An Evolutionary Approach to the Design of Spiking Neural P Circuits

Alberto Leporati[1 [0000−0002−8105−4371]] and Lorenzo Rovida[1 [0000−0001−5093−7932]]

University of Milano-Bicocca
Department of Informatics, Systems, and Communication
Viale Sarca 336, Edificio U14, 20126 Milano, Italy
{alberto.leporati,lorenzo.rovida}@unimib.it

Abstract. In this paper we define spiking neural P circuits (SN P circuits) as an acyclic variant of spiking neural P systems. We then study how well genetic algorithms (GA) are able to find an SN P circuit that computes a given Boolean function, possibly partially defined. The proposed technique can be used to find SN P circuits that solve binary classification problems. We performed several computer experiments, testing different mutation operators and several combinations of hyperparameter values. The preliminary results obtained show that the probability of success of GA strongly depends upon the structure (in particular, the algebraic degree and the number of input/output variables) of the Boolean function to be computed.

Keywords: Spiking neural P systems · Evolutionary algorithms · Boolean functions · Binary classification.

1 Introduction

Spiking neural P systems (SN P systems, for short) have been introduced in [13] as the first neural-like model of membrane systems [22,23], abstracted from the spiking mechanism of biological neurons. Inspired by spiking neurons, SN P systems and variants are considered as the third generation of neural network models, similar to spiking neural networks [18]. Structurally, an SN P system is a directed graph composed of several neurons, where the nodes are spiking neurons, and the directed arcs indicate the synaptic connections between neurons. Depending on predefined conditions, neurons emit spike objects (which represent the electrical impulses emitted by biological neurons); these spikes are sent to adjacent neurons via synaptic connections. So doing, a computational model can be defined, that takes appropriately encoded input values from the environments and computes appropriately encoded output values.

Over the years, several variants have been studied that are inspired by different biological mechanisms. These include astrocytes [29], anti-spikes [21], the presence of multiple channels [28], inhibition rules [25], dendrite mechanisms [24], polarities [36], structural plasticity [5], communication on request [35],

autapses [31], delay on synapses [16], plasmids [6], dynamic threshold [27], and coupling mechanism [26]. Most of these variants have been proven to work as Turing universal devices for generating or accepting sets of numbers, for computing functions, or for generating languages. Furthermore, SN P systems and variants have been used for some real-world application problems such as image fusion [15,4], image segmentation [7], edge detection [39,37,38], medical image processing [40], financial time series prediction [43], sentiment analysis [12,11,17], recommendation systems [1], numerical optimization [41,44], and classification problems [42].

In this paper we introduce an acyclic variant of SN P system, named *SN P circuits*. These circuits are composed of layers, each one made of *SN P gates*. Similarly to what happens with traditional Boolean circuits made of AND/OR/NOT gates, a n-input/m-output SN P circuit computes a n-input/m-output Boolean function $f : \{0,1\}^n \to \{0,1\}^m$. As we will see, given a Boolean function, there may be several SN P circuits that compute it. This depends also on the fact that the Boolean functions to be calculated will only be *partially defined*; that is, not all input/output $(\mathbf{x}, \mathbf{y})$ pairs will be specified. As it will become clear in the next section, a partially defined Boolean function can also be seen as the training/testing dataset for a classification problem. Hence, finding a SN P circuit that correctly computes the given input/output pairs of a Boolean function corresponds to finding a circuit that solves a given classification problem. Our goal is to automatically find one of these circuits, using evolutionary techniques. In particular, our aim is to answer the following research questions:

RQ1 How well (or poorly) do evolutionary algorithms work, in finding the SN P circuits that compute a given Boolean function/solve a given classification task?

RQ2 How does the quality of the solutions found vary depending on some characteristics of the Boolean function (for example, non-linearity)?

Fully answering the first research question requires testing a large number of evolutionary algorithms, varying the corresponding hyperparameters. This article reports the results of the first study we have conducted to answer these questions. For this reason, we will start by considering genetic algorithms (GA), by far the most famous evolutionary algorithms. We will then define appropriate crossover and mutation operators for the SN P circuits, and we will perform some computer experiments to test the behavior of GA when their parameters vary.

The rest of the paper is structured as follows. In Section 2 we give some mathematical preliminaries, to fix the notation and to establish a common background of notions. In Section 3 we give the details of the genetic algorithm used in the following experiments. In particular, we define the crossover and mutation operators, as well as the parameters that may influence the behavior of our algorithm. In Section 4 we give a detailed description on how the genetic algorithm and its variation operators (crossover and mutations) have been implemented. Furthermore, we describe how SN P gates and SN P circuits are represented in the computer's memory. Section 5 contains a thorough description of the experiments carried out, as well as an in-depth discussion of the results obtained.

Section 6 concludes the paper, and provides some ideas and directions for future research.

2 Mathematical Preliminaries

In what follows, we denote the Boolean values FALSE and TRUE as the integer numbers 0 and 1, respectively. A Boolean variable is a variable that may assume a Boolean value. A n-input/m-output Boolean function, for some fixed integers $n, m \geq 1$, is a function of the type $f : \{0,1\}^n \to \{0,1\}^m$.

A *Boolean circuit* is a directed acyclic graph whose internal nodes (named *gates*) compute Boolean functions. The nodes of in-degree 0 are called *input nodes* and are usually labeled with input variables. The nodes with out-degree 0 are called *output nodes*. The value of the circuit is computed in the natural way by giving values to the input variables, applying the gates to these values, and computing the output values. In what follows we will refer to the usual circuits — whose gates compute the "standard" Boolean functions AND, OR and NOT — as AND/OR/NOT circuits. It can be easily seen that, without loss of generality, we can arrange the gates of a circuit in *layers*, such that each gate takes its input values among the output values of the previous layer (from the input nodes, for the first layer). Evaluating a circuit thus amounts to computing the Boolean vector produced by each layer, starting from the first layer and proceeding up to the output layer. The *size* of a circuit is the number of gates it contains, while the *depth* of a circuit is its number of layers.

A *symmetric gate* is a gate that computes a Boolean function whose value only depends upon the number of 1s given as input. As an example, the n-input AND gate is symmetric: its output value is 1 if and only if the number of 1s given as input is n. Similarly, the n-input OR gate is symmetric: its output value is 1 if and only if the number of 1s in input is one of the values $1, 2, \ldots, n$. Also the (1-input) NOT gate is symmetric: its output value is 1 if and only if the number of 1s given as input is 0. A symmetric gate with n inputs can thus be defined by giving the set of numbers $G \subseteq \{0, 1, \ldots, n\}$ such that the output of the gate is 1 if and only if the number of 1s given as input is in G. Other two examples of symmetric gates are the XOR gate, that computes the Boolean XOR among its two input values, and its n-input extension, usually called the PARITY gate. The former is defined by the set XOR$= \{1\}$, while the latter is defined by the set PARITY$= \{1, 3, 5, \ldots, n\}$ for n odd, or PARITY$= \{1, 3, 5, \ldots, n-1\}$ for n even.

Inspired from SN P systems, let us now define SN P circuits, which are made of SN P gates.

Definition 1. *A* spiking neural P gate *(*SN P gate*, for short) is a spiking neuron that contains a subset of the following spiking rules:*

$$a^i \to a \qquad \text{for all } i \in \{0, 1, \ldots, \ell\}$$

for a fixed integer value $\ell \geq 1$.

As we can see, a SN P gate is a special case of spiking neuron, as defined in the original version of spiking neural P systems [13], where:

- there are no forgetting rules;
- firing rules have no delay;
- the regular expression that enables the spiking rule of the kind $a^i \to a$ has the form $E = a^i$. That is to say, the rule is enabled if and only if the neuron contains exactly i spikes;
- there may be the firing rule $a^0 \to a$, that emits a spike whenever the neuron contains 0 spikes. This rule is not allowed in the standard model of spiking neural P systems. It is included here because it can be easily proved that, without it, it is not possible to compute the NOT Boolean function. More in general, without such a rule it is not possible to compute non-monotone Boolean functions, which are those functions whose output value may pass from 0 to 1 when one of their arguments passes from 1 to 0.
  Note that the use of rules $a^0 \to a$ can be avoided by changing the encoding of the inputs and outputs for our gates. Currently, the Boolean values 1 and 0 are represented by the presence of a spike and the absence of spikes, respectively. So, in a sense, the rules $a^0 \to a$ perform an emptiness/appearance check. If we want to avoid this type of check we can encode, for example, the Boolean values 1 and 0 as the presence of two spikes and the presence of one spike, respectively. Such an encoding requires to change the rules in the SN P gates accordingly; for example, if an SN P gate has m input lines, then the minimum number of spikes that may arrive from its input lines is m, which happens if all the input Boolean values are 0. As a consequence, all firing rules $a^i \to a$ in the SN P gate will be defined for $i \geq m$.

