Evolving Spiking Neural P Systems with Polarization

Jules Gerard E. Juico, Jerico L. Silapan, Francis George C. Cabarle*
Ivan Cedric Macababayao, Ren Tristan De la Cruz
University of the Philippines Diliman
Quezon City, Philippines

ABSTRACT

In this work, we introduce a representation of spiking neural P systems with polarization (or PSNP Systems) and an algorithm for simulation. An existing representation and algorithm for spiking neural P systems is modified to handle neurons with polarizations, neurons firing spikes and charges, and rules are checked using the charge of the neuron instead of regular expressions. In addition to the representation and algorithm, this work also presents a genetic algorithm framework (GA framework) that aims to reduce the number of resources (rules and synapses) of an existing PSNP system. We use the framework on PSN P systems that perform bitwise AND and OR. The GA framework will have two selection methods, Fitness Proportionate and Tournament Selection. A discussion on the effectiveness of the framework in obtaining a PSN P System with less number of rules and synapses will be done at the end.

KEYWORDS

spiking neural p systems, neural p systems, polarization, genetic algorithm, simulation

1 INTRODUCTION

Learning starts with us imitating what our parents or guardians do. By observing their movements or by listening and repeating the words they say, we slowly understand and gain knowledge about simple to complicated things about the world. We then broaden our knowledge by reading books and listening to valuable lessons from our teachers and professors. But it does not end there. Driven by curiosity and craving for knowledge led us to discover information beyond present sources has to offer and for this study, we focus on the very place where us, humans, interact all the time, nature.

Biological processes such as changes in individuals due to natural selection and cooperation of millions of neurons in the brain has been an inspiration for many and it gave rise to the development of natural computing. This lead to further understanding algorithms present in the nature which can be a guide to produce better algorithms and spark up interesting ideas [7].

Membrane computing is one of the youngest areas of natural computing. Computational model are inspired by cells in terms of structure and functioning. It concerns distributed and parallel computing models called P systems. P systems have three main categories that differ in structure: cell-like P systems which are hierarchically arranged, tissue-like P systems which are arranged in an undirected graph, or neural-like P systems which are arranged in a directed graph [7].

A work by [4] is about a kind of neural-like P system called Spiking Neural P Systems or SN P systems. SN P systems are third generation neural networks that have neurons placed in the nodes of a directed graph that utilized spikes to send signals from one node to another similar to how nerve cells communicate with one another. These neurons are connected by edges called synapses where the spikes are transmitted to and from. Each neuron contains rules that will determine the spikes to be sent by the neuron to all neighbouring neurons.

A recent work by [9] is about a variant of the SN P system called Spiking Neural P Systems with Polarization. The distinct difference between the two would be the use of polarization to determine whether a rule inside a neuron will apply or not.

Simulation algorithms for SN P systems has been present and an inspiration for this work is the work of [1] which is a algorithm to simulate SN P systems using a GPU. An algorithm for simulation is a great help as it reduces the time to simulate. In addition, longer input spike trains and larger systems can be simulated with the help of a computer.

Another area in natural computing is evolutionary computing which is inspired by biological evolution. This opened a new area to which problems can be solved. As it suggests, it is a process of continuous selection, mutation, and crossover until a certain criterion has been met.

Both simple and complex PSN P systems can be performed using a pen and paper given a decent amount of time. For complex PSN P systems, it is not practical to perform it by hand as time is important and mistakes could occur. Having a representation and an algorithm

*corresponding author: fccabarle@up.edu.ph. 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.

for PSN P systems can assist future works involving complex PSN P systems. The first main objective for this work is to create a tool to assist individuals who want to simulate PSN P systems using a computer. At the moment this work is done, there exists no other studies about transforming a PSN P system so that it would compute approximately the same output where the system contains the same or reduced number of synapses and rules. The number of possible combinations of synapses and rules forms a large solution space that would make it difficult to find a PSN P system using a brute force approach. As discussed above, genetic algorithms are suited to find solutions in a large solution space. Another motivation of this work is: How do we create a framework for PSN P systems using genetic algorithms so that we could transform the system to perform its function but with a possibly reduced number of synapses and rules?

In this work we introduce a modified version of the existing algorithm in [1]. Modifications are applied to be able to simulate PSN P systems. To be specific, in this work introduce the following results: **PSN P Representation**, **PSN P Simulator Algorithm**, **PSN P Simulator Program**. The PSN P representation is a modified version from the work of [1]. New vectors are modified to handle the polarization of neurons and rules are now checked using charges instead for regular expressions. Similarly, the PSN P simulation algorithm is also inspired from the work of [1]. Changes will be done since the this work is concerned with PSN P systems.

The next main objective is to design a genetic algorithm framework for reducing the number of rules and synapses of a PSN P System while still making it perform correctly the operation it supposed to do. The genetic algorithm has a framework defined later in Section 3.

For this work, we discuss definitions and details of spiking neural P systems with polarization or PSN P systems in Section 2. Section 3 will discuss details of the genetic algorithm and theoretical framework of this work, including the scope and limitations of this work. Related literature of this work will be discussed in Section 4. A discussion on how the SNP algorithm from the work of [1] is modified for PSN P systems can be found on Section 5. Details of the specific parts of the genetic algorithm framework are discussed in Section 6. How the experiments are conducted is presented in Section 7 and an analysis of the said experiments are found right after in Section 8. The conclusions and future work from our experiments are on Section 9.

