Networks of Evolutionary Processors: Computationally Complete Normal Forms

Jürgen Dassow
Otto-von-Guericke-Universität Magdeburg, Fakultät für Informatik
PSF 4120, D-39016 Magdeburg, Germany
dassow@iws.cs.uni-magdeburg.de

Florin Manea
Christian-Albrechts-Universität zu Kiel, Institut für Informatik
D-24098 Kiel, Germany
and
Faculty of Mathematics and Computer Science, University of Bucharest,
Str. Academiei 14, RO-010014 Bucharest, Romania
flm@informatik.uni-kiel.de

Bianca Truthe
Otto-von-Guericke-Universität Magdeburg, Fakultät für Informatik
PSF 4120, D-39016 Magdeburg, Germany
truthe@iws.cs.uni-magdeburg.de

Abstract. Networks of evolutionary processors (NEPs, for short) form a bio-inspired language generating computational model that was shown to be equivalent to the model of phrase-structure grammars. In this paper, we analyse different restricted variants of NEPs that preserve the computational power of the general model. We prove that any recursively enumerable language can be generated by a NEP where the derivation rules can be applied at arbitrarily chosen positions, the control of the communication is done by finite automata with at most three states, and either the rule sets are singletons or the underlying graph is a complete graph. If one uses networks with arbitrary underlying graphs and allows the additional application of insertions and deletions only to the right-most or the to left-most position of the derived words for some nodes, then we only need automata with only one state to control the communication in the network. Clearly, this result is optimal; moreover, finite automata with two states are necessary and sufficient in order to generate all the recursively enumerable languages when the derivation rules can be applied only at arbitrarily chosen positions.

Keywords: Bio-inspired Language Generating Models, Generating Networks of Evolutionary Processors, Computational Completeness, Normal Form, Restricted Filtering

1. Introduction

Motivated by several basic computing paradigms for parallel and distributed symbolic processing (Hillis, 1986; Errico and Jesshope, 1994; Fahlman et al., 1983), E. Csuhaj-Varjú and A. Salomaa defined networks of language processors as a formal languages generating model (Csuhaj-Varjú and Salomaa, 1997). Such a network can be viewed as a graph whose nodes contain a set of production rules each and, at any moment of time, a language is associated with a node. In a derivation step, any node derives from its language all possible words as its new language. In a communication step, any node sends those words that satisfy filtering conditions, that require the membership to a given regular language, to other nodes; any node receives those words sent by the other nodes that satisfy input conditions, also requiring the membership to a regular language. The language generated by a network of language processors consists of all words which occur in the languages associated with a given node.

In (Csuhaj-Varjú and Mitrana, 2000), one considers a computing model inspired by the evolution of cell populations. More precisely, a formal language generating model was used to model some properties of evolving cell communities at the syntactical level. In this model, cells were represented by words which describe their DNA sequences and the possible events that may occur in their evolution (for instance, mutations and division) were given as formal operations on words. Informally, at any moment of time, the evolutionary system is described by a collection of words, a representation of a collection of cells. Cells belong to species and their community evolves according to the events (operations) that may be applied on them. Only those cells which are represented by a word in a given set of words, called the genotype space of the species, are accepted as the surviving (correct) ones. This feature parallels with the natural processes of evolution and selection. Similar ideas may be met in other bio-inspired models, such as *tissue-like membrane systems* (Păun, 2000) as well as models from the area of distributed computing, like *parallel communicating grammar systems* (Păun and Sântean, 1989).

In this context, networks of evolutionary processors (NEPs for short) were defined (Castellanos et al., 2003). More precisely, one considers that in each node of such a network (a directed graph) there exists a processor which is able to perform very simple operations that mimic the point mutations in a DNA sequence (insertion, deletion, or substitution of a pair of nucleotides). Moreover, each node is specialized just for one of these evolutionary operations. Furthermore, the data in each node are organized in the form of multisets of words, each word appearing in an arbitrarily large number of copies and all the copies are processed as follows: if at least one rule can be applied to a word w, we obtain all the words that are derived from the word w by applying exactly one of the possible rules at exactly one feasible position in the word w; otherwise, w remains unchanged. We stress that this computational process is not exactly an evolutionary process in the Darwinian sense, but the rewriting operations considered might be viewed as mutations and the filtering process 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 ((Sankoff et al., 1992)). The computation of a NEP is conducted just as in the case of networks of language processors: initially, the nodes contain some finite sets of words, further, these words are processed according to the rules in each node, and then, they are communicated to the other nodes, as permitted by some filtering condition associated with the nodes, and so on; the language generated by a NEP consists of all the words which appear in a given node, called the output node. Results on NEPs, seen as formal languages generating devices, can be found, e. g., in (Martín-Vide and Mitrana, 2005; Castellanos et al., 2003; Alhazov et al., 2009b; Alhazov et al., 2009a; Castellanos et al., 2001). In the seminal paper (Castellanos et al., 2003), it was shown that such networks are computationally complete, i. e., they are able to generate all recursively enumerable languages; however, in these various constructions, different types of underlying graphs and relatively large automata for the control of the communication were used.

In this paper, we show that some aspects of evolutionary networks can be normalized or simplified while preserving the generative power. Especially, we are interested in a use of very small finite automata for the control of the communication. We first prove that the networks with evolutionary processors remain computationally complete if one restricts the control automata to have only one state. However, the resulting underlying graphs have no fixed structure and the rules are applied in three different modes. In the remaining part of the paper, we show that one can generate all recursively enumerable languages by networks where the rules are applied arbitrary (any rule can be applied to any position) and either all the rule sets are singletons or the underlying graph is a complete graph. However, the automata used to control the communication are a little larger; they have at most two states.

2. Definitions

We assume that the reader is familiar with the basic concepts of formal language theory (see, e. g., *Handbook of Formal Languages* (Rozenberg and Salomaa, 1997)). We here only recall some notations used in the paper.

By V^* we denote the set of all words (strings) over an alphabet V (including the empty word λ). The length of a word w is denoted by $|w|$. The number of occurrences of a letter a or of letters from a set A is denoted by $|w|_a$ and $|w|_A$, respectively. For the number of elements of a set A, we write $\#(A)$. The minimal alphabet of a word w (respectively, of a language L) is denoted by $alph(w)$ (and, $alph(L)$, respectively). In the proofs we shall often add the letters of an alphabet U to a given alphabet V; in all these situations, we assume that $V \cap U = \emptyset$.

A phrase structure grammar is a quadruple $G = (N, T, P, S)$ where N is a set of non-terminals, T is a set of terminals, P is a finite set of productions which are written as $\alpha \to \beta$ with $\alpha \in (N \cup T)^+ \setminus T^*$ and $\beta \in (N \cup T)^*$, and $S \in N$ is the axiom. A grammar $G = (N, T, P, S)$ is in Geffert normal form (Geffert, 1991) if the set of non-terminals only consists of the axiom S and three additional letters A, B, C and all rules in P have the form $ABC \to \lambda$ or $S \to v$ with $v \in (N \cup T)^*$.

By REG and RE we denote the families of regular and recursively enumerable languages, respectively. For $i \in \mathbb{N}$, we designate the family of all languages L that can be accepted by a deterministic finite automaton with at most i states working over the alphabet $alph(L)$ by MIN_i.

We call a production $\alpha \to \beta$ a substitution rule if $|\alpha| = |\beta| = 1$, and, respectively, a deletion rule if $|\alpha| = 1$ and $\beta = \lambda$. We introduce insertion rules as a counterpart of deletion rules and write such a rule as $\lambda \to a$, where a is a letter.

Besides the usual context-free rewriting we also consider derivations where the rules are applied to the left or right end of the word. Formally, for a substitution, deletion or insertion rule $p : \alpha \to \beta$ and words v and w, we define

- $v \Longrightarrow_{*,p} w$ by $v = x\alpha y$, $w = x\beta y$ for some $x, y \in V^*$,
- $v \Longrightarrow_{l,p} w$ by $v = \alpha y$, $w = \beta y$ for some $y \in V^*$,
- $v \Longrightarrow_{r,p} w$ by $v = x\alpha$, $w = x\beta$ for some $x \in V^*$.

The indices are omitted in most of the cases, as they can be easily determined from the context.

We now introduce the basic concept of this paper, the networks of evolutionary processors with regular filters.

DEFINITION 1.

- *A network of evolutionary processors (for short, NEP) of size n with filters of the set X is a tuple*

$$\mathcal{N} = (V, N_1, N_2, \ldots, N_n, E, j)$$

where

- V *is a finite alphabet,*
- *for* $1 \leq i \leq n$, $N_i = (M_i, A_i, I_i, O_i, \alpha_i)$ *where*
  - $*$ M_i *is a set of evolutionary rules of a certain type, i. e.,* $M_i \subseteq \{ a \to b \mid a, b \in V \}$ *or* $M_i \subseteq \{ a \to \lambda \mid a \in V \}$, *or* $M_i \subseteq \{ \lambda \to b \mid b \in V \}$,
  - $*$ A_i *is a finite subset of* V^*,
  - $*$ I_i *and* O_i *are regular languages over* V,
  - $*$ $\alpha_i \in \{*, l, r\}$ *indicates the way the rules from* M_i *are applied: arbitrary in the word* ($*$), *at the left* (l) *or right* (r) *end of the word,*
- E *is a subset of* $\{1, 2, \ldots, n\} \times \{1, 2, \ldots, n\}$, *and*
- j *is a natural number such that* $1 \leq j \leq n$.

