MASARYK
UNIVERSITY

FACULTY OF INFORMATICS

Inference of Boolean Networks

Bachelor's Thesis

ANDREJ ŠIMURKA

Brno, Fall 2022

MASARYK UNIVERSITY

FACULTY OF INFORMATICS

Inference of Boolean Networks

Bachelor's Thesis

ANDREJ ŠIMURKA

Advisor: doc. RNDr. David Šafránek, Ph.D.

Department of Machine Learning and Data Processing

Brno, Fall 2022


FACULTAS ARTIS INFORMATICÆ
UNIVERSITAS MASARYKIANA

Declaration

Hereby I declare that this paper is my original authorial work, which I have worked out on my own. All sources, references, and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Andrej Šimurka

Advisor: doc. RNDr. David Šafránek, Ph.D.

Acknowledgements

First and foremost, I would like to thank my advisor, doc. RNDr. David Šafránek, Ph.D., for his guidance and advice and for providing valuable feedback during my writing. I want to express my gratitude to my family for their unconditional support. I am also very thankful to my friend and colleague, Oto Stanko, with whom we exchanged knowledge throughout the research.

Abstract

Most of recent automatic Boolean network inference methods have been using synchronous update semantics. However, the latest sequencing technologies allow us to obtain expression levels of single cells and discover the heterogeneity of cell populations, which is connected with the cell differentiation process. In addition, it may be desirable to require variety of complex properties from the derived network. However, synchronous semantics is insufficient to capture such advanced behaviour and verify some non-trivial properties of Boolean networks because it only generates deterministic paths of state transitions. Consequently, the use of asynchronous semantics becomes necessary.

In this thesis, we formulate the idea of creating a complex inference framework that uses multiple types of single-cell expression data and is able to reflect a wide range of static and dynamic properties stated about the underlying network. In connection with this idea, we provide a prototype of automatic Boolean networks inference algorithm from single-cell steady-state data that employs the asynchronous semantics and is able to reflect a subset of the listed properties stated about the underlying network. As a part of the provided algorithm, we propose a novel fitness-evaluating metric that determines the level of similarity of two Boolean networks identified by sets of their steady-states obtained from multiple experiments.

Subsequently, we implement the proposed algorithm and evaluate it on several models from *DREAM-3 In silico challenge* dataset, random scale-free Boolean networks and literature-based models and achieved satisfactory results that motivate us to continue its development. Finally, we identify the shortcomings of the current implementation and propose further steps for improvement.

Keywords

Boolean networks, inference, expression data, algorithm, asynchronous semantics

Contents

Introduction

The main goal of computational systems biology is to understand biological processes at the systems level, i.e., to characterise the system's components (genes, proteins...) and relationships between them (regulations) [1]. Since biological systems are dynamic systems, in addition to structural relationships, it is also necessary to describe the dynamics of interactions between components. Proteins often play the role of transcription factors that retroactively regulate the expression of genes. Such interaction networks that control the expression of genes are commonly called gene regulatory networks (GRN).

The problem of inference of gene regulatory networks is a problem of identifying the unknown target GRN from various experimental data and prior knowledge and is one of the fundamental problems of systems biology. Recently, many automatic methods for solving this problem have been proposed [2, 3, 4, 5].

The Boolean Network (BN) model proved to be one of the most suitable computational models for representing GRNs, mainly for its computational simplicity and efficiency [6]. That is due to the absence of any quantitative reaction kinetic parameters, which are practically hard to estimate.

However, most existing BN inference algorithms consider only synchronous update semantics, which proved to be insufficient to capture the overall dynamics of networks as it depicts only deterministic paths in a state transition system. Therefore, any system behaviour that contains differentiation points cannot be captured, which is an extreme limitation.

Moreover, modern sequencing technologies, such as single-cell RNA-seq, allow us to collect expression data at the individual cell level. Thus, we are able to capture more detailed behaviour of biological systems. In addition, it may be wanted to consider additional knowledge about systems from related studies or include hypotheses, both in the form of additional restrictions to the inferred BN. That creates demand for more complex inference algorithms that can reflect a wide range of data types and be able to enforce some properties of the inferred network. One of the possible solutions is to extend the problem of Boolean network inference by considering asynchronous

semantics that can capture the full dynamics of biological systems and, based on this, verify the more complex properties of networks.