2 PRELIMINARIES

Spiking Neural P Systems with Polarization

In this work, we deal with a variant of spiking neural P systems with polarization or PSN P systems. PSN P systems are similar to SN P systems such both of them have neurons that consume and produce spikes. The main difference between them is that a new mechanism is used for determining whether a rule applies or not. Neurons and rules are now associated with a charge, rather than using regular expressions. This change is inspired by the nature of neurons having charges and avoids the NP-complete problem of deciding whether or not a regular expression is accepted [9].

DEFINITION 1 (PSN P SYSTEM). *As defined from [8], spiking neural p system with polarization of degree $m \geq 1$ is a construct of the form*

$$\Pi = (O, \sigma_1, \sigma_2, , \sigma_m, syn, in, out),$$

where

(1) $O = a$ *is the singleton alphabet (a is called spike);*
(2) $\sigma_1, \sigma_2, ..., \sigma_m$ *are neurons, of the form*

$$\sigma_i = (\alpha_i, n_i, R_i), 1 \leq i \leq m,$$

where:
   (a) $\alpha_i \in \{+, 0, -\}$ *is initial polarization of neuron* σ_i*;*
   (b) $n_i \geq 0$ *is the initial number of spikes contained in* σ_i*;*
   (c) R_i *is a finite set of rules of the following two forms:*
      (i) $\alpha/a^c \rightarrow a; \beta$, *for* $\alpha, \beta \in \{+, 0, -\}$, $c \geq 1$ *(spiking rules);*
      (ii) $\alpha/a^s \rightarrow \lambda; \beta$, *for* $\alpha, \beta \in \{+, 0, -\}$, $s \geq 1$ *(forgetting rules);*
(3) $syn \subseteq \{1, 2, ..., m\} \times \{1, 2, ..., m\}$ *with* $i \neq j$ *for each* $(i, j) \in syn$, $1 \leq i, j \leq m$ *(synapses between neurons);*
(4) $in, out \in \{1, 2, , m\}$ *indicate the input and the output neurons, respectively.*

Unlike SN P systems, the neurons and rules of a PSN P system each contain a charge. These charges replace regular expressions in determining if a rule is applicable or not. The same types of rules are in PSN P systems, namely, the spiking rules and the forgetting rules. In the same case to SN P systems, a rule of a neuron must be applied whenever possible.

A spiking rule of the form $\alpha/a^c \rightarrow a;\beta$ will apply if the neuron has the charge α and contains k spikes, such that $k \geq c$. Once applied, the neuron will consume c spikes and produce a spike carrying the charge β. The spike is sent to all neurons that has a synapse connecting the neuron that produced the spike.

A forgetting rule of PSN P systems does not need to have the exact number of spikes for the rule to apply. It will apply as long as the neuron contains k spikes and that $k \geq s$. Moreover, aside from the neuron consuming s number of spikes, it will also send a charge β to all neighbouring neurons. Once the charges are received, a computation of these charges is done following the steps enumerated sequentially:

(1) several positive charges (+), several neutral charges (0), several negative charges (-) lead to one positive charge (+), one neutral charge (0), one negative charge (-), respectively.
(2) a positive charge (+) and a negative charge (-) cancel each other and give a neutral charge (0);
(3) a positive charge (+) and a negative charge(-) is not changed by a neutral charge (0).

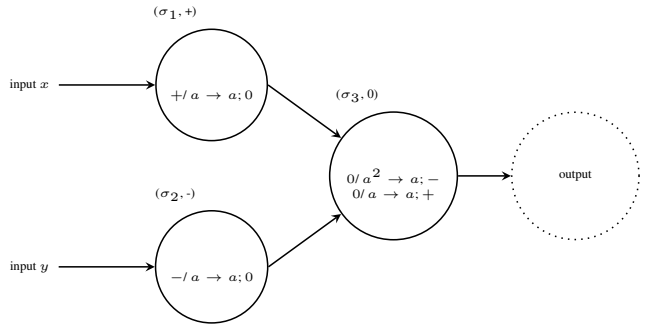

Figure 1: An example of an PSN P system for the logical OR function

Alike SN P systems, neurons will work in parallel and will be synchronous using a global clock. If several rules inside a neuron can can be applied, then the rule to be used is chosen non-deterministically.

The *configuration* of the system is described by both the number of spikes and the charge of each neuron, and as a result, the *initial configuration* of the system is $C_0 = \langle\ n_1,\ n_2,...,n_m;\alpha_1,\alpha_2,...,\alpha_m\rangle$

We denote a *transition* between two configuration, say C_1 and C_2 using $C_1 \Rightarrow C_2$. A *computation* is any sequence of transitions that starts from the initial configuration. We say that a computation is *successful* and *halting* if the system reaches a configuration in which no rules can be applied in any neurons.

Input and output neuron of a PSN P system may or may not be defined, thus there are several ways to define the *result of a computation*. To elaborate, a PSN P system can be a generative, an accepting, or a computing device. Input neuron *in* is ignored for a generative device while the output neuron *out* is ignored. But when both input and output neuron is considered, the PSN P system can be used to compute numerical functions.

step	input x	input y	σ_1	σ_2	σ_3	output
0	a	a	0, +	0, -	0, +	0
1	0	0	a, +	a, -	0, +	0
2	0	0	0, +	0, -	a^2, 0	0
3	0	0	0, +	0, -	0, 0	a

Table 1: A computation of the PSN P system in Figure 1 with (bit) inputs $x = 1$ and $y = 1$.