A SN P gate is a symmetric gate, which is defined by the set $G \subseteq \{0, 1, \ldots, \ell\}$ such that the neuron contains a rule of the kind $a^i \to a$ if and only if $i \in G$. Clearly, the value of ℓ may differ among SN P gates.

Definition 2. *A* spiking neural P circuit *(*SN P circuit*, for short) is a circuit composed of SN P gates.*

As mentioned in Section 2, the SN P gates of a SN P circuit can be arranged in layers, such that each gate takes its input values only among the output values of the previous layer (from the input nodes, for the first layer). A layer is made up of SN P gates that are not connected to each other. That is, if two SN P gates belong to the same layer, then the output of one of the two gates is not used as input by the other gate. Evaluating a circuit thus amounts to computing the Boolean vector produced by each layer, starting from the first layer and proceeding up to the output layer. Precisely, suppose that an SN P circuit computes a function $f : \{0,1\}^n \to \{0,1\}^m$, and that it is made up of k layers. First, we prepare an *input vector* for the circuit, consisting of n elements. Each element is the encoding of the corresponding input Boolean value, and therefore contains zero or one spike to indicate the Boolean value 0 or 1, respectively. From this input vector the SN P gates of the first layer take their inputs, according to their own

input lines. The number of spikes arriving at an SN P gate through its input lines are summed, and activate a firing rule or a forgetting rule. Consequently the gate fires or cancels its spikes, thus producing one or zero spikes as output, corresponding to the Boolean value 1 or 0, respectively. All the Boolean values produced in this way by the gates of the first layer form a vector of Boolean values, which constitute the input values for the SN P gates of the second layer. All the SN P gate layers of the circuit are evaluated in this way, one after the other, until the k-th layer is evaluated, which produces the circuit's vector of output Boolean values.

Let us note that, since we are working with acyclic circuits, whose output values are computed by evaluating their layers sequentially, it is not necessary to consider delays greater than 0 in the spiking rules. For the same reason, it is not even necessary to consider forgetting rules, which delete a certain number of spikes present in the neuron without making it emit spikes towards adjacent neurons. If we do not want to make a SN P gate spike for a certain number i of input spikes, we can simply omit the spiking rule $a^i \to a$; the i input spikes will just remain in the neuron. Of course, this works because SN P circuits are assumed to be "cleaned" before evaluating them; that is, we assume that each gate of the SN P circuit contains no spike before being evaluated. This is done so that the output value of the gates depends only on their input values, and not on residual spikes left over from previous computations. Alternatively, to facilitate the reuse of circuits, it can be assumed that for each spiking rule $a^i \to a$ missing there is a corresponding forgetting rule $a^i \to \lambda$. In what follows, we will simply clean each SN P circuit before evaluating it.

In this paper, our aim is to apply genetic algorithms (GA) to find SN P circuits that compute a given (partially defined) Boolean function. Given a Boolean function $f : \{0,1\}^n \to \{0,1\}^m$, we say that it is *partially defined* if we only know some pairs $(\mathbf{x}, \mathbf{y})$ such that $\mathbf{x} \in \{0,1\}^n$, $\mathbf{y} \in \{0,1\}^m$, and $f(\mathbf{x}) = \mathbf{y}$. For the input values $\mathbf{x} \in \{0,1\}^n$ not comprised in this list of pairs, the function may assume any output value. Let us note that partially defined Boolean functions may describe the training/testing dataset of classification problems. In fact, let us consider a binary classification problem whose output (TRUE or FALSE) depends upon the values of n input Boolean features. These features are given as input to the classifier as n-tuples of Boolean values, and the classifier produces a Boolean output value, thus computing a Boolean function. The case in which the input and the output values are not Boolean can be easily treated by suitably encoding such values as binary strings.

Simple calculations will show the following facts. The number of Boolean functions $f : \{0,1\}^n \to \{0,1\}^m$ is $(2^m)^{2^n}$; even restricting to the case $m = 1$, this number grows in a doubly exponential way with respect to n, being 2^{2^n}. Hence exploring the entire search space of Boolean functions is impractical already for a small number of input values, such as $n = 8$. As many SN P circuits may compute the same Boolean function, the search space of SN P circuits is even larger, and finding a specific circuit means looking for the proverbial needle in

the haystack. This is the reason why we are using evolutionary algorithms as heuristics for guiding us in this search.

Given a set of pairs $(\mathbf{x}, \mathbf{y})$ that partially define a function $f : \{0,1\}^n \to \{0,1\}$, each pair identifies one row in the truth table of f. How many Boolean functions are consistent with this partial definition? Denoted by k the unspecified rows, for each of them the output value can be 0 or 1, and therefore there are 2^k possible combinations. Each combination corresponds to a Boolean function consistent with the given $(\mathbf{x}, \mathbf{y})$ pairs, hence the number of Boolean functions which are coherent with such a partial definition is 2^k. This argumentation can be generalized to functions of the type $f : \{0,1\}^n \to \{0,1\}^m$: for each of the unspecified k rows the output value is any m-tuple of $\{0,1\}^m$; therefore, there are $(2^m)^k = 2^{k \cdot m}$ possible functions consistent with the given $(\mathbf{x}, \mathbf{y})$ pairs.

Therefore, the fewer the pairs that partially define the Boolean function to be calculated, the more Boolean functions will be consistent with it, and the easier it will therefore be to find an SN P circuit that calculates it. Conversely, the more pairs there are that partially define the Boolean function to be calculated, the more difficult it will be to find an SN P circuit that calculates it.

A final observation is related with the fact that SN P gates are symmetric. As shown in [9], AND/OR/NOT circuits whose size is polynomial in the number n of inputs, and whose depth is constant — that is, does not depend upon n — cannot compute the n-input PARITY function. As this function can be computed by a single symmetric gate, circuits that are composed of symmetric gates are more powerful than traditional AND/OR/NOT circuits. Hence, in theory, finding SN P circuits that compute a given (possibly partially defined) Boolean function by means of evolutionary techniques should be easier than finding AND/OR/NOT circuits by the same techniques.

3 A Genetic Algorithm for Evolving Spiking Neural P Circuits

As stated above, our goal is to investigate which are the most suitable evolutionary algorithms [8] for identifying SN P circuits that compute a given, possibly partially defined, Boolean function. In this paper we begin this investigation starting from the most famous and used evolutionary algorithms in the literature: Genetic Algorithms (GA). In future work, other evolutionary algorithms will be tested, such as Genetic Programming [14], Grammatical Evolution [30], Differential Evolution [33], Evolution Strategies [2], and Memetic Algorithms [20].

Genetic algorithms [19] are meta-heuristic optimization algorithms for solving both constrained and unconstrained optimization problems based on a natural selection process that mimics biological evolution. In a genetic algorithm, a population of candidate solutions (called individuals) to an optimization problem is evolved toward better solutions. Each candidate solution has a set of properties which can be mutated and altered. The evolution is an iterative process, with the population in each iteration called a generation. The process usually starts from a population of randomly generated individuals. In each generation, the

fitness of every individual in the population is evaluated; the fitness is usually the value of the objective function in the optimization problem being solved. The individuals are then randomly selected from the current population, are recombined and possibly randomly mutated to form offspring, who makes up a new generation. The new generation of candidate solutions is then used in the next iteration of the algorithm. Usually, the algorithm terminates when either a maximum number of generations has been produced, or a satisfactory fitness level has been reached for the population.