To the best of our knowledge, recent automatic BN inference tools usually concentrate on the inference of GRNs only from a single type of experimental data (time-series or steady-state) and are not able to verify multiple types of properties of the inferred network. Therefore, our global goal is to create a complex BN inference framework that will be able to reflect as many different experimental data types as possible. In addition, we want it to be able to accept and apply a large set of requirements to the inferred network. This thesis aims to design and implement a part of the overall framework that will receive single-cell steady-state data and reflect a subset of our desired requirements for the inferred network.

The thesis consists of seven chapters. Chapter 1 defines some frequently used terms and concepts related to Boolean networks. Chapter 2 summarises the most common types of experiments and the data storing records of their progress or results. Chapter 3 leads to the definition of the problem of inference of Boolean networks in the context of our thesis. Chapter 4 focuses on the design and description of a prototype of an asynchronous-based automatic inference framework that considers some prior knowledge about the target Boolean network. Though the framework adopts a classical genetic programming structure used in other automatic inference methods [2], we propose a novel fitness evaluation metric based on a minimal weighted assignment problem on a complete bipartite graph. The metric determines a similarity between two Boolean networks defined by sets of their single-cell steady-states obtained from multiple experiments. Chapter 5 comprises the implementation details and a list of used third-party libraries. Chapter 6 is devoted to the evaluation of the proposed framework. Finally, in Chapter 7, we discuss the limitations of the current framework version and suggest future improvements.

1 Overview of Boolean networks

Boolean Network models (first introduced in [7]) provide a simple qualitative representation of gene regulatory networks. Despite its simplicity, this model can capture a broad range of networks and dynamic behaviours. Boolean networks contain only Boolean variables and have no kinetic parameters, which makes them computationally more efficient than detailed quantitative models [8].

It is proved that discretisation of gene expression levels need not result in significant loss of information and may conform well to actual cell behaviour. The ability to reduce the dynamic properties of Gene Regulatory Networks proves to be crucial in terms of conceptual simplicity and transparency [6].

In this thesis, the following notation is used. We use $\mathbb{B}$ to denote Boolean domain $\{0,1\}$ and accordingly $\mathbb{B}^n$ to denote a set of Boolean vectors of n-th dimension. In addition, we introduce the notation $s[i \to b] = (s_1, \ldots, s_{i-1}, b, s_{i+1}, \ldots, s_n)$ for Boolean vector $s \in \mathbb{B}^n$. We also use *Hamming distance* of two Boolean vectors $b, b' \in \mathbb{B}^n$, which is defined as $\sum_{i=1}^{n} \|b_i - b'_i\|$.

1.1 Nested Canalyzing Functions

Consider a Boolean function $f : \mathbb{B}^n \to \mathbb{B}$ with inputs $x_1, \ldots, x_n$. Let π be a permutation of n. Function f is considered as *nested canalyzing function* (NCF) in the variable tuple $(x_{\pi(1)}, \ldots, x_{\pi(n)})$ having *canalyzing values* $(i_1, \ldots, i_n) \in \mathbb{B}^n$ and *canalyzed values* $(o_1, \ldots, o_n) \in \mathbb{B}^n$ if it can be defined in the following way:

$$f(x_1, \ldots, x_n) = \begin{cases} o_1 & \text{if } x_{\pi(1)} = i_1, \\ o_2 & \text{if } x_{\pi(1)} \neq i_1 \wedge x_{\pi(2)} = i_2, \\ \vdots & \vdots \\ o_n & \text{if } x_{\pi(1)} \neq i_1 \wedge \ldots x_{\pi(n-1)} \neq i_{n-1} \wedge x_{\pi(n)} = i_n, \\ \overline{o_n} & \text{if } x_{\pi(1)} \neq i_1 \wedge \ldots x_{\pi(n)} \neq i_n. \end{cases}$$

1.2 Boolean network definition

We define a Boolean Network of n-th dimension as a tuple $\mathcal{B}^{(n)} = (\mathcal{V}, \mathcal{F})$. Note that for the sake of simplicity, the dimension is often omitted. Components of the $\mathcal{B}^{(n)}$ are defined as follows:

$\mathcal{V} = \{v_1, \ldots, v_n\}$ is a finite set of n Boolean variables. Although variables can represent different biological substances like genes, mRNAs, signal proteins, or non-physical elements, such as phenotypes [9], they are often referred to simply as genes. Value of $v_i \in \mathbb{B}$ indicates whether the component represented by it is significantly expressed / present (1) or not (0).

