Theoretical Computer Science 456 (2012) 18–29

Contents lists available at SciVerse ScienceDirect

Theoretical Computer Science

journal homepage: www.elsevier.com/locate/tcs

Accepting Networks of Genetic Processors are computationally complete ☆

Marcelino Campos, José M. Sempere *

Departamento de Sistemas Informáticos y Computación, Universidad Politécnica de Valencia, Camino de Vera s/n, 46022 Valencia, Spain

ARTICLE INFO

Article history:
Received 19 September 2011
Received in revised form 7 May 2012
Accepted 25 June 2012
Communicated by L. Kari

Keywords:
Parallel Genetic Algorithms
Networks of biologically-inspired processors
Computational completeness

ABSTRACT

We propose a computational model that is inspired by genetic operations over strings such as mutation and crossover. The model, *Accepting Network of Genetic Processors*, is highly related to previously proposed ones such as *Networks of Evolutionary Processors* and *Networks of Splicing Processors*. These models are complete computational models inspired by DNA evolution and recombination. Here, we prove that the proposed model is computationally complete (it is equivalent to the Turing machine). Hence, it can accept any recursively enumerable language. In addition, we relate the proposed model with (parallel) Genetic Algorithms or Evolutionary Programs and we set these techniques as decision problem solvers.

© 2012 Elsevier B.V. All rights reserved.

1. Introduction

In the last few years, there has been an increasing and renewed interest in looking to biological nature in order to propose new (complete) computational models. There were two milestones in this research area which is part of what is called *natural computing* [14]: first, the experiment performed by Adleman [1], in which he implemented an algorithm using only DNA strands and enzymes to solve a combinatorial problem; second, the model proposed by Păun [22], which was inspired by the membrane structure of the living cell and the interchange of molecules and energy (performed inside it) through different membranes.

Recently, there have been new proposals that take the genetic information and genome evolution inside the cell as a source of inspiration. Castellanos et al. proposed *Networks of Evolutionary Processors* (NEP) as a computational model inspired by point mutations and evolutive selection on the DNA genome [6,7]. Some years before, T. Head introduced the *splicing* operation in the so-called H systems [12]. The *Network of Splicing Processors* (NSP) proposed by Manea et al. [17] substitutes point mutation operations by splicing operations over strings in the NEP model. Both models, NEP and NSP, have been proved to be complete models of computation. Therefore, they are equivalent to Turing machines in their computational power. Furthermore, they have been used to solve NP-complete problems in polynomial time with a constant number of processors [6,15–17].

In this work, we introduce a variant of the above models in which we replace point mutation (insertion, substitution, or deletion) and splicing operations (with nonempty contexts) by classical mutation (only substitution) and crossover (splicing with empty context) operations over strings. In the framework of genetic algorithms and evolutionary computation, our proposal can be considered as a finite set of genetic algorithms running in parallel. These models have been studied in the

☆ Work partially supported by the Spanish Ministry of Science and Innovation under coordinated research project TIN2011-28260-C03-00 and research project TIN2011-28260-C03-01.

* Corresponding author. Tel.: +34 963877353; fax: +34 963877359.
E-mail addresses: mcampos@dsic.upv.es (M. Campos), jsempere@dsic.upv.es (J.M. Sempere).

0304-3975/$ – see front matter © 2012 Elsevier B.V. All rights reserved.
doi:10.1016/j.tcs.2012.06.028

recent times in order to increase the efficiency in solving optimization problems [2,3,5]. Parallel genetic algorithms hold different populations that evolve independently and provide mechanisms to communicate their populations (the *migration phenomena*). We show how the proposed model of *Networks of Genetic Processors* satisfies both requirements and can be considered as a suitable theoretical model to study the computational power of Parallel Genetic Algorithms as a decision problem solver.

The structure of this work is as follows: first, we introduce basic concepts on formal language theory and computability that we will need in the sequel. Then, we formulate the model of *Networks of Genetic Processors* (NGP) with two different variants such as the acceptor and the generator case. We prove that NGPs are computationally complete for the accepting case, and we study the computational time complexity with respect to the $\mathcal{NP}$ complexity class. Then, we formalize the proposed model as Parallel Genetic Algorithms. Finally, we summarize the differences between our proposal and previously referred ones, and we outline our future research on this topic.

2. Basic concepts and notation

In the following, we will introduce basic concepts about the Turing machine and formal language theory from [13] and about computational complexity from [11].

An *alphabet* is a finite set of elements named *symbols*. A *string* is any ordered finite sequence of symbols. The empty string is denoted by ε and is defined as the string with no symbols. Given a string w, the length of the string is the number of symbols that it contains, and it will be denoted by $|w|$ (note that $|\epsilon| = 0$). The infinite set of all the strings defined over a given alphabet V will be denoted by V^*. Given the alphabet V, the set V^+ is defined as $V^+ = V^* - \{\varepsilon\}$. Given the string $x \in V^*$, we denote the minimal subset $W \subseteq V$ such that $x \in W^*$ by $alph(x)$. Given the string $x \in V^*$, we denote the set of segments of x by $seg(x)$, and it is defined as the set $\{\beta \in V^* : x = \alpha\beta\gamma$ with $\alpha, \gamma \in V^*\}$. Obviously, given any string $x \in V^*$, the set $alph(x)$ is a subset of $seg(x)$. A language defined over an alphabet V is a subset of strings of V^*.

A *deterministic Turing machine* is defined by the tuple $M = (Q, \Sigma, \Gamma, \delta, q_0, B, Q_f)$, where Q is a finite set of states, Σ and Γ are the input and the tape alphabets with $\Sigma \subset \Gamma$, $q_0 \in Q$ is an initial state, B is an special *blank* symbol from $\Gamma - \Sigma$, $Q_f \subseteq Q$ is a set of final states, and $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$ is a (possibly partially defined) transition function. The machine has a potentially infinite tape that is divided into cells, a finite state control that stores a state from Q, and a tape head to access the cells of the tape. We consider that the tape has a left bound while the infinite space grows to the right. Every cell of the tape holds a symbol from Γ. Initially, an input string x from Σ^* is loaded in the tape by introducing every symbol of x in a tape cell. The loading starts from the leftmost cell of the tape. The rest of the cells to the right of the input string hold the special blank symbol B. The tape head points to the first symbol of x and the finite control stores the initial state q_0. An *instantaneous description* of the Turing machine M is defined by the string $\alpha q\beta$, where $\alpha\beta$ is the content of the tape from the leftmost cell to the rightmost nonblank symbol or the symbol to the left of the head (whichever is rightmost), q is the current state of the finite control, and the tape head is assumed to be scanning the leftmost symbol of β. Observe that the initial instantaneous description q_0x means that we have loading the string x in the tape and the machine can start to make movements.

We define a movement of M as follows: Let $x_1x_2 \cdots x_{i-1}qx_i \cdots x_n$ be an instantaneous description of M. Let us suppose that $\delta(q, x_i) = (p, Y, L)$ where if $i - 1 = n$, then $x_i = B$. If $i = 1$, then the machine halts (there is no next instantaneous description) due to the fact that the machine tries to move to the left end of the tape (here, L means left movement). If $i > 1$, we write

$$x_1x_2 \cdots x_{i-1}qx_i \cdots x_n \underset{M}{\mapsto} x_1x_2 \cdots x_{i-2}px_{i-1}Yx_{i+1} \cdots x_n.$$

Alternatively, let us suppose that $\delta(q, x_i) = (p, Y, R)$. Then, we write

$$x_1x_2 \cdots x_{i-1}qx_i \cdots x_n \underset{M}{\mapsto} x_1x_2 \cdots x_{i-1}Ypx_{i+1} \cdots x_n.$$

Observe that $\underset{M}{\mapsto}$ is a relation between instantaneous descriptions of the Turing machine M. We denote the reflexive and transitive closure of $\underset{M}{\mapsto}$ by $\overset{*}{\underset{M}{\mapsto}}$. Hence, $\alpha_1q\beta_1 \overset{*}{\underset{M}{\mapsto}} \alpha_2p\beta_2$ denotes that the machine has made a finite number of movements (possibly including zero movements) to obtain the right instantaneous description from the left one.

