Evolving Spiking Neural P Systems by Fixing Neurons, and Varying Rules and Synapses

Carlo Cezar R. Zarate, Francis George C. Cabarle, Ivan Cedric Macababayao, Ren Tristan De la Cruz
Dept. of Computer Science, University of the Philippines Diliman, Quezon city, Philippines

ABSTRACT

In this study, we explore evolving Spiking Neural P Systems, where a genetic algorithm framework is applied on Spiking Neural P Systems. The framework aims to (1) reduce the number of rules of an SN P system with 100% fitness and to (2) find a 100% fit SN P system using an initial SN P system with less than 100% fitness. Fitness, in this paper, is a measurement from 0% to 100% of an SN P system's accuracy in performing its intended function by getting the longest common substring of its output and its ideal output. The framework is limited to modifying the synapses and the rules of the given SN P systems. The framework is tested on bitwise addition and subtraction SN P systems, under three categories: Type A, with an initial fitness of 100%, Type B, with an initial fitness less than 100%, and Type C, with an initial fitness less than 100% fitness and with extra neurons. Such categories refer tothe different characteristics of the SN P systems which will be the basis for initial population of our experiments.

The results of the experiments show that the framework is successful on both objectives, however finding SN P systems with 100% fitness deemed more challenging as the size of the system grows, especially in subtraction of Type B and C, and addition of Type C.

1 INTRODUCTION

When it comes to solving problems, taking inspiration from nature to form a solution is nothing new. Back in the early 1990s, Eiji Nakatsu, was faced with the task of designing a bullet train that would go as fast as 350 kilometers per hour. The speed, however, was not the only challenge in designing the train - noise and sudden pressure increase need to be considered, too. He turned to problem-solving with solutions inspired from nature for inspiration. He designed the train borrowing characteristics from animals to address the problems. The pantograph of the train was shaped like an owl's wing, to minimize vibrations and noise, the shaft took the shape of a penguin's body to lower the wind resistance, and the nose of the train took inspiration from the kingfishers beak to address the sudden increase in pressure when passing through a tunnel [13]. There are two fields in computer science that are relevant in this paper that both focus on problem-solving with nature-inspired solutions: Membrane Computing, and Evolutionary Computing.

Membrane computing is a field in computer science where models are designed after living cells. In the field emerged P systems - models inspired by cell biology. P systems can be cell-like, inspired from the membrane structure of a living cell, or tissue-like, from many cells contained in an environment, or neural-like, where cells communicate, arranged in a graph-like manner [6]. Under neural-like P systems is the interest of this work - Spiking Neural P Systems. This P system will be further explained in the next section.

Evolutionary computing, on the other hand, takes the concept of natural selection and applies it to computer science. An Evolutionary Algorithm places a population of candidate solutions to a given problem in an environment that favors better performing ones, forcing the fitness of the population to increase [4]. Genetic Algorithms, a type of Evolutionary Algorithm, is a point of interest in this study.

Problem-solving can also be approached using a combination of membrane computing and evolutionary computing. Evolutionary Membrane Computing combines the potential of evolutionary computing in real-world applications and the membrane systems of membrane computing. Two topics emerged from EMCs: Membrane-Inspired Evolutionary Algorithms (MIEA) and Automated Design of Membrane Computing Models (ADMCM). MIEAs combine the search principles of evolutionary computing and the computation mechanisms of P systems and puts an emphasis on the capability of membrane computing for real-world applications. MIEAs take an optional parameter set as an input and outputs an optimal parameter set. ADMCMs deal with the programmability of P systems, and studies in this topic take a computational task as an input and outputs a successful P system. More information on EMCs can be found on [15]. In 2015, [14] approaches traveling salesman problems by proposing an

Appendices that accompany this work include supplementary information such as figures, tables. The appendices may be available in a separate manner from the main paper. Correspondence: `fccabarle@up.edu.ph` .

approximate algorithm combining P systems and active evolutionary algorithms (AEAPS).

In this paper, we explore the idea of a framework using Genetic Algorithms to evolve Spiking Neural P Systems. This framework, which may be considered under ADMCM, is a tool to produce an SN P system that successfully performs a certain computational task with less or equal resources than the given SN P system that may or may not accurately perform the said computational task. In Section 2 we provide the preliminaries for the study. In Section 4, we introduce the general and specific goals of this work. In Section 3, we discuss related literature. In Section 4, we present the Theoretical Framework; In Section 5, we go in depth with the genetic algorithm, and its implementation in Section 6, which also includes the chromosome representation of the SN P systems. In Section 7, we discuss the results of the experiments. In Section 8, we provide conclusions and recommendations for future studies. Ideas for this work were taken from the BWMC2018 presentation in [1].

2 PRELIMINARIES

Spiking Neural P Systems

The way Spiking Neural P Systems compute is similar to how actual neurons compute. Neurons pulse spikes to neighboring neurons through synapses. These systems are represented as a directed graph. Nodes are called neurons, edges are called synapses, and inside neurons are rules, which dictate when the neuron will spike [12].

In [6], Spiking Neural P Systems were proposed to incorporate time as an information carrier of spiking neurons. These SN P systems only use one object for a spike because most neural signals are nearly identical, as a response to the models in [8]. A neuron spikes when a specific number of spikes has been accumulated, consuming spikes and sending one spike. When a neuron spikes, signals are sent to all connected neurons through synapses, creating as many copies as needed. The results of the computation are based on the spikes produced by a designated output neuron in the system, and the time interval in between the spikes. The system halts when exactly two spikes are produced by the output neuron. In [6] is also a proof of the computational completeness of SN P systems. A slightly extended variant of SN P systems shows that it is able two solve Subset Sum and 3-SAT problems, with nondeterminism. [8].

