Pattern Analysis in Binary Input Repertoires of Boolean Networks

June 13, 2025

Abstract

This document presents a mathematical framework for identifying specific binary patterns in the input repertoire of a Boolean network, where nodes perform Boolean operations based on a connectivity matrix. We introduce the method intuitively, provide a rigorous mathematical formulation using 1-based indexing, and explain the formula step-by-step for clarity. The approach leverages the periodicity of binary sequences to efficiently compute pattern locations, making it suitable for analyzing complex networks. This formulation is intended for academic use and as a reference for computational modeling with large language models.

1 Developing Intuition

Imagine a network of light switches, each of which can be either ON (1) or OFF (0). These switches are connected in a specific way, and each switch follows a rule (like "turn ON only if both inputs are ON") to decide its state. To understand how this network behaves, we need to test every possible combination of switch settings. These combinations form the *input repertoire*, a list of all possible ON/OFF patterns for the switches.

Our task is to find specific patterns in this list. For example, we might want to know which combinations have the first switch OFF and the second switch ON. Since the input repertoire can be very large (for n switches, there are 2^n combinations), checking each combination one by one is inefficient. Instead, we can use the structure of the list to find these patterns quickly.

Let's consider a small network with three switches. The input repertoire lists all $2^3 = 8$ combinations, from (0, 0, 0) to (1, 1, 1), in the order of their binary representation (like counting from 0 to 7 in binary). If we look at the first switch, its state (ON or OFF) changes every single combination: OFF, ON, OFF, ON, and so on. The second switch changes more slowly, staying OFF for two combinations, then ON for two, and so forth. The third switch is even slower, staying OFF for four combinations before switching to ON.

This pattern of changing states is like a set of clocks ticking at different speeds. The first switch's "clock" ticks every combination, the second every two, and the third every four. To find combinations where, say, the first switch is OFF, we look at the "ticks" where its clock shows OFF. For the second switch to be ON, we look at its slower clock. The key is that each switch's state appears in predictable, repeating blocks. By identifying the start of these blocks and their length, we can list all positions where a switch has the desired state. To find combinations where multiple switches have specific states, we take the common positions (intersection) of their respective lists. This approach avoids checking every combination and uses the rhythm of binary counting to pinpoint the patterns efficiently.

The rest of this document formalizes this intuition, turning the idea of "clocks" and "blocks" into a precise mathematical expression that works for any number of switches and any pattern.

2 Problem Definition

Consider a Boolean network with n nodes, where the connectivity is defined by an $n \times n$ adjacency matrix C, with $C_{i,j} = 1$ if node i receives input from node j, and $C_{i,j} = 0$ otherwise. Each node i applies a Boolean function (e.g., AND, OR) specified in a dynamics list D. The input repertoire is the set of all 2^n possible binary vectors $\{b_1, b_2, \ldots, b_n\}$, where $b_i \in \{0, 1\}$, indexed from 1 to 2^n. Each index j corresponds to the binary representation of $j - 1$, with b_1 as the most significant bit.

Given a subset of nodes $\{k_1, k_2, \ldots, k_m\} \subseteq \{1, 2, \ldots, n\}$ and a desired pattern $\{p_1, p_2, \ldots, p_m\}$, where $p_i \in \{0, 1\}$, the task is to find all indices $j \in \{1, 2, \ldots, 2^n\}$ such that the k_i-th bit of the binary vector at index j equals p_i for all $i = 1, 2, \ldots, m$.

3 Input Repertoire Structure

The input repertoire is a list of 2^n binary vectors, ordered by their decimal index (1-based). For $n = 3$, the repertoire is:

Index	Binary Vector	Decimal (0-based)
1	$(0, 0, 0)$	000
2	$(1, 0, 0)$	001
3	$(0, 1, 0)$	010
4	$(1, 1, 0)$	011
5	$(0, 0, 1)$	100
6	$(1, 0, 1)$	101
7	$(0, 1, 1)$	110
8	$(1, 1, 1)$	111

The k-th bit of index j (1-based) is:

$$b_k(j) = \lfloor (j - 1)/2^{k-1} \rfloor \mod 2$$

This bit alternates with a period of 2^{k-1}, creating sequences of consecutive 0s or 1s.

