Journal of Membrane Computing (2023) 5:129–143
https://doi.org/10.1007/s41965-023-00125-w

RESEARCH PAPER

A grammatical evolution approach to the automatic inference of P systems

Giorgia Nadizar[1] · **Gloria Pietropolli**[1,2]

Received: 28 February 2023 / Accepted: 17 June 2023 / Published online: 10 August 2023

Abstract

P systems are a bio-inspired framework for defining parallel models of computation. Despite their relevance for both theoretical and application scenarios, the design and the identification of P systems remain tedious and demanding tasks, requiring considerable time and expertise. In this work, we try to address these problems by proposing an automated methodology based on grammatical evolution (GE)—an evolutionary computation technique—which does not require any domain knowledge. We consider a setting where observations of successive configurations of a P system are available, and we rely on GE for automatically inferring the P system, i.e., its ruleset. Such approach directly addresses the identification problem, but it can also be employed for automated design, requiring the designer to simply express the configurations of the P system rather than its full ruleset. We assess the practicability of the proposed method on six problems of various difficulties and evaluate its behavior in terms of inference capability and time consumption. Experimental results confirm our approach is a viable strategy for small problem sizes, where it achieves perfect inference in a few seconds without any human intervention. Moreover, we also obtain promising results for larger problem sizes in a human-aided context, paving the way for fully or partially automated design of P systems.

Keywords P system · Membrane computing · Grammatical evolution · Evolutionary algorithms

1 Introduction

Membrane computing (MC), also called membrane systems or P systems, is a parallel computational model inspired by the functioning of living cell membranes [1]. MC is a widely investigated interdisciplinary field of research, because of its main characteristics, including the non-determinism of computations, maximal parallelism, and locality of interactions [2]. In fact, especially in recent years, MC has been successfully applied in various fields, such as theoretical computer science, parallel and distributed algorithms, graphics, linguistics, cryptography, economy [3], and, more recently, in systems and synthetic biology [4]. For a comprehensive review of the principal concepts of MC, together with examples of research trends in this area, we refer the reader to [5].

Handcrafting a P system for a given application is, however, a non-trivial and time-consuming task, requiring precision and expertise. Oftentimes, the designer has an approximate idea of the membrane structure, initial multisets, and set of rules necessary to describe the P system; however, undesired consequences can arise from small mistakes in the description of the initial configuration or in the set of rules [6]. As a matter of fact, specifying desired subsequent configurations of a P system is much easier than directly engineering its set of rules. Hence, a convenient approach relies on automatic inference techniques, which aim at inferring the structure of the P system from its successive configurations. Several methods have been proposed in the literature, among which we find *automated design of membrane computing models*, a sub-field of *evolutionary membrane computing* [7], which exploits evolutionary computation (EC) for tackling the programmability issue of P systems.

Giorgia Nadizar and Gloria Pietropolli have contributed equally to this work.

✉ Giorgia Nadizar
giorgia.nadizar@phd.units.it

✉ Gloria Pietropolli
gloria.pietropolli@phd.units.it

1 Department of Mathematics and Geosciences, University of Trieste, Trieste, Italy

2 National Institute of Oceanography and Applied Geophysics, OGS, Sgonico, Italy

EC is a natural computing technique based on principles of biological evolution that consists of a wide range of problem-solving optimization methods. The principal evolutionary paradigms are: genetic algorithm (GA) [8], genetic programming (GP) [9], evolution strategies (ES) [10], grammatical evolution (GE) [11], particle swarm optimization (PSO) [12], quantum-inspired evolutionary algorithm (QIEA) [13], and many more.

In this work, we follow along with the idea of making use of EC for the identification task, relying on GE [14]—a branch of EC dealing with the automated synthesis of programs, or, more broadly, strings, of a language defined by a context-free grammar (CFG). We resort to GE for its demonstrated capabilities of inferring underlying structures of systems whose functioning can be expressed with a CFG [11]. To this end, we introduce a CFG to describe the rules of a P system, and we apply GE as an inferring tool, given a set of successive configurations of a P system.

In the literature, an intersection between P systems and GE has already been investigated in the work of Nishida et al. [15], where the authors propose a tissue evolutionary P system that evolves a CFG to generate a given target language. However, to the best of our knowledge, this is the first attempt to infer the P system rules by means of GE.

The strength of our proposed GE-based method is that—unlike those previously investigated in the literature [16]—it does not require the user any specific knowledge of the P system under analysis, meaning that the discovery is completely left to the evolutionary search. This derives from the fact that the CFG used does not impose constraints on either the size of the ruleset, or the size of the multisets, or the type of rules that shall be used.

We validate our approach by testing the inference ability on three benchmark problems (send-in, send-out, and variable assignment) and three basic arithmetic operations (addition, multiplication, and division). Experimental results show that our approach is able to always achieve correct and fast inference on all arithmetic operations, and on the benchmark problems with small problem size. Moreover, we manage to partially overcome the scaling limitation by leveraging some knowledge about the system, which could be attained with limited human intervention.

The remainder of the manuscript is organized as follows. Section 2 links this study to the existing literature. Thereafter, Sect. 3 reviews some background concepts, specifically introducing P systems and GE; while Sect. 4 presents the proposed GE-based inference method. We detail the experimental settings together with a description of the benchmark problems used to assess the validity of our approach in Sect. 5, and present and comment the results achieved in Sect. 6. Finally, we summarize the main findings of the paper and suggest future research directions in Sect. 7.

2 Related works

The combination of MC and EC is receiving growing attention in the last few years. On one side, MC has the rigor and sound theoretical development, as well as it provides a parallel distributed framework. On the other side, EC has outstanding characteristics, such as easy understanding, robust performance, flexibility, convenience of use for real-world problems, and a very large scope of applications. These features strongly encourage the exploration of the interactions between MC and EC: this branch of research falls under the name of *Evolutionary Membrane Computing* (EMC) [7].

More in detail, we can identify two main research lines regarding the interplay of MC and EC: *membrane-inspired evolutionary algorithms* (MIEAs) and *automated design of membrane computing models* (ADMCM). MIEA, originally named *membrane algorithms* (MA) [17], consists of meta-heuristic algorithms where P system is used as a part of an evolutionary algorithm. These methods have been successfully applied in many real-world problems ranging from optimization to engineering design and machine learning [2]. Conversely, ADMCM is envisaged to obtain the automated synthesis of membrane computing models or of a high-level specification of them by applying various meta-heuristic search methods. Specifically, among the interest areas covered by ADMCM, emerges the *automatic design of P systems* (ADPS), which is also the research field to which our investigation belongs.

The first attempt at correctly inferring a P system can be found in [6], where the authors proposed *PSystemEvolver*, an evolutionary algorithm based on generative encoding, and tested it with a simple mathematical problem: the computation of squared numbers n^2. The same mathematical problem was later tackled in [18], where the authors proposed a binary encoding for representing a P system, and exploited a QIEA for evolving the P system population. Also Tudose et al. [19] focused on the computation of squared numbers employing a binary encoding and GA for the evolutionary part. Yet, this work differs from the previous ones as it encompasses non-determinism and model checking. Namely, multiple non-deterministic simulations are performed for the fitness evaluations, which are followed by formal verification of the model. Moving towards a more general framework, Ou et al. [20] still relied on an elitist GA for the same mathematical problem (actually, they considered only the square of 4), but introduced a method for automatically designing a cell-like membrane system by tuning membrane structures, initial objects, and evolution rules. On the other hand, Chen et al. [21] aimed at generalization in terms of operations to be simulated within the same framework, considering five arithmetic operations—addition, subtraction, multiplication, division, and power. To this end, the authors leveraged a QIEA, yet they considered only a predefined membrane structure and fixed initial objects. Along the same line, in [7] the authors, besides surveying the state of art, also proposed an automatic design for the computation of various arithmetic operations ($2(n-1), 2n-1, n^2, \frac{1}{2}(n(n-1)), n(n-1), (n-1)^2+2^n+2$, $a^{2^n}b^{3^n}$ and $\frac{1}{2}(3^n-1)$).

The most recent work devoted to the automatic identification of P systems with an evolutionary approach is [16]. Therein, the authors presented an evolutionary algorithm that uses a direct encoding to evolve a population of (initially random) P systems with active membranes and four types of cooperative rules. The inference process only requires the observation of subsequent configurations of the P system, and relies on a fitness defined as edit distance between membranes [22], similar to that employed in [23]. The authors tested their method on several benchmark problems, including three arithmetical operations. This work is the most similar to our proposition, for we share the same general setting and fitness measure. However, there are a few key differences that distinguish our approach and add significant novelty. First, we rely on an indirect encoding for P systems, which is based on an easily customizable CFG. Second, we do not require the user to have any specific knowledge of the system under examination, as we do not pose any constraints on either the size of the ruleset, of the multisets or on the types of rules to be employed. Last, we provide an estimate of the time needed for the automatic inference, to evaluate its practicability in a real-world scenario.