The version relevant for this work is an extended version of the SN P system in [12]. The extended version has an accepting mode, where there is a designated input neuron that can receive spikes from the environment, neurons are capable of producing more than one spike in a single time step, and computation results, called spike trains, can be interpreted as binary.

Definition 1 (SN P system). *A spiking neural P system with extended rules (rules that can produce more than one spike) of degree $m \geq 1$, as defined formally in [12] is of the form:*

$$\Pi = (O, \sigma_1, \ldots, \sigma_m, syn, in, out), \text{ where:}$$

(1) *$O = \{a\}$ is the singleton alphabet (a is called spike);*
(2) *$\sigma_1, \ldots, \sigma_m$ are neurons, of the form $\sigma_i = (n_i, R_i), 1 \leq i \leq m$, where:*
  (a) *$n_i \geq 0$ is the initial number of spikes contained in σ_i;*
  (b) *R_i is a finite set of rules of the following two forms:*
    (i) *$E/a^c \rightarrow a^p; d$, where E is a regular expression over $O = \{a\}$ and $c \geq p \geq 1$, $d \geq 0$;*
    (ii) *$a^s \rightarrow \lambda$, for $s \geq 1$, with the restriction that for each rule $E/a^c \rightarrow a^p; d$ of type (1) from R_i, we have $a^s \notin L(E)$*
(3) *$syn \subseteq \{1, 2, \ldots, m\} \times \{1, 2, \ldots, m\}$ with $i \neq j$ for all $(i, j) \in syn$, $1 \leq i, j \leq m$ (synapses between neurons).*
(4) *$in, out \in \{1, 2, \ldots, m\}$ indicate the input and the output neurons, respectively.*

The rules have a form of $E/a^c \rightarrow a^p; d$. The E is the regular expression of the rule, and if the number of spikes in the neuron falls in the language E, then the rule will activate and will consume exactly c spikes and produce p spikes to all connected neurons on a d delay. Some rules have the form $E/a^c \rightarrow \lambda; d$, which only means that $p = 0$. This form is called a forgetting rule, where the rule only consumes spikes. Another form is $a^c \rightarrow a^p; d$, whenever E is the same as a^c. An example of an SN P system that computes the bitwise AND of two bit strings is in Appendix A, Figure 5.

Genetic Algorithms

Genetic Algorithms is an evolution-inspired approach to solve computation problems. It is a method generally used for problems with a wide search space of candidate solutions. Genetic Algorithms can be used for problems with large spaces that are not smooth, and for problems where a good, not perfect, solution is sufficient. However, it can get stuck at a local optimum [10], i.e. the optimal solution is not always guaranteed to be found.

The algorithm starts from a population of candidate solutions to a problem. In this context, solutions are often referred to as chromosomes. Some chromosomes are fitter than others. Fitness may refer to a chromosomes chance to reproduce, and is often determined by how well a chromosome suits its environment. In the context of genetic algorithms, fitness is determined by how well the solution solves the problem using some abstract fitness function. Chromosomes with higher fitness are more likely to reproduce. Chromosomes that do reproduce, undergo recombination, where genes from each parent recombine to produce an offspring. In genetic algorithms, recombination is referred to as crossover. Offspring have a chance to mutate, where some of the genes may change. Crossover and mutation are repeated until enough offspring have been produced to populate the next generation. The new generation will again undergo fitness evaluation, parent selection, crossover, and mutation. The algorithm stops when either a chromosome with the maximum fitness has been found, or when a maximum number of generations has been reached [10].

3 RELATED WORK

[9] developed a genetic algorithm framework for Neural Networks. The neural networks were represented by constraint connectivity matrices of size N by N+1, where N corresponds to the number of neurons. The first N columns contain the constraints between the neurons, and the final column is reserved for the threshold biases. The algorithms crossover function simple swaps rows from two different matrices, and its mutation function changes a value in the matrix. The fitness of each network was determined by their total sum squared error, and the selection method used was fitness-proportionate. The algorithm was tested on three problems: XOR Problem, Four-Quadrant Problem, and Pattern Copying.

[11] explored for a framework to use over Spiking Neural P Systems with Rules on Synapses or RSSN P systems. RSSN P systems are variants of SN P systems where instead of having the rules in the neurons, the rules are in the synapses. The goal of the experiment is to use a genetic algorithm framework to evolve RSSN P systems to reduce resources such as the initial number of spikes, rules (including consumed and produced spikes), and synapses. The framework is tested on five different functions: NOT, AND, OR, ADD, and SUB.

[3] also worked on a framework for Spiking Neural P Systems. The framework evolves the SN P systems by modifying the topology of the system - its neurons and synapses. The algorithm was tested on binary addition and binary subtraction over three categories: baseline (topology and precision with the closest approximations), original (larger topology and smaller precision), and adversarial (additional m neurons connected pseudo-randomly with pseudo-randomly added rules).

[7] developed a representation, a simulator, and a genetic algorithm framework for Spiking Neural P Systems with Polarization, a variant of SN P systems that rules depend on the charge of the neuron instead of the number of spikes to activate. The framework modifies the rules and synapses to evolve the PSN P systems. The framework was tested on two functions: AND and OR, over three categories: Perfect, Extra, and Mutated.

[11] [3] and [7] all proposed a genetic algorithm framework on their respective SN P system variant. This study intends to do the same with SN P systems, like [3], but instead of modifying the neurons and synapses of the system, the framework in this study intends to modify the rules and synapses, like in [11] and [7]. The framework in this study will also be tested on bitwise addition and subtraction, like in [3] and [11]. Some parameters similar to [11], [3], and [7] is mentioned in Section 5. A brief comparison of the results with [11] is in Section 7.