4 Mathematical Formulation

To find indices j where the nodes $\{k_1, k_2, \ldots, k_m\}$ have values $\{p_1, p_2, \ldots, p_m\}$, we analyze each node independently and intersect the results.

4.1 Single Node Analysis

For node k_i, the k_i-th bit has a period of 2^{k_i-1}, meaning it remains constant for 2^{k_i-1} consecutive indices before switching. The number of such sequences is:

$$r_i = \frac{2^n}{2^{k_i-1}} = 2^{n-k_i+1}$$

Each sequence has length:

$$l_i = 2^{k_i-1}$$

The sequences where $b_{k_i} = 0$ occur at even multiples of l_i, and those where $b_{k_i} = 1$ occur at odd multiples, due to the binary counting pattern. The pivot indices (starting points of sequences) are:

$$S_i = \begin{cases} \{(2m+1) \cdot 2^{k_i-1} \mid m = 0, 1, \dots, \lfloor (2^{n-k_i+1} - 1)/2 \rfloor - 1\} & \text{if } p_i = 1 \\ \{2m \cdot 2^{k_i-1} \mid m = 0, 1, \dots, \lfloor (2^{n-k_i+1} - 2)/2 \rfloor - 1\} & \text{if } p_i = 0 \end{cases}$$

The indices for node k_i where $b_{k_i} = p_i$ are:

$$F_i = \bigcup_{s \in S_i} \{s+1, s+2, \dots, s + 2^{k_i-1}\}$$

4.2 Multi-Node Intersection

The indices where all nodes satisfy the pattern are the intersection of the individual sets:

$$L = \bigcap_{i=1}^{m} F_i$$

The final expression is:

$$L = \bigcap_{i=1}^{m} \left(\bigcup_{m=0}^{\lfloor (2^{n-k_i+1} - [p_i=1])/2 \rfloor - 1} \left[(2m + [p_i = 1]) \cdot 2^{k_i-1} + 1, (2m + [p_i = 1] + 1) \cdot 2^{k_i-1} \right] \right)$$

where $[p_i = 1] = 1$ if $p_i = 1$, 0 otherwise, and $[a, b]$ denotes the integer range $\{a, a + 1, \dots, b\}$.

5 Explaining the Formal Definition

To make the mathematical formulation clear, let's walk through how it works step-by-step for a general case, where we have n nodes and want to find indices where a subset of m nodes, labeled $\{k_1, k_2, \dots, k_m\}$, have specific values $\{p_1, p_2, \dots, p_m\}$. Think of this as a recipe for finding the right rows in the input repertoire without checking every single one.

1. **Understand Each Node's Role**: Each node corresponds to a position in the binary vectors of the input repertoire. For node k_i, we're interested in the k_i-th bit (counting from 1 as the most significant bit). This bit tells us whether node k_i is 0 or 1 in a given row. Our goal is to find rows where this bit equals p_i.

2. **Notice the Pattern of Bits**: In the input repertoire, each node's bit changes at a regular interval. For node k_i, the bit stays the same (0 or 1) for 2^{k_i-1} rows before switching. For example, if $n = 5$ and $k_i = 2$, the second bit changes every $2^{2-1} = 2$ rows. This creates blocks of consecutive 0s or 1s.

3. **Count the Blocks**: Since the repertoire has 2^n rows, we calculate how many blocks of 2^{k_i-1} rows fit:
$$\text{Number of blocks} = \frac{2^n}{2^{k_i-1}} = 2^{n-k_i+1}$$
Half of these blocks have the bit as 0, and half as 1, alternating like a slow clock.