– *A configuration* C *of* $\mathcal{N}$ *is an* n*-tuple* $C = (C(1), C(2), \ldots, C(n))$ *where* $C(i)$ *is a subset of* V^* *for* $1 \leq i \leq n$.

– *Let* $C = (C(1), \ldots, C(n))$ *and* $C' = (C'(1), \ldots, C'(n))$ *be two configurations of* $\mathcal{N}$. *We say that* C *derives* C' *in one*

- *evolutionary step (written as* $C \Longrightarrow C'$*) if, for* $1 \leq i \leq n$, $C'(i)$ *consists of all words* $w \in C(i)$ *to which no rule of* M_i *is applicable and of all words* w *for which there are a word* $v \in C(i)$ *and a rule* $p \in M_i$ *such that* $v \Longrightarrow_{\alpha_i, p} w$ *holds,*
- *communication step (written as* $C \vdash C'$*) if, for* $1 \leq i \leq n$,

$$C'(i) = (C(i) \setminus O_i) \cup \bigcup_{(k,i) \in E} (C(k) \cap O_k \cap I_i).$$

The computation of $\mathcal{N}$ *is a sequence of configurations*

$$C_t = (C_t(1), C_t(2), \ldots, C_t(n)), \quad t \geq 0,$$

such that

- $C_0 = (A_1, A_2, \ldots, A_n)$,
- C_{2t} *derives* C_{2t+1} *in an evolutionary step:*$C_{2t} \Longrightarrow C_{2t+1}$ *(for all* $t \geq 0$*),*
- C_{2t+1} *derives* C_{2t+2} *in a communication step:* $C_{2t+1} \vdash C_{2t+2}$ *(for all* $t \geq 0$*).*

– *The language $L(\mathcal{N})$ generated by $\mathcal{N}$ is defined as*

$$L(\mathcal{N}) = \bigcup_{t\geq 0} C_t(j)$$

where the sequence of configurations $C_t = (C_t(1), \ldots, C_t(n))$, with $t \geq 0$, is a computation of $\mathcal{N}$.

Intuitively, a NEP as above is a graph consisting of n nodes (called processors) $N_1, N_2, \ldots, N_n$ and the set of edges given by E such that there is a directed edge from N_k to N_i if and only if $(k, i) \in E$. Any processor N_i consists of a set of evolutionary rules M_i, a set of words A_i, and two regular languages, namely an input filter I_i and an output filter O_i. We say that N_i is a substitution, deletion, or insertion node if $M_i \subseteq \{\, a \to b \mid a, b \in V \,\}$ or $M_i \subseteq \{\, a \to \lambda \mid a \in V \,\}$ or, respectively, $M_i \subseteq \{\, \lambda \to b \mid b \in V \,\}$, respectively. The input filter I_i and the output filter O_i define the words which are allowed to enter and to leave the node, respectively. With any node N_i and any time moment $t \geq 0$, we associate a set $C_t(i)$ of words (the words contained in the node at time t). Initially, N_i contains the words of A_i. In an evolutionary step, we derive from $C_t(i)$ all words applying rules from the set M_i. In a communication step, any processor N_i sends out all words $C_t(i) \cap O_i$ (which pass the output filter) to all processors to which a directed edge exists (the words from $C_t(i) \setminus O_i$ remain in the set associated with N_i) and, moreover, it receives from any processor N_k such that $(k, i) \in E$ all words sent by N_k and passing the input filter I_i of N_i, i. e., the processor N_i gets in addition all words of $C_t(k) \cap O_k \cap I_i$. We start with an evolutionary step and then communication and evolutionary steps are alternately performed. The language generated by the NEP consists of all words which appear in a distinguished node N_j (called output node) during the computation.

We say that a NEP $\mathcal{N} = (V, N_1, N_2, \ldots, N_n, E, j)$ is in weak normal form if the working mode of N_i $(1 \leq i \leq n)$ is $*$ and we say it is in normal form if it is in weak normal form and $E = \{(i, j) \mid 1 \leq i \neq j \leq n\}$ (i. e., it has a complete underlying graph). We call two networks equivalent to each other if they generate the same language.

For a family $X \subseteq REG$, we denote the family of languages generated by networks of evolutionary processors (in weak normal form and normal form, respectively), where all filters are of type X by $\mathcal{E}(X)$ ($\mathcal{E}_*(X)$ and $\mathcal{E}_N(X)$, respectively).

The following results are known (see, e. g., (Castellanos et al., 2003), (Dassow et al., 2011, Theorem 4)).

THEOREM 1. *i) $\mathcal{E}_N(REG) = RE$. ii) $\mathcal{E}_*(MIN_2) = RE$.* □

3. Simplifying the Filters

We start by showing that the second part of Theorem 1 is optimal in that sense that one state automata are not sufficient.

LEMMA 1. *The regular language $L = \{ wb \mid w \in \{a, b\}^* \}$ is not contained in $\mathcal{E}_*(MIN_1)$.*

Proof. Suppose $L \in \mathcal{E}_*(MIN_1)$. Then there is a NEP $\mathcal{N}$ which has only filters that belong to the class MIN_1 (i. e., each filter has the form W^* for some alphabet W or is the empty set) and which generates the language L. Since L is infinite and networks with only substitution and deletion nodes generate finite languages, $\mathcal{N}$ contains an insertion node. The number of a's is unbounded (for each natural number n, there is a word $w \in L$ with more than n occurrences of a). Hence, there are a natural number $s \geq 0$ and letters $x_0, x_1, \ldots, x_s$ with $x_s = a$ such that the network contains the rules $\lambda \to x_0$ and $x_i \to x_{i+1}$ for $0 \leq i \leq s-1$ and there is a word $w_1 a w_2 \in L$ which is derived from a word $v_1 v_2$ by applying these rules (possibly, not only these rules), starting with the insertion of x_0 between v_1 and v_2. But x_0 can also be inserted at the end of the word $v_1 v_2$. All words derived from $v_1 x_0 v_2$ are letter equivalent to those derived from $v_1 v_2 x_0$. Thus, if a word derived from $v_1 x_0 v_2$ can pass a filter then also a word that is derived from $v_1 v_2 x_0$ in the same manner can pass that filter. Hence, in the same way as $w_1 a w_2$ is derived and communicated to the output node, the word $w_1 w_2 a$ is derived and communicated to the output node. But $w_1 w_2 a \notin L$. Thus, the language L cannot be generated by a NEP where the filters belong to MIN_1. This implies $L \notin \mathcal{E}_*(MIN_1)$. As $\mathcal{E}_N(MIN_1)$ is included in $\mathcal{E}_*(MIN_1)$, it follows that $L \notin \mathcal{E}_N(MIN_1)$ as well. □

However, if we also allow the other two modes of derivation, then we can improve the bound given in the second part of Theorem 1 by proving that every recursively enumerable language can be generated by a network of evolutionary processors where each filter is accepted by a deterministic finite automaton with one state only.

THEOREM 2. *$\mathcal{E}(MIN_1) = RE$.*

Proof. Let L be a recursively enumerable language, G be a grammar in Geffert normal form generating the language $L(G) = L$ where the set of non-terminals is $N = \{ S, A, B, C \}$ with the axiom S, the set of terminal symbols is T, and the set of rules is P with the rules being $S \to v$ with $v \in (N \cup T)^*$ or $ABC \to \lambda$. We construct a NEP $\mathcal{N}$ that simulates the derivation process in G and, hence, generates the same language. The idea is to rotate the sentential form until the subword which has to be replaced is a suffix and then to delete and insert (if necessary) at the right end of the word.

Let $V = N \cup T$ and $\# \notin V$ be a symbol that marks the actual end of a word during the rotation. Let $U = V \cup \{\#\}$, $u = \#(U)$ be the number of the elements of U, $x_1, x_2, \ldots, x_u$ be the elements of U, and $U' = \{ x' \mid x \in U \}$. Furthermore, let p be the number of the rules of the form $S \to v$ and let these rules be $P_i = S \to a_{i,1}a_{i,2}\cdots a_{i,l_i}$ with $1 \leq i \leq p$, $l_i \geq 0$, and $a_{i,j} \in V$ for $1 \leq j \leq l_i$.

We construct subnetworks for rotating a word and for simulating the application of a rule. These subnetworks are connected by a special node (the 'master' N_0) that belongs to every subnetwork. The structure of the network can be seen in Figure 1.