Apart from finding a suitable representation for individuals, and an appropriate fitness function, a crucial point in genetic algorithms is defining selection, crossover and mutation operators. The precise choices made for these operators, as well as those made for the parameters on which the functioning of the algorithm depends (such as the probability of mutation, the size of the generations, etc.) will be described in the next section, where the implementation details will be presented. Here we only give a high-level description of the functioning of the algorithm, highlighting some possible variations related to the choices of the operators.

A *selection* operator is used to choose one or more individuals in the current population, for later breeding (e.g., using the crossover operator). Usually, the selection operator takes into account the fitnesses of the individuals of the current generation. Among all possible selection operators, in this preliminary study we decided to use the one most used in the literature, namely the *fitness proportionate selection*, also known as the *roulette-wheel selection*. In this selection method, the probability of choosing an individual for breeding of the next generation is proportional to its fitness: the better the fitness is, the higher chance for that individual to be chosen. Furthermore, we decided to make use of *elitism*: at each iteration of the GA, a percentage (10%, in this study) of the individuals with the highest fitness are copied unchanged to the next generation.

Also regarding the *crossover* operator, there are several possible choices. Let us first remember that the objective of this operator, also known as *recombination*, is to randomly combine the genetic information of two parents to generate new offspring[1]. Thus, each individual of the next generation will be generated by a pair of parents, chosen by the selection operator mentioned above. Each individual is an SN P circuit, and can be seen as a graph whose nodes are the SN P gates. Inspired by *one-point crossover*, which is one of the most used crossover operators in the literature, a sub-graph of the first parent should be swapped with a sub-graph of the second parent. Although this approach has already been attempted in the literature [10,32,34], in this study we decided to adopt a simpler approach: since the parent circuits are formed by layers of SN P gates, we cut each circuit before a randomly chosen layer; we then combine the first piece of the first circuit with the second piece of the second circuit, and the second piece of the first circuit with the first piece of the second circuit. So doing, the

[1] To be precise, this corresponds to *sexual* reproduction in biology. Solutions can also be generated by *cloning* an existing solution, which is analogous to *asexual* reproduction.

two parents produce two circuits; as an alternative, we can keep only the one of the two children that has the highest fitness value, discarding the other. Further details on the choices made for this study are given in the next section.

The newly generated solutions may be mutated before being added to the population. This is made by applying one (or more) *mutation* operators. Also in this case there are several possibilities; referring to SN P circuits, we can, for example:

1. randomly modify the set of rules in one or more SN P gates. As a particular case, when considering AND/OR/NOT circuits, some AND gates will become OR gates, and vice versa;
2. one or more gates can be added in random positions;
3. one or more randomly chosen gates can be deleted;
4. one or more input lines can be added or removed in randomly chosen gates.

In reality, in this paper we will also use two more drastic mutation operators, which have a much more important impact on the structure of SN P circuits:

1. removing a randomly chosen layer;
2. adding a new layer at a randomly chosen location.

Finally, the *fitness* value of an individual will be calculated as follows. Given a partially defined Boolean function $f : \{0,1\}^n \to \{0,1\}^m$, the SN P circuit will be evaluated on all the input vectors $\mathbf{x} \in \{0,1\}^n$ that have been provided in the specification; the fitness value will simply be the number of times in which the output of the circuit coincides with the given output value $\mathbf{y} \in \{0,1\}^m$. Calling c the circuit whose fitness we want to evaluate, the fitness function is thus defined as follows:

$$\text{fitness}(c, f) = 1 - \frac{1}{k}\left(\sum_{i=1}^{k} \big(c(\mathbf{x}_i) \neq \mathbf{y}_i\big)\right) \tag{1}$$

where $(\mathbf{x}_1, \mathbf{y}_1), \ldots, (\mathbf{x}_k, \mathbf{y}_k)$ are the pairs that (partially) define the Boolean function $f : \{0,1\}^n \to \{0,1\}^m$, and $c(\mathbf{x}_i)$ is the output produced by circuit c on input $\mathbf{x}_i$. The sum counts the number of times the output of circuit c differs from the expected output $\mathbf{y}_i$. Such a number is averaged over the number k of input values, and the result is subtracted from 1, so that the fitness value is always between 0 and 1, with 1 indicating that the circuit always gives the correct answer, and 0 indicating that the circuit gives the wrong answer every time.

4 Implementation of the Genetic Algorithm

We implemented the GA described above in the Julia programming language [3]. In what follows we give some details about our implementation.

A Boolean function is represented in our software as a list containing the input/output pairs $(\mathbf{x}, \mathbf{y})$ that constitute the (possibly partial) definition of the function. For simplicity in the implementation, the Boolean values are represented as the Julia constants `false` and `true`, instead of using the binary values 0 and 1.

A SN P circuit is implemented as a list of layers. The length of the list is the depth of the circuit. Each layer is a list of SN P gates. In this case, the length of the list is the size of the layer. The sum of the sizes of the layers is the size of the circuit (i.e. the number of gates present in the circuit).

A SN P gate is a triple (implemented as a list) of the kind:

[`rules` `inputs` `number_of_spikes`]

where:

- `rules` is a set (implemented as a list) of non-negative integer numbers. These numbers represent the cases in which the gate must emit a 1 as output; that is, the gate emits a 1 if and only if the number of spikes contained in the gate is equal to one of the numbers of the list;
- `inputs` is a list that indicates the connections (links) between the gate and the gates of the previous layer. To be precise, the numbers contained in this list are indexes over the Boolean vector that contains the result of the evaluation of the previous layer. That is, these values indicate at which positions of the vector the gate must take its input values. In case the gate is in the input layer, the indices refer to the circuit's vector of input values;
- `number_of_spikes` indicates the number of spikes currently contained in the gate. Although in theory a gate can be defined by specifying a number of spikes present at its creation, in this study each gate initially contains zero spikes.

As we said in the previous section, we use *fitness proportionate selection* to choose individuals who will generate offspring to be included in the next generation. This means that each individual has a probability of being selected equal to the value of its fitness divided by the sum of the fitness values of all individuals in the current population. As a consequence, the larger the fitness value for a given circuit, the larger the probability to be chosen as a parent for the next generation. Furthermore, we use *elitism*: each generation will contain the 10% of the best individuals — i.e. those having the highest fitness value — of the previous generation.

Concerning *crossover*, we propose a procedure that produces a new circuit (an individual) given two parent circuits. The idea is to exchange genetic information between two parents in order to create a new individual with new genetic characteristics. As stated in the previous section, inspired by *one-point crossover*, we cut each circuit before a randomly chosen layer; we then combine the first piece of the first circuit with the second piece of the second circuit, and the second piece of the first circuit with the first piece of the second circuit. So doing, the two parents produce two circuits; however, we can keep only the one of the two children that has the highest fitness value, discarding the other. Called C_1 and C_2 the two parent circuits, the procedure generates a new individual, called C_{new}, as shown in Figure 1.

This procedure can be applied to any pair of SN P circuits with at least one hidden layer. In this paper we assume that the crossover is performed between

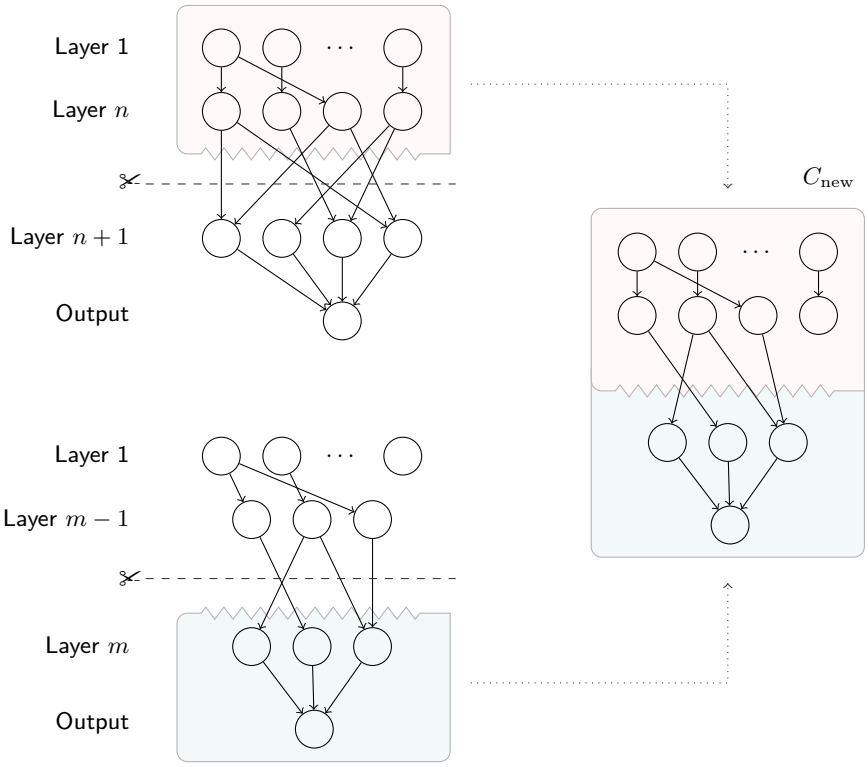


Fig. 1. Cutting two parent circuits at two crossover points: n for the parent circuit C_1 and $m-1$ for the parent circuit C_2

two circuits having the same number of input and output gates. The crossover procedure performs the following steps:

1. A random value r is uniformly sampled from the uniform distribution $\mathcal{U}(a, b)$, with $a < b < 1$ (in our implementation $a = 0.4$ and $b = 0.6$). This indicates the proportion of information preserved from C_1.
2. The first circuit, C_1, is cut at the first crossover point $n = \lfloor r \cdot |C_1| \rfloor$.
3. The second circuit C_2, on the other hand, is cut at $m = \lfloor (1 - r) \cdot |C_2| \rfloor$.

To conclude the crossover operation, the two selected chromosomes (sub-circuits) are combined in a single individual (circuit). The pseudocode of the crossover procedure is given in Algorithm 1.

Recalling that each neuron contains a set of input connections, two different cases can occur:

1. The number of neurons of the layer n of C_1 is equal to the number of neurons of layer $m-1$ of C_2. In this case, no further operation is required, since all the connections are preserved (Figure 2).

Algorithm 1 Crossover operation between two circuits C_1 and C_2

1: **procedure** CROSSOVER(C_1, C_2)
2: $r \sim \mathcal{U}(a, b)$ ▷ Uniformly sampling r
3: $n \leftarrow \lfloor r \cdot |C_1| \rfloor$ ▷ n is the first crossover point for C_1
4: $m \leftarrow \lfloor (1 - r) \cdot |C_2| \rfloor$ ▷ m is the second crossover point for C_2
5: $C_{\text{new}} \leftarrow \text{append}\,(C_1[1 : n], C_2[m : end])$ ▷ Appending the two sub-circuits
6: **return** C_{new}
7: **end procedure**

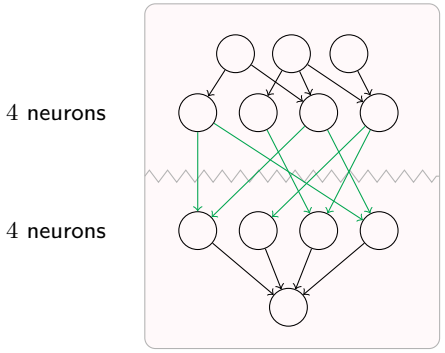


Fig. 2. Situation after a crossover where all edges are preserved

2. The number of neurons in the two layers is not the same. This implies that there might be an invalid input connection to some neurons in the layer m of the new circuit C_{new} (see Figure 3).

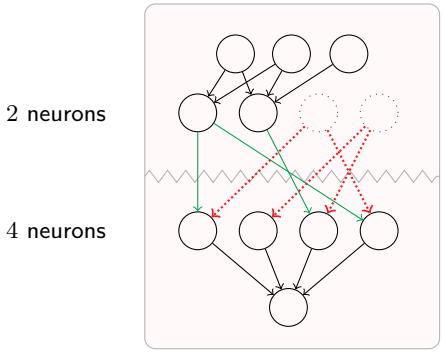


Fig. 3. Situation after a crossover with invalid (red) edges

For instance, if layer $m - 1$ of C_2 has four neurons, then the layer m can have input lines coming from four neurons. After performing the crossover,

these connections will refer to the layer n of C_1. If the number of neurons in that layer is less than four, then there might be invalid connections.

We propose to remove the invalid edges using a CLEAN procedure, described in Algorithm 2. Ideally, this iterative procedure controls that all neurons have connections to existing neurons. If a neuron has an invalid connection (i.e., an edge to a non-existing neuron), this is removed. If, after executing the procedure, the neuron has no edges left, it is removed. Eventually, if a layer remains without neurons, it will be removed, too.

Algorithm 2 Cleaning a circuit C

```
1: procedure CLEAN(C)
2:     for i ← 1 to |C| do                                  ▷ i indexes the layers
3:         for j ← 1 to |C_i| do              ▷ j indexes the neurons of the i-th layer
4:             in ← input lines of the j-th neuron of the i-th layer
5:             in ← remove all values greater than |C_{i−1}|
6:             if inputs is empty then                ▷ The neuron has no input lines
7:                 delete the j-th neuron of the i-th layer
8:             end if
9:         end for
10:        if |C_i| = 0 then                          ▷ The layer has no neurons left
11:            delete the i-th layer
12:        end if
13:    end for
14: end procedure
```

After the crossover operation, the newly produced individuals undergo a *mutation* phase. Inspired from genetic variation in biological evolution, the aim of mutations is to introduce diversity into the population, allowing new individuals to explore new spaces, and escape local optima. Mutations are implemented as random changes applied to the circuits produced by the crossover operation. In this paper, we apply the following set of mutations:

– Adding a new rule to the set of rules in random neurons. For example:

$$\begin{cases} a^0 \to a \\ a^4 \to a \end{cases} \longrightarrow \begin{cases} a^0 \to a \\ a^1 \to a \\ a^4 \to a \end{cases}$$

– Removing an existing rule from a set of rules of a randomly chosen neuron:

$$\begin{cases} a^0 \to a \\ a^2 \to a \\ a^4 \to a \end{cases} \longrightarrow \begin{cases} a^0 \to a \\ a^4 \to a \end{cases}$$

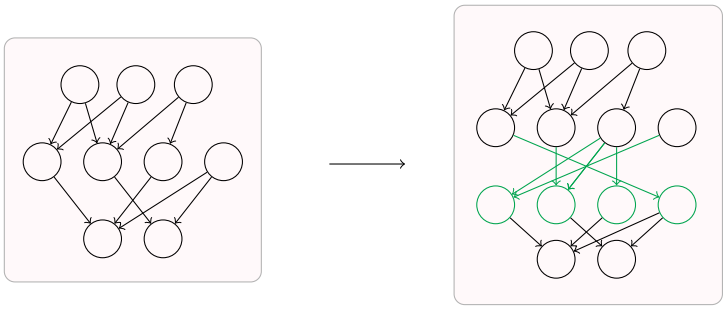
– Adding a new gate in a randomly chosen layer;

– Removing an existing gate in a randomly chosen layer;
– Adding a new input line to a randomly chosen gate;
– Removing existing input lines to a randomly chosen gate.

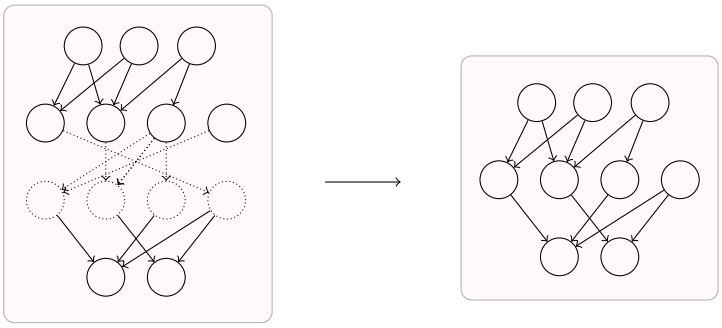
Notice that, after the application of some mutations (e.g., removing an existing gate), the CLEAN procedure must be executed in order to fix invalid edges, gates or layers.

