contents of each neuron consist of a number of copies of a single object type, called the *spike*. Every cell may also contain a number of *firing* and *forgetting* rules. Firing rules allow a neuron to send information to other neurons in the form of electrical impulses (also called spikes) which are accumulated at the target cell. The applicability of each rule is determined by checking the contents of the neuron against a regular set associated with the rule. In each time unit, if a neuron can use one of its rules, then one of such rules must be used. If two or more rules could be applied, then only one of them is nondeterministically chosen. Thus, the rules are used in the sequential manner in each neuron, but neurons function in parallel with each other. Observe that, as usually happens in membrane computing, a global clock is assumed, marking the time for the whole system, hence the functioning of the system is synchronized. When a cell sends out spikes it becomes "closed" (inactive) for a specified period of time, that reflects the refractory period of biological neurons. During this period, the neuron does not accept new inputs and cannot "fire" (that is, emit spikes). Another important feature of biological neurons is that the length of the axon may cause a time delay before a spike arrives at the target. In SN P systems this delay is modeled by associating a delay parameter to each rule which occurs in the system. If no firing rule can be applied in a neuron, there may be the possibility to apply a *forgetting rule*, that removes from the neuron a predefined number of spikes.

Formally, a *spiking neural membrane system* (SN P system, for short) of degree $m \geq 1$, as defined in [10] in the computing version (i.e., able to take an input and provide and output), is a construct of the form

$$\Pi = (O, \sigma_1, \sigma_2, \ldots, \sigma_m, syn, in, out),$$

where:

1. $O = \{a\}$ is the singleton alphabet ($a$ is called *spike*);
2. $\sigma_1, \sigma_2, \ldots, \sigma_m$ are *neurons*, of the form $\sigma_i = (n_i, R_i)$, $1 \leq i \leq m$, where:
   a) $n_i \geq 0$ is the *initial number of spikes* contained in $\sigma_i$;
   b) $R_i$ is a finite set of *rules* of the following two forms:
      (1) *firing* (also *spiking)* rules $E/a^c \rightarrow a; d$, where $E$ is a regular expression over $a$, and $c \geq 1$, $d \geq 0$ are integer numbers; if $E = a^c$, then it is usually written in the following simplified form: $a^c \rightarrow a; d$;
      (2) *forgetting* rules $a^s \rightarrow \lambda$, for $s \geq 1$, with the restriction that for each rule $E/a^c \rightarrow a; d$ of type (1) from $R_i$, we have $a^s \notin L(E)$ (the regular language defined by $E$);
3. $syn \subseteq \{1, 2, \ldots, m\} \times \{1, 2, \ldots, m\}$, with $(i, i) \notin syn$ for $1 \leq i \leq m$, is the directed graph of *synapses* between neurons;
4. $in, out \in \{1, 2, \ldots, m\}$ indicate the *input* and the *output* neurons of $\Pi$.

A firing rule $E/a^c \rightarrow a; d \in R_i$ can be applied in neuron $\sigma_i$ if it contains $k \geq c$ spikes, and $a^k \in L(E)$. The execution of this rule removes $c$ spikes from $\sigma_i$ (thus

leaving $k-c$ spikes), and prepares one spike to be delivered to all the neurons $\sigma_j$ such that $(i,j) \in syn$. If $d=0$ then the spike is immediately emitted, otherwise it is emitted after $d$ computation steps of the system. As stated above, during these $d$ computation steps the neuron is *closed*, and it cannot receive new spikes (if a neuron has a synapse to a closed neuron and tries to send a spike along it, then that particular spike is lost), and cannot fire (and even select) rules. A *forgetting* rule $a^s \to \lambda$ can be applied in neuron $\sigma_i$ if it contains *exactly* $s$ spikes, and no firing rules are applicable. The execution of this rule simply removes all the $s$ spikes from $\sigma_i$.

The *initial configuration* of the system is described by the numbers $n_1, n_2, \ldots, n_m$ of spikes present in each neuron, with all neurons being open. During the computation, a configuration is described by both the contents of each neuron and its *state*, which can be expressed as the number of steps to wait until it becomes open (zero if the neuron is already open). Thus, $\langle r_1/t_1, \ldots, r_m/t_m \rangle$ is the configuration where neuron $\sigma_i$ contains $r_i \geq 0$ spikes and it will be open after $t_i \geq 0$ steps, for $i = 1, 2, \ldots, m$; with this notation, the initial configuration of the system is $C_0 = \langle n_1/0, \ldots, n_m/0 \rangle$.

A *computation* starts in the initial configuration. In order to compute a function $f : \mathbb{N} \to \mathbb{N}$ (functions of the kind $f : \mathbb{N}^\alpha \to \mathbb{N}^\beta$, for any fixed pair of integers $\alpha \geq 1$ and $\beta \geq 1$, can also be computed by using appropriate bijections from $\mathbb{N}^\alpha$ and $\mathbb{N}^\beta$ to $\mathbb{N}$), a positive integer number is given in input to a specified *input neuron*. In the original model, as well as in some early variants, the number is encoded as the interval of time steps elapsed between the insertion of two spikes into the neuron. To pass from a configuration to another one, for each neuron a rule is chosen among the set of applicable rules, and is executed. Generally, a computation may not halt. However, in any case the output of the system is considered to be the time elapsed between the arrival of two spikes in a designated *output cell*. Other possibilities exist to encode input and output numbers, as discussed in [10]: as the number of spikes contained in a given neuron at the beginning (resp., the end) of the computation, as the number of spikes fired in a given interval of time, etc.

A useful extension to the standard model defined above, already considered in [15, 16, 17, 12], is to use several input neurons, so that the introduction of the encoding of an instance of the problem to be solved can be done in a faster way, introducing parts of the code in parallel in various input neurons. Formally, we can define an SN P system of degree $(m, \ell)$, with $m \geq 1$ and $0 \leq \ell \leq m$, just like a standard SN P system of degree $m$, the only difference being that now there are $\ell$ input neurons denoted by $in_1, \ldots, in_\ell$. A *valid input* for an SN P system of degree $(m, \ell)$ is a set of $\ell$ binary sequences, that collectively encode an instance of a problem.

The previous definitions cover many types of systems/behaviors. By neglecting the output neuron we can define *accepting* SN P systems, in which the natural number (or the vector of natural numbers, in the case of systems having $\ell > 1$ input neurons) given in input is accepted if the computation halts. On the other hand, by ignoring the input neuron (and thus starting from a predefined input configuration)

we can define *generative* SN P systems. In [11] it was shown that generative SN P systems are universal, that is, can generate any recursively enumerable set of natural numbers. Moreover, a characterization of semilinear sets was obtained by spiking neural P systems with a bounded number of spikes in the neurons. These results can be obtained also for some restricted forms of SN P systems: [9] shows that one of the following features can be avoided while keeping universality: time delay greater than 0, forgetting rules, outdegree of the synapse graph greater than 2, and regular expressions of complex form. In [6] it is shown that universality is kept even if we remove some combinations of two of the above features. Finally, in [29] the behavior of SN P systems on infinite strings and the generation of infinite sequences of 0 and 1 was investigated, whereas in [3] SN P systems were studied as language generators (over the binary alphabet $\{0, 1\}$).