There exist two situations that make the machine halt: first, the instantaneous description cannot be followed by a subsequent one because the movement function is not defined with the current scanned symbol and the state of the finite control; second, the machine tries to move to the left of the left end of the tape. We do not define movements from final states. The halting situation is denoted by the symbol $\downarrow$ as follows: $\alpha_1q\beta_1 \overset{*}{\underset{M\downarrow}{\mapsto}} \alpha_2p\beta_2$ means that the machine M changes the instantaneous description $\alpha_1q\beta_1$ to the description $\alpha_2p\beta_2$ and straight afterwards it halts. The language accepted by a Turing machine M is defined to be the set $L(M) = \{w \in \Sigma^* : q_0w \overset{*}{\underset{M\downarrow}{\mapsto}} \alpha q\beta$ with $q \in Q_f\}$. It is widely known that the family of languages accepted by deterministic Turing machines is the family of *recursively enumerable* languages which will be denoted by $\mathcal{RE}$.

A *nondeterministic Turing machine* is defined by the tuple $M = (Q, \Sigma, \Gamma, \delta, q_0, B, Q_f)$ where every component is defined as in the deterministic case with the exception of the δ function, which is defined as $\delta : Q \times \Gamma \to \mathcal{P}(Q \times \Gamma \times \{L, R\})$[1] and is a (possibly partially defined) transition function. The transition function $\delta(q, a) = \{(q_1, a_1, z_1), \ldots, (q_p, a_p, z_p)\}$ where $q_i \in Q$, $a_i \in \Gamma$ and $z_i \in \{L, R\}$ $1 \le i \le p$ has the following meaning: if the tape head is scanning the symbol a and the finite control is in state q, the machine nondeterministically selects a movement (q_j, a_j, z_j), so the tape head substitutes a by a_j, the finite control changes from q to q_j and the tape head moves to the z_j direction (left or right). Observe that in the same situation the machine could select a different tuple and the result could be different. The halting criterion is the same as in the deterministic case. The machine accepts an input string w if a sequence of movements such that $q_0 w \overset{*}{\underset{M\downarrow}{\mapsto}} \alpha q \beta$ with $q \in F$ exists.

We say that a computational model is *complete* if it can accept or generate any language in $\mathcal{RE}$. Alternatively, we can say that the model has the computational power of the Turing machine or the model is able to simulate any arbitrary Turing machine.

In the following, we introduce some basic concepts of computational complexity theory from [11]. Let M be a deterministic Turing machine; if for every input string w of length n, M makes at most $T(n)$ movements before halting, we say that M is a $T_M(n)$ *deterministically time-bounded* Turing machine or of *deterministic time complexity* $T_M(n)$. The language accepted by M is said to be of deterministic time complexity $T_M(n)$. In the case that M be non deterministic, then it is said that M is $T_M(n)$ *nondeterministically time-bound* if for every input string $w \in L(M)$ the machine has a minimum sequence of $T(n)$ movements to accept w. In this case, we say that the language accepted by M is of *nondeterministic time complexity* $T_M(n)$. We can define larger classes of languages depending on the time complexity function $T(n)$ as follows:

- $DTIME(T(n))$ is the class of languages accepted by deterministic Turing machines with deterministic time complexity $T(n)$.
- $NTIME(T(n))$ is the class of languages accepted by nondeterministic Turing machines with nondeterministic time complexity $T(n)$.

If we fix a collection of integer functions, then we can define different time complexity classes of languages. In this case, for a given collection of integer functions $\mathcal{C}$, we have:

- $DTIME(\mathcal{C}) = \bigcup_{f \in \mathcal{C}} DTIME(f)$
- $NTIME(\mathcal{C}) = \bigcup_{f \in \mathcal{C}} NTIME(f)$.

Let *poly* be the collection of all integer polynomial functions with nonnegative coefficients. Then, $\mathcal{P} = DTIME(poly)$ and $\mathcal{NP} = NTIME(poly)$.

A *decision problem* is a (mathematically defined) problem where the answer is the affirmation or negation of a predicate over the parameters of the problem. Any decision problem D defines a formal language L_D which contains the encoded parameter instances such that the answer is affirmative. Formally, a decision problem X is a pair (I_X, θ_X) such that I_X is a language over a finite alphabet (whose elements are called *instances*) and θ_X is a total Boolean function over I_X.

Therefore, the computational complexity of a decision problem can be transferred to the computational complexity of its associated formal language. We will come back to this approach when we study the relation between Parallel Genetic Algorithms and the computational model that we propose in the following section.

3. Networks of Genetic Processors

In the following, we define the *Networks of Genetic Processors*. Some basic concepts come from previous works on NEPs [6,7] and NSPs [16,17], while some others are referred to classical genetic algorithms or evolutionary programs [19].

Given the alphabet V, a *mutation rule* $a \to b$, with $a, b \in V$, can be applied over the string xay to produce the new string xby (observe that a mutation rule can be viewed as a substitution rule introduced in the NEP [7]).

A *crossover operation* is an operation over strings defined as follows: Let x and y be two strings, then $x \bowtie y = \{x_1y_2, y_1x_2 : x = x_1x_2 \text{ and } y = y_1y_2\}$. Observe that $x, y \in x \bowtie y$ given that we can take ϵ to be a part of x or y. The operation can be extended over languages as $L_1 \bowtie L_2 = \bigcup_{x \in L_1, y \in L_2} x \bowtie y$. Obviously, for any language L, $L \bowtie L$ is well defined. Observe that the crossover operation can be considered as a splicing operation over strings where the contexts of the strings are empty [23].

Let P and F be two disjoint subsets of an alphabet V, and let $w \in V^*$. We define the predicates $\varphi^{(1)}$ and $\varphi^{(2)}$ as follows:

1. $\varphi^{(1)}(w, P, F) \equiv (P \subseteq alph(w)) \wedge (F \cap alph(w) = \emptyset)$ (*strong predicate*)
2. $\varphi^{(2)}(w, P, F) \equiv (alph(w) \cap P \neq \emptyset) \wedge (F \cap alph(w) = \emptyset)$ (*weak predicate*).

[1] The set $\mathcal{P}(A)$ is the set of all the subsets of A.

We can extend the previous predicates to act over segments instead of symbols. Let P and F be two disjoint sets of finite strings over V, and let $w \in V^*$. We extend the predicates $\varphi^{(1)}$ and $\varphi^{(2)}$ as follows:

1. $\varphi^{(1)}(w, P, F) \equiv (P \subseteq seg(w)) \wedge (F \cap seg(w) = \emptyset)$ (*strong predicate*)
2. $\varphi^{(2)}(w, P, F) \equiv (seg(w) \cap P \neq \emptyset) \wedge (F \cap seg(w) = \emptyset)$ (*weak predicate*).

In the following, we work with this extension over segments instead of symbols. We can use single symbols that depend on the definition of P and F. The construction of these predicates is based on random-context conditions that are defined by the sets P (*permitting contexts*) and F (*forbidding contexts*). Let V be an alphabet and $L \subseteq V^*$, and let $\beta \in \{(1), (2)\}$; we define $\varphi^{\beta}(L, P, F) = \{w \in L : \varphi^{\beta}(w; P, F)\}$.

