Swarm and Evolutionary Computation 76 (2023) 101223

Contents lists available at ScienceDirect

Swarm and Evolutionary Computation

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

Inferring P systems from their computing steps: An evolutionary approach

Alberto Leporati [a,*], Luca Manzoni [b], Giancarlo Mauri [a], Gloria Pietropolli [b], Claudio Zandron [a]

[a] Dipartimento di Informatica, Sistemistica e Comunicazione, Università degli Studi di Milano-Bicocca, Viale Sarca 336, 20126, Milan, Italy
[b] Dipartimento di Matematica e Geoscienze, Università degli Studi di Trieste, Via Alfonso Valerio 12/1, 34127, Trieste, Italy

ARTICLE INFO

MSC:
68Q07

Keywords:
P systems
Evolutionary algorithms
$(\mu+\lambda)$-EA

ABSTRACT

Inferring the structure and operation of a computing model, given some observations of its behavior, is in general a desirable but daunting task. In this paper we try to solve a constrained version of this problem. We consider a P system Π with active membranes and using cooperative rewriting, communication, and division rules and a collection of pairs of its consecutive configurations. Then, we feed this collection of configurations as input to a $(\mu+\lambda)$ evolutionary algorithm that evolves a population of (initially random) P systems, each with its own rules, with the aim of obtaining an individual that approximates Π as well as possible. We discuss the results obtained on different benchmark problems, designed to test the ability to infer cooperative rewriting, communication, and membrane division rules. We will also provide a description of how fitness results are influenced by different setting of the hyperparameters of the evolutionary algorithm. The results show that the proposed approach is able to find correct solutions for small problems, and it is a promising research direction for the automatic synthesis of P systems.

1. Introduction

Membrane systems, also called *P systems*, have been introduced by Gh. Păun in 1998 as a framework for defining parallel models of computation inspired from the functioning of living cells. In the basic model, a tree-like membrane hierarchy divides the Euclidean space into regions. Each membrane can contain other membranes; the outermost membrane is called the *skin*, and divides the P system from the surrounding *environment*. The leaves of the membrane hierarchy correspond to *elementary* membranes, that is, membranes that do not contain any other membrane. Each membrane delimits a *region*, that contains a (possibly empty) finite *multiset* of objects over a given finite alphabet. Every region contains also a set of *rewriting rules*, written following an appropriate formal grammar; these rules are usually applied in the so-called *maximally parallel way* to evolve the multiset of objects located in the region. Each of the objects obtained from the application of the rewriting rules can possibly cross the membrane enclosing the region, thus moving up in the membrane hierarchy, or can cross one of the membranes contained into the region in which it has been produced, moving down towards the bottom of the membrane hierarchy. This is made, respectively, through so-called *send-out* and *send-in communication* rules.

Since the introduction of this basic model of computation, many variants of P systems have been proposed and studied in the literature, giving rise to the research topic known as *membrane computing*. This is currently a very active field of research, where three types of models are extensively studied: *cell-like* P systems [1], based on the tree-like hierarchical structure of the membranes within a living cell; *tissue-like* P systems [2], based on the intercommunication between the cells in biological tissues, and usually modeled as a graph-like interconnection structure of computing elements, each representing a cell; and *spiking neural* P systems [3], inspired by the electrical impulses (called *spikes*) that neurons emit as information, also represented as networks of cells exchanging information in the form of spike symbols. Two nice and clear introductions to P systems can be found in [4,5], while a comprehensive presentation of the state of knowledge in 2010 is given in *The Oxford Handbook of Membrane Computing* [6]. For the latest developments and an extensive bibliography, we refer the reader to the P systems Web page [7].

The reason why *P Systems* are an attractive and widely studied computing model of computation can be essentially summarized by the fact that they are an intrinsically parallel computing model, different variants are universal, they can be used to attack computationally hard problems by trading space for time [8], can be applied to various real life problems and to the description of various biological systems. Since their birth, P systems have stimulated a relevant number of investigations both on the theoretical side – dealing with their computing power and efficiency in solving computationally difficult

* Corresponding author.
E-mail addresses: alberto.leporati@unimib.it (A. Leporati), lmanzoni@units.it (L. Manzoni), giancarlo.mauri@unimib.it (G. Mauri), GLORIA.PIETROPOLLI@phd.units.it (G. Pietropolli), claudio.zandron@unimib.it (C. Zandron).

https://doi.org/10.1016/j.swevo.2022.101223
Received 4 May 2022; Received in revised form 3 October 2022; Accepted 24 November 2022
Available online 7 December 2022

problems – and from the point of view of applications – studying how P systems can be applied to simulate several kinds of natural phenomena. For these reasons, a significant number of applications were reported in several areas – biology, bio-medicine, linguistics, computer graphics, economics, approximate optimization, cryptography, etc [9]. To empathize how *P systems* have been applied among different frameworks and used for a diversity of tasks, let us provide some of their applications proposed in recent years. In [10], authors propose a membrane computing model to directly derive an approximate solution of combinatorial problems with a specific reference to the 0/1 knapsack problem On the other hand, in [11], a complete arithmetic calculator implemented by spiking neural P systems (SNPS) is proposed. Lastly, in [12], authors proposes a novel multi-behaviors coordination controller model using enzymatic numerical P systems for autonomous mobile robots navigation in unknown environments.

Focusing on cell-like P systems, one of the models that have attracted the most attention are P systems *with active membranes* [13]. In this kind of P systems the membranes take a more active role during the computations: not only they contain objects, rules, and other membranes, but each membrane may also have an associated electrical charge, or a thickness, or some other feature, that affects the applicability of the rules. In this paper we will not consider any of these additional membrane features; however, membranes will have the ability to duplicate themselves through *division* rules. These rules, inspired by biological mitosis, duplicate an existing membrane, thus modifying the membrane hierarchy, copying the contents (objects, and membranes) of the membrane into the two offspring membranes. Moreover, all the rules we will consider – either rewriting, communication, or division rules – will be *cooperative*, meaning that they are triggered by the simultaneous presence of a prescribed multiset of objects.

Designing a P system for a given application is often tedious handiwork, requiring typically many hours of work, focus, and skill. This is why it is of great interest to look for design techniques that would help researchers in this work. In this paper, we investigate the possibility to use *evolutionary algorithms* to automatically synthesize P systems with active membranes. These algorithms start with a population of randomly generated P systems, and are given as input a collection of configuration pairs, each describing the desired effect of a computation step of the target P system. The population of P systems is evolved for a number of generations, with the aim of obtaining at least one individual that matches the described computation steps as well as possible.

Notice that the idea of combining evolutionary algorithms and membrane computing is not new. In fact, *Evolutionary Membrane Computing* (**EMC**) is a topic by itself, with multiple works published in the literature; for a survey, we refer the reader to [14]. In general, we can identify two ways of combining evolutionary algorithms and P systems: Membrane-Inspired Evolutionary Algorithm (**MIEA**) and Automated Design of Membrane Computing Models (**ADMCM**). In MIEA, the P systems are used as a part of an evolutionary algorithm. That is, P systems are not the output of a MIEA, but a way in which an optimization process can be improved. On the other side, ADMCM aims to circumvent the programmability issue of membrane-based models for membrane systems by searching for automated synthesis of such models. Thus, in this approach, an evolutionary optimization process produces, as output, a P system to solve a specific computational task. The exact kind of output depends on the specific algorithm and task. The first approach to perform this kind of task came from [15], where an evolutionary algorithm, the *PSystemEvolver*, was introduced for calculating n^2. In [16] the previous method has been extended in order to compute generic squared numbers, by introducing a binary encoding technique to represent an evaluation rule set of a P system. Moreover, a QIEA (Quantum-Inspired gate update) has been exploited to make the population of P system evolve towards a successful one. In [17], again a binary encoding technique has been applied to the genetic algorithm and, moreover, the evaluation method has been improved by non-determinism and halting penalty factors. In [18], it is introduced an automatic design method of a cell-like P system for performing five basic arithmetic operations. In [14], the investigations concentrated on how to design a redundant rule set and how to select a proper subset of the latter with the aim of compute: $2(n-1)$, $2n-1$, n^2, $\frac{1}{2}[n(n-1)]$, $n(n-1)$, $(n-1)^2+2^n+2$, $a^{2^n}b^{3^n}$ and $\frac{1}{2}(3^n-1)$, $(n>1$ or $2)$. In [19] has been designed a cell-like halting P system for 4^2, by tuning for the first time membrane structures, initial objects and evolution rules. In [20], an automatic design method, the *Permutation Penalty Genetic Algorithm*, is proposed for a deterministic and non-halting membrane system by tuning membrane structures, initial objects and evolution rules.