The *state space* of the Boolean network $\mathcal{B}^{(n)}$, denoted $\mathcal{S}(\mathcal{B}^{(n)}) = \mathbb{B}^n$, is a set of all possible configurations of values of Boolean variables $\mathcal{V}$. The *state* of Boolean Network $s \in \mathcal{S}(\mathcal{B})$ denotes one specific configuration of values of variables $\mathcal{V}$. State s_0 is called the *initial state* and represents the state of the Boolean network $\mathcal{B}$, from which it is executed [4].

$\mathcal{F} = \{f_1, \ldots, f_n\}$ is a finite set of n Boolean functions (each corresponds to one Boolean variable), such that $f_i : \mathbb{B}^n \to \mathbb{B}$. The Boolean function f_i defines the dependence of the value of v_i formally on values of all variables $\mathcal{V}$. However, individual variable v_i is typically observably influenced only by a subset of variables $\mathcal{V}$. Such subset of variable indices is called *regulators* of v_i (denoted $reg(v_i)$). Formally:

$$j \in reg(v_i) \iff \exists s \in \mathcal{S}(\mathcal{B}) \,.\, f_i(s[j \to 0]) \neq f_i(s[j \to 1])$$

Let $\Pi(reg(v_i))$ be the set of all possible permutations of $reg(v_i)$ and π_i a single permutation, such that $\pi_i \in \Pi(reg(v_i))$.

We restrict ourselves to such functions $f_i \in \mathcal{F}$ that can be represented by equivalent functions $f_i' : \mathbb{B}^{\|reg(v_i)\|} \to \mathbb{B}$ that are NCFs in π_i, more precisely the following formula has to be fulfilled.

$$\forall f_i \in \mathcal{F}\ \exists f_i' \,.\, f_i = f_i' \wedge \exists \pi_i \in \Pi(reg(v_i)) \,.\, f_i' \text{ is NCF in } \pi_i$$

Although NCFs cannot represent all possible Boolean functions, several studies [10, 11, 12] suggest that they are sufficient to represent the most known biological regulatory dependencies. Moreover, they are easily mutable and representable.

We specially denote variable v_i that is independent on all variables (formally $reg(v_i) = \emptyset$) as *input variable*. On the contrary, a variable v_o that does not observably regulates any variable (formally $\forall v_j \in \mathcal{V} \,.\, o \notin reg(v_j)$) is denoted as *output variable*.

1.3 Boolean network dynamics

Boolean network semantics interprets the dynamics of a Boolean network model. Specifically, semantics are defined as directed *state transition graphs* (STG) between states. There are two semantics commonly used in connection with Boolean networks. Both of them describe different types of network behaviour. Therefore, their STGs use to differ significantly.

In *Synchronous semantics*, the next state is computed by updating all variables $\mathcal{V}$ synchronously. Therefore, the resulting STG is entirely deterministic [2].
Formally $\text{STG}_{Syn}(\mathcal{B}^{(n)}) = (S, T)$ where $S = \mathcal{S}(\mathcal{B}^{(n)})$ is network's state space, and the transition relation between states $T \subseteq S \times S$ is defined as:

$$(s,t) \in T \iff t = (f_1(s), \ldots, f_n(s))$$

In *Asynchronous semantics*, individual variables are updated independently. Therefore, the resulting STG is not deterministic. One state can have up to n successors (also considering self-loops).
Formally $\text{STG}_{Asyn}(\mathcal{B}^{(n)}) = (S, T)$ where $S = \mathcal{S}(\mathcal{B}^{(n)})$, and the transition relation between states $T \subseteq S \times S$ is defined as:

$$(s,t) \in T \iff \exists i \in \{1, \ldots, n\} \,.\, t = (s[i \to f_i(s)])$$

The notation $\text{STG}(\mathcal{B})$ later in the text refers to either kind of STG of Boolean network $\mathcal{B}$, defined above.

1.4 Boolean network attractors

Since the size of the state space of a Boolean network $\mathcal{B}^{(n)}$ is exactly 2^n, after a large enough number of transitions performed, the network must necessarily return to a previously visited state. This fact

implies the existence of repeating transition sequences in the BN state transition system [13].