Spiking neural P systems can also be used to solve decision problems, both in a *semi–uniform* and in a *uniform* way. When solving a problem $Q$ in the semi–uniform setting, for each specified instance $\mathcal{I}$ of $Q$ we build an SN P system $\Pi_{Q,\mathcal{I}}$, whose structure and initial configuration depend upon $\mathcal{I}$, that halts (or emits a specified number of spikes in a given interval of time) if and only if $\mathcal{I}$ is a positive instance of $Q$. On the other hand, a uniform solution of $Q$ consists in a family $\{\Pi_Q(n)\}_{n\in\mathbb{N}}$ of SN P systems such that, when having an instance $\mathcal{I} \in Q$ of size $n$, we introduce a polynomial (in $n$) number of spikes in a designated (set of) input neuron(s) of $\Pi_Q(n)$ and the computation halts (or, alternatively, a specified number of spikes is emitted in a given interval of time) if and only if $\mathcal{I}$ is a positive instance. The preference for uniform solutions over semi–uniform ones is given by the fact that they are more strictly related to the structure of the problem, rather than to specific instances. If the instances of a problem $Q$ depend upon two parameters (as is the case of Subset Sum, where $n + 1$ is the number of integer values contained into the generic instance $(V = \{v_1, v_2, \ldots, v_n\}, S)$, and $k$ is the number of bits needed to represent each of these values), then we will denote the family of SN P systems that solves Q by $\{\Pi_Q(\langle n, k\rangle)\}_{n,k\in\mathbb{N}}$, where $\langle n, k\rangle$ indicates the positive integer number obtained by applying an appropriate bijection (for example, Cantor's pairing) from $\mathbb{N}^2$ to $\mathbb{N}$.

The present paper considers SN P systems for solving decision problems, continuing the papers [17], [16] and [15], where we dealt with the **NP**-complete decision problems Subset Sum, sat and 3-sat. For all these problems, constant time and polynomial time solutions were provided by using SN P systems constructed both in the semi-uniform and in the uniform setting, working in a non-deterministic way, and also using a series of ingredients added to SN P systems of the standard form: rules that produce several spikes at a time, the possibility to have a choice between spiking rules and forgetting rules, forgetting rules controlled by regular expressions, rules applied in the maximal parallel way, etc. Here we consider a different situation: we assume that a pre-computed (standard) SN P system is given in advance, possibly having an exponential size with respect to the size of the instances of the problem we want to solve, and we provide a semi–uniform and a uniform constructions that solve Subset Sum in a polynomial time. All the

systems we will propose work in a *deterministic* way. Note that this setting was already considered in [12], where polynomial time uniform solutions to SAT and 3-SAT were provided.

An important observation is that we will not specify how our pre-computed systems could be built. However, we require that such systems have a *regular* structure, and that they do not contain neither "hidden information" that simplify the solution of specific instances, nor an encoding of all possible solutions (that is, an exponential amount of information that allows to cheat while solving the instances of the problem). These requirements were inspired by open problem Q27 in [27]. Let us note in passing that the regularity of the structure of the system is related to the concept of *uniformity*, that in some sense measures the difficulty of constructing the system. For example, when considering families $\{C(n)\}_{n \in \mathbb{N}}$ of boolean circuits, or other computing devices whose number of inputs depends upon an integer parameter $n \geq 1$, it is required that for each $n \in \mathbb{N}$ a "reasonable" description (see [2] for further discussion on the meaning of the term "reasonable" in this context) of $C(n)$, the circuit of the family which has $n$ inputs, can be produced in polynomial time and logarithmic space (with respect to $n$) by a deterministic Turing machine whose input is $1^n$, the unary representation of $n$. In this paper we will not delve further into the details concerning uniformity; we just rely on reader's intuition, by stating that it should be possible to build the entire structure of the system using only a polynomial amount of information and a controlled replication mechanism, as it already happens in P systems with cell division.

The paper is organized as follows. In section 2 we recall the definition of the SUBSET SUM problem, as well as a classical solution algorithm based on the dynamic programming paradigm. In section 3 we elaborate such an algorithm to obtain a family of SN P systems that solves SUBSET SUM in a semi–uniform way. In section 4 we propose a completely different construction, that allows to uniformly solve all the instances of SUBSET SUM of any specified size; the instances are provided in input to the systems of the family by specifying their values in binary form. Finally, section 5 contains the conclusions and some directions for further research.

## 2 The SUBSET SUM Problem

SUBSET SUM is one of the most known **NP**-complete decision problems. We can state it as follows, in a form which is equivalent to the one given in [7, p. 223].

**Problem 1.** NAME: SUBSET SUM.

- INSTANCE: a (multi)set $V = \{v_1, v_2, \ldots, v_n\}$ of positive integer numbers, and a positive integer number $S$.
- QUESTION: is there a sub(multi)set $B \subseteq V$ such that $\sum_{b \in B} b = S$?

The following well known algorithm [5] solves SUBSET SUM by using the Dynamic Programming technique. In particular, the algorithm returns 1 on positive instances, and 0 on negative instances.

```
SUBSET SUM({v_1, v_2, ..., v_n}, S)
for j ← 0 to S
    do M[1, j] ← 0
M[1, 0] ← M[1, v_1] ← 1
for i ← 2 to n
     do for j ← 0 to S
              do M[i, j] ← M[i − 1, j]
                  if j ≥ v_i and M[i − 1, j − v_i] > M[i, j]
                     then M[i, j] ← M[i − 1, j − v_i]
return M[n, S]
```

In order to look for a subset $B \subseteq V$ such that $\sum_{b \in B} b = S$, the algorithm uses an $n \times (S+1)$ 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 $\{v_1, v_2, \ldots, v_i\}$ whose elements sum up to $j$. The given instance of SUBSET SUM is thus a positive instance if and only if $M[n, S] = 1$ 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 $n(S+1) = \Theta(nS)$. This means that the difficulty of the problem depends on the value of $S$, as well as on the magnitude of the values in $V$. In fact, let $K = max\{v_1, v_2, \ldots, v_n, S\}$. If $K$ is polynomially bounded with respect to $n$, then the above algorithm works in polynomial time. On the other hand, if $K$ is exponential with respect to $n$, say $K = 2^n$, then the above algorithm may work in exponential time and space. This behavior is usually referred to in the literature by telling that SUBSET SUM is a *pseudo–polynomial* **NP**–complete problem.