As stated above, in this paper we propose a way of performing the automatic synthesis of the rules of P systems with active membranes using cooperative evolution, communication, and division rules, by means of evolutionary algorithms and using only pairs of configurations in consecutive time steps to drive the synthesis. In particular, we define a $(\mu+\lambda)$ evolutionary algorithm using sets of P systems' rules, where the fitness function is based on a distance measure between labeled trees, each representing a configuration of a P system. The choice of using $(\mu+\lambda)$ evolutionary algorithms is twofold: first of all, the algorithm is simple but generally effective, on the other hand, the distance measure used should be sufficient to guide the search process. We then define seven different benchmark problems, each one dedicated to investigating the ability of the evolutionary algorithm to learn different types of rules, namely send-in, send-out, division, and object rewriting rules. Let us note that, while we employ our specific model of P systems with active membranes, the approach proposed here is quite general and can, in principle, be applied to a large number of different P system models.

The rest of this paper is organized as follows: Section 2 recalls some basic notions and notations which are used in the following. Section 3 details how the proposed $(\mu+\lambda)$ evolutionary algorithm works. The experimental settings and the benchmark problems used during our computer experiments are exposed in Section 4, while the results are presented in Section 5. Section 6 contains the conclusions, and proposes some directions for future research.

2. Preliminaries

In this section, we introduce some notions and notations that will be used in the rest of the paper. In particular, we recall the definition and operation of P systems with active membranes and cooperative rules, and the definition and operation of $(\mu+\lambda)$ evolutionary algorithms.

2.1. P systems with active membranes and cooperative rules

In this paper, we consider P systems with active membranes, without electrical charges and using cooperative rewriting rules, cooperative communication (send-in and send-out) rules, and cooperative weak division rules for elementary and non-elementary membranes. A detailed introduction to this model of computation can be found in *The Oxford Handbook of Membrane Computing* [6], which also contains a more general introduction to P systems, how they operate and the related notions of formal language theory and multiset processing.

Definition 1. A *P system with active membranes and cooperative rules*, of initial degree $d \geq 1$, is a tuple

$$\Pi = (\Gamma, \Lambda, \mu, w_{h_1}, \ldots, w_{h_d}, R)$$

where:

- Γ is an alphabet, i.e., a finite non-empty set of symbols, usually called *objects*;
- Λ is a finite set of labels;
- μ is a membrane structure (i.e., a rooted *unordered* tree, usually represented by nested brackets) consisting of d membranes labeled by elements of Λ in a one-to-one way;

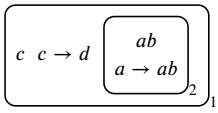
- $w_{h_1}, \ldots, w_{h_d}$, with $h_1, \ldots, h_d \in \Lambda$, are multisets (finite sets with multiplicity) of objects in Γ, describing the initial contents of each of the d regions of μ;
- R is a finite set of rules.

The rules in R considered in this paper are of the following types:

- *Cooperative rewriting rules*, of the form $[u \to v]_h$ for $h \in \Lambda$ and $u, v \in \Gamma^\star$. They can be applied inside a membrane labeled by h and containing the multiset of objects u; this multiset is rewritten into the multiset v, i.e., the objects of u are removed from the multiset in h and replaced by the objects in v.
- *Cooperative communication send-in rules*, of the form $u\,[\,]_h \to [v]_h$ for $h \in \Lambda$ and $u, v \in \Gamma^\star$. They can be applied to a membrane labeled by h and such the multiset of objects u occurs in its parent region, i.e., the region containing membrane h; the objects of u are removed from the content of the parent region, and the objects of multiset v are added to the content of membrane h.
- *Cooperative communication send-out rules*, of the form $[u]_h \to [\,]_h\, v$ for $h \in \Lambda$ and $u, v \in \Gamma^\star$. They can be applied inside a membrane labeled h and containing the multiset of objects u; the multiset u is removed from the content of membrane h, and the multiset v is added to the content of the parent membrane of h.
- *Cooperative weak division rules*, of the form $[u]_h \to [v]_h\,[w]_h$ for $h \in \Lambda$ and $u, v, w \in \Gamma^\star$. They can be applied to a membrane labeled by h (excluding the skin membrane), containing the multiset of objects u, and possibly further membranes; the membrane is divided into two new membranes having label h; the multiset u is replaced in the two offspring membranes, respectively, by the multisets of objects v and w, while the rest of the content (including whole membrane substructures) is replicated in both.

The *configuration of a membrane* h at a given time step t is the multiset of objects contained in it at that moment. The *configuration of a P system* Π, at time step t, is described by its membrane structure μ at that moment, represented as a *labeled* rooted unordered tree: each node of the tree, representing a membrane of Π, is labeled with the configuration of that membrane at time step t.

Let us illustrate the definition of *P System*, and the relative initial configuration, with the following example:



Here it is represented a *P system* (of degree 3) with the following characteristics:

- $\Gamma = \{a, b, c, d\}$ and $\Lambda = \{1, 2\}$
- $\mu = [_1[_2]_2]_1$
- $w_1 = c$ and $R_1 = \{c \to cd\}$
- $w_2 = ab$ and $R_2 = \{a \to ab\}$

A *computation step* changes the current configuration of the P system according to the following set of principles:

(i) Each object and membrane can be subject to at most one rule per step, except for rewriting rules: inside each membrane, several rewriting rules can be applied simultaneously.

(ii) The application of rules is *maximally parallel* [6]. Each object appearing on the left-hand side of rewriting, communication or division rules must be subject to exactly one copy of them. Analogously, each membrane can only be subject to one communication or division rule per computation step; for this reason, these rules are said to be *blocking rules*. As a result, the only objects and membranes that do not evolve are those associated with no rule.

(iii) When several conflicting rules can be applied at the same time, a nondeterministic choice is performed; this implies that, in general, multiple possible configurations can be reached after a computation step from a certain configuration.

(iv) In each computation step, all the chosen rules are applied simultaneously in an atomic way. However, in order to clarify the operational semantics, each computation step is conventionally described as a sequence of micro-steps as follows. First, all rewriting rules are applied inside the elementary membranes, followed by all communication and division rules involving the membranes themselves; this process is then repeated on the membranes containing them, and so on towards the root (outermost membrane). In other words, the membranes evolve only after their internal configuration has been updated. For instance, before a membrane division occurs, all chosen object rewriting rules must be applied inside it; this way, the objects that are duplicated during the division are already the final ones.

(v) The outermost membrane cannot be divided, and any object sent out from it cannot re-enter the system again.

A *halting computation* of the P system Π is a finite sequence $\vec{C} = (C_0, \ldots, C_k)$ of configurations, where C_0 is the initial configuration, every C_{i+1} is reachable from C_i via a single computation step, and no rules of Π are applicable in C_k. A *non-halting* computation $\vec{C} = (C_i : i \in \mathbb{N})$ consists of infinitely many configurations, again starting from the initial one and generated by successive computation steps, where the applicable rules are never exhausted.

The *result* of a halting computation can be defined in several ways; for example, the P system may have a designated *output membrane*, and the result is defined as the multiset of objects contained in such membrane in the halting configuration. An alternative consists of considering the objects emitted from the system to the environment. Indeed, sometimes P systems are used as *language recognizers* (or, equivalently, to solve *decision problems*) by employing two distinguished objects yes and no: in this case we assume that all computations are halting, and that either one copy of object yes or one copy of object no is sent-out from the outermost membrane, and only in the last computation step, in order to signal acceptance or rejection, respectively. If all computations starting from the same initial configuration are accepting, or all are rejecting, then the P system is said to be *confluent*; if this is not necessarily the case, then we have a *non-confluent* P system, and the overall result is established as for nondeterministic Turing machines: it is acceptance if and only if an accepting computation exists [21]. All that said, in this paper we will only deal with *deterministic* P systems, which are confluent by construction. Moreover, the consecutive configuration pairs (C_i, C_{i+1}) will define a finite portion of a computation, but not necessarily a halting one; stated otherwise, it could happen that no one of the configurations C_{i+1}, for $0 \le i < n$, will be a final one, in which no further rules can be applied.

2.2. $(\mu + \lambda)$ evolutionary algorithms

The $(\mu + \lambda)$ *evolutionary algorithms*, or $(\mu + \lambda) - EA$ for short, belong to the class of so-called Evolutionary Strategy (ES), that has been introduced in [22] as a sub-field of a more general class of optimization techniques called *population-based* methods.

These techniques, inspired by biology, evolve and iteratively improve a population of individuals that encode candidate solutions to a given problem. At each iteration, called a *generation*, the candidate solutions are selected and mutated in order to obtain a new offspring population. These operations are performed by considering the fitness of individuals, computed using a *fitness function*, which is a measurement of the quality of the candidate solutions encoded by the individuals.