4 GOALS AND FRAMEWORK OF THIS WORK

Creating SN P systems by hand can be difficult and time-consuming. A tool or framework that could create SN P systems that could perform certain functions would be beneficial to researchers. In producing such a framework, this work proposes using genetic algorithms to find SN P systems with 100% fitness. Fitness, in this paper, is a measurement between 0% and 100% of how accurately an SN P system performs its intended function. It is calculated using some fitness function, which compares the SN P system's output with its ideal output. In this work the number of neurons are fixed, while the number of rules and synapses are variables. Further, given an initial SN P system the framework aims to evolve the system to: reduce or maintain the number of its rules, and find a system with up to 100% fitness if the initial system has less than 100% fitness.

More specifically we construct a framework with a genetic algorithm that when given the following:

- Initial SN P system $\Pi_{init} = (O, \sigma_1, \ldots, \sigma_m, syn, in, out)$ that performs a function $f(a)$ with fitness p, where $0 \leq p \leq 100$,
- A set $S = \{(a_1, b_1), (a_2, b_2), \ldots, (a_n, b_n)\}$ where a_i is a set of input spike trains $\{a_{i1}, \ldots, a_{ij}\}$ where $j = |in|$, and b_i is its corresponding output spike train, and
- $\forall((a_i, b_i) \in S,\ f(a_i) = b_i$,

will construct an SN P system $\Pi_{final} = (O, \sigma_1',$ $\sigma_m', syn', in, out)$ such that $\forall((a_i, b_i) \in S, \Pi_{final}(a_i) = b_i$, $m' = m$, and $|R'| \leq |R|$.

The framework will be tested on the SN P systems for the functions addition and subtraction. The SN P systems that will be used are based on [5].

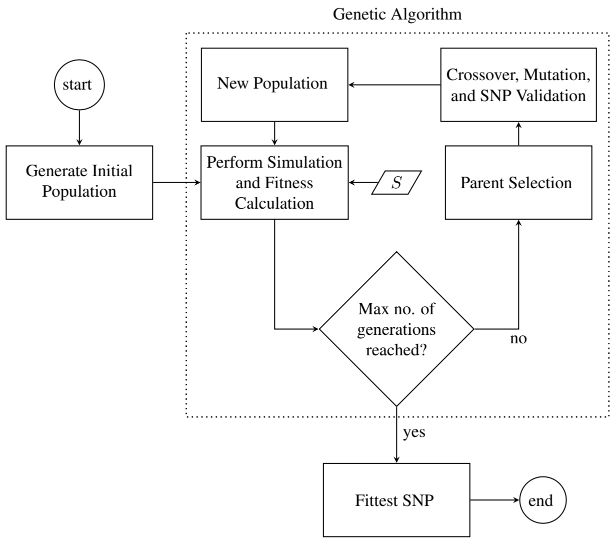
Genetic Algorithm
start
Generate Initial Population
New Population
Perform Simulation and Fitness Calculation
S
Crossover, Mutation, and SNP Validation
Parent Selection
Max no. of generations reached?
no
yes
Fittest SNP
end

Figure 1: The framework

The framework is shown in Figure 1. The framework starts with an Initial Spiking Neural P System Π_{init} that computes a function $f(a)$ but with a fitness less than or equal to 100 (refer to Section 4 for the definition of Fitness), and a set of input-output spike train pairs $S = \{(a_1, b_1), (a_2, b_2), \ldots, (a_n, b_n)\}$ where a_i is a set of input spike trains $\{a_{i1}, \ldots, a_{ij}\}$ where $j = |in|$, b_n is an output spike train, and $f(a_i) = b_i$. The input-output pairs will serve as test cases for evaluating the SN P systems. The goal of the framework is to improve the fitness and to reduce the number of rules in the system.

The flow starts in the leftmost rectangle with the creation of the first generation for the genetic algorithm. The initial SN P system will populate the first generation by mutating the initial SN P system multiple times using many different mutation functions.

In the next rectangle, all SN P systems in the population will be simulated with the given set of inputs from the set of input-output pairs S. The set of outputs generated from every SN P system in the population will be compared to the set of outputs from S using some fitness function. Details of the fitness function are in Section 5, in the Fitness Evaluation subsection. The score obtained from the evaluation function will serve as the fitness of the SN P system. The highest-scoring SN P system with the lowest number of rules are considered the fittest and saved for the next generation.

The next step is the diamond, where the framework checks for the halting condition. While the maximum number of generations is not yet reached, the program proceeds to the next rectangle, which is parent selection.

After all SN P systems from the population has been evaluated, the algorithm will select parents that will populate the next generation. The likelihood of an SN P system getting selected will depend on its fitness. Higher fitness will correspond to a higher probability of getting selected. After selection, the framework will proceed to Crossover, Mutation, and SN P Validation.

In crossover, the parents will be paired up with another until enough pairs have been established to populate the next generation. Every pair of parents will produce two children. A parent can be paired up more than once or not at all, but fitness does not play a role in pairing. After enough pairs have been established, all pairs will have a chance to undergo crossover. If a pair undergoes crossover, the children will be the resulting SN P systems. Otherwise, the children will be an exact copy of their parents. After crossover, all children will have a chance to undergo mutation, where synapses may be deleted or added, and rules may change. After mutation is Validation, which makes sure that the SN P systems after crossover and mutation satisfy Definition 1. Valid SN P systems will now populate the next generation, which again will undergo fitness evaluation, parent selection, crossover, mutation, and SN P validation. The cycle will repeat until the maximum number of generations has been reached. The fittest SN P system can be retrieved once the program halts.