Now, we define a *genetic processor*, which can be viewed as a simple abstract machine that is capable of applying mutation or crossover rules over a multiset of strings.

Definition 1. Let V be an alphabet. A genetic processor N over V is defined by the tuple $(M_R, A, PI, FI, PO, FO, \alpha, \beta)$, where:

- M_R is a finite set of mutation rules over V
- A is a multiset of strings over V with a finite support and an arbitrary large number of copies of every string.
- $PI, FI \subseteq V^*$ are finite sets with the input permitting/forbidding contexts
- $PO, FO \subseteq V^*$ are finite sets with the output permitting/forbidding contexts
- $\alpha \in \{1, 2\}$ defines the function mode with the following values
  - If $\alpha = 1$ the processor applies mutation rules
  - If $\alpha = 2$ the processor applies crossover rules and $M_R = \emptyset$
- $\beta \in \{(1), (2)\}$ defines the type of the input/output filters of the processor. More precisely, for any word $w \in V^*$, we define an input filter $\rho(w) = \varphi^{\beta}(w, PI, FI)$ and an output filter $\tau(w) = \varphi^{\beta}(w, PO, FO)$. That is, $\rho(w)$ (resp. $\tau(w)$) indicates whether or not the word w passes the input (resp. the output) filter of the processor. We can extend the filters to act over languages. Thus, $\rho(L)$ (resp. $\tau(L)$) is the set of words of L that can pass the input (resp. output) filter of the processor.

Definition 2. A Network of Genetic Processors (NGP) is defined by the tuple $\Pi = (V, N_1, N_2, \ldots, N_n, G, \mathcal{N})$, where V is an alphabet, $G = (X_G, E_G)$ is a graph, N_i $(1 \leq i \leq n)$ is a genetic processor over V, and $\mathcal{N} : X_G \to \{N_1, N_2, \ldots, N_n\}$ is a mapping that associates the genetic processor N_i to the node $i \in X_G$.

We distinguish two types of Networks of Genetic Processors: The *accepting* one and the *generating* one. In the accepting case, the network will be denoted by ANGP and it has two distinguished processors, the *input* and the *output* processors, N_{input} and N_{output}, respectively. A *configuration* of an ANGP $\Pi = (V, N_1, N_2, \ldots, N_n, G, \mathcal{N})$ is defined by the tuple $C = (L_1, L_2, \ldots, L_n)$, where L_i is a multiset of strings defined over V for all $1 \leq i \leq n$. A configuration represents the multisets of strings that are present in any processor at a given moment (remember that every string appears in an arbitrarily large number of copies). The initial configuration of the network is $C_0 = (A_1, A_2, \ldots, A_n)$. Observe that since the input string w is allocated in the input node, $A_{input} = \{w\}$, while the output node is empty, so $A_{output} = \emptyset$.

Every copy of any string in L_i can be changed by applying a *genetic step* in accordance with the mutation or crossover rules associated with the processor N_i. Formally, we say that the configuration $C_1 = (L_1, L_2, \ldots, L_n)$ directly changes into the configuration $C_2 = (L'_1, L'_2, \ldots, L'_n)$ by a genetic step, written as $C_1 \Rightarrow C_2$, if L'_i is the multiset of strings obtained by applying the mutation or crossover rules of N_i to the strings in L_i. An arbitrarily large number of copies of each string is available in each node. Therefore, after a genetic step, one gets an arbitrarily large number of copies of any string, which can be obtained by using all possible mutation or crossover rules associated with that node. By definition, if L_i is empty for some $1 \leq i \leq n$, then L'_i is empty as well. In a *communication step*, each processor N_i sends all copies of the strings to all the processors connected to N_i according to G, provided that the strings are able to pass the output filter of N_i. In addition, it receives all the copies of the strings sent by the processors connected to N_i according to G, provided 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 $C \vdash C'$, iff

$$C'(x) = (C(x) - \tau_x(C(x))) \cup \bigcup_{\{x,y\} \in E_G} (\tau_y(C(y)) \cap \rho_x(C(y))) \quad \text{for all } x \in X_G$$

Let $\Pi = (V, N_1, N_2, \ldots, N_n, G, \mathcal{N})$ be an ANGP. A *computation sequence* in Π is a sequence of configurations $C_0, C_1, \ldots,$ where C_0 is the initial configuration of Π, $C_{2i} \Rightarrow C_{2i+1}$, and $C_{2i+1} \vdash C_{2i+2}$ for all $i \geq 0$. This sequence must be maximal (no other sequence follows from the previous one).

We consider that a sequence of configurations is finite whenever at least one of the following two conditions holds:

1. The output node contains at least one string. That is, if N_k is the designated output node, then $L_k \neq \emptyset$. In this case, we say that the network accepts the input string.
2. In a genetic step, the operations cannot be applied (the strings at every processor do not change) and no string is received or transmitted in the next communication step. After two consecutive genetic or communication steps, the configuration of the network does not change.

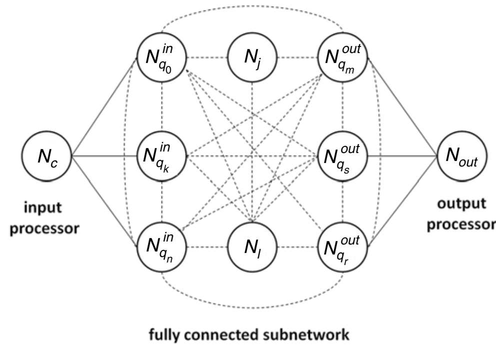

Fig. 1. The proposed ANGP to simulate an arbitrary deterministic Turing machine to process an arbitrary input string. This topology is denoted by the graph $\hat{K}$.

The language accepted by an Accepting Network of Genetic Processors is the set of input strings such as the network halts with at least one string in the output processor. Observe that, from the definitions given above, any ANGP is a deterministic device and we can predict the network behavior from a given input configuration.

Since the NGP can operate as a generating device, we have Generating Networks of Genetic Processors (GNGP). The main differences with respect to the accepting case are the following:

1. There is no defined input processor.
2. If the output node contains any string, then the network does not necessarily halt.

Observe that, in this case, the output processor collects a possibly infinite language and it constitutes its output language. In the following, we work with accepting networks of genetic processors.

4. Accepting Networks of Genetic Processors are computationally complete

In this section, we prove that every recursively enumerable language can be accepted by an ANGP.

Theorem 1. *Accepting Networks of Genetic Processors are computationally complete.*

Proof. The proof will be based on the simulation of any arbitrary deterministic Turing machine during the computation of any input string. We prove that whenever M halts in an accepting state, the ANGP computes a finite sequence with at least one string in the output processor. Conversely, if the Turing machine rejects the input string or it does not halt, then the ANGP does not accept the corresponding input string.

Let $M = (\Sigma, \Gamma, Q, \delta, q_0, B, Q_f)$ be an arbitrary Turing machine. We consider an instantaneous description of M in the form xq_iay, where $x, y \in \Gamma^*$, $a \in \Gamma$ and $q_i \in Q$. We define the alphabets $\Gamma' = \{a' : a \in \Gamma\}$ and $\bar{\Gamma} = \{\bar{a} : a \in \Gamma\}$.

