Acta Informatica (2010) 47:133–146
DOI 10.1007/s00236-009-0113-8

ORIGINAL ARTICLE

Small universal accepting hybrid networks of evolutionary processors

Remco Loos · Florin Manea · Victor Mitrana

Received: 30 March 2009 / Accepted: 18 November 2009 / Published online: 18 December 2009

Abstract In this paper, we improve some results regarding the size complexity of accepting hybrid networks of evolutionary processors (AHNEPs). We show that there are universal AHNEPs of size 6, by devising a method for simulating 2-tag systems. This result improves the best upper bound for the size of universal AHNEPs which was 7. We also propose a computationally and descriptionally efficient simulation of nondeterministic Turing machines with AHNEPs. More precisely, we prove that AHNEPs with ten nodes can simulate any nondeterministic Turing machine of time complexity $f(n)$ in time $O(f(n))$. This result significantly improves the best known upper bound for the number of nodes in a network simulating in linear time an arbitrary Turing machine, namely 24.

Work is supported by the research grant ES-2006-0146 of the Spanish Ministry of Science and Innovation and the Romanian Ministry of Education and Research (PN-II Program, Projects *GlobalComp*, *SEFIN* and *SELF*). The work of Florin Manea is also supported by the *Alexander von Humboldt Foundation*.

R. Loos
EMBL, European Bioinformatics Institute, Wellcome Trust Genome Campus,
Hinxton, Cambridge CB10 1SD, UK
e-mail: remco.loos@ebi.ac.uk

F. Manea · V. Mitrana (✉)
Faculty of Mathematics and Computer Science, University of Bucharest,
Academiei 14, 010014 Bucharest, Romania
e-mail: mitrana@fmi.unibuc.ro

F. Manea
Faculty of Computer Science, Otto-von-Guericke University of Magdeburg,
PSF 4120, 39016 Magdeburg, Germany
e-mail: flmanea@gmail.com

V. Mitrana
Department of Information Systems and Computation, Technical University of Valencia,
Camino de Vera s/n, 46022 Valencia, Spain

1 Introduction

The computational model considered in this paper was introduced in [7]. It is a bio-inspired model based on an architecture considered in [4]. An AHNEP consists in a graph such that a very simple processor, a so-called evolutionary processor, is placed in every node of the graph. By an evolutionary processor we mean a processor which is able to perform very simple operations, namely point mutations in a DNA sequence (insertion, deletion or substitution of a pair of nucleotides). More generally, each node may be viewed as a cell having genetic information encoded in DNA sequences which may evolve by local evolutionary events, that is point mutations. Each node is specialized just for one of these evolutionary operations. Furthermore, the data in each node is organized in the form of multisets of strings (each string appears in an arbitrarily large number of copies), and all copies are processed in parallel such that all the possible events that can take place do actually take place. The reader interested in a more detailed discussion about the model is referred to [5,7].

Here, we focus on the descriptional complexity of AHNEPs. We aim to improve some results reported in [1,2,5,6]. More precisely, we are looking for a small size universal AHNEP, where the size of an AHNEP is defined as the number of nodes of the network. In [5] it was shown that ten nodes are sufficient to obtain a universal AHNEPs. In a recent contribution, Alhazov et al. [2] obtained a sharper bound of 7. We further improve this bound to 6, via a simulation of 2-tag systems introduced in [10]. Note that in [2] it is also shown that AHNEPs of two nodes are not universal, suggesting this bound is approaching the absolute upper bound.

Also, we are interested in finding a way to design concise AHNEPs simulating computationally efficient nondeterministic Turing machines. From the simulation of a tag system we can obtain a concise universal AHNEP, but the above problem is not necessarily solved. Indeed, a 2-tag system can efficiently simulate any deterministic Turing machine but not nondeterministic ones. To this aim, we propose a simulation of nondeterministic Turing machines with AHNEPs of size 10 which maintains the working time of the Turing machine. That is, every language accepted by a one-tape nondeterministic Turing machine in time $f(n)$ can be accepted by an AHNEP of size 10 in time $O(f(n))$. This result considerably improves the best known bound, of 24, reported in [6] for the size of AHNEPs simulating nondeterministic one-tape Turing machines in linear time. Though smaller AHNEPs that can be used to simulate Turing machines are known, like those in [1,2] of size 10 and 7, respectively, these simulations need quadratic time, and are thus less efficient from a computational point of view.

2 Basic definitions

We start by summarizing the notions used throughout the paper; for all unexplained notions the reader is referred to [12]. An *alphabet* is a finite and nonempty set of symbols. The cardinality of a finite set A is written $card(A)$. Any sequence of symbols from an alphabet V is called $word(string)$ over V. The set of all words over V is denoted by V^* and the empty word is denoted by ε. The length of a word x is denoted by $|x|$ while $alph(x)$ denotes the minimal alphabet W such that $x \in W^*$. For a word $x \in W^*$, x^r denotes the reversal of the word.

We consider here the following definition of 2-tag systems that appears in [11]. This type of tag-system, namely the type $\mathcal{T}_2$ 2-tag-systems that appear in Sect. 8 of [11], is slightly different but equivalent to those from [8,10]. A 2-tag system $T = (V, \phi)$ consists of a finite alphabet of symbols V, containing a special *halting symbol* H (denoted in [11] with *STOP*)

and a finite set of rules $\phi : V\backslash\{H\} \to V^+$ such that $|\phi(x)| \geq 2$ or $\phi(x) = H$. Furthermore, $\phi(x) = H$ for just one $x \in V\backslash\{H\}$. A halting word for the system T is a word that contains the halting symbol H or whose length is <2. The transformation t_T (called the tag operation) is defined on the set of non-halting words as follows: if x is the leftmost symbol of a non-halting word w, then $t_T(w)$ is the result of deleting the leftmost 2 symbols of w and then appending the word $\phi(x)$ at the right end of the obtained word. A computation by a 2-tag system as above is a finite sequence of words produced by iterating the transformation t, starting with an initially given non-halting word w and halting when a halting word is produced. A computation is not considered to exist unless a halting word is produced in finitely-many iterations. Note that in [11] the halting words are defined a little bit different, as the words starting with the only symbol y such that $\phi(y) = H$, or the words whose length is <2. However, our way of defining halting words is equivalent to that in [11], in the sense that there exists a bijection between the valid computations obtained in each of these two cases. Indeed, if we consider the stopping condition from [11], and obtain in a valid computation a word starting with y, thus a halting word, it is enough to apply once more t_T on this word to obtain a word containing H, a halting word according to our definition, and transform the initial valid computation in a valid computation according to our definition. Conversely, if a word containing H, a halting word for our definition, is obtained in a valid computation, then the halting symbol could not have appeared in that word in other way than by applying t_T on a word starting with y, a halting word for the definition from [11], therefore, we have a corresponding valid computation, by that definition. As shown in [11], such restricted 2-tag systems are universal.