The set of attractors of $\mathcal{B}$, denoted as $\mathcal{A}_{STG(\mathcal{B})} = \{a_1, \ldots, a_k\}$, defines its long-term behaviour options and is dependent on used semantics [14]. The set of attractors corresponds to the selected STG's terminal (bottom) strongly connected components (TSCC).

Formally, when considering STG($\mathcal{B}$) = (S, T) where T^* denotes reflexive and transitive closure of T, the attractor $a_i \in \mathcal{A}_{STG(\mathcal{B})}$ is defined as:

$$\forall s_1, s_2 \in a_i \,.\, (s_1, s_2) \in T^*$$
$$\forall s_1 \in a_i \,.\, (s_1, s_2) \in T^* \implies s_2 \in a_i$$

According to multiple studies [14, 15, 16], four long-term behaviours of BN can be recognised, namely:

- *Steady-state* consists of exactly one state and indicates the system's stability. Steady-states are represented as self-loops in the STG. Therefore, further transitions will not result in any change of state. They are often referred to as *sinks*.
- *Simple oscillation* is a cyclic attractor which consists of two or more states. Each state has exactly one successor in the STG, and any two adjacent states have Hamming distance of one.
- *Complex oscillation* is similar to Simple oscillation. However, at least one tuple of adjacent states has Hamming distance of more than one.
- *Disorder* includes multiple interlinked simple oscillations and self-loops. Each state can have more than one successor in the STG, and there is only Hamming distance of one between any two adjacent states. The system tends to behave unpredictably.

Steady-states and *Simple oscillations* preserve Hamming distance between adjacent states less or equal to one. Therefore, they can be present in attractor sets of both semantics. Nevertheless, the *Complex oscillations* logically appear only in the attractor set of the synchronous semantics as they may have Hamming distance greater than one between some adjacent states. On the contrary, *Disorders* only exist in asynchronous semantics because their states may have more than one successor [14, 16].

1.5 Paths in Boolean network dynamics

We formally define set of all possible *paths* of a given STG($\mathcal{B}$) = (S, T), denoted $\mathcal{P}_{STG(\mathcal{B})}$, as:

$$(s_1, ..., s_m) \in \mathcal{P}_{STG(\mathcal{B})} \iff \forall i \in \{1, ..., m-1\} \,.\, (s_i, s_{i+1}) \in T$$

Terminating path is a path where the last state s_m belongs to some attractor. Formally, we define the set of all terminating paths of a given STG($\mathcal{B}$) = (S, T), denoted $\mathcal{P}^t_{STG(\mathcal{B})}$, as:

$$p = (s_1, ..., s_m) \in \mathcal{P}^t_{STG(\mathcal{B})} \iff p \in \mathcal{P}_{STG(\mathcal{B})} \wedge \exists a \in \mathcal{A}_{STG(\mathcal{B})} \,.\, s_m \in a$$

1.6 Basins of attraction

The *Strong basin* of the specific attractor $a \in \mathcal{A}_{STG(\mathcal{B})}$, denoted $Sb(a)$, is a set containing states that are either state of attractor a or all terminating paths outgoing from such states end only by a state of the attractor a. Formally:

$$s \in Sb(a) \iff \forall p \in \mathcal{P}^t_{STG(\mathcal{B})} \,.\, p = (s, \ldots, s_m) \implies s_m \in a$$

The *Weak basin* of the specific attractor $a \in \mathcal{A}_{STG(\mathcal{B})}$, denoted $Wb(a)$, is a set containing states that are either state of attractor a or, there exists terminating path outgoing from such state that leads to the attractor a. Formally:

$$s \in Wb(a) \iff \exists p \in \mathcal{P}^t_{STG(\mathcal{B})} \,.\, p = (s, \ldots, s_m) \wedge s_m \in a$$

Since the synchronous semantics is deterministic, each state belongs to exactly one basin when it is used. Thus, in synchronous semantics, the strong and weak basins of each attractor are equal. The difference between strong and weak basins appears only in asynchronous semantics. The non-determinacy of the asynchronous semantics directly implies the fact that it is possible from one particular state to reach multiple different attractors. This possibility corresponds to the situation where such a state is a part of the weak basins of two distinct attractors.

The size of a basin of attraction is biologically relevant. Attractors with large basins often represent the natural system's behaviour, while small basins may indicate unexpected, potentially critical behaviour [7].