This set of mutations can easily be extended with new operations. In particular, we consider two drastic mutation operations that, by adding and removing entire layers, have a much larger impact to the structure of the SN P circuits:

– Adding a new random layer of gates:



– Removing an existing layer of gates:



Each mutation is applied according to a corresponding value of probability. It is well known that applying mutations with high frequency (i.e., with large values of probabilities) may lead to random search behavior, while applying a small number of mutations may cause premature convergence to sub-optimal solutions. Fine-tuning the mutation rate and the type of mutation operators is often an important part of optimizing a genetic algorithm for a specific problem. For this reason, during the experiments we have adopted a grid-search approach to find optimal probability values for each mutation.

Putting all these elements together, we obtain a genetic algorithm to evolve a population of spiking neural P circuits. Recall that the objective is to, given a (possibly partially defined) Boolean function $f : \{0,1\}^n \to \{0,1\}^m$, find a SN P circuit that is able to compute f for all the given input/output pairs. The algorithm, whose pseudocode is shown in Algorithm 3, starts from a randomly generated population P, of whose elements the fitness is calculated. Until the established criterion to terminate the execution occurs, a new population of SN P circuits is generated. First, elitism is applied, copying unaltered a percentage p (set to 10%, in our experiments) of the best individuals to the new generation. Then, to fill the rest of the new generation, pairs of parent circuits are selected form the current population, according to the fitness proportionate selection method; the above described crossover operator is applied to these parent circuits, and only the fittest among the two children is kept. With small probabilities, the mutations described above are applied to such a child circuit, and the result is added to the new generation. For each generation, the maximum value of the fitness in the population is computed.

In this preliminary study, the stop criterion we have adopted is just a fixed number of generations. Concerning the other parameters that rule the behavior of the GA, such as population size, the number of layers and gates in randomly generated circuits, mutations' probabilities, etc., we have made some experiments — using a grid-search approach — starting from a set of empirically found parameters, depending on the experiment.

Algorithm 3 Genetic Algorithm for finding a SN P circuit that computes a given Boolean function f

procedure EVOLVE(f)
  ▷ f is (possibly partially) defined by a list of pairs $(\mathbf{x}, \mathbf{y}) \in \{0,1\}^n \times \{0,1\}^m$
  $P \leftarrow$ initial population of randomly generated SN P circuits
  FIT $\leftarrow$ fitness values of each circuit in P
  save the maximum and average fitness values in a list
  while stopping criterion is not verified **do**
    ▷ P' and FIT$'$ constitute the new generation
    $P' \leftarrow$ best p percent circuits from P ▷ p typically is in $[0, 10]$
    FIT$' \leftarrow$ fitness values of circuits in P'
    for $i \leftarrow 1$ to $(1 - p/100) \cdot |P|$ **do**
      parent1, parent2 $\leftarrow$ FITNESSPROPORTIONATESELECTION(P)
      child $\leftarrow$ CROSSOVER(parent1, parent2) ▷ only the fittest among the two children is kept
      add MUTATION(child) to P'
      add the fitness of child to FIT$'$
    end for
    save the maximum and average fitness values
    $P \leftarrow P'$ ▷ substitute P with P'
  end while
end procedure

5 Experiments

In order to evaluate the performance of the GA described above in finding SN P circuits that compute (possibly partially defined) Boolean functions, we performed some computer experiments, which we are now going to describe. Since these are the first experiments we have performed, we will only consider 1-output Boolean functions of the type $f : \{0,1\}^n \to \{0,1\}$. Some of them will be completely specified, whereas others will only be partially specified.

Each experiment is based on the following method. First, two sets of parameters are initially defined: `EvolutionParameters` and `MutationProbabilities`. The former set defines some parameters that control the evolution: the number of simulations, the number of iterations for each simulation, the number of individuals in a population, the minimum and maximum number of layers in randomly generated circuits, and the stopping criterion. This can be the maximum number of iterations, the maximum number of fitness evaluations, or a threshold on the maximum fitness value obtained by the population. On the other hand, the `MutationProbabilities` set of parameters defines the probabilities for applying the mutation operators presented in Section 4.

For each experiment, an initial phase aims to empirically observe some possible sets of "good" evolution parameters. A good set of parameters is defined as one obtaining a large average of maximum fitness values, among all the simulations. We focus on average fitness, rather than on the best one, because we are interested in producing well-behaving circuits from good genetic material, and not by chance. A grid-search is performed over the observed sets of good parameters, and the best combination thus found is used to perform the same procedure over the set of mutation probabilities. Informally, we follow this approach:

1. Experimentally observe possibly good sets of evolution parameters.
2. Perform a grid-search in order to find the best among these sets.
3. Fixed the evolution parameters set, experimentally observe possibly good sets of mutation probabilities.
4. Perform a grid-search in order to find the best set of probabilities.

This approach does not ensure to find the best possible combination of parameters, but it defines a heuristic that guides the process of finding good parameters. The first steps are performed using a set of mutation parameters, reported in Table 1, that have been experimentally found to be a good starting point.

Table 1. Default values of mutation probabilities

	Add layer	Remove layer	Add neuron	Remove neuron	Add rule	Remove rule	Random input lines
Probability	0.008	0.080	0.030	0.200	0.005	0.080	0.010

5.1 Fitting the PARITY (XOR) function

The first experiments aim to find SN P circuits that compute a simple Boolean function: PARITY, that is, the n-input/1-output XOR function. Such a function is considered simple because it is symmetric, just like SN P gates. Hence, in theory, one SN P gate suffices to compute it. For this reason, we expect the GA to show good performance on this function. In what follows we describe the experiments we have performed regarding the PARITY functions of $n = 5$ and $n = 8$ variables. In both cases the Boolean function was completely defined, that is, all its possible input/output pairs were given.

$\boldsymbol{n = 5}$ For this experiment, a grid-search over the following combinations of parameters has been performed:

- Population size: $|P| \in \{20, 30, 40\}$.
- Minimum number of intermediate layers in randomly generated circuits: $l_{min} \in \{1, 2\}$.
- Maximum number of intermediate layers in randomly generated circuits: $l_{max} \in \{l_{min}, l_{min} + 1\}$.

The stopping criterion for this experiment has been set to a maximum of 200 iterations. Each set of parameters has been evaluated for 15 simulations; in the end only the best circuit, in terms of fitness, is considered. So doing, each set of parameters generates 15 possible circuits (the best for each simulation); among these, the average fitness value is computed. The set of parameters achieving the highest average fitness value is chosen as the best. In case different sets of parameters show the same highest average fitness value, the one having the smaller population size (or, in case of equal values, the smaller circuits size) is chosen, following Occam's razor principle. The results are presented in Table 2.

Table 2. Some evolutions of SN P circuits with different parameters sets to compute the XOR function with $n = 5$ input values. Each set has been evaluated for 15 simulations

Population	Min layers	Max layer	Mean Fitness	Successful simulations
40	3	4	0.93125	2/15
40	1	2	0.9125	1/15
30	3	3	0.9125	2/15
30	3	4	0.9125	2/15
40	2	3	0.910417	1/15
20	3	4	0.908333	5/15
30	2	3	0.908333	4/15
		⋮		
20	1	1	0.879167	1/15

As it can be seen, the proposed GA is able to quickly evolve a population of circuits towards high fitness values. In particular, for all parameter configurations the GA is able to achieve the fitness target value 1 in at least one simulation. That is, for every parameter configuration at least one simulation obtains a SN P circuit that computes the prescribed Boolean function *perfectly*, on all input/output pairs. As stated above, we want to conduct a robust analysis for a GA that does not rely on luck to find a good SN P circuit; instead, we want to ensure that a good SN P circuit emerges from a population containing (on average) good genetic material. For this reason, we select as the "best" configuration the one achieving the largest average fitness value. For this introductory experiment, no grid-search has been applied over the mutation probabilities, since the different evolution procedures have already found circuits with the maximum fitness value. We also note that the function to be approximated has a relatively small input space.

$\boldsymbol{n = 8}$ The GA is now tested over a larger input space: the PARITY function of $n = 8$ input variables, thus having $2^8 = 256$ possible input values. The set of evolutionary parameters on which grid-search was performed is the following:

- Population size: $|P| \in \{60, 80, 100\}$.
- Minimum number of intermediate layers in randomly generated circuits: $l_{min} \in \{1, 2, 3\}$.
- Maximum number of intermediate layers in randomly generated circuits: $l_{max} \in \{l_{min}, l_{min} + 1\}$.
- Stopping criterion: stop after 500 iterations.
- Number of simulations for each set of parameters: 15.