In a $(\mu + \lambda)$-EA the initial population is composed by μ individuals, randomly generated. By mutation, λ/μ children are produced for each

of the μ parents, thus generating a total of λ offsprings. Then the μ parents, together with their λ children, compete against each other and only the individuals having the best fitness survive until the next generation, while the remaining ones are simply eliminated from the population. The process is then repeated again, with the best individuals of the previous generation as the new parents. After a predefined number of generations, the resulting population will hopefully contain an optimal solution — that is, a solution having the highest possible fitness value (or the lowest, depending upon how the fitness function is defined).

The steps performed by a $(\mu+\lambda)$-EA are summarized in Algorithm 1. In line 1, the initial population of μ individuals is randomly generated. Each iteration of the `for` loop in lines 2–8 performs a generation, in which a new offspring population is produced. This is made by first initializing the auxiliary variable `offsprings` to the empty set (line 3); the content of this variable is then updated in the `for` loop of lines 4–6, adding at each iteration one individual, produced by mutation from one of the μ parents; finally, in line 7 the set of all offsprings is put together with the current population, and a new population is created by selecting only the best μ individuals. At the end of the outer `for` loop, at line 9, the best individual of the last population is returned as the result. In what follows we will assume that λ is a multiple of μ; in this way, each of the μ individuals in the parent population will produce the same number of offsprings when the `for` loop in lines 4–6 is executed.

For further details about this technique, we refer the reader to [23].

Algorithm 1 The generic scheme of a $(\mu + \lambda)$-EA

```
1: population ← random-population(μ)
2: for i = 1 to num-gen do
3:     offsprings ← ∅
4:     for j = 1 to λ do
5:         offsprings ← offsprings ∪ mutate(population[j mod μ])
6:     end for
7:     population ← truncated-selection(population ∪ offsprings, μ)
8: end for
9: return best(population)
```

3. A $(\mu + \lambda)$-EA for the inference of p systems

In this section we detail how the generic scheme of a $(\mu + \lambda)$-EA described above can be adapted to evolve a population of P systems, each with its own set of rules. In particular, the $(\mu + \lambda)$-EA will be used to synthesize the sets of rules of the P systems, so it is necessary to:

- describe how the rules of a P system will be represented;
- provide a fitness function to evaluate the solutions encoded by the individuals of the population.
- detail how to generate the initial population;
- say precisely how to perform the mutation of a set of rules;

Each of these points is discussed in detail in the following.

Representation of solutions. The set of rules of a P system is represented via a list, containing a number of rules comprised between a *Minimum number of rules* and a *Maximum number of rules*, which are two parameters of the algorithm. Each entry in the list represents a rule, by using an appropriate data structure, that stores the type of the rule (rewriting, send-in, send-out, division), the membrane label to which it is applied, and the left-hand side (LHS) and the right-hand side (RHS) of the rule. Each of the two sides indicates the multiset of objects which is either consumed (in the LHS) or produced (in the RHS) by the rule. By means of two parameters, named *Maximum LHS size* and *Maximum RHS size*, we can impose that the LHS and/or the RHS contain a predefined maximum number of objects; this can be used, for example, to force the evolutionary algorithm to produce only non-cooperative rules, by setting *Maximum LHS size* to 1. In case of division rules, the RHS contains two entries, each corresponding to one of the membranes generated.

To ensure that the resulting P system is *deterministic*, the rules will be applied with a priority given by their position in the list. Observe that this choice entails the following behavior: if (for example) rules $[a \to b]_h$ and $[a \to c]_h$ are both present in a list in that order, then the first one is always applied and the second one will never be applied, because all copies of object a that occur in membrane h are assigned to the first rule. Notice that this restriction is only imposed here to obtain deterministic systems in the computer experiments described later; for nondeterministic systems rule priorities can be ignored. However, let us observe that nondeterministic systems pose additional challenges; for example, we cannot simply require a nondeterministic system to be confluent, since the property of being confluent is in general undecidable. For this reason, in the rest of this paper we will only deal with deterministic P systems, leaving the investigation on nondeterministic ones to future work.

Fitness. The definition of a fitness function is necessary to quantify how well the rules inferred by the evolutionary algorithm approximate the rules of the original *P system*. In this scenario, the fitness function is a measure that compare two *P system* (sub)configurations, and aims to quantify how much they differ one from another. Given two P system (sub)configurations $C_1 = [w_1[w_2]_{h_2} \dots [w_m]_{h_m}]_{h_1}$ and $C_2 = [v_1[v_2]_{k_2} \dots [v_\ell]_{k_\ell}]_{k_1}$, our idea is to define the distance between them as the *minimum* among all the costs of the sequences of operations $o_1, \dots, o_m$ that transform C_1 in C_2. So, now it is necessary to define which are the possible operations that can occur between two *P systems*. To do that, we take inspiration by the fact that the membrane structure of a P system is a rooted unordered tree: we define, in fact, these operations by emulating a distance inspired by the *edit distance* between labeled trees.

Specifically, given a P system configuration, we consider three main operations: *adding* a membrane and its content (including other membranes), *removing* a membrane and its content, or *changing* the content of one membrane. To each operation we associate a cost in the following way:

- *Addition* of a membrane and its content (including other membranes): the cost is equal to the number of membranes added to the membrane structure.
- *Removal* of a membrane and its content: the cost of removing a subtree in the membrane structure is equal to the number of membranes removed.
- *Change* of the objects contained into a membrane: the cost is computed using the *Jaccard distance*, calculated over the current multiset A of objects contained in the membrane and the multiset B of objects obtained after the change:

  $$d_J(A, B) = 1 - \frac{|A \cap B|}{|A \cup B|}$$

  This distance measures the *dissimilarity* between multisets A and B. Its value is comprised between 0 and 1: it is 0 if A and B are the same multiset, and it is 1 when A and B have no objects in common.

We denote by $\gamma(o)$ the cost of one operation o. Given a sequence of addition, removal, or change operations $o_1, \dots, o_m$, the total cost of the sequence will be given by $\sum_{i=1}^{m} \gamma(o_i)$. Thus, now we know how to theoretically define the distance between two (sub)configurations C_1 and C_2. Anyway, computing this minimum cost could be a daunting task as it requires to consider all possible sequences of operations that transform C_1 in C_2. Hence, we will approximate it as follows:

1. If the two root membranes have different labels (i.e., $h_1 \neq k_1$), then the distance is the cost of removing the entire membrane structure of C_1 and adding the entire membrane structure of C_2.

2. Otherwise (if the two root membranes have the same labels), the distance is the cost $d_J(w_1, v_1)$ of changing the multiset w_1 of objects contained in the root membrane of C_1 into the multiset of objects v_1 contained into the root membrane of C_2, plus the cost of changing the rest of the membrane structure recursively, as follows:

   (a) first, all the membranes contained in the root membranes of C_1 and C_2 are partitioned according to their label: so, for each $h \in \Lambda$, we define the sets $C_{1,h} = \{[w_i]_{h_i} : h_i = h\}$ and $C_{2,h} = \{[v_i]_{k_i} : k_i = h\}$;

   (b) then, for each $h \in \Lambda$, the cost of transforming the set of membranes $C_{1,h}$ into the set of membranes $C_{2,h}$ is obtained by:

      i. sorting the content of each set $C_{1,h}$ and $C_{2,h}$ according to the objects contained in the outermost membrane, in lexicographic order;
      ii. computing the sum of the distances between each pair of membranes in $C_{1,h}$ and $C_{2,h}$, paired according to their order.

   Notice that the lexicographic ordering made in the first item ensures that if two systems have the same content then their distance is zero.

The resulting approximated distance has the property that membranes with equal label can change their content, while membranes with different labels need to be removed and replaced. This is consistent with the fact that, since the rules are "attached" to the membranes, if two membranes with the same content have different labels, there is still no easy way to "move" the rules acting on one label to another label.

In order to render completely clear the definition of fitness that we introduced, let us provide a practical example of its operation. Let us consider the following membrane configurations:

where we have two *P Systems* that are almost identical except for the content of the membrane labeled as k. In fact, inside this membrane, while the objects a and b coincides in the *P Systems*, the third object differs, and specifically is c in the first configuration and d in the second one. As the number and structure of membrane, together with their names, coincide, no addition nor subtraction has to be performed to transform the first *P System* into the second one. It is necessary, instead, to perform a *changing* operation between the element c and d of the membrane k. The cost, according to the fitness definition that we introduced, correspond to the Jaccard distance between the two multi-sets, computed as:

$$d_J = 1 - \frac{|\{a,b,c\} \cap \{a,b,d\}|}{|\{a,b,c\} \cup \{a,b,d\}|} = 1 - \frac{|\{a,b\}|}{|\{a,b,c,d\}|} = 1 - \frac{1}{2} = 0.5$$

As no other operations have to performed, the total cost (i.e. the fitness) corresponds also to 0.5.

