On the Efficiency of Spiking Neural P Systems

Haiming Chen[⋆1], Mihai Ionescu[⋆⋆2], Tseren-Onolt Isdorj[3]

[1] Computer Science Laboratory, Institute of Software
Chinese Academy of Sciences
100080 Beijing, China
chm@ios.ac.cn

[2] Research Group on Mathematical Linguistics
Rovira i Virgili University
Pl. Imperial Tarraco, 1, 43005 Tarragona, Spain
armandmihai.ionescu@urv.cat

[3] Research Group on Natural Computing
Department of Computer Science and Artificial Intelligence
Sevilla University
Avda. Reina Mercedes s/n, 41012 Sevilla, Spain
tserren@yahoo.com

Summary. Spiking neural P systems were recently introduced in [4] and proved to be Turing complete as number computing devices. In this paper we show that these systems are also computationally efficient. Specifically, we present a variant of spiking neural P systems which have, in their initial configuration, an arbitrarily large number of inactive neurons which can be activated (in an exponential number) in polynomial time. Using this model of P systems we can deterministically solve the satisfiability problem (SAT) in constant time.

1 Introduction

Spiking neural P systems (in short, SN P systems) were recently introduced in [4] and proved [8, 9] to be Turing complete as number computing devices and also complete modulo direct and inverse morphisms in the case when they are used as language generators (see [1]). In this paper we show that if some precomputed resources (namely, the neurons disposed in a particular – initially inactive – structure) are considered, SN P systems having a self-activation behavior prove to be also computationally efficient, and can solve SAT, one of the best known NP complete problems, in constant time.

⋆ Work supported by the National Natural Science Foundation of China under Grants numbers 60573013 and 60421001.

⋆⋆ Work possible thanks to the fellowship "Programa Nacional para la Formación del Profesorado Universitario" from the Spanish Ministry of Education, Culture and Sport.

An SN P system consists of a set of neurons placed in the nodes of a graph, which communicate through signals (spikes) emitted along the synapses (edges of the graph) and controlled by firing and forgetting rules. Within the system the spikes are moved, created, or deleted (we consider only one type of objects in the system).

In this paper we address the question of the efficiency of solving problems by means of SN P systems, and to this aim we use systems with *precomputed resources*. The basic idea, initially mentioned in [3], is that, instead of producing in linear time an exponential workspace, we start from the beginning with an exponentially large precomputed workspace in the form of an exponentially large number of inactive neurons, which will be activated and used in constant time in our computation.

We assume a structure of neurons given "for free" as a result of a pre-computation whose duration does not matter – although the structure is not completely independent from the problem we want to solve. The neurons in the structure are initially inactive and become active immediately when a spike enters them.

This strategy corresponds to the well-known fact that the human brain contains a huge number of neurons out of which only a small part are currently used, and (some of) the inactive neurons can change their status as soon as they receive (electrical) signals from active neurons.

2 Preliminaries

The reader is assumed to be familiar with basic notions on languages, automata theory (for details, see [10]), and complexity theory (see [11]). We only briefly introduce some notations and notions used later in the paper.

For an alphabet V, V^* denotes the set of all finite strings of symbols from V; the empty string is denoted by λ, and the set of all nonempty strings over V is denoted by V^+. When $V = \{a\}$ is a singleton, then we write simply a^* and a^+ instead of $\{a\}^*, \{a\}^+$.

Regular expressions over a given alphabet V are constructed starting from λ and the symbols of V and using the operations of union, concatenation, and Kleene +, using parentheses when necessary for specifying the order of operations. Specifically, (i) λ and each $a \in V$ are regular expressions, (ii) if E_1, E_2 are regular expressions over V, then also $(E_1) \cup (E_2)$, $(E_1)(E_2)$, $(E_1)^+$ are regular expressions over V, and (iii) nothing else is a regular expression over V. With each expression E we associate a language $L(E)$, defined in the following way: (i) $L(\lambda) = \{\lambda\}$ and $L(a) = \{a\}$, for all $a \in V$, (ii) $L((E_1) \cup (E_2)) = L(E_1) \cup L(E_2)$, $L((E_1)(E_2)) = L(E_1)L(E_2)$, and $L((E_1)^+) = L(E_1)^+$, for all regular expressions E_1, E_2 over V. Non-necessary parentheses are omitted when writing a regular expression, and $(E)^+ \cup \{\lambda\}$ is written in the from $(E)^*$.

Let M be a deterministic Turing machine that halts on all inputs. The running time or time complexity of M is the function $f : \mathbf{N} \to \mathbf{N}$, where $f(n)$ is the

maximum number of steps that M uses on any input of length n. If $f(n)$ is the running time of M, we say that M runs in time $f(n)$ and that M is an $f(n)$ time Turing machine. Let $t : \mathbf{N} \to \mathbf{N}$ be a function. The time complexity class TIME($t(n)$) is defined as TIME($t(n)$)=$\{L \mid L$ is a language decided by an $O(t(n))$ time Turing machine$\}$.