An example of a PSN P system can be seen in Figure 1. If a spike is entered from each input, a computation is provided on Table 1. These two spikes enter the system through input x and input y. Initially, all three neurons do not contain any spikes, and neurons σ_1 and σ_3 contain a positive charge while σ_2 contain a negative charge. At step 1, σ_1 and σ_2 will contain a spike each and at the next step, given the charges they contain, the rules will accept the spikes and the charges which will lead to producing a spike for each neuron. The spikes produced by both σ_1 and σ_2 is fired to σ_3. Neurons σ_1 and σ_2 also fires a positive charge and a negative charge, respectively. Following the computation for charges, σ_3 contains a positive charge and receives a positive charge from σ_1 which first leads to one positive charge. The negative charge sent by the neuron σ_2 then cancels out the positive charge. This will result to σ_3 containing a neutral charge as well as two spikes at step 2. At step 3, the rule $0/a^2 \rightarrow a; 0$ accepts the neutral charge and consumes the two spikes to produce a single spike sent out as the output of the system. The output completes the computation of the logical OR function, such that the equation $1OR1$ produces 1 as how the system outputs a single spike with an input two separate spikes.

Genetic Algorithm

Genetic algorithms are an approach to solving problems by taking an inspiration from biological evolution. According to Melanie [5], evolution is a method of searching through a large number of constantly changing possibilities for solutions. Genetic algorithms are patterned after evolution to solve computational problems by repeatedly searching and evolving candidate solutions for the problem. By evolving candidate solutions, genetic algorithms create solutions and narrow them down which makes them suitable for large solution spaces. These candidate solutions are called chromosomes, often encoded as bit strings, which are composed of genes that are usually either bits or blocks of bits that represent an element

of the candidate solution. Alleles are the alphabet of a gene, which in the case of a gene represented as bit strings, is either 1 or 0.

Figure 9 shows the flow of how genetic algorithms. A population is first generated by inputting chromosomes. A fitness function assigns a score to a chromosome which can be used to determine which chromosomes will crossover and mutate, or which chromosome will be selected as the solution. A stop criterion is provided which may be a certain number of iterations of crossing over and mutating or if a chromosome has achieved certain fitness score. The stop criterion is provided to determine whether a chromosome is selected to become the output or if the genetic algorithm will continue to evolve the chromosomes. In the case of the latter, chromosomes are selected to be evolved for the population of the next generation of chromosomes. To evolve chromosomes, genetic algorithms utilize crossover and mutation to the chromosomes in a population. Crossover is the exchange of genes between two chromosomes while mutation consists of flipping random genes of a chromosome.

3 GENETIC ALGORITHM AND THEORETICAL FRAMEWORK

Details of the genetic algorithm framework, continuing the main objectives from Section 1, is as follows. Given an initial deterministic PSN P $\Pi_{init} = (O, \sigma_1, \sigma_2, ..., \sigma_m, syn, in, out)$ and a set of finite triplets $S = \{(a_1, b_1, c_1), (a_2, b_2, c_2), ..., (a_n, b_n, c_n)\}$ where

(1) $a_i, b_i, c_i \in \{0, 1\}^+$
(2) $\Pi_{init}(a_i, b_i) = c_i$

construct a genetic algorithm that outputs a deterministic PSN P $\Pi_{final} = (O, \sigma_1^{'}, \sigma_2^{'}, ..., \sigma_m^{'}, syn, in, out)$ such that the output of $\Pi_{final}(a_i, b_i)$ is within a certain fitness value relative to b_i and $size(\Pi_{final}) \leq size(\Pi_{init})$, where size is a computation of the number of rules and synapses.

A diagram for the proposed framework for evolving PSN P systems is shown in Figure 2. The framework is to work similarly to how genetic algorithms work as shown in Figure 9 and how genetic algorithm is applied to neural networks in [3]. The framework contains the same elements as most genetic algorithms with a few modifications to be able to evolve PSN P systems. An initial PSN P system is entered as input to the genetic algorithm along with a pair of input and output spike trains. Initially, population of PSN P systems are generated once. The fitness scores of the PSN P systems in the population are computed by using the input spike trains to generate an output spike train for each PSN P system and comparing them to the initial output spike train

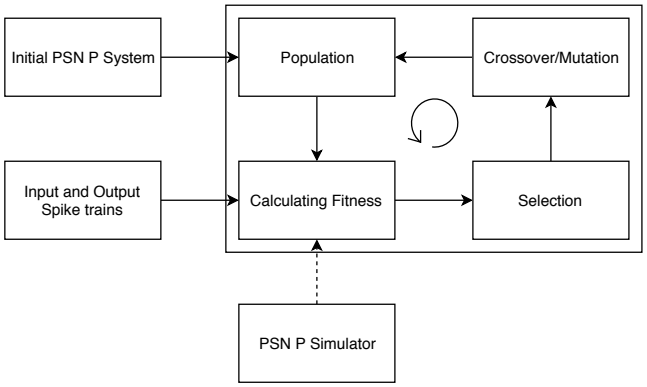

Figure 2: A framework for evolving PSN P systems using genetic algorithms

entered to the genetic algorithm. The algorithm then selects PSN P systems for crossover and mutation, based on their fitness score. The resulting PSN P systems are placed back to the population and will continue to iterate over these cycle of steps until the stop criterion is met. Once the cycle stops, the fittest PSN P system is selected and becomes the output of the genetic algorithm.