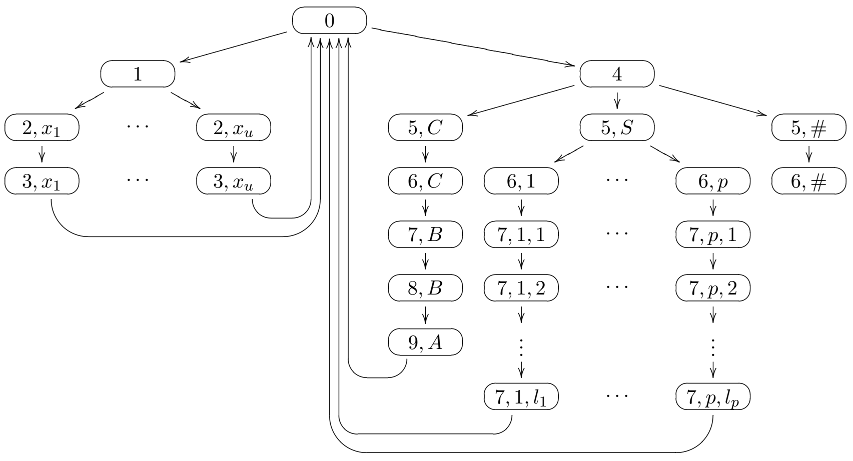

Figure 1. NEP for a Geffert normal form grammar

The master N_0 is defined by $A_0 = \{ S\# \}$, $M_0 = \emptyset$, $I_0 = U^*$, $O_0 = U^*$, and $\alpha_0 = *$. The work starts with the first sentential form of G where the end is marked by #. This node does nothing but in the next communication step the word will be sent to the nodes N_1 where the rotation will start and N_4 where the simulation of a rule will start.

The initial languages A_i of all other nodes N_i will be empty. If the mode α_i is not explicitly given then it is $*$.

The node N_1 is defined by $I_1 = U^*$, $M_1 = \{ x \to x' \mid x \in U \}$, $\alpha_1 = l$, and $O_1 = (U \cup U')^*$. This node marks the left-most symbol by changing it to a primed version. Then the word is sent to all nodes $N_{2,x}$ where $x \in U$ but only one of these nodes gives access to the word, namely the node $N_{2,x}$ where x is the symbol whose primed version stands at the beginning of the word. Thus, the node $N_{2,x}$ for $x \in U$ is defined by $I_{2,x} = (\{x'\} \cup U)^*$, $M_{2,x} = \{ x' \to \lambda \}$, and $O_{2,x} = U^*$. The primed x is deleted (which is allowed to occur everywhere but because of node N_1

it appears only in the left-most position). Then the word is sent to the corresponding node $N_{3,x}$ where x is inserted at the right end of the word (appended to the word). Hence, $N_{3,x}$ for $x \in U$ is defined by $I_{3,x} = U^*$, $M_{3,x} = \{ \lambda \to x \}$, $\alpha_{3,x} = r$, and $O_{3,x} = U^*$. Now, the word was rotated by one symbol and will be sent to the master.

The simulation of a rule will start in the node N_4. This node is defined by $I_4 = U^*$, $M_4 = \{ x \to x' \mid x \in U \}$, $\alpha_4 = r$, and $O_4 = (U \cup U')^*$. In this node, the right-most symbol will be changed to the primed version. If the last symbol was C and the rule $ABC \to \lambda$ exists in P then this rule can be started to be simulated (maybe C is not preceded by AB then the simulation does not work which will be noticed later). If the last symbol was S then one of the rules P_i $(1 \le i \le p)$ can be simulated. If the last symbol was # then this end marker should be removed to obtain a real sentential form of G (which is the end of the simulation of G). In all other cases, we do not need the word anymore.

We now construct subnetworks for these three cases.

If the rule $ABC \to \lambda$ exists in P then we define the following nodes: $N_{5,C}$ is defined by $I_{5,C} = (\{C'\} \cup U)^*$ (the primed symbol is allowed everywhere and arbitrarily often but N_4 ensures that there is exactly one occurrence and this is in the last position), $M_{5,C} = \{ C' \to \lambda \}$, and $O_{5,C} = U^*$. This node takes a word if the last symbol is C', it deletes this symbol and passes on the word to the next node $N_{6,C}$ which is responsible for the B. The node $N_{6,C}$ is defined by $I_{6,C} = U^*$, $M_{6,C} = \{ x \to x' \mid x \in U \}$, $\alpha_{6,C} = r$, and $O_{6,C} = (U \cup U')^*$. This node marks the last symbol and sends the word to node $N_{7,B}$ defined by $I_{7,B} = (\{B'\} \cup U)^*$, $M_{7,B} = \{ B' \to \lambda \}$, and $O_{7,B} = U^*$. If the last symbol of the word sent is B' then the simulation continues, otherwise the word is lost (the previously selected C does not belong to a subword ABC and therefore the rule cannot be applied). In $N_{7,B}$ the symbol B' is deleted and the word moves to the node $N_{8,B}$ defined by $I_{8,B} = U^*$, $M_{8,B} = \{ x \to x' \mid x \in U \}$, $\alpha_{8,B} = r$, and $O_{8,B} = (U \cup U')^*$. This node again marks the last symbol. If it is A then the simulation continues, otherwise the word is lost (the previously selected word BC does not belong to a subword ABC and therefore the rule cannot be applied). The word is sent to node $N_{9,A}$ defined by the sets $I_{9,A} = (\{A'\} \cup U)^*$, $M_{9,A} = \{ A' \to \lambda \}$, and $O_{9,A} = U^*$. There, A' is deleted. Then a word $u\#v$ has been derived by simulating the rule $ABC \to \lambda$ where vu is a sentential form of G. The word $u\#v$ is sent to the master node N_0.

For the context-free rules (the rules $S \to v$), we define the node $N_{5,S}$ by the sets $I_{5,S} = (\{S'\} \cup U)^*$, $M_{5,S} = \{ S' \to S_i \mid 1 \le i \le p \}$, and $O_{5,S} = (U \cup \{ S_i \mid 1 \le i \le p \})^*$. This node chooses a rule P_i that is simulated afterwards. The word obtained is sent to all nodes $N_{6,j}$ with $1 \le j \le p$ but it is accepted only by node $N_{6,i}$ which corresponds to

the rule selected.
For each rule P_i with $1 \leq i \leq p$, we define the following nodes: the node $N_{6,i}$ is defined by $I_{6,i} = (\{S_i\} \cup U)^*$, $M_{6,i} = \{\, S_i \to \lambda \,\}$, and $O_{6,i} = U^*$. This node deletes S_i (the symbol corresponding to the left hand side of the rule under consideration). If $P_i = S \to \lambda$ then this word is sent back to the master node, otherwise the rule is $S \to a_{i,1}a_{i,2}\cdots a_{i,l_i}$ for a natural number $l_i \geq 1$ and symbols $a_{i,j} \in V$ for $1 \leq j \leq l_i$. These symbols will now be appended to the word in a chain of nodes (one node for each symbol). For each number j with $1 \leq j \leq l_i$, we define the node $N_{7,i,j}$ by $I_{7,i,j} = U^*$, $M_{7,i,j} = \{\, \lambda \to a_{i,j} \,\}$, $\alpha_{7,i,j} = r$, and $O_{7,i,j} = U^*$. After the right hand side of the rule P_i is appended, the rule has been simulated, and the word is sent to the master node.

For the case that the last symbol in N_4 is the end marker #, we define the following nodes: $N_{5,\#}$ by $I_{5,\#} = (\{\#'\} \cup U)^*$, $M_{5,\#} = \{\, \#' \to \lambda \,\}$, $O_{5,\#} = U^*$ and $N_{6,\#}$ by $I_{6,\#} = T^*$, $M_{6,\#} = \emptyset$, $O_{6,\#} = T^*$. In $N_{5,\#}$, the end marker is deleted. The word obtained is a sentential form of G and it is sent to node $N_{6,\#}$ which serves as the output node and accepts the word only if it is a terminal word.

The NEP $\mathcal{N}$ is defined as $\mathcal{N} = (X, N_0, \ldots, N_{6,\#}, E, N_{6,\#})$ with the working alphabet $X = U \cup U' \cup \{\, S_i \mid 1 \leq i \leq p \,\}$, the nodes defined above, the set E of edges given in Figure 1, and the output node $N_{6,\#}$. From the explanations given along with the definitions of the nodes and edges, it follows that the network $\mathcal{N}$ generates the same language as the grammar G: $L(\mathcal{N}) = L$. Moreover, all filters are sets of the form Y^* for some alphabet Y. Hence, $RE \subseteq \mathcal{E}(MIN_1)$. With Theorem 1, the equality $\mathcal{E}(MIN_1) = RE$ follows. □

We note that in (Dassow and Truthe, 2011) the bound 2 was given for the number of states accepting the filter languages and that this bound cannot be improved. However, in that paper, for each regular language L used as a filter the authors count the number of states of a (completely defined) deterministic finite automaton accepting L but having as input alphabet the entire alphabet of the network. Here, when a regular language L is used as a filter we only count the states of a (completely defined) deterministic finite automata accepting L and having as input alphabet $alph(L)$. The transformation from an automata with input alphabet $alph(L)$ to one having as input alphabet the whole alphabet of the network often requires one additional state. For instance, in the previous proofs we have often filters equal to U^* and we say that they are accepted by automata with one state; if we would consider that this automata may have as input letters from $X \setminus U$, not only from U, then each of them would need an extra errorstate, in which the automaton moves as soon as it reads a letter from $X\ setminus U$.

Moreover, it is worth noting that Theorem 2 does not follow from the constructions in (Dassow and Truthe, 2011).

4. Transformations into Weak Normal Form

The results in this section show how a given NEP can be transformed into a NEP in weak normal form such that the sets of rules of all nodes are singletons.