In time complexity theory polynomial differences in running time are considered to be small, whereas exponential differences are considered to be large. Polynomial time algorithms are fast enough for many purposes, but exponential time algorithms rarely are useful. All reasonable deterministic computational models are polynomially equivalent, that is, any of them can simulate another with only a polynomial increase in running time.

NP is the class of languages that are decidable in polynomial time on a nondeterministic Turing machine. In other words, **NP**=$\bigcup_k$ NTIME(n^k). Inside **NP** there are problems such that if a polynomial time algorithm exists for any of these problems, then all problems in **NP** would be solvable in polynomial time. These problems are called **NP**-complete.

A well known **NP**-complete problem is satisfiability problem, `SAT`. It asks whether or not for a given propositional formula in the conjunctive normal form there is a truth-assignment of variables such that the formula assumes the value *true*.

A propositional formula C is in the conjunctive normal form if it is a conjunction of disjunctions, $C = C_1 \wedge C_2 \wedge \ldots \wedge C_m$, where each $C_i, 1 \leq i \leq m$, is a *clause* of the form $C_i = y_{i,1} \vee y_{i,2} \vee \ldots \vee y_{i,p_i}$ with each literal y_j being either a propositional variable, x_s, or its negation, $\neg x_s$.

2.1 Spiking Neural P Systems

We recall the basic definition of SN P systems as shown in [4].

A *spiking neural P system* of degree $m \geq 1$ is a construct of the form $\Pi = (O, \sigma_1, \ldots, \sigma_m, syn, i_0)$, where:

1. $O = \{a\}$ is the singleton alphabet (a is called *spike*);
2. $\sigma_1, \ldots, \sigma_m$ 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* present in the neuron;
   b) R_i is a finite set of *rules* of the following two forms:
      (1) $E/a^r \to a; d$, where E is a regular expression over O, $r \geq 1$, and $d \geq 0$;
      (2) $a^s \to \lambda$, for some $s \geq 1$, with the restriction that $a^s \notin L(E)$ for any rule $E/a^r \to a; d$ of type (1) from R_i;
3. $syn \subseteq \{1, 2, \ldots, m\} \times \{1, 2, \ldots, m\}$ with $(i, i) \notin syn$ for $1 \leq i \leq m$ (*synapses* among cells);
4. $i_0 \in \{1, 2, \ldots, m\}$ indicates the *output neuron*.

The rules of type (1) are *firing* (we also say *spiking*) *rules*, and they are applied as follows. If the neuron σ_i contains k spikes, and $a^k \in L(E), k \geq c$, then the rule $E/a^c \to a; d$ can be applied. The application of this rule means consuming

(removing) c spikes (thus only $k - c$ remain in σ_i), the neuron is fired, and it produces a spike after d time units (a global clock is assumed, marking the time for the whole system, hence the functioning of the system is synchronized).

If $d = 0$, then the spike is emitted immediately, if $d = 1$, then the spike is emitted in the next step, etc. In the steps before emitting the spike, the neuron is *closed*, it cannot use any rule and cannot receive spikes from other neurons. The delay in sending the spike along the synapses models the refractory period of the neuron from neurobiology – more details on its formalization can be found in [4].

A spike emitted by a neuron σ_i (is replicated and a copy of it) goes to each neuron σ_j such that $(i, j) \in syn$.

The rules of type (2) are *forgetting* rules and they are applied as follows: if the neuron σ_i contains exactly s spikes, then the rule $a^s \to \lambda$ from R_i can be used, meaning that all s spikes are removed from σ_i.

If a rule $E/a^c \to a; d$ of type (1) has $E = a^c$, then we will write it in the simplified form $a^c \to a; d$.

In each time unit, if a neuron σ_i can use one of its rules, then a rule from R_i *must* be used. Since two firing rules, $E_1/a^{c_1} \to a; d_1$ and $E_2/a^{c_2} \to a; d_2$, can have $L(E_1) \cap L(E_2) \neq \emptyset$, it is possible that two or more rules can be applied in a neuron, and in that case, only one of them is chosen non-deterministically. By definition, if a firing rule is applicable, then no forgetting rule is applicable, and vice versa.

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 number of spikes present in each neuron and by the state of the neuron, more precisely, by the number of steps from now on until it becomes open (this number is zero if the neuron is already open). Thus, $\langle r_1/t_1, \ldots, r_m/t_m \rangle$ is the configuration where neuron $i = 1, 2, \ldots, m$ contains $r_i \geq 0$ spikes and it will be open after $t_i \geq 0$ steps; with this notation, the initial configuration is $C_0 = \langle n_1/0, \ldots, n_m/0 \rangle$.