The systems to be used for this work are specific variant of the Spiking Neural P Systems with Polarization, or PSN P, wherein the following restrictions are used: **Fixed initial neuron charges**: All neurons in the system have preset and initial charges. **Fixed number of neurons**: The PSN P system used have a constant number of neurons. The number of neurons do not increase or decrease during and after evolution with the genetic algorithm. **Synchronous mode**: Neurons in the PSN P system work in parallel and will be synchronous using a global clock. **Constant input and output neurons**: Input and output neurons will remain the same during and after evolution. **Computing device**: Spike trains are required for the input neurons. The output neuron will produce a spike train.

4 RELATED WORK

[6] evolved weights of a neural network with fixed units called neurons and connections using genetic algorithm. The weights are represented or encoded using a list of real numbers. The weights are then subjected to operations of GA such as mutation, crossover, and selection using an evaluation function. This concept helps to guide encoding PSN P systems for input, transformation and output of the genetic algorithm.

[1] simulated SN P systems both with CPU and GPU simulators. The simulators used matrices to encode SN P systems by representing different elements of the SN

P systems, like the amount of spikes in a neuron at a certain time step, the delay of each rule in the neuron, the number of spikes consumed or sent, and the status, whether open or close, of each neuron at a certain time step. The study also provided a runtime comparison between the CPU and GPU simulators, and showed up to $50x$ speed up for the GPU simulator over the CPU simulator. The SN P simulators used in this study can be modified to simulate PSN P systems which could be used for the evolution of PSN P systems in our study.

[3] used a framework in optimizing the consumption of fuel and electricity in an absorption chiller system using genetic algorithm and neural networks. The framework used is visualized in Figure Here, the framework is structured similar to how the genetic algorithm can be applied in evolving PSN P systems.

In Figure 2, we can see the framework of [3] which can be used in this study. The encoding of PSN P systems for this framework takes inspiration for how [6] encoded the weights of a neural network for the genetic algorithm that was used. The SN P simulators in [1] could be used, with modifications to simulate PSN P systems instead of SN P. It could then be used for calculating the fitness for selection by checking if the output of a candidate PSN P system is within a certain range of the output of the initial PSN P system entered to the system.

5 ALGORITHM FOR PSN P SYSTEM

The algorithm of [1] was modified to create a simulator since the neurons are now associated with a charge and the mechanism to determine if a rule will apply is based on charges rather than using regular expressions. Let Π be an PSN P system with n number of neurons and r number of rules. The following definitions are used to represent Π.

Definition 2 (Configuration vector). *The configuration vector $CnV = < c_1,..., c_n >$, where c_i is the number of spikes contained in neuron i*

Definition 3 (Spiking Vector). *The spiking vector $SpV = < sp_1,.., sp_n >$, where for each $i \in \{ 1, 2,..,n\}$,*

$$sp_i = \begin{cases} 1, \textit{if a rule in neuron } i \textit{ is applied} \\ 0, \textit{ otherwise} \end{cases}$$

Definition 4 (Status Vector). *The status vector $StV = < st_1,..., st_n >$, where for each $i \in \{ 1, 2,..,n\}$,*

$$st_i = \begin{cases} 1, \textit{if neuron } i \textit{ is open} \\ 0, \textit{otherwise} \end{cases}$$

Definition 5 (Rule Representation Matrix). *The rule representation matrix $RRM = < r_1,..., r_r >$, where for each $i = 1,..., n$, $r_i = < sc, j, d', ca, cf, sf >$.*

sc is the number of spikes consumed by rule i

j is the neuron contained by rule i

$$d' = \begin{cases} -1, \textit{if rule } i \textit{ us not fired} \\ 0, \textit{if it is fired} \end{cases}$$

ca is the charge accepted by rule i

cf is the charge fired by neuron i

sf is the number of spikes consumed by rule i if it will fire

Definition 6 (Loss Vector). *The loss vector $LV = < l_1,..., l_n >$, where for each $i \in \{ 1, 2,..,n\}$, l_i is the number of consumed spikes by neuron i at the current step.*

Definition 7 (Gain Vector). *The gain vector $GV = < g_1,..., g_n >$, where for each $i \in \{ 1, 2,..,n\}$, g_i is the number of spikes sent by neighboring neurons to neuron i at the current time step.*

Definition 8 (Transition Matrix). *The transition matrix $TM = < tv_1,..., tv_r >$, where for each $i = 1,...,r$, $tv_i = < p_1,..., p_n >$ such that p_i is the number of spikes received by neuron i given that rule r fires.*

Definition 9 (Indicator Vector). *The indicator vector $IV = < iv_1,..., iv_r >$, where iv_i is equal to 1 if rule i will fire at the current time step and 0 otherwise.*