Generation of the initial population. To make an initial population we need to define a way to generate sets of rules – as described above – for P systems. A possibility is to assign the value m to both parameters *Minimum number of rules* and *Maximum number of rules*, and thus generate exactly m rules in every P system. Each rule is generated in the following way:

1. A rule type is selected; evolution, send-in, send-out, or division;
2. A membrane label for the rule is selected, chosen uniformly at random across the set of possible labels for the P system;
3. The LHS is generated by creating a multiset (represented as a list) of objects; these are simply sampled with replacement from the alphabet of the P system. The size of the multiset is uniformly selected between 1 and the value of the parameter *Maximum LHS size*. As noted above, if this parameter is set to 1 then only *non-cooperative* rules will be generated;
4. One or two multisets of objects, of size between 1 and *Maximum RHS size*, are generated, depending upon the type or rule: one multiset for evolution and communication rules, and two multisets for division rules. Each multiset is generated through a sampling procedure with replacement, just like it happens for the LHS.

The same procedure for the generation of a random rule is also applied when a set of rules is modified via the *mutation* operator.

Mutation. In the proposed EA, a mutation is defined as either the insertion of a randomly generated rule or the deletion of a randomly selected rule. In particular, a single mutation operator consists of one between:

- Removing a rule uniformly selected at random across all the rules of the system, provided that this removal will not decrease the size of the system below the *Minimum number of rules*.
- Adding a randomly generated rule in front of the list of rules, provided that this addition will not increase the size of the system above the *Maximum number of rules*. Notice that, by performing the addition in front of the list, the new rule is given the highest priority.

The maximum number of mutation operations performed on each individual is governed by the parameter q of the algorithm (a more comfortable name with respect to *Maximum number of rules changed by mutation*). However, the number of rules changed by mutation is *not* selected uniformly at random. In fact, the probability of performing $1 \le i \le q$ mutation operations is given by:

$$p_i = \frac{2(q - i + 1)}{q(q + 1)}$$

This means that the probability of performing i mutation operations decreases when i increases, assuring that, while a large number of mutation operations remains possible, the common case will be a limited amount of mutations. So doing, on one side mutation allows to possibly escape from local minima, while on the other side we limit extensive mutations that can be destructive for the individuals.

4. Experimental setting

To assess the ability of the proposed $(\mu + \lambda)$-EA, we performed multiple experiments with different benchmark problems, each of these involving a parametric class of P systems. Specifically, we will consider the following problems: *send-in* and *send-out* (to test the ability to learn communication rules), *variable assignment* (to test the ability to learn weak membrane division rules), and *Turing machine simulation* (to test the ability to learn cooperative rewriting rules). Moreover, we will test our method also to automatically design three basic arithmetic (unary) operations, namely: addition, multiplication and division.

The size of each *P system* depends upon the parameter n, which determines the input size of the system. In what follows we describe the results of the experiments we have performed for values of n varying from 2 to 5. Intuitively, the larger the input size, the more difficult it will be to infer the rules of the *P system*.

In all cases the P systems employed are *deterministic*, as discussed above, hence there exists a unique computation $C_0, C_1, \ldots$ starting from the initial configuration C_0. To obtain a collection of input–output pairs for the $(\mu + \lambda)$-EA, a finite prefix of the computation $C_0, C_1, \ldots, C_m$ is extracted and the m pairs (C_i, C_{i+1}), for $0 \le i < m$, form the training set. In the case of the send-in, send-out, and variable assignment problems,

the number m is set equal to n, since after n steps there are no more applicable rules in the P system. For the Turing machine simulation problem the number m of pairs is set instead to n^2, since the operations performed by the Turing machine being simulated will require more steps to actually produce a meaningful set of input–output pairs. Code, for the complete reproducibility of the proposed experiments, is available at https://github.com/gpietrop/Psystem-GA [24].

4.1. Benchmark problems

Let us now define the problems we will use to assess the learning ability of the proposed $(\mu + \lambda)$-EA. For a more detailed description of these benchmark problems, the reader can refer to [8].

Send-in problem. The first problem consists of sending n objects from the outer membrane h to the inner membrane k by following a specified ordered sequence, as detailed below. The initial configuration of the system is the following:

$$[x_{0,0}\ x_{1,1}\ \dots\ x_{n-1,n-1}\ [\ ]_k]_h$$

The P system's dynamics is given by the following rules:

$$[x_{i,j} \to x_{i,j-1}]_h \qquad \text{for } 0 \le i < n \text{ and } 0 < j \le i$$
$$x_{i,0}\ [\ ]_k \to [x_i]_k \qquad \text{for } 0 \le i < n$$

where the first rule simply "counts down" i times, for each $0 \le i < n$, starting from the object $x_{i,i}$. In this way, after i time steps, the object $x_{i,i}$ has been rewritten into $x_{i,0}$, that can be then sent in the membrane k as x_i.

Send-out problem. Symmetric with respect to the send-in problem, the send-out problem starts with the objects $x_{i,i}$, for $0 \le i < n$, inside the innermost membrane k:

$$\left[[x_{0,0}\ x_{1,1}\ \dots\ x_{n-1,n-1}]_k\right]_h$$

The dynamics of the P system is governed by the following rules:

$$[x_{i,j} \to x_{i,j-1}]_k \qquad \text{for } 0 \le i < n \text{ and } 0 < j \le i$$
$$[x_{i,0}]_k \to [\ ]_k\ x_i \qquad \text{for } 0 \le i < n$$

As before, the first kind of rules is simply used to perform a "count down" from each object $x_{i,i}$ to the corresponding object $x_{i,0}$, before allowing the second kind of rules to trigger, sending out $x_{i,0}$ as x_i in the outermost membrane h.

Variable assignment problem. This problem consists in the generation of 2^n membranes containing all 2^n assignments of n Boolean variables. The initial configuration of the P system is:

$$\left[[x_{0,0}\ x_{1,1}\ \dots\ x_{n-1,n-1}]_k\right]_h$$

and the P system's dynamics is given by the following rules:

$$[x_{i,j} \to x_{i,j-1}]_k \qquad \text{for } 0 \le i < n \text{ and } 0 < j \le i$$
$$[x_{i,0}]_k \to [t_i]_k\ [f_i]_k \qquad \text{for } 0 \le i < n$$

where the first kind of rules performs a "count down" for i times starting from each variable $x_{i,i}$, just like in the send-in and send-out problems. Then, the second kind of rules is applied and membrane k is divided into two other membranes, one containing the object t_i and the other one containing the object f_i. The division of membrane k is repeated n times, with each resulting membrane containing n objects of the form t_i or f_i, for $0 \le i < n$, representing one assignment of n Boolean variables.

Turing machine simulation problem. A Turing machine is a mathematical model that describes an abstract machine that changes symbols on a tape according to some rules. One of the main strengths of this model is the possibility to implement every computer program algorithm [25]. This problem consists of performing the simulation of a Turing machine, when the space employed (i.e., the tape of length n) is known *a priori*. In particular, the simulated Turing machine has two states, q and r, the binary alphabet $\{0, 1\}$, and its transition function performs the increment of a binary counter starting at zero. In the P system that performs the simulation, the object q_i encodes both the information of which is the state of the Turing machine (i.e., q) and of which is the current position of the tape head (i.e., i). The content of the tape is encoded by $\sigma_{0,0}, \dots, \sigma_{n-1,n-1}$ where $\sigma_i \in \{0, 1\}$ is a symbol of the alphabet, and the second index indicates its position on the tape. The initial configuration of the P system is:

$$[0_0\ 0_1\ \dots\ 0_{n-1}\ q_0]_h$$

The rules that govern this P system simultaneously update the state and head position of the simulated Turing machine, and the content of the cell previously located under the tape head, as follows:

$$[q_i\ 1_i \to q_{i+1}\ 0_i]_h \qquad \text{for } 0 \le i < n-1$$
$$[q_i\ 0_i \to r_{i-1}\ 1_i]_h \qquad \text{for } 0 < i < n$$
$$[r_i\ 0_i \to r_{i-1}\ 0_i]_h \qquad \text{for } 0 < i < n$$
$$[r_0 \to q_0]_h$$
$$[q_0\ 0_0 \to q_0\ 1_0]_h$$

The first rule keeps the state unchanged, whereas the head position changes from i to $i+1$ and the content of the cell i turns from 1 to 0. The second rule mutates the state from q to r, the head position from i to $i-1$ and the content of the cell i from 0 to 1. The third rule maintains unaltered state and content of the cell i, altering only the head position from i to $i-1$. The fourth rule simply changes the state from r to q when the head position corresponds to the beginning of the tape. Finally, the fifth rule changes the content of the cell from 0 to 1 when both the head position is 0 and the state is q.

