A Methodology for Evaluating the Extensibility of Boolean Networks' Structure and Function

Rémi Segretain, Sergiu Ivanov, Laurent Trilling, Nicolas Glade

A Methodology for Evaluating the Extensibility of Boolean Networks' Structure and Function

Rémi Segretain[1], Sergiu Ivanov[2], Laurent Trilling[1] and Nicolas Glade[1]

[1] University Grenoble Alpes, CNRS UMR5525, CHU Grenoble Alpes, Grenoble INP, TIMC-IMAG, F-38000 Grenoble, France
{remi.segretain,laurent.trilling,nicolas.glade}@univ-grenoble-alpes.fr
[2] IBISC, Univ Évry, Paris-Saclay University, 91025, Évry, France
sergiu.ivanov@ibisc.univ-evry.fr

Abstract. Formal interaction networks are well suited for representing complex biological systems and have been used to model signalling pathways, gene regulatory networks, interaction within ecosystems, etc. In this paper, we introduce Sign Boolean Networks (SBNs), which are a uniform variant of Threshold Boolean Networks (TBFs). We continue the study of the complexity of SBNs and build a new framework for evaluating their ability to extend, *i.e.* the potential to gain new functions by addition of nodes, while also maintaining the original functions. We describe our software implementation of this framework and show some first results. These results seem to confirm the conjecture that networks of moderate complexity are the most able to grow, because they are not too simple, but also not too constrained, like the highly complex ones.

Keywords: Biological Regulation, Biological Networks, Sign Boolean Networks, Complexity, Extensibility, Network Growth

1 Introduction

Numerous biological systems (gene systems, ecosystems, metabolic systems, *etc.*) can be modeled by formal interaction networks, in which the nodes of the network embody their constituting elements (genes, living species or individuals, molecules, *etc.*) and the directed edges their interactions [3, 4]. In this paper we introduce *Sign Boolean Networks* (SBNs), a particular class of Boolean Networks, which present the advantage of being a quite simple formalism that allows performing numerous large computations of various types, and which easily express formal networks in a logical constraint-based programming language, while also capturing the whole structure and functioning of real systems.
As SBNs are Boolean, the nodes' states indicate the presence and absence or activation and inhibition of biological species. Nevertheless, it is easy to replace Boolean states by a multi-valued ones, or to simulate a multi-valued state by a set of Boolean nodes, in order to represent biological amounts more realistically (*e.g.* [5]). Moreover weighted edges reflect the relative strength of interactions that biological elements exert onto each other. Then, the functioning of a biological system is readable as a deterministic transition graph in which fixed points

and cyclic attractors can be reached by the network's nodes from one initial state or another (*e.g.* [4]).
Biological networked systems evolve: genes that are subject to genetic changes like mutation, duplication or deletion, or living species that suffer from invasive species within their ecosystems, make such systems adapt to new configurations. As long as biological systems are subject to natural selection under environmental constraints, all selected features give the living system they support an evolutionary advantage. They evolve when new elements (new genes or duplicates, invasive species), start interacting with the existing ones, when existing elements are deleted, or when mutations modify existing elements or the manner in which they interact by modulation of their strength or specificity. When new biological elements start to interact with the existing ones, networks grow toward larger structures. However, during evolution, the cards are rarely entirely reshuffled: maintaining the functioning of the whole system, especially when it is complex, seems essential to its survival. Some additions could indeed change the whole functioning of a network, breaking the precious biological function it supports.
The question we are interested in and that motivates this methodological article is: *How can a network grow to a larger one while keeping both the original network as an inner module, and its original function as a part or a sub-function of the more complex one?*
More formally, we are searching for all networks N of dimension $d+1$ (*i.e.* made of $d+1$ nodes) that produce a repeating binary word S (that we associate to a behavior [2]) on at least one node, from all networks N_1 of dimension d that play a repeating binary word S_1 on at least one node, such that $N_1 \subset N$ (meaning that N contains the structure of network N_1) and $S_1 \subset S$ (the behavior S_1 of N_1 becomes a sub-pattern of the extended behavior S).

Moreover, we ask how such constrained extension ability is related to both network structure and function. We indeed expect certain couples $\{N_1, S_1\}$ to be more extensible than others, thus yielding more networks N than others. Complexity is a manner to characterize network structure or behavior as previously proposed [2]. How this ability to grow relates to network and behavior complexities is an open question that could give clues on how real-world networked systems could evolve. Does the growth of a network entail an increase in robustness of its functions (*i.e.* an increase of the size of the corresponding basins of attraction, collection of states that converge to a given asymptotic behavior), or does the number of functions it can perform (*i.e.* the number of asymptotic behaviors, or attractors) augment instead?
Before being able to realize such studies and answer these questions, we need powerful algorithms to explore the large sets of extended networks yielded by the structural and dynamical constraints expressing the conservation of the original structure and function of network N_1. To help the reader asses the huge amount of computations we are dealing with, the number of quadruplets (*original network, original behaviour, extended network, extended behaviour*) approaches 770000 for networks N_1 of dimension 2, and for one-node extensions only (*i.e.* extensions to the space of networks of dimension 3). We also need efficient methods

to compute the properties of the obtained networks, including structural and behavioural network complexities, but also the characteristics of their dynamics like the number of attractors of the original and the extended networks, or the size of their basins of attraction. In this paper, we first present the formalism of Sign Boolean Functions (SBFs) and Networks (SBNs), and the manner in which we solve the extension problem and compute network and behavior complexities.

2 Definitions and Methodology

2.1 Sign Boolean Networks

A Boolean function is any function $f : \{0,1\}^n \to \{0,1\}$. A Boolean network is typically defined as a tuple (V, F), where V is the set of variables of the network and F is a mapping associating a Boolean function to each variable in V. For any given $x \in V$, the Boolean function associated to x, $F(x)$, computes the new value for x from the values of the variables in V. Depending on the types of update functions F, one can distinguish between different kinds of Boolean networks [1, 2, 9–13, 17].
Threshold Boolean Networks (TBNs) are a particular kind of Boolean networks in which the updates of Boolean variables depend on the sum of weighted interactions compared to a threshold that represents the level of activation of the Threshold Boolean Function (TBF), as follows:

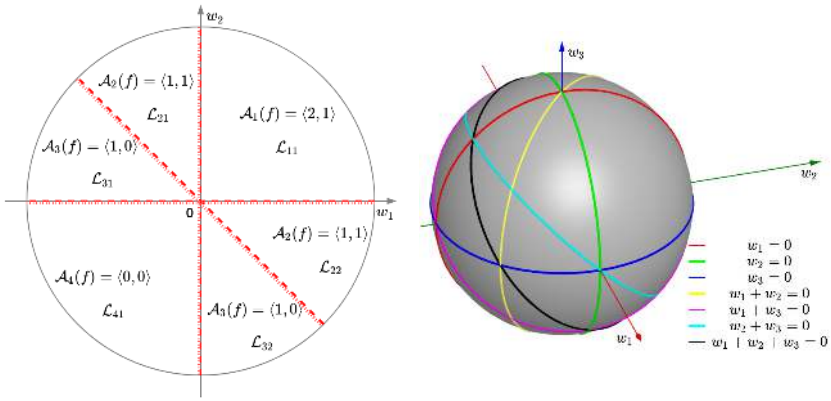
Definition 1. *Given d weights $w_1, \ldots, w_d \in \mathbb{R}$ and the threshold $\theta \in \mathbb{R}$, a* TBF *f given by $(w_1, \ldots, w_d, \theta)$ is defined as follows:*

$$f(x_1, \ldots, x_d) = \begin{cases} 1, & \text{if } \sum_{i=1}^{d} w_i x_i > \theta, \\ 0, & \text{otherwise.} \end{cases}$$

Compared to other types of Boolean Networks, TBNs capture the inhibiting and promoting relationships in regulatory networks with far fewer parameters [13]. It turns out however that using two types of parameters for a TBF, the weights and a threshold, creates a strong asymmetry for complexity analysis (see [2, Section 2.2.2]). Indeed, to compute TBF complexities, we were evaluating the volume of their equivalence classes in the unit ball in the space of parameters $\{W, \theta\}$ which is not symmetrical in all directions due to the parameter θ, because the nature of the threshold is quite different from that of the weights. Networks of TBFs are thus heterogeneous, and dealing with their heterogeneity may be rather challenging. Therefore, we settled to use the more uniform variant: *Sign* Boolean Networks (SBNs), consisting of Sign Boolean Functions (SBFs):

Definition 2. *Given d weights $w_1, \ldots, w_d \in \mathbb{R}$, a* Sign Boolean Function *f is defined by its weights $\langle w_1, \ldots, w_d \rangle$ as follows:*

$$f(x_1, \ldots, x_d) = \begin{cases} 1, & \text{if } \sum_{i=1}^{d} w_i x_i > 0, \\ 0, & \text{otherwise.} \end{cases}$$



(a) 2D weight space (b) 3D weight space

Fig. 1: **SBF parameter space and equivalence classes.** (a) The dashed lines delimit different conic regions of the unit ball of SBF parameters within which all SBFs share the same truth table. The rug lines indicate the equivalence class into which a dashed line is included. Each equivalence class is designated by a value of $\mathcal{A}_i(f)$ and can comprise multiple layouts $\mathcal{L}_{ij}$ of SBFs represented by the different cones in the unit ball [2]. In dimension 2, there are four different equivalence classes with the abstract representations $\mathcal{A}(f) = \langle 0,0 \rangle$, $\langle 0,1 \rangle$, $\langle 1,1 \rangle$ or $\langle 2,1 \rangle$ (see Section 2.1). (b) In 3 dimensions, each line represents an intersection between a 2D plane and the unit sphere. Each plane is defined by an equation over the weights detailed in the legend. As in (a) equivalence classes are constituted by one or several cones.