3 Background

In this section, we formally define P systems (in Sect. 3.1) and we introduce grammatical evolution (in Sect. 3.2), the evolutionary computation technique we exploit for inferring them.

3.1 P systems

Membrane systems, also referred to as P systems, consist of a framework for defining parallel models of computation, inspired by some basic features of biological membranes. In P systems, multisets of objects are located in compartments, i.e., in *membrane structures*, which evolve thanks to *rewriting rules* associated with specific compartments, applied in a maximally parallel, non-deterministic way [1]. In this paper, we consider P systems with active membranes without electrical charges.

Definition 1 A P system with active membranes and cooperative rules, of initial degree $d \geq 1$, is a tuple $\Pi = (\Gamma, \Lambda, \mu, w_{h_1}, \dots, w_{h_d}, R)$, where

- Γ is an alphabet of symbols, called *objects*,
- Λ is a finite set of labels,
- μ is a membrane structure (represented as a rooted unordered tree) consisting of d membranes labeled by elements of Λ,
- $w_{h_1}, \dots, w_{h_d}$, with $h_1, \dots, h_d \in \Lambda$, are multisets describing the initial contents of each of the d regions of μ, and
- R is a finite set of rules.

We consider the following 4 types of cooperative rules for the set R, taking inspiration from [16].

- *Cooperative rewriting rules*, $[\, u \to v \,]_h$ for $h \in \Lambda$ and $u \in \Gamma^+$, $v \in \Gamma^\star$: when applied inside a membrane h prescribe the substitution of all objects in u with the objects in v.
- *Cooperative communication send-in rules*, $u\,[\;]_h \to [\, v \,]_h$ for $h \in \Lambda$ and $u \in \Gamma^+, v \in \Gamma^\star$: when applied to a membrane h prescribe the removal of the objects in u present in the parent region, and the insertion of the objects in v in the membrane h.
- *Cooperative communication send-out rules*, $[\, u \,]_h \to v\,[\;]_h$ for $h \in \Lambda$ and $u \in \Gamma^+, v \in \Gamma^\star$): when applied inside a membrane h prescribe the removal of all objects in u from its content, and the addition of all objects in v to the parent region.
- *Cooperative weak division rules*, $[\, u \,]_h \to [\, v \,]_h\,[\, w \,]_h$ for $h \in \Lambda$ and $u \in \Gamma^+, v, w \in \Gamma^\star$, when applied to a membrane h prescribe its replication (together with its content) into two separate membranes, both labeled with h. The objects of u are removed from both newly created membranes, and replaced by those in v for the first one, and by those in w for the second one.

We recall these rules are *cooperative* as they are triggered by the co-occurrence of multiple objects within a membrane, i.e., they act on multisets rather than on single objects. We call left-hand side (LHS) of the rule the multiset it acts on, (i.e., u for the rules listed here) and right-hand side (RHS) of the rule the multiset(s) being produced (i.e., v and w).

At each time step i, a membrane h is in a configuration defined as the multiset of objects it contains. Hence, the *configuration of a P system* Π at i, $\mathscr{C}_i$, is given by its membrane structure μ, i.e., by the configuration of all its membranes, at i. Such configuration can be changed by the application of the rules in R, i.e., by a *computation step*, which leads to a new configuration $\mathscr{C}_{i+1}$.

A computation step changes the current configuration of the P system according to a series of principles, specifically:

(1) Each object is subject to at most one rule per step, yet more objects can cross membranes at each step;
(2) The application of rules is *maximally parallel*;

(3) A non-deterministic choice is performed if more than one rule can be applied at the same time;
(4) Rules are applied simultaneously in an atomic way;
(5) The outermost membrane cannot be divided and any object sent out from it can not re-enter the system again.

However, in this paper, we will consider solely deterministic P systems, meaning that we enforce determinism in the application of the rules, differently from what principle (3) states. Hence, there exists a unique computation $\mathscr{C}_0, \mathscr{C}_1, \ldots$ starting from the initial configuration $\mathscr{C}_0$.

3.2 Grammatical evolution

Evolutionary algorithms (EAs) [24], among which we find grammatical evolution (GE), are a class of population-based optimization algorithms. As the name suggests, EAs take strong inspiration from Darwin's theory of evolution [25], in that they *evolve* a *population*, i.e., a multiset, of candidate solutions for a problem. Similarly to what happens in biological evolution, solutions are encoded with a *genotype*. To compute the actual solution, i.e., the *phenotype*, we define a genotype–phenotype *mapping*. At each *generation*, i.e., at each iteration of the evolutionary loop, we generate new individuals by means of mutation and recombination, we evaluate their *fitness*, i.e., their quality, and we retain some individuals according to a certain criterion.

GE is a variant of genetic programming [9], which can evolve programs in any language. Its strength lies in its indirect encoding and in a mapping procedure that solves the "closure" problem [11], always yielding valid solutions. More in detail, the genotype in GE consists of a variable-length bit string, where bits are grouped into *codons*—consecutive groups of 8 bits, which can be converted into integer values in $[0, 2^8 - 1 = 255]$. Conversely, the phenotype can be a program, i.e., a string, in any language $\mathcal{L}(\mathcal{G})$ that can be specified with a Context-Free Grammar (CGF) $\mathcal{G}$ (more in the following). Thanks to this dichotomy, the end-user can rely on standard techniques for the evolutionary loop, and obtain a solution to their problem by simply specifying a suitable grammar $\mathcal{G}$.

We provide the pseudo-code for evolving a solution with GE in Algorithm 1. After initializing a population of n_{pop} individuals, the evolutionary search proceeds iteratively for n_{gen} generations. At each generation, we generate n_{off} new individuals, by first selecting two parents from the population P, and then applying crossover and mutation to their genotypes. From the newly obtained genotypes, we compute the individuals with a mapping procedure followed by the fitness evaluation (see Lines 10 and 11). Last, we merge the offspring with an elite of the current population, to constitute the population for the next evolutionary loop. Concerning the operators involved, namely initialization, mutation, and crossover, it is in principle possible to use any operator suitable for a bit string genotype.

```
⟨expr⟩ ::= ( ⟨expr⟩ ⟨op⟩ ⟨expr⟩ ) | ⟨var⟩ | ⟨num⟩
⟨op⟩ ::= + | - | * | /
⟨var⟩ ::= x | y
⟨num⟩ ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

Fig. 1 A CFG in the Backus–Naur Form (BNF) for mathematical expressions

Algorithm 1 The evolutionary loop of GE.

```
1: function EVOLVE( )
2:    $\mathcal{I} \leftarrow$ INITIALIZE($n_{\text{pop}}$)
3:    for all $g \in \{1, \ldots, n_{\text{gen}}\}$ do
4:       $C \leftarrow \emptyset$
5:       for all $i \in \{1, \ldots, \frac{n_{\text{off}}}{2}\}$ do
6:          $g_1 \leftarrow$ SELECTPARENT($\mathcal{I}$)
7:          $g_2 \leftarrow$ SELECTPARENT($\mathcal{I}$)
8:          $g'_1, g'_2 \leftarrow$ CROSSOVER($g_1, g_2$)
9:          $g''_1, g''_2 \leftarrow$ MUTATE($g'_1$), MUTATE($g'_2$)
10:         $p_1, p_2 \leftarrow$ MAP($g''_1$), MAP($g''_2$)
11:         $f_1, f_2 \leftarrow$ EVALUATE($p''_1$), EVALUATE($p''_2$)
12:         $i_1, i_2 \leftarrow (g_1, p_1, f_1), (g_2, p_2, f_2)$
13:         $\mathcal{I}' \leftarrow \mathcal{I}' \cup \{i_1, i_2\}$
14:      end for
15:      $\mathcal{I} \leftarrow \mathcal{I}' \cup$ BEST($\mathcal{I}, n_{\text{pop}} - n_{\text{off}}$)
16:   end for
17:   return BEST($\mathcal{I}$, 1)
18: end function
```

The core of GE, however, does not lie in its rather standard evolutionary loop. Instead, it resides within the CFG, which specifies the syntax of the desired solution, and in the mapping, procedure based on it.

Definition 2 A context-free grammar (CFG) is defined as a tuple $\mathcal{G} = (N, T, s_0, P)$, where

- N is the set of *non-terminal* symbols,
- T is the set of *terminal* symbols (with $N \cap T = \emptyset$),
- $s_0 \in N$ is the starting symbol, and
- P is the set of *production rules*.

The production rules P, usually expressed in the Backus–Naur Form (BNF) [26], constitute the essence of GE, for the genotype-phenotype mapping is built upon them. We provide an example of CFG for mathematical expressions in Fig. 1, with $N = \{\langle\text{expr}\rangle, \langle\text{op}\rangle, \langle\text{var}\rangle, \langle\text{num}\rangle\}$ and $T = \{+, -, *, /, x, y, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$. Following the usual convention, the starting symbol corresponds to the non-terminal on the left side of the first rule, i.e., $s_0 = \langle\text{expr}\rangle$.

Moving back to GE, we detail the genotype–phenotype mapping procedure in Algorithm 2. The rationale of the mapping is to begin from the starting symbol s_0, and then iteratively replace non-terminal symbols with other symbols according to the production rules, until the string is only composed of terminal symbols. To disambiguate among the available replacement options P_s for the considered symbol s, we rely on the current codon $g[c]$. Namely, we divide the integer value of $g[c]$ by $|P_s|$, and use the remainder of the division j to select the new symbol from P_s (with zero-based indexing). Clearly, we slide the genotype g for selecting the current codon, possibly restarting from the beginning if needed.

Note that in some edge cases an incomplete mapping might occur if the genome has been completely traversed (even with multiple restarts from the beginning) and the derivation string still contains non-terminals. In such cases (not reported in Algorithms 1 and 2 for brevity), we consider the individual as invalid and we assign it a poor fitness value to minimize its chances of propagation in the next generations.

Algorithm 2 Genotype-phenotype mapping in GE.