The space is much larger with respect to the previous experiment, in which the circuits have to fit truth tables consisting of only $2^5 = 32$ input/output pairs. The results of this experiment are reported in Table 3.

Table 3. Some evolutions of SN P circuits with different parameters sets to compute the XOR function with $n = 8$ input values.

Population	Min layers	Max layer	Mean Fitness	Successful simulations
100	2	2	0.889844	0/15
100	3	3	0.835417	0/15
60	3	4	0.835156	1/15
80	3	3	0.827604	1/15
60	3	3	0.819792	0/15
60	1	2	0.811979	0/15
100	1	1	0.810156	0/15
		⋮		
60	1	1	0.765104	0/15

Differently from the previous experiment, only two out of 18 sets of parameters were able to find a SN P circuit that computes the prescribed Boolean function perfectly, that is, with a fitness value equal to 1. Nevertheless, we use the set of parameters achieving the highest average fitness value to execute the second phase, in which a second grid-search procedure is performed over mutation probabilities. The aim is to increase the average fitness value, and to obtain more successful simulations.

In particular, starting from the values presented in Table 1, a grid-search is executed over the following parameters:

- Probability of adding a new layer at each iteration $\in \{0.005, 0.010, 0.015\}$.
- Probability of adding a new neuron in a random layer $\in \{0.030, 0.050\}$.
- Probability of removing a neuron from a random layer $\in \{0.05, 0.15, 0.25\}$.

The obtained results, reported in Table 4, confirm the effectiveness of applying the grid-search over the mutation probabilities.

Table 4. Some evolutions of SN P circuits over different sets of mutation probabilities to compute the XOR function with $n = 8$ input values. Each set has been evaluated for 15 simulations

Add layer probability	Add neuron probability	Remove neuron probability	Average Fitness	Successful simulations
0.01	0.05	0.05	0.9125	2/15
0.01	0.05	0.25	0.8930	0/15
0.01	0.03	0.15	0.8883	0/15
0.01	0.03	0.25	0.8628	0/15
0.01	0.05	0.15	0.8529	1/15
0.02	0.05	0.05	0.8487	0/15
0.005	0.03	0.05	0.8482	0/15
		⋮		
0.02	0.03	0.05	0.8016	0/15

The best average fitness value has been increased up to 0.9125, and it is possible to find sets of probabilities that yield a higher number of successful simulations (that is, simulations in which at least one SN P circuit is found that calculates the Boolean function perfectly): up to 2 over 15, with respect to the maximum 1 over 15 obtained before applying the grid-search over mutation probabilities.

We complete the analysis by showing in Figure 4 the growth of fitness value for each individual in the population, using the best set of parameters.

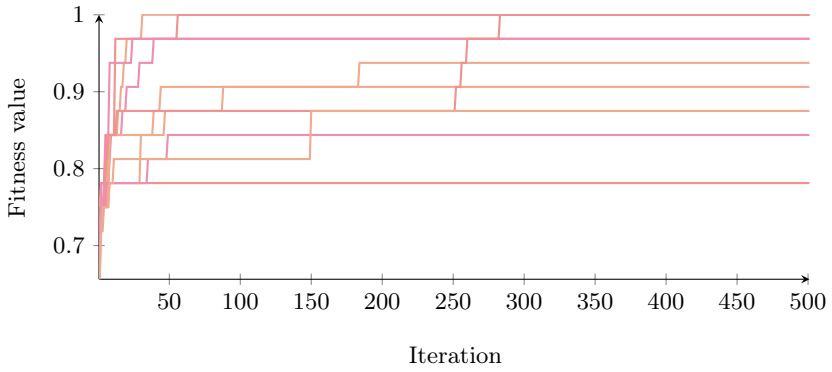


Fig. 4. Fitness value of the 15 best circuits of the best simulation. Each i-th line represents the fitness value of the i-th best circuit at a certain iteration

5.2 Fitting ANF functions

To carry out tests with different types of Boolean functions, in particular with different degrees of non-linearity, we also considered n-input/1-output functions $f : \{0,1\}^n \to \{0,1\}$ written in the algebraic normal form (ANF):

$$P_f(x_1, \ldots, x_n) = \bigoplus_{I \in \mathcal{P}(n)} a_I \left(\prod_{i \in I} x_i \right)$$

where $\mathcal{P}(n)$ is the power set of $\{1, \ldots, n\}$, and a_I is the coefficient of the monomial defined by the subset $I \in \mathcal{P}(n)$. As it can be seen, the algebraic normal form represents a function $f : \{0,1\}^n \to \{0,1\}$ as a multivariate polynomial. The *algebraic degree* of f is defined as the cardinality of the largest subset I such that $a_I \neq 0$, that is, as the largest degree over non-null monomials. In particular, *affine* functions are defined as Boolean functions with degree at most 1. Notice that the ANF is a unique representation of a Boolean function, and in particular, one can retrieve the truth table back from the ANF coefficients through the *Möbius transform*:

$$f(x_1, \ldots, x_n) = \bigoplus_{I \in \mathcal{P}(n) : I \subseteq supp(f)} a_I$$

where $supp(f)$ is the *support* of f, that is, the set of input vectors $\mathbf{x} \in \{0,1\}^n$ such that $f(\mathbf{x}) = 1$. We expect lower fitness values for these experiments, as ANF functions are not symmetric and may have a large degree of non-linearity.

Differently from the PARITY function with n inputs, which is unique, there are 2^n ANF functions with n input values, since each monomial corresponds to one possible subset of the n variables. We thus consider two different randomly generated ANF functions of degree $n = 5$ to evaluate the GA: f_1 and f_2, defined

as follows:

$$\begin{aligned} f_1(x_1,\ldots,x_5) = 1 &\oplus x_2 \oplus (x_1 \wedge x_2) \oplus (x_1 \wedge x_4) \oplus (x_1 \wedge x_5) \oplus (x_2 \wedge x_3) \oplus \\ &(x_3 \wedge x_5) \oplus (x_1 \wedge x_2 \wedge x_3) \oplus (x_1 \wedge x_2 \wedge x_5) \oplus (x_1 \wedge x_3 \wedge x_4) \oplus \\ &(x_1 \wedge x_3 \wedge x_5) \oplus (x_1 \wedge x_4 \wedge x_5) \oplus (x_3 \wedge x_4 \wedge x_5) \oplus \\ &(x_1 \wedge x_3 \wedge x_4 \wedge x_5) \oplus (x_2 \wedge x_3 \wedge x_4 \wedge x_5) \oplus \\ &(x_1 \wedge x_2 \wedge x_3 \wedge x_4 \wedge x_5) \end{aligned}$$

$$\begin{aligned} f_2(x_1,\ldots,x_5) = x_1 &\oplus (x_1 \wedge x_4) \oplus (x_1 \wedge x_5) \oplus (x_2 \wedge x_4) \oplus (x_3 \wedge x_5) \oplus \\ &(x_4 \wedge x_5) \oplus (x_1 \wedge x_2 \wedge x_3) \oplus (x_1 \wedge x_2 \wedge x_4) \oplus (x_1 \wedge x_2 \wedge x_5) \oplus \\ &(x_2 \wedge x_3 \wedge x_4) \oplus (x_2 \wedge x_3 \wedge x_5) \oplus (x_1 \wedge x_2 \wedge x_3 \wedge x_4) \oplus \\ &(x_1 \wedge x_3 \wedge x_4 \wedge x_5) \oplus (x_2 \wedge x_3 \wedge x_4 \wedge x_5) \oplus \\ &(x_1 \wedge x_2 \wedge x_3 \wedge x_4 \wedge x_5) \end{aligned}$$

In particular, we will use both a partial input set, consisting of 80% of all the possible 2^5 combinations, and a complete input set. As in previous experiments, we start from a grid-search over some evolution parameters, in particular:

- Population size: $|P| \in \{75, 100, 125\}$.
- Minimum number of intermediate layers in randomly generated circuits: $l_{min} \in \{1, 2, 3\}$.
- Maximum number of intermediate layers in randomly generated circuits: $l_{max} \in \{l_{min}, l_{min} + 1\}$.