The fact that in general the running time of the above algorithm is not polynomial can be immediately understood by comparing its time complexity with the instance size. The usual size for the instances of SUBSET SUM is $\Theta(n \log K)$, since for conciseness every "reasonable" encoding is assumed to represent each element of $V$ (as well as $S$) 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 $S$ and all the integer numbers which occur in $V$. 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. Indeed, SUBSET SUM is not **NP**-complete in the *strong sense*, meaning that it does

not remain **NP**-complete when we represent its instances in unary form [7]. Stated otherwise, strongly **NP**-complete problems remain **NP**-complete even when the numbers contained into their instances are small.

As a consequence of these observations, the SN P systems that we will consider in section 4 will take in input the instances of SUBSET SUM as $n+1$ strings encoded in binary form, where the length of each string will be $k = \log K$. Before presenting the uniform solution of section 4, in the next section we first elaborate the above dynamic programming algorithm to provide a semi–uniform family of SN P systems that solves the SUBSET SUM problem.

## 3 A Semi–uniform Solution to SUBSET SUM

Let $SS(n,k)$ denote the set of instances of SUBSET SUM which can be built by using $n+1$ positive $k$-bit integer numbers. In this section we present a semi–uniform family $\{\Pi(\mathcal{I})\}_{\mathcal{I} \in SS(n,k)}$ of SN P systems such that for every $\mathcal{I} \in SS(n,k)$ the system $\Pi(\mathcal{I})$ determines whether $\mathcal{I} = (\{v_1, v_2, \ldots, v_n\}, S)$ is a positive instance of SUBSET SUM. The size of $\Pi(\mathcal{I})$ will be $\Theta(nS)$, hence exponential with respect to the instance size. However, the computation time of $\Pi(\mathcal{I})$ will be linear in $n$ and independent of $k$.

System $\Pi(\mathcal{I})$ is depicted in Figure 1 in a schematic way. The system is composed by $n$ layers, horizontally arranged, one for each iteration of the dynamic programming algorithm illustrated in the previous section. The computation starts in the first (the uppermost) layer, and proceeds downwards until the lowest (i.e., the $n$-th) layer has been reached. The neurons of the first layer contain the firing rule $a \to a; 0$, that propagates the spikes eventually contained in these neurons to the appropriate neurons of the second layer. All the other neurons, from layer 2 down to layer $n$, contain two firing rules:

$$a \to a; 0 \qquad \text{and} \qquad a^2 \to a; 0$$

that make the neurons operate like OR boolean gates.

The connections among the neurons depend upon the instance $\mathcal{I} = (\{v_1, v_2, \ldots, v_n\}, S)$ of SUBSET SUM to be solved. Precisely, to determine the value of $M[i,j]$ in the above algorithm we need to compute the maximum between the values $M[i-1,j]$ and $M[i-1,j-v_i]$, provided that $j - v_i \geq 0$, otherwise we put $M[i,j]$ equal to $M[i-1,j]$. The rationale behind these formulas is the following: as stated above, $M[i,j]$ has to be set to 1 if and only if there exists a subset of $\{v_1, v_2, \ldots, v_i\}$ such that the sum of its elements is equal to $j$. Thus we have two possibilities: either the subset contains $v_i$, or not. In the former case, there must be a subset of $\{v_1, v_2, \ldots, v_{i-1}\}$ such that the sum of its elements is equal to $j - v_i$ (that is, $M[i-1, j-v_i]$ must be 1); in the latter case, there must be a subset of $\{v_1, v_2, \ldots, v_{i-1}\}$ whose elements sum up to $j$ (that is, $M[i-1,j] = 1$). If $j < v_i$ then clearly $v_i$ cannot be in any subset of $\{v_1, v_2, \ldots, v_i\}$ whose sum is equal to $j$, and thus in this case we only check the value of $M[i-1,j]$. If $i = 1$ then these

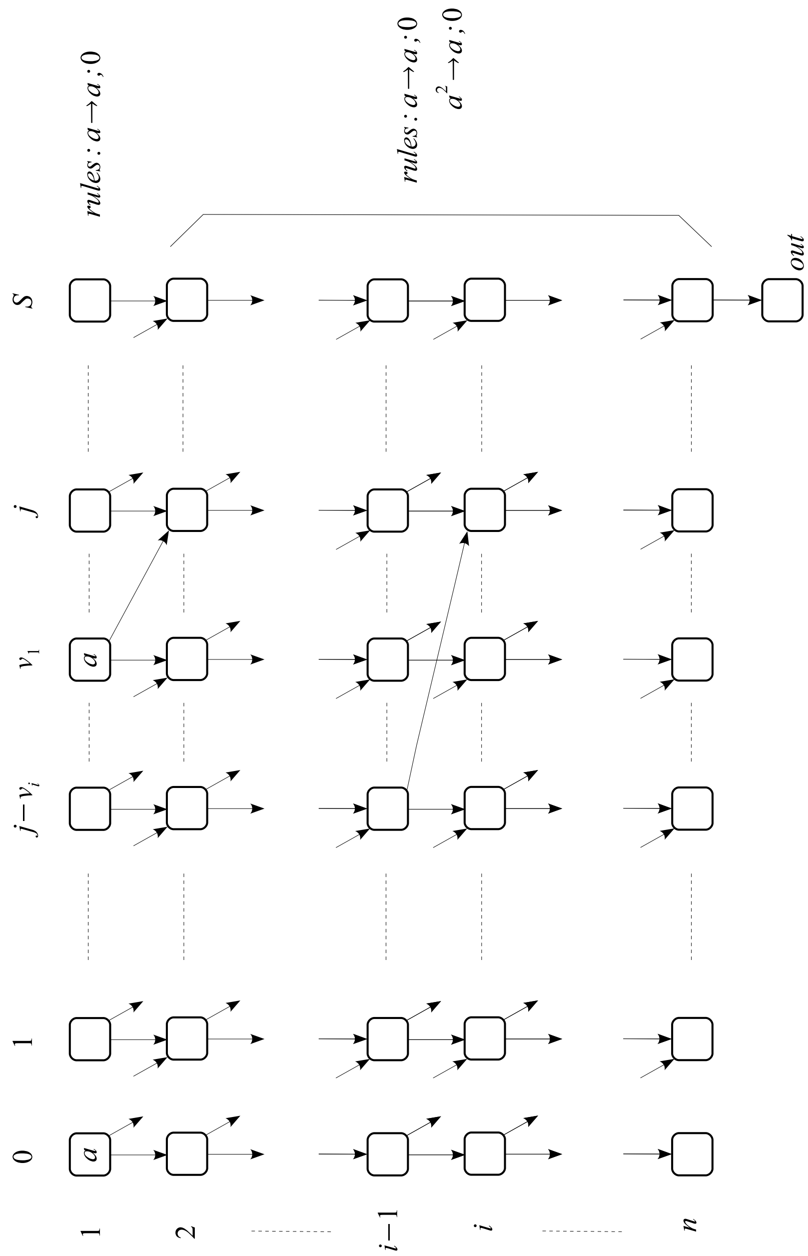


**Fig. 1.** A schematic view of the system $\Pi(\mathcal{I})$ used to solve a specified instance $\mathcal{I} = (\{v_1, v_2, \ldots, v_n\}, S)$ of Subset Sum, where each of the values $v_1, v_2, \ldots, v_n, S$ is a $k$-bit positive integer number

formulas cannot clearly be applied. However, we note that the only two subsets of $\{v_1\}$ we can build are the empty set $\emptyset$ and $\{v_1\}$ itself, hence $M[1,0] = M[1,v_1] = 1$ whereas $M[1,j] = 0$ for all $j \notin \{0, v_1\}$. Since the admissible values of $M[i-1,j]$ and of $M[i-1,j-v_i]$ are 0 and 1, computing the maximum is the same as computing a logical OR. In the system depicted in Figure 1, the $j$-th neuron from the left, $0 \leq j \leq S$, corresponds to $M[i,j]$. We denote 1 (resp., 0) by the presence (resp., absence) of a spike. Such a neuron, for $1 \leq i \leq n$, has a synapse going to the