LEMMA 2. *Let $k \geq 1$ and $k' = \max\{k, 2\}$. Further, let $\mathcal{N}$ be a NEP with filters from MIN_k. Then a NEP $\mathcal{N}' = (U, N'_1, N'_2, \ldots, N'_m, E', j')$ can be algorithmically constructed such that $L(\mathcal{N}') = L(\mathcal{N})$, all nodes N'_i, $1 \leq i \leq m$, have input and output filters from the class $MIN_{k'}$, $N'_{j'} = (\emptyset, \emptyset, I', O', *)$, and no edge is leaving $N'_{j'}$.*

Proof. Let $\mathcal{N} = (V, N_1, N_2, \ldots, N_n, E, j)$ be a network of evolutionary processors where the output node N_j has not the required property: $N_j \neq (\emptyset, \emptyset, I_j, O_j, *)$ for any sets I_j, O_j or there is an edge leaving node N_j. We define a new network $\mathcal{N}' = (V, N'_1, N'_2, \ldots, N'_{n+4}, E', n+4)$ by

$$\begin{aligned}
N'_i &= N_i \text{ for } 1 \leq i \leq n, \\
N'_i &= (M_i, \emptyset, I_i, O_i, \alpha_i) \text{ for } n+1 \leq i \leq n+4, \\
E' &= E \cup \{\, (i, n+1) \mid (i,j) \in E \,\} \cup \{\, (n+1, n+2), (n+1, n+4) \,\} \\
&\qquad \cup \{\, (n+2, n+3), (n+2, n+4), (n+3, n+2) \,\}
\end{aligned}$$

where

$$\begin{array}{llll}
M_{n+1} = \emptyset, & M_{n+2} = M_j, & M_{n+3} = \emptyset, & M_{n+4} = \emptyset, \\
A_{n+1} = A_j, & A_{n+2} = \emptyset, & A_{n+3} = \emptyset, & A_{n+4} = \emptyset, \\
I_{n+1} = I_j, & I_{n+2} = V^*, & I_{n+3} = V^* \setminus O_j, & I_{n+4} = V^*, \\
O_{n+1} = V^*, & O_{n+2} = V^*, & O_{n+3} = V^*, & O_{n+4} = V^*, \\
\alpha_{n+1} = *, & \alpha_{n+2} = \alpha_j, & \alpha_{n+3} = *, & \alpha_{n+4} = *.
\end{array}$$

The network is illustrated in Figure 2.

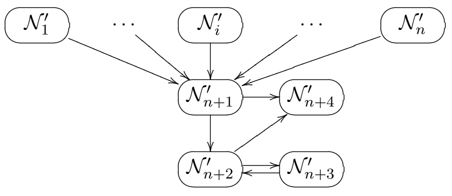

Figure 2. NEP for the proof of Lemma 2

The new output node $N'_{n+4} = (\emptyset, \emptyset, V^*, V^*)$, from which no edge leaves, satisfies the condition. We now show that $L(\mathcal{N}') = L(\mathcal{N})$.

The subnetwork consisting of $N'_1, N'_2, \ldots, N'_n$ is the same as $\mathcal{N}$. The initial sets of N'_j and N'_{n+1} as well as the input filters and incoming edges coincide. Hence, if a word w is in N_j at an even moment t, then w is also in this moment in node N'_j and N'_{n+1}. The word is then sent unchanged to the output node N'_{n+4}. Thus, $w \in L(\mathcal{N})$ and $w \in L(\mathcal{N}')$. Additionally, w is also sent to N'_{n+2} where the same rules as in N_j can be applied. Hence, if a word v is derived in N_j (and, hence, $v \in L(\mathcal{N})$) then v is derived in N'_{n+2} and will be sent to the output node in the next communication step, hence, $v \in L(\mathcal{N}')$. If the word v remains in N_j then a word $u \in L(\mathcal{N})$ will be derived from v in N_j. In $\mathcal{N}'$, the word v will also be sent to N'_{n+3} which takes the word and sends it back to N'_{n+2} where it will be derived to u which will be sent to the output node afterwards. Hence, as long as a word is modified in N_j, the same word is modified in N'_{n+2} with intermediate communication to N'_{n+3} and all these words also arrive in the output node. Thus, $L(\mathcal{N}) \subseteq L(\mathcal{N}')$.

Every word $w \in L(\mathcal{N}')$ came to node N'_{n+4} from node N'_{n+1} or N'_{n+2}. If it came from N'_{n+1} then the word was also in node N_j, hence, $w \in L(\mathcal{N})$. If it came from N'_{n+2} then it has been derived from a word v which came from N'_{n+1} or N'_{n+3}. If v came from N'_{n+1} then v was also in N_j and has derived w, hence, $w \in L(\mathcal{N})$. If v came from N'_{n+3} then v was previously in node N'_{n+2} and was derived from a word u. Furthermore, $v \notin O_j$. If u came from N'_{n+1} then u was also in N_j and has derived v which remained there and derived w, hence, $w \in L(\mathcal{N})$. If u came from N'_{n+3} then the argumentation can be repeated because for every word in u in N'_{n+2} there was a word $\tilde{u}$ in N'_{n+1} with $\tilde{u} \Longrightarrow^*_{M_j} u$ and all words during this derivation did not belong to O_j. Hence, $\tilde{u}$ was also in N_j where the same derivation of u took place. Thus, $L(\mathcal{N}') \subseteq L(\mathcal{N})$.

Since $L(\mathcal{N}') = L(\mathcal{N})$, the network $\mathcal{N}'$ has the required properties. □

After normalizing the output node, we continue with normalizing the remaining nodes.

LEMMA 3. *Let $k \geq 1$ and $\mathcal{N} = (V, N_1, N_2, \ldots, N_n, E, j)$ be a NEP with n nodes with filters from MIN$_k$ and $k' = \max\{k, 3\}$. Then an equivalent network $\mathcal{N}' = (U, N'_1, N'_2, \ldots, N'_m, E', j')$ in weak normal form can be algorithmically constructed such that all nodes N'_i for $1 \leq i \leq m$ have input and output filters in the class MIN$_{k'}$.*

Proof. We can assume without losing generality that the output node of $\mathcal{N}$ has no rules and axioms. We will show how we can construct, for each node x of the network, a subnetwork $s(x)$ that simulates its computation; altogether, these subnetworks will form the new network $\mathcal{N}'$. We denote by $n(s(x))$ the nodes of the subnetwork $s(x)$, and distinguish two nodes of this subnetwork, namely $i(s(x))$ and $o(s(x))$ that facilitate the communication with the subnetworks constructed for the other nodes of the original network (these two nodes can be seen as the entrance node, and, respectively, the exit node of the subnetwork). We also denote by $e(s(x))$ the set of edges of the subnetwork (that is, the edges connecting the nodes of the subnetwork). Note that all the processors of the new network work in arbitrary mode. Finally, let $V_\# = \{ \#_a \mid a \in V \}$ be a set of symbols with $V \cap V_\# = \emptyset$, and $U = V \cup V_\# \cup \{\#\}$.

First we approach the simple cases: nodes with no rules, nodes that have only substitutions of the type $a \to a$ with $a \in V$, and nodes where rules can be applied anywhere in the words. If x is such a node we will simply copy it in the second network with the only difference that the way it applies the rules is set now to arbitrary (i. e., $*$). Therefore, if x is associated with the processor (M, A, I, O, α) with $\alpha = *$ or $M \subseteq \{a \to a \mid a \in V\}$ we construct the subnetwork that has $n(s(x)) = \{x\}$, $i(s(x)) = o(s(x)) = x$, $e(s(x)) = \emptyset$ and the processor placed in the node is $(M, A, I, O, *)$.

Now, let us consider the case of left-insertion nodes. Assume that the processor in x is (M, A, I, O, l). Let $V_1 = \{ a \in V \mid \lambda \to a \in M \}$ and $V_2 = alph(O)$. For this node, we define the following subnetwork $s(x)$:

- $n(s(x)) = \{x_0, x_1, x_2, x_3\}$;
- $e(s(x)) = \{(x_0, x_1), (x_1, x_2), (x_1, x_3), (x_2, x_1)\}$;
- $i(s(x)) = x_0$ and $o(s(x)) = x_3$.

$$\begin{tikzcd}
\arrow[r, dotted, no head] & \boxed{x_0} \arrow[r] & \boxed{x_1} \arrow[r] \arrow[d, shift left] & \boxed{x_3} \arrow[r, dotted, no head] & {} \\
 & & \boxed{x_2} \arrow[u, shift left] & &
\end{tikzcd}$$

The processors in the nodes are:

- $x_0 = (\{\lambda \to \#\}, A, I, (V \cup \{\#\})^*, *)$;
- $x_1 = (\{ \# \to a \mid a \in V_1 \}, \emptyset, \{\#\}(V \cup \{\#\})^*, V^*, *)$;
- $x_2 = (\{\lambda \to \#\}, \emptyset, U^* \setminus O, (V_2 \cup \{\#\})^*, *)$ where $V_2 = alph(O)$;
- $x_3 = (\emptyset, \emptyset, O, V^*, *)$.

All the filters of this subnetwork are in MIN_k, given that I and O are in MIN_k, or in MIN_3 (where exactly 3 states are needed for a deterministic finite automaton accepting the input filter of node x_1 and one state is needed for the other filters that do not depend on I or O).