It is easy to show that any given TBF_i (with threshold θ_i) can always be replaced by a combination of two SBFs. Indeed, take SBF_i (threshold 0) which has all the inputs of TBF_i with the same weights, plus an input with weight $w_{ij} = -\theta_i$ connected to the self-activated SBF_j (threshold 0) with a loop arc $w_{jj} = 1$ and initialized to 1. SBF_j will be constantly activated, so for SBF_i to produce 1, the sum of all its other inputs (*i.e.* the original inputs of TBF_i) plus $-\theta_i$ should be greater than 0, meaning that the simple network consisting of SBF_i and SBF_j simulates TBF_i faithfully. So, the threshold of a TBF can be seen as incorporating a particular kind of a sub-network into the TBF itself!

Remark 1. While any given TBF can be simulated by two SBFs, and therefore any given TBN can be simulated by an SBN, the expressive power of individual SBFs is inferior to that of individual TBFs: consider for example the single-input TBF f_σ given by the tuple $(w_1 = 1, \theta = -1)$. $f_\sigma(x) = 1$, for any $x \in \{0,1\}$. There exists no SBF with the same truth table, because any SBF is 0 when all of its inputs are 0.

Remark 2. We mainly use integer values for weight parameters because they are easier to work with but real values can also be used. Only the relationship between the weights and their signs is important. In particular, we use floating point values comprised between -1 and 1 to compute SBF complexities (see section 2.3). For example $W(f) = \langle -0.1, 0.2\rangle \equiv \langle -1, 2\rangle$.

Equivalence classes and layouts of SBFs The d-dimensional parameter space (weight space) can be divided into several regions that contain instances of SBFs. These regions are delimited by $(d-1)$-dimensional planes that correspond to conditions on the weights. For example, in the 2D parameter space shown in Figure 1, the quadrant designated by $\mathcal{A}_1(f)$ corresponds to all SBFs in which $w_1 > 0$ and $w_2 > 0$.
All SBFs f that present the same truth table, up to a permutation of their inputs, belong to the same equivalence class, denoted $\mathcal{A}(f)$. We use the same notation to refer to the abstract representation of SBFs (see below). Within an equivalence class $\mathcal{A}_i(f)$, a particular order of the inputs in the truth table is called the *layout* $\mathcal{L}_{ij}(f)$ of this equivalence class.

Minimal SBF and minimal SBN For integer weights, one can always define a *minimal* SBF as the SBF for which the sum of absolute values of weights is minimal. Such a minimal SBF is unique, up to the layout. All SBFs can be reduced to such a minimal SBF, without impacting its truth table. For example, a SBF having the weights $\langle -1, 3\rangle$ can be reduced to the minimal SBF $\langle -1, 2\rangle$.
The corollary is that any SBN can be reduced to a minimal SBN in which all SBFs are minimal. For a given configuration of the connections between its SBFs, the minimal SBN is unique.

Abstract representation of SBFs A Boolean vector $X = \langle x_1, ..., x_d\rangle \in \{0,1\}^d$ represents the activation state of the inputs of a SBF f of dimension d. There are 2^d possible configurations for X forming all the entries of the truth table of f. It is possible to classify these configurations according to the number of activated inputs (inputs set to 1). The resulting classes are called Nai_i, for *Number of activated inputs* $= i$, $i \in [0; d]$. For $d = 3$ we have the four classes Nai_0, Nai_1, Nai_2 and Nai_3.
Then, a SBF can be described by a vector of integers $Y = \langle y_0, ..., y_d\rangle$, where the y_i is the number of configurations of X in Nai_i in which the sum of weights linked to the activated inputs is strictly positive:

$$y_i = \sum_{X \in Nai_i} \begin{cases} 1 & \text{if } \sum_{x_j \in X} x_j w_j > 0 \\ 0 & \text{otherwise} \end{cases}$$

Remark 3. By construction, Nai_0 will always contain only the configuration where all the inputs are deactivated, *i.e.* $\mathbf{0} = \langle 0, 0, \ldots, 0\rangle$. Since no SBF f is activated on $\mathbf{0}$ ($f(\mathbf{0}) = 0$ for any SBF f), y_0 will always be equal to 0. Therefore we choose to discard y_0 from the vectors Y in the following.

The vectors Y allow us to define the following notion capturing the fundamental equivalence of SBFs with respect to reordering of the inputs.

Definition 3. *The* abstract representation *of a SBF f is the vector*

$$\mathcal{A}(f) = \langle y_1, ..., y_d \rangle$$

where y_i is the number of Boolean input vectors with i activated inputs for which the SBF is 1 (as above).

The abstract representations of any two SBFs f_1 and f_2, whose weight vectors are permutations of one another, are the same: $\mathcal{A}(f_1) = \mathcal{A}(f_2)$. We empirically verified the converse statement for dimensions $d \in \{1, 2, 3\}$: any two minimal SBFs f_1 and f_2 of dimension $d \in \{1, 2, 3\}$ with the property $\mathcal{A}(f_1) = \mathcal{A}(f_2)$ are given by weight vectors which are permutations of one another. For example, as shown in Table 1, $f_1(x_1, x_2) = f_2(x_2, x_1)$ and $\mathcal{A}(f_1) = \mathcal{A}(f_2)$.

	X		f_1			f_2		
	x_1	x_2	$f_1(X)$	inequalities	Y	$f_2(X)$	inequalities	Y
Nai_0	0	0	0		0	0		0
Nai_1	1	0	1	$w_1 > 0$	1	0	$w_1 \leq 0$	1
	0	1	0	$w_2 \leq 0$		1	$w_2 > 0$	
Nai_2	1	1	1	$w_1 + w_2 > 0$	1	1	$w_1 + w_2 > 0$	1

Table 1: Example of two equivalent SBFs of dimension 2 sharing a common abstract representation.

Interaction Graph of a SBN The *interaction graph* of a SBN is the weighted directed graph in which the nodes are the variables of the SBN, and which contains the weighted edge $E_{n_A n_B} : n_A \xrightarrow{w} n_B$ between nodes n_A and n_B if the SBF updating n_B receives the state of n_A weighted by w.
Let us remark that:

- the output of a SBF only depends on the sign of the weighted sum of its inputs,
- the interaction graph of a SBN completely describes the SBN,
- setting a weight to 0 models an absence of interaction and consequently an absence of the corresponding edge,
- we require networks to have connected interaction graphs: disconnected nodes are not allowed.

Transition Graph of a SBN A state of a SBN with d nodes $\{n_1, ...n_d\}$ is a vector $s = \langle x_1, \dots x_d \rangle \in \{0, 1\}^d$ giving the value of each of the nodes of the SBN.

A network updates all of its nodes at every step. In this article we only consider updates under the parallel update mode (also called synchronous mode), however, our method may be adapted to non-synchronous update modes. The initial state of a SBN is given by the vector $s_0 \in \{0,1\}^n$, which sets the initial values of the nodes of the network. We will often refer to nodes by their update functions (SBFs). While several nodes may have the same update function, we assume that the labels of the SBFs are different and are in bijective correspondence with nodes.
Given a SBN N, its transition graph is a graph whose vertices are the states of N and which has an edge $s_1 \to s_2$ iff updating all the nodes in s_1 according to their update functions yields s_2. The dynamics of a SBN is deterministic: if the transition graph contains the edges $s_1 \to s_2$ and $s_1 \to s_3$, then $s_2 = s_3$. Consequently, the connected components of the state graph are cycles, possibly with some pre-cycle (non-cyclic prefixes) attached. These cycles are generally posited to correspond to particular behaviours (phenotypes) of biological networks.
The output of a SBN N is recovered first by designating an output node and then listing the successive values it may have when N evolves from a particular starting state. Since the dynamics of a SBN always ends up in a cycle, and since SBN never stop updating their states, any node output sequence they generate has the form uv^*, $u, v \in \{0,1\}^*$ (that is, u is a prefix and v is a suffix which can be repeated arbitrarily many times). Because we limit the definition of behavior to the repeated suffix v^*, we will ignore the prefix u and associate the sequence S played by a node of N with the repeating suffix v^*.