We selected larger values for the population size, with respect to the experiment on the $n = 5$ Parity function, since we empirically observed that the ANF functions require a larger population to achieve good fitness values. The stopping criterion for this experiment has been set to a maximum of 500 iterations. In order to achieve a good level of robustness, each set of parameters has been evaluated for 20 simulations The best three results obtained for each of the two functions f_1, f_2 are presented in Table 5.

The first observation that can be made is on the size of the input set. When using a partial input set (22 examples out of 32), the mean fitness of the best parameters set reaches, in the case of f_1, a value of 0.91818, with 3 successful simulations out of 20. On the other hand, when considering the entire input set (32 examples out of 32), the mean fitness has a drop. This confirms that, as stated in Section 2, the fewer the pairs that partially define the Boolean function to be calculated, the more Boolean functions will be consistent with it, and the easier it will therefore be to find an SN P circuit that calculates it.

Considering the two experiments on the complete input set of f_1 and f_2, the difference from the Parity function with $n = 5$ input values (see Table 2), is evident. The GA showed much larger values of mean fitness and successful simulations for the Parity function. This result provides evidence in favor of the question posed in RQ2: the intrinsic characteristics of a Boolean function makes it harder (or easier) for the GA to find a SN P circuit that fits it.

Table 5. Best three evolutions of SN P circuits fitting the f_1 and the f_2 functions. The latter were evaluated both on a partial and a complete input set.

	Input set size	Population	Min layers	Max layer	Mean Fitness	Successful simulations
f_1	Partial (22/32)	100	2	2	0.91818	3/20
		125	2	2	0.91591	1/20
		100	2	3	0.91136	1/20
	Complete (32/32)	125	1	1	0.89531	1/20
		125	3	3	0.89375	0/20
		100	1	1	0.88906	0/20
f_2	Partial (22/32)	100	3	4	0.92500	1/20
		100	2	3	0.925	1/20
		125	1	1	0.92273	1/20
	Complete (32/32)	125	2	3	0.86406	0/20
		125	3	3	0.86406	0/20
		125	1	1	0.86094	0/20

The obtained mean fitness values in this first phase are very close to each other. For this reason, we continued the experiment by performing a grid-search over a sparse amount of probabilities, thus enlarging the space in which the algorithm can look for possible solutions:

- Probability of adding a new layer at each iteration $\in \{0.005, 0.010, 0.015\}$.
- Probability of adding a new neuron in a random layer $\in \{0.030, 0.050\}$.
- Probability of removing a neuron from a random layer $\in \{0.05, 0.15, 0.25\}$.

The results of the grid-search over these probabilities are presented in Table 6.

Table 6. Best three evolutions of SN P circuits fitting the f_1 and the f_2 functions, using the complete input set. A grid-search is applied over some mutation probabilities

	Add layer probability	Add neuron probability	Remove neuron probability	Average Fitness	Successful simulations
f_1	0.001	0.01	0.25	0.90469	0/30
	0.003	0.01	0.25	0.90156	0/30
	0.003	0.03	0.15	0.89687	1/30
f_2	0.003	0.03	0.05	0.87187	0/20
	0.005	0.01	0.05	0.86875	0/20
	0.003	0.03	0.25	0.8625	0/20

By applying a grid-search over the probabilities, the GA was able to find a SN P circuit that achieves the maximum fitness on f_1. In general, all the mean fitness values have increased, with respect to the results illustrated in Table 5. On the other hand, this approach was not able to find a SN P circuit that perfectly fits the truth table of the f_2 function.

6 Conclusions and Directions for Future Work

In this paper we have defined a new acyclic model of spiking neural P systems, named SN P circuits, which are able to compute n-input/m-output Boolean functions. We have then studied how well genetic algorithms (GA) are able to find an SN P circuit that computes a given Boolean function, possibly partially defined. We performed several computer experiments, testing different mutation operators and several combinations of hyperparameter values, using several Boolean functions.

This is the first step of a broader study, whose aim is to establish whether evolutionary algorithms can be effectively used to design SN P circuits that compute a given (possibly partially defined) Boolean function. As many classification problems can be described through appropriate Boolean functions, the proposed technique can also be used to find SN P circuits that solve such classification problems.

This paper opens many possibilities for future work. Even restricting our attention to the proposed genetic algorithm, there are at least three possible improvements:

1. Try to use more advanced (and complicated) crossover operations, that hopefully preserve better what the parent SN P circuits have learned about the structure of the Boolean function to be computed.
2. Perform experiments using other stopping criteria, to make it more suitable for comparison with other evolutionary techniques. For instance, instead of considering the number of generations — independent of the population size — it is more fair to consider the number of fitness evaluations. Thus, for example, it would be interesting to see which evolutionary algorithm produces the fittest individuals for a given budget of fitness evaluations. Furthermore, it makes no sense to continue producing new generations when the fitness of the best individual produced up to now has reached the optimal value, or exceeds a pre-established threshold.
3. Perform an ablation study, to show which is the effect of each operator (selection, elitism, crossover, mutation) on the results obtained by the GA. To make such a study, further computer experiments are necessary, each time excluding one of the operators, or using only a subset of the operators, then comparing the results obtained with those of the complete GA.

Last, but not the least, recall that our goal is to find which evolutionary technique is more suitable for designing SN P circuits. Thus, in future work, other

evolutionary algorithms will be tested, such as Genetic Programming, Grammatical Evoution, Differential Evolution, Evolution Strategies, and Memetic Algorithms.

References

1. Bai, X., Huang, Y., Peng, H., Wang, J., Yang, Q., Orellana-Martín, D., de Arellano, A.R., Pérez-Jiménez, M.J.: Sequence recommendation using multi-level self-attention network with gated spiking neural P systems. Information Sciences **656**, 119916 (2024). https://doi.org/https://doi.org/10.1016/j.ins.2023.119916
2. Beyer, H.G.: The Theory of Evolution Strategies. Springer Berlin, Heidelberg (2010). https://doi.org/10.1007/978-3-662-04378-3
3. Bezanson, J., Edelman, A., Karpinski, S., Shah, V.B.: Julia: A fresh approach to numerical computing. SIAM Review **59**(1), 65–98 (2017). https://doi.org/10.1137/141000671
4. Bo, L., Peng, H., Luo, X., Wang, J., Xiaoxiao, S., Pérez-Jiménez, M., Riscos-Núñez, A.: Medical Image Fusion Method Based on Coupled Neural P Systems in Nonsubsampled Shearlet Transform Domain. International Journal of Neural Systems **31**, 2050050 (06 2020). https://doi.org/10.1142/S0129065720500501
5. Cabarle, F.G.C., Adorna, H.N., Pérez-Jiménez, M.J., Song, T.: Spiking neural P systems with structural plasticity. Neural Computing and Applications **26**(8), 1905–1917 (2015). https://doi.org/10.1007/s00521-015-1857-4
6. Cabarle, F.G.C., Zeng, X., Murphy, N., Song, T., Rodríguez-Patón, A., Liu, X.: Neural-like P systems with plasmids. Information and Computation **281**, 104766 (2021). https://doi.org/https://doi.org/10.1016/j.ic.2021.104766
7. Cai, Y., Mi, S., Yan, J., Peng, H., Luo, X., Yang, Q., Wang, J.: An unsupervised segmentation method based on dynamic threshold neural P systems for color images. Information Sciences **587**, 473–484 (2022). https://doi.org/https://doi.org/10.1016/j.ins.2021.12.058
8. Eiben, A.E., Smith, J.E.: Introduction to Evolutionary Computing. Springer Berlin, Heidelberg (2015). https://doi.org/10.1007/978-3-662-44874-8
9. Furst, M., Saxe, J.B., Sipser, M.: Parity, circuits, and the polynomial-time hierarchy. Mathematical Systems Theory **95**, 13–27 (1984). https://doi.org/10.1007/BF01744431
10. Globus, A., Lawton, J., Wipke, T.: JavaGenes: Evolving Graphs with Crossover. Tech. rep., NASA Advanced Supercomputing (NAS) Division (2000), https://www.nas.nasa.gov/assets/nas/pdf/techreports/2000/nas-00-018.pdf
11. Huang, Y., Liu, Q., Peng, H., Wang, J., Yang, Q., Orellana-Martín, D.: Sentiment classification using bidirectional LSTM-SNP model and attention mechanism. Expert Systems with Applications **221**, 119730 (2023). https://doi.org/https://doi.org/10.1016/j.eswa.2023.119730
12. Huang, Y., Peng, H., Liu, Q., Yang, Q., Wang, J., Orellana-Martín, D., Pérez-Jiménez, M.J.: Attention-enabled gated spiking neural P model for aspect-level sentiment classification. Neural Networks **157**, 437–443 (2023). https://doi.org/https://doi.org/10.1016/j.neunet.2022.11.006
13. Ionescu, M., Păun, Gh., Yokomori, T.: Spiking Neural P Systems. Fundamenta Informaticae **71**(2,3), 279–308 (2006)
14. Koza, J.R.: Genetic Programming: On the Programming of Computers by Means of Natural Evolution. MIT Press (1992)