Only SN P systems as defined in Definition 1 from [12] are considered in this work, e.g. a global clock is used, neurons operate in parallel. Spiking rules used have no delay and have regular expressions limited to the form of a^i only. This form is used because the initial SN P systems are based from [5], which only contains rules of the mentioned form. Input and output neurons only contain the rule $a \to a; 0$ and $a \to \lambda; 0$, respectively.

The Π_{final} outputs of the framework in Figure 1 will have the same number of neurons as Π_{init}, i.e. the framework can only mutate the rules and the synapses of the SN P systems. The mutation functions are discussed in Section 5. SN P systems considered in this work are for binary operations, e.g. bitwise addition, hence our systems are limited to two input neurons only.

5 THE GENETIC ALGORITHM

Set S. All bitwise addition (respectively, subtraction) SN P systems share the same input spike trains and output spike trains. There are 10 pairs of 7-bit input spike trains for the input - one for each input neuron, and 10 10-bit output spike trains.

Fitness Evaluation. The fitness evaluation method used in this framework is Longest Common Substring, or LCS in short. LCS returns the length of the longest substring present in two strings. In this paper, the length of the longest substring of the ideal output spike train b_i and the resulting output spike train b'_i will be computed. The sum of the LCS of all ideal output spike trains b_i and resulting output spike trains b'_i will determine the fitness of the SN P system. [3] and [7] use LCS as the fitness function. [11] uses both LCS and Longest Common Subsequence.

Selection Method. The framework has two selection methods: *Fitness-proportionate*, and *Fitness-proportionate with Elitism.*

Fitness-proportionate. The probability P of an SN P system Π being added to the pool of parents to populate the next generation is described by Equation 1.

$$P(\Pi_j) = \frac{g(\Pi_j)}{\sum_{i=0}^{p} g(\Pi_i)} \tag{1}$$

where $g(\Pi)$ is the fitness function and p is the number of SN P systems in the population. This method allows fitter chromosomes to be more likely selected than less fit ones. Less fit chromosomes can still be selected, although with a lower chance. This allows the pool to still have a diverse set of chromosomes.

Fitness-proportionate with Elitism. For every generation, the top 10 chromosomes are added directly to the next generation without any modifications. After adding the top 10 chromosomes, the rest of the population is filled up using the fitness-proportionate selection method. This ensures that the best chromosomes are preserved, making sure that the best chromosome in generation i is at least the same or better than the best chromosome in generation $i-1$. This technique, however, may result in a less diverse genetic pool.

In two selection methods from [11], "25% of the population based on fitness" and "Top 25% of the population + 25% of the population based on fitness", the "based on fitness" also uses Equation 1.

Crossover Functions. Crossover functions are *Swap Rules*, and *Swap Synapses*, with examples given in Figure 7 and Figure 8, respectively (Appendix B).

Swap Rules. Given two SN P systems Π_1 and Π_2 with m neurons $\sigma_1, \ldots, \sigma_m$ and $\sigma'_1, \ldots, \sigma'_m$ respectively, select neurons σ_n, and σ'_n. Swap all rules in σ_n and σ'_n.

Swap Synapses. Given two SN P systems Π_1 and Π_2 with m neurons $\sigma_1, \ldots, \sigma_m$ and $\sigma'_1, \ldots, \sigma'_m$ respectively, and a set of synapses syn and syn' respectively, select neurons σ_n and σ'_n. Swap all outgoing synapses of σ_n and σ'_n $(n, j) \in syn$ and $(n', j') \in syn'$.

Mutation Functions. Six mutation functions in total are used: four for the rules and two for the synapses. Examples of each of the six mutation functions are in Figures 9b, 10a, 10b, 11a, 11b, 12 (Appendix B).

Mutate Regular Expression. The function mutates the regular expression, a^i, of a chosen rule, $a^i/a^c \to a^p; d$, in the system. All the rules in the system are eligible for this function except for rules inside the input and output neurons. Once a rule has been chosen, the regular expression of the rule can be mutated from a^i to any regular expression in the interval $[a, a^{i+1}]$.

Mutate Number of Consumed Spikes. The function mutates the number of consumed spikes, c, of a chosen rule, $a^i/a^c \to a^p; d$, in the system. The rules that are eligible for the function are rules that are not inside the input and output neurons and rules that have a regular expression of at least a^2. Once a rule has been chosen, the number of consumed spikes of the rule can be mutated from c to any value in the interval $[1, i]$.

Mutate Number of Produced Spikes. The function mutates the number of produced spikes, p, of a chosen rule, $a^i/a^c \to a^p; d$, in the system. The rules that are eligible for the function are rules that are not inside the input and output neurons and rules that have a regular expression of at least a^2. Once a rule has been chosen, the number of produced spikes of the rule can be mutated from p to any value in the interval $[1, c]$.

Delete Rule. The function deletes a rule in the system. All the rules can be deleted in the system except the rule in the input neurons and output neuron.

Delete Synapse. The function selects a neuron in the system and deletes an outgoing synapse. All neurons can be selected except the output neuron and the input neuron if the input neuron only has one outgoing synapse.

Add Synapse. The function selects a neuron in the system and adds an outgoing synapse connected to another neuron. All neurons can be selected as the source of the new synapse except the output neuron. The destination of the synapse cannot be the same as the source, and cannot be the input or output neuron.

6 IMPLEMENTATION

Details of the implementation of the framework are given in this section, while experiment results are in Section 7. Experiment sources (e.g. codes, test files) are available from the corresponding author upon request.