Definition 10 (Removing Matrix). *The removing matrix $RM = < rm_i,..., rm_r >$, where for each $i = 1,...,r$, $rm_i = rs_1,..., rs_n >$ such that rs_i is the number of spikes consumed by neuron i given that rule r fires.*

Definition 11 (Net Gain Vector). *The net gain vector $NV = < ng_i,..., ng_n >$, where ng_i is the number of spikes gained by neuron i at the current time step.*

Definition 12 (Charge Vector). *The charge vector $ChV = < ch_1,..., ch_n >$, where ch_i is the charge contained by neuron i.*

Definition 13 (Charge Transition Matrix). *The charge transition matrix $TchM = < tcv_1,..., tcv_r >$, where for each $i = 1,..,r$, $tcv_i = < cp_1,..., cp_n >$ such that cp_i is the charge received by neuron i if rule r fires.*

Output neurons will send spikes to the environment. For this experiment, the environment will be considered as another neuron labeled as σ_{env}.

PSN P systems used for the algorithm is represented using a string that has the following:

(1) Number of neurons including σ_{env}

(2) Number of rules
(3) Synapse Matrix
(4) Rule Representation Matrix
(5) Initial charges of neurons
(6) Input neurons

The addition Charge Vector and Charge Transition Matrix to the algorithm of [1] is to accommodate the charges that a PSN P system works with. The Charge Transition Matrix is computed the same way the Transition Matrix is, but instead of getting the spike of the rule, the charge fired by the rule is taken instead. The Charge Vector has the charges contained the neurons. The Charge Vector is computed by collecting the charges that will be received by the neuron if for the rules that will apply and the charge of the neuron itself. Then for each neuron, the computation for the charges is applied to the charges collected.

6 GENETIC ALGORITHM

Initial of Population

A given initial PSN P system called $\Pi_{init} = (O, \sigma_1, \sigma_2, , \sigma_m, syn, in, out)$ and a population size $psize$ is used to make an initial population $P = \{ \Pi_1, \Pi_2,...,\Pi_{psize}\}$.

To build the population, the initial PSN P system Π_{init} will undergo mutation which contains the following operations:

Fitness Calculation

Each member of population P will undergo simulation and will be given input and output spike trains $S = \{(a_1,b_1,c_1), (a_2,b_2,c_2),..., (a_{nt}, b_{nt}, c_{nt})\}$, where a and b are input spike trains, c is the respective output spike train of a and b, and nt is a user selected number. After the simulation for each pair of input spike trains a and b, each chromosome will have a an output spike train c_i' where $1 < i < psize$. These output spike trains c_i' will be compared to their respective c_i using a string matching method f. Using f, we can obtain a $score$. For every member of the population, the average of the $scores$ for all output spike trains will be computed. This average will be taken as the fitness of the chromosome. For this work, the string matching method used will be Longest Common Substring or LCS. LCS finds the longest spiketrain substring common to both c_i and c_i'. This method ensures that whenever we find a 100% match, then it means there exists a substring in c_i' that is exactly c_i.

Selection

After obtaining the population P fitness scores, we are now going to the process of selecting parents that will later undergo crossover. For this work, we used two selection methods based from [5]:

(1) Fitness Proportionate: Computes the sum of fitness scores of all chromosomes in the population and pseudo-randomly chooses a number between 0 and this sum. Then, for each chromosome in the population, add it's fitness to a partial sum until the partial sum exceeds the chosen number. The chromosome whose fitness score makes the partial sum exceed the chosen number is selected as a parent.
(2) Tournament Selection: Pseudo-randomly chooses 2 parents and an integer from 1 to 100. If the chosen integer is higher than the threshold set by the user, the chromosome with the higher fitness is selected, else, the chromosome with the lower fitness is selected as a parent for crossover.

These selection methods are used to introduce variety into the population while still attempting to select fit chromosomes as parents. This aids in avoiding reaching local maxima for fitness of chromosomes.

Crossover

After the selection process, the parent chromosomes are paired and a crossover is performed for each pair. A neuron is pseudo-randomly selected from both each parent chromosome and the following crossover processes may be performed:

(1) Rule Swap: All rules for each selected neuron will be removed and then added to the other neuron.
(2) Synapse Swap: All outgoing synapses of each selected neuron will be removed and then added to the other neuron. The receiving neurons of these synapses will remain the same.

Either of the two or both swaps may take place and is pseudo-randomly chosen based on thresholds set by the user.

Mutation

The child chromosomes from the crossover has a chance to be selected for mutation depending on the mutation rate parameter set by the user. If a child chromosome is selected, it will undergo mutation which contains the following operations:

(1) Rule Removal: Removes an existing rule from the system.
(2) Rule Addition: Adds a new rule that doesn't exist yet in the system.
(3) Rule Replace: Replaces an existing rule in the system with a new rule. A check is implemented

to make sure the the newly created rule will not be the same with the replaced rule.
(4) Synapse Removal: Removes an existing synapses in the system.
(5) Synapse Addition: Add a synapse that doesn't exist yet in the system.
(6) Synapse Replace: Replaces the receiving neuron of the synapse of a neuron in the system.