Let us provide an example of how Turing machine perform a binary counter.

⋯	0	0	0	0	**state:** q_0
⋯	0	0	0	1	**state:** q_0
⋯	0	0	0	0	**state:** q_1
⋯	0	0	1	0	**state:** r_0
⋯	0	0	1	0	**state:** q_0
⋯	0	0	1	1	**state:** q_0

Here we can see that at the beginning we have a tape that represents the number 0 and then, applying the rules defined above, the counter reaches number 1, number 2 (that in binary notation corresponds to 10) and number 3 (that in binary notation corresponds to 11).

As stated above, the goal is to simulate a binary counter starting at zero. The process begins by executing the last rule, which rewrites the least significant bit 0 as 1. The computation then continues with the first rule; notice that, in general, the Turing machine is in state q while moving the head on the tape from left to right, and is in state r when moving the head from right to left. So the first rule changes all 1s, from left to right, to 0; when a 0 is encountered it is rewritten as 1, by means of the second rule. Such a rule changes the state of the Turing machine from q to r, which makes the head move on the tape – by the third rule – until the leftmost position. At this point, the fourth rule changes the state of the Turing machine back to q, and the entire process restarts.

Addition. One simple task that a P system should be able to tackle is performing addition in *unary*. That is, given a membrane containing n copies of an object a and m copies of an object b, the P system should be able to obtain $n + m$ copies of the object c. The P system of this benchmark has a simple membrane structure, with two nested membranes: $[[\ ]_k]_h$. For a problem of size n, membrane k will contain p objects of type a and $n - p$ objects of type b, where p is a randomly selected integer in $\{0, \ldots, n\}$. Two send-out rules can then be applied:

$$[a]_k \to [\ ]_k\ c \qquad\qquad [b]_k \to [\ ]_k\ c$$

After n applications of the two rules the final results, n copies of c in membrane h, will be obtained. Clearly this kind of unary addition does not require any kind of cooperation.

This task, together with the multiplication and division tasks, is inspired by the work of [18]. The resulting systems are not identical for one main reason: the kind of P systems employed in this work has slightly different rules allowed.

Multiplication. A slightly more complex task compared to unary addition is *unary multiplication*. The task is, in this case, a multiplication by a constant n. Thus, p objects of type a inside a membrane k must produce $p \times n$ objects of type b inside the parent membrane h of k. In this case the size of the problem is the constant n and the number of objects of type a initially present is a randomly selected number $k \in \{0, \ldots, n\}$. Multiplication can be achieved by only one send-out rule:

$$[a]_k \to [\ ]_k\ b^n$$

In this case, while only one rule need to be applied – compared to two in the case of unary addition – the rule has a larger RHS, making it more difficult to be found.

Division. The last task we are going to explore is *unary division* by a constant n. In this case p copies of object a are present inside membrane k in the initial configuration. The P system will halt with $\lfloor \frac{p}{n} \rfloor$ copies of b in the parent membrane h of k and $k \bmod n$ copies of a still in membrane k. In this case the value of p is randomly selected in $\{0, \ldots, n^2\}$. Division by a constant n can be achieved by a single *cooperative* send-out rule of the form:

$$[a^n]_k \to [\ ]_k\ b$$

Differently from unary multiplication this time the rule is cooperative and with a LHS growing with n.

Thus, with unary addition, multiplication, and division we cover the three cases of non-cooperative send-out with small RHS, non-cooperative send-out with a large RHS, and cooperative send-out with a large LHS.

4.2. Parameters settings

For all problems, we used the same number of fitness evaluations (10^6) which, due to the value of λ chosen (10), implied a bound of 10^5 generations. The parameters employed in the experiments are detailed in Table 1.

For all considered problems, we perform experiments both generating no cooperative rules, i.e., setting LHS and RHS to 1, both generating cooperative rules, i.e. assigning to LHS and RHS values greater than 1 (here experiments with both 2 and 3 will be performed).

In order to provide also a study of the impact of the hyperparameter setting on the problem of inferring the P system with EA, different values for the population size μ, for the maximum number of rules m and for the maximum number of rules changed by mutation q will be considered. In particular, the population size (μ) will range in the set $\{1, 5, 10\}$, while the value for the maximum number of rules (m) and for the maximum number of rules changed by mutation (q) will always be the same, and will take values in the set $\{10, 20\}$. Therefore, combining all these possible hyperparameter values, for each P system with fixed size n and with fixed LHS and RHS size, six different configurations will be studied.

Table 1
All settings employed in the experiments with the proposed $(1 + \lambda)$-EA. Notice that the Turing machine simulation problem has different settings, due to the fact that it requires cooperative rules.

Parameter	Value	Value (TM problem)
Maximum LHS size	1, 2, 3	2, 3
Maximum RHS size	1, 2, 3	2, 3
Minimum number of rules	1	1
Maximum number of rules (m)	10, 20	10, 20
Max. # of rules changed by mutation (q)	10, 20	10, 20
Population size (μ)	1, 5, 10	1, 5, 10
Number of offsprings (λ)	10	10
Number of generations	$\frac{10^5}{\mu}$	$\frac{10^5}{\mu}$

Moreover, as we require that for each different problem the number of fitness evaluations does not vary, to render fair the comparison between experiments with different structures, the number of epochs must be divided by the population size μ considered. Notice that there are two sets of parameters depending on the nature of the problem: for the send-in, send-out, and variable assignment problems we consider LHS and RHS of size 1 i.e., no cooperative rule can be generated, and of size 2 and 3, i.e., thus allowing the creation of cooperative rewriting rules. While, for the Turing machine simulation problem, only consider LHS and RHS equal to 2 and 3, as it requires cooperative rules.

Finally, since $(\mu + \lambda)$-EA is inherently stochastic, we performed 30 independent runs for each problem and problem size, thus allowing us to obtain a distribution of the fitness values of the best individuals with respect to the number of generations.

We recall that a fitness value of 0 denotes that the obtained P system behaves exactly as desired, i.e., that the evolutionary algorithm was able to learn a correct set of rules for the P system.

5. Results and discussion

The results of the experiments are presented in Figs. 2–5. Each figure shows, for every considered benchmark problem, the fitness distribution at last generation, obtained through the different settings of the hyperparameters, as one of the goals of this work is also to find the best hyperparameter for the evolutionary algorithm, both when dealing with cooperative and non-cooperative rules. Moreover, the last generation fitness of the best hyperparameter setting is plotted in Fig. 1, via box plot for all the considered dimensions of the problem and for all the considered dimensions of LHS and RHS. These box plots are, anyway, not displayed for arithmetical operations. In fact, as Fig. 4 suggests, the fitness is almost always equal to zero, then the distribution is not significant as (most of the times) all the 30 runs performed lead to the same identical result: a *P System* able to infer correctly the proposed benchmark problem.

5.1. Send-in and send-out problems

Fig. 2 shows that for both send-in and send-out problem, considering a higher value for the maximum number of rules and for the number of rules changed by mutation ($m = q = 20$) lead to better results for cooperative and for the non-cooperative scenario. On the contrary, considering a higher population size has as a consequence an increase of the fitness, meaning that the evolutionary algorithm is less precise in inferring the correct P system. Consequently, the best hyperparameter setting for the prediction is obtained with $m = q = 20$ and by $\mu = 1$. The box plot of the fitness for this framework is shown in Fig. 1(a) and Fig. 1(b). Considering the non-cooperative scenario (LHS = RHS = 1), for both the send-in and send-out problems, for sizes $n = 2$, $n = 3$ and $n = 4$ the evolutionary algorithm is always able to find a P system that reproduces the desired dynamics, while for size