A nondeterministic Turing machine is a construct $M = (Q, V, U, \delta, q_0, B, F)$, where Q is a finite set of states, V is the input alphabet, U is the tape alphabet, $V \subset U$, q_0 is the initial state, $B \in U\backslash V$ is the "blank" symbol, $F \subseteq Q$ is the set of final states, and δ is the transition mapping, $\delta : (Q\backslash F) \times U \to 2^{Q\times(U\backslash\{B\})\times\{R,L\}}$. In this paper, we assume without loss of generality that any Turing machine we consider has a semi-infinite tape (bounded to the left) and makes no stationary moves; the computation of such a machine is described in [3,9,12]. An input word is accepted if and only if after a finite number of moves the Turing machine enters a final state. The language accepted by the Turing machine is a set of all accepted words. We say a Turing machine *decides* a language L if it accepts L and moreover halts on every input. The reader is referred to [3,9] for the classical time and space complexity classes defined for Turing machines.

We say that a rule $a \to b$, with $a, b \in V \cup \{\varepsilon\}$ is a *substitution rule* if both a and b are not ε; it is a *deletion rule* if $a \neq \varepsilon$ and $b = \varepsilon$; it is an *insertion rule* if $a = \varepsilon$ and $b \neq \varepsilon$. The set of all substitution, deletion, and insertion rules over an alphabet V are denoted by Sub_V, Del_V, and Ins_V, respectively.

Given a rule σ as above and a word $w \in V^*$, we define the following *actions* of σ on w:

- If $\sigma \equiv a \to b \in Sub_V$, then $\sigma^*(w) = \begin{cases} \{ubv : \exists u, v \in V^* (w = uav)\}, \\ \{w\}, \text{ otherwise} \end{cases}$
- If $\sigma \equiv a \to \varepsilon \in Del_V$, then $\sigma^*(w) = \begin{cases} \{uv : \exists u, v \in V^* (w = uav)\}, \\ \{w\}, \text{ otherwise} \end{cases}$

$$\sigma^r(w) = \begin{cases} \{u : w = ua\}, \\ \{w\}, \text{ otherwise} \end{cases} \qquad \sigma^l(w) = \begin{cases} \{v : w = av\}, \\ \{w\}, \text{ otherwise} \end{cases}$$

- If $\sigma \equiv \varepsilon \to a \in Ins_V$, then

$$\sigma^*(w) = \{uav : \exists u, v \in V^* (w = uv)\}, \quad \sigma^r(w) = \{wa\}, \quad \sigma^l(w) = \{aw\}.$$

$\alpha \in \{*, l, r\}$ expresses the way of applying a deletion or insertion rule to a word, namely at any position ($\alpha = *$), in the left ($\alpha = l$), or in the right ($\alpha = r$) end of the word, respectively. For every rule σ, action $\alpha \in \{*, l, r\}$, and $L \subseteq V^*$, we define the α-action of σ on L by $\sigma^\alpha(L) = \bigcup_{w \in L} \sigma^\alpha(w)$. Given a finite set of rules M, we define the α-action of M on the word w and the language L by:

$$M^\alpha(w) = \bigcup_{\sigma \in M} \sigma^\alpha(w) \quad \text{and} \quad M^\alpha(L) = \bigcup_{w \in L} M^\alpha(w),$$

respectively. In what follows, we shall refer to the rewriting operations defined above as *evolutionary operations* since they may be viewed as linguistic formulations of local DNA mutations.

For two disjoint subsets P and F of an alphabet V and a word w over V, we define the predicates:

$$\varphi^{(s)}(w; P, F) \equiv P \subseteq alph(w) \wedge F \cap alph(w) = \emptyset$$
$$\varphi^{(w)}(w; P, F) \equiv alph(w) \cap P \neq \emptyset \wedge F \cap alph(w) = \emptyset.$$

The construction of these predicates is based on *random-context conditions* defined by the two sets P (*permitting contexts/symbols*) and F (*forbidding contexts/symbols*). Informally, the first condition requires that all permitting symbols are present in w and no forbidding symbol is present in w, while the second one is a weaker variant of the first, requiring that at least on permitting symbol appears in w and no forbidding symbol is present in w. For every language $L \subseteq V^*$ and $\beta \in \{(s), (w)\}$, we define:

$$\varphi^\beta(L, P, F) = \{w \in L \mid \varphi^\beta(w; P, F)\}.$$

An *evolutionary processor over* V is a tuple (M, PI, FI, PO, FO), where:

- M is a set of substitution, deletion or insertion rules over the alphabet V. Formally: $(M \subseteq Sub_V)$ or $(M \subseteq Del_V)$ or $(M \subseteq Ins_V)$. The set M represents the set of evolutionary rules of the processor. As one can see, a processor is "specialized" in one evolutionary operation, only.
- $PI, FI \subseteq V$ are the *input* permitting/forbidding contexts of the processor, while PO, $FO \subseteq V$ are the *output* permitting/forbidding contexts of the processor. Informally, the permitting input/output contexts are the set of symbols that should be present in a word, when it enters/leaves the processor, while the forbidding contexts are the set of symbols that should not be present in a word in order to enter/leave the processor.

We denote the set of evolutionary processors over V by EP_V. Obviously, the evolutionary processor described here is a mathematical concept similar to that of an evolutionary algorithm, both being inspired from the Darwinian evolution. The rewriting operations we have considered might be interpreted as mutations and the filtering process described above might be viewed as a selection process. Recombination is missing but it was asserted that evolutionary and functional relationships between genes can be captured by taking only local mutations into consideration [13]. Furthermore, we are not concerned here with a possible biological implementation of these processors, though a matter of great importance.

An *accepting hybrid network of evolutionary processors* (AHNEP for short) is a 7-tuple $\Gamma = (V, U, G, \mathcal{N}, \alpha, \beta, x_I, x_O)$, where:

- V and U are the input and network alphabets, respectively, $V \subseteq U$.
- $G = (X_G, E_G)$ is an undirected graph, with the set of nodes X_G and the set of edges E_G. G is called the *underlying graph* of the network. In this paper, we consider *complete*

AHNEPs, i.e. AHNEPs having a complete underlying graph denoted by K_m, where m is the number of nodes. We say that m is the size of Γ and denote it by $size(\Gamma)$.