```
1:  function MAP($g$)
2:      $p \leftarrow s_0$
3:      $c \leftarrow 0$
4:      while CONTAINSNT($p$) do
5:          $s \leftarrow$ GETFIRSTNT($p$)
6:          $P_s \leftarrow$ GETPRODUCTIONRULES($s$)
7:          $j \leftarrow$ MOD($g[c], |P_s|$)
8:          $r \leftarrow P_s[j]$
9:          $p \leftarrow$ REPLACEFIRST($p, s, r$)
10:         $c \leftarrow c + 1$
11:         if $c \geq$ SIZE($g$) then
12:             $c \leftarrow 0$
13:         end if
14:     end while
15:     return $p$
16: end function
```

The mapping used in GE ensures that all produced valid individuals conform to the used CFG. We provide an example of mapping in Fig. 2 for a genotype g of 48 bits (6 codons), using the CFG of Fig. 1.

$$\begin{aligned} g &= 11100111\ 11110000\ 10100001 \\ &\quad\ 01110001\ 01001101\ 00000111 && \text{(bits)} \\ &= 231\ 15\ 133\ 142\ 178\ 224 && \text{(integers)} \end{aligned}$$

c	$g[c]$	$\|P_s\|$	j	Phenotype p
0	231	3	0	⟨**expr**⟩
1	15	3	0	(⟨**expr**⟩ ⟨op⟩ ⟨expr⟩)
2	133	3	1	((⟨**expr**⟩ ⟨op⟩ ⟨expr⟩) ⟨op⟩ ⟨expr⟩)
3	142	10	2	((⟨**num**⟩ ⟨op⟩ ⟨expr⟩) ⟨op⟩ ⟨expr⟩)
4	178	4	2	((2 ⟨**op**⟩ ⟨expr⟩) ⟨op⟩ ⟨expr⟩)
5	224	3	2	((2 * ⟨**expr**⟩) ⟨op⟩ ⟨expr⟩)
0	231	2	1	((2 * ⟨**var**⟩) ⟨op⟩ ⟨expr⟩)
1	15	4	3	((2 * y) ⟨**op**⟩ ⟨expr⟩)
2	133	3	1	((2 * y) / ⟨**expr**⟩)
3	142	10	2	((2 * y) / ⟨**num**⟩)
				((2 * y) / 2)

Fig. 2 Steps of the GE mapping procedure with a genotype g of 48 bits, i.e., 6 codons, and the grammar of Fig. 1. The rightmost column shows the phenotype p before the derivation of the highlighted non-terminal