Initial SN P systems. The framework will be tested using SN P systems of two functions: ADD and SUB. For each function, there will be three SN P systems, the first one (labeled Type A) having 100% fitness but with extra rules, the second one (Type B) having less than

100% fitness, with extra rules, and the third one (Type C) having less than 100% fitness, but with extra neurons and extra rules. A summary is shown in Table 1.

By testing SN P systems of Type A, the framework is tested on how well it reduces the number of rules in the SN P system, without having to worry about finding a 100% fit SN P system. In Type B, the framework is tested on its capability of finding 100% fit SN P systems from a system with less initial fitness. In Type C, the framework is tested on finding 100% fit SN P systems as well, but how it handles extra neurons is also observed.

The ADD and SUB SN P systems mentioned are based from [5]. The ADD SN P system from [5] had 5 rules, and SUB had 15. The initial SN P systems are shown in Figures 13, 14, 15, 16, 17, and 18 in the Appendices. A table of the fitness and the number of rules of the SN P systems are shown in Table 2. Ideally, the genetic algorithm should reduce the number of rules of the SN P systems to 6 for ADD and 16 for SUB because of the rule from the added output neuron.

Chromosome Representation. The chromosomes in the context of this work are SN P systems. The SN P systems are encoded in a way that can be easily mutated and recombined with other systems. An SN P system is stored as a collection of vectors and matrices in a text file. The file contains the following information: number of neurons, number of rules, input vector, adjacency matrix, rule representation, status vector, spiking vector, and configuration vector. In Appendix C a representation of the SN P system in Figure 5 is shown in Table 9.

Population Initialization. This is the first step in the framework, as shown in Figure 1. The population of the first generation is generated by taking the initial SN P system and mutating it to produce a child. This process is repeated until enough children have been generated for the initial population. Mutation rates of the mutation functions are in Table 10 (Appendix D), with initial population at 100 chromosomes (see Section E).

More than one mutation can occur per child, occurring in the order listed above, e.g. an SN P system undergoes mutated regular expression first, followed by deleted synapse. The values listed are arbitrarily chosen. All experiments of the same SN P system type (see Table 1) use the same initial population.

Parent Selection. After every mutation, all chromosomes are simulated with the input spike trains from S. The produced output spike trains are compared with the output spike trains from S using LCS. The fitness of an SN P system is determined by the score from LCS.

After fitness calculation, the parents are selected.

For Fitness-proportionate, the probability of a chromosome being selected as a parent is described by Equation 1. Half are selected as parents.

The same goes for Fitness-proportionate with Elitism, except the SN P systems are ranked by fitness and number of rules, and the top 10 are directly added to the next generation as children, without any modifications.

After Parent Selection follows Crossover, Mutation, and Validation, as shown in Figure 1.

Crossover. Once the pool of parents is complete, the parents are paired up at random with each other until enough pairs are available to populate the next generation. For Fitness-proportionate and Fitness-proportionate with Elitism, 25 and 20 pairs are used, respectively.

A parent can be paired up more than once. Each pair will produce two children. All pairs have a chance to undergo crossover. SN P systems that are confirmed to undergo crossover are equally likely to get either one of the two crossover functions. Only one crossover function can be selected per pair at most. For pairs that do not get the chance to undergo crossover, their children will be an exact copy of themselves. The crossover rate assigned per crossover function is arbitrary.

Mutation. Once enough children have been produced from crossover, the children can undergo mutation. Each child can be mutated once at most. Mutation rates per mutation function are shown in Table 11 (Appendix D).

After mutation, the child is added to the next generation. Children who do not undergo mutation proceed to validation. The mutation rates per mutation function are arbitrary.

Validation. Chromosomes produced after crossover and mutation might be invalid. In this experiment, an SN P system is considered invalid if non-deterministic rule sets exist, or if there is no path from the input neurons to the output neurons.

For the former, detecting non-deterministic rule sets is easy, since the form of the regular expressions is a^i. For any given neuron, if two rules exist with the same regular expression, then the rule set of the neuron is non-deterministic, therefore the SN P system is considered invalid. For the latter, path-checking is done by running a depth first search on every input neuron to check if a path exists to the output neuron.

Halting Condition. The diamond in Figure 1 represents the halting condition. The algorithm will cycle between simulation, fitness calculation, parent selection, crossover, and mutation. Upon finding an SN P system with 100% fitness, the algorithm will not stop hoping to get another SN P system with 100% fitness but with less number of rules. The algorithm will only halt once the maximum number of generations is reached.

Type	100% fitness?	Extra rules?	Extra neurons?	Figures	
A	Yes	Yes	No	(addA) Figure 13	(subA) Figure 16
B	No	Yes	Yes	(addB) Figure 14	(subB) Figure 17
C	No	Yes	Yes	(addC) Figure 15	(subC) Figure 18

Table 1: A table showing the three types of SN P systems to be tested

SN P system	Fitness	Number of Rules
addA	100	9
addB	29	9
addC	55	14
subA	100	21
subB	40	21
subC	57	22

Table 2: The fitness, number of neurons and number of rules of the initial SN P systems.

	15% 5%	20% 10%	25% 15%	30% 20%	35% 25%
addA	10	10	10	10	10
addB	3	4	4	4	4
addC	0	1	0	0	0
subA	10	10	10	10	10
subB	0	0	0	0	0
subC	0	0	0	0	0

Table 3: Number of successful runs for experiments with the fitness-proportionate selection method.

Setup of the framework for our experiments are in Appendix E.

7 RESULTS AND DISCUSSION

The results of the experiments are discussed in this section. See Section 6 for details of the experiments and Table 1 for the details of Π_{init} used. The number of successful runs, i.e. runs yielding 100% fitness, per experiment is shown in Tables 3 and 4. Figures 28, 29, 3, and 4 use the same legend as Figure 2. The mutation and crossover rates in Figures 28, 29, 3, 4, and Tables 5, 6, and 7 are from Table 12.