2 Experimental data

The fundamental input of all recent inference approaches is experimental data. When referring to the inference of GRN, we are interested in so-called *expression data*. Such data is obtained by periodic sampling of gene expression levels of the infering GRN, aiming to capture their evolution. These expression level measurements allow us to have an outside look at the network behaviour. Mainly, there are two features of network behaviour that we want to capture, namely, *long-term behaviour* and *transient behaviour*.

Measurements of gene expression levels of the network that is the experiment's object are performed using traditional wet-lab techniques such as RNA sequencing. Potentially, already measured data can be obtained from various free-access databases that store expression data from many researches, such as ArrayExpress [17] or Gene Expression Omnibus [18]. Recently, *in silico* benchmarks generating techniques, for example, BoolODE [19], or GeneNetWeaver [20], have been proposed. Such methods can be used for performance profiling of inference tools. Their main advantage is that they have no material requirements and are easily applicable.

In this chapter, we will describe the experiment types that allow capturing mentioned features of networks. In advance, we will specify expression data formats that allow us to store the progress or results of such experiments. Finally, we describe how we approximate them in the context of BNs.

2.1 RNA sequencing

This section paraphrases multiple articles [21, 22, 23, 24, 25] discussing RNA sequencing.

RNA sequencing (RNA-seq) is a genomic approach for detecting and quantitatively analyzing mRNA molecules. Their absolute quantity strongly correlates with the expression levels of their corresponding genes. The evolution of relevant gene expression levels reasonably determines cellular state changes.

As the absolute quantity of molecules in a single cell is tiny, earlier methods could not measure such a small amount of molecules. That

is why in the past, *bulk RNA-seq* measurements were performed on large samples of cells, where the absolute concentration of mRNA was higher. The result of sequencing such huge cell samples is called *bulk expression data*. Thus, bulk RNA-seq provides the average expression level in a sample for each gene, which may contain different cell types. This fact makes it impossible to observe individual cells' behavior and study their differentiation.

Conversely, *single-cell RNA-seq* is a relatively new technology that measures the gene expression levels for each transcript within each individual cell of the sample. It allows a representation of the distribution of expression levels among measured cells. Thus, single-cell RNA-seq aims to study a particular cell or cell type behaviour. At the same time, bulk RNA-seq was primarily designed to characterise the average behaviour of larger cell samples.

One of the critical advantages of single-cell RNA-seq is that it enables the analysis of cells that are rare in number, such as tissue stem cells. In addition, obtaining the expression profiles of single cells is very useful for dissecting the heterogeneity within seemingly homogenous cell populations.

On the contrary, the main disadvantage of single-cell data is that it is noisier than bulk data. The technical noise in single-cell expression data arises due to the low absolute concentration of mRNAs in a single cell. This fact leads to two primary sources of technical noise, which are *PCR amplification bias* (distribution in the amplified sample may differ from the original sample due to different efficiency of amplification of different templates) and *drop-outs*. Drop-outs represent false-negative results, where genes are measured as not significantly expressed due to the low efficiency of mRNA even though they are expressed. Therefore, network inference techniques that are robust to noise are required when reconstructing networks using single-cell expression data. Since Boolean models are relatively robust due to the binarisation of expression levels, we decided to work only with single-cell expression data.

2.2 Long-term behaviour experiments

In Boolean networks, the network's long-term behaviour possibilities are characterised by a set of attractors. We discuss types of attractors in Section 1.4. Therefore, the aim of the experiments, which are to analyse the network's long-term behaviour for BN inference, is to identify the set of BN attractors. Recent inference methods mostly use so-called *steady-state* data, which only provides information about the experimented network's single-state attractors. The reason is that the set of steady-states is also preserved in synchronous semantics, which is used in most recent BN inference tools. The other, the more technical explanation, is that steady-states are much easier to measure than multi-state attractor types.

The fundamental problem associated with measuring multi-state attractors is that biological systems are continuous systems. Therefore, the transition between states is smooth. On the contrary, RNA-seq measurements are based on a periodic sampling of actual gene expression levels. Thus, the system's behavior between measurements is unknown and can never be unambiguously determined. The advantage of capturing steady-state attractors lies in the fact that the system persists at approximately fixed expression levels and can be identified with less probability of inaccuracy. However, some recent studies also studied the oscillatory behaviour of systems [26, 27, 28]. Therefore, we aim to enrich the inference with data from more complex types of attractors in the future.