```
⟨ruleset⟩ ::= ⟨ruleset⟩∪{⟨rule⟩} | {⟨rule⟩}
⟨rule⟩ ::= ⟨membrane⟩⟨multiset⟩ rewrite⟨multiset⟩ |
           ⟨membrane⟩⟨multiset⟩ sendin⟨multiset⟩ |
           ⟨membrane⟩⟨multiset⟩ sendout⟨multiset⟩ |
           ⟨membrane⟩⟨multiset⟩ division⟨multiset⟩⟨multiset⟩
⟨multiset⟩ ::= {⟨object⟩}∪⟨multiset⟩ | {⟨object⟩}
⟨membrane⟩ ::= m1 | ... | mΛ
⟨object⟩ ::= o1 | ... | oΓ
```

Fig. 3 The proposed CFG in BNF for describing the ruleset of a P system

4 Inferring P systems with GE

The goal of this work is to leverage GE to automatically infer a P system, i.e., its ruleset, given an observation of its successive configurations. Namely, we observe a sequence of configurations of a P system, $\overrightarrow{\mathscr{C}} = (\mathscr{C}_0, \ldots, \mathscr{C}_m)$ of length $m + 1$, and we aim at applying GE to find the ruleset whose application would give rise to said sequence.

As GE is able to evolve a string in any language, as long as its syntax can be specified with a CFG, we need to introduce a CFG to express the ruleset of a P system. Moreover, we need to define a quality measure of a candidate solution, i.e., the fitness of a ruleset, to guide GE towards the identification of the correct P system. We devote Sects. 4.1 and 4.2 to the definition of a CFG and the specification of a fitness measure, respectively.

4.1 A CFG for P systems rulesets

In compliance with the P system features described in Sect. 3.1, we propose the CFG reported in Fig. 3, with $N = \{$⟨ruleset⟩, ⟨rule⟩, ⟨multiset⟩, ⟨membrane⟩, ⟨object⟩$\}$, $T = \{$m1, ..., mΛ, o1, ..., o$\Gamma\}$, and $s_0 =$ ⟨ruleset⟩. We leave the number of membranes and objects as free parameters of the CFG, as they can easily be set on-the-fly before the inference, according to those involved in the observed P system.

As we can see in Fig. 3, a ruleset can either be comprised of a single rule or of multiple rules, each of which can be of one of the 4 types listed in Sect. 3.1. For each rule we require to select a membrane on which to apply it and the multisets, either 2 or 3, it operates on. However, we remark that the

$$
\begin{aligned}
g &= 11100111\ 00011000\ 01110001\ 00010001\ 00100011 \\
&\quad 10010000\ 11100111\ 00110111\ 00001100\ 00000110 \qquad \text{(bits)} \\
&= 231\ 24\ 113\ 17\ 35\ 144\ 231\ 55\ 12\ 6 \qquad \text{(integers)}
\end{aligned}
$$

c	$g[c]$	$\lvert P_s\rvert$	j	Phenotype p
0	231	2	1	⟨**ruleset**⟩
1	24	4	0	⟨**rule**⟩
2	113	10	3	⟨**membrane**⟩⟨multiset⟩rewrite⟨multiset⟩
3	17	2	1	m4⟨**multiset**⟩rewrite⟨multiset⟩
4	35	10	5	m4{⟨**object**⟩}rewrite⟨multiset⟩
5	144	2	0	m4 {o6} rewrite ⟨**multiset**⟩
6	231	10	1	m4 {o6} rewrite {⟨**object**⟩}∪⟨multiset⟩
7	55	2	1	m4 {o6} rewrite {o2} ∪⟨**multiset**⟩
8	12	10	2	m4 {o6} rewrite {o2} ∪{⟨**object**⟩}
				m4 {o6} rewrite {o2} ∪ {o3}
				$[\{o_6\} \to \{o_2, o_3\}]_{m_4}$

Fig. 4 Steps of the GE mapping procedure with a genotype g of 80 bits, i.e., 10 codons, and the grammar of Fig. 3. For this example, we set both Λ and Γ to 10. As for the previous example, the rightmost column shows the phenotype p before the derivation of the highlighted non-terminal

introduced CFG does not pose any constraint neither on the size of the ruleset and of the multisets, nor on the types of rules to be used. This way the user is not required to have any specific knowledge of the observed P system, and the discovery is completely left to the evolutionary search. Moreover, the proposed CFG can be easily altered to encompass different scenarios, e.g., by simply adding more rule types.

We provide an example of genotype–phenotype mapping in Fig. 4 for a genotype with 10 codons. For this example, we set both Λ and Γ to 10.

4.2 Computing the fitness

To assess the performance of the inferred rules in approximating the observed P system, we introduce a fitness measure based on that proposed in [16]. Namely, the goal is to quantify how well the transitions in $\overrightarrow{\mathscr{C}}$ are captured by the considered ruleset. To this end, for each $\mathscr{C}_i \in \overrightarrow{\mathscr{C}}$, with $i = 0, \dots, m-1$, we compute $d\left(\mathscr{C}_{i+1}, \tilde{\mathscr{C}}_{i+1}\right)$, where $\mathscr{C}_{i+1}$ is the next observed configuration and $\tilde{\mathscr{C}}_{i+1}$ is the configuration obtained by applying the candidate ruleset for one computation step to $\mathscr{C}_i$. Clearly, a lower distance (averaged across all m transitions) means a better P system approximation and a better quality of the ruleset. Hence, we aim at finding a ruleset R that minimizes the average distance across transitions.

We define d as an *edit distance* between membranes based on the following operations:

(1) *Addition* of a membrane and its content (including other membranes), whose cost is equal to the number of membranes added;
(2) *Removal* of a membrane and its content, whose cost is equal to the number of membranes removed;
(3) *Change* of the objects contained in a membrane, whose cost is computed as the *Jaccard distance* between the two multisets contained in the membranes. We recall that the Jaccard distance measures the dissimilarity between A and B, as

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

which ranges from $d_J(A, B) = 0$, if the two multisets are exactly the same, to $d_J(A, B) = 1$ if the intersection is empty.

The edit distance between two membrane structures is measured by counting the minimum number of edit operations required to transform one into the other. We define $(e_1 \dots, e_l)$ as the minimum sequence of operations to transform a configuration $\mathscr{C}_1 = [w_1[w_2]_{h_2} \dots [w_m]_{h_m}]_{h_1}$ into $\mathscr{C}_2 = [v_1[v_2]_{k_2} \dots [v_\ell]_{k_\ell}]_{k_1}$. Thus, if the cost of a single operation e_i is $\gamma(e_i)$, the total cost of the sequence, i.e., the distance between the two configurations, is given by

$$d\left(\mathscr{C}_1, \mathscr{C}_2\right) = \sum_{i=1}^{l} \gamma(e_i).$$

To overcome the limitation given by the computational burden implied by finding the minimum cost, some practical simplifications can be introduced—without impacting the quality of the final results. First, when considering two root membranes with different labels, the distance is given by the cost of completely removing the first membrane plus the cost of adding the whole second membrane. Conversely, if the two membranes share the same root, the total distance $d(w_1, v_1)$ is defined by the cost of changing w_1 into v_1 plus the cost of changing the membrane structure recursively. Specifically, all the membranes contained in the root membrane are partitioned according to their label, and, subsequently, for each one of these partitions, the elements in the outermost membrane are sorted (in lexicographic order) and the distance between each pair of membranes is computed.

This approximation returns a distance that preserves two fundamental properties: membranes with equal labels can change their order and membranes with different labels need to be removed and replaced. This is consistent with the fact that in P systems when two membranes have the same content but different label, there is no easy way to transform one into another, as the rules are naturally bounded with membranes by their definition.

5 Experimental setting

We experiment with six different benchmark problems in order to assess the capability of the proposed approach to correctly infer the ruleset of a P system. Each of the considered benchmark problems deals with a different class of P systems, and is, hence, targeted at testing the inference ability with respect to a specific task. Namely, the send-in and send-out problems test the ability to infer communication rules, while the variable assignment problem tests the ability to learn weak membrane division rules. Finally, we also test and discuss the ability to infer three basic arithmetic operations: unary addition, unary multiplication, and unary division.