ADD of Type A. Since all Π_{init} of Type A have initially 100% fitness (see Table 1) all runs of experiments for such Π_{init} retained their fitness (see Tables 3 and 4). All experiments were able to achieve the minimum number of rules of 6 for ADD. In Table 5, on average, the experiments show that the framework performed better with fitness-proportionate with elitism, consistently achieving the minimum 6 rules unlike fitness-proportionate. Experiments with 30% mutation rate and 20% crossover rate had the same average for both selection methods. Shown in Appendix F, the framework generated Figure 19, the Π_{final} with only 6 rules, on both methods with Figure 13 as Π_{init} .

	15% 5%	20% 10%	25% 15%	30% 20%	35% 25%
addA	10	10	10	10	10
addB	1	1	1	1	3
addC	0	0	0	0	1
subA	10	10	10	10	10
subB	0	0	0	0	1
subC	0	0	0	0	1

Table 4: Number of successful runs for experiments using fitness-proportionate with elitism.

Mutation & Crossover Rate	FP	FP with Elitism
15% 5%	6.4	6
20% 10%	6.2	6
25% 15%	6.3	6
30% 20%	6.0	6
35% 25%	6.3	6

Table 5: Average minimum number of rules of each successful experiment for ADD of Type A.

Mutation & Crossover Rate	FP	FP with Elitism
15% 5%	7	6
20% 10%	8	6
25% 15%	7.75	6
30% 20%	6.25	6
35% 25%	6.5	6

Table 6: Average minimum number of rules of each successful experiment for ADD of Type B.

ADD of Type B. From Tables 3 and 4, all experiments had at least one successful run, with fitness-proportionate performing better than fitness-proportionate with elitism for most experiments. However, Table 6 shows that on average fitness-proportionate with elitism is better at reducing the number of rules. In Appendix F, the framework generated Figure 19 for both selection methods, with Figure 14 as Π_{init}.

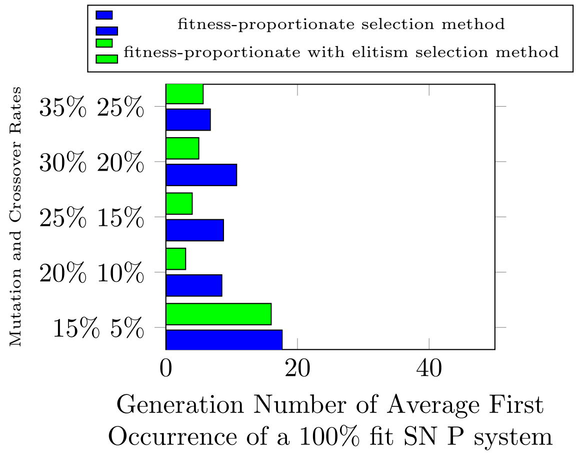

Figure 2: Generation number of average first occurrence of a 100% fit Π_{final} of each successful experiment for ADD of Type B.

In Figure 2, fitness-proportionate with elitism finds the first Π_{final} with 100% fitness earlier than its counterpart for all mutation and crossover rate pairs on average, around the 3rd generation for the 20% and 10% pair, and at 16th generation for 15% and 5% pair. Fitness-proportionate on average finds a 100% fit Π_{final} at generation 6.75 for 35% and 25% pair, and on average at generation 17.66 for 15% and 5% pair.

ADD of Type C. Only two runs found a 100% fit Π_{final}, at fitness-proportionate method of 20% and 10% pair, and at fitness-proportionate with elitism method of 35% and 25% pair. For the successful run with the former method, the framework was unable to reduce the number of rules, however it was able to find a 100% fit Π_{final}. The latter method performed better, reducing the number of rules to 10 from 14. In Appendix F the successful Π_{final} from both selection methods are shown in Figures 20 and 21.

Fitness-proportionate with elitism for the rate pair 20% and 10% achieved a fitness of about 67%, while the other method achieved around 80% fitness for the rate pair 35% and 25%. The framework found the 100% fit Π_{final} at generations 50 and 39 for fitness-proportionate and fitness-proportionate with elitism, respectively. See Figures 28 and 29 in Appendix G for details.

SUB of Type A. Similar to ADD of Type A, all runs were able to find a 100% fit SN P system. Table 7 shows that fitness-proportionate with elitism consistently performs better than fitness-proportionate, e.g. on average, a Π_{init} with 21 rules was reduced by the former method to as few as 16 rules (35% and 25% rate pair), while the latter method only reduced it to 19.2 rules (30% and 20% rate pair). Figures 22 and 23 in the Appendix show some subA Π_{final}.

Mutation & Crossover Rate	FP	FP with Elitism
15% 5%	19.5	17.2
20% 10%	19.5	16.2
25% 15%	19.6	16.4
30% 20%	19.2	16.3
35% 25%	19.5	16.0

Table 7: Average minimum number of rules of each successful experiment for SUB of Type A for both selection methods.

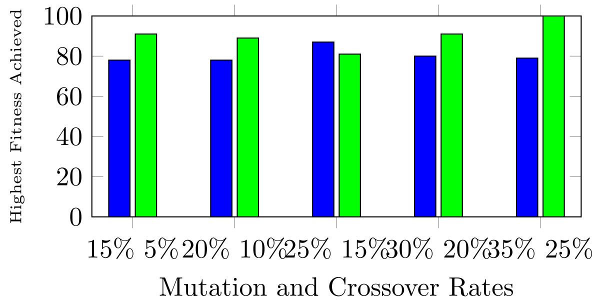

Figure 3: Highest fitness achieved for SUB of Type B for both selection methods and each pair of rates.