2.2 Extension of structures and behaviours

We will now define the central question addressed in this paper: the *ability of SBNs to extend by addition of new elements while maintaining existing structure and function*. To formulate the extension problem in a *logical way* we only need to fix a given binary sequence S_1, a suffix S_k ($S_1, S_k \in \{0,1\}^*$), the dimension d, and the constraints over the quadruplet (N_1, S_1, N, S) as follows :

- N_1 is a SBN composed of d nodes,
- N_1 shows the behavior described by S_1, on a node n_i. Meaning that the successive states of node n_i along a cycle in the transition graph match the binary sequence S_1,
- S is a binary sequence defined as $S = S_1 \cdot S_k$, where $\cdot$ stands for sequence concatenation,
- N is a SBN composed of $d+1$ nodes,
- N shows the behavior S on the same node n_i as in N_1,
- N contains a sub-network N_1' such that:
  - N_1' is composed of d nodes.
  - N_1' and N_1 share the same structure of edges:

$$\forall Edge_{n_i \to n_j} \in N_1, \exists! Edge'_{n_i \to n_j} \in N_1'.\ i, j \in [1; d],$$

- N_1' and N_1 share the same SBFs:

$$\forall f_i \in N_1, \exists! f_i' \in N_1', f_i \equiv f_i'.\ i \in [1; d],$$

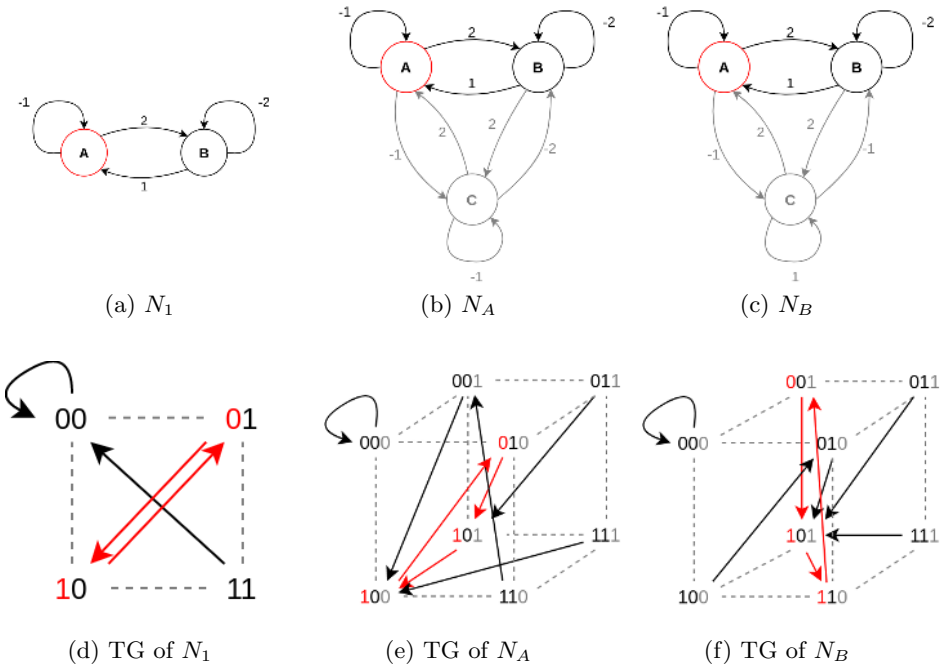
  where $\equiv$ stands for equivalence as defined in Section 2.1.
- N_1' and N_1 possibly differ regarding the weights of their edges with the following restriction:

$$\forall w_{n_i \to n_j} \in N_1, \exists! w'_{n_i \to n_j} \in N_1', w_{n_i \to n_j} \leq w'_{n_i \to n_j}.\ i, j \in [1; d].$$

Figure 2 gives two detailed examples of extensions of 2-dimensional SBNs.
In practice, due notably to the huge number of possible d-size networks and associated binary sequences, it is not possible to directly calculate all the quadruplets (N_1, S_1, N, S) in a reasonable CPU time and amount of memory. To avoid this issue, we used a particular (N_1, S_1) instance-centered implementation of this logical formulation of the network and behaviors extension problem: for a given couple (N_1, S_1) and a given behavior extension S_k, we infer (in ASP) the set of networks N that comply with those constraints. In order to avoid duplicates of N_1 networks, *i.e.* keep only one network instance per equivalence class, we must exploit the abstract representation of their functions presented above (see Section 2.1). Full details concerning the implementation of this computation involving a Java processing pipeline orchestrating ASP modules are given in appendices A and B.

2.3 Complexity of SBNs and SBFs

In a recent past, we described a manner to compute both the complexity of binary sequences S played by network nodes (sequences that we associate to *behaviors*) and the structural complexity of TBFs and TBNs [2]. In the present paper, we present an improved version of the latter. As before for TBFs, to compute the structural complexity of a SBF f, we consider the equivalence class $\mathcal{A}(f)$ of f and evaluate the probability $\mathcal{P}$ of randomly picking an instance of f in this equivalence class, under the uniform distribution over the unit ball of parameter space (Figure 1 illustrates this for SBFs). The complexity of a given TBF was defined as the inverse of this probability. To compute the complexity of a TBN N in [2], we used to multiply the individual complexities of its constituent TBFs.
This computation however presents an issue : it does not take into account the way these TBFs are connected together *i.e.* the topology of the network and the associated parameters (weights), in other terms its structure. In the present paper, we now take network structure into account in the form of another probability measurement $\mathcal{C}^s$. Furthermore, although the complexity of a TBF $\mathcal{C}^f$ is related to a probability, it is not a probability anymore. Combining it with the structural complexity C^s is therefore not easy. We decided to make it more uniform: both SBFs complexities $\mathcal{C}^f$ and structural complexities $\mathcal{C}^s$ are now probabilities. In order to do that we updated C^f to $\mathcal{C}^f = 1 - \mathcal{P}(\mathcal{A}(f))$. C^f is therefore the probability of *not* picking f. Thus, the complexity $\mathcal{C}^f$ of a function



(a) N_1 (b) N_A (c) N_B

(d) TG of N_1 (e) TG of N_A (f) TG of N_B

Fig. 2: **Two examples of SBN extension from a 2D SBN.** In the networks N_1 (a), N_A (b) and N_B (c), the nodes are labelled A, B or C. The node for which we follow the behavior is red and the additional node in the extended networks is gray. The corresponding transition graphs (TG) are shown below in (d), (e) and (f). In the TGs, the states and behavior of node A are shown in red while the state of the additional node in extended networks is shown in gray. From network N_1 (a) and its asymptotic behavior $S_1 = (10)^*$ (d) played on node A, networks N_A (b) and N_B (c) are examples of extended networks that can be found when asking for an extension of N_1 by one node, and an extension of one of its behaviors from $S1 = (10)^*$ to $S = (101)^*$. Within the potentially large set of extended networks that satisfy the constraints given above, in this case 48 extended SBN from N_1, S_1 and S, some will partly preserve the initial transition graph as shown in (e) for network N_A, while it may be largely reconfigured for other as shown in (f). In any case, the structure of network N_1 is preserved in the extended networks, and the initial behavior $(10)^*$ is encapsulated in a larger one, here $(101)^*$.

f is high when $\mathcal{P}(\mathcal{A}(f))$ is low.

We choose the structural complexity $\mathcal{C}^s$ to be a centrality measure because centrality naturally expresses the concept of influence of one node n_i on another one. We construct $\mathcal{C}_i^s$ as follows: for each directed edge of the network $E_{ij} : n_i \xrightarrow{w_{ij}} n_j$,

$(i, j \in [1; d])$, we define the influence probability $P^I(E_{ij})$ that node n_i influences node n_j due to the weighted directed edge E_{ij}, among all the incoming edges E_{kj} that influence node n_j:

$$P^I(E_{ij}) = \frac{|w_{ij}|}{\sum_{k=1}^{d} |w_{kj}|}.$$

We can then define the P^I of a longer path, going from a node a to a node b ($P^I(Path_{ab})$) as the geometric mean of the influence probabilities $P^I(E_{ij})$ of the edges of this path, $E_{ij} \in Path_{ab}$:

$$P^I(Path_{ab}) = \sqrt[L_{ab}]{\prod_{E \in Path_{ab}} P^I(E)}.$$

where L_{ab} it the length of the path.

Remark 4. In the former equation, we express the mean influence of a given path as an average probability that can be compared to that of other paths. We used the geometric mean because each SBF along the path receives a variable number of input edges, so the influence $|w_{ij}|$ at each step on one node (one edge E_{ij}) may be normalized by a different number of weights, from 1 (single edge) to d (edges from all nodes including n_j).

Since several different paths can exist from a to b, the general probability P^I_{ab} that a influences b must take into account all of them. We therefore calculate P^I_{ab} as follows:

$$P^I_{ab} = P\left(\bigcup_{Path \in Paths_{ab}} Path\right).$$

where $Paths_{ab}$ is the set of all paths from node a to node b, and we overload $Path$ to refer both to a path from a to b and the event that node a influences node b over this path. The probability of this event is $P^I(Path)$.
We define the centrality $\mathcal{C}^s_i$ of a node n_i as the probability that it influences at least one node n_k, including itself:

$$\mathcal{C}^s_i = P\left(\bigcup_{k=1}^{d}\left(\bigcup_{Path \in Paths_{ik}} Path\right)\right).$$

Finally we use the centrality $\mathcal{C}^s_i$ of each node n_i to modulate the corresponding SBF complexity $\mathcal{C}^f_i$, and define the complexity of a network N as follows:

$$\mathcal{C}(N) = \prod_{i=1}^{d} \begin{cases} \mathcal{C}^f_i \times \mathcal{C}^s_i & \text{if } \mathcal{C}^s_i > 0 \\ 1 & \text{otherwise,} \end{cases}$$

Remark 5. Nodes with centrality equal to zero (*i.e.* the nodes that are pure readers, their state is never read by any other node including themselves) are *de facto* excluded from the network complexity calculation.

3 Discussion

In this article we presented a scientific question that concerns various biological networks including regulatory networks, ecosystems, neural networks, *etc.*: how do networks and their associated behavior grow together in the case the initial network structure and behavior must be preserved? We also wonder how much such extensible character is related to the complexity of networks and behaviors. Among the further research directions that emerge in this context, we hypothesize that most of the highly complex d-dimensional networks playing complex behaviors cannot extend into complex $(d+1)$-dimensional networks easily because they are too constrained. Conversely, too simple networks cannot play complex behaviors and cannot become larger complex networks, playing complex extended behaviors. We however expect networks of moderate structural and behavioral complexity to be the most capable of generating complex extended networks and behaviors when growing. In this case, the network structure and its asymptotic behavior are not too much linked, a large part of the transition graph being occupied by transitory states or other basins of attraction, so additional nodes do not necessarily break the initial behavior.