To achieve general results we experiment with different settings concerning the benchmark problems, i.e., with different problem sizes. Moreover, we evaluate the impact of including some a priori knowledge in the evolutionary search, by reducing the rule types in the CFG to only those actually employed by the observed P system.

Regarding the code, we employ PonyGE2 [27] for the GE part and Psystem-GA [16] for the P systems simulation. For the complete reproducibility of the following results, we make our code available at https://github.com/giorgia-nadizar/psystems-ge.

5.1 Benchmark problems

We hereby introduce the benchmark problems used to assess the effectiveness of our approach. We summarize their features with respect to the problem size n in Table 1. Namely, we report the number of rules needed n_{rules}, the number of rule types used n_{rtypes}, the number of different objects involved n_{obj}, the number of computation steps needed to solve the problem n_{steps}, the cardinality of the LHS and of the RHS, and whether cooperative rules are required, i.e., whether they are actually triggered by the presence of multiple objects or if a single object suffices. We detail the benchmark problems in the following.

5.1.1 Send-in problem

The goal of the send-in problem is to send n objects from the outer membrane h to an inner membrane k. Hence, to move from an initial configuration

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

to a final configuration

$$[[x_0\ x_1\ \dots\ x_{n-1}]_k]_h$$

following a specified order. In more details, the dynamic of the P system is given by two types of rules:

$$\begin{aligned} &\big[\, x_{i,j} \to x_{i,j-1} \,\big]_h && \text{for } 0 \le i < n \text{ and } 0 < j \le i,\\ &x_{i,0}\ [\ ]_k \to [\, x_i\,]_k && \text{for } 0 \le i < n. \end{aligned}$$

The former act as a counter to determine when an object can be sent in the inner membrane, whereas the latter ones actually perform the send-in action.

5.1.2 Send-out problem

The send-out problem is symmetric with respect to the send-in problem, meaning that the goal is to send the n objects from the innermost membrane k to an outer membrane h, starting from

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

to achieve

$$\big[x_0\ x_1\ \dots\ x_{n-1}\ [\ ]_k\big]_h.$$

The rules governing the system are:

$$\begin{aligned} &\big[\, x_{i,j} \to x_{i,j-1} \,\big]_k && \text{for } 0 \le i < n \text{ and } 0 < j \le i,\\ &\big[\, x_{i,0} \,\big]_k \to x_i\ [\ ]_k && \text{for } 0 \le i < n, \end{aligned}$$

respectively, acting as counters and sending out the objects.

5.1.3 Variable assignment problem

In this problem, we consider n Boolean variables stored within a membrane. The aim is to generate 2^n membranes, each containing a possible assignment of the aforementioned variables. Thus, starting from

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

we want to reach

$$\Big[\underbrace{[t_0\ t_1\ \dots\ t_{n-1}]_k, [t_0\ t_1\ \dots\ f_{n-1}]_k, \dots, [f_0\ f_1\ \dots\ f_{n-1}]_k}_{2^n}\Big]_h,$$

where t_i and f_i indicate variables which are set to true and false, respectively. The needed rules are

$$\begin{aligned} &\big[\, x_{i,j} \to x_{i,j-1} \,\big]_k && \text{for } 0 \le i < n \text{ and } 0 < j \le i,\\ &\big[\, x_{i,0} \,\big]_k \to [\, t_i\,]_k\, [\, f_i\,]_k && \text{for } 0 \le i < n, \end{aligned}$$

where, as before, the first ones act as counters, while the latter ones actually perform the assignment task.

5.1.4 Elementary operations

Last, we use three elementary functions as benchmark problems. Namely, we consider unary addition, unary multiplication, and unary division, inspired by [21]. The first two problems require only non-cooperative rules, unlike the last one which needs a cooperative rule, as we can note from the definition of the rules needed for performing such operations, provided below. We decided to focus on the study of these three elementary functions—and not on all the five listed in [21]—since the considered functions cover a wide enough range of cases needed to test the performance of an automatic inference method, as also done in [16].

Unary addition Given a membrane k containing $n - p$ copies of an object a and p copies of an object b, the goal is to send out n copies of an object c, with p randomly selected in $\{0, \dots, n\}$. To model this operation we can combine two send-out rules

$$[\,a\,]_k \rightarrow c\ [\ ]_k$$
$$[\,b\,]_k \rightarrow c\ [\ ]_k\,,$$

which need to be applied for a total of n times.

Unary multiplication Given a membrane k containing p copies of an object a, the goal is to send out $p \cdot n$ copies of an object b, with p randomly selected in $\{1, \dots, n\}$. To model the multiplication, we require one single send-out rule

$$[\,a\,]_k \rightarrow b^n\ [\ ]_k\,.$$

A single application of the rule suffices as we do not enforce that only one object can cross a membrane at each step.

Unary division Given a membrane k containing p copies of an object a, the goal is to send out $\lfloor \frac{p}{n} \rfloor$ copies on an object b, with p randomly selected in $\{1, \dots, n^2\}$. Similarly to the unary multiplication, to achieve the desired behavior, we require a single application one send-out rule

$$\left[\,a^n\,\right]_k \rightarrow b\ [\ ]_k\,,$$

5.2 Parameters settings

For the benchmarks, we experiment with increasing problem sizes, $n \in \{2, 3, 4, 5\}$. Clearly, a larger problem size corresponds to a harder inference task, as we can understand by examining Table 1. First, we notice a quadratic dependence of n_{rules} and n_{obj} from n for send-in, send-out, and variable assignment problems. The growth of n_{rules} indicates that the GE algorithm needs to produce more rules, and all of them need to be correct, while the increase in n_{objects} implies that, when generating a rule, there is a greater chance of picking the wrong object as the pool grows. Second, for multiplication and division, there is a linear dependence on n for |RHS| and |LHS|, respectively, which increases the difficulty, since more correct objects need to be selected to form a rule.

Regarding the elementary operations, there is an additional parameter at play, that is p, as mentioned in Sect. 5.1. For addition and multiplication, we set $p = \lceil \frac{n}{2} \rceil$, whereas for division we set $p = n$. For addition, we choose p to have an approximately equal number of objects of each type, a and b, within the membrane. Conversely, for multiplication and division, we select p as a reasonable value within its range of variability, after a preliminary experimental phase in which we observed it played a minor role with respect to the identification difficulty.

Concerning the length $m + 1$ of the observed sequence $\overrightarrow{\mathscr{C}} = (\mathscr{C}_0, \dots, \mathscr{C}_m)$, we always consider the full computation for each problem. In other words, we observe all the configurations the P system goes through to pass from the initial configuration to the target one, i.e., to the solution of the problem. Hence, we always set $m = n_{\text{steps}}$, that is $m = n$ for send-in, send-out, assignment, and addition, and $m = 1$ for multiplication and division.

For the GE algorithm, we rely on Algorithm 1 with $n_{\text{pop}} = 1000$, $n_{\text{off}} = 990$, and $n_{\text{gen}} = 2000$, and the following operators [28].

- *Initialization* We use position-independent grow as initialization [29], that builds each individual as a tree, by randomly picking production rules in the CFG. This way it is possible to control the depth of the tree, which we

Table 1 Summary of the features of the considered benchmark problems, with respect to the problem size n

Problem	n_{rules}	n_{rtypes}	n_{obj}	n_{steps}	|LHS|	|RHS|	Cooperative
Send-in	$\frac{n(n+1)}{2}$	2	$\frac{n(n+3)}{2}$	n	1	1	×
Send-out	$\frac{n(n+1)}{2}$	2	$\frac{n(n+3)}{2}$	n	1	1	×
Assignment	$\frac{n(n+1)}{2}$	2	$\frac{n(n+5)}{2}$	n	1	(1, 1)	×
Addition	2	1	3	n	1	1	×
Multiplication	1	1	2	1	1	n	×
Division	1	1	2	1	n	1	✓

For the RHS of the variable assignment problem, we report the size of both the produced multisets

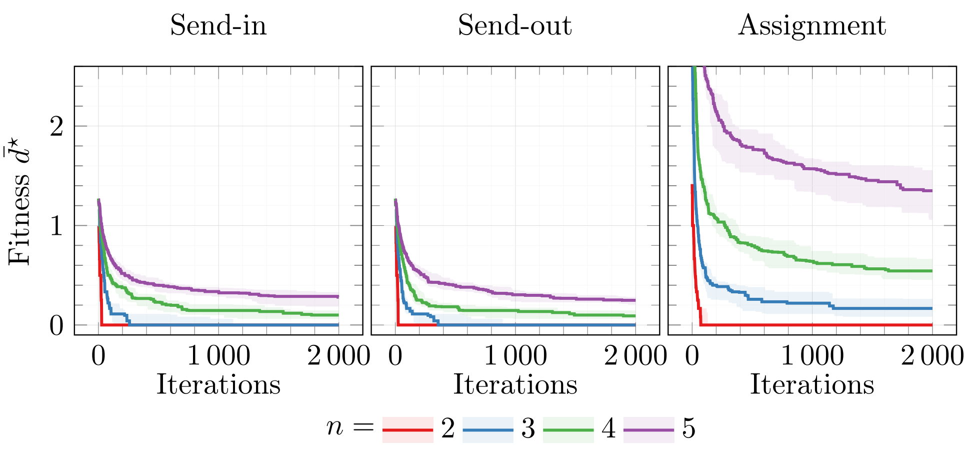