4. **Pick the Right Blocks**: We need the blocks where the bit matches p_i. If $p_i = 0$, we want the "even" blocks (starting at indices like 0, $2 \cdot 2^{k_i-1}$, $4 \cdot 2^{k_i-1}$, etc.). If $p_i = 1$, we want the "odd" blocks (starting at $1 \cdot 2^{k_i-1}$, $3 \cdot 2^{k_i-1}$, etc.). This even/odd rule comes from how binary numbers count.

5. **List Indices in Those Blocks**: For each chosen block, we include all indices from its start to its end. If a block starts at index $s = m \cdot 2^{k_i-1}$, it includes indices $s+1, s+2, \ldots, s+2^{k_i-1}$. This gives us a list of all rows where node k_i has value p_i.

6. **Combine Across Nodes**: Repeat steps 1–5 for each of the m nodes, getting a list of indices for each. To find rows where *all* nodes match the pattern, we take the intersection of these lists. This ensures every node's condition is satisfied at the same time.

7. **Output the Result**: The final list of indices tells us exactly which rows in the input repertoire have the desired pattern for the specified nodes.

This process is efficient because it uses the regular, predictable structure of binary sequences. Instead of checking 2^n rows, we calculate the start and length of blocks directly and narrow down the candidates with intersections. The formula in Section **??** captures this logic precisely, using mathematical notation to define the blocks and their intersections for any n, m, and pattern.

6 Example

For $n = 5$, $nodes = \{1, 2, 4\}$, $wantedPatt = \{0, 0, 0\}$:

- **Node 1** ($k_1 = 1$, $p_1 = 0$):
$$l_1 = 2^{1-1} = 1, \quad r_1 = 2^{5-1+1} = 32$$
Even sequences: $S_1 = \{0 \cdot 1, 2 \cdot 1, \ldots, 30 \cdot 1\} = \{0, 2, \ldots, 30\}$.
$$F_1 = \{1, 3, \ldots, 31\}$$

- **Node 2** ($k_2 = 2$, $p_2 = 0$):
$$l_2 = 2^{2-1} = 2, \quad r_2 = 2^{5-2+1} = 16$$
Even sequences: $S_2 = \{0 \cdot 2, 2 \cdot 2, \ldots, 14 \cdot 2\} = \{0, 4, \ldots, 28\}$.
$$F_2 = \{1, 2, 5, 6, \ldots, 29, 30\}$$

- **Node 4** ($k_3 = 4$, $p_3 = 0$):

$$l_3 = 2^{4-1} = 8, \quad r_3 = 2^{5-4+1} = 4$$

  Even sequences: $S_3 = \{0 \cdot 8, 2 \cdot 8\} = \{0, 16\}$.

$$F_3 = \{1, \ldots, 8, 17, \ldots, 24\}$$

- **Intersection**:

$$L = F_1 \cap F_2 \cap F_3 = \{1, 5, 17, 21\}$$

Check: Rows 1, 5, 17, 21 (0-based indices 0, 4, 16, 20) are:

- {0, 0, 0, 0, 0}
- {0, 0, 1, 0, 0}
- {0, 1, 0, 0, 0}
- {0, 1, 0, 1, 0}

Nodes 1, 2, 4 yield $\{0, 0, 0\}, \{0, 0, 0\}, \{0, 1, 0\}, \{0, 1, 1\}$, indicating only {1, 5} are correct, suggesting a potential issue in the original implementation's output.

7 Discussion

The formulation, rooted in the intuitive idea of periodic "clocks," leverages the structure of binary sequences to achieve efficiency. Each bit alternates at a power-of-2 interval, with 0s at even multiples and 1s at odd multiples. The method is computationally efficient, avoiding brute-force enumeration. However, the example suggests the implementation may include indices (e.g., 17, 21) that do not match the pattern, possibly due to specific helper functions (e.g., `evening`, `odding`).

8 Conclusion

We have presented a mathematical framework for finding binary patterns in the input repertoire of a Boolean network, with an intuitive explanation, formal expression, and step-by-step breakdown. The method is generalizable and efficient, though care must be taken with indexing and pivot definitions. Future work could extend this to output repertoires or clarify implementation details.