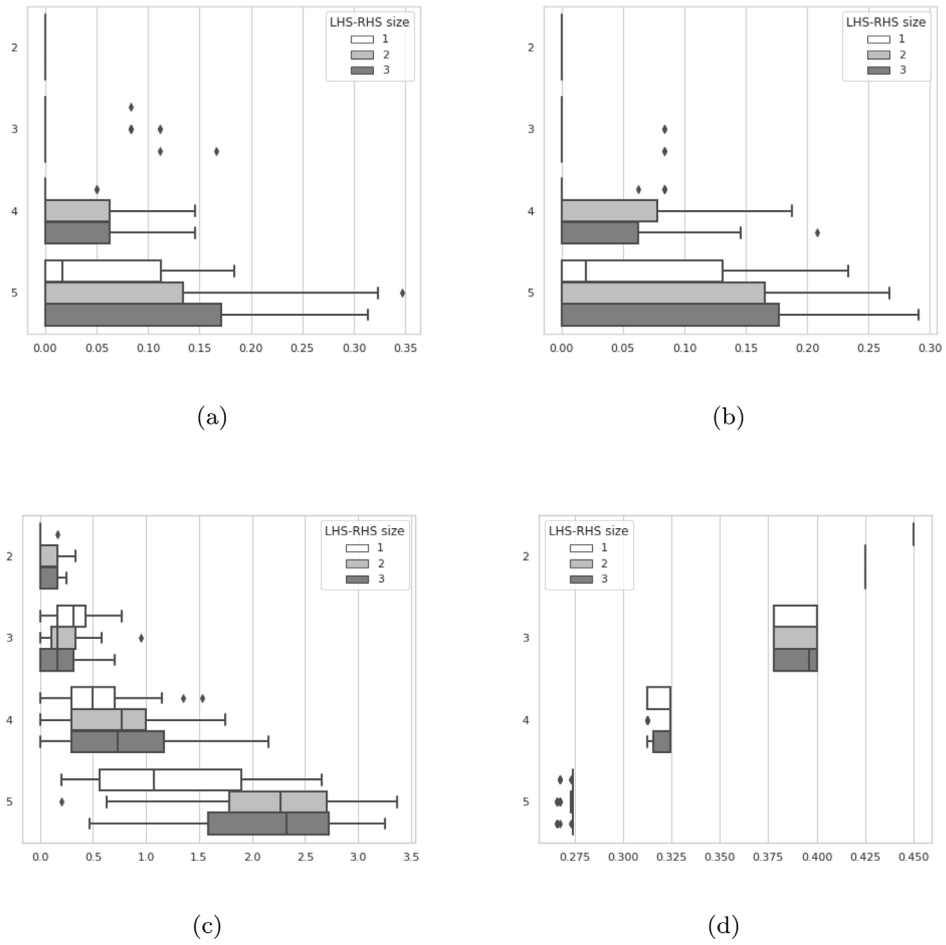


Fig. 1. Box plot of the fitness at the last generation, considering as hyperparameters $\mu = 1$ and $m = k = 20$, for the *send-in* problem (a), *send-out* problem (b), *assignment* problem (c) and *Turing machine* problem (d).

$n = 5$, the EA is still able to find the correct solution, but not all the times. Regarding cooperative problems (LHS = RHS = 2 or 3), again EA provides a P system able to reproduce the dynamics for $n = 2$ and $n = 3$. For $n = 4$ and $n = 5$ the correct solution is obtained in most of the run performed. To show an example of an individual (i.e., a P system) being generated by the evolutionary process, let us consider the following solution to the send-in problem of size 2:

$\mathbf{x_{1,0}}\ [\ ]_{\mathbf{k}} \rightarrow [\mathbf{x_1}]_{\mathbf{k}}$ $\quad$ $x_{1,1}\ [\ ]_h \rightarrow [x_0]_h$

$\mathbf{x_{0,0}}\ [\ ]_{\mathbf{k}} \rightarrow [\mathbf{x_0}]_{\mathbf{k}}$ $\quad$ $x_{0,0}\ [\ ]_k \rightarrow [x_{0,0}]_k$

$[\mathbf{x_1} \rightarrow \mathbf{x_1}]_{\mathbf{k}}$ $\quad$ $[\mathbf{x_{1,1}} \rightarrow \mathbf{x_{1,0}}]_{\mathbf{h}}$

$[x_{1,1} \rightarrow x_{1,0}]_h$ $\quad$ $x_{1,1}\ [\ ]_h \rightarrow [x_{0,0}]_h$

$[x_{0,0}]_h \rightarrow [\ ]_h\ x_0$ $\quad$ $[x_0]_h \rightarrow [x_1]_h\ [x_{0,0}]_h$

$x_{1,0}\ [\ ]_k \rightarrow [x_{0,0}]_k$

It is important to note that only the four rules in **bold** can actually be triggered during computation and, in particular, the rule $[x_1 \rightarrow x_1]_k$ has actually no effect on the configurations being generated, since it just rewrites the object x_1 in itself. All the other rules are either impossible to be a trigger (e.g., because they try to send-in an object from the environment into the skin membrane, or try to divide the skin membrane) or, while they could potentially be triggered, they are always superseded by other rules that occur earlier in the list of rules, and hence have a higher priority. This happens, for example, with the rule $x_{1,0}\ [\ ]_k \rightarrow [x_{0,0}]_k$ at the end of the left column, which is always superseded by the first rule in the same column, $x_{1,0}\ [\ ]_k \rightarrow [x_1]_k$ (here we are assuming that the rules are listed row by row, and in each row, the list contains first the rule in the left column, and then the rule in the right column).

5.2. Variable assignment problem

Fig. 3 shows that for the assignment problem, as for the send-in and the send-out problems, the best hyperparameter setting inferring the correct P system is given by $m = q = 20$ and by $\mu = 1$. Again, therefore, better results are achieved with higher values assigned to m and q and with a lower population size. Fig. 1(c) displays the relative box plot of the fitness, showing that the only case where the EA is capable to find a P system that generates always the correct dynamics is the non-cooperative framework for size $n = 2$. For size $n = 3$ and $n = 4$, we can observe that the fitness reach zero for some run, but there are also situations in which an optimal solution is not achieved. On the other hand, if the size considered is $n = 5$, the evolutionary algorithm is never able to find a correct solution.

Comparing these results with the previous ones, corresponding to the send-in and send-out problems, we can see that the value of the median is higher in the variable assignment problem having the same size and parameters. This behavior suggests that for our evolutionary algorithm it is more difficult to infer the rules when dealing with the variable assignment problem rather than for the send-in or the send-out problem.

5.3. Basic arithmetic operations

Fig. 4 shows that our method proves to be particularly successful in approximating *P Systems* framework for performing basic arithmetic operations. In fact, these benchmark problems are the ones that lead to the best results of all those considered. The median error over the

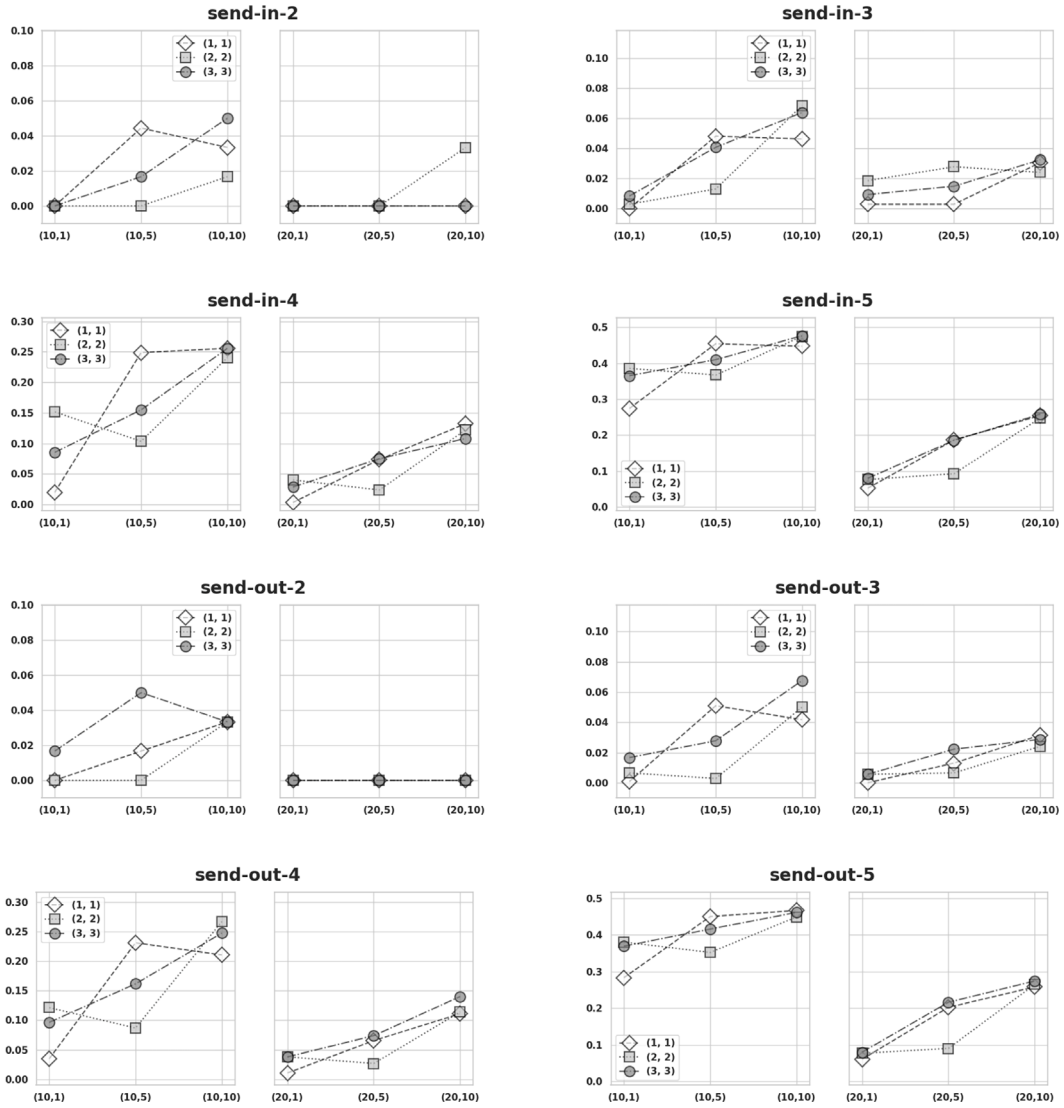


Fig. 2. Comparison of *send-in* and *send-out* problem fitness values for different hyperparameters setting combinations. The x axes contains the hyperparameter values considered, represented by a tuple where the first element represents m (always taken equal to q) and the second element represents μ.