Here, we focus on the exhaustive description of the method we developed, from both the theoretical and the implementation point of views, to compute both network extensions and network characteristics like complexities. Using Sign Boolean Networks is well adapted to materialize our questions and test our hypotheses. SBNs are simple enough to work with in logical constraint programming and to limit the number of extensions obtained from any triplet (N_1, S_1, S). The homogeneous description of their constituting SBFs, entirely determined by their input weights, also allows expressing complexity scores for SBNs easier than with ordinary Boolean Networks or even Threshold Boolean Networks.

We tested our method by computing the extension of 2-dimensional SBNs

d	SBF	SBN	$\{N_1, S_1, S\}$	N
2	7	101	96 10^3	777216
3	17	206 10^3	[180 10^6 – 32 10^9]	[1.5 10^9 – 280 10^9]
4	47	76 10^9		

Table 2: Number of SBFs, SBNs, triplet (N_1, S_1, S) and extended networks N per dimension d. Ranged values are estimations based on SBN extension from dimension 2 to 3.

towards 3-dimensional SBNs, *i.e.* 777216 quadruplets $(N_1^{d=2}, S_1, N^{d=3}, S)$. For each network (initial and extended), we also computed several characteristics including network and behavioral complexities, but also the number of attractors and the size of basins of attraction. The volume of data to process is then already huge when starting from dimension 2 for which only 7 SBFs and 101 SBNs exist. Our program running on a workstation with 32 CPU threads (2 x Intel(R) Xeon(R) CPU E5-2667 v4 @ 3.20GHz) and 64Go RAM (ECC DDR4

2400MHz) took 20 minutes to complete. This is quite efficient but computation times for extensions from larger dimensions may grow fast as shown in Table 2. To be applicable to larger networks, in particular biological ones, our method should be scalable. Looking for exhaustive results, *i.e.* infer all quadruplets (N_1, S_1, N, S) as we did for 2 to 3 dimensional extensions but for larger dimensions, the computation as it is realized here may quickly reach its own limits. The interest of exhaustive exploration is to embrace the entire diversity of network structures and behaviors, and study their relationships in a systematical way. At the least, one would aim to yield comparable information for larger dimensions or for larger extensions (*i.e.* extensions that involve more than one additional node at a time). The get around is to downgrade the exploration of quadruplets. There are several ways by which this can be done. In larger dimensions, a manner to save computational time is to use Monte Carlo approaches. Another manner to reduce the number of quadruplets to compute while allowing us to explore larger extensions, is to make thin slices in initial and extended behavior complexities (*e.g.* 3 slices along initial behavior complexities: small, moderate and large complexities, and 3 slices in extended behaviors also limited to small, moderate and large complexities). Finally, even the extension of very large specific networks (dozens of nodes) and for specific behaviors could be computed in a reasonable time using the same ASP modules. All these downgraded exploration methods will be evaluated on the benchmark that constitutes our full 2 to 3 dimension exploration. In addition, from the technical point of view, computation of larger extensions (*e.g.* dimension 3 to 4) will take advantage of the optimization realized (modular structure, parallelism, *etc.*) in our implementation. We are now only beginning to analyse and interpret the results obtained for the 2D to 3D extension and will compute partial data for larger extensions soon. As an example of what is obtained for the 2D to 3D extension, the 3D histogram in Figure 3, shows notably that most 2D SBN behaviors are not complex and most of them, when extended, show behaviors of limited complexity played by extended networks of limited complexity too. It also shows that complex extended behaviors and networks (red points) are mostly obtained from networks of moderate complexity. Although such a result seems to go in the good direction, this brief result overview must be refined with reinforced statistic (*e.g.* min, max, standard deviation, in addition to average complexity values) of network and behavior complexities. Increase in behavior complexity is not the only way networks can evolve: extended networks can also develop new behaviors (increase of the number of attractors) or the increase of complexity of networks is used to reinforce the robustness of their behaviors, *i.e.* by increasing the size of their basins of attraction. Finally, beyond the theoretical study, the position of the complexity cursor in biological systems is an open question. We expect our work to give clues on how networked systems can or cannot evolve as they are constrained by the existing conditions and by the necessity to maintain vital functions. A specific article will be dedicated to this analysis and to developing the biological question.

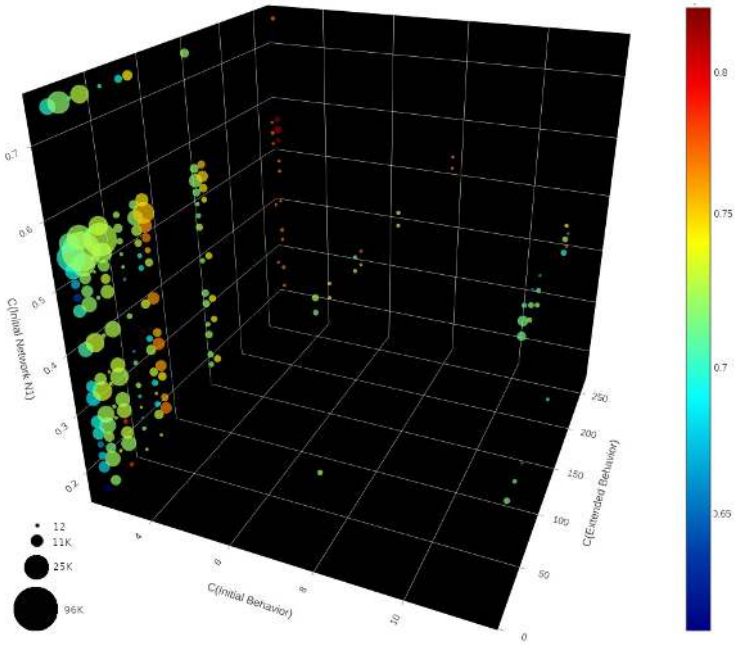


Fig. 3: **3D histogram of network structure and behavior complexities.** Counts of all 777216 (N_1, S_1, N, S) quadruplets obtained by extension are divided into 3D classes of complexities: network N_1 complexity along the *C(Initial Network N_1)* axis, initial S_1 and extended S behavior complexities along the *C(Initial Behavior)* and *C(Extended Behavior)* axes respectively. Point size (in logarithmic scale) denotes the number of networks in a class, while their color corresponds to the average complexity of extended networks N in this class.

Acknowledgements

Sergiu Ivanov is partially supported by the Paris region via the project DIM RFSI n°2018-03 "Modèles informatiques pour la reprogrammation cellulaire". The authors would also like to thank the IDEX program of the University Grenoble Alpes for its support through the projects COOL : this work is supported by the French National Research Agency in the framework of the Investissements d'Avenir program (ANR-15-IDEX-02). This work is also supported by the Innovation in Strategic Research program of the University Grenoble Alpes. The authors would thanks Ibrahim Cheddadi for fruitful discussions.

References

1. Pardo, J., Ivanov, S., Delaplace, F., Sequential Reprogramming of Biological Network Fate; Lecture Notes in Computer Science, Proceedings of the Computational Methods in Systems Biology - 17th International Conference (2019)DOI: 10.1007/978-3-030-31304-3_2

2. Christen, U., Ivanov, S., Segretain, R., Trilling, L., Glade, N., On computing structural and behavioral complexities of threshold Boolean networks; Acta Biotheor. **68** 119–138 (2020)
3. Thomas, R.: On the relation between the logical structure of systems and their ability to generate multiple steady states or sustained oscillations; Springer Ser. Synerg. **9**, 180–193 (1980)
4. Mendoza, L., Alvarez-Buylla, E.R.: Dynamics of the genetic regulatory network for *Arabidopsis thaliana* flower morphogenesis; J. Theor. Biol. **193**, 307–319 (1998)
5. Delaplace, F., Ivanov, S.: Bisimilar Booleanization of multivalued networks; Biosystems **197**, in press (2020)
6. Delahaye J.-P., Zenil, H.: Numerical Evaluation of the complexity of short strings: a glance into the innermost structure of algorithmic randomness; Applied Mathematics and Computation **219**, pp. 63-77, (2012)
7. Soler-Toscano F., Zenil H., Delahaye J.-P., Gauvrit N.: Calculating Kolmogorov complexity from the output frequency distributions of small Turing machines; PLoS ONE 9(5): e96223 (2014).
8. Segretain R., Repository of the inference pipeline in ASP and java. https://gitlab.com/rsegretain/java-parallel-pipeline
9. Hopfield, J.J.: Neural networks and physical systems with emergent collective computational abilities; Proc. Natl. Acad. Sci. USA **79**, 2554–2558 (1982).
10. Glass, L., Kauffman, S.: The logical analysis of continuous, nonlinear biochemical control networks; J. Theor. Biol. **39**, 103—129 (1973).
11. Zañudo, J.G.T., Aldana, M., Martínez-Mekler, G. Boolean Threshold Networks: Virtues and Limitations for Biological Modeling. In: Niiranen S., Ribeiro A. (eds) Information Processing and Biological Systems. Intelligent Systems Reference Library, vol 11. Springer, Berlin, Heidelberg, 2011.
12. Bornhold, S.: Boolean network models of cellular regulation: prospects and limitations; J. R. Soc. Interface **5**, S85–S94 (2008).
13. Tran, V., McCall, M.N., McMurray, H.R., Almudevar, A.: On the underlying assumptions of threshold Boolean networks as a model for genetic regulatory network behavior; Frontiers Gen. **4**:263 (2013).
14. Gebser, M., Kaminski, R., Kaufmann, B., Ostrowski, M., Schaub, T., Schneider, M.: Potassco: The Potsdam Answer Set Solving Collection; AI Comm. **24**, 107–124 (2011).
15. Ostrowski, M., Schaub, T.: ASP modulo CSP: The clingcon system; Theory and Practice of Logic Programming, (2012).
16. Banbara, M., Inoue, K., Kaufmann, B., Okimoto, T., Schaub, T., Soh, T., Tamura, N. and Wanko, P. *teaspoon*: solving the curriculum-based course timetabling problems with answer set programming. Ann. Oper. Res. 275, 3–37 (2019).
17. Vuong, Q.-T., Chauvin, R., Ivanov, S., Glade, N., Trilling, L., A logical constraint-based approach to infer and explore diversity and composition in threshold Boolean automaton networks, Studies in Computational Intelligence Series; Proceedings of the Complex Networks 2017 conference (2017) DOI: 10.1007/978-3-319-72150-7_46

A Implementation of the extension problem

In section 2.2 we formulated our network extension problem in a logical way. As mentioned, such formulation is not effectively usable in its raw form. In practice, we first enumerate exhaustively all networks N_1, all their behaviours S_1 and all the extended behaviours S, then find all corresponding extended networks N that satisfy the constraints enumerated in section 2.2.
In the following, we first give an overview of the software architecture and then show how to solve the crucial issue of inferring efficiently only different networks N.