First, the network encodes the initial instantaneous description q_0w as $q_0\bar{\$}wF$. We define the processor N_c to encode the input string as follows:

$$N_c = (\emptyset, \{w, q_0\bar{\$}, F\}, \emptyset, \emptyset, \{q_0\bar{\$}wF\}, \{aq_0, Fa : a \in (\Sigma \cup \{q_0, F, \bar{\$}\})\}, 2, (2)),$$

where w is the input string considered in the proof. Observe that the processor N_c operates only with crossover operations between w, $q_0\bar{\$}$ and F with $q_0\bar{\$}wF \in (q_0\bar{\$} \bowtie w) \bowtie F$. The output filters (here, segment filters) ensure that the output string will have the form $q_0\bar{\$}wF$ (permitted segments) and no symbol is to the right of the F mark or to the left of the q_0 mark (forbidden segments). In addition, no string can enter into this processor given that the permitted input filter is empty and the processor works in a weak manner.

The output processor is denoted by N_{out} and is defined as follows:

$$N_{out} = (\emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset, 2, (1))$$

Observe that N_{out} admits any string. We prove that this processor receives a string iff the Turing machine halts in a final state.

In the following, we take $x, y \in \Gamma^*$, $a, b, c \in \Gamma$ and $q_i, q_j \in Q$. We propose an ANGP with an underlying fully connected graph that is connected to the input node N_c and the output node N_{out}. The topology of the network is shown in Fig. 1 and is denoted by the graph $\hat{K}$.

The network architecture is described as follows: The *input processor* holds the input string to be computed; it transforms the initial string into an initial instantaneous description according to the encoding scheme that we have defined above. The *output processor* receives a string whenever the input string is accepted by the Turing machine. Finally, the fully connected subnetwork is composed by several processors: the processors in the form N_q^{in} play the role of the states before applying a transition step in the Turing machine; the processors in the form N_q^{out} play the role of the final states of the Turing machine;

and, finally, the rest of processors, such as N_j or N_l are defined to carry out the simulation of a transition step in the Turing machine. A complete description of these processors is made below.

The network is defined by the tuple $R = (V, N_c, N_1, N_2, \ldots, N_n, N_{out}, \hat{K}, f)$, where $V = \Gamma \cup \Gamma' \cup \bar{\Gamma} \cup Q \cup \{F, \#, \$, \bar{\$}\}$, with $F, \#, \$, \bar{\$} \notin \Gamma$, and f is the correspondence from vertexes to processors shown in Fig. 1. The processors are defined as follows:

1. For every state $q \in Q$

$$N_q^{in} = (M_q^{in}, \emptyset, \{q, \bar{\$}\}, \{\#\}, \emptyset, \{j', \bar{j} : j \in \Gamma\} \cup \{\bar{\$}\}, 1, (1)), \text{ where}$$
$$M_q^{in} = \{k' \to k : k \in \Gamma\} \cup \{\bar{k} \to k : k \in \Gamma\} \cup \{\bar{\$} \to \$\}$$

2. For every state $q \in Q_f$

$$N_q^{out} = (\emptyset, \emptyset, \{q, \$\}, \{\#, \$F\}, \emptyset, \emptyset, 1, (1))$$

3. $N_B = (\emptyset, \{\#BF\}, \{\$F, \#BF\}, \emptyset, V, \emptyset, 2, (2))$
4. $N_{B2} = (\emptyset, \{\#BF\}, \{\#BF\}, \{a\#, Fa : a \in V\}, \emptyset, \emptyset, 1, (2))$
5. For every state $q_i \in Q$ and every symbol $a \in \Gamma$ such that $\delta(q_i, a) = (q_j, b, R)$

$$N_{q_iaR} = (\{q_i \to q_j, \$ \to b', a \to \bar{\$}\}, \emptyset, \{q_i, \$a\}, \{\#, \$F\}, \{q_j, b'\bar{\$}\}, \{c\bar{\$} : c \in \Gamma\} \cup \{q_j\bar{\$}, \bar{\$}\bar{\$}\}, 1, (1))$$

6. For every state $q_i \in Q$ and every symbol $a \in \Gamma$ such that $\delta(q_i, a) = (q_j, b, L)$ and for every symbol $c \in \Gamma$

$$N_{q_iacL1} = (\{\$ \to \bar{c}, q_i \to q_j, a \to b'\}, \emptyset, \{q_i, \$a\}, \{\#, \$F\}, \{q_j, c\bar{c}b'\}, \emptyset, 1, (1))$$

7. For every state $q_i \in Q$ and every symbol $a \in \Gamma$ such that $\delta(q_i, a) = (q_j, b, L)$, and for every $c \in \Gamma$

$$N_{q_iacL2} = (\{c \to \bar{\$}\}, \emptyset, \{q_j, c\bar{c}b'\}, \{\$, \#\} \cup \{kb', k'b' : k \in \Gamma \cup Q, k' \in \Gamma'\}, \{q_j, \bar{\$}\bar{c}b'\}, \{\bar{\$}k : k \in \Gamma\} \cup \{\bar{\$}F\}, 1, (1))$$

We explain the role of every processor that we have defined. The network strings encode the instantaneous descriptions of the Turing machine during the computing time. Processors N_q^{in} hold the encoded instantaneous descriptions after the simulation of any movement in the Turing machine. Processors N_q^{out} receive the encoded instantaneous descriptions with final states. Processors N_B and N_{B2} are used to add a blank symbol before the F symbol. This happens when the tape head of the Turing machine visits new cells by moving the tape head to the right. Processors N_{q_iaR} simulate a movement of the Turing machine which moves the tape head to the right. Finally, processors N_{q_iacL1} and N_{q_iacL2} are used to simulate a movement of the Turing machine that moves the tape head to the left. Here, we need to consider all the possible symbols to the left of the tape head and the simulation is more complex than the movements to the right.

Now, we explain how the proposed ANGP works: Initially, $N_{q_0}^{in}$ receives the encoded initial instantaneous description of the Turing machine from processor N_c. The remaining processors N_q^{in} will not admit any string with a state that is different from q. We have explained above how the processor N_c encodes the input string w as the string $q_0\bar{\$}wF$. Each movement of the Turing machine is simulated by sending the instantaneous description to the corresponding processor that simulates the movement (observe that we have defined one processor for every movement that moves the tape head to the right, and a couple of processors for every movement that moves the tape head to the left). The processors simulate the movement and return the new instantaneous description to N_q^{in}. The encoded instantaneous description $q_ix\$F$ means that the tape head of the Turing machine has moved to the rightmost cell and it contains a blank symbol. In this case, the encoded string is sent to N_B, and this processor returns the string $q_ix\$BF$ to the rest of the processors. Finally, whenever strings in the form $q_ix\$yF$ are obtained and $q_i \in Q_f$, these strings are sent to $N_{q_i}^{out}$, then they are sent from $N_{q_i}^{out}$ to N_{out} and the network halts in an acceptance mode.

Observe that only the processor $N_{q_i}^{in}$ admits the strings with the $\bar{\$}$ and the q_i symbols. The remaining processors N_q^{in} will not admit any string with a state that is different from q and the remaining processors will not admit strings with the symbol $\bar{\$}$ (observe that some of them require the strings to have the segment $\$a$). Therefore, the operativity of the network implies that the processors N_q^{in} distribute the information over the complete network and they receive the results from the remaining processors. The symbol $\bar{\$}$ is used to ensure that only the processors N_q^{in} can receive an encoded instantaneous description after simulating a movement in the Turing machine.

The formal proof that the proposed ANGP simulates the Turing machine is made by induction over the number of movements that the Turing machine carries out. The key idea is that, at the beginning and the end of any movement of the Turing machine, the resulting encoded instantaneous description will be in one of the N_q^{in} processors. Formally, we prove the following predicate:

$$\text{If } q_0w \overset{*}{\underset{M}{\mapsto}} \alpha q\beta, \quad \text{then } (\exists k \geq 0)[C_0 \Rightarrow C_1 \vdash C_2 \cdots \vdash C_k \text{ with } q\alpha\$\beta F \in C_k(N_q^{in})]$$

Induction base

The base case of the induction is the initial configuration of the network that represents the initial instantaneous description of the Turing machine $q_0w = \alpha q\beta$. If w is the input string, then $N_{q_0}^{in}$ holds the string $q_0\$wF$ that is received from processor N_c after a communication step.