- $\mathcal{N} : X_G \longrightarrow EP_U$ is a mapping which associates with each node $x \in X_G$ the evolutionary processor $\mathcal{N}(x) = (M_x, PI_x, FI_x, PO_x, FO_x)$.
- $\alpha : X_G \longrightarrow \{*, l, r\}$; $\alpha(x)$ gives the action mode of the rules of node x on the words existing in that node.
- $\beta : X_G \longrightarrow \{(s), (w)\}$ defines the type of the *input/output filters* of a node. More precisely, for every node, $x \in X_G$, the following filters are defined:

$$\text{input filter: } \rho_x(\cdot) = \varphi^{\beta(x)}(\cdot; PI_x, FI_x),$$
$$\text{output filter: } \tau_x(\cdot) = \varphi^{\beta(x)}(\cdot; PO_x, FO_x).$$

  That is, $\rho_x(w)$ (resp. τ_x) indicates whether or not the word w can pass the input (resp. output) filter of x. More generally, $\rho_x(L)$ (resp. $\tau_x(L)$) is the set of words of L that can pass the input (resp. output) filter of x.
- x_I and $x_O \in X_G$ is the *input node*, and the *output node*, respectively, of the AHNEP.

We say that $card(X_G)$ is the size of Γ. Generally, the AHNEPs considered in the literature, and in this paper as well, have complete underlying graphs, namely graphs without loops in which every two nodes are connected.

A *configuration* of an AHNEP Γ as above is a mapping $C : X_G \longrightarrow 2^{V^*}$ which associates a set of words with every node of the graph. A configuration may be understood as the sets of words which are present in any node at a given moment. A configuration can change either by an *evolutionary step* or by a *communication step*.

When changing by an evolutionary step each component $C(x)$ of the configuration C is changed in accordance with the set of evolutionary rules M_x associated with the node x and the way of applying these rules $\alpha(x)$. Formally, we say that the configuration C' is obtained in *one evolutionary step* from the configuration C, written as $C \Longrightarrow C'$, if and only if $C'(x) = M_x^{\alpha(x)}(C(x))$ for all $x \in X_G$.

When changing by a communication step, each node processor $x \in X_G$ sends one copy of each word it has, which is able to pass the output filter of x, to all the node processors connected to x and receives all the words sent by any node processor connected with x providing that they can pass its input filter. Formally, we say that the configuration C' is obtained in *one communication step* from configuration C, written as $C \vdash C'$, if and only if $C'(x) = (C(x) - \tau_x(C(x))) \cup \bigcup_{\{x,y\} \in E_G} (\tau_y(C(y)) \cap \rho_x(C(y)))$ for all $x \in X_G$. Note that words which leave a node are eliminated from that node. If they cannot pass the input filter of any node, they are lost.

Let Γ be an AHNEP, the computation of Γ on the input word $w \in V^*$ is a sequence of configurations $C_0^{(w)}, C_1^{(w)}, C_2^{(w)}, \ldots$, where $C_0^{(w)}$ is the initial configuration of Γ defined by $C_0^{(w)}(x_I) = \{w\}$ and $C_0^{(w)}(x) = \emptyset$ for all $x \in X_G$, $x \neq x_I$, $C_{2i}^{(w)} \Longrightarrow C_{2i+1}^{(w)}$ and $C_{2i+1}^{(w)} \vdash C_{2i+2}^{(w)}$, for all $i \geq 0$. By the previous definitions, each configuration $C_i^{(w)}$ is uniquely determined by the configuration $C_{i-1}^{(w)}$. Otherwise stated, each computation in an AHNEP is deterministic. A computation as above immediately halts if one of the following two conditions holds:

(i) There exists a configuration in which the set of words existing in the output node x_O is non-empty. In this case, the computation is said to be an *accepting computation*.
(ii) There exist two identical configurations obtained either in consecutive evolutionary steps or in consecutive communication steps.

In the aforementioned cases the computation is said to be finite. The *language accepted* by Γ is

$$L_a(\Gamma) = \{w \in V^* \mid \text{ the computation of } \Gamma \text{ on } w \text{ is an accepting one}\}.$$

We say that an AHNEP Γ decides the language $L \subseteq V^*$, and write $L(\Gamma) = L$ if and only if $L_a(\Gamma) = L$ and the computation of Γ on every $x \in V^*$ halts.

Let Γ be an AHNEP deciding the language L. The *time complexity* of the finite computation $C_0^{(x)}, C_1^{(x)}, C_2^{(x)}, \ldots, C_m^{(x)}$ of Γ on $x \in L$ is denoted by $Time_\Gamma(x)$ and equals m. The time complexity of Γ is the function from $\mathbf{N}$ to $\mathbf{N}$,

$$Time_\Gamma(n) = \sup\{Time_\Gamma(x) \mid |x| = n\}.$$

For a function $f : \mathbf{N} \longrightarrow \mathbf{N}$ we define

$$\mathbf{Time}_{AHNEP_p}(f(n)) = \{L \mid L = L(\Gamma) \text{ for an AHNEP } \Gamma \text{ of size at most } p \text{halting on every input, such that } Time_\Gamma(n) \leq f(n) \text{ for some } n \geq n_0\}.$$

Moreover, we write $\mathbf{PTime}_{AHNEP_p} = \bigcup_{k\geq 0} \mathbf{Time}_{AHNEP_p}(n^k)$ for all $p \geq 1$ as well as $\mathbf{PTime}_{AHNEP} = \bigcup_{p\geq 1} \mathbf{PTime}_{AHNEP_p}$.

We can easily derive the following result from [2]:

Theorem 1 $\mathbf{NP} = \mathbf{PTime}_{AHNEP_7}$.

3 Decreasing the size of universal AHNEPs

In the following we show how a 2-tag system can be simulated by an AHNEP of size 6.

Theorem 2 *For every 2-tag system $T = (V, \phi)$ there exists a complete AHNEP Γ of size 6 such that $L(\Gamma) = \{w \mid T \text{ halts on } w\}$.*

Proof Let $V = \{a_1, a_2, \ldots, a_n, a_{n+1}\}$ be the alphabet of the tag system T with $a_{n+1} = H$ and $V' = V\backslash\{H\}$. We consider the AHNEP $\Gamma = (V', U, K_6, \mathcal{N}, \alpha, \beta, 1, 6)$ with the 6 nodes $1, 2, \ldots, 6$. The working alphabet of the network is defined as follows:

$$U = V \cup \left\{a_0', a_0''\right\} \cup \{a', a'', a^\circ \mid a \in V'\} \cup \{[x], \langle x\rangle, \prec x \succ \mid x \in X\},$$

where $X = \{x \in (V \cup \{a_0\})^*, |x| \leq \max\{|\phi(a)| \mid a \in V'\}$. The processors placed in the six nodes of the network are defined as follows:

- The node 1:
  - $M = \{a \to [\phi(a)], a \to a^\circ \mid a \in V'\}$,
  - $PI = V'$, $FI = U\backslash V'$, $PO = \{a^\circ \mid a \in V'\}$, $FO = \{a_{n+1}\}$,
  - $\alpha = *$, $\beta = (w)$.
- The node 2:
  - $M = \left\{\varepsilon \to a_0'\right\}$,
  - $PI = \{\prec a_0 x \succ, [x] \mid x \in X\backslash\{\varepsilon\}\}$, $FI = \{a', a'' \mid a \in V' \cup \{a_0\}\}$, $PO = U$, $FO = \emptyset$,
  - $\alpha = r$, $\beta = (w)$.