A.1 Overall software architecture

We used a combination of two programming languages, Answer Set Programming (ASP) [14], a non-monotonic logical based programming language, and Java, a classical imperative language. The main software, written in Java, orchestrates the execution and uses multiple ASP modules when needed.

Inference modules in ASP We first give here a very short introduction to ASP. This logical language allows to express facts and rules, like Prolog, with the help of logical literals. For example, the following rules `p(1). p(2).` and `c :- p(1), p(2).` mean that the two facts `p(1)` and `p(2)` are true and that their conjunction implies `c`.
An ASP program infers all logical models (sets of literals) that comply with the facts and rules it specifies: they are called Answer Sets (AS). With the help of *integrity constraints*, logically expressed as rules producing false, some AS can be eliminated. For example, let us consider the two rules `a :- not b.` and `b :- not a.` ; they accept the two different AS {`a`} and {`b`} (*i.e.* as `b` cannot be inferred, `b` is considered to be false in ASP, then `not b` is true and `a` is also true). If we add the fact `c.` and the integrity constraint `:- c, not b.` then only the AS {`b, c`} is valid and not {`a, c`}: the integrity constraint discriminates the ASs where the conjunction of `c` and `not b` is be false, then `b` should be true (and `a` is rejected to be true).
The ASP solver that we use, namely *clingcon* [15], proceeds in two steps. First, a *grounder* translates the rules to a propositional form (with only Boolean variables). Then a SAT-like algorithm is applied to this program. We greatly benefit from a recent improvement based on [16] which uses a lazy approach for grounding and then allows the use of numerical variables with a very large range.
In order to increase the flexibility and the re-usability of our ASP code, we cut it into several inference modules dedicated to specific tasks and compatible between each others. Each one introduces a main literal that can be set to configure its module :

- `sbn(N, D)` : implies literals describing a SBN `N` of dimension `D`.
- `composedSbn(`$\mathtt{N}_a$`,` $\mathtt{N}_b$`)` : constrains two given SBNs $\mathtt{N}_a$ and $\mathtt{N}_b$ so that one SBN is an extension of the other as defined in 2.2.

– `sequencePlayedBySbn(N, S, I)` : constrains a SBN `N` to play a given sequence `S` on his node of index `I`.
– `orderedSbn(N)` : from a given SBN `N`, generates the equivalent ordered one by permutation of nodes (see section A.2).

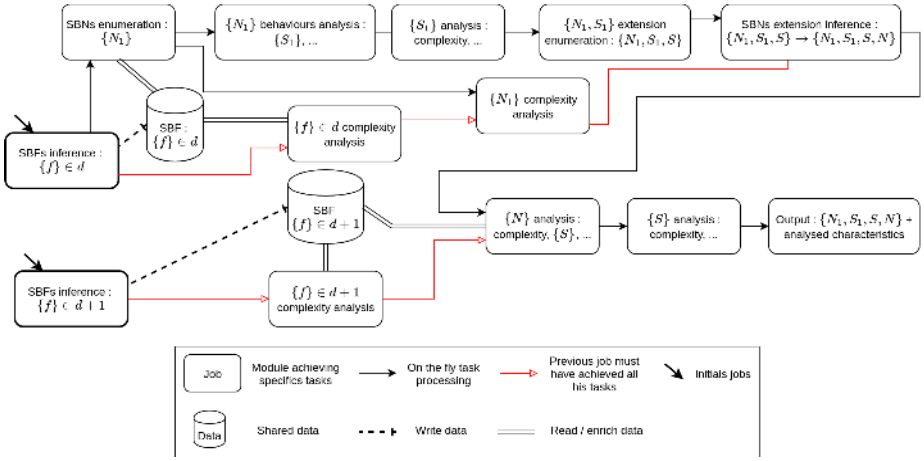


Fig. 4: **Overview of the processing pipeline, jobs division and ordering.** Initial jobs use inference modules to find the SBFs, which are used to generate the SBNs. Then, all SBN behaviors are analysed. The complexity of individual SBFs, SBNs and behaviors is determined and every reachable extended behaviors are listed (the program generates sequence with S_1 as sub-sequence). At this point we have got all the triplets (N_1, S_1, S). For each of them, inference modules are called another time to find all the extended networks N.

Processing Pipeline The main software is organized as a pipeline of processors (called jobs): each job, programmed in Java, does its own part of the work, then furnishes its results to the next job, *etc.* Using a custom Java library [8], the workload is divided into tasks that we scatter across the different jobs, allowing to make the execution parallel on any number of CPU cores, thus enhancing the overall performances. An overview of this pipeline is given figure 4. The potential of the combination of an imperative language and inference modules lay in the facts that we can take advantages from both: efficiently find data sets by ASP inference, filter and enrich them in the Java pipeline. Moreover, those enriched data set can be used in return to configure other calls to inference modules. This could not have been easily done using only ASP.
The jobs use inference modules by generating literals that link the modules they need. In practice, such a job generates an ASP file that imports and configures the needed modules. Here is a casual example: a job knows a SBF f and aims

to find all SBNs that contain f and play the sequence $s = (100)^*$. This job configures the necessary inference modules this way:

```
//implies the generation of a SBN n of dimension 3
sbn(n, 3).
//constrains n to contain f
:- not sbf(n, f).
//constrain n to play the sequence s
:- not sequencePlayedBySbn(n, s).
//define the bits of s and their order
sequence(s, 1, 1). // the first element of s is 1.
sequence(s, 2, 0).
sequence(s, 3, 0).
```

Remark 6. To infer ASs different only on specified predicates, jobs that call inference modules take advantage of an option of the solver: *project*, If several AS are formed by the sames atoms belonging to a list of literals, this option will force the program to keep only one of those AS. For example, let us consider the two following AS $\texttt{AS}_1$ `: {p(a), q(b), r(c)}` and $\texttt{AS}_2$ `:{ p(a), q(b), r(d)}`. The projection on `p(X)` and `q(X)` will only provide us either $\texttt{AS}_1$ or $\texttt{AS}_2$ as they both contains the same literals for `p(X)` and `q(X)`.

A.2 SBF and SBN generation issues

When generating SBF and SBN we encountered several issues that needed particular implementation solutions:

1. How to find all unique SBFs of dimension d ?
   Within multiple sets of weights like $\{w_1, w_2, w_3\}$ to define several SBFs, their truth table output may be equal or equivalent through inputs permutation, so they belong to the same equivalence class $\mathcal{A}$. To avoid duplicates, we must use only one representative per equivalence class. The core problem is then *how can we found all those equivalence classes reliably ?*

2. How to enumerate only unique SBNs from those SBFs ?
   (a) Given a set of d SBFs, redundant SBNs can be enumerated by way of permutation of the SBFs over the nodes, *e.g.* f_1 associated to node 1 and f_2 associated to node 2 or the alternative.
   (b) There are also two others sources of variations that produce the same sets of SBNs. Both correspond to layout variations, but they are obtained in two different ways:
      – Permutation of the source node for the inputs of the SBF, *e.g.* input 1 from node 1 and input 2 from node 2 or the alternative.
      – Two SBFs can also be obtained from each other through permutation of their input weights. For example $Weights(f_1) = \langle -1, 2, 1\rangle$ is equivalent to $Weights(f_2) = \langle 2, -1, 1\rangle$ by permutation of w_1 and w_2.

As we must avoid the exploration of duplicate SBNs, which would alter the results and be very costly in computational power, we must take into account only one of these variations in our SBN enumeration. For performance issues detailed below, we choose to keep the layout variation due to input's source node permutations and neutralize the other.

Inference of SBFs Satisfiable answer sets of SBFs are generated using both an inference module and job post-processing. During these operations, we must generate only one SBF per equivalence class and neutralize the redundancy induced by input weight permutations.
Within the inference module in ASP, a SBF is addressed by its abstract form $\mathcal{A}(f)$. Thanks to the *project* option of *clingcon* (see remark 6 above) the SBFs from different layouts $\mathcal{L}_ij(f)$ but belonging to the same $\mathcal{A}_i(f)$ are regrouped in only one equivalent class $\mathcal{A}_i(f)$. Since a $\mathcal{A}(f)$ is by definition a set of constraints over the SBF weights, it is particularly easy to specify $\mathcal{A}(f)$ in ASP, particularly when using *integer linear constraints*, a new ASP improvement. We can then infer a set of SBFs classified by $\mathcal{A}(f)$ and by the sum of absolutes values of weights. Once inference of SBFs is done, Java keeps only one minimal representative in each equivalence class (see 2.1).
The generated set of SBFs solve the two issues 1 and 2b presented in A.2. By dealing with the 2b issue from the beginning, we limit the number of processed SBFs in the future jobs, saving both CPU time and RAM consumption.