Induction hypothesis

Let us suppose that, if $q_0 w \overset{*}{\underset{M}{\mapsto}} \alpha q \beta$ in, at most, j steps, then

$$(\exists k \geq 0) \quad [C_0 \Rightarrow C_1 \vdash C_2 \cdots \vdash C_k \text{ with } q\alpha\$\beta F \in C_k(N_q^{in})]$$

Induction step

For the inductive step we will consider that $q_0 w \overset{*}{\underset{M}{\mapsto}} \alpha_1 q_i \beta_1 \underset{M}{\mapsto} \alpha q \beta$. We must analyze all the movements that allow the transition from $\alpha_1 q_1 \beta_1$ to $\alpha q \beta$ depending on the symbol being scanned, the state of the finite control, and the tape head movement (to the left or to the right). The change from $q_0 w$ to $\alpha_1 q_i \beta_1$ will be made in, at most, j movements, and, by our induction hypothesis, $q_i \alpha_1 \$ \beta_1 F \in C_k(N_{q_i}^{in})$.

First, let us suppose that the encoded instantaneous description $q_i x\$F$ is obtained in $N_{q_i}^{in}$. In this case, the network transforms $q_i x\$F$ into $q_i x\$BF$ in order to add a new blank symbol before applying a new movement in the next step. Only the processor N_B can receive the string $q_i x\$F$ given that the rest of the processors cannot receive any string with the segment $\$F$. Therefore, $q_i x\$F$ is received by N_B and the crossover operation with $\#BF$ is applied. It is easy to see that $q_i x\$BF$ is obtained after one genetic step in N_B. In the next communication step, all the strings in the processor N_B are sent out (it includes the string $\#BF$). The string $q_i x\$BF$ passes the input filter of the corresponding processor to apply the movement defined by $\delta(q_i, B)$. Observe that, in the nodes N_q^{in}, the string will not be admitted given that the $\bar{\$}$ symbol is required. At the same time, processor N_{B2} sends out the string $\#BF$ which will be only admitted in processor N_B. The pair of processors N_B and N_{B2} will operate in a synchronized mode to ensure that the string $\#BF$ is always present in processor N_B.

Now that we have shown how the network adds a new blank symbol, we analyze every movement as follows:

Case 1. Let us suppose that $N_{q_i}^{in}$ holds the string $q_i x\$ayF$ (which corresponds to the instantaneous description $\alpha_1 q_i \beta_1$ with $\alpha_1 = x$ and $\beta_1 = ay$) and the network must simulate the movement $\delta(q_i, a) = (q_j, b, R)$. The only node which can receive this string is $N_{q_i aR}$ given that its input filter accepts a sequence with the segments q_i and $\$a$. In $N_{q_i aR}$, the mutation rule $q_i \to q_j$ is applied to change the state, and the mutation rules $\$ \to b'$ and $a \to \bar{\$}$ are applied to simulate the tape head movement and writing. The rule $a \to \bar{\$}$ can be applied over all the symbols a of the strings, but only the string $q_j x b' \bar{\$} yF$ can pass the output filter with $PO = \{q_j, b'\bar{\$}\}$ and $FO = \{c\bar{\$} : c \in \Gamma\} \cup \{q_j\bar{\$}, \bar{\$}\bar{\$}\}$. This string is sent out, and it passes the input filter of the node $N_{q_j}^{in}$ since it permits the segments q_j and $\bar{\$}$. Then, in processor $N_{q_j}^{in}$, the rules $b' \to b$ and $\bar{\$} \to \$$ are applied and the result is the sequence $q_j x b\$ yF$. If we apply the rule $\delta(q_i, a) = (q_j, b, R)$ to the description $x q_i a y$, the result is $x b q_j y$, and $q_j x b \$ y F$ is obtained in $N_{q_j}^{in}$.

Case 2. Let us suppose that $N_{q_i}^{in}$ holds the string $q_i xc\$ayF$ (which corresponds to the instantaneous description $\alpha_1 q_i \beta_1$ with $\alpha_1 = xc$ and $\beta_1 = ay$) and the network must simulate the rule $\delta(q_i, a) = (q_j, b, L)$. Only the processors $N_{q_i acL1}$ can receive the string due to their permitted input filter with the segments q_i and $\$a$. In $N_{q_i acL1}$, the mutation rules $\$ \to \bar{c}$, $a \to b'$ and $q_i \to q_j$ are applied, but only the string $q_j x c \bar{c} b' y F$ passes the output filter with $PO = \{q_j, c\bar{c}b'\}$. Observe that the string $q_j x c \bar{c} b' y F$ does not contain any symbol \$ and only the processor $N_{q_i acL2}$ admits this string. The application of the rule $c \to \bar{\$}$ and the definition of the forbidden output filter ensures that the only string that can leave the processor has a segment $\bar{\$}\bar{c}b'$. Therefore, the string $q_j x \bar{\$} \bar{c} b' y F$ leaves the processor and it is only admitted in the processor $N_{q_j}^{in}$ due to the current state and the $\bar{\$}$ symbol. In the processor $N_{q_j}^{in}$, the rules $b' \to b$, $\bar{c} \to c$, and $\bar{\$} \to \$$ are applied and the result is the string $q_j x\$cbyF$, which corresponds to the instantaneous description $x q_j c b y$.

Case 3. Let us suppose that $N_{q_i}^{in}$ holds the string $q_i\$ayF$ and the network must simulate the rule $\delta(q_i, a) = (q_j, b, L)$. In this case, the Turing machine halts and it rejects the input string given that the tape head cannot move to the left of the first cell of the tape. Here, only the processors $N_{q_i acL1}$ can receive the string and the rule $\$ \to \bar{c}$ is applied. Given that the \$ symbol appears just to the right of the q_i symbol, the mutated string will have the form $q_j \bar{c} a y F$. The new strings cannot pass the output filter due to the permitted segments $c\bar{c}$ and they will remain in this node. Given that no new strings are communicated, the mutations over the rest of the strings at the rest of the processors will be finite. The only processors that are still active are N_B and N_{B2}. They will interchange the string $\#BF$, so the network will repeat two consecutive configurations and it will halt in a nonacceptance mode.

We now consider the acceptance situation in the Turing machine. Let us suppose that $N_{q_i}^{in}$ holds the string $q_i x\$yF$ with $q_i \in Q_f$. In this case, the Turing machine halts and it accepts the input string. The network sends the string to the processors N_q^{out}. The processor $N_{q_i}^{out}$ admits the string, and then it sends the string to the processor N_{out}. Then the network halts and it accepts the input string.

In this reasoning, we have excluded the case when $\epsilon \in L(M)$. Here, the initial instantaneous description of the Turing machine is q_0, with $q_0 \in Q_f$, and the corresponding encoded string is $q_0\bar{\$}F$. The processor $N_{q_0}^{in}$ receives the encoded string $q_0\bar{\$}F$ and mutates $\bar{\$}$ by \$. Then the string $q_0\$F$ is sent to N_B which adds the blank symbol, and then it is admitted in the processor $N_{q_0}^{out}$. Therefore, the empty string is accepted by the network.