To see that the subnetwork $s(x)$, presented above, simulates the behaviour of the node x, let us assume that w is a word that is communicated to x in $\mathcal{N}$, and it was also communicated to x_0 in $\mathcal{N}'$. Clearly,

w enters x if and only if it enters x_0. The word becomes in the initial network $a_k a_{k-1} \ldots a_1 w$, with $a_1, a_2, \ldots, a_k \in V_1$ and $k \geq 1$ such that $a_\ell a_{\ell-1} \ldots a_1 w \notin O$ for all $\ell < k$ as well as $a_k a_{k-1} \ldots a_1 w \in O$ and the word exits the node or remains blocked in the node. In the subnetwork $s(x)$ it first enters the node x_0. In this node exactly one symbol # is inserted in w and the word is sent out; it goes to x_1, but only in the case when it has the form $\#w$. Here it becomes aw with $a \in V_1$ and leaves the node towards x_2 or x_3. It enters x_3 if and only if $aw \in O$ and then goes out of the network. Otherwise, it enters x_2 where a new # symbol is inserted; then it is sent back to x_1, if the word contains only symbols that may appear in the words of O and #, and the process described above is restarted. If a word does not belong to O and contains a symbol which is not in V_2, then it is trapped in node x forever and it is blocked in the node x_2 of the subnetwork $s(x)$ which means in both cases that the word is lost for the computation. Hence, the subnetwork $s(x)$ simulates correctly the node x.

A right-insertion node can be simulated similarly. We only change the input filter of node x_1 to $(V \cup \{\#\})^*\{\#\}$ (we check for the inserted symbol # at the end of a word rather than in the beginning). This filter can be accepted by a deterministic finite automaton with two states.

Next, we discuss the case of left-deletion nodes. Let (M, A, I, O, l) be the processor in node x. Also, let $V_1 = \{\, a \in V \mid a \to \lambda \in M \,\}$ and $V_2 = \mathit{alph}(O)$. For this node, we define the subnetwork $s(x)$:

- $n(s(x)) = \{x_0, x_1, x_2, x_3, x_4, x_5\}$;
- $e(s(x))$ as in the picture:
- $i(s(x)) = x_0$ and $o(s(x)) = x_5$;

$$\begin{tikzcd}
 & & x_2 \arrow[d, shift left] & x_3 \arrow[dl, shift left] & \\
{} \arrow[r, dash, dotted] & x_0 \arrow[r] \arrow[dr] & x_1 \arrow[u, shift left] \arrow[ur, shift left] \arrow[r] & x_5 \arrow[r, dash, dotted] & {} \\
 & & x_4 \arrow[ur] & &
\end{tikzcd}$$

The processors in the nodes are defined by:

- $x_0 = (\{\, a \to \#_a \mid a \in V_1 \,\}, A, I, (V \cup V_\#)^*, *)$;
- $x_1 = (\{\, \#_a \to \lambda \mid a \in V_1 \,\}, \emptyset, V_\# U^*, V^*, *)$;
- $x_2 = (\{\, a \to \#_a \mid a \in V_1 \,\}, \emptyset, U^* \setminus O, (V_2 \cup V_\#)^*, *)$;
- $x_3 = (\{\, a \to \#_a \mid a \in V_1 \,\}, \emptyset, U^* B U^*, (V \cup V_\#)^*, *)$
  where $B = V \setminus V_2$;
- $x_4 = (\{\, \#_a \to a \mid a \in V_1 \,\}, \emptyset, B' U^*, V^*, *)$, where $B' = V \setminus V_1$;
- $x_5 = (\emptyset, \emptyset, O, V^*, *)$.

All the filters of this subnetwork are in MIN_k, given that I and O are in MIN_k, or in MIN_3 (the input filters of nodes x_1, x_3, and x_4 as well as all output filters).

We show that the subnetwork $s(x)$ presented above simulates the behaviour of the node x. Let w be a word communicated to x in $\mathcal{N}$ which was also communicated to x_0 in the new network. This word enters x in $\mathcal{N}$ if and only if it enters x_0 in $\mathcal{N}'$. It is processed by x in the following manner: the processor deletes successively zero or more symbols from the beginning of the word according to the rules until it can leave the node. We will show that the subnetwork $s(x)$ implements the same behaviour. First, assume that w enters the node x_0; as in the previous case, two things may happen here to the word: it can be left unchanged (if no rule can be applied to it) or an a symbol for $a \in V_1$ can be replaced by $\#_a$ (by applying a rule $a \to \#_a$ with $a \in V_1$). All the obtained words leave this node. The words that start with a symbol that cannot be deleted by x may go to x_4; in this node, any $\#_a$ symbol contained in these words is restored to a, then they are sent to x_5 and finally exit the subnetwork if $w \in O$. The word that contains a $\#_a$ symbol on the left-most position goes to x_1. The words that contain a $\#_a$ symbol not on the left-most position but start with a symbol that can be deleted in x are lost. The computation continues as follows for the word beginning with $\#_a$ for some $a \in V_1$: in x_1 the $\#_a$ symbol is deleted and copies of the newly obtained word are sent to the nodes x_2, x_3, and x_5. The node x_5 will accept the word obtained from w after the deletion of the left-most symbol if this word is in O. If the word is not in O, it goes to x_2 and x_3. The node x_3 takes the word if it contains a symbol which is not in V_2.

In any of these nodes, a symbol a is substituted by a $\#_a$ symbol. A newly obtained word leaves x_2 only if all appearing letters belong to V_2 or are symbols of V marked for deletion; a new word leaves x_3 only if all appearing letters belong to V or are symbols of V marked for deletion. Then the processed word goes back to x_1 where the simulation of the deletion is repeated. Finally, the word will enter x_5 when it becomes part of O (if this eventually happens) and can leave the network. The above reasoning shows that the words obtained from w in $s(x)$ which can leave the subnetwork are exactly those that are obtained in the node x and leave it.

For the case of left-substitution nodes, we only change the set V_1 of all left hand side symbols to $V_1 = \{ a \in V \mid \exists c \in V : a \to c \in M \}$ and the rule set of x_1 to $\{ \#_a \to c \mid a \to c \in M \}$.

The right-deletion and right-substitution nodes can be simulated similarly. We only change the input filter of node x_1 to $U^* V_\#$ and the input filter of node x_4 to $U^* B'$ (we check for the inserted symbol $\#_a$

or the impossibility of applying a rule at the end of a word rather than in the beginning). All the filters, except for I and $U^* \setminus O$, can be accepted by deterministic finite automata with two states; the filters I and $U^* \setminus O$, respectively, can be accepted with as many states as I and O, respectively.

Finally, for given network $\mathcal{N} = (V, N_1, \ldots, N_n, E, j)$ we define the network $\mathcal{N}' = (V \cup V_\# \cup \{\#\}, N'_1, \ldots, N'_m, E', j')$ where

– $N'_1, \ldots, N'_m$ is an enumeration of the nodes from the set

$$\bigcup_{i \in \{1,\ldots,n\}} n(s(N_i));$$

– the output node of the network is the only node from the subnetwork $s(N_j)$ (which contains only a copy of N_j, since the original node had no rules);

– the set E' of edges is

$$\{\, (o(s(N_i)), i(s(N_k))) \mid (i,k) \in E \,\} \cup \bigcup_{i \in \{1,\ldots,n\}} e(s(N_i));$$

– the rules, axioms, and filters of the nodes are defined as above.

From the remarks made when we explained how each subnetwork works it follows that the network $\mathcal{N}'$ generates exactly $L(\mathcal{N})$. □

Obviously, the application of Lemma 3 to the networks constructed in Theorem 2 gives only the bound 3 for the number of states whereas Theorem 1 ii) gives the better bound 2. Nevertheless, Lemma 3 is of interest since we have to consider the form in which a recursively enumerable language is given; Lemma 3 requires a description by a network with evolutionary processors, whereas Theorem 1 ii) uses a description by a grammar in Kuroda normal form.

Now we normalize the number of rules present in each node.

LEMMA 4. *Let $k \geq 1$, $\mathcal{N} = (V, N_1, N_2, \ldots, N_n, E, j)$ be a NEP in weak normal form with filters from the class MIN_k, and $k' = \max\{k, 2\}$. Then an equivalent network $\mathcal{N}' = (V, N'_1, N'_2, \ldots, N'_m, E', j')$ in weak normal form can be algorithmically constructed such that each node $N'_i = (M'_i, A'_i, I'_i, O'_i, *)$, $1 \leq i \leq m$, has a singleton set M'_i and input and output filters from the class $MIN_{k'}$.*

Proof. Without loss of generality, we can assume that the output node of $\mathcal{N}$ has the form $N_j = (\emptyset, \emptyset, I_j, O_j, *)$. Those nodes in $\mathcal{N}$ that have at most one rule can be used for $\mathcal{N}'$ without change. The other nodes can be simulated by a network in which there is a node for each rule that takes a word if and only if the rule can be applied and a node for the case that no rule can be applied. As shown in the proof of Lemma 3, we need three more nodes that simulate the original output filter. We construct for each node x of the network $\mathcal{N}$ a subnetwork $s(x)$, contained in the new network $\mathcal{N}'$, that simulates the computation of the processor found in the node x. Again, we denote by $n(s(x))$ the set of the nodes of the subnetwork $s(x)$, by $i(s(x))$ and $o(s(x))$ the entrance and exit nodes, and by $e(s(x))$ the set of the edges of the subnetwork.