- The node 3:
  - $M = \{[a_k x] \rightarrow \langle a_{k-1} x\rangle, \prec a_k x \succ \rightarrow \langle a_{k-1} x\rangle, \prec a_0 a_k x \succ \rightarrow \langle a_{k-1} x\rangle \mid x \in X, 1 \leq k \leq n+1\} \cup \{a'_k \rightarrow a''_k \mid 0 \leq k \leq n+1\}$,
  - $PI = \{a'_0\} \cup \{\prec x \succ \mid x \in X \backslash (a_0 V^*)\}$, $FI = \emptyset$, $PO = \{\langle x\rangle \mid x \in X\}$, $FO = \emptyset$,
  - $\alpha = *, \beta = (w)$.
- The node 4:
  - $M = \{a''_{k-1} \rightarrow a'_k \mid 1 \leq k \leq n+1\} \cup \{a''_{k-1} \rightarrow a_k \mid 1 \leq k \leq n+1\} \cup \{\langle x\rangle \rightarrow \prec x \succ \mid x \in X\}$,
  - $PI = \{a''_k \mid 0 \leq k \leq n+1\}$, $FI = \{[x], \prec x \succ \mid a \in V, x \in X\}$, $PO = U$, $FO = \{a''_k \mid 0 \leq k \leq n+1\}$,
  - $\alpha = *, \beta = (w)$.
- The node 5:
  - $M = \{\prec a_0 \succ \rightarrow \varepsilon\} \cup \{a^\circ \rightarrow \varepsilon \mid a \in V'\}$,
  - $PI = \{\prec a_0 \succ\}$, $FI = \{a', a'' \mid a \in V' \cup \{a_0\}\}$, $PO = U$, $FO = \{a^\circ, \mid a \in V'\}$,
  - $\alpha = l, \beta = (w)$.
- The node 6
  - $M = \emptyset$,
  - $PI = \{a_{n+1}\}$, $FI = U \backslash V$, $PO = FO = \emptyset$,
  - $\beta = (w)$.

We show that Γ accepts a word w that does not contain H if an only if T eventually halts on w.

Let $w = aby$, $a, b \in V$, $y \in V^*$ be a word that does not contain H such that T eventually halts on w. We show how w can be accepted by Γ. At the beginning of the computation w is found in node 1, where the first symbol a can be replaced with $[\phi(a)]$ but the new string cannot pass the output filter of 1. In the next step, we can rewrite b as b°, getting the new word $[\phi(a)]b^\circ y$ which is sent out. It can only enter node 2 where the symbol a'_0 is inserted to its right end obtaining $[\phi(a)]b^\circ y a'_0$. This word can only enter node 3.

Let $\phi(a) = a_i x$, for some $1 \leq i \leq n+1$ and $x \in X$. In node 3, $[a_i x]b^\circ y a'_0$ is first converted into $[a_i x]b^\circ y a''_0$, which remains in 3, and then into $\langle a_{i-1} x\rangle b^\circ y a''_0$. This string is sent out and can only enter node 4, where it is rewritten as $\prec a_{i-1} x \succ b^\circ y a'_1$ via $\prec a_{i-1} x \succ b^\circ y a''_0$. If $i > 1$, this string first returns to node 3, resulting in $\langle a_{i-2} x\rangle b^\circ y a''_1$, and then returns to node 4. This process back and forth between nodes 3 and 4 continues until a string of the form $\langle a_0 x\rangle b^\circ y a''_{i-1}$ is obtained in node 3. Now the current word is sent to node 4 where it can be rewritten into $\prec a_0 x \succ b^\circ y a_i$. This string can only enter node 2, where a symbol a'_0 is inserted at the right end.

The resulting string $\langle a_0 x\rangle b^\circ y a_i a'_0$ is sent to node 3, where the same process recommences, with the only difference of a first rule $\prec a_0 a_k x \succ \rightarrow \langle a_{k-1} x\rangle$ used in order to eliminate the a_0 while decreasing the counter.

Finally, we will reach a string of the form $\prec a_0 \succ b^\circ y \phi(a)$ in node 4. This string can only enter node 5, where the symbol $\prec a_0 \succ$ is deleted, yielding $b^\circ y \phi(a)$. This string cannot leave the node, and in the next step, the symbol b° is deleted to give $y\phi(a)$. It is easy to see that we have correctly simulated the transition $a \rightarrow \phi(a)$ in the tag system.

Note that as soon as the current word contains H it enters the output node and input w is accepted. Otherwise, that is $\phi(a) \neq H$, then the current word enters again node 1, where the simulation of the next transition in T is performed.

We now argue why the above simulation is the only possible derivation in T, so that it halts on a word w if and only if w is accepted by Γ. In many steps, the derivation stated above is the only possible derivation. However, there are a few cases we need to consider more closely.

First of all, in node 1, only one substitution $a \to a^\circ$ can be performed before the string is sent out, but potentially zero or more than one $a \to [\phi(a)]$ substitutions. If no such substitution is performed, the resulting string cannot pass any input filter, so it is lost. If the outgoing word contains more than one symbol $[x]$, $x \in X$, then only one of them will be rewritten in node 3. The string is then sent out and also lost because it cannot pass any input filter.

Thus, an accepting computation is only possible if exactly one of each of the symbols a° and $[x]$ are present when leaving node 1. However, both symbols could be on any position of the string. Assume that they do not occupy the first two positions in the way described above. The simulation would then go on as described, until a string $y_1 \prec a_0 \succ y_2 b^\circ y_3$ or $y_1 b^\circ y_2 \prec a_0 \succ y_3$, $y_1, y_2, y_3 \in (V \backslash \{H\})^*$ is reached. The string will then enter node 5, where a symbol is only deleted at the left end of the string. If the leftmost symbol is in V', no deletion can be performed and the word remains trapped in node 5. If the leftmost symbol is of the type a°, it is deleted and the word is sent out. Since this word still contains a symbol $\prec a_0 \succ$, it cannot pass any input filter, and is lost. Finally, if the leftmost symbol is of the type $\prec a_0 \succ$, but the second one is not of type a°, then $\prec a_0 \succ$ can be deleted, after which the word remains trapped as in the first case.