SUB of Type B. Successful runs of SUB are more difficult to achieve due to their larger size, e.g. only one run found a 100% fit Π_{final} with fitness-proportionate with elitism for the 35% and 25% rate pair, reducing the number of rules from 21 to 19. Figure 3 shows that fitness-proportionate runs reached at least 78% fitness (e.g. at 15% and 5% rate pair), with the highest fitness of 87% at 25% and 15% rate pair. The single Π_{final} with 100% fitness was found at generation 48. The Π_{final} with 87% and 100% fitness are shown in Figures 24 and 25 in Appendix F.

SUB of Type C. As in SUB of Type B, only one run achieved a Π_{final} of 100% fitness under the same selection method, mutation and crossover rates. The framework was able to reduce the number of rules from 22 to 17. In Figure 4 the fitness-proportionate run achieved a max fitness of 93% at 35% and 25% rate pair. The Π_{final} with 100% fitness was from generation 27 at the 35% and 25% rate pair. The Π_{final} with 100% and 93% fitness are shown in Figures 27 and 26 in Appendix F.

In summary, it is clear that the fitness-proportionate with elitism selection method performs better in reducing the number of rules. In ADD of Type A and Type B, the experiments under the mentioned selection method showed a consistent result of 6 rules, compared to its

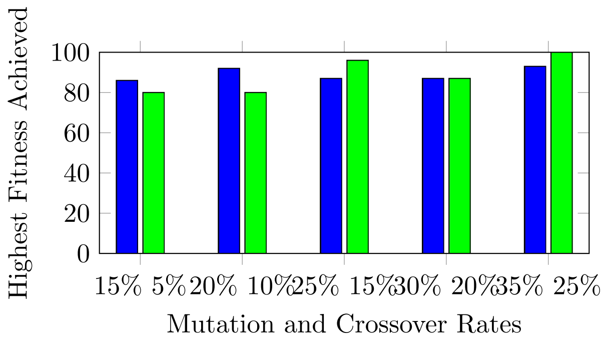

Figure 4: Highest fitness achieved for SUB of Type C for both selection methods for each pair of rates.

counter part without elitism. At ADD of Type C, while fitness-proportionate was able to find a 100% fit SN P system, it was unable to reduce the number of rules, while its counterpart was able to reduce the rules by 4. At SUB of Type A, the same goes for all mutation rate and crossover rate pairs, as seen in Table 7. For SUB of Type B and Type C, a comparison cannot be made as no runs from fitness-proportionate were successful.

Fitness-proportionate with Elitism outperforming just fitness-proportionate might be because elitism incentivizes rule reduction for SN P systems which already have 100% fitness, ranking the chromosomes by fitness and number of rules in every generation and directly copying them over to the next generation, making sure that 100% fit SN P systems with lesser rules survive. While there is no mutation function that increases the number of rules, this behaviour is not incentivized in fitness-proportionate *without* elitism. With elitism, the only factor accounted in selection is the fitness of each chromosome. When two chromosomes have the same fitness but one has fewer rules than the other, the selection method does not favour the one with fewer rules.

Inspecting the number of successful runs, both selection methods tied at Type A SN P systems, and at ADD of Type C. Fitness-proportionate has more successful runs in add of Type B compared to its counterpart. On SUB of Type B and C however, fitness-proportionate with elitism has at least one successful run, while fitness-proportionate has none. For the mutation and crossover rate pairs, fitness-proportionate with elitism had more success with the 35% and 25% rate pair for all Π_{final} with at least one successful run, and fitness-proportionate with the 20% and 10% rate pair for 4 out of 6 of the Π_{final} with successful runs.

In [11], the fitness-proportionate selection method performed the best among two other selection methods, in terms of the highest average fitness. The selection methods in [11] are: selecting the top 50% of the population as parents; 25% using fitness-proportionate; and the top 25% of the population + 25% using fitness-proportionate. In this paper, however, fitness-proportionate alone is outperformed by fitness-proportionate with elitism. When it comes to the SN P systems, [11] found that finding RSSN P systems of at least 90% fitness was challenging for addition and most especially for subtraction.

The initial RSSN P systems that [11] used had extra rules and neurons, with fitness less than 100%, similar to Type C in this paper. In their framework, only 4 out of 64 experiments yielded an RSSN P system with at least 90% fitness, and SUB had none, with the highest at only 88%. Looking at the Type C results in this framework, the framework was able to find SN P systems of 100% fitness, but only 2 out of 100 were successful for ADD of Type C, and only 1 for SUB of Type C. [11] says that the difficulty might be caused by the complexity of rules and the large number of rules and neurons in the tested addition and subtraction RSSN P systems, as they had success with RSSN P systems for the bitwise operators NOT, AND, and OR. In this framework, the difficulty in finding SN P systems with 100% fitness for Type C might have similar reasons, especially due to the larger number of rules of SUB compared to ADD.

Referring back to Section 4 the framework is successful in reducing the number of rules from a system with 100% fitness, as shown in experiments with Type A. The number of rules of Type B and Type C were also reduced, as long as 100% fit Π_{final} were found. The framework is also successful in finding Π_{final} with 100% fitness, as shown in experiments with Type B and C.

In systems of Type C, we see that the framework did not fully isolate extra neurons by removing all its rules or detaching it (i.e. removing synapses) from the rest of the system. However, it was able to modify the extra neurons such that it no longer contributes to the output to the system. The framework was able to render an extra neuron useless by removing all the rules in that neuron, or removing enough synapses that it does not affect the system any more, or by changing its spiking rules to forgetting rules, or sometimes a combination of these, as in Figures 21 and 27 in Appendix F.