A child chromosome who is selected for mutation will be pseudo-randomly determined whether it will undergo a rule mutation, synapse mutation, or combination of both. It will also be pseudo-randomly determined whether a removal, addition, or replacement will take place for the selected mutation. The parameters for mutation such as elements of the rules and the neuron containing the rule, or the sending and receiving neurons are all pseudo-randomly selected. In the case where the selected mutation cannot be performed, the genetic algorithm will reattempt to mutate but with different parameters. Several failed attempts to mutate will result to a reattempt pseudo-randomly choosing the mutation type. Continuous failed reattempts with different mutation types will leave the chromosome unchanged. The number of reattempts is set by the user.

PSN P Validation

Each PSN P chromosome is checked whether it is valid or not. A PSN P chromosome is valid if there is a path from each input neuron to the output neuron. This check is performed, after the crossover or a mutation if the chromosome is selected for mutation. If the chromosome is invalid, the mutation and crossover is reverted, and will be performed again.

7 EXPERIMENTS

Input Design

The genetic algorithm will accept an initial PSN P $\Pi_i nit$ as an input used to create a population. For this study, there are three (3) categories of initial PSN P designed to test the genetic algorithm, namely: Perfect, Extra, Mutated. These were designed to contain extra or modified neurons, rules, and synapses on a PSN P that will approximate a bitwise AND and bitwise OR operation. Through this, we can check if the algorithm will be able to obtain a PSN P chromosome that can approximate the functions by adding, removing, or replacing rules and synapses.

Perfect category. This category of PSN P system input is determined by the authors to be one of the smallest PSN P systems that can approximate a bitwise OR and bitwise AND operations. By using this as input in the genetic algorithm, the initial population would contain one or two changes from the considered smallest PSN P for the specified operations.

Perfect AND			
	10%	20%	30%
Experiment 1	99	100	100
Experiment 2	99	100	100
Experiment 3	99	100	100

Table 2: Perfect AND Experiment for Tournament Selection and Fitness Proportionate

Extra AND			
	10%	20%	30%
Experiment 1	53	59	100
Experiment 2	53	56	59
Experiment 3	58	55	56

Table 3: Extra AND Experiment - Tournament Selection

Extra category. This category of PSN P system is handmade by the authors to contain a few extra neurons from the Perfect category of PSN P system input. This aids in checking if the PSN P is able to obtain the Perfect category PSN P system by disconnecting the extra neurons through removing or replacing rules and synapses of these extra neurons.

Mutated category. This category of PSN P system is generated by repeatedly running the Perfect category PSN P system through the Mutation function in 6. The final output after several mutation iterations is the PSN P to be used as the input for the genetic algorithm.

Experiment Setup

Experiment sources (e.g. codes, test files) are available from the corresponding author upon request. For each PSN P category and operation, six (6) types of experiments is performed. Three (3) of these is using Tournament selection and the other three (3) uses Fitness Proportionate type of parent selection. These three (3) for both types of selections vary in mutation rate: 10%, 20%, and 30% mutation rate. Each experiment is run three (3) times in this work to be able to grasp certain trends or outliers in the duration of the experiment. This amounts to a total one hundred and eighteen (118) tests, each of which is a single run of the genetic algorithm. In Section A of the Appendix we list the parameters we used in this work.

Extra AND			
	10%	20%	30%
Experiment 1	55	53	56
Experiment 2	54	57	60
Experiment 3	55	61	85

Table 4: Extra AND Experiment - Fitness Proportionate

Mutated AND			
	10%	20%	30%
Experiment 1	60	64	58
Experiment 2	61	65	58
Experiment 3	63	69	58

Table 5: Mutated AND Experiment - Tournament Selection

Mutated AND			
	10%	20%	30%
Experiment 1	61	65	58
Experiment 2	54	69	59
Experiment 3	59	65	58

Table 6: Mutated AND Experiment - Fitness Proportionate

Perfect OR			
	10%	20%	30%
Experiment 1	100	100	100
Experiment 2	100	100	100
Experiment 3	100	100	100

Table 7: Perfect OR Experiment for Tournament Selection and Fitness Proportionate

8 ANALYSIS AND DISCUSSION

For the Perfect OR and AND, after 100 generations, the PSN P keeps a hundred fitness and always have equal resources as the initial PSN P. This result is to be expected because chromosomes in the populations used for this category are close to Perfect PSN P system for computing bitwise Or and And operations as stated in Section 7.

As we can see in Table 8, the average fitness across all experiments of Extra OR are 100. This is due to the initial population having the Perfect OR in it's subgraph. And if we look at Figure 17, the GA framework is able to remove unnecessary neurons and arrive at the Perfect OR PSNP.

Extra OR			
	10%	20%	30%
Experiment 1	100	100	100
Experiment 2	100	100	100
Experiment 3	100	100	100

Table 8: Extra OR Experiment for Tournament Selection and Fitness Proportionate

Mutated OR			
	10%	20%	30%
Experiment 1	39	45	41
Experiment 2	37	40	44
Experiment 3	37	22	40

Table 9: Mutated OR Experiment - Tournament Selection

Mutated OR			
	10%	20%	30%
Experiment 1	26	49	32
Experiment 2	39	40	41
Experiment 3	26	52	41

Table 10: Mutated OR Experiment - Fitness Proportionate

For Extra AND, as we can see from Table 3 and Table 4, across all experiments, no experiment has an average fitness of the highest reach 100. But in one test in Experiment 1 with 30% mutation rate, the GA framework manages to at least get one 100% fitness Extra AND PSN P system. We can see the only 100% fitness PSN P system in all of the experiments for Extra AND in Figure 15, the unnecessary neurons was removed leading to the Perfect AND PSN P.