Remark 7. In addition of the minimal representative, several others SBFs may be kept from an equivalent class if they have weights set to zero. Edges with $w = 0$ are considered as non-existing edges. In consequence, even if two SBFs belong to the same equivalence class, different directed graphs can be obtained when some edges are absent. As the structural complexity is based on the topology of the interaction graph, this lead to different and unique SBNs that we must also explore.

Order over SBF Using $\mathcal{A}(f)$ makes it possible to determine an order over the SBFs.
$\mathcal{A}(f) = \langle y_1, y_2, ...y_d \rangle$ is composed of a unique ordered set of numerical values (y_i), each value bounded between 0 and the number of configurations $Card(X_i)$ of X in Nai_i. Each digit of the vector $\mathcal{A}(f)$ is encoded using a different numerical base $Base_i$ in such a way this vector could be converted into a number in decimal base $\mathcal{A}(f)_{10}$. The numerical base corresponds to the number of configurations $Card(X_i)$ of X in Nai_i, increased by 1 to include 0 ($y_i \in [0; Card(X_i)]$) : $Base_i = Card(X_i) + 1$.
In $\mathcal{A}(f) = \langle y_1, ..., y_{d-1}, y_d \rangle$, y_d code for the units, y_{d-1} for the decades, y_{d-2} for the hundreds, *etc*, so it matches with the conventional order in which number are read. To convert $\mathcal{A}(f)$ into a $\mathcal{A}(f)_{10}$ we have:

$$\mathcal{A}(f)_{10} = y_d + \sum_{i=d-1}^{1} (y_i \times \prod_{j=d}^{i+1} Base_j)$$

Every different $\mathcal{A}(f)$ of the same dimension will be converted into a unique $\mathcal{A}(f)_{10}$, as illustrated with 2 SBFs in table 3, that we can use to define a order over SBFs such that :

$$f_1 \leq f_2 \text{ if } \mathcal{A}(f_1)_{10} \leq \mathcal{A}(f_2)_{10}$$

	y_1	y_2	y_3	$\mathcal{A}(f_1)_{10}$
$\mathcal{A}(f_1)$	2	2	1	
$Card(X_i)$	3	3	1	21
$Base_i$	4	4	2	

	y_1	y_2	y_3	$\mathcal{A}(f_2)_{10}$
$\mathcal{A}(f_2)$	1	2	0	
$Card(X_i)$	3	3	1	12
$Base_i$	4	4	2	

Table 3: Examples of conversion of $\mathcal{A}(f)$ into a $\mathcal{A}(f)_{10}$ with $\mathcal{A}(f_1) = \langle 2, 2, 1\rangle$ and $\mathcal{A}(f_2) = \langle 1, 2, 0\rangle$

SBN enumeration In order to avoid the generation of multiple SBN composed of the same SBFs but assigned to different nodes (issue 2a in section A.2), we use the order over SBFs and follow a simple rule to assign the SBFs to labelled nodes, the one after the other: the unassigned SBF with the lower $\mathcal{A}(f)_{10}$ is always linked to the unassigned node with the lower index, and so forth.
The last step is the construction of the network layout, *i.e.* the assignation of a source node to each SBF input. For a given dimension, the number of different layouts is fixed and equal to $d!^d$. Figure 5 gives the 4 different layouts available in dimension 2 and shows how nodes are assigned to SBF inputs.

$\mathcal{L}_1$

	x_1	x_2
f_1	n_1	n_2
f_2	n_1	n_2

$\mathcal{L}_2$

	x_1	x_2
f_1	n_1	n_2
f_2	n_2	n_1

$\mathcal{L}_3$

	x_1	x_2
f_1	n_2	n_1
f_2	n_1	n_2

$\mathcal{L}_4$

	x_1	x_2
f_1	n_2	n_1
f_2	n_2	n_1

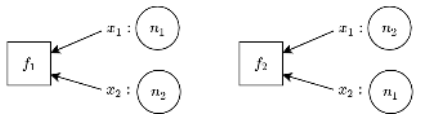


Fig. 5: List of the 4 possible layouts for 2-dimension SBNs: $\mathcal{L}_1$ to $\mathcal{L}_4$. There is a directed edge going from the node n_i to the node associated to the SBF f_i when an input x_i of f_i reads the value of n_i. The two diagrams below illustrate the input assignments of SBFs f_1 and f_2 to nodes n_1 and n_2 according to layout $\mathcal{L}_2$.

The generation of layouts is independent to both the assignment of the SBFs f_i

to the nodes n_i, and that of the nodes n_i to the inputs x_i of SBFs. Consequently we only need to generate these layouts once before SBF-node assignment. This lead to a huge saving in CPU time and RAM consumption.
Once all layouts are obtained, we combine every composition SBFs/nodes to all layouts to finally generate all possible SBNs. In the end we obtain the set of all unique SBN such that they are built with minimal weights and that there is no other SBN figuring the same set of SBFs with the same layout.

Remark 8. As zero-weighted edges are considered absent, we may obtain networks figuring disconnected nodes. Those "networks" are not considered as functional networks, so they are discarded.

B Inference modules in ASP

The following sections contain the ASP code of the modules listed in A.1.