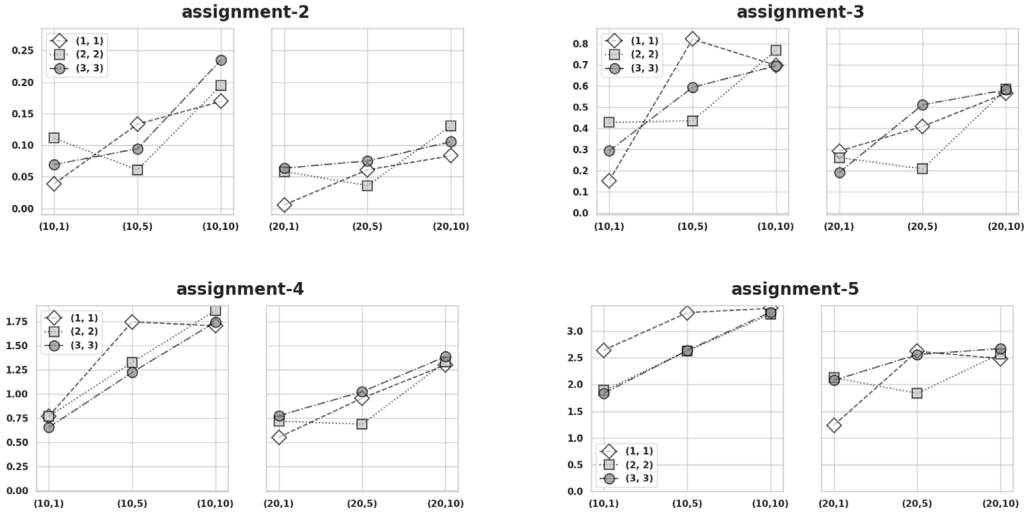


Fig. 3. Comparison of *assignment* problem fitness values for different hyperparameters setting combinations. The x axes contains the hyperparameter values considered, represented by a tuple where the first element represents m (always taken equal to q) and the second element represents μ.

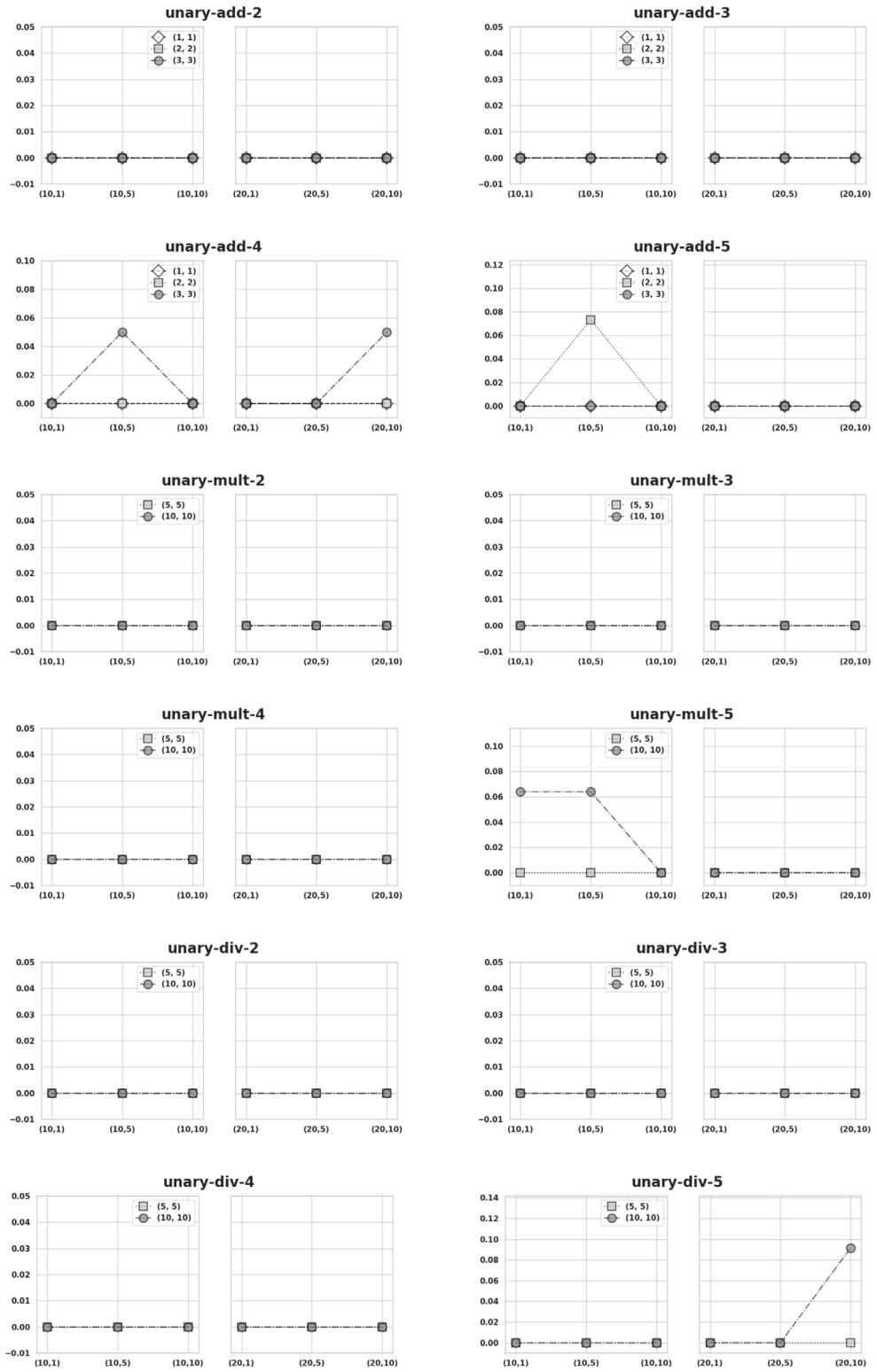


Fig. 4. Comparison of *addition*, *multiplication* and *division* problem fitness values for different hyperparameters setting combinations. The x axes contains the hyperparameter values considered, represented by a tuple where the first element represents m (always taken equal to q) and the second element represents μ.

30 runs considered is almost always equal to zero, thus our method is capable of precisely inferring the *P System* most of the times. Here, no combinations of hyperparameters emerge that lead to better solutions than others, but all lead to excellent results. We would also underline that experimental results shows convergence to the solutions for the EA algorithm after just a few thousand generations.

5.4. Turing machine simulation problem

Fig. 5 shows a different behavior with respect to the previous benchmark problems analyzed above. In fact here, for a fixed dimension of the problem, every choice of the hyperparameters leads to the same fitness values. In Fig. 1(d), we display the box plot for the $m = q = 20$ and by $\mu = 1$, to be consistent with the box plot of the previously