Mutated OR didn't succeed in yielding significant output. Across all experiments, the highest fitness the GA framework achieved is 45. As we can see here in Figure 18, the GA framework managed to put rules in the neurons that does not have on during the initial but it failed to remove the extra synapses.

Same with problem in mutated OR, the GA framework was able to insert new rules to the empty neurons in initial mutated AND as we can see in Figure 20 but failed to reduce the synapses.

As mentioned previously, we used the same initial population for every experiment in each PSN P category. This would hopefully give us a clue on what selection method would be better based on our experiments. But unfortunately, as we can see from the graphs in Figures 5,

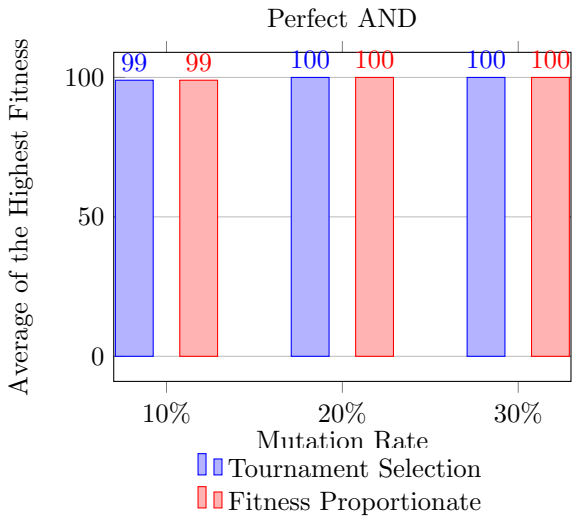

Figure 3: Bar graph about the Average of the Highest Fitness across all experiments in Perfect AND

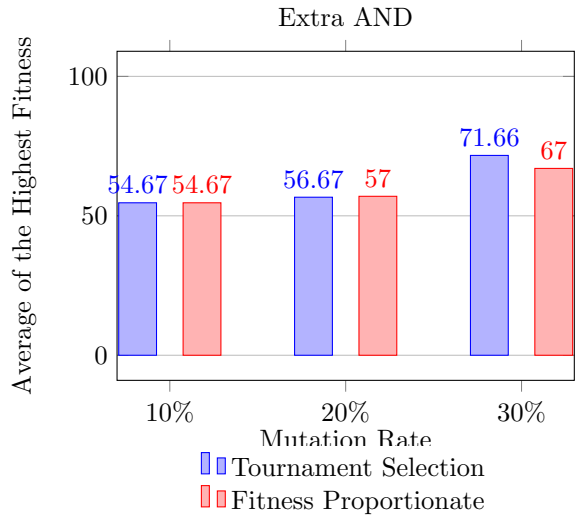

Figure 4: Bar graph about the Average of the Highest Fitness across all experiments in Extra AND

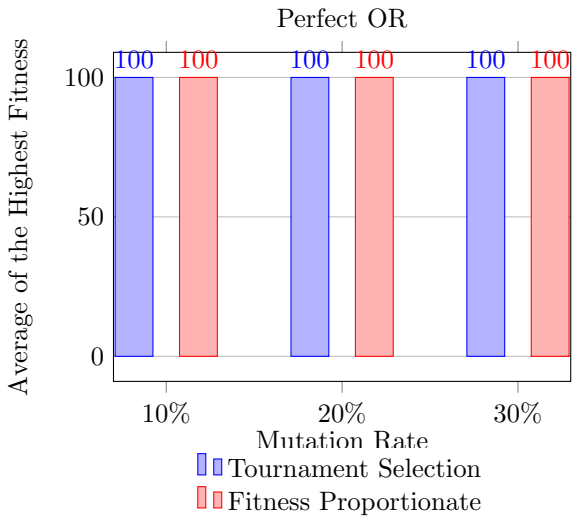

Figure 5: Bar graph about the Average of the Highest Fitness across all experiments in Perfect OR

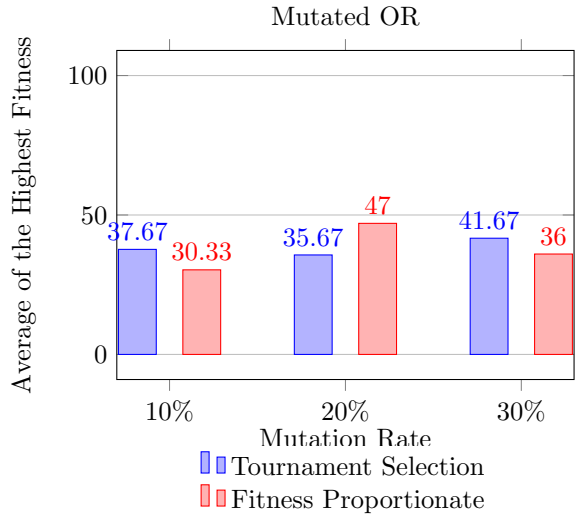

Figure 6: Bar graph about the Average of the Highest Fitness across all experiments in Mutated OR

3, 6, 8, 7, and 4, there is no visible trend. The results of the various experiments are too close to each other and there is little number of tests per category and operation to determine a certain trend with the selection type.