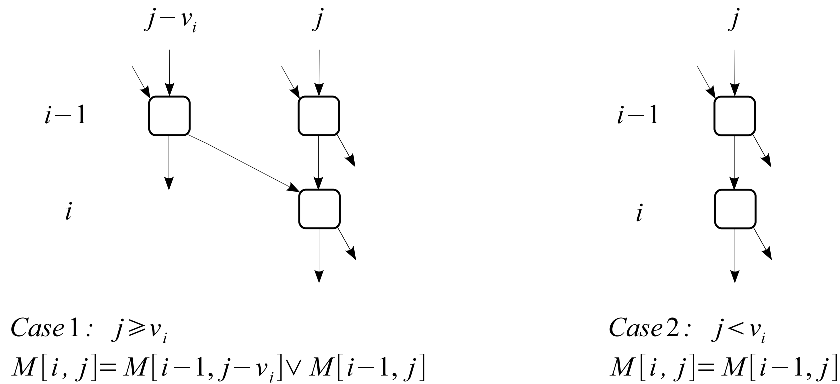


**Fig. 2.** The two cases to be considered to compute the value of $M[i,j]$

neuron that corresponds to $M[i+1,j]$, and possibly (if $v_{i+1} + j \leq S$) another synapse going to the neuron that corresponds to $M[i+1, j+v_{i+1}]$ (see Figure 2). In the last layer, only the neuron that corresponds to $M[n,S]$ has a synapse going to a neuron named *out*, which is the output neuron and does not contain any rule.

In the initial configuration of the system, one spike is put in the neurons that correspond to $M[1,0]$ and $M[1,v_1]$; all the other neurons are empty. During the $i$-th computation step, with $1 \leq i \leq n-1$, the neurons in the $i$-th layer perform their computation, and send the corresponding result to the appropriate neurons of the next layer. At the $n$-th computation step, all the neurons in the last layer send the spikes produced by them to the environment (where they are lost) but the rightmost neuron, that sends the result of its computation (0 or 1 spikes) to neuron *out*. Hence, the instance $\mathcal{I}$ of SUBSET SUM represented by the structure and the initial configuration of $\Pi(\mathcal{I})$ is positive if and only if one spike arrives in neuron *out* during the $n$-th computation step. After the result of the computation (0 or 1 spikes in neuron *out*) has been produced, the computation halts and the spike eventually contained in neuron *out* remains there. The computation time of $\Pi(\mathcal{I})$ is linear in $n$, independent of the values $v_1, v_2, \ldots, v_n$ and $S$ contained in $\mathcal{I}$, but the number of neurons in the system is $n(S+1)+1$, which is exponential with respect to the instance size. This last fact would be considered unacceptable in traditional complexity theory, but recall that in this paper (as well as in [12]) we

are assuming that exponential size resources — encoded in exponential size SN P systems of regular structure — are admitted.

The structure of $\Pi(\mathcal{I})$ is indeed very regular: all the instances composed by $n$ integer values plus a required sum equal to $S$ produce systems having $n$ layers, each composed by $S+1$ neurons. The values $v_1, v_2, \ldots, v_n$ determine some of the connections between the neurons (all the other connections go from every neuron in each layer to the neuron that occurs in the same position in the next layer); precisely, for all $i \in \{1, 2, \ldots, n-1\}$ the value $v_i$ determines the presencce of a synapse from every $j$-th neuron in layer $i$, such that $j + v_{i+1} \leq S$, to the $(j + v_{i+1})$-th neuron of layer $i+1$. Value $v_1$ also determines the neuron in the first layer (apart from the leftmost) that will receive one spike in the initial configuration. An open question, that we will not address in this paper, is: what kind of operations are needed to augment the power of deterministic Turing machines so that, given any instance $\mathcal{I}$ of Subset Sum, the new machine is able to produce a "reasonable" description of $\Pi(\mathcal{I})$ in a polynomial time? Note that in this case we should also recast the meaning of the term "reasonable", since in [7] this notion concerns only polynomial size constructions.

## 4 A Uniform Solution to Subset Sum

Let us present now a uniform family $\{\Pi(\langle n, k\rangle)\}_{n,k \in \mathbb{N}}$ of SN P systems that solves the Subset Sum problem in a uniform way. Precisely, for all $n, k \in \mathbb{N}$ the system $\Pi(\langle n, k\rangle)$ will solve all the instances $\mathcal{I} \in SS(n, k)$ which are composed by $n+1$ positive $k$-bit integer numbers. Such instances are provided in input in binary form, as a sequence of $(n+1)k$ bits that are fed to the system in parallel (which means that each bit is inserted into an appropriate input neuron).

Figure 3 depicts the system $\Pi(\langle n, k\rangle)$ in a schematic way. The instance $\mathcal{I} \in SS(n, k)$ is inserted into the leftmost neurons, which are labelled with a name that indicates the bit which has to be inserted. These neurons simply propagate their spikes to subsystems $\textsc{Sum}_1$, $\textsc{Sum}_2$, ..., $\textsc{Sum}_{2^n-1}$ by using a firing rule of type $a \to a; 0$. The Sum subsystems are bijectively associated to every possible non-empty subset of $\{v_1, v_2, \ldots, v_n\}$. As the name indicates, every Sum subsystem computes the sum of the elements of the corresponding subset of $\{v_1, v_2, \ldots, v_n\}$, and thus the synapses outgoing from the leftmost neurons reflect this situation; that is, a synapse leaving from neuron $v_{i,j}$, $1 \leq i \leq n$ and $1 \leq j \leq k$, reaches the subsystem $\textsc{Sum}_\ell$ if and only if value $v_i$ is involved in the sum computed by $\textsc{Sum}_\ell$. The sums are computed in binary (we will return later on this point) and hence every Sum subsystem produces a bit vector as a result. This vector is then compared with the sequence of bits that compose the value $S$; the comparison is performed by the Compare subsystems, that produce a 1 (that is, a spike) if and only if the two sequences given in input are equal. Recall that two integer numbers expressed in binary form are equal if and only if their binary expansions are equal; the comparison thus amounts to compute the following boolean function:

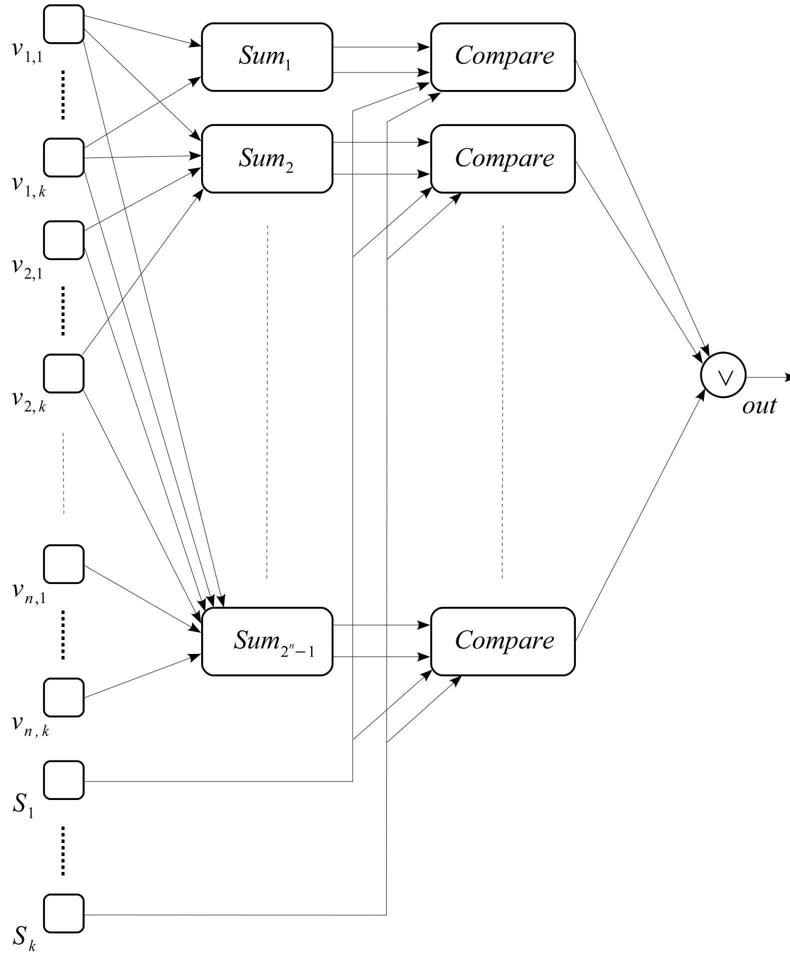


**Fig. 3.** A schematic view of the system $\Pi(\langle n, k\rangle)$ used to solve all the instances of SUBSET SUM which are composed by $n+1$ positive $k$-bit integer numbers

$$\text{COMPARE}(x_0, \ldots, x_{k-1}, y_0, \ldots, y_{k-1}) = \bigwedge_{i=0}^{k-1} \big(\neg(x_i \oplus y_i)\big) = \neg \left( \bigvee_{i=0}^{k-1} (x_i \oplus y_i) \right)$$

where $x = \sum_{i=0}^{k-1} x_i 2^i$ and $y = \sum_{i=0}^{k-1} y_i 2^i$ are the numbers to be compared, and $\vee, \wedge, \neg, \oplus$ denote the OR, AND, NOT and XOR (also PARITY) logical connectives, respectively. Figure 4 shows an SN P (sub)system which can be used to compute this function. This subsystem works as follows. Bits $x_i$ and $y_i$ are XORed by the neurons depicted on the left of the figure. The neuron labelled with $\vee$ computes the logical OR of its inputs: precisely, it emits one spike if and only if at least one spike enters into the neuron. Neuron *res* receives the output produced by $\vee$ and

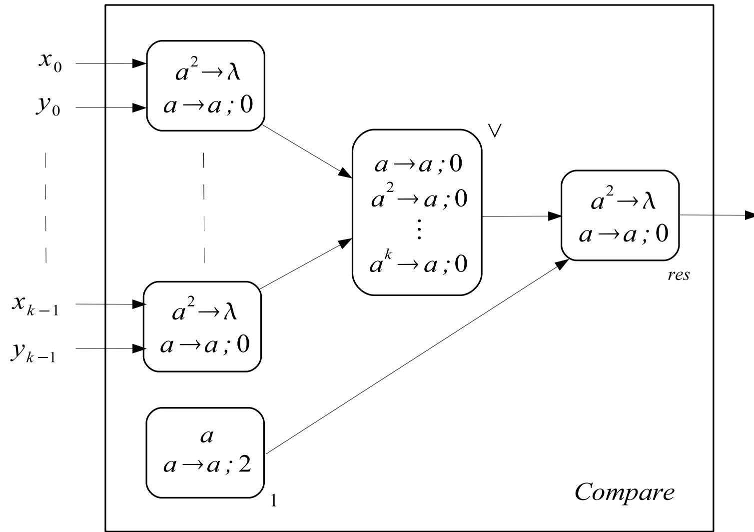


**Fig. 4.** The subsystem of $\Pi(\langle n, k \rangle)$ that compares two $k$-bit natural numbers

computes its logical negation (NOT). In order to be able to produce one spike if no spikes come from $res$, we use one auxiliary neuron that sends to $res$ one spike after two computation steps. Indeed, the delay of the rule contained in neuron 1 (whose contents will be initialized with one spike at the beginning of the computation) should be set in order to make neuron 1 fire exactly when the results computed by the SUM subsystems reach the COMPARE subsystems (plus two steps).

Observe that $S$ is a $k$-bit number, just like $v_1, v_2, \ldots, v_n$, and thus if we sum a subset of these latter values we could easily end up with a result that needs more than $k$ bits to be expressed in binary form, thus complicating a little bit the comparison with $S$. However, recall that $k = \log K$ where $K = \max\{v_1, v_2, \ldots, v_n, S\}$, and thus for reasonable values it is very likely that a large portion of the most significant bits of $v_1, v_2, \ldots, v_n$ is equal to zero. Anyway, just to be cautious, since the COMPARE subsystems perform a $k$-bit comparison, we should avoid the situation in which a SUM subsystem produces an $m$-bit sequence, with $m > k$, such that its $k$ less significant bits coincide with the bits that compose $S$. Fortunately it is easy to check whether we are in this situation: we just design each of the SUM subsystems so that it produces an $m$-bit sequence, where $m = k + \lceil \log_2 n \rceil$ (in facts the maximum integer number that we can represent using $k$ bits is $2^k - 1$, so if we sum $n$ of such numbers we obtain a result which is less than $n2^k$, that requires $k + \lceil \log_2 n \rceil$ bits to be represented in binary form), and we check that the $m - k$ most significant bits of this sequence are all zero. This is easily done by sending these bits (that is, the corresponding spikes) to a neuron whose contents (the presence of at least one spike) signals to the user of the system that the above situation occured.