The only successful run for ADD of Type C using fitness-proportionate with elitism shows an interesting result, as seen in Figure 21. This Π_{final} appears far from ideal as many of its rules are not what are expected when compared to the Π_{final} in Figure 19, but both have 100% fitness.

8 FINAL REMARKS

Based on experiments (Section 6) and results (Section 7) our framework found SN P systems with 100% fitness and fewer number of rules from an initial SN P system Π_{init}. The evolving SN P system framework using a genetic algorithm is successful in finding SN P systems with higher fitness, or finding systems with fewer number of rules from a Π_{init} with 100% fitness.

For experiments of Π_{init} with 100% fitness, reducing the number of rules have shown desirable results. For experiments of Π_{init} with less than 100% fitness, finding a 100% fit Π_{final} becomes more difficult as the size of the system increases. While the framework was also successful in this matter, its low success rate is not enough to say that the framework is efficient in finding SN P systems with 100% fitness.

The framework, while successful with the given parameters and SN P systems, may not be as effective for other SN P systems with more complex rules and a larger topology. Possible improvements in this framework are mentioned next.

Exploring more parameters can be useful in improving the framework. Since the framework only uses LCS, as in [3] and [7], to measure the fitness of an SN P system, it is encouraged to use other fitness metrics. The same goes for the selection methods, as fitness-proportionate is primarily used, only with the addition of elitism. [7] used Tournament Selection as one of their selection methods for their framework for PSN P systems. Additionally, this work had success with fitness-proportionate by including elitism. The inclusion of elitism to other selection methods might prove useful, as it is able to incentivize the behaviour of reducing the number of rules in the system. Varying the top number of chromosomes directly copied to the next generation might also be considered.

Varying the population size for each initial SN P system can also be considered, like in [3], where the population size is set differently based on the size of each SN P system.

It is also encouraged to explore SN P systems with other forms of regular expressions. The framework in this work only deals with rules of the form $E/a^c \to a^p; 0$ where E is of the form a^i. Also, exploring SN P systems that perform operations with more than two inputs, e.g. sorting, decision problems, is worth investigating.

REFERENCES

[1] Cabarle, F. G. C., Martínez-del Amor, M. Á., Zeng, X., and Adorna, H. N. Evolving Spiking Neural P Systems. https://www.gcn.us.es/files/bwmc2018-evolsnp-present.pdf, 2018. 16th Brainstorming Week on Membrane Computing (BWMC16), University of Seville, Seville, Spain.

[2] Carandang, J., Villaflores, J. M. B., Cabarle, F. G. C., Adorna, H. N., and Martinezdel-Amor, M. Cusnp: Spiking neural p systems simulators in cuda. *ROMJIST 20*, 1 (2017), 57–70.

[3] Casauay, L. J., Macababayao, I. C. H., Cabarle, F. G. C., de la Cruz, R. T. A., Adorna, H. N., Zeng, X., and Martinez-del Amor, M. A. A framework for evolving spiking neural p systems. In *ACMC2019: The International Conference on Membrane Computing (Asian Branch)*. 2019. (in press) International Journal of Unconventional Computing, Old City Publishing.

[4] Eiben, A. E., and Schoenauer, M. Evolutionary computing. *Information Processing Letters 82*, 1 (2002), 1–6.

[5] Gutiérrez Naranjo, M. Á., and Leporati, A. Performing arithmetic operations with spiking neural p systems. *Proceedings of the Seventh Brainstorming Week on Membrane Computing, vol. I, 181-198. Sevilla, ETS de Ingeniería Informática, 2-6 de Febrero, 2009* (2009).

[6] Ionescu, M., Păun, G., and Yokomori, T. Spiking neural p systems. *Fundamenta informaticae 71*, 2, 3 (2006), 279–308.

[7] Juico, J. G. E., Silapan, J. L., Cabarle, F. G. C., and Adorna, H. N. Evolving spiking neural p systems with polarization. *CS199 manuscript* (2019).

[8] Leporati, A., Zandron, C., Ferretti, C., and Mauri, G. On the computational power of spiking neural p systems. *Proceedings of the Fifth Brainstorming Week on Membrane Computing, 227-245. Sevilla, ETS de Ingeniería Informática, 29 de Enero-2 de Febrero, 2007* (2007).

[9] Miller, G. F., Todd, P. M., and Hegde, S. U. Designing neural networks using genetic algorithms. In *ICGA* (1989), vol. 89, pp. 379–384.

[10] Mitchell, M. *An introduction to genetic algorithms.* MIT press, 1998.

[11] Moredo, C. A. A., Supelana, R. C. J., Cailipan, D. P., Cabarle, F. G. C., de la Cruz, R. T. A., Adorna, H. N., Zeng, X., and Martinez-del Amor, M. A. A framework for evolving spiking neural p systems with rules on synapses. *ACMC2019: The International Conference on Membrane Computing (Asian Branch)* (2019).

[12] Păun, G. Spiking neural p systems. a tutorial. *Bulletin of the EATCS 91* (2007), 145–149.

[13] Sheppard, S. Eiji nakatsu: Lecture on biomimicry as applied to a japanese train. https://labs.blogs.com/its_alive_in_the_lab/2012/04/biomimicry-japanese-train.html, 2012.

[14] Song, X., and Wang, J. An approximate algorithm combining p systems and active evolutionary algorithms for traveling salesman problems. *International Journal of Computers Communications & Control 10*, 1 (2015), 89–99.

[15] Zhang, G., Gheorghe, M., Pan, L., and Perez-Jimenez, M. J. Evolutionary membrane computing: a comprehensive survey and new results. *Information Sciences 279* (2014), 528–551.