B.1 SBN

```
%*
 *  Constructs all the possibilities for the network sbn(Name, Dimension)
 *  At least one predicate of this from must exist.
 *%

% means that constraints operated by clingcon 2017 are involved.
#include <csp>.

%% LIMIT CONSTRAINTS
maxDim(4).
possibleDim(1..MaxDim) :- maxDim(MaxDim).
possibleIdx(1..MaxDim) :- maxDim(MaxDim).

:-
    not sbn(_, _)
.


%% NETWORK GENERATION

% Generation of possible indices for a network based on the dimension
possibleNetworkIndex(NetworkName, Idx) :-
    sbn(NetworkName, Dimension)
  , possibleDim(Dimension)
  , Idx > 0
  , Idx <= Dimension
  , possibleIdx(Idx)
```

```
.


% Generation of network nodes
node(NetworkName, Idx) :-
    sbn(NetworkName, Dimension)
  , possibleNetworkIndex(NetworkName, Idx)
.


% Generation of node inputs
{
    nodeInput(NetworkName, NodeIdx, InputIdx, SrcNodeIdx) :
        node(NetworkName, NodeIdx)
      , node(NetworkName, SrcNodeIdx)
      , possibleNetworkIndex(NetworkName, InputIdx)
}.

% A node only has a single input coming from any other given node
:-
    nodeInput(NetworkName, NodeIdx, InputIdx1, SrcNodeIdx)
  , nodeInput(NetworkName, NodeIdx, InputIdx2, SrcNodeIdx)
  , InputIdx1 != InputIdx2
.

% A node receives any given input from a single other node
% (an input arc cannot originate at more than one source node)
:-
    nodeInput(NetworkName, NodeIdx, InputIdx, SrcNodeIdx1)
  , nodeInput(NetworkName, NodeIdx, InputIdx, SrcNodeIdx2)
  , SrcNodeIdx1 != SrcNodeIdx2
.

% For every node, an input coming from every other node in the network must exist.
% Therefore, Dimension^2 nodeInputs must exist for each network.
:-
    #count{ NodeIdx, InputIdx, SrcNodeIdx :
        nodeInput(NetworkName, NodeIdx, InputIdx, SrcNodeIdx)
    } != Dimension**2
  , sbn(NetworkName, Dimension)
.


% Generation of weights for every input of every node
&dom{-Dimension..Dimension} = weight(NetworkName, NodeIdx, InputIdx) :-
```

```
node(NetworkName, NodeIdx)
, possibleNetworkIndex(NetworkName, InputIdx)
  , sbn(NetworkName, Dimension)
.


% Enumeration of the inequalities of the functions implemented by the nodes
{
ineq(NetworkName, I, Input1) :
      node(NetworkName, I)
, possibleNetworkIndex(NetworkName, Input1)

;ineq(NetworkName, I, Input1, Input2) :
      node(NetworkName, I)
    , possibleNetworkIndex(NetworkName, Input1)
    , possibleNetworkIndex(NetworkName, Input2)
, Input1 < Input2

;ineq(NetworkName, I, Input1, Input2, Input3) :
      node(NetworkName, I)
    , possibleNetworkIndex(NetworkName, Input1)
    , possibleNetworkIndex(NetworkName, Input2)
    , possibleNetworkIndex(NetworkName, Input3)
, Input1 < Input2
, Input2 < Input3

;ineq(NetworkName, I, Input1, Input2, Input3, Input4) :
      node(NetworkName, I)
    , possibleNetworkIndex(NetworkName, Input1)
    , possibleNetworkIndex(NetworkName, Input2)
    , possibleNetworkIndex(NetworkName, Input3)
    , possibleNetworkIndex(NetworkName, Input4)
, Input1 < Input2
, Input2 < Input3
, Input3 < Input4
}.


% Constraints between the weights and the inequalities
&sum{weight(NetworkName, NodeIdx, Input1)} > 0 :-
   ineq(NetworkName, NodeIdx, Input1)
.
&sum{weight(NetworkName, NodeIdx, Input1)} <= 0 :-
    not ineq(NetworkName, NodeIdx, Input1)
  , node(NetworkName, NodeIdx)
```

```
  , possibleNetworkIndex(NetworkName, Input1)
.

&sum{weight(NetworkName, NodeIdx, Input1); weight(NetworkName, NodeIdx, Input2)} > 0 :-
    ineq(NetworkName, NodeIdx, Input1, Input2)
.
&sum{weight(NetworkName, NodeIdx, Input1); weight(NetworkName, NodeIdx, Input2)} <= 0 :-
    not ineq(NetworkName, NodeIdx, Input1, Input2)
  , node(NetworkName, NodeIdx)
  , possibleNetworkIndex(NetworkName, Input1)
  , possibleNetworkIndex(NetworkName, Input2)
  , Input1 < Input2
.

&sum{weight(NetworkName, NodeIdx, Input1); weight(NetworkName, NodeIdx, Input2); weight(NetworkName, NodeIdx,
    ineq(NetworkName, NodeIdx, Input1, Input2, Input3)                                                  Input3)} > 0 :-
.
&sum{weight(NetworkName, NodeIdx, Input1); weight(NetworkName, NodeIdx, Input2); weight(NetworkName, NodeIdx,
    not ineq(NetworkName, NodeIdx, Input1, Input2, Input3)                                             Input3)} <= 0 :-
  , node(NetworkName, NodeIdx)
  , possibleNetworkIndex(NetworkName, Input1)
  , possibleNetworkIndex(NetworkName, Input2)
  , possibleNetworkIndex(NetworkName, Input3)
  , Input1 < Input2
  , Input2 < Input3
.

&sum{weight(NetworkName, NodeIdx, Input1); weight(NetworkName, NodeIdx, Input2); weight(NetworkName, NodeIdx,
                                                    Input3); weight(NetworkName, NodeIdx, Input4)} > 0 :-
    ineq(NetworkName, NodeIdx, Input1, Input2, Input3, Input4)
.
&sum{weight(NetworkName, NodeIdx, Input1); weight(NetworkName, NodeIdx, Input2); weight(NetworkName, NodeIdx,
                                                  Input3); weight(NetworkName, NodeIdx, Input4)} <= 0 :-


    not ineq(NetworkName, NodeIdx, Input1, Input2, Input3, Input4)
  , node(NetworkName, NodeIdx)
  , possibleNetworkIndex(NetworkName, Input1)
  , possibleNetworkIndex(NetworkName, Input2)
  , possibleNetworkIndex(NetworkName, Input3)
  , possibleNetworkIndex(NetworkName, Input4)
  , Input1 < Input2
  , Input2 < Input3
  , Input3 < Input4
.


% Determines the components of the abstract representations of the functions
% implemented by the nodes.
```

```
% sbf(NetworkName, NodeIdx, IneqInputs, IneqCount)
sbf(NetworkName, NodeIdx, 1, X) :-
  #count{
Input1 :
ineq(NetworkName, NodeIdx, Input1)
} = X
  , node(NetworkName, NodeIdx)
  , sbn(NetworkName, Dimension)
  , Dimension >= 1
.
sbf(NetworkName, NodeIdx, 2, X) :-
  #count{
Input1, Input2 :
ineq(NetworkName, NodeIdx, Input1, Input2)
} = X
  , node(NetworkName, NodeIdx)
  , sbn(NetworkName, Dimension)
  , Dimension >= 2
.
sbf(NetworkName, NodeIdx, 3, X) :-
  #count{
Input1, Input2, Input3 :
ineq(NetworkName, NodeIdx, Input1, Input2, Input3)
} = X
  , node(NetworkName, NodeIdx)
  , sbn(NetworkName, Dimension)
  , Dimension >= 3
.
sbf(NetworkName, NodeIdx, 4, X) :-
#count{
Input1, Input2, Input3, Input4 :
ineq(NetworkName, NodeIdx, Input1, Input2, Input3, Input4)
} = X
  , node(NetworkName, NodeIdx)
  , sbn(NetworkName, Dimension)
  , Dimension >= 4
.

% There must exist as many SBFs as there are nodes.
:-
    #count{ NodeIdx, IneqInputs :
          sbf(NetworkName, NodeIdx, IneqInputs, _)
    } != Dimension**2
  , sbn(NetworkName, Dimension)
.
```

```
% Compute the factorial values needed for the SBF/Node assignation
factorial(0, 1).
factorial(1, 1).
maxFactorial(MaxFactorial) :- maxDim(MaxFactorial).
factorial(X, Value1) :-
    Value1 = X * Value2
  , factorial(X-1, Value2)
  , maxFactorial(Max)
  , X <= Max
.

% Compute the maximum number of configuration in each ineqInputs category (Nai)
maxIneq(NetworkName, IneqInputs, MaxValue) :-
    sbn(NetworkName, Dimension)
  , sbf(NetworkName, _, IneqInputs, _)
  , factorial(Dimension, FactDim)
  , factorial(IneqInputs, FactIneqInputs)
  , factorial(Dimension - IneqInputs, FactDimMinusIneqInputs)
  , MaxValue = (FactDim/(FactIneqInputs * FactDimMinusIneqInputs))
.

% compute the numerical base of each category if IneqInputs (Nai)
weightNbIneq(NetworkName, IneqInputs, Weight) :-
    maxIneq(NetworkName, IneqInputs, MaxValue)
  , sbn(NetworkName, Dimension)
  , IneqInputs = Dimension
  , Weight = (MaxValue+1)
.
weightNbIneq(NetworkName, IneqInputs, Weight) :-
    maxIneq(NetworkName, IneqInputs, MaxValue)
  , weightNbIneq(NetworkName, IneqInputs+1, WeightPrec)
  , Weight = (MaxValue+1) * WeightPrec
.

% compute the SBF decimal "value"
nodeFunctionValue(NetworkName, NodeIdx, FunctionValue) :-
    #sum{ NbIneq * Weight :
        sbf(NetworkName, NodeIdx, IneqInputs, NbIneq)
      , weightNbIneq(NetworkName, IneqInputs, Weight)

    } = FunctionValue
  , node(NetworkName, NodeIdx)
.
```

```
% ensure that the attribution of the SBFs over the nodes follow the SBF order and the node index order.
:-
    nodeFunctionValue(NetworkName, NodeIdx1, FunctionValue1)
  , nodeFunctionValue(NetworkName, NodeIdx2, FunctionValue2)
  , NodeIdx1 < NodeIdx2
  , FunctionValue1 > FunctionValue2
  , orderedNodeFunction(NetworkName)
.
```

B.2 SBN Composition

```
%%% Constraint a network to be composed by a other one.
%%% composedSbn(Sbn1, Sbn2) : Sbn1 is composed of Sbn2
%%% Can be use only once per call in this actual form


% means that constraints operated by clingcon 2017 are involved.
#include <csp>.

% force the presence of this predicates to use the module
:-
  not composedSbn(_, _)
.

% there can be only one use of this module per call
:-
    not #count{ Sbn1, Sbn2 : composedSbn(Sbn1,Sbn2) } = 1
.

% the given SBNs must exists
:-
    composedSbn(Sbn1, Sbn2)
  , not sbn(Sbn1, _)
.
:-
    composedSbn(Sbn1, Sbn2)
  , not sbn(Sbn2, _)
.

% The dimensions of the given SBNs must be compatible
:-
    composedSbn(Sbn1, Sbn2)
  , sbn(Sbn1, Dimension1)
  , sbn(Sbn2, Dimension2)
  , not Dimension1 >= Dimension2
.
```

```
% creation of a subnetwork called "extrusion" include in Sbn1
sbn(extrusion, Dimension) :-
    composedSbn(Sbn1, Sbn2)
  , sbn(Sbn2, Dimension)
.

% constraint the extrusion weights to be the same as Sbn1
:-
    sbn(extrusion, Dimension)
  , &sum{weight(Sbn1, NodeIdx, InputIdx)} = W1
  , &sum{weight(extrusion, NodeIdx, InputIdx)} = W2
  , not W1 = W2
  , limitWeights(Sbn1, W1)
  , limitWeights(Sbn1, W2)
  , possibleNetworkIndex(extrusion, NodeIdx)
  , possibleNetworkIndex(extrusion, InputIdx)
.

% constraint the structure of the extrusion to be the same as Sbn1 and Sbn2
:-
  sbn(extrusion, Dimension)
  , composedSbn(Sbn1, Sbn2)
  , sbn(Sbn2, Dimension)
  , nodeInput(extrusion, NodeIdx, InputIdx, SrcNodeIdx1)
  , nodeInput(Sbn2, NodeIdx, InputIdx, SrcNodeIdx2)
  , not SrcNodeIdx1 = SrcNodeIdx2
.

% constraint the SBFs of the extrusion to be the same as Sbn2
:-
    sbn(extrusion, Dimension)
  , composedSbn(Sbn1, Sbn2)
  , sbn(Sbn2, Dimension)
  , sbf(extrusion, NodeIdx, NbIneqInputs, IneqCount1)
  , sbf(Sbn2, NodeIdx, NbIneqInputs, IneqCount2)
  , not IneqCount1 = IneqCount2
.

% specify the weight limit for a SBN according to his dimension
limitWeights(NetworkName, -Dimension..Dimension) :-
    sbn(NetworkName, Dimension)
.
```