In each of the nodes 3 and 4, two rules are applied. If the rules are applied in a different order than described above, the derivation cannot lead to acceptance. Indeed, suppose that in node 3, we apply a rule $\prec a_k x \succ \to \langle a_{k-1} x \rangle$ to the incoming word $\prec a_i x \succ b^\circ y a'_j$, giving $\langle a_{i-1} x \rangle b^\circ y a'_j$. This word is sent out of the node, but cannot enter node 4 nor any other node, and it is lost. Similarly, in node 4, if we first apply a rule $a''_{k-1} \to a'_k$, we obtain a word of the form $\langle a_i x \rangle b^\circ y a'_j$, which leaves the node but does not enter any other node. Then, in node 4, a symbol a'_i can be rewritten as a_{i+1} while the created $\prec a_j x \succ$ have $j \neq 0$. The resulting string can pass the output filter, and then it can enter node 3; however, here no symbol a'' can be obtained, meaning that after conversion of the symbol $\prec a_j x \succ$ to $\langle a_j x \rangle$, the string is sent out and lost. Similarly, in node 4 symbol a'_i can be rewritten as a''_{i+1}, together with the appearance of $\prec a_0 x \succ$. Again, the resulting string can pass the output filter, but none of the input filters.

This covers all possible cases, thus proving that if $w \in L(\Gamma)$ then T will eventually halt on w. □

Since 2-tag systems are universal [8], the following corollary is immediate:

Corollary 1 *There exists a universal AHNEP with* 6 *nodes.*

This result improves the result reported in [2] where a universal AHNEP with seven nodes was constructed. In fact, we expect that this bound is very close to the absolute lower bound. Indeed, it was shown in [1,2] that AHNEPs with two nodes are not universal. Clearly, an insertion node is necessary to obtain the infinite workspace required for universal computation. Moreover, we conjecture that also at least one substitution and one deletion node is needed for universality. Thus, we leave as open questions determining the power of AHNEPs of size 4 and 5, which would definitively settle this issue.

4 Decreasing the size of AHNEPs efficiently accepting recursively enumerable languages

Although 2-tag systems efficiently simulate deterministic Turing machines, via cyclic tag systems (see, e.g., [14]), the previous result does not allow us to infer a bound on the size of the networks accepting in a computationally efficient way all recursively enumerable languages. We now discuss how an efficient (from the time and size complexity points of view) AHNEP accepting (deciding) every recursively enumerable (recursive) language can be constructed.

Theorem 3 *For any recursively enumerable (recursive) language L accepted (decided) by a Turing machine there exists a complete AHNEP Γ, of size 10, accepting (deciding) L. Moreover, if $L \in NTIME(f(n))$, then $Time_\Gamma(n) \in \mathcal{O}(f(n))$. (The constant hidden by the $\mathcal{O}$ notation depends on L.)*

Proof Let $M = (Q, V, W, q_0, B, F, \delta)$ be a nondeterministic Turing machine. We regard the working alphabet W as the ordered alphabet $\{a_1, a_2, \ldots, a_n\}$, such that $V = \{a_1, \ldots, a_m\}$ for some $m < n$ and $B = a_n$. Also let $a_0, a_0', \$, \$^+, \#, \perp, \Delta$ be symbols not contained in W. We may assume without loss of generality that the Turing machine M verifies the property that $\{(q_0, a, X) \mid a \in W, x \in \{L, R\}\} \cap \delta(Q, W) = \emptyset$. We consider the AHNEP $\Gamma_M = (V, U, G, \mathcal{N}, \alpha, \beta, 1, 10)$ with ten nodes (labeled with the numbers 1–10). The working alphabet of the network is defined as follows:

$$\begin{aligned} U = \big\{&[q_1, a, q_2, b, X], [q_1, a, q_2, b, X]^\circ, [q_1, a^*, q_2, b, X], \\ &[q_1, \$, q_2, b^*, X], [q_1, \$, q_2, \#, X] \mid q_1, q_2 \in Q, a, b \in W, X \in \{R, L\}\big\} \\ &\cup \{a, a', a^\dagger, a^\diamond, a^*, a^+, a^\circ \mid a \in W\} \cup \big\{a_0', \$, \$^+, \#, \perp, \Delta\big\} \end{aligned}$$

The processors placed in the ten nodes of the network are defined as follows (where only those filters which do not allow all strings to pass are specified):

- The node 1 (the input node of the network):
  - $M(1) = \big\{\varepsilon \to B', \varepsilon \to a_0'\big\}$,
  - $\alpha(1) = r, \beta(1) = (w)$
  - $PI(1) = \{B\}$, $FI(1) = U \backslash (W \cup \{[q_1, \$, q_2, b, R] \mid q_1, q_2 \in Q, b \in W\})$.
- The node 2
  - $M(2) = \{\varepsilon \to [q_0, a, q_1, b, X] \mid q_1 \in Q, a, b \in W, X \in \{R, L\}, (q_1, b, X) \in \delta(q_0, a)\}$,
  - $\alpha(2) = r, \beta(2) = (w)$
  - $PI(2) = \{B'\}$, $FI(2) = U \backslash (V \cup \{B'\})$.
- The node 3:
  - $M(3) = \{B' \to B\} \cup \{a \to a^*, a \to a^\diamond \mid a \in W\} \cup \{[q_1, a, q_2, b, X] \to [q_1, a^*, q_2, b, X] \mid q_1, q_2 \in Q, a, b \in W, X \in \{R, L\}\} \cup \{[q_1, \$, q_2, \#, L] \to [q_2, a, q_3, b, X]^\circ \mid q_1, q_2, q_3 \in Q, a, b \in W, X \in \{R, L\}, (q_3, b, X) \in \delta(q_2, a)\} \cup \{[q_1, \$, q_2, \#, R] \to \Delta \mid q_1 \in Q, q_2 \in F\} \cup \{[q_1, \$, q_2, \#, L] \to \Delta^\circ \mid q_1 \in Q, q_2 \in F\} \cup \{[q_1, \$, q_2, \#, R] \to [q_2, a^*, q_3, b, X] \mid q_1, q_2, q_3 \in Q, a, b \in W, X \in \{R, L\}, (q_3, b, X) \in \delta(q_2, a)\}$,

- $\alpha(3) = *, \beta(3) = (w)$
- $PI(3) = \{[q_1, a, q_2, b, X] \mid q_1, q_2 \in Q, a, b \in W, X \in \{R, L\}\} \cup \{a_0'\}$, $FI(3) = U \backslash (PI(3) \cup \{B'\} \cup W \cup \{[q_1, \$, q_2, \#, X] \mid q_1, q_2 \in Q, X \in \{R, L\}\})$, $PO(3) = \{a^*, a^\diamond \mid a \in W\} \cup \{\Delta\}$, $FO(3) = (PI(3) \cup \{B'\} \cup \{[q_1, \$, q_2, \#, X] \mid q_1, q_2 \in Q, X \in \{R, L\}\}) \backslash \{a_0'\}$.