2.3 Transient behaviour experiments

The transient behaviour of experimented GRN is characterised by the evolution of expression levels of its genes. Time series experiments provide information about transient expression levels of genes in GRN sampled in discrete events within a limited time period. One time series experiment captures the expression level progress of one specific run of a network from the initial state, usually until a long-term behaviour is converged. The such run can be later aligned with one particular path of BN's STG. The data format that stores the results of the experiment revealing the transient behaviour of GRN is called

time-series data. Such data contains time-ordered samples of transient gene expression levels. However, they cannot be aligned with paths of synchronous STG because it under-approximates natural systems' behaviour. Consequently, we must use asynchronous BN simulation to be able to align it with real single-cell time-series data.

2.4 Wild-type and mutant-type experiments

In connection with GRN experiments, we often encounter the term *perturbation*. A gene is *perturbed* if its natural expression level is purposefully altered. On this basis, we refer to the *unperturbed network* if none of its genes are perturbed in a given experiment. On the contrary, a *perturbed network* has at least one gene perturbed.

We consider two types of experiments based on the network that is the object of two previously mentioned experiment types. Any experiment aimed at the study of an unperturbed network is referred to as *wild-type* experiment. On the contrary, we denote the experiment as *mutant-type* if performed on a perturbed network. In this thesis, we consider these types of mutant-type experiments, depending on the type of perturbation:

- *Knockout* - the expression of a single gene is totally silenced for the whole length of an experiment,
- *Over-expression* - the expression of a single gene is maximised for the whole length of an experiment.

Other types of mutant experiments with multiple gene perturbations or fractional reduction of expression are being performed, too, such as dual knockouts, knockdowns, etc. Still, we do not use them in the fulfillment of this thesis.

When inferring BN from expression data, it is necessary to perform a comprehensive measurement of gene expression. Generally, the dataset containing only wild-type experiments is not comprehensive enough to detect all the unknown gene regulations. A usable expression dataset should contain as many mutant-type experiments as possible [1]. Alternation of the expression level of a specific gene allows us to reveal its closest relationships with other genes.

2.5 Expression data binarisation

Clearly, since Boolean networks operate over Boolean domain $\mathbb{B}$, binarisation of the original expression data is required. For this purpose, we can use clustering algorithms to decide which genes are highly expressed and which are not. Thus, if we consider a GRN with n genes, we can binarise each experiment record (one steady-state or one time-series sample) into a Boolean vector $\mathbb{B}^n$.

As a result, we obtain a tuple of binarised expression data $\mathcal{E}(Wt, Ko, Oe)$ that characterises an unknown BN $\mathcal{B}^{(n)}$ by sets of *single-cell steady-states* obtained in multiple experiments. These correspond to the states in which the individual cells of a given experiment have stabilized. Formally, $Wt \in 2^{\mathbb{B}^n}$ denotes a set of single-cell wild-type steady-states and $Ko, Oe : \{1, \ldots, n\} \to 2^{\mathbb{B}^n}$ denote mappings of a perturbed gene index to its corresponding set of knock-out (resp. over-expression) single-cell steady-states.

3 Boolean network inference

In this chapter, we introduce the notion of so-called *target Boolean network* alongside its components, which unifies single-cell steady-state data of multiple experiments and properties required from the infering BN. Based on that, we formulate the inference problem and clarify the motivation for the asynchronous-based inference from single-cell expression data.

3.1 Prior knowledge

In addition to enabling the use of multiple types of expression data for inference, our long-term goal is to allow the application of different requirements to the target network, with which the resulting model must be consistent. Therefore, we introduce several restrictions on the inferred network, including literature-based prior knowledge or hypothesis about the underlying network. It helps narrow the space of all possible BNs that fit the provided expression data. This narrowing is crucial in terms of the accuracy and time complexity of inference methods. However, we must be careful with their imposition since we require strict compliance with them in all the inference steps. Here, we list types of prior knowledge that we consider meaningful and outline the method of their implementation.

3.1.1 Static properties

Static properties aim to constrain the topology of target BN.