The core of the system is composed by the SUM subsystems. In a generic SUM subsystem, $r$ values from the set $\{v_1, v_2, \ldots, v_n\}$ have to be summed together, and this sum has to be performed in polynomial time. If $r = 2$ then we can use either a traditional or a carry look-ahead adder [32, p. 6]. Let $x = \sum_{i=0}^{k-1} x_i 2^i$ and $y = \sum_{i=0}^{k-1} y_i 2^i$ be the two $k$-bit binary numbers to be summed. We denote by $s_0, s_1, \ldots, s_k$ the bits of the sum, and by $c_0, c_1, \ldots, c_k$ the carries generated during the addition. The traditional addition algorithm (which can be trivially implemented using a boolean circuit) puts $s_0 = x_0 \oplus y_0$, $c_0 = 0$, and then defines inductively $c_i = (x_{i-1} \wedge y_{i-1}) \vee (x_{i-1} \wedge c_{i-1}) \vee (y_{i-1} \wedge c_{i-1})$ (that is, $c_i = 1$ if and only if at least two of $x_{i-1}$, $y_{i-1}$, $c_{i-1}$ is 1), $s_i = x_i \oplus y_i \oplus c_i$ for $1 \leq i < k$, and $s_k = c_k$. Such an algorithm sums the two $k$-bit integer numbers in $O(k)$ steps.

A carry look-ahead adder operates by computing the values of the carries $c_i$ in a finite number of steps, indepedent of $k$, starting from the values of $x_0, x_1, \ldots, x_{k-1}$ and $y_0, y_1, \ldots, y_{k-1}$. The crucial observation is that a carry is generated at position $i$ if and only if both input bits $x_i$ and $y_i$ are 1, and a carry is eliminated at position $i$ if and only if both input bits $x_i$ and $y_i$ are 0. This observation yields to the following definitions: for $0 \leq i < k$, let:

$$g_i = x_i \wedge y_i \qquad \text{(position } i \textit{ generates} \text{ a carry)}$$
$$p_i = x_i \vee y_i \qquad \text{(position } i \textit{ propagates} \text{ a carry)}$$

Now, a carry ripples into position $i$ if and only if there exists a position $j < i$ where a carry is generated, and all positions in between propagate it. Formally:

$$c_i = \bigvee_{j=0}^{i-1} \left( g_j \wedge \bigwedge_{k=j+1}^{i-1} p_k \right) \qquad \text{for } 1 \leq i \leq k \tag{1}$$

Once we have computed the carries, the bits of the sum are computed as before: $s_0 = x_0 \oplus y_0$, $s_i = x_i \oplus y_i \oplus c_i$ for $1 \leq i < k$, and $s_k = c_k$. It is easily seen that the above formulas allow to compute all the $c_i$ in parallel, since they only depend on the input bits $x_0, x_1, \ldots, x_{k-1}$ and $y_0, y_1, \ldots, y_{k-1}$, in constant time: all $g_i$ and $p_i$ are computed in one step, and two more steps are needed to compute the ANDs and the ORs that appear in (1). By using XOR ($\oplus$) gates, all the bits of the sum are computed in one more step.

The boolean circuit that implements a carry look-ahead adder can be easily simulated by an SN P system, simply substituting every logical gate with an appropriate neuron. Figure 5 shows this mapping from AND, OR and XOR gates to neurons. When needed, for example when the output value of a gate has to skip one or more layers and go directly to one of the subsequent layers, for synchronization purposes we can also use delay neurons, that contain the rule $a \to a; d$ for an appropriate value of $d$. It is clear that the size of the SN P system thus obtained is polynomially related with the size of the simulated boolean circuit, and that if the simulated circuit performs its computations in constant time then also the SN P system performs its computations in constant time.

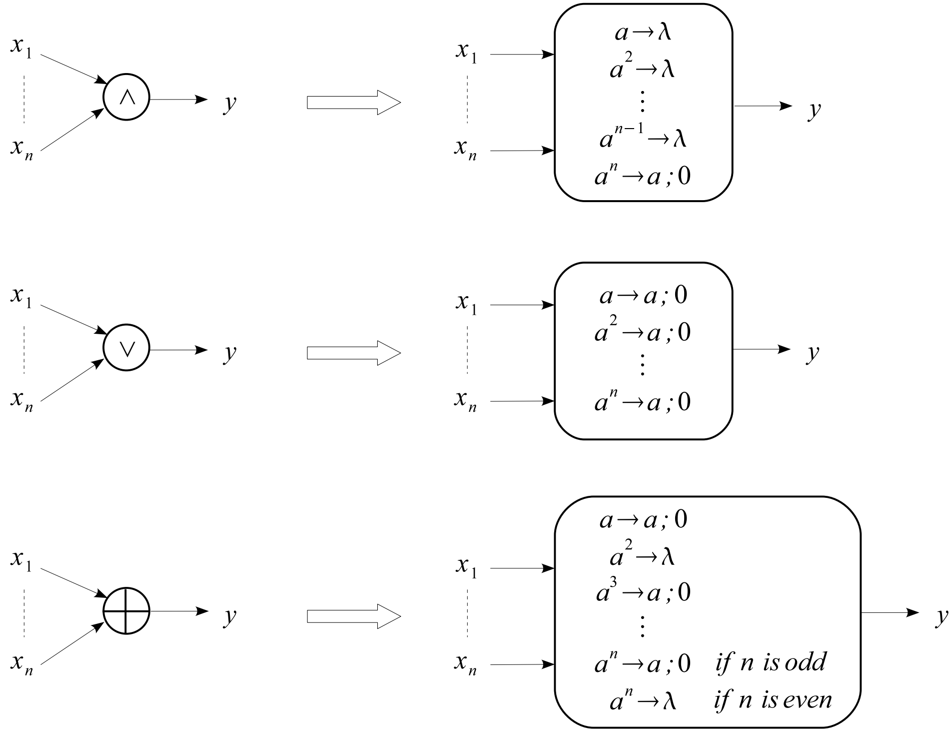


**Fig. 5.** Simulation of $n$-input AND, OR and XOR gates by means of single–neuron SN P systems

If we need to compute the sum of $r > 2$ binary numbers of length $k$, then a slightly more complicated construction is needed. As shown in [32, p. 13], while designing a boolean circuit that computes the ITERATED ADDITION (that is, the sum of $n$ natural numbers, each of $n$ bits), the addition of three $k$-bit binary numbers $a = \sum_{i=0}^{k-1} a_i 2^i$, $b = \sum_{i=0}^{k-1} b_i 2^i$ and $c = \sum_{i=0}^{k-1} c_i 2^i$ can be reduced to the addition of two $(k+1)$-bit numbers $e$ and $d$, by defining:

$$e_0 = 0$$
$$e_i = (a_{i-1} \wedge b_{i-1}) \vee (a_{i-1} \wedge c_{i-1}) \vee (b_{i-1} \wedge c_{i-1}) \qquad \text{for all } 1 \leq i \leq k$$
$$d_i = a_i \oplus b_i \oplus c_i \qquad \text{for } 0 \leq i < k$$
$$d_k = 0$$

The rationale behind these formulas is the following. If we look at a single position $i$, then we have to add $a_i$, $b_i$ and $c_i$. The result is given by the two bit number $e_{i+1}d_i$; bit $e_{i+1}$ is 1 if and only if at least two of the bits $a_i$, $b_i$ and $c_i$ are 1, and $d_i = 1$ if and only if an odd number of $a_i$, $b_i$ and $c_i$ is 1. We can thus conclude that $a + b + c = d + e$.