Let us assume that the node x is the processor $(M, A, I, O, *)$ with $M = \{r_1, \ldots, r_p\}$ where $p \geq 1$. If the node is a deletion or substitution node, any rule r_i has the form $a_i \to c_i$ $(1 \leq i \leq p)$ and we define $I_i = V^*\{a_i\}V^*$ for $1 \leq i \leq p$ and $I_0 = (V \setminus \{a_1, \ldots, a_p\})^*$. If the node is an insertion node we define $I_i = V^*$ for $1 \leq i \leq p$ and $I_0 = \emptyset$. Then we construct the subnetwork as follows:

– $n(s(x)) = \{x_0, x_0', x_1, x_2, x_3\} \cup \{x_1^r, \ldots, x_p^r\}$,
$$e(s(x)) = \{(x_0', x_3)\} \cup \bigcup_{k=0}^{2} (\{(x_k, x_0')\} \cup \bigcup_{i=1}^{p} \{(x_k, x_i^r), (x_i^r, x_{k+1})\})$$

– $i(s(x)) = x_0$ and $o(s(x)) = x_3$;

– $x_0 = (\emptyset, A, I, V^*, *)$;

– $x_i^r = (\{r_i\}, \emptyset, I_i, V^*, *)$ for $i \in \{1, \ldots, p\}$;

– $x_0' = (\emptyset, \emptyset, I_0, V^*, *)$;

– $x_1 = (\emptyset, \emptyset, V^* \setminus O, V_1^*, *)$, where $V_1 = alph(O)$;

– $x_2 = (\emptyset, \emptyset, (V \setminus B)^*BV^*, V^*, *)$, where $B = V \setminus alph(O)$;

– $x_3 = (\emptyset, \emptyset, O, V^*, *)$.

All the filters of this network are in MIN_k or in MIN_2 (we need deterministic finite automata with two states to accept the input filters of x_2 and x_i^r for $i \in \{1, \ldots, p\}$ and with one state for the other filters that do not depend on I or O).

To see that the subnetwork $s(x)$ defined above simulates the behaviour of the node x, let us assume that w is a word that was sent towards the node x in a communication step of $\mathcal{N}$, and the same word was also sent towards x_0 in $\mathcal{N}'$. The word w enters x if and only if it enters x_0. In x, a rule r_i is applied to w; this is simulated in $s(x)$ in the following steps: w goes from x_0 to x_i^r, where the rule r_i is applied to it. Back in $\mathcal{N}$, the word exits the node x if it is contained in O

or, otherwise, remains in the node and is further processed. In $\mathcal{N}'$ the word goes to x_3 if it is in O and leaves the subnetwork in the next communication step; if it is not in O, it goes to x_1 (and it is blocked there unless all its symbols are in $alph(O)$) and to x_2 (if it contains a symbol from $V \setminus alph(O)$). The word is returned by these nodes to the nodes x_ℓ^r for $\ell \in \{1, \ldots, p\}$ to be further processed. Also, there are words that enter x but are not processed in this node; in $\mathcal{N}'$, these strings enter in x_0, are sent to x_0', and further they are sent to x_3, and leave the network if they verify the output conditions. Hence, the subnetwork $s(x)$ behaves exactly like the node x.

If a node x has no rules, it is kept in exactly the same form in the new network (i. e., $n(s(x)) = \{x\}$, $i(s(x)) = o(s(x)) = x$, $e(s(x)) = \emptyset$ and the processor placed in the node remains unchanged).

To finish the construction of the network $\mathcal{N}'$ we set:

– $N_1', \ldots, N_m'$ is an enumeration of the nodes in the set
$$\bigcup_{i \in \{1,\ldots,n\}} n(s(N_i)).$$

– The output node of the network is the only node from the subnetwork $s(N_j)$ (which contains only a copy of N_j, since the original node had no rules).

– The set E' of edges equals
$$\{\, (o(s(N_i)), i(s(N_k))) \mid (i,k) \in E \,\} \cup \bigcup_{i \in \{1,\ldots,n\}} e(s(N_i)).$$

– The rules, axioms and filters of the nodes are defined as above.

From the explanations provided together with the definitions of the subnetworks follows that the network $\mathcal{N}'$ generates exactly $L(\mathcal{N})$. □

5. Transformation into Normal Form

Once we know how to transform the NEPs in order to have only processors containing at most one rule and where the rules can be applied at an arbitrary position, we can focus on normalizing the topology of the network, i. e., to get networks in normal form.

LEMMA 5. *Let $k \geq 1$, $\mathcal{N} = (V, N_1, N_2, \ldots, N_n, E, j)$ be a NEP in weak normal form with filters from MIN_k which has, for $1 \leq i \leq n$, nodes $N_i = (M_i, A_i, I_i, O_i, *)$ with $\#(M_i) \leq 1$ and $N_j = (\emptyset, \emptyset, I_j, O_j, *)$, and $k' = \max\{k, 2\}$. An equivalent NEP $\mathcal{N}' = (V', N'_1, N'_2, \ldots, N'_m, E', j')$ in normal form can be algorithmically constructed such that all filters of N'_i, $1 \leq i \leq m$, are from the class $MIN_{k'}$.*

Proof. The main idea of the proof is to construct a network $\mathcal{N}'$ that simulates the computation of $\mathcal{N}$ and implements the following idea: we always have in the words processed in the new network exactly one special symbol $\#_x$ (or primed versions of this symbol), where x is a node of $\mathcal{N}$, that indicates the processor of the initial network whose actions must be simulated at that point. Such a symbol $\#_x$ is inserted in the axioms corresponding to x, at the beginning of the computation; once such a symbol is inserted in the word, no other symbol of this type can be inserted. This special symbol is modified during the computation of the new network in such a manner that it gives, in parallel, all the possible paths that the computation may follow in the initial network. A problem occurs when we deal with the output node: we cannot allow the entrance of strings with #-symbols in this node. Therefore, the new network will work with copies of the working symbols from V, and once we obtain a copy of a word that would have been accepted in the output node of $\mathcal{N}$, we delete the #-symbol present in that word, we restore the rest of the symbols to the original version, and send them to the output node of $\mathcal{N}'$.

As in the previous proof, we will show how we can construct for each node x of the network $\mathcal{N}$ a subnetwork $s(x)$ contained in $\mathcal{N}'$ that simulates the computation of the processor in x. We denote by $n(s(x))$ the nodes of the subnetwork $s(x)$ but, in this case, we do not have to care about the edges or special nodes in the subnetwork since every node of the new network will communicate to all the others.

Assume that V° is an alphabet with copies of the symbols in V, i. e., it contains the symbol a° instead of a, for all $a \in V$. Moreover, for a word w, we denote by $\circ(w)$ the word obtained by replacing the symbols of V in w by their copies and leaving the others unchanged. Also, if L is a language, we denote by $\circ(L)$ the language obtained by replacing the symbols of V that appear in the words of L by their copies from V°.

For a language L and a symbol $x \notin alph(L)$, we define the language L_x as the set of all the words w' where w' was obtained by inserting in a word w from L several symbols x. If $L \in MIN_k$ then $L_x \in MIN_k$. The definition can be easily extended to finite sets of symbols: for a language L and the set of symbols $S = \{x_1, \ldots, x_p\}$, with $x_i \notin alph(L)$ for all $i \leq p$, we define the language L_S as the set of all the words w' where w' was obtained by inserting in a word w from L several symbols from the set $\{x_1, \ldots, x_p\}$. Again, if $L \in MIN_k$ then $L_S \in MIN_k$. Also, for a language L and a symbol $x \in alph(L)$, we define the language $L^{x'}$ as the set of all the words w' where w' was obtained by substituting in a word w from L several symbols x by x'. If $L \in MIN_k$ then $L^{x'} \in MIN_k$.

The alphabet of the network $\mathcal{N}'$ is defined as:

$$\begin{aligned} U \;=\; & V \cup V^\circ \cup \{\$\} \cup \{\, b' \mid b \in V \,\} \\ & \cup \{\#_o\} \cup \{\, \#_x, \#'_x, \#''_x \mid x \text{ is a node of } \mathcal{N} \,\}. \end{aligned}$$

For each node x of $\mathcal{N}$, we define two sets of symbols that will be useful in our construction:

$$\begin{aligned} S_x &= U \setminus \{\, \#_y, \#'_y, \#''_y \mid y \text{ is a node of } \mathcal{N},\ y \neq x \,\} \text{ and} \\ E_x &= \{\, \#_y \mid y \text{ is a node of } \mathcal{N},\ (x,y) \in E \,\}. \end{aligned}$$

Further, we define the rest of the network, following the main lines we mentioned before. We will split our discussion in four cases, according to the type of each node (with no rules, insertion, substitution, deletion).