Using the rules as described above, one can define transitions among configurations. A transition between two configurations C_1, C_2 is denoted by $C_1 \Longrightarrow C_2$. Any sequence of transitions starting in the initial configuration is called a *computation*. A computation halts if it reaches a configuration where all neurons are open and no rule can be used.

In the spirit of spiking neurons, see, e.g., [5], as the result of a computation, in [4] and [8] one considers the distance between two consecutive spikes which exit a distinguished neuron of the system. Then, in [1] one considers as the result of a computation the so-called spike train of the computation, the sequence of symbols 0 and 1 obtained by associating 1 with a step when a spike exits the system and 0 otherwise. Languages over the binary alphabet are computed in this way.

3 Spiking Neural P Systems with Self-Activation

In the following we define P systems with self-activation. As already discussed in Section 1 we consider a neuron to be *inactive* if it contains no spike (hence no

rule can be applied in it). As soon as a spike enters a neuron (as input in the initial configuration or from another neuron through a synapse), it makes it *active* altogether with the synapses that it establishes with other neurons.

In a self-activating SN P system we have an arbitrarily large number of neurons which differ by the number of spikes and/or of rules they contain. Some of these neurons will be active, the others will be inactive.

In what follows, we construct an SN P system for solving `SAT` problem in constant time. Let us consider n variables $x_1, x_2, \ldots, x_n, n \geq 1$, and a propositional formula with m clauses, $\gamma = C_1 \wedge \cdots \wedge C_m$, such that each clause $C_i, 1 \leq i \leq m$, is of the form $C_i = y_{i,1} \vee \cdots \vee y_{i,k_i}, k_i \geq 1$, where $y_{i,j} \in \{x_k, \neg x_k \mid 1 \leq k \leq n\}$.

The set of all instances of `SAT` with n variables and m clauses is denoted by $SAT(\langle n, m\rangle)$.

The instance γ is encoded as a set over

$$X = \{x_{i,j}, x'_{i,j} \mid 1 \leq i \leq m, 1 \leq j \leq n\},$$

where $x_{i,j}$ represents variable x_j appearing in clause C_i without negation, while $x'_{i,j}$ represents variable x_j appearing in clause C_i with negation.

Now, we give an informal description of the precomputed resource structure devoted to $SAT(\langle n, m\rangle)$. We look at Figure 1 which depicts an SN P system working in self-activating manner using precomputed resources, where the nodes and the arrows represent the neurons and the synapses, respectively. One can notice that the nodes have (four) different shapes (◯, ⊚, □, ▷), but this is just a way to make the construction easier to understand (the shape does not imply any differences in the behavior of the nodes). Also, we see that the structure has a sort of symmetry. Namely, for each clause we have a block of ◯-neurons and ⊚-neurons.

The Device Structure The precomputed device (initially inactive) able to deal with any $\gamma \in SAT(\langle n, m\rangle)$ is formed by $2^n(m+1) + 2nm + 1$ neurons and $2^n(3m+1)$ synapses. Further on we are giving some details on the components of this structure.

Neurons of type $\bigcirc_{c_i x_j 1/0}$: For each variable x_j of a clause C_i, we associate 2 neurons $\bigcirc_{c_i x_j 1}$ and $\bigcirc_{c_i x_j 0}$, $1 \leq i \leq m, 1 \leq j \leq n$. Obviously, the subscript of the neurons indicates the clause (c_i for the clause C_i), and the variable (x_j for the jth variable in the clause). 1 and 0 are used to mark differently the two neurons needed to encode the same variable; their use will be detailed further on in the paper where the encoding part will be explained. However, each clause is described by $2n$ neurons, and there are exactly $2nm$ neurons of type $\bigcirc_{c_i x_j 1/0}$ associated with m clauses.

Neurons of type $\circledcirc_{c_i bin}$: There are 2^n $\circledcirc_{c_i bin}$ neurons, associated to each clause C_i, injectively labeled with elements of $\{c_i bin \mid bin \in \{1,0\}^n\}$. They correspond to the 2^n truth-assignments for variables variables $x_1, \ldots, x_n$. In total, the device has $2^n m$ $\circledcirc_{c_i bin}$ neurons (2^n for each of the m clauses). Later on, we will see that these

neurons will handle, during the computation, boolean operation $\vee$ (OR) present in the clauses of the formula.

Synapses $\bigcirc_{c_i x_j 1/0} \longrightarrow \circledcirc_{c_i bin}$: The connections are in one direction from $\bigcirc_{c_i x_j 1/0}$ to $\circledcirc_{c_i bin}$. The synapses are designed in such a way that the two neurons linked by a connection have the same prefix of the labels (c_i), and the last symbol of label $c_i x_j 0/1$ is the same with the jth symbol (0 or 1) of the string *bin*. Each $\circledcirc_{c_i bin}$ neuron is connected to n $\bigcirc_{c_i x_j 1/0}$ neurons.