Fig. 5 Median and inter-quartile range (across 30 runs) of the fitness $\bar{d}^\star$ of the best individual along iterations for the elementary operations and problem size $n \in \{2, \ldots, 5\}$

limit to 10, to prevent excessively large individuals at initialization. To obtain a bitstring, when choosing production rules in the tree construction, we select a suitable codon and append it to the genome.

- *Selection*. For selection, we rely on tournament selection [30], with a tournament size of 2. Namely, to select an individual, we randomly sample 2 individuals from the entire population, and pick the fittest among them.
- *Crossover* After selecting the two parents, we recombine their genomes via variable one-point crossover. That is, we uniformly select a crossover point for each of the two genomes and we swap the bits to the right of each point between the two parents. This way genomes are also allowed to grow or shrink. We apply crossover with a probability $p_{\text{xo}} = 0.75$, meaning that in 25% of cases, the parents are simply copied to form the two children.
- *Mutation* To further modify the genome, we resort to a int-flip per codon mutation, with probability $p_{\text{mut}} = \frac{1}{|g|}$ for each codon. This means that we randomly change each codon to another integer value with a probability $p_{\text{mut}} = \frac{1}{|g|}$.

We choose these settings for GE as they are commonly used in the literature. Moreover, we performed several preliminary experiments with different values for n_{pop} and n_{gen} and converged to the aforementioned ones as they constitute a reasonable trade-off between computational power and time consumption, as we shall see in Sect. 6.1.

6 Results and discussion

In the following, we report the results obtained on the aforementioned benchmark problems.

First, we display the progression of the fitness $\bar{d}^\star$ of the best individual in the population along the iterations of the GE algorithm for the first three benchmark problems—send-in, send-out, and variable assignment—in Fig. 5. Namely, each plot refers to a problem, and each line refers to a problem size $n \in \{2, \ldots, 5\}$. Given the stochastic nature of the GE algorithm execution, we run it 30 independent times for each configuration, to ensure the consistency of results, as suggested in the GE literature. Hence, in the plots, we report the median and the inter-quartile range (shaded) across those independent runs.

Observing the figure, we notice that in all cases, the fitness decreases at the progression of iterations, meaning that the algorithm is indeed getting closer to the correct solution, i.e., to the true ruleset. However, it is clear that the efficacy of GE is strongly dependent on the problem size n. Namely, we can notice two aspects of such dependency. First, the smaller the n the faster the fitness decrease, and second, only for smaller values of n the fitness reaches 0 (meaning that the evolutionary approach infers correctly all the P system rules), whereas in all other cases, it decreases but stalls at higher values.

To further analyze the second aspect, i.e., the performance at end of the GE algorithm execution, we introduce another indicator that we call *correct inference ratio* (CIR), defined as the number of runs in which GE correctly infers the P system at the end of evolution divided by the total number of runs performed. We report the CIR for each problem and problem size n in Table 2. As hinted by Fig. 5, for $n = 2$ GE can always infer the P system correctly (CIR $= 1$), whereas for $n = 5$ the correct inference never occurs (CIR $= 0$). For the intermediate values of n, instead, the values of the CIR are greater than 0 (besides for the assignment problem with $n = 4$). This gives us more insight into our results: even though the median of the fitness does not reach 0, there are still some executions in which the ruleset is correctly identified. Moreover, this draws our attention to the stochasticity of the GE execution, which, even in the very same conditions, can achieve different outcomes.

Anyway, the results in Table 2 confirm the behavior observed in Fig. 5 concerning the relation between n and the GE execution outcome. This is a direct consequence of the

Table 2 Correct inference ratio, i.e., number of runs in which GE correctly infers the P system divided by the total number of runs performed, for the first three benchmark problems for different problem sizes $n \in \{2, \dots, 5\}$

Problem	Correct inference ratio (CIR)			
	$n = 2$	$n = 3$	$n = 4$	$n = 5$
Send-in	1.00	0.80	0.10	0.00
Send-out	1.00	0.90	0.17	0.00
Assignment	1.00	0.17	0.00	0.00

neat increase in the number of objects employed for larger n (see Table 1), as this enlarges the *search space*, i.e., the number of existing rulesets complying with the target CFG, clearly making it more difficult for GE to find an optimal solution. These results suggest that the proposed approach is successful for small problem sizes, but suffers when scaling.

Concerning the other family of benchmark problems considered, the elementary operations, we report the results achieved in Fig. 6, where we use the same visual syntax employed for Fig. 5. To ease the visual examination of the results, we clip the x-axis to 25 iterations, as for all the elementary operations the median of the fitness reaches the perfect value of 0 before said limit.

The paramount trait of the plots in Fig. 6 is that the inference of all elementary operations is perfectly achieved for all the considered problem sizes. Moreover, it takes significantly less than 2000 iterations for the correct inference, implying an overall computation that is way less costly if compared to the other benchmark problems considered. To further reason on the convergence speed, we report in Fig. 7 the distribution of the total number of iterations necessary to correctly infer the ruleset of the P system, $i_{\bar{d}^\star=0}$, for each setting.

From both Figs. 6 and 7, we notice a clear dependence of the convergence speed from n for both the multiplication and division operations. This is an expected result given Table 1, where we note that, for these operations, the size of either the LHS or the RHS grows with n. However, GE can always effectively infer the correct ruleset because, differently from before, the search space does not grow with n. For the addition operation, instead, n seems to play a minor role, in compliance with Table 1, where we do not highlight any relationship between n and the inference difficulty.

Summarizing, from the experiments conducted, we clearly noticed that the main limitation of our approach lies in the size of the search space of GE: when this grows too large it is difficult to converge to the correct solution with the chosen computation budget. To try to overcome this issue, we attempt at restricting the search space by leveraging some

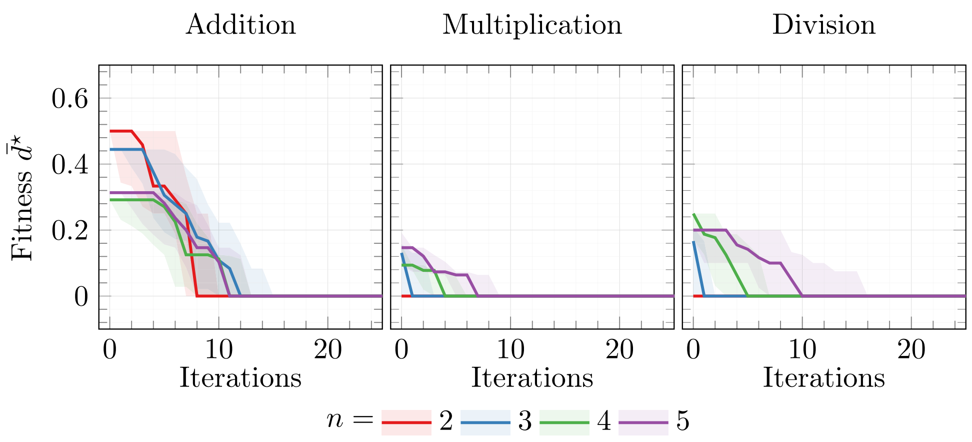

Fig. 6 Median and inter-quartile range (across 30 runs) of the fitness $\bar{d}^\star$ of the best individual along iterations for the elementary operations and problem size $n \in \{2, \dots, 5\}$

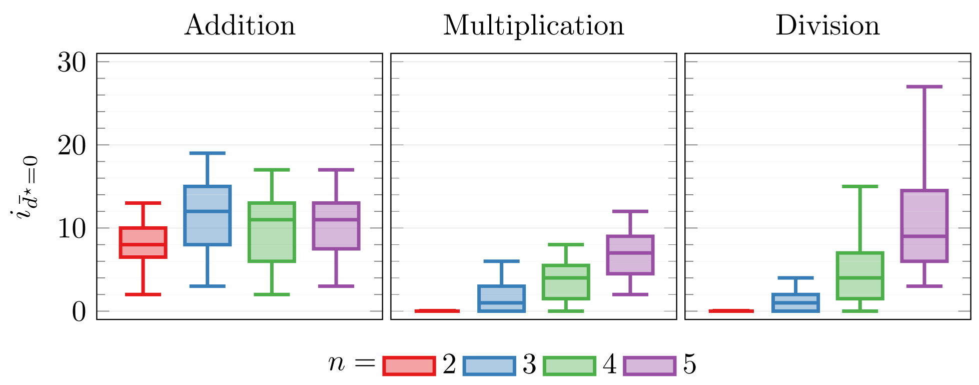

Fig. 7 Distribution of $i_{\bar{d}^\star=0}$, i.e., the amount of iterations needed to reach best fitness $\bar{d}^\star = 0$, for the elementary operations for different problem sizes

⟨ruleset⟩ ::= ⟨ruleset⟩∪ {⟨rule⟩} | {⟨rule⟩}
⟨rule⟩ ::= ⟨membrane⟩{⟨object⟩} rewrite{⟨object⟩} |
⟨membrane⟩{⟨object⟩} sendin{⟨object⟩} |
⟨membrane⟩ ::= m1 | ... | mΛ
⟨object⟩ ::= o1 | ... | oΓ

(a) Send-in

⟨ruleset⟩ ::= ⟨ruleset⟩∪ {⟨rule⟩} | {⟨rule⟩}
⟨rule⟩ ::= ⟨membrane⟩{⟨object⟩} rewrite{⟨object⟩} |
⟨membrane⟩{⟨object⟩} sendout{⟨object⟩} |
⟨membrane⟩ ::= m1 | ... | mΛ
⟨object⟩ ::= o1 | ... | oΓ

(b) Send-out

⟨ruleset⟩ ::= ⟨ruleset⟩∪ {⟨rule⟩} | {⟨rule⟩}
⟨rule⟩ ::= ⟨membrane⟩{⟨object⟩} rewrite{⟨object⟩} |
⟨membrane⟩{⟨object⟩} division{⟨object⟩}{⟨object⟩}
⟨membrane⟩ ::= m1 | ... | mΛ
⟨object⟩ ::= o1 | ... | oΓ

(c) Variable Assignment