Let us assume that the node x has the processor $(\emptyset, A, I, O, *)$; also, assume that x is not the output node. Then we set

- $n(s(x)) = \{x_0, x'_0\}$;
- $x_0 = (\{\, \#_x \to \#_y \mid (x,y) \in E \,\}, \emptyset, \circ(I_{\#_x}), \circ(O_{E_x}), *)$;
- $x'_0 = (\{\, \lambda \to \#_y \mid (x,y) \in E \,\}, \circ(A), \emptyset, \circ(O_{E_x}), *)$.

In such a subnetwork, we only change the symbol $\#_x$ into a new symbol $\#_y$ telling us that the word can now go towards the nodes of the subnetwork associated with y, and cannot enter the nodes of any other subnetwork. Also, at the beginning of the computation, the node x'_0 inserts $\#_y$ in the axioms, with the same effect as the above. In both nodes, the rest of the word is left unchanged, as it was also the case in the initial network, where the whole word stayed unchanged.

The case of the output node should be discussed separately. Let us assume that the node x has the processor $(\emptyset, A, I, O, *)$. Then we set

- $n(s(x)) = \{x_0, x_1, x_2, x_3, x_4\}$, and x_4 is the output node of $\mathcal{N}'$;
- $x_0 = (\{\#_x \to \#'_x, \#_x \to \#_o\}, \emptyset, \circ(I_{\#_x}), U^*\{\#'_x, \#_o\}U^*, *)$;
- $x_1 = (\{\, \#'_x \to \#_y \mid (x,y) \in E \,\}, \emptyset, I_{x_1}, \circ(O_{E_x}), *)$
  with $I_{x_1} = (U \setminus \{\#_o\})^*\{\#'_x\}(U \setminus \{\#_o\})^*$;
- $x_2 = (\{\, a^\circ \to a \mid a \in V \,\}, \emptyset, U^*\{\#_o\}U^*, V^*\#_oV^*, *)$;

– $x_3 = (\{\#_o \to \lambda\}, \emptyset, V^*\{\#_o\}V^*, V^*, *)$;

– $x_4 = (\emptyset, \emptyset, V^*, \emptyset, *)$.

The processing of this subnetwork follows the idea that each string must go to the other subnetwork, if it can pass the output filter, and it must be saved as part of the generated language. In the first case, in x_0 and x_1 we can change the symbol $\#_x$ into a new symbol $\#_y$ telling us that the word can now go towards the nodes of the subnetwork associated with y, if it can pass the output filter of x. In the second case, we change in x_0 the symbol $\#_x$ into $\#_o$, and send the string out. It can only enter x_2 where all the symbols from V° are replaced with their original versions from V. Further, the string can only enter x_3, where $\#_o$ is deleted. After the next communication step, the obtained string can only enter x_4, where we collect all the words in the generated language. All the strings that enter the output node in the original network are collected in x_4 and no other string can enter this node in $\mathcal{N}'$.

Assume that the node x has the processor $(\{\, \lambda \to a \,\}, A, I, O, *)$. Then we set:

– $n(s(x)) = \{x_0, x_0', x_1, x_2\}$;

– $x_0 = (\{\#_x \to \#_x'\}, \emptyset, \circ(I_{\#_x}), U^*\{\#_x'\}U^*, *)$;

– $x_0' = (\{\lambda \to \#_x'\}, \circ(A), \emptyset, U^*\{\#_x'\}U^*, *)$;

– $x_1 = (\{\lambda \to a'\}, \emptyset, U^*\{\#_x'\}U^*, \circ(O^{a'}_{\#_x'}), *)$;

– $x_2 = (M_{x_2}, \emptyset, S_x^*\{a'\}S_x^*, (U \setminus \{a', \#_x'\})^*, *)$
with $M_{x_2} = \{a' \to a^\circ\} \cup \{\, \#_x' \to \#_y \mid (x,y) \in E \,\}$.

In what follows, we explain how the subnetwork works. Let us assume that w is a word that was sent towards the node x in a communication step of $\mathcal{N}$, and the word $w_1\#_x w_2$, with $w_1w_2 = \circ(w)$, was communicated in the complete network $\mathcal{N}'$. If the string w can pass the input filter of x then $w_1\#_x w_2$ can also enter x_0 (and no other node); the converse also holds. In the node x, we either obtain from w a word $w' \in O$ by inserting k $a-$symbols into w (as long as we have inserted less than k symbols we get a word that is not contained in O) or the string is blocked in this node. In the network $\mathcal{N}'$ the string is processed as follows. In x_0, it becomes $w_1\#_x' w_2$ and is sent out. As we will see after the whole network is defined, it can only enter x_1 (no other node allows it in according to the input filters); here it becomes $w_1'\#_x' w_2'$, with $w_1'w_2' = \circ(w')$, and may leave the node. However, only the strings from $\circ(O_{\#_x'})$ can leave the node x_1. Therefore, if the word leaves the node x_1 it can only go to node x_2 where we obtain all the strings $w_1'\#_y w_2'$, for a node y such that $(x, y) \in E$. Each such string leaves the node and can go to the nodes of the subnetworks associated with the node y. Also, x_0' sends, at the beginning of the computation, all the strings $w_1\#_x' w_2$, where $w_1 w_2 \in \circ(A)$, to node x_1 in order to be processed as in the node x. Clearly, $s(x)$ simulates correctly the computation done by x.

Now we move on to substitution nodes. Let us assume that the node x has the processor $(\{a \to b\}, A, I, O, *)$. Then we set

- $n(s(x)) = \{x_0, x_0', x_1, x_1', x_2\}$;
- $x_0 = (\{\#_x \to \#_x'\}, \emptyset, \circ(I_{\#_x}), U^*\{\#_x'\}U^*, *)$;
- $x_0' = (\{\lambda \to \#_x'\}, \circ(A), \emptyset, U^*\{\#_x'\}U^*, *)$;
- $x_1 = (\{a^\circ \to b'\}, \emptyset, U^*\{\#_x'\}U^*, \circ(O^{b'}_{\#_x'}), *)$ if $b \in alph(O)$;
- $x_1 = (\{a^\circ \to b'\}, \emptyset, U^*\{\#_x'\}U^*, \emptyset, *)$ if $b \notin alph(O)$;
- $x_1' = (\{\#_x' \to \#_x''\}, \emptyset, (U \setminus \{b', a^\circ\})^*\{\#_x'\}(U \setminus \{b', a^\circ\})^*, O_{\#_x''}, *)$;
- $x_2 = (M, \emptyset, S_x^*\{b', \#_x''\}S_x^*, (U \setminus \{b', \#_x', \#_x''\})^*, *)$
  with $M = \{b' \to b^\circ\} \cup \{\, \#_x' \to \#_y, \#_x'' \to \#_y \mid (x, y) \in E \,\}$.

In this case, the simulation works as follows. Let us assume that w is a word that was sent towards the node x in a communication step of $\mathcal{N}$, and the word $w_1\#_x w_2$, with $w_1 w_2 = \circ(w)$, was communicated in the complete network $\mathcal{N}'$. The string w can pass the input filters of x if and only if $w_1\#_x w_2$ can also enter x_0. In the node x we either obtain from w a word $w' \in O$, by substituting several a symbols of w by b symbols (at least one if w contains an a, otherwise none), or the string is blocked in this node. In the network $\mathcal{N}'$ the string is processed as follows. In x_0 it becomes $w_1\#_x' w_2$ and is sent out. It can only enter x_1 or x_1', but this only if it has no a° symbol. In x_1 it becomes $w_1'\#_x' w_2'$, such that if we replace the b' symbols from $w_1' w_2'$ by b° we get $\circ(w')$, and leaves the node; in x_1' it becomes $w_1\#_x'' w_2$. It is not hard to see that all the other strings obtained in x_1 are blocked, lost, or go to x_1' (where copies of them were already processed), because the only words that leave this node are from $\circ(O^{b'}_{\#_x'})$, and they can enter only x_2, where a b' is needed, or x_1', if they contained no a° from the beginning. Also, all the strings obtained in x_1' that are not part of $\circ(O_{\#_x''})$ are blocked. Therefore, if the word leaves the node x_1 or x_1' it can only go to node x_2 where we obtain all the strings $w_1''\#_y w_2''$ with $w_1'' w_2'' = \circ(w')$ and $(x, y) \in E$. Each such string leaves the node and can go to the nodes of the subnetworks associated with the node y. Also, x_0' sends at the beginning of the computation all the strings $w_1\#_x'w_2$ where $w_1w_2 \in \circ(A)$ to node x_1 in order to process them as in the original network. Hence, $s(x)$ simulates correctly the computation done by x.