Neurons of type $\square_{bin}$: There are exactly 2^n $\square_{bin}$ neurons in the device labeled injectively with strings from $bin \in \{0,1\}^n$. These neurons are designed to handle boolean operation $\wedge$ (AND) between the clauses of the formula.

Synapses $\circledcirc_{c_i bin} \longrightarrow \square_{bin}$: Each $\square_{bin}$ neuron is connected to one $\circledcirc_{c_i bin}$ neuron from each clause block, hence, m double circled neurons may send spikes to each square neuron. Strings *bin* from the labels of the connected $\circledcirc_{c_i bin}$ and $\square_{bin}$ neurons are the same.

Neuron of type $\triangleright_4$: Finally, there is a unique output neuron $\triangleright$ with label 4. By choosing label 4 for this neuron we emphasize that it spikes in the 4th step of the computation if the problem has at least a solution and does not spike in this step, otherwise. All $\square_{bin}$ neurons are connected to the output neuron, hence, there are 2^n connections of type $\square_{bin} \longrightarrow \triangleright_4$.

Rules: Here are the rules which apply to each type of neurons:
$R_\circ = \{a \to \lambda,\ a^2 \to a; 0,\ a^3 \to \lambda,\ a^4 \to a; 0\}$,
$R_\circledcirc = \{a^+/a \to a; 0\}$,
$R_\square = \{a^m \to a; 0\}$,
$R_\triangleright = \{a^+/a \to a; 0\}$.

We have described the precomputed device structure for solving `SAT` problem as depicted in Figure 1, with the neurons in the inactive mode. Note that this structure is independent of the instance of SAT we want to solve, and it depends only on n and m. Let us explain now how we encode the particular instance of the problem into the device.

The Problem Encoding The variables are encoded by spikes as follows: one assigns values 1 and 0 to each variable x_j and $\neg x_j$. Further, a variable x_j is encoded by two spikes (a^2), and one spike (a) if we assign to x_j values 1 and 0, respectively. Similarly, we use a^3 and a^4 to encode variable $\neg x_j$, which has assigned values 1 and 0, respectively.

	1	0
$x_{i,j}$	$(a^2, \bigcirc_{c_i x_j 1})$	$(a, \bigcirc_{c_i x_j 0})$
$\neg x_{i,j}$	$(a^3, \bigcirc_{c_i x_j 1})$	$(a^4, \bigcirc_{c_i x_j 0})$

Table 1. The variable encoding.

In Table 1 one can notice how we introduce the encoded variables (the spikes) into the precomputed device. The (encoded) variables x_j or $\neg x_j$, from a clause C_i,

assigned with value 1 are introduced in the neuron with label $c_i x_j 1$ ($\bigcirc_{c_i x_j 1}$), while the other "half" of the encoding, the one where variable is assigned with value 0 is introduced in the neuron labeled $c_i x_j 0$, (hence $\bigcirc_{c_i x_j 0}$). Some of the $\bigcirc_{c_i x_j 1/0}$ neurons will not be activated in the case the corresponding variables are missing form the given instance of the problem. Anyway, we stress the fact that at most $2nm$ $\bigcirc_{c_i x_j 1/0}$ neurons are activated in when the device is initialized, the other neurons in the system remaining inactive.

The Computation Starting this moment (when the device is initialized and the corresponding neurons activated), the computation can be performed and the problems will be solved in 4 steps. Once the system starts to evolve the spikes follow the one-directional path:

$$\bigcirc\text{-neurons} \longrightarrow \circledcirc\text{-neurons} \longrightarrow \square\text{-neurons} \longrightarrow \triangleright\text{-neuron},$$

as shown in Figure 2.

3.1 An Example

In order to illustrate the procedure discussed above, let us examine a simple example.

We consider the following instance of SAT, with two variables and three clauses, $\gamma \in SAT(\langle 2, 3\rangle)$,