Fig. 8 The CFGs in BNF to describe the rulesets of the P systems used to solve the considered benchmark problems in the easier setting. Starting from Fig. 3, we enhance the CFG with additional knowledge about each problem, by replacing ⟨multiset⟩ with ⟨{object}⟩ and reducing the replacement options for the ⟨rule⟩

knowledge of the observed systems. To this end, we repeat the previous experimental evaluation for problems where we could not find the optimal solution (send-in, send-out, and variable assignment) employing a CFG that leverages some a priori knowledge of the considered P systems. In practical terms, we examine two aspects: (1) both the LHS and the RHS of the rules are always composed of one single object, and (2) each ruleset only contains a subset of the four considered rule types.

Given these two observations, we introduce an enhanced CFG for each problem, to include the aforementioned additional knowledge; we report said CFGs in Fig. 8. Namely, for all problems, we substitute ⟨multiset⟩ with {⟨object⟩} in the structure of each rule (as prescribed by (1)). Moreover, we define only rewrite and send-in rules for the send-in problem, rewrite and send-out rules for the send-out problem, and rewrite and division rules for the variable assignment problem.

We remark that this setting is not unrealistic, as this knowledge about a P system can be derived by a human observer. Clearly, this requires additional effort with respect to the standard setting previously considered, yet it is still limited compared to that needed to infer the P system completely by hand.

We display the results of the supplementary evaluation in Fig. 9. From these plots, we note the same trend observed in the standard setting, regarding both the decrease of the fitness and the dependency with respect to n. To ease the comparison with the previous setting, we also provide comparative plots in Fig. 10. A twofold improvement emerges from these plots: (1) a speed up in the fitness decrease, and (2) convergence at 0 at the end of the GE algorithm execution even for larger values of n.

As before, we delve into a deeper analysis considering the CIR, reported in Table 3. Again, we notice an improvement with respect to the standard setting: not only there are more problems where CIR $= 1$, but also in all but one case we get CIR > 0. However, similarly to Table 2, we observe decreasing values of CIR at the increase of n, yet in this case the decrement is less dramatic. In fact, in both cases the search space grows with n, although here it is clearly smaller (because of the restricted CFGs employed).

Hence, we can conclude that with some preliminary observation of the system and, possibly, with multiple GE runs, it is feasible to correctly infer the P system in all the considered scenarios.

6.1 Computational effort

To complete our experimental evaluation, we provide an estimate of the computational effort required for the P system inference with GE. To evaluate the feasibility of the approach, we aim at quantifying the time required for executing the GE algorithm until the correct inference is achieved.

To this end, we consider only configurations where the observed CIR is equal to 1, that is: all the elementary operations with the standard setting; the send-in, send-out, and assignment problems in both settings with $n = 2$; and the send-in and send-out problems in the easier setting with $n = 3$. For each of these cases, we run the GE algorithm 10 times in a "controlled environment", i.e., without executing any other program on the same machine, and measure $t_{\bar{d}^\star=0}$, that is the amount of time taken to achieve $\bar{d}^\star = 0$. As machinery, we employ a 64 core workstation (AMD EPYC 7542 W-2295 with 64GB RAM), for which we enable only the usage of 8 cores.

We report the distribution of $t_{\bar{d}^\star=0}$ in seconds for all the considered cases in Fig. 11. These plots confirm the practicability of the proposed method, as the inference requires less than one minute for all the reported cases. As observed in Sect. 6, GE takes more time to converge to the correct solution when the search space is larger (i.e., for send-in, send-out, and variable assignment problems), although the inference times are still always reasonable.

We believe the results obtained in terms of time consumption constitute an added value of the proposed method and make it a promising research direction for future devel-

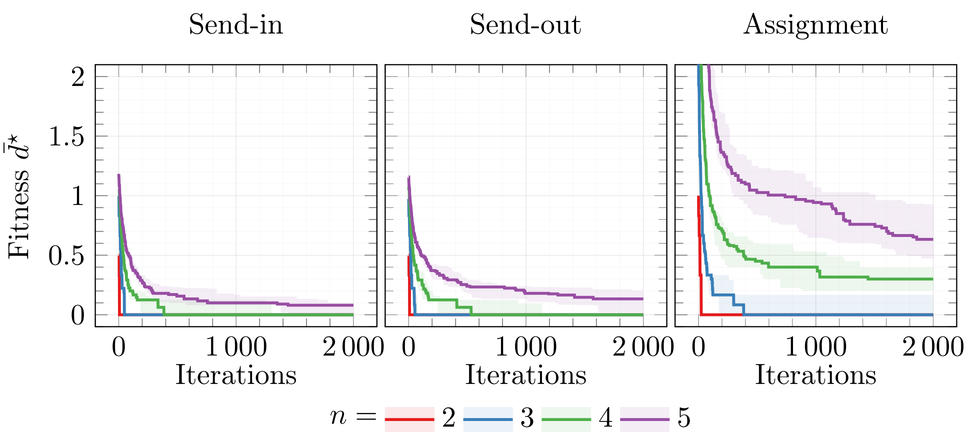

Fig. 9 Median and inter-quartile range (across 30 runs) of the fitness $\bar{d}^\star$ of the best individual along iterations for the first three benchmark problems and problem size $n \in \{2, \ldots, 5\}$ in the easier setting

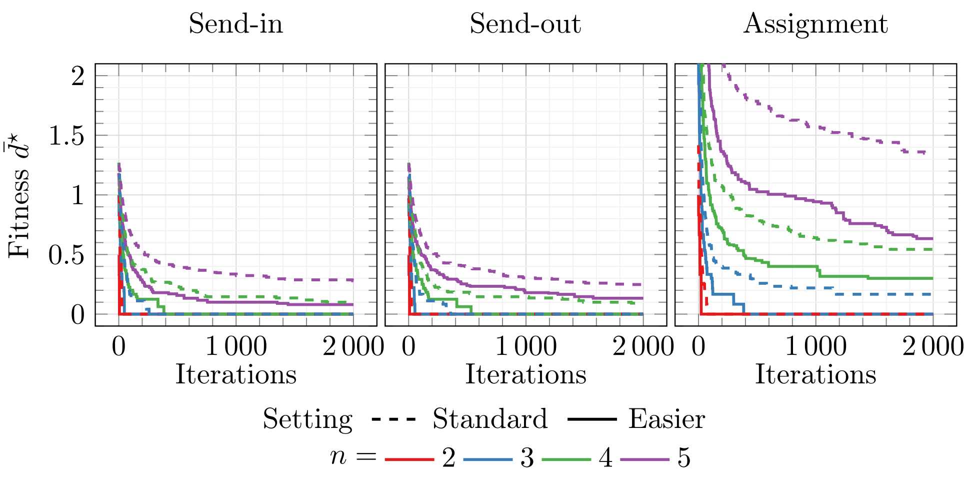

Fig. 10 Median (across 30 runs) of the fitness $\bar{d}^\star$ of the best individual along iterations for the first three benchmark problems and problem size $n \in \{2, \ldots, 5\}$ in the easier setting. We use a dashed line for the standard setting and a solid line for the easier setting

Table 3 Correct inference ratio, i.e., number of runs in which GE correctly infers the P system divided by the total number of runs performed, for the first three benchmark problems for different problem sizes $n \in \{2, \ldots, 5\}$ in the easier setting

Problem	Correct inference ratio (CIR)			
	$n = 2$	$n = 3$	$n = 4$	$n = 5$
Send-in	1.00	1.00	0.87	0.43
Send-out	1.00	1.00	0.87	0.27
Assignment	1.00	0.67	0.07	0.00

opments for the automated synthesis of P systems with a GE-based approach.

7 Conclusion and future directions

In this work, we leveraged grammatical evolution (GE) to automatically infer the ruleset of a P system given a sequence of its configurations. We tested our approach on six benchmark problems, involving different classes of P systems. During our analysis, we considered both a general framework, but also a more specific scenario where we included some additional knowledge, derived by human observation of the P system, in the GE search. Experimental results show that our method skilfully converges to the correct set of rules for basic arithmetic operations, with slight computational effort. In addition, we achieved remarkable results also for the three other benchmark problems considered—the send-in, send-out and variable assignment problems—for small problem sizes, especially when some additional information was provided.

The proposed approach could be relevant for P system design tasks, but also for structure and operation identification. This work is, to the best of our knowledge, the first attempt to automatically infer P system rules by means of GE, hence it paves the way for multiple future developments. In particular, it would be meaningful to test more advanced versions of GE, to assess whether they could enhance our method. For instance, structured grammatical evolution [31], where the genotypic representation explicitly links each gene to a non-terminal of the grammar, or probabilistic grammatical evolution [32–34], where production rules in the CFG are associated with different probabilities, or weighted hierarchical grammatical evolution [35], where a form of hierarchy

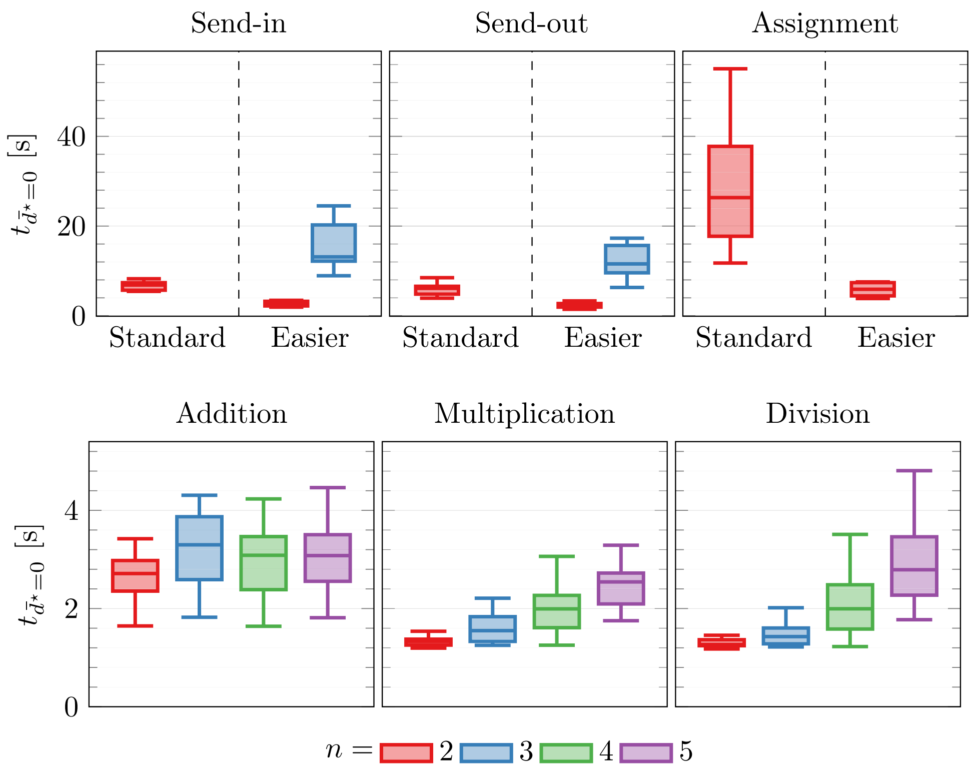

Fig. 11 Distribution of $t_{\bar{d}^\star=0}$ (in seconds), i.e., the amount of time needed to reach best fitness $\bar{d}^\star = 0$, for the first three benchmark problems (only the settings with CIR $= 1$) and the elementary operations (only with the standard setting)

on the genotype is imposed. In fact, all these GE variants have yielded improvements with respect to standard GE in various domains thanks to their increased locality [36]—a desirable property in EAs, which links the size of the steps in the search space to the size of the steps in the solution space—which could be of paramount importance for addressing our inference cases with a larger search space. Last, it would be noteworthy to experiment with different classes of P systems, also in non-deterministic settings.

Acknowledgements We wish to thank Luca Manzoni and Eric Medvet for fruitful discussions and feedback.