- The node 4:
  - $M(4) = \{a_i^* \to a_{i-1}^+ \mid n \geq i > 1\} \cup \{a_1^* \to \$^+\} \cup \{a_i^\diamond \to a_{i-1}^\dagger \mid n \geq i > 1\} \cup \{a_1^\diamond \to \bot^\dagger\} \cup \{a_i' \to a_{i+1}^\circ \mid n \geq i > 1\} \cup \{a_1^* \to \$^+\} \cup \{[q_1, a_i^*, q_2, b, X] \to [q_1, a_{i-1}^+, q_2, b, X], [q_1, a_1^*, q_2, b, X] \to [q_1, \$^+, q_2, b, X] \mid q_1, q_2 \in Q, b \in W, n \geq i > 1, X \in \{R, L\}\} \cup \{[q_1, \$, q_2, a_i^*, X] \to [q_1, \$, q_2, a_{i-1}^+, X], [q_1, \$, q_2, a_i, X] \to [q_1, \$, q_2, a_{i-1}^+, X], [q_1, \$, q_2, a_1^*, X] \to [q_1, \$, q_2, \#^+, X] \mid q_1, q_2 \in Q, n \geq i > 1, X \in \{R, L\}\} \cup \{[q_1, a, q_2, b, X]^\circ \to [q_1, a, q_2, b, X] \mid q_1, q_2 \in Q, a, b \in W, X \in \{R, L\}\} \cup \{\Delta^\circ \to \Delta\}$,
  - $\alpha(4) = *, \beta(4) = (w)$
  - $PI(4) = \{a^*, a', a^\diamond \mid a \in W\}$, $FI(4) = U \backslash (PI(4) \cup W \cup \{\Delta^\circ, \Delta\} \cup \{[q_1, a^*, q_2, b, X] \mid q_1, q_2 \in Q, a, b \in W, X \in \{R, L\}\} \cup \{[q_1, \$, q_2, a^*, X], [q_1, \$, q_2, a, X] \mid q_1, q_2 \in Q, a \in W, X \in \{R, L\}\} \cup \{[q_1, a, q_2, b, X]^\circ, [q_1, a, q_2, b, X] \mid q_1, q_2 \in Q, a, b \in W, X \in \{R, L\}\})$ $PO(4) = \{a^+, a^\circ, a^\dagger \mid a \in W\} \cup \{\$^+, \bot^\dagger\}$, $FO(4) = \{[q_1, a^*, q_2, b, X] \mid q_1, q_2 \in Q, a, b \in W, X \in \{R, L\}\} \cup \{[q_1, \$, q_2, a^*, X], [q_1, \$, q_2, a, X] \mid q_1, q_2 \in Q, a \in W, X \in \{R, L\}\} \cup \{[q_1, a, q_2, b, X]^\circ \mid q_1, q_2 \in Q, a, b \in W, X \in \{R, L\}\} \cup \{\Delta^\circ\} \cup PI(4)$.

- The node 5:
  - $M(5) = \{a^+ \to a^*, a^\dagger \to a^\diamond, a^\circ \to a', a^\circ \to a \mid a \in W\} \cup \{\$^+ \to \$, \bot^\dagger \to \bot\} \cup \{[q_1, a^+, q_2, b, X] \to [q_1, a^*, q_2, b, X] \mid q_1, q_2 \in Q, a, b \in W, X \in \{R, L\}\} \cup \{[q_1, \$^+, q_2, b, X] \to [q_1, \$, q_2, b, X] \mid q_1, q_2 \in Q, b \in W, X \in \{R, L\}\} \cup \{[q_1, \$, q_2, a^+, X] \to [q_1, \$, q_2, a^*, X] \mid q_1, q_2 \in Q, a \in W, X \in \{R, L\}\} \cup \{[q_1, \$, q_2, \#^+, X] \to [q_1, \$, q_2, \#, X] \mid q_1, q_2 \in Q, X \in \{R, L\}\}$,
  - $\alpha(5) = *, \beta(5) = (w)$
  - $PI(5) = \{a^+, a^\circ, a^\dagger \mid a \in W\} \cup \{\$^+, \bot^\dagger\}$, $FI(5) = U \backslash (W \cup PI(5) \cup \{\Delta, [q_1, a, q_2, b, X], [q_1, a^+, q_2, b, X], [q_1, \$^+, q_2, b, X], [q_1, \$, q_2, a^+, X], [q_1, \$, q_2, \#^+, X] \mid q_1, q_2 \in Q, a, b \in W, X \in \{R, L\}\})$, $PO(5) = \{[q_1, a, q_2, b, X], [q_1, a^*, q_2, b, X], [q_1, \$, q_2, b, X], [q_1, \$, q_2, a^*, X], [q_1, \$, q_2, \#, X] \mid q_1, q_2 \in Q, a, b \in W, X \in \{R, L\}\}$, $FO(5) = PI(5)$.

- The node 6:
  - $M(6) = \{\$ \to \varepsilon\}$,
  - $\alpha(6) = l, \beta(6) = (w)$
  - $PI(6) = \{\$\}$, $FI(5) = U \backslash (\{\$\} \cup W \cup \{[q_1, \$, q_2, b, X] \mid q_1, q_2 \in Q, b \in W, X \in \{R, L\}\})$.

- The node 7:
  - $M(7) = \{\varepsilon \to a_0'\}$,
  - $\alpha(7) = l, \beta(7) = (w)$
  - $PI(7) = \{B\}$, $FI(7) = U \backslash (W \cup \{[q_1, \$, q_2, b, L], [q_1, \$, q_2, \#, L] \mid q_1, q_2 \in Q, b \in W\})$.

- The node 8:
  - $M(8) = \{\varepsilon \to B\}$,
  - $\alpha(8) = l, \beta(8) = (w)$
  - $PI(8) = \{[q_1, \$, q_2, b, X] \mid q_1, q_2 \in Q, b \in W, X \in \{R, L\}\}$, $FI(8) = \{B, B'\} \cup (U \backslash (W \cup PI(8)))$.
- The node 9:
  - $M(9) = \{\perp \to \varepsilon\}$,
  - $\alpha(9) = r, \beta(9) = (w)$
  - $PI(9) = \{\perp\}$, $FI(9) = U \backslash (W \cup \{\perp\} \cup \{[q_1, \$, q_2, \#, X] \mid q_1, q_2 \in Q, X \in \{R, L\}\})$.
- The node 10 (the output node of the network):
  - $\beta(10) = (w)$, $PI(10) = \{\Delta\}$, $FI(10) = U \backslash (W \cup \{\Delta\})$.