Finally, we present the simulation of the deletion nodes. Let us assume that the node x has the processor $(\{a \to \lambda\}, A, I, O, *)$. Then we set:

– $n(s(x)) = \{x_0, x_0', x_1, x_1', x_2, x_3, x_4\}$;

– $x_0 = (\{\#_x \to \#_x'\}, \emptyset, \circ(I_{\#_x}), U^*\{\#_x'\}U^*, *)$;

– $x_0' = (\{\lambda \to \#_x'\}, \circ(A), \emptyset, U^*\{\#_x'\}U^*, *)$;

– $x_1 = (\{a^\circ \to \$\}, \emptyset, U^*\{\#_x'\}U^*, \circ(O_{\$,\#_x'}), *)$;

– $x_1' = (\{\#_x' \to \#_x''\}, \emptyset, (U \setminus \{b', a^\circ\})^*\{\#_x'\}(U \setminus \{b', a^\circ\})^*, \circ(O_{\#_x''}), *)$;

– $x_2 = (\{\#_x' \to \#_x''\}, \emptyset, S_x^*\{\$\}S_x^*, O_{\{\$,\#_x''\}}, *)$;

– $x_3 = (\{\$ \to \lambda\}, \emptyset, (S_x \setminus \{\#_x'\})^*\{\$\}(S_x \setminus \{\#_x'\})^*, \circ(O_{\#_x''}), *)$;

– $x_4 = (M, \emptyset, (S_x \setminus \{\$\})^*\{\#_x''\}(S_x \setminus \{\$\})^*, (U \setminus \{\#_x''\})^*, *)$ with $M = \{ \#_x'' \to \#_y, \#_x' \to \#_y \mid (x, y) \in E \}$.

The simulation works pretty similar to the substitution case. Let us assume that w is a word that was sent towards the node x in a communication step of $\mathcal{N}$, and the word $w_1\#_xw_2$, with $w_1w_2 = \circ(w)$, was communicated in the complete network $\mathcal{N}'$. The string w can pass the input filters of x if and only if $w_1\#_xw_2$ can also enter x_0. In the node x we either obtain from w a word $w' \in O$, by deleting several a symbols (at least one if w contains an a, otherwise none), or the string is blocked in this node. In the network $\mathcal{N}'$ the string is processed as follows. In x_0, it becomes $w_1\#_x'w_2$ and is sent out. It can only enter x_1 or x_1', but this only if it has no a° symbol. In x_1 it becomes $w_1'\#_x'w_2'$, such that $\circ(w')$ can be obtained from $w_1'w_2'$ by deleting the \$ symbols, and leaves the node; in x_1', it becomes $w_1\#_x''w_2$. It is not hard to see that all the other strings obtained in x_1 are blocked, lost, or go to x_1' (where copies of them were already processed), because the only words that leave this node are from $\circ(O_{\{\$,\#_x'\}})$ and they can enter only x_2 where a \$ is needed or x_1' if they contained no a° from the beginning. Also, all the strings obtained in x_1' that are not part of $\circ(O_{\#_x''})$ are blocked; therefore, if the word leaves the node x_1' it can only go to node x_4 where we obtain all the strings $w_1''\#_yw_2''$ with $w_1''w_2'' = \circ(w')$ and $(x, y) \in E$. If a string leaves x_1 and goes to x_2 (thus, the string has at least one \$ symbol), the $\#_x'$ contained in it is transformed into $\#_x''$ and the string can only go now to x_3. In this node, all the \$ symbols are deleted, and the string can go now to x_4 where it is transformed in the same way as the strings that went there from x_1'. All the strings obtained in x_4 leave the node and can go to the nodes of the subnetworks associated with the node y. Also, x_0' sends at the beginning of the computation all the strings $w_1\#_x' w_2$ with $w_1 w_2 \in \circ(A)$ to node x_1 in order to process them as in the original network. Hence, $s(x)$ simulates correctly the computation done by x.

To completely define the subnetwork $\mathcal{N}'$ just take $N_1', \ldots, N_m'$ as an enumeration of the nodes of all the subnetworks defined above. The new network has $E' = \{1, \ldots, m\} \times \{1, \ldots, m\}$. From the way the subnetworks work, we can state now that the following two statements are equivalent:

– w is a word that was sent towards the node x in a communication step of $\mathcal{N}$ and was processed by this node to obtain w', which was further communicated to node y;

– the word $w_1\#_x w_2$, with $w_1 w_2 = \circ(w)$, was communicated in the network $\mathcal{N}'$ and entered the node x_0 from the subnetwork $s(x)$ and was processed by the network $\mathcal{N}'$ (more precisely by the nodes from $s(x)$) to obtain $w_1'\#_y w_2'$, with $w_1' w_2' = \circ(w')$.

Also, by the description of the subnetwork corresponding to the output node it follows that w is generated by $\mathcal{N}$ if and only if it is generated by $\mathcal{N}'$. □

The Lemmas 3, 4, and 5 yield the following result.

THEOREM 3. *For a network of evolutionary processors $\mathcal{N}$ with filters from the class MIN_k, a network $\mathcal{N}'$ in normal form can be algorithmically constructed such that $L(\mathcal{N}) = L(\mathcal{N}')$ and all filters of $\mathcal{N}'$ are from the class $MIN_{k'}$, where $k' = \max\{k, 3\}$.* □

Let L be a recursively enumerable language. By Theorem 1, we have $L = L(\mathcal{N})$ for a network of evolutionary processors in weak normal form, where all filters are in the class MIN_2. If we now apply the Lemmas 4 and 5, then we obtain a network in normal form where all filters are in MIN_2. This result can be reformulated as follows.

THEOREM 4. *$\mathcal{E}_N(MIN_2) = RE$.* □

This result shows that the bound for the number of states to accept the filter languages is not increased if we go from networks in weak normal form to networks in normal form.

Since $\mathcal{E}_N(MIN_1)$ is properly contained in RE (by Lemma 1), the result in Theorem 4 is optimal.

Acknowledgements

Florin Manea's work was partially supported by the Alexander von Humboldt Foundation through a Research Fellowship at the Otto-von-Guericke University Magdeburg, Germany, between July 2009 and June 2011 and a subsequent Return Fellowship at the University of Bucharest, Romania, between July 2011 and October 2011. His work is currently supported by *DFG - Deutsche Forschungsgemeinschaft* through the grant 582014.

References

Alhazov, A., E. Csuhaj-Varjú, C. Martín-Vide, and Y. Rogozhin: 2009a, 'On the size of computationally complete hybrid networks of evolutionary processors'. *Theoretical Computer Science* **410**, 3188–3197.

Alhazov, A., J. Dassow, C. Martín-Vide, Y. Rogozhin, and B. Truthe: 2009b, 'On Networks of Evolutionary Processors with Nodes of Two Types'. *Fundamenta Informaticae* **91**, 1–15.

Castellanos, J., C. Martín-Vide, V. Mitrana, and J. M. Sempere: 2001, 'Solving NP-Complete Problems With Networks of Evolutionary Processors'. In: *Proc. IWANN 2001*, Vol. 2084 of *LNCS*. pp. 621–628, Springer-Verlag, Berlin.

Castellanos, J., C. Martín-Vide, V. Mitrana, and J. M. Sempere: 2003, 'Networks of Evolutionary Processors'. *Acta Informatica* **39**(6–7), 517–529.

Csuhaj-Varjú, E. and V. Mitrana: 2000, 'Evolutionary Systems: A Language Generating Device Inspired by Evolving Communities of Cells'. *Acta Informatica* **36**(11), 913–926.

Csuhaj-Varjú, E. and A. Salomaa: 1997, 'Networks of Parallel Language Processors'. In: *New Trends in Formal Languages – Control, Cooperation, and Combinatorics*, Vol. 1218 of *LNCS*. pp. 299–318, Springer-Verlag, Berlin.

Dassow, J., F. Manea, and B. Truthe: 2011, 'Networks of Evolutionary Processors with Subregular Filters'. In: *Proc. LATA 2011*, Vol. 6638 of *LNCS*. pp. 262–273, Springer-Verlag, Berlin.

Dassow, J. and B. Truthe: 2011, 'On networks of evolutionary processors with filters accepted by two-state-automata'. *Fundamenta Informaticae* **112**(2–3), 157–170.

Errico, L. D. and C. Jesshope: 1994, 'Towards a New Architecture for Symbolic Processing'. In: *AIICSR'94: Proceedings of the sixth international conference on Artificial intelligence and information-control systems of robots*. River Edge, NJ, USA, pp. 31–40, World Scientific Publishing Co., Inc.

Fahlman, S. E., G. E. Hinton, and T. J. Sejnowski: 1983, 'Massively Parallel Architectures for AI: NETL, Thistle, and Boltzmann Machines'. In: *Proc. AAAI 1983*. pp. 109–113.

Geffert, V.: 1991, 'Normal forms for phrase-structure grammars'. *RAIRO – Theoretical Informatics and Applications* **25**, 473–496.

Hillis, W. D.: 1986, *The Connection Machine.* Cambridge, MA, USA: MIT Press.

Martín-Vide, C. and V. Mitrana: 2005, 'Networks of Evolutionary Processors: Results and Perspectives'. In: *Molecular Computational Models: Unconventional Approaches.* pp. 78–114.

Păun, G.: 2000, 'Computing with Membranes'. *J. Comput. Syst. Sci.* **61**(1), 108–143.

Păun, G. and L. Sântean: 1989, 'Parallel Communicating Grammar Systems: The Regular Case'. *Annals of University of Bucharest, Ser. Matematica-Informatica* **38**, 55–63.

Rozenberg, G. and A. Salomaa: 1997, *Handbook of Formal Languages.* Springer-Verlag, Berlin.

Sankoff, D., G. Leduc, N. Antoine, B. Paquin, F. Lang, and R. Cedergren: 1992, 'Gene Order Comparisons for Phylogenetic Inference: Evolution of the Mitochondrial Genome'. *Proceedings of the National Academy of Sciences of the United States of America* **89**(14), 6575–6579.