If we are given $r > 2$ binary numbers of length $k$, we can group them into three elements sets (plus one set with only one or two numbers, if $r$ is not a multiple of 3), and then compute for each set as just explained two numbers whose sum is equal to the sum of all the three numbers from the set. In this way we end up with $r'$ numbers of $k + 1$ bits each, where:

$$r' = \begin{cases} \frac{2}{3}\, r & \text{if } r \equiv 0 \mod 3 \\ \frac{2}{3}\,(r-1)+1 & \text{if } r \equiv 1 \mod 3 \\ \frac{2}{3}\,(r-2)+2 & \text{if } r \equiv 2 \mod 3 \end{cases}$$

In any case, if $r > 2$ then $r' \leq \frac{4}{5}\, r$. Thus, given $r$ numbers of $k$ bits each, by iterating this reduction procedure $O(\log r)$ times we end up with two numbers of $k + O(\log r)$ bits each. These two numbers can then be added using a carry look-ahead adder, as explained above. In the worst case, we have to add all the numbers from $\{v_1, v_2, \ldots, v_n\}$. The reduction process can thus be implemented by a $O(\log n)$ depth boolean circuit, since each reduction involves a constant depth (and bounded fan-in) circuit. At the end of the reduction process we have to add two $(k + O(\log n))$-bit numbers, which can be done by a boolean circuit of polynomial (quadratic in $k + O(\log n)$) size and constant depth. The fan-in of such a circuit is unbounded, and thus also the in-degree of the neurons of the SN P system that simulates it is unbounded. However, any unbounded fan-in AND or OR gate can be simulated by a polynomial size logarithmic depth circuit composed by bounded fan-in AND and OR gates, and thus we can conclude that the SUM subsystems can be implemented by polynomial size SN P systems which are composed by a logarithmic number of layers and whose in-degree is bounded (that is, constant). The same argumentation holds for the COMPARE subsystems: they can be implemented as polynomial size logarithmic depth OR/XOR circuits of bouded fan-in, and hence as polynomial size SN P systems composed by a logarithmic number of layers, each composed by constant in-degree neurons. Finally, the large OR that provides the output to the environment has $2^n - 1$ inputs, and thus it can be realized as an exponential size polynomial depth tree of bounded fan-in OR gates.

The system $\Pi(\langle n, k\rangle)$ thus obtained is able to solve all the instances $\mathcal{I} \in SS(n,k)$ of SUBSET SUM which can be expressed as sequences of $n+1$ natural numbers, each of $k$ bits. The family $\{\Pi(\langle n, k\rangle)\}_{n,k\in\mathbb{N}}$ thus constitutes a uniform solution to the SUBSET SUM problem. The size of $\Pi(\langle n, k\rangle)$ is exponential with respect to the instance size, but the computation time it takes to determine whether the instance $\mathcal{I} \in SS(n,k)$ is positive or not is polynomial with respect to $n$ and $k$. The fact that $\mathcal{I}$ is a positive instance is signalled by the emission of a spike from neuron *out*; in any case, after computing the solution the system halts. An important observation is that the system $\Pi(\langle n, k\rangle)$ has a very regular structure, and hence also in this case we can assume that it can be built in a polynomial time by a deterministic Turing machine whose computational power has been augmented by adding some controlled duplication instruction. Just like in the case of

the semi–uniform solution illustrated in the previous section, it is an open problem to determine how precisely this controlled duplication instruction should work.

## 5 Conclusions and Directions for Future Research

We have proposed two families of spiking neural P systems that solve Subset Sum, the well known **NP**-complete decision problem. The peculiarity and importance of Subset Sum, while trying to assess the computational power of a new computational device, is that it is a *numerical* **NP**-complete problem, and the difficulty of solving it depends upon the magnitude of the integer numbers that appear in its instances. To be precise it is not **NP**-complete in the strong sense, and hence the problem becomes easy to solve (through a well known algorithm which is based on the dynamic programming paradigm) when the numbers contained into the instances are small; equivalently, we can say that it becomes easy to solve when its instances are expressed in unary form.

For this reason, after showing in section 3 how for any instance of Subset Sum an SN P system that solves it can be built (thus working in the so called semi–uniform setting), in section 4 we have illustrated a uniform solution. Precisely, we have defined a family $\{\Pi(\langle n,k\rangle)\}_{n,k\in\mathbb{N}}$ of SN P systems such that for all $n,k \in \mathbb{N}$ the system $\Pi(\langle n,k\rangle)$ solves all the instances $\mathcal{I} \in SS(n,k)$ which are composed by $n+1$ positive $k$-bit integer numbers. The system $\Pi(\langle n,k\rangle)$ performs its computations in a time which is polynomial in $n$ and $k$, but its size generally grows exponentially with respect to these parameters. However the structure of $\Pi(\langle n,k\rangle)$ is so regular that we can assume that the system may be built in a polynomial time by a deterministic Turing machine whose computational power has been augmented by adding to its set of instructions some form of controlled duplication, that replicates (possibly substituting some pieces of the structure) part of the output it has built up to that moment.

It is important to note that, as proved in [16], an SN P system of polynomial size cannot solve in a deterministic way and in a polynomial time an **NP**-complete problem (unless **P** = **NP**), hence efficient solutions to **NP**-complete problems cannot be obtained without introducing features which enhance the efficiency (pre-computed resources, ways to exponentially grow the workspace during the computation, non-determinism, and so on). A more careful examination of such features – in particular, possible relations with the well known notions of *uniformity* traditionally studied in the theory of circuit complexity – is a research direction of a clear interest.

### Acknowledgements

The ideas exposed in this paper emerged during the Sixth Brainstorming Week on Membrane Computing, held in Seville from February 4 to February 8, 2008.

The first author wishes to acknowledge the support of the project TIN2006-13425 of the Ministerio de Educación y Ciencia of Spain, cofinanced by FEDER

funds, and the support of the project of excellence TIC-581 of the Junta de Andalucía.

The work of both authors was partially supported by the project "Azioni Integrate Italia-Spagna - Theory and Practice of Membrane Computing" (Acción Integrada Hispano-Italiana HI 2005-0194).