We define a mapping function *constraints* : $\mathcal{V} \times \mathcal{V} \to \{-1, 0, 1\}$ that which for each pair of Boolean variables $(v_s, v_t) \in \mathcal{V} \times \mathcal{V}$ returns their apriori known relationship stereotype, such that:

$$\mathit{constraints}(v_s, v_t) = \begin{cases} 1 & \text{if } v_s \text{ positively regulates } v_t, \\ -1 & \text{if } v_s \text{ negatively regulates } v_t, \\ 0 & \text{for no known regulation from } v_s \text{ to } v_t. \end{cases}$$

The *constraints* mapping function helps to narrow the search space of all possible Boolean networks by fixing some of its regulations. Each

function $f_i \in \mathcal{F}$ must stay consistent with it. Details are described later in Section 4.4.

In some cases, it is known that some Boolean variables represent input components of BN (e.g., input signals) or output components (e.g., phenotypes). Therefore, it makes sense to explicitly enumerate sets of *fixed input and output variables*. In practice, this means that we prevent that in any step of BN inference, the variables marked as input are regulated by other variables and that the output variables regulate other variables of the network.

In addition, we may want to limit *in-out degrees* of other variables from above in order to limit the overall density of regulations in BN or otherwise ensure compliance with the required network topology, e.g., scale-free topology. However, this request is directed to future work.

3.1.2 Dynamic properties

Dynamic properties restrict the shape of STG. As part of future work, we plan to enable the specification of so-called *decision points*, i.e., states in which the network decides between several strong basins of attraction. Moreover, we want to be able to (partially) specify the basins of some attractors. Both these dynamic requirements will be verified by a hybrid CTL model checker [29], which is currently being developed in our lab.

3.2 Inference problem definition

Finally, we define the structure of *target Boolean network*, which combines several requirements for an inferred BN with expression data. A Target Boolean network $\mathcal{T}(n, \mathcal{E}, In, Out, constraints)$ is a tuple containing:

- number of desired Boolean variables n,
- tuple $\mathcal{E}(Wt, Ko, Oe)$ combining expression data types,
- set of input variables $In \subset \mathcal{V}$,
- set of output variables $Out \subset \mathcal{V}$,
- mapping of fixed regulations *constraints*.

The Boolean network inference problem can be described as a problem of finding a set of Boolean networks $\mathcal{B}$ (defined in Section 1.2) that best fits the provided experimental data and are consistent with all the requirements, specified in the target Boolean network $\mathcal{T}$. Generally, it is expected that the more precise data and properties are given as inputs, the more accurate inference can be obtained.

3.3 The selection of semantics

It is well known that synchronous semantics provides a relatively strong assumption of biological systems, and the asynchronous reflects reality better. It is due to the fact that biological reactions take place independently of each other and last for different lengths of time. Reactions cause a change in the concentration or expression level of molecules, represented in the Boolean network by changes in the states of network variables. By updating all the variables synchronously, synchronous semantics limits the execution of all system responses at once. This limitation results in the generation of only deterministic trajectories and the inability to capture, for example, a cellular differentiation process. On the contrary, asynchronous semantics even over-approximate reality since some generated trajectories are not biologically feasible due to higher priorities of some reactions [30].

Nevertheless, most existing studies adopt a synchronous semantics approach because of the lack of more complex expression data that capture advanced behaviour of biological systems that cannot be described by synchronous semantics. An asynchronous semantics can reach remarkably more states from a given initial state but consequently is more computationally demanding as the number of transitions in STG grows exponentially with the number of variables.

The synchronous semantics of a particular BN differs from its corresponding asynchronous one in transitions with a Hamming distance greater than one. The reason is that in asynchronous semantics, the Hamming distance of state and its successor is at most one because exactly one Boolean function is applied. Thus, at most, one variable value can be changed in a single time step [16].

It is also straightforward to see that the set of steady-state attractors of an asynchronous semantics of given BN is the same as in its corresponding synchronous semantics because the Hamming distance of self-loops is zero. Nevertheless, the reachability of steady-states is not preserved between semantics. The set of reachable steady-states from a specific state may differ between semantics, although the sets of steady-states of complete transition graphs of both semantics are equal [3]. If the synchronous semantics contains two or more steady-states, strong basins attractors do not intersect [31]. Therefore, the strong basin of each attractor forms an isolated component of the STG. In contrast, in asynchronous semantics, multiple attractors' weak basins can intersect. Therefore, more than one attractor can be reachable from one specific state.