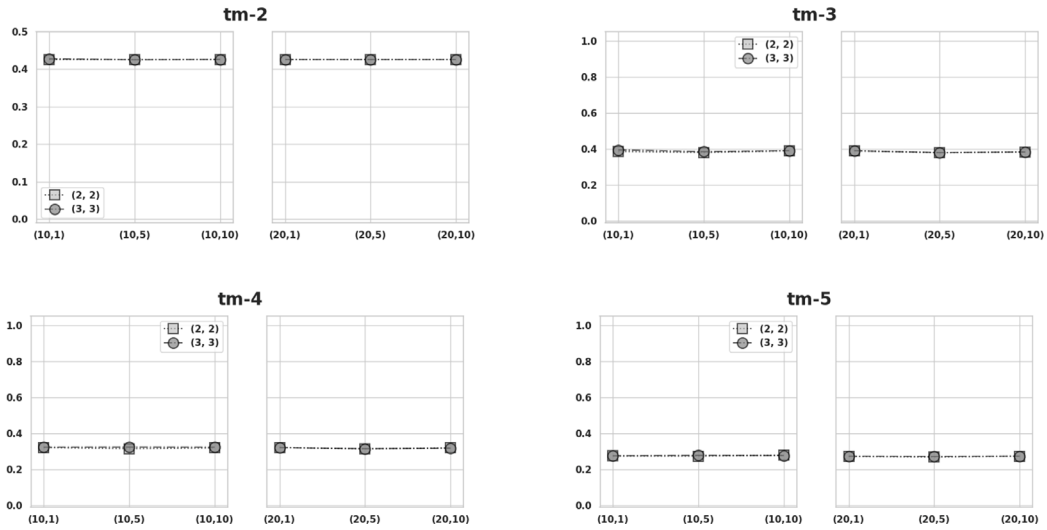


Fig. 5. Comparison of *Turing machine* problem fitness values for different hyperparameters setting combinations. The x axes contains the hyperparameter values considered, represented by a tuple where the first element represents m (always taken equal to q) and the second element represents μ.

considered problems. Here, the fitness seems to decrease when the size of the problem increases, contrary to what has been observed in send-in, send-out, and assignment problems. Anyway, our EA is not capable to find a P system that generates the correct dynamic for none of the sizes considered for the training. For n equal to 1 and 5, we can also observe that, for all the runs performed, at the last generation of the training the output fitness value is the same, except in very rare cases. This is clear if we look at the box plots corresponding to the mentioned size values; indeed, in these cases, the median value corresponds both to the minimum and to the maximum values, and the box plot is represented by a single line.

6. Conclusions and directions for future work

In this paper, we have defined a new way of performing automatic synthesis of P systems via an evolutionary-based approach. To assess the ability of the proposed $(\mu + \lambda)$-EA to perform this task, we defined different parametric P systems to be used as benchmark problems, focusing on one particular feature to test: learning send-in or send-out rules, performing membrane division, synthesis of cooperative multiset rewriting rules, and learning how to perform basic arithmetic operations.

We performed computational experiments for P systems of different sizes for each benchmark problem and for each different combinations of hyperparameters settings. In each experiment, a random P system of the prescribed size was built; by simulating it, it has been used to generate a list of input–output pairs for the $(\mu+\lambda)$-EA, each representing two consecutive configurations of the P system. This list of pairs has then been used as the input dataset for the evolutionary algorithm, to let it try to learn a set of P system rules that are able to replicate the consecutive configurations of the original P system. The results suggest that basic arithmetic operations can be inferred very accurately for each hyperparameter setting selected, while send-in, send-out, and membrane division rules can be learned — even if with some limitations. Cooperative multiset rewriting rules, instead, are much harder to learn: the exact dynamics of our Turing machine simulations were not reproduced, and in general, the results obtained from the experiments shows that further investigations are required.

Indeed, the work presented here has multiple possible future developments. First of all, it is necessary to devise a mutation operator or a generation procedure which are able to efficiently produce cooperative rewriting rules; as stated above, we see this as the main weakness of the current approach. Furthermore, it will be useful to test multiple strategies to remove bloat, i.e., rules that can never be activated and do not contribute to the dynamics of the learned P system. Finally, it could be interesting to explore the possibility of evolving nondeterministic systems, with the additional challenge that even two identical sets of rules might produce different computations.

CRediT authorship contribution statement

Alberto Leporati: Conceptualization, Writing – review & editing, Validation. **Luca Manzoni:** Conceptualization, Software, Writing – original draft, Supervision. **Giancarlo Mauri:** Writing – review & editing, Validation. **Gloria Pietropolli:** Conceptualization, Software, Writing – original draft, Investigation, Visualization. **Claudio Zandron:** Conceptualization, Writing – review & editing, Validation.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

Data will be made available on request.

References

[1] G. Păun, Computing with membranes, J. Comput. System Sci. 61 (1) (2000) 108–143, URL https://doi.org/10.1006/jcss.1999.1693.
[2] C. Martín-Vide, G. Păun, J. Pazos, A. Rodríguez-Patón, Tissue P systems, Theoret. Comput. Sci. 296 (2) (2003) 295–326, URL https://doi.org/10.1016/S0304-3975(02)00659-X.
[3] M. Ionescu, G. Păun, T. Yokomori, Spiking Neural P systems, Fund. Inform. 71 (2–3) (2006) 279–308, URL https://content.iospress.com/articles/fundamenta-informaticae/fi71-2-3-08.
[4] G. Păun, G. Rozenberg, A guide to membrane computing, Theoret. Comput. Sci. 287 (1) (2002) 73–100, URL https://doi.org/10.1016/S0304-3975(02)00136-6.
[5] G. Păun, Membrane Computing: An Introduction, Springer, 2002.
[6] G. Păun, G. Rozenberg, A. Salomaa (Eds.), The Oxford Handbook of Membrane Computing, Oxford University Press, 2010.
[7] The P systems webpage, http://ppage.psystems.eu/.
[8] G. Păun, Introduction to membrane computing, in: Applications of Membrane Computing, Springer, 2006, pp. 1–42.
[9] G. Păun, A quick introduction to membrane computing, J. Log. Algebr. Program. 79 (6) (2010) 291–294.

[10] M. Zhu, Q. Yang, J. Dong, G. Zhang, X. Gou, H. Rong, P. Paul, F. Neri, An adaptive optimization spiking neural P system for binary problems, Int. J. Neural Syst. 31 (01) (2021) 2050054.
[11] G. Zhang, H. Rong, P. Paul, Y. He, F. Neri, M.J. Pérez-Jiménez, A complete arithmetic calculator constructed from spiking neural P systems and its application to information fusion, Int. J. Neural Syst. 31 (01) (2021) 2050055.
[12] X. Wang, G. Zhang, X. Gou, P. Paul, F. Neri, H. Rong, Q. Yang, H. Zhang, Multi-behaviors coordination controller design with enzymatic numerical P systems for robots, Integr. Comput.-Aided Eng. 28 (2) (2021) 119–140.
[13] G. Păun, P systems with active membranes: Attacking NP-complete problems, J. Autom. Lang. Comb. 6 (1) (2001) 75–90.
[14] G. Zhang, M. Gheorghe, L. Pan, M.J. Pérez-Jiménez, Evolutionary membrane computing: A comprehensive survey and new results, Inform. Sci. 279 (2014) 528–551, http://dx.doi.org/10.1016/j.ins.2014.04.007, URL https://www.sciencedirect.com/science/article/pii/S002002551400454X.
[15] G. Escuela, M.Á. Gutiérrez-Naranjo, An application of genetic algorithms to membrane computing, in: Proceedings of the Eighth Brainstorming Week on Membrane Computing, 101-108. Sevilla, ETS de Ingeniería Informática, 1-5 de Febrero, 2010, Fénix Editora, 2010.
[16] X. Huang, G. Zhang, H. Rong, F. Ipate, Evolutionary design of a simple membrane system, in: International Conference on Membrane Computing, Springer, 2011, pp. 203–214.
[17] C. Tudose, R. Lefticaru, F. Ipate, Using genetic algorithms and model checking for P systems automatic design, in: Nature Inspired Cooperative Strategies for Optimization (NICSO 2011), Springer, 2011, pp. 285–302.
[18] Y. Chen, G. Zhang, T. Wang, X. Huang, Automatic design of P systems for five basic arithmetic operations within one framework, Chin. J. Electron. 23 (2) (2014) 302–304.
[19] Z. Ou, G. Zhang, T. Wang, X. Huang, Automatic design of cell-like P systems through tuning membrane structures, initial objects and evolution rules, Int. J. Unconv. Comput. 9 (5–6) (2013) 425–443.
[20] G. Zhang, H. Rong, Z. Ou, M.J. Pérez-Jiménez, M. Gheorghe, Automatic design of deterministic and non-halting membrane systems by tuning syntactical ingredients, IEEE Trans. Nanobiosci. 13 (3) (2014) 363–371.
[21] M.J. Pérez-Jiménez, Á. Romero-Jiménez, F. Sancho-Caparrini, Complexity classes in models of cellular computing with membranes, Nat. Comput. 2 (3) (2003) 265–284, URL https://doi.org/10.1023/A:1025449224520.
[22] I. Rechenberg, Evolutionsstrategie: Optimierung Technischer Systeme nach Prinzipien der Biologischen Evolution, Fromman-Holzbook, Stuttgart, Germany, 1973.
[23] S. Luke, Essential of Metaheuristics, second ed., Lulu, 2013.
[24] G. Pietropolli, Psystem-GA, 2022, https://github.com/gpietrop/Psystem-GA.
[25] R. Herken, The Universal Turing Machine: A Half-Century Survey, Springer, 1988.