Finally, if the Turing machine has no defined movement for a given state and tape symbol, then there is no processor to receive the encoded instantaneous description and the only processors that are still active are N_B and N_{B2}. They interchange

the string $\#BF$, so the network repeats two consecutive configurations and it halts in a nonacceptance mode. It can be observed that, if the Turing machine performs an infinite computation, then the network also performs an infinite computation, and the input string will never be accepted. □

In summary, we have proposed an ANGP that works for a given input string w and a given Turing machine M. Here, the number of processors depends on the size of the next movement function of M. Therefore, it is bounded by $2 \cdot |Q| + |Q| \cdot |\Gamma| + 2 \cdot |Q| \cdot |\Gamma|^2 + 4$. Observe that this is an upper bound in the number of processors given that for any symbol and state there will be only one processor (if the movement is to the right) or $2 \cdot |\Gamma|$ processors (if the movement is to the left). Both cases cannot occur simultaneously given that the Turing machine is deterministic.

An alternative proposal is based on an ANGP which can accept any encoded instance of the input word for the Turing machine. In this case, the processor N_c would be removed and the processor $N_{q_0}^{in}$ will be the input one that receives the encoded string $q_0\$wF$.

The nondeterministic case

We have shown how an ANGP simulates a deterministic Turing machine; hence, the model is proved to be computationally complete. Now, we show a direct simulation of nondeterministic Turing machines by ANGPs. Although this result does not add anything new about computational completeness, it is important to relate complexity results.

Theorem 2. *Every nondeterministic Turing machine can be simulated by an ANGP.*

Proof. The nondeterministic Turing machine differs from the deterministic one in the definition of the next movement function. Therefore, $\delta(q, a) = \{(q_1, a_1, z_1), \ldots, (q_p, a_p, z_p)\}$. In the deterministic case we have defined a processor for every value of the next movement function. In this case, we will define a processor for every choice of the next movement function. The network topology is the same as in the deterministic case. Here, processors N_c and N_{out} are defined as in the deterministic case, and a complete definition of the rest of processors follows:

1. For every state $q \in Q$

$$N_q^{in} = (M_q^{in}, \emptyset, \{q, \bar{\$}\}, \{\#\}, \emptyset, \{j', \bar{j} : j \in \Gamma\} \cup \{\bar{\$}\}, 1, (1)), \text{ where}$$

$$M_q^{in} = \{k' \to k : k \in \Gamma\} \cup \{\bar{k} \to k : k \in \Gamma\} \cup \{\bar{\$} \to \$\}$$

2. For every state $q \in Q_f$

$$N_q^{out} = (\emptyset, \emptyset, \{q, \$\}, \{\#, \$F\}, \emptyset, \emptyset, 1, (1))$$

3. $N_B = (\emptyset, \{\#BF\}, \{\$F, \#BF\}, \emptyset, V, \emptyset, 2, (2))$
4. $N_{B2} = (\emptyset, \{\#BF\}, \{\#BF\}, \{a\#, Fa : a \in V\}, \emptyset, \emptyset, 1, (2))$
5. For every pair of states $q_i, q_j \in Q$ and every pair of symbols $a, b \in \Gamma$ such that $(q_j, b, R) \in \delta(q_i, a)$

$$N_{q_i a q_j b R} = (\{q_i \to q_j, \$ \to b', a \to \bar{\$}\}, \emptyset, \{q_i, \$a\}, \{\#, \$F\}, \{q_j, b'\bar{\$}\}, \{c\bar{\$} : c \in \Gamma\} \cup \{q_j\bar{\$}, \bar{\$}\bar{\$}\}, 1, (1))$$

6. For every pair of states $q_i, q_j \in Q$ and every pair of symbols $a, b \in \Gamma$ such that $(q_j, b, L) \in \delta(q_i, a)$ and for every symbol $c \in \Gamma$

$$N_{q_i a q_j b c L1} = (\{\$ \to \bar{c}, q_i \to q_j, a \to b'\}, \emptyset, \{q_i, \$a\}, \{\#, \$F\}, \{q_j, c\bar{c}b'\}, \emptyset, 1, (1))$$

7. For every pair of states $q_i, q_j \in Q$ and every pair of symbols $a, b \in \Gamma$ such that $(q_j, b, L) \in \delta(q_i, a)$, and for every $c \in \Gamma$

$$N_{q_i a q_j b c L2} = (\{c \to \bar{\$}\}, \emptyset, \{q_j, c\bar{c}b'\}, \{\$, \#\} \cup \{kb', k'b' : k \in \Gamma \cup Q, k' \in \Gamma'\}, \{q_j, \bar{\$}\bar{c}b'\}, \{\bar{\$}k : k \in \Gamma\} \cup \{\bar{\$}F\}, 1, (1))$$

The operativity of this ANGP is the same as in Theorem 1. The only difference is that the encoded instance $q\alpha\$a\beta F$ will be sent out from processor N_q^{in} and it will probably enter in more than one processor N_{qaq_jbR} or N_{qaq_jbcL1}. In the case that the encoded string enters in at least two different processors N_{qaq_jbR}, then $q\alpha\$a\beta F$ will be transformed at every processor independently of each other and the transformed strings will enter at nodes N_q^{in}. Observe that if more than one string enters in a single processor N_q^{in}, then they will be transformed again independently of each other and they will keep the computation path according to the Turing machine movements. In the case that the encoded string enters in at least two different processors N_{qaq_jbcL1}, they will be transformed independently of each other. In processors N_{qaq_jbcL2}, the transformations depend on the symbol that is to the left of the one that is changed, so, these symbols will be transformed independently of each other. Finally, in the case that the encoded string enters in the processors N_{qaq_jbcL1} and N_{qaq_jbR}, the transformed strings will not enter the same processors. The operativity of the N_B and N_{B2} processors ensures that, whenever two or more strings enter into them, the crossover between them will be lost after a communication step (hence, only the addition of an encoded blank symbol will be effective for the rest of the computation). □

Introducing the time complexity measure

Now that we have provided a full simulation of nondeterministic Turing machines by ANGPs, we focus our attention on the time complexity of this new model. The time complexity of the Networks of Evolutionary Processors (NEPs) was first defined in [18]. We can follow the same definitions given in that work.

First, we can establish the following time complexity measure for the ANGP model: Let us consider an ANGP R that halts on every input string and the language L accepted by R. The time complexity of the accepting computation of R, if x is given as an input string, is denoted by $Time_R(x)$ and it is defined as the number of steps (both communication and evolutionary ones) such that the network R halts on x in an acceptance mode. Observe that this measure is not defined whenever the machine does not halt or it repeats two nonacceptance consecutive configurations. This measure, as defined above, fully satisfies Blum's axioms for abstract complexity measures [4]. In addition, we can define the partial function $Time_R : \mathbb{N} \rightarrow \mathbb{N}$ as follows:

$$Time_R(n) = \max\{Time_R(x) : x \in L(R), |x| = n\}$$

If we take an integer function $f : \mathbb{N} \rightarrow \mathbb{N}$, then we can define the following language class provided that there exists an ANGP R that holds the property required in the definition

$Time_{ANGP}(f) = \{L :$ There exists an ANGP, R, and a natural number n_0 such that $L = L(R)$ and $\forall n \geq n_0$ $(Time_R(n) \leq f(n))\}$

Finally, for a set of integer functions $\mathbb{C}$, we define:

$$Time_{ANGP}(\mathbb{C}) = \bigcup_{f \in \mathbb{C}} Time_{ANGP}(f).$$

We consider the set of integer functions *poly* as the set of integer polynomial functions, and we denote $Time_{ANGP}(poly)$ by $PTime_{ANGP}$ in order to preserve the classical notation in computational complexity theory. Then, we have the following result:

Lemma 3. $\mathcal{NP} \subset PTime_{ANGP}$.