B.3 Sequence played by SBN

```
%%% Check the network behavior to match a given music (sequence) on a given node.

% means that constraints operated by clingcon 2017 are involved.
#include <csp>.

% forbid the use of the module without the presence of this predicate
:-
    not musicPlayBySbn(_, _, _)
.

% the given network and music must exists
:-
    musicPlayBySbn(_, MusicName, _)
  , not music(MusicName, _, _)
.
:-
    musicPlayBySbn(NetworkName, _, _)
  , not sbn(NetworkName, _)
.

% the given node must exist in the network
:-
    musicPlayBySbn(NetworkName, _, NodeIdx)
  , sbn(NetworkName, Dimension)
  , not possibleNetworkIndex(NetworkName, NodeIdx)
.

% the music length must fit with the network dimension
:-
  musicPlayBySbn(NetworkName, MusicName, _)
  , sbn(NetworkName, Dimension)
  , musicSize(MusicName, MusicSize)
  , not MusicSize <= Dimension**2
.

% tell the music length
musicSize(MusicName, Size) :-
music(MusicName, _, _)
, Size = {music(MusicName, _, _)}
.

% give the possible index for the note of the music
musicStep(MusicName, StepIdx+1) :-
musicSize(MusicName, StepIdx)
```

```
.
musicStep(MusicName, StepIdx) :-
StepIdx > 0
, musicStep(MusicName, StepIdx+1)
.

% give the State of node NodeIdx at the step StepIdx of a music MusicName : nodeState(NodeIdx, MusicName, StepIdx,
                                                                                                                State
% initialise the first state                                                                                    )
1{nodeState(NetworkName, NodeIdx, MusicName, 1, 0);nodeState(NetworkName, NodeIdx, MusicName, 1, 1)}1 :-
possibleNetworkIndex(NetworkName, NodeIdx)
, musicStep(MusicName, 1)
.

% transfom the enumerated weight into predicates
limitWeights(NetworkName, -Dimension..Dimension) :-
    sbn(NetworkName, Dimension)
.
inputWeightForStep(NetworkName, NodeIdx, MusicName, InputIdx, StepIdx, Value) :-
State = 1
, &sum{weight(NetworkName, NodeIdx, InputIdx)} = Value
, limitWeights(NetworkName, Value)
, nodeInput(NetworkName, NodeIdx, InputIdx, SrcNodeIdx)
, nodeState(NetworkName, SrcNodeIdx, MusicName, StepIdx-1, State)
.

inputWeightForStep(NetworkName, NodeIdx, MusicName, InputIdx, StepIdx, 0) :-
State = 0
, nodeInput(NetworkName, NodeIdx, InputIdx, SrcNodeIdx)
, nodeState(NetworkName, SrcNodeIdx, MusicName, StepIdx-1, State)
.

% specify the state of a node at a given step
nodeState(NetworkName, NodeIdx, MusicName, StepIdx, 1) :-
  musicStep(MusicName, StepIdx)
, musicStep(MusicName, StepIdx-1)
, #sum{Value, InputIdx : inputWeightForStep(NetworkName, NodeIdx, MusicName, InputIdx, StepIdx, Value)} > 0
  , sbn(NetworkName, _)
  , possibleNetworkIndex(NetworkName, NodeIdx)
.

nodeState(NetworkName, NodeIdx, MusicName, StepIdx, 0) :-
  musicStep(MusicName, StepIdx)
, musicStep(MusicName, StepIdx-1)
, #sum{Value, InputIdx : inputWeightForStep(NetworkName, NodeIdx, MusicName, InputIdx, StepIdx, Value)} <= 0
```

```
  , sbn(NetworkName, _)
  , possibleNetworkIndex(NetworkName, NodeIdx)
.

% force the state of the playing node to match the music
:-
  nodeState(NetworkName, NodeIdx, MusicName, StepIdx, NodeState)
  , musicPlayBySbn(NetworkName, MusicName, NodeIdx)
  , music(MusicName, StepIdx, MusicNote)
  , musicSize(MusicName, MusicSize)
  , StepIdx <= MusicSize
  , NodeState != MusicNote
.

% specify the network state at a given step : networkState(MusicName, StepIdx, State)
possibleState(NetworkName, 0..(2**Dimension)) :-
    sbn(NetworkName, Dimension)
.
networkState(NetworkName, MusicName, StepIdx, State) :-
State = #sum{(2**(NodeIdx-1))* NodeState :
nodeState(NetworkName, NodeIdx, MusicName, StepIdx, NodeState)
}
, possibleState(NetworkName, State)
, musicStep(MusicName, StepIdx)
.

% ensure different network state for each step of the music
:-
networkState(NetworkName, MusicName, StepIdx1, State)
  , networkState(NetworkName, MusicName, StepIdx2, State)
  , musicPlayBySbn(NetworkName, MusicName, _)
  , musicSize(MusicName, MusicSize)
  , StepIdx1 <= MusicSize
  , StepIdx2 <= MusicSize
, StepIdx1 != StepIdx2
.

% ensure the path along the transition graph is a cycle
:-
networkState(NetworkName, MusicName, 1, State1)
  , networkState(NetworkName, MusicName, MusicSize+1, State2)
  , musicPlayBySbn(NetworkName, MusicName, _)
  , musicSize(MusicName, MusicSize)
, State1 != State2
.
```

B.4 Ordered version of SBN

```
% means that constraints operated by clingcon 2017 are involved.
#include <csp>.

% forbid the use of the module without this predicate
:-
  not generateOrderedFunctionsVersionOfSbn(_)
.

% the concerned network must exist
:-
    generateOrderedFunctionsVersionOfSbn(NetworkName)
  , not sbn(NetworkName, _)
.


% map the function over the node
% save the index mapping
2{orderedNodeFunctionValue(NetworkName, NewNodeIdx+1, FunctionValue); mapIndex(NetworkName,
    #count{ NodeIdx, ValueX :                                          OldNodeIdx, NewNodeIdx+1)}2 :-
        nodeFunctionValue(NetworkName, NodeIdx, ValueX)
      , ValueX < FunctionValue
      , NodeIdx != OldNodeIdx
    } = NewNodeIdx
  , possibleNetworkIndex(NetworkName, NewNodeIdx+1)
  , nodeFunctionValue(NetworkName, OldNodeIdx, FunctionValue)
  , sbn(NetworkName, _)
  , generateOrderedFunctionsVersionOfSbn(NetworkName)
  , not nodeFunctionValue(NetworkName, OldNodeIdxB, FunctionValue) :
      OldNodeIdxB < OldNodeIdx
      , possibleNetworkIndex(NetworkName, OldNodeIdxB)
.
2{orderedNodeFunctionValue(NetworkName, NewNodeIdx + Shift + 1, FunctionValue); mapIndex(NetworkName,
                                                                  OldNodeIdx, NewNodeIdx + Shift + 1)}2 :-
    #count{ NodeIdx, ValueX :
        nodeFunctionValue(NetworkName, NodeIdx, ValueX)
      , ValueX < FunctionValue
      , NodeIdx != OldNodeIdx
    } = NewNodeIdx
  , possibleNetworkIndex(NetworkName, NewNodeIdx+Shift + 1)
  , nodeFunctionValue(NetworkName, OldNodeIdx, FunctionValue)
  , #count{OldNodeIdxB :
        nodeFunctionValue(NetworkName, OldNodeIdxB, FunctionValue)
      , OldNodeIdxB < OldNodeIdx
      , possibleNetworkIndex(NetworkName, OldNodeIdxB)
```

```
    } = Shift
  , sbn(NetworkName, _)
  , generateOrderedFunctionsVersionOfSbn(NetworkName)
.

% every and each function must be remapped
:-
    #count{ OldNodeIdx, NewNodeIdx :
        mapIndex(NetworkName, OldNodeIdx, NewNodeIdx)
    } != Dimension
  , sbn(NetworkName, Dimension)
  , generateOrderedFunctionsVersionOfSbn(NetworkName)
.
:-
    #count{ NodeIdx, FunctionValue :
        orderedNodeFunctionValue(NetworkName, NodeIdx, FunctionValue)
    } != Dimension
  , sbn(NetworkName, Dimension)
  , generateOrderedFunctionsVersionOfSbn(NetworkName)
.

% SBFs remapping
orderedSbf(NetworkName, NewNodeIdx, IneqInputs, IneqCount) :-
    sbf(NetworkName, OldNodeIdx, IneqInputs, IneqCount)
  , mapIndex(NetworkName, OldNodeIdx, NewNodeIdx)
  , generateOrderedFunctionsVersionOfSbn(NetworkName)
.


% network layout remapping
orderedNodeInput(NetworkName, NewNodeIdx, InputIdx, NewSrcNodeIdx) :-
    nodeInput(NetworkName, OldNodeIdx, InputIdx, OldSrcNodeIdx)
  , mapIndex(NetworkName, OldNodeIdx, NewNodeIdx)
  , mapIndex(NetworkName, OldSrcNodeIdx, NewSrcNodeIdx)
  , generateOrderedFunctionsVersionOfSbn(NetworkName)
.


% weight generation for every input of each node
&dom{-Dimension..Dimension} = orderedWeight(NetworkName, NodeIdx, InputIdx) :-
possibleNetworkIndex(NetworkName, NodeIdx)
, possibleNetworkIndex(NetworkName, InputIdx)
  , sbn(NetworkName, Dimension)
  , generateOrderedFunctionsVersionOfSbn(NetworkName)
.
```

```
% networks weight's remapping
:-
    generateOrderedFunctionsVersionOfSbn(NetworkName)
  , mapIndex(NetworkName, OldNodeIdx, NewNodeIdx)
  , &sum{weight(NetworkName, OldNodeIdx, InputIdx)} = W1
  , &sum{orderedWeight(NetworkName, NewNodeIdx, InputIdx)} = W2
  , not W1 = W2
  , limitWeights(NetworkName, W1)
  , limitWeights(NetworkName, W2)
  , possibleNetworkIndex(NetworkName, InputIdx)
.
limitWeights(NetworkName, -Dimension..Dimension) :-
    sbn(NetworkName, Dimension)
.
```