$$\gamma = (x_1) \wedge (\neg x_1 \vee x_2) \wedge (\neg x_2).$$

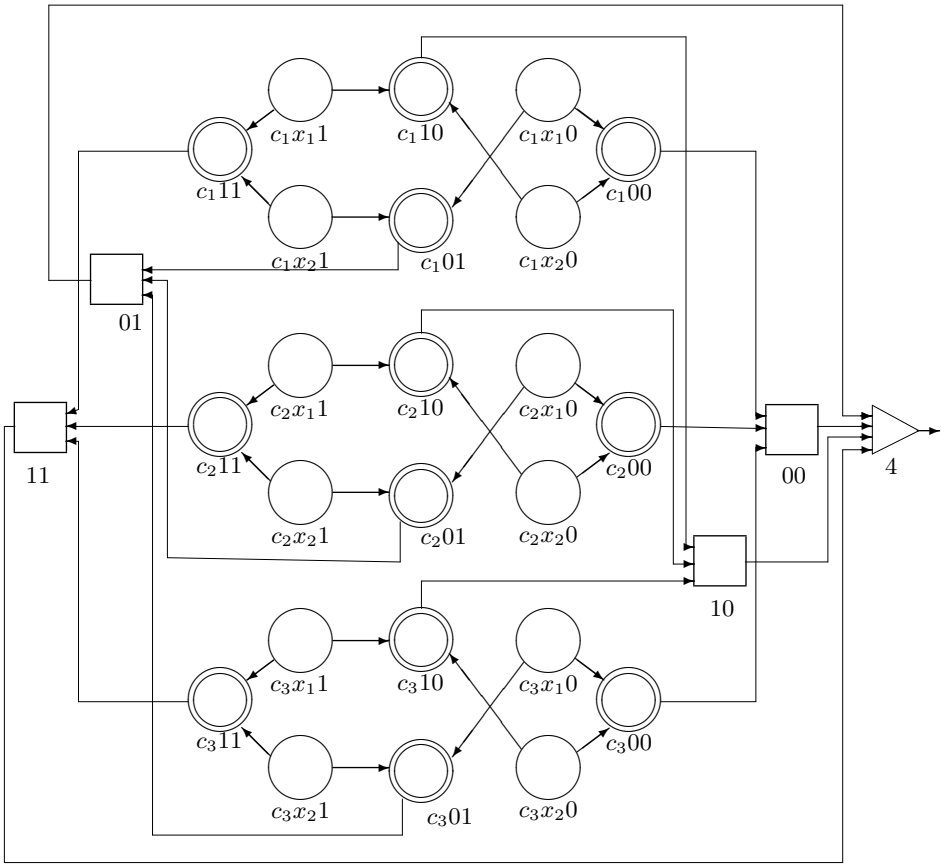
The device structure The system we construct is given in a pictorial way in Figure 1, and has 29 inactive neurons ($4 * 2^2 + 2 * 2 * 3 + 1$) and 40 synapses. There are 3 blocks of neurons, each dealing with a clause of the problem. Each block contains 4 solid circled ($\bigcirc_{c_i x_1 1}$, $\bigcirc_{c_i x_1 0}$, $\bigcirc_{c_i x_2 1}$, $\bigcirc_{c_i x_2 0}$) neurons – 2 for each variable – to which we assign 1 and 0 (see the labels). In each block, there are also 4 double circled neurons ($\circledcirc_{c_i 11}, \circledcirc_{c_i 10}, \circledcirc_{c_i 01}, \circledcirc_{c_i 00}$) connected to the $\bigcirc_{c_i x_j 1/0}$ neurons, corresponding to the 2^2 truth-assignments (see the labels). Moreover, we have 4 $\square_{bin}$ neurons connected to the corresponding $\circledcirc_{c_i bin}$ neurons.

Encoding According to the formula γ, in the first clause C_1, there is only one variable x_1. We assign values 1 and 0 to x_1 and encode it by two spikes (a^2) and one spike (a^1) that are placed in $\bigcirc_{c_1 x_1 1}$ and $\bigcirc_{c_1 x_1 0}$, respectively. The other two neurons $\bigcirc_{c_1 x_2 1}$ and $\bigcirc_{c_1 x_2 0}$ corresponding to variable x_2 remain empty, since there is no variable x_2 or $\neg x_2$ in the clause.

The second clause (C_2) is encoded in the following clause block. To each variable $\neg x_1$ and x_2 are assigned values 1 and 0. Further, they are encoded by a^3 in $\bigcirc_{c_2 x_1 1}$, a^4 in $\bigcirc_{c_2 x_1 0}$, for $\neg x_1$, and a^2 in $\bigcirc_{c_2 x_2 1}$, a^1 in $\bigcirc_{c_2 x_2 0}$, for x_2.

The last clause C_3 has only one variable, $\neg x_2$, which is encoded in the neurons $\bigcirc_{c_3 x_2 1}$ and $\bigcirc_{c_3 x_2 0}$. The other two neurons corresponding to this clause remain empty. See Step 1 of Figure 2.

Computation Once the encoding is done, single circled neurons are activated and send signals, or erase spikes according to the rules they contain. In the first step, neurons having inside a^2 and a^4 fire, while spikes a^1 and a^3 from the other



Rules used in $\Pi_{SAT(\langle 2,3\rangle)}$: $R_\circ = \{a \to \lambda,\ a^2 \to a; 0,\ a^3 \to \lambda,\ a^4 \to a; 0\}$;
$R_{\circledcirc} = \{a^+/a \to a; 0\}$; $R_\square = \{a^3 \to a; 0\}$; $R_\triangleright = \{a^+/a \to a; 0\}$.