Proof. If any language $L \in \mathcal{NP}$, then there exists a nondeterministic Turing machine M that works in polynomial time to accept any input string from L. In this case, we can construct an ANGP from M to work with the same input string according to Theorem 2. This ANGP carries out a constant number of steps (genetic and communication ones) to simulate a transition of the Turing machine.

We can can make the following analysis to ensure the previous affirmation: first, if the Turing machine performs a configuration transition by using a movement choice to the right (i.e. (q, a, R)), then the ANGP performs three genetic steps to simulate this movement (it changes the state, the symbol, and the tape head position). Then, the new string is communicated to the N_q^{in} processor, and it performs two genetic steps to change the symbol a' by a and $\bar{\$}$ by \$. Second, if the Turing machine makes a configuration transition by using a movement choice to the left (i.e. (q, a, L)), then the ANGP performs three genetic steps to simulate this movement (it changes the state, the symbol, and the tape head position) in processors $N_{q_i a q_j bcL1}$ and one genetic step in processors $N_{q_i a q_j bcL2}$. Again, the new string is communicated to the N_q^{in} processor, and it performs three genetic steps to change the symbol a' by a, the symbol $\bar{b}$ by b, and the symbol $\bar{\$}$ by \$. Finally, if the Turing machine explores a new cell with the blank symbol, the network performs two genetic steps (in this case, with the crossover operation in processor N_B) to add this extra symbol to the string.

The ANGP proposed in Theorem 2 simulates the Turing machine behavior by considering all the transition combinations simultaneously. If the Turing machine M works in polynomial time then the ANGP works in polynomial time too, given that the simulation of every transition of the Turing machine takes a constant number of steps in the network. Hence, $\mathcal{NP} \subset PTime_{ANGP}$. $\square$

5. Networks of Genetic Processors and Parallel Genetic Algorithms

Genetic Algorithms (GA) were proposed as programming techniques to solve optimization problems. The source of inspiration of this approach comes from the evolution of a population of individuals according to the *Darwinian* laws of mutation of individuals to face the difficulties of the environment and the survival of the best adapted ones. An overview of this approach is [19]. According to that work, the main components of a genetic algorithm (or evolution program) are:

- a genetic representation for potential solutions to the problem
- a way to create an initial population of potential solutions
- an evaluation function that plays the role of the environment, rating solutions in terms of their "*fitness*"
- genetic operators that alter the composition of the potential solutions
- values for various parameters that the genetic algorithm uses (population size, probabilities of applying genetic operators, etc.).

The GAs were proposed to find optimal solutions for optimization problems. The power of GAs is based on the quick search of optimal solutions in the search space defined by the problem. This is an advantage, but it implies that some times these techniques are unable to converge to the optimal solution due to the fact that they quickly converge to local optimal solutions. There have been many proposals to tackle this disadvantage but they are not of interest in this work.

Parallel Genetic Algorithms (PGAs) have been proposed to speed-up the efficiency of simple GAs during the search for optimal solutions. According to different reviews and Refs. [2,3,5,24], the main components for proposing parallel and

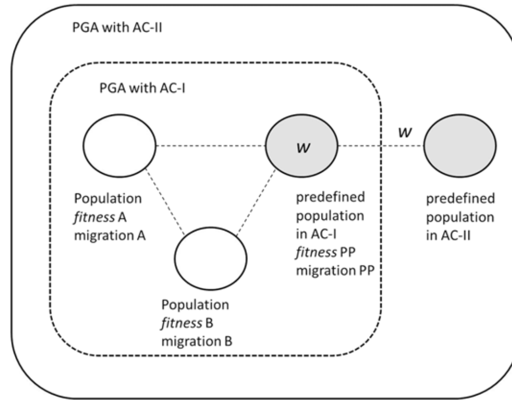

Fig. 2. The scheme to convert AC-I in AC-II.

distributed GAs are the following:

- The distribution of the individuals in different populations. They can be organized in different topologies: master–slave, multiple populations or islands, fine-grained populations or hierarchical and hybrid populations. In addition, the neighborhood connections can be rings, m, n-complete, ladders, grids, etc.
- The synchronicity of evolution and communication of the populations
- The migration phenomena: migration rates (the percentage of individuals that migrate from one population to a different one), migration selection (the selections of the individuals that migrate) and migration frequency.

All of these components have been introduced in the ANGP model proposed in this work: a topology of fully-connected populations or islands, a universal clock to synchronize the evolution and communication operations, a migration rate and a migration frequency of one hundred per cent, and a migration selection based on the input/output filters attached to the processors. Nevertheless, the Networks of Genetic Processors as acceptors that we have proposed in this work are not used to solve optimization problems but to solve decision problems. Therefore, in order to consider our approach as a classical proposal of PGAs, we need to formulate these techniques as decision problem solvers.

We propose two reasonable criteria to work with PGAs as decision problem solvers:

- *Acceptance criterion I* (AC-I)
  Let w be an input string. We say that a PGA accepts w if w appears in a predefined survival population after a finite number of iterations (operators applications, fitness selection, and individuals migration).
- *Acceptance criterion II* (AC-II)
  Let w be an input string. We say that a PGA accepts w if a distinguished individual x_{yes} appears in a predefined survival population after a finite number of iterations (operators applications, fitness selection, and individual migration). We say that the PGA rejects the input string if a distinguished individual x_{not} appears in a predefined survival population after a finite number of iterations (operators applications, fitness selection, and individual migration).

We can prove that both acceptance criteria are equivalent in the following:

Theorem 4. *Let D be a decision problem and L_D be its acceptance language. D can be solved by a Parallel Genetic Algorithm with acceptance criterion I iff it can be solved with acceptance criterion II.*

Proof. Let us suppose that D can be solved by a parallel genetic algorithm A_{D-I} with AC-I. Then, after a finite number of operators applications and fitness selection, A_{D-I} has a predefined population with the individual w. One can formulate a PGA with AC-II A_{D-II} as follows: A_{D-II} simulates A_{D-I} and it obtains the individual w in the predefined population of A_{D-I}. Let us suppose that the *fitness* function in the predefined population of A_{D-I} is g, and we reformulate this function as follows:

$$g'(x) = \begin{cases} g(y) + 1(\text{with } y = \text{argmax}_g(z)\,) & \text{if } x = w \\ g(x) & \text{otherwise} \end{cases}$$

This function ensures that the input w obtains the maximum fitness value in the predefined population of A_{D-I}. The migration rate and frequency can be adjusted to ensure that w migrates to the new predefined population in A_{D-II}. When the string arrives to the new predefined population, it can be mutated to x_{yes} (by introducing the specific mutation rules); therefore the string is accepted by A_{D-II}. Fig. 2 shows this scheme.