Before showing that Γ_M accepts exactly $L(M)$ we describe the meaning of a class of new symbols that we use: A symbol $[q, a, q', b, X]$ shall mean that the Turing machine M is now in state q and will go to state q' by reading a, will write b and move in the direction X. Thus these symbols store the complete information of a transition of the machine. The simulation of the machine's computation will then consist of:

1. First, introducing a blank symbol at the leftmost end of the input word (in order to mark the beginning of the infinite part, containing only blank symbols, of the Turing machine's tape), and then introducing the symbol for a transition starting from the initial state. At this moment, the string has the general form $v_1 B[q, a, q', b, X]v_2$, which corresponds to a configuration of M where the current state is q, the tape content is $v_2 v_1 BB \ldots$ and the head of M reads the first symbol of v_1 (or B, if $v_1 = \varepsilon$). Of course, initially, $v_2 = \varepsilon$ and q is the initial state of M.
2. Checking whether the first letter of the input word is the same as the one read in that transition, and, if this is true, deleting that letter.
3. Writing the correct letter (the one stored as the fourth component of the symbol encoding the transition) in the correct position, rotating the string such the first symbol is the one that will be read by the Turing machine, and, also, replacing the transition's symbol by the one of a possible following transition.
4. Repeating the above procedure, from step 2, for the newly obtained string, encoding a new configuration of the Turing machine M. In other words, we simulate a new step of the Turing machine.

The main technical problems here are matching the input word's letters with the ones in the symbols corresponding to transitions, writing the correct new symbol in the corresponding position, and, if necessary, rotating the string. Because none of the rules have left hand sides of length two or longer, it is not possible to let one rule application verify these equalities. Therefore, in the first case, we will verify the equality of the two letters by first marking both of them and then decreasing them simultaneously according to the order of the alphabet. If they matched in the beginning, then they will we both transformed into \$ in the same cycle, and only then the computation should proceed. In the second case, the process is quite similar: we introduce a'_0 in the correct position (according to the last component of the symbol encoding the transition of M, which tells us how the head of the machine moves), and, then we start increasing it while simultaneously decreasing the letter that should be written from the transition-symbol. Once this last letter is rewritten into # we stop the increasing of the former one, and we have simulated the writing operation of the Turing machine. Finally, in

the case of a rotation of the string, we proceed just as in the second case, but now we mark, and decrease, the letter that should be moved from one end of the string to the other; once we cannot decrease it anymore, we delete it. Note that in the case of the first verification, as well as in the case of the rotation, we cannot tell if the letter we picked and decreased is found at the left end, respectively right end, of the string; thus, we need to verify, when deleting it, that it really is in the correct position. If any of the above verification fails (that is, the synchronous decreasing/increasing of the symbols could not be carried out, or the deletion could not be made because the letters were not in the correct position) then the filters of the nodes do not allow the string to influence the rest of the computation (i.e., the string is either lost or blocked).

In the following we show that Γ_M accepts a word w if and only if $w \in L(M)$. Note that in our simulation,

1. $L(\Gamma_M) \supseteq L(M)$.
   Let w be a word from $L(M)$. Assume that w is present in the node 1 at the beginning of the computation. In this node the string becomes wB' and it is communicated in the network. It can only enter the node 2, where it is transformed into $wB'[q_0, a, q_1, b, X]$, with $w = aw'$ and $(q_1, b, X) \in \delta(q_0, a)$; then, the node is communicated to all the other nodes of the network, and enters node 3. The string becomes in two evolutionary steps $a^*w'B[q_0, a^*, q_1, b, X]$ and is communicated to the other nodes. The order of the steps is irrelevant, since in both cases the intermediate product stays in node 3. Now a so-called *Simulation Phase* begins; during this phase a move of the Turing machine M is simulated. Assume that the content of M's tape is ya_kx, its current state is q_1, and it executes the move $(q_2, a_t, X) \in \delta(q_1, a_k)$, $q_1, q_2 \in Q$, $1 \leq k, t \leq n$, $X \in \{R, L\}$ and $x, y \in W^*$. Assume, also, that at the beginning of this phase the string $a_k^*x[q_1, a_k^*, q_2, a_t, X]y$, $y \in W^*$, is present in node 3 and the network starts the execution of a communication step (this assumption clearly holds after the first four evolutionary steps). Now the first cycle of the phase begins. The string enters node 4 where it becomes $a_{k-1}^+x[q_1, a_{k-1}^+, q_2, a_t, X]y$ and then goes to node 5 where it is transformed into $a_{k-1}^*x[q_1, a_{k-1}^*, q_2, a_t, X]y$; further, it goes back to node 4. The string follows repeatedly the path from node 4 to node 5 until it becomes $a_1^*x[q_1, a_1^*, q_2, a_t, X]y$ and enters node 4. Here it is transformed into $\$^+x[q_1, \$^+, q_2, a_t, X]y$ and enters again node 5. Finally, it is transformed in this node into $\$x[q_1, \$, q_2, a_t, X]y$ and communicated to node 6. Here the leftmost symbol is deleted if and only if it is \$, thus the string becomes $x[q_1, \$, q_2, a_t, X]y$. If $x = \varepsilon$, the string enters node 8 where B is inserted as its leftmost symbol, and the string is communicated in the network. In both cases, $x \neq \varepsilon$ or $x = \varepsilon$, we are at a point when a string $x'[q_1, \$, q_2, a_t, X]y$ (with x' having its last symbol B) was communicated in the network. If $X = L$ this string enters node 7 where a_0' is inserted as its leftmost symbol; otherwise it enters node 1 where a_0' is inserted as its rightmost symbol. This string ($a_0'x'[q_1, \$, q_2, a_t, L]y$ or $x'[q_1, \$, q_2, a_t, R]ya_0'$, respectively), is communicated in the network and enters node 4; here it is transformed into $a_1^\circ x'[q_1, \$, q_2, a_{t-1}^+, L]y$ or, respectively, $x'[q_1, \$, q_2, a_{t-1}^+, R]ya_1^\circ$ and goes to node 5. The second cycle begins with these moves. Again, the string follows repeatedly the path from node 4 to node 5 and back, until it becomes $a_t^\circ x'[q_1, \$, q_2, \#^+, L]y$ or, respectively, $x'[q_1, \$, q_2, \#^+, R]ya_t^\circ$ and leaves node 4. In node 5 this string is transformed into $a_tx'[q_1, \$, q_2, \#, L]y$ or, respectively, $x'[q_1, \$, q_2, \#, R]ya_t$. The string $x'[q_1, \$, q_2, \#, R]ya_t$ enters node 3 and here it is transformed into $a^*x''[q_2, a^*, q_3, b, X]ya_t$, given that $x' = ax''$ and $(q_3, b, X) \in \delta(q_2, a)$, or $x'\Delta ya_t$, if $q_2 \in F$; the Simulation Phase is restarted in the first case, while in the second case the string exits node 3, enters node 10, the input string being accepted. The