Fig. 1. Precomputed spiking neural net for $\texttt{SAT}(\langle 2, 3\rangle)$

neurons are deleted. We see in Step 2 from Figure 2, that only 7 double circled neurons out of 12 contain spikes, hence only 7 will be activated in the second step. Then, in next step of computation, double circled neurons containing spikes fire because of the rules $a^+/a \to a$ inside. One can notice that neurons $\square_{11}$, $\square_{00}$, and $\square_{10}$ have received two spikes, and neuron $\square_{01}$ has received only one spike. In the third step of the computation, only neuron $\circledcirc_{c_201}$ fires since there was one spike remained, and nothing else happens in the system. The rule present inside the square neurons, $a^3/a \to a$, cannot be applied, because there is no such neuron containing three spikes. At the fourth step, the output neuron does not spike, since

no spike has arrived here in the third step of the computation, hence the given problem has no solution.

We have mentioned before that what happens after the fourth step of the computation is out of our interest because it does not give further details with respect to the satisfiability of the given problem. We just want to mention that the system may not stop after giving the answer to our problem.

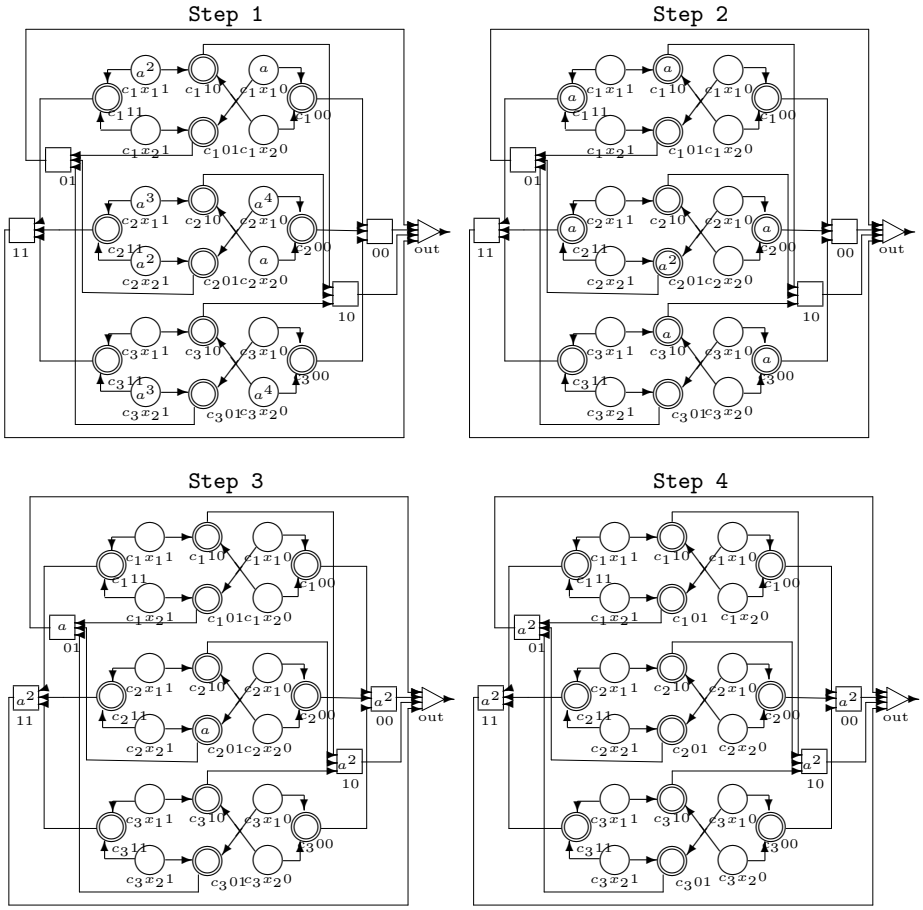


Fig. 2. The steps of the computation for $\gamma \in SAT(\langle 2, 3\rangle)$.