Author Contributions All authors contributed to the study conception and design. Material preparation, data collection and analysis were performed by all authors. The first draft of the manuscript was written by all authors, and all authors commented on previous versions of the manuscript. All authors read and approved the final manuscript.

Funding Open access funding provided by Università degli Studi di Trieste within the CRUI-CARE Agreement. The authors did not receive support from any organization for the submitted work.

Data availability The code to generate the datasets presented in this study can be found in online repositories. The names of the repository/repositories and accession number(s) can be found in the article.

Declarations

Conflict of interest The authors have no relevant financial or non-financial interests to disclose.

References

1. Păun, G., & Rozenberg, G. (2002). A guide to membrane computing. *Theoretical Computer Science, 287*(1), 73–100.
2. Alsalibi, B., Mirjalili, S., Abualigah, L., Yahya, R. I., & Gandomi, A. H. (2022). A comprehensive survey on the recent variants and applications of membrane-inspired evolutionary algorithms. *Archives of Computational Methods in Engineering, 29*(5), 3041–3057.
3. Ciobanu, G., Păun, G., & Pérez-Jiménez, M. J. (2006). *Applications of membrane computing* (Vol. 17). Springer.
4. Frisco, P., Gheorghe, M., & Pérez-Jiménez, M. J. (2014). *Applications of membrane computing in systems and synthetic biology*. Springer.
5. Păun, G. (2006)Introduction to membrane computing. In Applications of Membrane Computing (pp. 1–42). Berlin, Springer.
6. Escuela, G., & Gutiérrez Naranjo, M. Á. (2010). 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.
7. Zhang, G., Gheorghe, M., Pan, L., & Pérez-Jiménez, M. J. (2014). Evolutionary membrane computing: A comprehensive survey and new results. *Information Sciences, 279*, 528–551.
8. Kumar, M., Husain, D., Upreti, N., & Gupta, D., et al. (2010). Genetic algorithm: Review and application. Available at SSRN 3529843.
9. Koza, J. R. (1994). Genetic programming as a means for programming computers by natural selection. *Statistics and Computing, 4*(2), 87–112.
10. Beyer, H.-G., & Schwefel, H.-P. (2002). Evolution strategies—A comprehensive introduction. *Natural Computing, 1*, 3–52.
11. O'Neill, M., & Ryan, C. (2001). Grammatical evolution. *IEEE Transactions on Evolutionary Computation, 5*(4), 349–358.
12. Kennedy, J., & Eberhart, R. (1995). Particle swarm optimization. In Proceedings of ICNN'95-international Conference on Neural Networks, vol. 4, pp. 1942–1948. IEEE.
13. Narayanan, A., & Moore, M. (1996). Quantum-inspired genetic algorithms. In Proceedings of IEEE International Conference on Evolutionary Computation, pp. 61–66. IEEE.
14. Ryan, C., Collins, J. J., & Neill, M. O. (1998). Grammatical evolution: Evolving programs for an arbitrary language. In European Conference on Genetic Programming, pp. 83–96. Springer.
15. Nishida, T. Y. (2006). Membrane algorithms. In Membrane Computing: 6th International Workshop, WMC 2005, Vienna, Austria, July 18–21, 2005, Revised Selected and Invited Papers 6, pp. 55–66. Springer.
16. Leporati, A., Manzoni, L., Mauri, G., Pietropolli, G., & Zandron, C. (2023). Inferring P systems from their computing steps: An evolutionary approach. *Swarm and Evolutionary Computation, 76*, 101223.
17. Nishida, T. Y. (2020). Evolutionary P systems: The notion and an example. In International Conference on Membrane Computing, pp. 126–134. Springer.
18. Huang, X., Zhang, G., Rong, H., & Ipate, F. (2012). Evolutionary design of a simple membrane system. In Membrane Computing: 12th International Conference, CMC 2011, Fontainebleau, France, August 23–26, 2011, Revised Selected Papers 12, pp. 203–214. Springer.
19. Tudose, C., Lefticaru, R., & Ipate, F. (2011). Using genetic algorithms and model checking for P systems automatic design. *NICSO, 387*, 285–302.
20. Ou, Z., Zhang, G., Wang, T., & Huang, X. (2013). Automatic design of cell-like P systems through tuning membrane structures, initial objects and evolution rules. *International Journal of Unconventional Computing, 9*(5–6), 425–443.
21. Chen, Y., Zhang, G., Wang, T., & Huang, X. (2014). Automatic design of P systems for five basic arithmetic operations within one framework. *Chinese Journal of Electronics, 23*(2), 302–304.
22. López, D., & Sempere, J. M. (2006). Editing distances between membrane structures. In F. Rudolf, P. Gheorghe, R. Grzegorz, & S. Arto (Eds.), Membrane Computing: 6th International Workshop, WMC 2005 (Vienna, Austria). LNCS, vol. 3850, pp. 326–341. Springer.
23. Sempere, J. M., & López, D. (2006). Identifying P rules from membrane structures with an error-correcting approach. In H. J. Hoogeboom, G. Paun, G. Rozenberg, & A. Salomaa (Eds.), 7th International Workshop on Membrane Computing (Leiden, The Netherlands). LNCS, vol. 4361, pp. 507–520. Springer.
24. Eiben, A. E., Smith, J. E., et al. (2003). *Introduction to evolutionary computing* (Vol. 53). Springer.
25. Darwin, C., & Bynum, W. F. (2009). *The origin of species by means of natural selection: Or, the preservation of favored races in the struggle for life*. AL Burt.
26. Backus, J. W., Bauer, F. L., Green, J., Katz, C., McCarthy, J., Naur, P., Perlis, A. J., Rutishauser, H., Samelson, K., Vauquois, B., et al. (1963). Revised report on the algorithmic language Algol 60. *The Computer Journal, 5*(4), 349–367.
27. Fenton, M., McDermott, J., Fagan, D., Forstenlechner, S., Hemberg, E., & O'Neill, M. (2017). Ponyge2: Grammatical evolution in python. In Proceedings of the Genetic and Evolutionary Computation Conference Companion, pp. 1194–1201.
28. O'Neill, M., & Ryan, C. (2003). *Grammatical evolution*. Springer.
29. Fagan, D., Fenton, M., & O'Neill, M. (2016). Exploring position independent initialisation in grammatical evolution. In 2016 IEEE Congress on Evolutionary Computation (CEC), pp. 5060–5067. IEEE
30. Blickle, T. (2000). Tournament selection. *Evolutionary Computation, 1*, 181–186.
31. Lourenço, N., Pereira, F. B., & Costa, E. (2015). SGE: A structured representation for grammatical evolution. In International Conference on Artificial Evolution (Evolution Artificielle), pp. 136–148. Springer.
32. Mégane, J., Lourenço, N., & Machado, P. (2021). Probabilistic grammatical evolution. In European Conference on Genetic Programming (Part of EvoStar), pp. 198–213. Springer.
33. Mégane, J., Lourenço, N., & Machado, P. (2022). Probabilistic structured grammatical evolution. In 2022 IEEE Congress on Evolutionary Computation (CEC), pp. 1–9.
34. Mégane, J., Lourenço, N., & Machado, P. (2022). Co-evolutionary probabilistic structured grammatical evolution. In Proceedings of the Genetic and Evolutionary Computation Conference, pp. 991–999.
35. Bartoli, A., Castelli, M., & Medvet, E. (2018). Weighted hierarchical grammatical evolution. *IEEE Transactions on Cybernetics, 50*(2), 476–488.
36. Rothlauf, F., & Oetzel, M. (2006). On the locality of grammatical evolution. In Genetic Programming: 9th European Conference, EuroGP 2006, Budapest, Hungary, April 10–12, 2006. Proceedings 9, pp. 320–330. Springer.

Giorgia Nadizar earned her Bachelor and Master in Electronics and Computer Engineering at the University of Trieste, Italy, in 2019 and 2021 respectively. She is currently pursuing her Ph.D. degree at the University of Trieste. Her research interests include Evolutionary Computation and its application for the synthesis of abst act and concrete artifacts, such as P Systems and robot controllers.

Gloria Pietropolli holds a B.Sc. in Mathematics (2018) and an M.Sc. in Mathematics (2020) from Padua University, Italy. Currently, she is in the final year of her Ph.D. program at the University of Trieste and OGS (Italian National Institute of Oceanography and Applied Geophysics). Her research interests lie in the realm of artificial intelligence, with a particular emphasis on evolutionary computation techniques. She is actively involved in developing novel algorithms and applying these methods to various scientific domains, including membrane computing, optimal control and ocean biogeochemical modeling.