string $a_t x'[q_1, \$, q_2, \#, L]y$ enters node 7 where it becomes $a'_0 a_t x'[q_1, \$, q_2, \#, L]y$ and then node 3 where it is transformed into $a'_0 a_t x'[q_2, a_l, q_3, b, X]^\circ y' a_l^\diamond$, if $(q_3, b, X) \in \delta(q_2, a_l)$, or $a'_0 a_t x' \Delta^\circ y' a_l^\diamond$, if $q_2 \in F$. In both cases, the string enters node 4 where it becomes $a_1^\circ a_t x'[q_2, a_l, q_3, b, X]y' a_{l-1}^\dagger$ or $a_1^\circ a_t x' \Delta y' a_{l-1}^\dagger$. Then it is communicated in the network, enters node 5, and it is transformed into $a'_1 a_t x'[q_2, a_l, q_3, b, X]y' a_{l-1}^\diamond$ or, respectively, $a'_1 a_t x' \Delta y' a_{l-1}^\diamond$. The third cycle begins with these moves. The string is processed repeatedly by nodes 4 and 5 until it becomes $a'_l a_t x'[q_2, a_l, q_3, b, X]y' \perp^\diamond$, or, respectively, $a'_l a_t x' \Delta y' \perp^\diamond$ in node 4. Then it enters node 5 where it is transformed into $a_l a_t x'[q_2, a_l, q_3, b, X]y' \perp$, or, respectively, $a_l a_t x' \Delta y' \perp$. Now it enters node 9 where the $\perp$ symbol is deleted. Further, in the first case, the string enters node 3 where it becomes $a_l^* a_t x'[q_2, a_l^*, q_3, b, X]y'$ and the Simulation Phase is restarted, or, in the second case, it enters the output node of the network, and the input string is accepted. To conclude, the above show that in one execution of the Simulation Phase, we simulate a move of the Turing machine M. If M accepts w, in $f(w)$ moves, Γ_M accepts w after the Simulation Phase is executed for $f(|w|)$ times.

2. $L(\Gamma_M) \subseteq L(M)$.
In most cases the filters ensure that the computation can be performed only as described above. Moreover, by the mechanism described before the first inclusion (which is, in fact, quite similar to that in Theorem 2), if the synchronization processes, performed in each of the three cycles by nodes 4 and 5, are unsuccessful, the resulting strings will be lost or blocked forever in a node. However, some cases require some closer attention. First, at the beginning of the computation on w there are two possible insertions that can be made at the rightmost end of w: we can insert a B' symbol, and the computation may continue as described above, or we can insert an a'_0 symbol. In the latter case, the string obtained can enter node 3, and after this node processes the string it goes to node 4, or directly node 4; in both cases, in node 4 a'_0 becomes a_1° and the string enters node 5, where it is blocked since it does not contain a symbol encoding a move of the Turing machine, thus cannot pass the output filters. Also, if some other time, during the computation, a string enters node 1 and B' is inserted in this string, then it cannot enter any node, and is lost. Further, assume that, at the beginning of the first cycle of the simulation phase, other symbol a_p, with $p \leq n$, than the first symbol of the string communicated in the network is transformed into a_p^*. This symbol is transformed into \$ in p iterations of the cycle, and the string, communicated by node 5, can enter node 6 only in the case when $p = k$, otherwise it cannot enter any node and is lost. Here the \$ is not deleted, and the string is, once again, lost. Also, to see that there cannot be harmful interference between different cycles, assume that before the execution of the first cycle a symbol a_p, $p \leq n$, is transformed by node 3 into $a_p^\diamond$. Again, in at most p steps either the string will contain a symbol $[q_1, \$, q_2, b, X]$ and one of the symbols $a^\diamond$, with $a \in W$, or $\perp$, so cannot enter any node and is lost. Consequently, the only symbols that can be transformed in the first cycle are the first symbol of the string and the symbol $[q_1, a, q_2, b, X]$. Similar arguments show that if a symbol a is rewritten to a^*, after the second cycle, no accepting computation can follow, thus the first cycle and the third one cannot interfere. Also, it is not hard to see that the second cycle cannot interfere with any of the other two. Finally, during the second cycle, if the string processed by the network becomes $a'_t x'[q_1, \$, q_2, \#, L]y$, $x'[q_1, \$, q_2, \#, R]y a'_t, a_t x'[q_1, \$, q_2, a_l^*, L]y$ or $x'[q_1, \$, q_2, a_l^*, R]y a_t$, the computation on this string blocks. These considerations show that only the strings that are processed during the iterative phase as described in the proof of the inclusion 1 can be accepted by the network. Thus $L(\Gamma_M) \subseteq L(M)$ and we have proved that $L(\Gamma_M) = L(M)$.

It is clear that if M stops on the input string w, in $f(|w|)$ steps, then Γ_M stops on the input string w after $f(|w|)$ executions of the Simulation Phase, described above. □

Acknowledgments We would like to thank the anonymous referees for their comments and suggestions, which improved the presentation of this paper.

References

1. Alhazov, A., Csuhaj-Varju, E., Martin-Vide, C., Rogozhin, Y.: About universal hybrid networks of evolutionary processors of small size. Lect. Notes Comput. Sci. **5196**, 28–39 (2008)
2. Alhazov, A., Csuhaj-Varju, E., Martin-Vide, C., Rogozhin, Y.: On the size of computationally complete hybrid networks of evolutionary processors. Theor. Comput. Sci. **410**(35), 3188–3197 (2009)
3. Hartmanis, J., Stearns, R.E.: On the computational complexity of algorithms. Trans. Am. Math. Soc. **117**, 533–546 (1965)
4. Hillis, W.D.: The Connection Machine. MIT, Cambridge (1985)
5. Manea, F., Martin-Vide, C., Mitrana, V.: On the size complexity of universal accepting hybrid networks of evolutionary processors. Math. Struct. Comput. Sci. **17**(4), 753–771 (2007)
6. Manea, F., Mitrana, V.: All NP-problems can be solved in polynomial time by accepting hybrid networks of evolutionary processors of constant size. Inf. Proc. Lett. **103**(3), 112–118 (2007)
7. Margenstern, M., Mitrana, V., Perez-Jimenez, M.: Accepting hybrid networks of evolutionary systems. Lect. Notes Comput. Sci. **3384**, 235–246 (2005)
8. Minsky, M.L.: Size and structure of universal turing machines using tag systems. Recursive Function Theory, Symposium in Pure Mathematics, vol. 5, pp. 229–238 (1962)
9. Papadimitriou, C.H.: Computational Complexity. Addison-Wesley, Reading (1994)
10. Post, E.L.: Formal reductions of the general combinatorial decision problem. Am. J. Math. **65**, 197–215 (1943)
11. Rogozhin, Y.: Small universal turing machines. Theor. Comput. Sci. **168**, 215–240 (1996)
12. Rozenberg, G., Salomaa, A. (eds.): Handbook of Formal Languages, vol. I–III. Springer, Berlin (1997)
13. Sankoff, D., et al.: Gene order comparisons for phylogenetic inference: evolution of the mitochondrial genome. Proc. Natl. Acad. Sci. USA **89**, 6575–6579 (1992)
14. Woods, D., Neary, T.:On the time complexity of 2-tag systems and small universal Turing machines. 47th Annual IEEE symposium on foundations of computer science FOCS '06, pp. 439–448 (2006)