3.2 SN P Systems Solving SAT

Formally, for given $(n, m) \in \mathbf{N}^2$, an SN P system using precomputed resources, working in a self-activating manner, devoted to solve SAT problem with n variables

and m clauses, is a construct

$$\Pi^{n,m}_{\texttt{SAT}} = (\Pi_{SAT(\langle n,m\rangle)}, \Sigma(\langle n,m\rangle))$$

with:

- $\Pi_{SAT(\langle n,m\rangle)} = (O, \mu, \triangleright_4)$, where:
  1. $O = \{a\}$ is the singleton alphabet;
  2. $\mu = (H, \Omega, R_H, syn)$ is the precomputed device structure, where:
     - $H = H_1 \cup H_2 \cup H_3 \cup H_4$ is a finite set of neuron labels, where
       $H_1 = \{c_i x_j 1, c_i x_j 0 \mid 1 \le i \le m, 1 \le j \le n\}$,
       $H_2 = \{c_i bin \mid 1 \le i \le m, bin \in \{0,1\}^n\}$,
       $H_3 = \{bin \mid bin \in \{0,1\}^n\}$,
       $H_4 = \{4\}$;
     - $\Omega = \{(0, \bigcirc_{h_1}), (0, \circledcirc_{h_2}), (0, \square_{h_3}), (0, \triangleright_{h_4}) \mid h_1 \in H_1, h_2 \in H_2, h_3 \in H_3, h_4 \in H_4\}$ are the empty (inactive) neurons present in the precomputed structure (with $|\Omega| = 2^n(m+1) + 2nm + 1$);
     - $R_H = R_{H_1} \cup R_{H_2} \cup R_{H_3} \cup R_{H_4}$ is a finite set of rules associated to the neurons, where
       R_{H_1}=$\{a^1 \to \lambda,\ a^2 \to a; 0,\ a^3 \to \lambda,\ a^4 \to a; 0\}$,
       $R_{H_2} = R_{H_4} = \{a^+/a \to a; 0\}$,
       $R_{H_3} = \{a^m \to a; 0\}$;
     - $syn = \bigcup_{i=1}^{m} \{(\bigcirc_{c_i x_j 1}, \circledcirc_{c_i bin}) \mid bin|_j = 1, 1 \le j \le n, bin \in \{0,1\}^n\}$
       $\cup \bigcup_{i=1}^{m} \{(\bigcirc_{c_i x_j 0}, \circledcirc_{c_i bin}) \mid bin|_j = 1, 1 \le j \le n, bin \in \{0,1\}^n\}$
       $\cup \{(\circledcirc_{c_i bin}, \square_{bin}) \mid bin \in \{0,1\}^n, 1 \le i \le m\} \cup \{\square_{bin}, \triangleright_4 \mid bin \in \{0,1\}^n\}$;
  3. $\triangleright_4$ is the output neuron;
- $\Sigma(\langle n,m\rangle)$ is a polynomial encoding from an instance γ of SAT into $\Pi_{SAT(\langle n,m\rangle)}$, providing the initialization of the system such that

$$\begin{aligned}\Sigma(\langle n,m\rangle)(\gamma) = {} & \{(1, \bigcirc_{c_i x_j 0}) \mid x_{i,j} \in \gamma, 1 \le i \le m, 1 \le j \le n\} \\ & \cup \{(2, \bigcirc_{c_i x_j 1}) \mid x_{i,j} \in \gamma, 1 \le i \le m, 1 \le j \le n\} \\ & \cup \{(3, \bigcirc_{c_i x_j 0}) \mid x'_{i,j} \in \gamma, 1 \le i \le m, 1 \le j \le n\} \\ & \cup \{(4, \bigcirc_{c_i x_j 1}) \mid x'_{i,j} \in \gamma, 1 \le i \le m, 1 \le j \le n\}.\end{aligned}$$

In the precomputed structure of $\Pi(\langle n,m\rangle)$ (for any SAT problem with n variables and m clauses), neurons are described as a pair *(spikes inside, neuron)*. There are $2^n(m+1)+2nm+1$ neurons and $2^n(3m+1)$ synapses initially inactive. $\Sigma(\langle n,m\rangle)$ encodes the given instance of the problem in spikes, that is, the encoded problem is introduced into the precomputed structure (spikes are assigned to solid circled neurons) activating the corresponding neurons. It is important to note that at most $2nm$ neurons will be activated, hence the initialization takes a polynomial time.

The system evolves exactly in the same manner as the basic SN P systems. The result of the computation is obtained in its 4th step. If the system (output neuron)

spikes, then the given problem has at least a solution. Otherwise, it does not have any solution. The evolution of the system after this step of the computation is ignored.

Thus, the system evolves as follows:

Step 1: After encoding the problem (the variables) in (the number of) spikes we introduce into the system, namely in at most $2nm$ neurons of type $\bigcirc_{c_i 0 x_j 1/0}$, these neurons are activated and evolve according to the rules inside.

Step 2: Neurons of type $\circledcirc_{c_i bin}$ which provide information on the truth-assignments at the level of clauses (OR operations are simulated) – at most $2^n m$ initially inactive neurons – will be activated, and they will send spikes to the 2^n $\square_{bin}$ neurons corresponding to the truth-assignments at the level of the system.