## References

1. A. Alhazov, M.J. Pérez-Jiménez. Uniform solution to QSAT using polarizationless active membranes. In M.A. Gutiérrez-Naranjo, Gh. Păun, A. Riscos-Núñez, F.J. Romero-Campero (eds.), *Fourth Brainstorming Week on Membrane Computing*, RGCN Report 02/2006, Sevilla University, Fénix Editora, Vol. I, 29–40.
2. J.L. Balcázar, J. Díaz, J. Gabarró. *Structural Complexity*. Voll. I and II, Springer-Verlag, Berlin, 1988–1990.
3. H. Chen, R. Freund, M. Ionescu, Gh. Păun, M.J. Pérez-Jiménez. On string languages generated by spiking neural P systems. In M.A. Gutiérrez–Naranjo, Gh. Păun, A. Riscos–Núñez, F.J. Romero–Campero (eds.), *Fourth Brainstorming Week on Membrane Computing*, RGCN Report 02/2006, Sevilla University, Fénix Editora, Vol. I, 169–194.
4. H. Chen, M. Ionescu, T.-O. Ishdorj. On the efficiency of spiking neural P systems. *Proc. 8th Intern. Conf. on Electronics, Information, and Communication*, Ulanbator, Mongolia, June 2006, 49–52.
5. T.H. Cormen, C.H. Leiserson, R.L. Rivest, *Introduction to Algorithms*. MIT Press, Boston, 1990.
6. M. García-Arnau, D. Peréz, A. Rodríguez-Patón, P. Sosík. Spiking Neural P Systems. Stronger Normal Forms. In M.A. Gutiérrez-Naranjo, Gh. Păun, A. Romero-Jiménez, A. Riscos-Núñez (eds.), *Fifth Brainstorming Week on Membrane Computing*, RGCN Report 01/2007, Sevilla University, Fénix Editora, 157–178.
7. M.R. Garey, D.S. Johnson. *Computers and Intractability. A Guide to the Theory on NP–Completeness*. W.H. Freeman and Company, 1979.
8. W. Gerstner, W. Kistler. *Spiking Neuron Models. Single Neurons, Populations, Plasticity*. Cambridge University Press, 2002.
9. O.H. Ibarra, A. Păun, Gh. Păun, A. Rodríguez-Patón, P. Sosík, S. Woodworth. Normal Forms for Spiking Neural P Systems. *Theoretical Computer Science*, 372(2-3):196–217, 2007.
10. M. Ionescu, A. Păun, Gh. Păun, M.J. Pérez-Jiménez. Computing with spiking neural P systems: Traces and small universal systems. In C. Mao, T. Yokomori, B.-T. Zhang (eds.), *DNA Computing,* $12^{\text{th}}$ *International Meeting on DNA Computing (DNA12)*, Revised Selected Papers, LNCS 4287, Springer-Verlag, Berlin, 1–16, 2006.
11. M. Ionescu, Gh. Păun, T. Yokomori. Spiking neural P systems. *Fundamenta Informaticae*, 71(2-3):279–308, 2006.
12. T.-O. Ishdorj, A. Leporati. Uniform Solutions to SAT and 3-SAT by Spiking Neural P Systems with Pre-computed Resources. *Natural Computing*, in press. A preliminary version appeared as Turku Centre for Computer Science – TUCS Report No. 876, 2008.
13. M.A. Gutiérrez-Naranjo, M.J. Pérez-Jiménez, A. Riscos-Núñez. A fast P system for finding a balanced 2–partition. *Soft Computing*, 9(9):673–678, 2005.

14. S.N. Krishna, R. Rama. A variant of P systems with active membranes: Solving **NP**-complete problems. *Romanian Journal of Information Science and Technology*, 2(4):357–367, 1999.
15. A. Leporati, G. Mauri, C. Zandron, Gh. Păun, M.J. Pérez-Jiménez. Uniform Solutions to SAT and Subset Sum by Spiking Neural P Systems. Submitted for publication, 2008.
16. A. Leporati, C. Zandron, C. Ferretti, G. Mauri. On the computational power of spiking neural P systems, *Intern. J. Unconventional Computing*, 2007, in press.
17. A. Leporati, C. Zandron, C. Ferretti, G. Mauri. Solving Numerical NP-complete Problems with Spiking Neural P Systems. In G. Eleftherakis, P. Kefalas, Gh. Păun, G. Rozenberg, A. Salomaa (eds.) *Membrane Computing, International Workshop, WMC8*, Selected and Invited Papers, LNCS 4860, Springer-Verlag, Berlin, 336–352, 2007.
18. A. Leporati, C. Zandron, M.A. Gutiérrez-Naranjo. P systems with input in binary form. *International Journal of Foundations of Computer Science*, 17(1):127–146, 2006.
19. W. Maass. Computing with spikes. *Special Issue on Foundations of Information Processing of TELEMATIK*, 8(1):32–36, 2002.
20. W. Maass, C. Bishop (eds.). *Pulsed Neural Networks*, MIT Press, Cambridge (MA), 1999.
21. A. Obtulowicz. Deterministic P systems for solving SAT problem. *Romanian Journal of Information Science and Technology*, 4(1–2):551–558, 2001.
22. C.H. Papadimitriou. *Computational Complexity*, Addison-Wesley, 1994.
23. A. Păun, Gh. Păun. Small universal spiking neural P systems. *BioSystems*, 90(1):48–60, 2007.
24. Gh. 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.
25. Gh. Păun. Computing with membranes. An introduction. *Bulletin of the EATCS*, 67:139–152, February 1999.
26. Gh. Păun. P systems with active membranes: Attacking **NP**-complete problems. *Journal of Automata, Languages and Combinatorics*, 6(1):75–90, 2001.
27. Gh. Păun. *Membrane Computing. An Introduction.* Springer–Verlag, Berlin, 2002.
28. Gh. Păun, G. Rozenberg. A guide to membrane computing. *Theoretical Computer Science*, 287(1):73–100, 2002.
29. Gh. Păun, M.J. Pérez-Jiménez, G. Rozenberg. Infinite spike trains in spiking neural P systems. Submitted for publication.
30. M.J. Pérez-Jiménez, A. Riscos-Núñez. Solving the SUBSET SUM problem by active membranes. *New Generation Computing*, 23(4):367–384, 2005.
31. M.J. Pérez-Jiménez, A. Riscos-Núñez. A linear–time solution to the KNAPSACK problem using P systems with active membranes. In C. Martín-Vide, Gh. Păun, G. Rozenberg, A. Salomaa (eds.), *Membrane Computing, 4th International Workshop, WMC 2003*, Revised Selected and Invited Papers, LNCS 2933, Springer-Verlag, Berlin, 250–268, 2004.
32. H. Vollmer, *Introduction to Circuit Complexity: A Uniform Approach.* Springer–Verlag, Berlin, 1999.
33. C. Zandron, C. Ferretti, G. Mauri. Solving **NP**-complete Problems Using P Systems with Active Membranes. In I. Antoniou, C.S. Calude, M.J. Dinneen (eds.), *Unconventional Models of Computation*, Springer-Verlag, Berlin, 289–301, 2000.
34. The P systems Web page: http://ppage.psystems.eu