9 CONCLUSIONS AND FUTURE WORKS

We have used a genetic algorithm to obtain a PSN P system that would approximate a function from an initial PSN P system and spike trains input. The resulting PSN P systems may vary in fitness and may contain outliers. These aren't ideal PSN P systems but they show results which present that the framework created in this study may be a feasible solution to the problem in Section 1. While the experiments still have lots of room for improvement, it introduces a new approach for transforming PSN P systems for future work.

The mutation used in our GA framework is limited to the operations in Section 6. Given this, a new operation such as being able to change the polarization of neurons can be considered. This could free neurons from being stuck in the same polarity for longer runs. In line

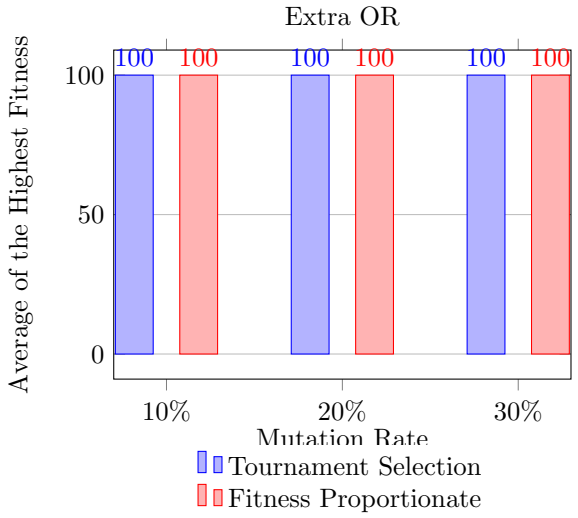

Figure 7: Bar graph about the Average of the Highest Fitness across all experiments in Extra OR

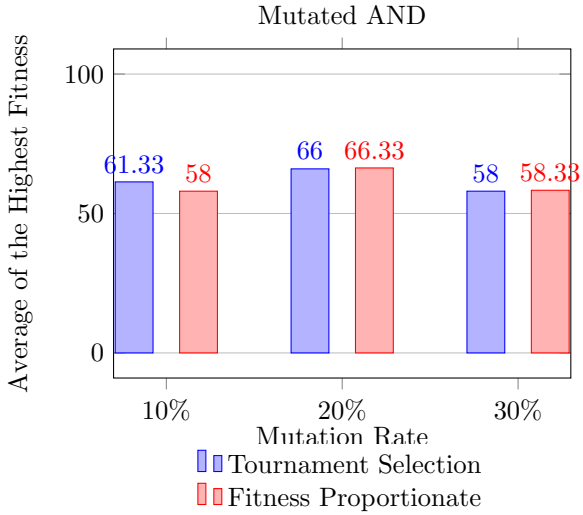

Figure 8: Bar graph about the Average of the Highest Fitness across all experiments in Mutated AND

with this, other semantics of polarity evaluation can be explored.

Additional factors like number of inactive and disconnected neurons from the work of [2] could be included in fitness calculation.

Other methods for parents selection can be considered that can help add variety to the population. Moreover,a different parent crossover mechanism could be used to improve off-springs.

REFERENCES

[1] Jym Paul Carandang, John Matthew B Villaflores, Francis George C Cabarle, Henry N Adorna, and MA Martınezdel-Amor. 2017. CuSNP: Spiking neural P systems simulators in CUDA. *Romanian Journal of Information Science and Technology* 20, 1 (2017), 57–70.

[2] Lovely Joy Casauay, Ivan Cedric H Macababayao, Francis George C Cabarle, Ren Tristan A de la Cruz, Henry N Adorna, Xiangxiang Zeng, and Miguel Ángel Martinez-del Amor. 2019. A Framework for Evolving Spiking Neural P Systems. In *ACMC2019: The International Conference on Membrane Computing (Asian Branch)*. (in press) International Journal of Unconventional Computing, Old City Publishing.

[3] TT Chow, GQ Zhang, Z Lin, and CL Song. 2002. Global optimization of absorption chiller system by genetic algorithm and neural network. *Energy and buildings* 34, 1 (2002), 103–109.

[4] Mihai Ionescu, Gheorghe Păun, and Takashi Yokomori. 2006. Spiking neural P systems. *Fundamenta informaticae* 71, 2, 3 (2006), 279–308.

[5] Melanie Mitchell. 1998. *An introduction to genetic algorithms*. MIT press.

[6] David J Montana and Lawrence Davis. 1989. Training Feedforward Neural Networks Using Genetic Algorithms.. In *IJCAI*, Vol. 89. 762–767.

[7] Gheorghe Paun. 2012. *Membrane computing: an introduction*. Springer Science & Business Media.

[8] Gheorghe Paun, Tingfang Wu, and Zhiqiang Zhang. 2016. Open Problems, Research Topics, Recent Results on Numerical and Spiking Neural P Systems (The" Curtea de Arge s 2015 Series"). *BWMC 2016: 14th Brainstorming Week on Membrane Computing: Sevilla, ETS de Ingeniería Informática, February 1-5 (2016), p 285-300* (2016), 285–300.

[9] Tingfang Wu, Andrei Păun, Zhiqiang Zhang, and Linqiang Pan. 2017. Spiking neural P systems with polarizations. *IEEE transactions on neural networks and learning systems* (2017).