Step 3: Now, only those neurons of type $\square_{bin}$ in which the threshold is reached (AND operations are simulated, if all m clauses are satisfied, hence there exist m spikes inside) will spike (because of the rule $a^m \to a; 0$) toward the output neuron.

Step 4: If in the output neuron will spike, it means that our problem has at least a solution. Otherwise, we do not have any solution for the given problem.

As already mentioned both in the definition of the system and in the example, it is not mandatory for the system to halt. We only observe its behavior in the fourth step of the computation.

Based on the previous explanations we can state that:

Theorem 1. *$\Pi^{n,m}_{\texttt{SAT}}$ can deterministically solve each instance of size (n, m) of* `SAT` *in constant time.*

4 Conclusions and Remarks

In this paper, we have shown that SN P systems are not only computationally universal, but also computationally efficient devices. We show that the idea of using an already existing, but inactive, workspace proves to be very efficient in solving **NP**-complete problems. We illustrate this possibility with the satisfiability problem. The initial system is fixed, depending only on the number of variables (n) and the number of clauses (m). Then any instance of `SAT` with size nxm is encoded in a polynomial time in spikes introduced in the system and then it is solved in exactly 4 steps by our device.

If we use rules of type $R_\circ = R_\triangleright = \{a^+ \to a; 0\}$ (and we interpret such a rule in the sense that if at least one spike is present in a moment inside a neuron, then it should spike immediately and all spikes are consumed) in the $(\Pi_{SAT(\langle n,m\rangle)}, \Sigma(\langle n, m\rangle))$, then the computation *halts* in the fourth step with the

system spiking if the problem has at least a solution. Otherwise the problem does not have any solution. After the fourth step, no spike will remain in the system.

Another way to stop the computation after sending the answer out is the following. Let us introduce two intermediate neurons in between each $\square_{bin}$ and $\triangleright_4$ neuron, so that in step 4 (the intermediate neurons take one step for transmitting the spikes further) neuron $\triangle_4$ receives an even number of spikes. Then, instead of the rule $a^+/a \to a; 0$, in neuron $\triangleright_4$ we use the rule $(a^2)^+/a \to a; 0$. Thus, if any spike reaches neuron $\triangleright_4$, then an even number of spikes arrive here; thee rule $(a^2)^+/a \to a; 0$ can be used only once, in step 5, because after that the number of remaining spikes is odd.

In this way, we add $2 \cdot 2^n$ neurons and further $3 \cdot 2^n$ synapses, while the computation lasts five steps only.

Moreover, if we consider `3SAT` problem, then each clause block will contain exactly $14 = 3 * 2 + 2^3$ neurons.

One possible line of research would be to try to investigate other computationally hard problems with the SN P systems using precomputed resources. Finally, investigating other computational complexity issues within this framework would be also very challenging.

References

1. H. Chen, R. Freund, M. Ionescu, Gh. Păun, M.J. Pérez-Jiménez: On string languages generated by spiking neural P systems. In the present volume.
2. H. Chen, T.-O. Ishdorj, Gh. Păun: Computing along the axon. In the present volume.
3. E. Czeizler: Self-activating P systems. In *Pre-Proceedings of Second Workshop on Membrane Computing*, Curtea de Argeş, Romania, August 2002, 193–206, and in *Membrane Computing. International Workshop, WMC-CdeA 2002, Curtea de Argeş, Romania, August 2002* (Gh. Păun, G. Rozenberg, A. Salomaa, C. Zandron, eds.), LNCS 2597, Springer, 2003, 234–246.
4. M. Ionescu, Gh. Păun, T. Yokomori: Spiking neural P systems. *Fundamenta Informaticae*, 71, 2-3 (2006), 279–308.
5. W. Maass: Computing with spikes. *Special Issue on Foundations of Information Processing of TELEMATIK*, 8, 1 (2002), 32–36.
6. Gh. Păun: Computing with membranes. *Journal of Computer and System Sciences*, 61, 1 (2000), 108–143, and *Turku Center for Computer Science-TUCS Report* No 208, 1998 (www.tucs.fi).
7. Gh. Păun: *Membrane Computing. An Introduction*, Springer, Berlin, 2002.
8. Gh. Păun, M.J. Pérez-Jiménez, G. Rozenberg: Spike trains in spiking neural P systems. *Intern. J. Found. Computer Sci.*, to appear (also available at [12]).
9. Gh. Păun, M.J. Pérez-Jiménez, G. Rozenberg: Infinite spike trains in spiking neural P systems. Submitted, 2006.
10. A. Salomaa: *Formal Languages*. Academic Press, New York, 1973.
11. M. Sipser: *Introduction to the Theory of Computation*. PWS Publishing Company, International Thomson Publishing Company, 1997.
12. The P Systems Web Page: `http://psystems.disco.unimib.it`.