15. Li, B., Peng, H., Wang, J., Huang, X.: Multi-focus image fusion based on dynamic threshold neural P systems and surfacelet transform. Knowledge-Based Systems **196**, 105794 (2020). https://doi.org/https://doi.org/10.1016/j.knosys.2020.105794
16. Li, Y., Song, B., Zeng, X.: Spiking neural P systems with weights and delays on synapses. Theoretical Computer Science **968**, 114028 (2023). https://doi.org/https://doi.org/10.1016/j.tcs.2023.114028
17. Liu, Q., Huang, Y., Yang, Q., Peng, H., Wang, J.: An Attention-Aware Long Short-Term Memory-Like Spiking Neural Model for Sentiment Analysis. International Journal of Neural Systems **33**(08), 2350037 (2023). https://doi.org/10.1142/S0129065723500375
18. Maass, W.: Networks of spiking neurons: The third generation of neural network models. Neural Networks **10**(9), 1659–1671 (1997). https://doi.org/10.1016/S0893-6080(97)00011-7
19. Mitchell, M.: An Introduction to Genetic Algorithms. MIT Press (1998)
20. Neri, F., Cotta, C., Moscato, P.: Handbook of Memetic Algorithms. Springer Publishing Company (2011). https://doi.org/10.1007/978-3-642-23247-3
21. Pan, L., Păun, Gh.: Spiking Neural P Systems with Anti-Spikes. International Journal of Computers Communications & Control **4**(3), 273–282 (2009)
22. Păun, Gh.: Introduction to Membrane Computing, pp. 1–42. Springer Berlin Heidelberg (2006). https://doi.org/10.1007/3-540-29937-8_1
23. Păun, Gh., Rozenberg, G., Salomaa, A.: The Oxford Handbook of Membrane Computing. Oxford University Press, Inc. (2010)
24. Peng, H., Bao, T., Luo, X., Wang, J., Song, X., Riscos-Núñez, A., Pérez-Jiménez, M.J.: Dendrite P systems. Neural Networks **127**, 110–120 (2020). https://doi.org/https://doi.org/10.1016/j.neunet.2020.04.014
25. Peng, H., Li, B., Wang, J., Song, X., Wang, T., Valencia-Cabrera, L., Pérez-Hurtado, I., Riscos-Núñez, A., Pérez-Jiménez, M.J.: Spiking neural P systems with inhibitory rules. Knowledge-Based Systems **188**, 105064 (2020). https://doi.org/https://doi.org/10.1016/j.knosys.2019.105064
26. Peng, H., Wang, J.: Coupled Neural P Systems. IEEE Transactions on Neural Networks and Learning Systems pp. 1–11 (10 2018). https://doi.org/10.1109/TNNLS.2018.2872999
27. Peng, H., Wang, J., Pérez-Jiménez, M.J., Riscos-Núñez, A.: Dynamic threshold neural P systems. Knowledge-Based Systems **163**, 875–884 (2019). https://doi.org/https://doi.org/10.1016/j.knosys.2018.10.016, https://www.sciencedirect.com/science/article/pii/S0950705118305021
28. Peng, H., Yang, J., Wang, J., Wang, T., Sun, Z., Song, X., Luo, X., Huang, X.: Spiking neural P systems with multiple channels. Neural Networks **95**, 66–71 (2017). https://doi.org/10.1016/j.neunet.2017.08.003
29. Păun, Gh.: Spiking Neural P Systems with Astrocyte-Like Control. JUCS - Journal of Universal Computer Science **13**(11), 1707–1721 (2007). https://doi.org/10.3217/jucs-013-11-1707
30. Ryan, C., O'Neill, M., Collins, J.J.: Handbook of Grammatical Evolution. Springer Publishing Company (2018)
31. Song, X., Valencia-Cabrera, L., Peng, H., Wang, J.: Spiking neural P systems with autapses. Information Sciences **570**, 383–402 (2021). https://doi.org/https://doi.org/10.1016/j.ins.2021.04.051
32. Stone, S., Pillmore, B., Cyre, W.: Crossover and mutation in genetic algorithms using graph-encoded chromosomes. In: GECCO 2004 Late Breaking Papers (2004)

33. Storn, R., Price, K.: Differential Evolution – A Simple and Efficient Heuristic for global Optimization over Continuous Spaces. Journal of Global Optimization **11**(4), 341–359 (1997). https://doi.org/10.1023/A:1008202821328
34. Thölke, H., Kosiol, J.: A multiplicity-preserving crossover operator on graphs (2022). https://doi.org/10.48550/arXiv.2208.10881
35. Wu, T., Neri, F., Pan, L.: On the tuning of the computation capability of spiking neural membrane systems with communication on request. International Journal of Neural Systems **32**(08), 2250037 (2022)
36. Wu, T., Păun, A., Zhang, Z., Pan, L.: Spiking Neural P Systems With Polarizations. IEEE Transactions on Neural Networks and Learning Systems **29**(8), 3349–3360 (2018). https://doi.org/10.1109/TNNLS.2017.2726119
37. Xian, R., Lugu, R., Peng, H., Yang, Q., Luo, X., Wang, J.: Edge Detection Method Based on Nonlinear Spiking Neural Systems. International Journal of Neural Systems **33**(01), 2250060 (2023). https://doi.org/10.1142/S0129065722500605
38. Xian, R., Xiong, X., Peng, H., Wang, J., de Arellano Marrero, A.R., Yang, Q.: Feature fusion method based on spiking neural convolutional network for edge detection. Pattern Recognition **147**, 110112 (2024). https://doi.org/https://doi.org/10.1016/j.patcog.2023.110112
39. Yan, J., Zhang, L., Luo, X., Peng, H., Wang, J.: A novel edge detection method based on dynamic threshold neural P systems with orientation. Digital Signal Processing **127**, 103526 (2022). https://doi.org/https://doi.org/10.1016/j.dsp.2022.103526
40. Yang, B., Qin, L., Peng, H., Guo, C., Luo, X., Wang, J.: SDDC-Net: A U-shaped deep spiking neural P convolutional network for retinal vessel segmentation. Digital Signal Processing **136**, 104002 (2023). https://doi.org/https://doi.org/10.1016/j.dsp.2023.104002
41. Zhang, G., Rong, H., Neri, F., Pérez-Jiménez, M.J.: An optimization spiking neural P system for approximately solving combinatorial optimization problems. International Journal of Neural Systems **24**(05), 1440006 (2014). https://doi.org/10.1142/S0129065714400061
42. Zhang, G., Zhang, X., Rong, H., Paul, P., Zhu, M., Neri, F., Ong, Y.S.: A Layered Spiking Neural System for Classification Problems. International Journal of Neural Systems **32**(08), 2250023 (2022). https://doi.org/10.1142/S012906572250023X
43. Zhang, Y., Yang, Q., Liu, Z., Peng, H., Wang, J.: A Prediction Model Based on Gated Nonlinear Spiking Neural Systems. International journal of neural systems **33**, 2350029 (04 2023). https://doi.org/10.1142/S0129065723500296
44. Zhu, M., Yang, Q., Dong, J., Zhang, G., Gou, X., Rong, H., Paul, P., Neri, F.: An Adaptive Optimization Spiking Neural P System for Binary Problems. International Journal of Neural Systems **31** (06 2020). https://doi.org/10.1142/S0129065720500549