In the opposite situation, let us suppose that D can be solved by a genetic algorithm A_{D-II} with AC-II. Then, after a finite number of operators applications and fitness selection, A_{D-II} has a predefined population with the individual x_{yes} or x_{not}. One can formulate A_{D-I} as follows: First, A_{D-I} simulates A_{D-II}. If the acceptance decision is x_{not}, then A_{D-I} does nothing (so w will

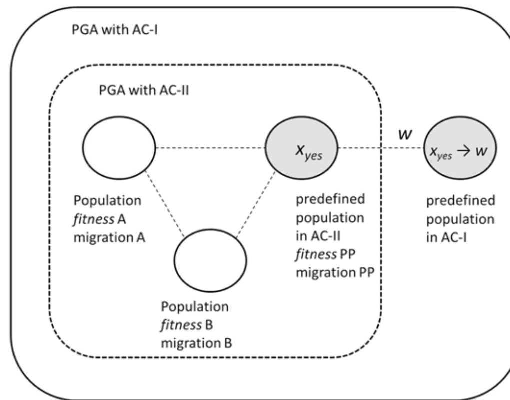

Fig. 3. The scheme to convert AC-II in AC-I.

never appear in the predefined population and A_{D-I} does not accept w). Let us suppose that A_{D-II} obtains the individual x_{yes} in the predefined population. The fitness function in the predefined population is reformulated as follows:

$$g'(x) = \begin{cases} g(y) + 1(\text{with } y = \text{argmax}_g(z)\ ) & \text{if } x = x_{yes} \\ g(x) & \text{otherwise} \end{cases}$$

This function ensures that the input x_{yes} obtains the maximum fitness value in the predefined population of A_{D-II}. The migration rate and frequency can be adjusted to ensure that x_{yes} migrates to the new predefined population in A_{D-I}. When the string arrives to the new predefined population, it can be mutated to w (by introducing the specific mutation rules). Therefore, the string is accepted by A_{D-I}. Fig. 3 shows this scheme. □

Now that we have established that both criteria AC-I and AC-II are equivalent, we have the following result, which summarizes the computational power of PGAs as decision problem solvers:

Theorem 5. *Parallel Genetic Algorithms with multiple-populations, synchronicity, and full migration phenomena are computationally complete.*

Proof. It is sufficient to consider that ANGP are PGAs with multiple-populations, synchronicity, and full migration phenomena. □

6. Final remarks

We have proposed a new computational model to achieve computational completeness by using crossover and mutation over strings. There have been previous works that explore the computational power of Networks of Evolutionary Processors with restricted processor operations [8–10,21,25]. In these previous works, it has been proved that at least one processor with insertion operation is needed in order to achieve computational completeness. Hence, the role of mutation in the Networks of Genetic Processors (and subsequently in (Parallel) Genetic Algorithms) in an isolated way is not sufficient to accept any recursively enumerable language. The other operation involved in the model is the crossover between strings. This can be formalized as a splicing operation with empty contexts proposed in previous models. Again, the role of the contexts in the splicing operation is a basic ingredient for achieving computational completeness. For example, in the works [16,17,20], it is required the splicing rules to have non empty contexts. Hence, the role of crossover in the Networks of Genetic Processors (and subsequently in (Parallel) Genetic Algorithms) in an isolated way is not sufficient to accept any recursively enumerable language.

The final conclusion that we have drawn is that the model that we have proposed is a novel approach that is based on well known operations which in an isolated way are not sufficient to get computational completeness. The new combination of operations that we have proposed, which directly relates to the classic paradigm of Genetic Algorithms, ensures computational completeness.

With respect to future works, we must explore the complexity issues of the proposed models. In this sense, the complete characterization of $PTime_{ANGP}$ remains open. This characterization will provide a formal framework to look to Genetic Algorithms to efficiently solve decision problems. Other complexity measures such as the space complexity should also be explored. In this case, the definition of *length complexity* as in [16] should help to understand the computational cost of operations such as crossover.

Acknowledgments

The authors gratefully acknowledge Prof. Dr. Mario de Jesús Pérez Jiménez (University of Seville) for the useful comments that he made, which have improved the final version of this work.

References

[1] L.M. Adleman, Molecular computation of solutions to combinatorial problems, Science 226 (1994) 1021–1024.
[2] E. Alba, J.M. Troya, A survey of parallel distributed genetic algorithms, Complexity 4 (4) (1999) 31–52.
[3] E. Alba, M. Tomassini, Parallelism and evolutionary algorithms, IEEE Transactions on Evolutionary Computation 6 (2) (2002) 443–462.
[4] M. Blum, A machine-independent theory of the complexity of recursive functions, Journal of the ACM 14 (2) (1967) 322–336.
[5] E. Cantú-Paz, Efficient and Accurate Parallel Genetic Algorithms, Kluwer Academic Publisher, 2001.
[6] J. Castellanos, C. Martín-Vide, V. Mitrana, José M. Sempere, Solving NP-complete problems with networks of evolutionary processors, in: Proceedings of the 6th International Work-conference on Artificial and Natural Neural Networks, IWANN 2001, in: LNCS, vol. 2084, Springer, 2001, pp. 621–628.
[7] J. Castellanos, Carlos Martín-Vide, Victor Mitrana, José M. Sempere, Networks of evolutionary processors, Acta Informatica 39 (2003) 517–529.
[8] J. Dässow, V. Mitrana, Networks of non-inserting evolutionary processors, in: I. Petre, G. Rozenberg (eds.), Proceedings of the Workshop on Natural Computing and Graph Transformations, NCGT 2008, 2008, pp. 29–41.
[9] J. Dässow, V. Mitrana, B. Truthe, The role of evolutionary operations in accepting hybrid networks of evolutionary processors, Information and Computation 209 (3) (2011) 368–382.
[10] J. Dässow, B. Truthe, On the power of networks of evolutionary processors, in: Proceedings of Machines, Computations and Universality, MCU 2007, in: LNCS, vol. 4664, Springer, 2007, pp. 158–169.
[11] D. Du, K. Ko, Theory of Computational Complexity, John Wiley & Sons, Inc., 2000.
[12] T. Head, Formal language theory and DNA: an analysis of the generative capacity of specific recombinant behaviors, Bulletin of Mathematical Biology 49 (1987) 737–759.
[13] J.E. Hopcroft, J.E. Ullman, Introduction to Automata Theory, Languages and Computation, Addison-Wesley Publishing Co., 1979.
[14] L. Kari, G. Rozenberg, The many facets of natural computing, Communications of the ACM 51 (10) (2008) 72–83.
[15] F. Manea, V. Mitrana, All NP-problems can be solved in polynomial time by accepting hybrid networks of evolutionary processors of constant size, Information Processing Letters 103 (2007) 112–118.
[16] F. Manea, C. Martín-Vide, V. Mitrana, Accepting networks of splicing processors: Complexity results, Theoretical Computer Science 371 (2007) 72–82.
[17] F. Manea, C. Martín-Vide, V. Mitrana, Accepting networks of splicing processors, in: Proceedings of the First Conference on Computability in Europe, CiE 2005, in: LNCS, vol. 3526, Springer, 2005, pp. 300–309.
[18] M. Margenstern, V. Mitrana, M.J. Pérez-Jiménez, Accepting hybrid networks of evolutionary processors, in: Proceedings of the International Meeting on DNA Computing, DNA 10, in: LNCS, vol. 3384, Springer, 2005, pp. 235–246.
[19] Z. Michalewicz, Genetic Algorithms + Data Structures = Evolution Programs, Third, Revised and Extended Edition, Springer, 1996.
[20] V. Mitrana, I. Petre, V. Rogojin, Accepting splicing systems, Theoretical Computer Science 411 (2010) 2414–2422.
[21] V. Mitrana, B. Truthe, On accepting networks of evolutionary processors with at most two types of nodes, in: Proceedings of Language and Automata Theory and Applications, LATA 2009, in: LNCS, vol. 5457, Springer, 2009, pp. 588–600.
[22] Gh. Păun, Membrane Computing. An Introduction, Springer-Verlag, 2002.
[23] Gh. Păun, G. Rozenberg, A. Salomaa, DNA Computing. New Computing Paradigms, Springer, 1998.
[24] M. Schwehm, Parallel Population Models for Genetic Algorithms, Universität Erlangen-Nürnberg 1996.
[25] B. Truthe, On small accepting networks of evolutionary processors with regular filters, in: J. Dassow, B. Truthe (Eds.) Proceedings of the Colloquium on Occasion of the 50th Birhtday of Victor Mitrana, Otto-von Guericke Universität Magdeburg, 2008, pp. 37–52.