Finally, it is essential to explicitly mention that even though there are TSCCs preserved across semantics, the overall state transition graphs typically significantly differ. Both semantics generate different trajectories and describe different types of network behaviour.

Based on this, we decided to build a purely asynchronous-based framework. Although the proposed framework, which is part of the overall desired framework, focuses only on inference from single-cell steady-state data, which we claim are preserved in synchronous semantics, the general motivation for the use of asynchronous semantics lies in the ease of extending the existing framework with the other mentioned experiment data types and properties on the target network which are not compatible with synchronous semantics.

The potential extension of the input of the current framework by complex attractors or single-cell time-series data directly excludes the use of synchronous semantics. The same applies to the enrichment of the fitness evaluation by the output of the CTL model checker, which is capable of deciding the eligibility of a certain state in the basins of the network's attractors. In order not to be forced to rebuild the whole framework from the ground up in the future, we decided to use asynchronous attractor analysis already in the initial version.

In addition, the existing fitness metric already relies on the output of the asynchronous attractor analysis in some cases to determine whether a given state is part of any attractor of the asynchronous semantics.

4 Framework design

Our framework adopts a classical genetic programming approach which generally has the following significant phases, well described in [32]. Firstly, an initial generation of possible solutions to a given problem is created so that the solutions are suitably distributed across the entire space of solutions. Each solution is then evaluated by defined fitness metrics. Elite-fitting solutions are *reproduced*, i.e., copied to the next generation without being mutated. Other solutions are selected with probability based on their fitness. The solutions so selected are mutated and then added to the next generation. Such a sequence of steps is iteratively performed until some termination condition is satisfied. Finally, the best-fitting solutions are returned as a result.

In the context of a Boolean network inference problem, we define a generation $\mathcal{G}$ of candidate solutions such that each solution is represented by a Boolean network $\mathcal{B}$.
A Generation $\mathcal{G}$ $(m, \mathcal{N}, \varrho)$ is a tuple containing:

- number of networks in generation m,
- list of m candidate Boolean networks $\mathcal{N}$,
- vector of fitness scores $\varrho \in \mathbb{R}^m$, such that $\varrho_i \in [0; 1]$ determines the similarity of candidate network $\mathcal{N}_i$ to the target Boolean network $\mathcal{T}$.

In Figure 4.1, we show the overall scheme of our framework. Algorithm 1 then shows the high-level perspective of the framework design.

Our genetic framework first processes the input data to create a target Boolean network $\mathcal{T}$ against which individual solutions will be evaluated. Then, it derives additional regulations from given steady-state data stored in $\mathcal{E}$ by computing gene-to-gene correlations. A given *threshold* value determines the value of the correlation between a pair of genes, which is considered significant enough to indicate potential regulation between these genes. In this way, the algorithm derives a reasonable set of regulations that have the potential to occur in the target network. Subsequently, it initialises the first generation consisting of m solutions using the input constraints and derived regulations

Algorithm 1: The inference algorithm

Input: $\mathcal{T}(n, \mathcal{E}, In, Out, constraints)$, *threshold*, *m*
Output: Set of BNs that best fit the requirements given by $\mathcal{T}$

```
1  derived_regs ← DeriveRegulations(T, threshold)
2  G ← CreateInitGen(m, T, derived_regs)
3  current_iter ← 1
4  while true
5  |  ComputeGenFitness(G)
6  |  if max number of iterations performed ∨ max fitness reached then
7  |  |  output best BNs of the last generation to separate files
8  |  |  return
9  |  G ← CreateNewGen(G)
10 |  current_iter ← current_iter + 1
```

(each solution consists of a single BN). Then the framework iteratively (i) computes the fitness of each BN in the current generation, (ii) selects some BNs from the current generation based on certain rules that are driven by the fitness of individual networks, (iii) constructs the next generation from selected networks, (iv) mutates a preset number of regulations of the next generation's networks. These steps are repeated for the preset number of iterations or until satisfying fitness is reached.

In this chapter, we describe in detail how the mentioned individual genetic components are designed.

4.1 Input specification

Our framework accepts three types of inputs. (i) Matrix of single-cell steady-states of multiple experiments performed on the underlying network, (ii) constraints matrix of fixed regulations, and (iii) two sets that firmly declare subsets of the input and output variables.

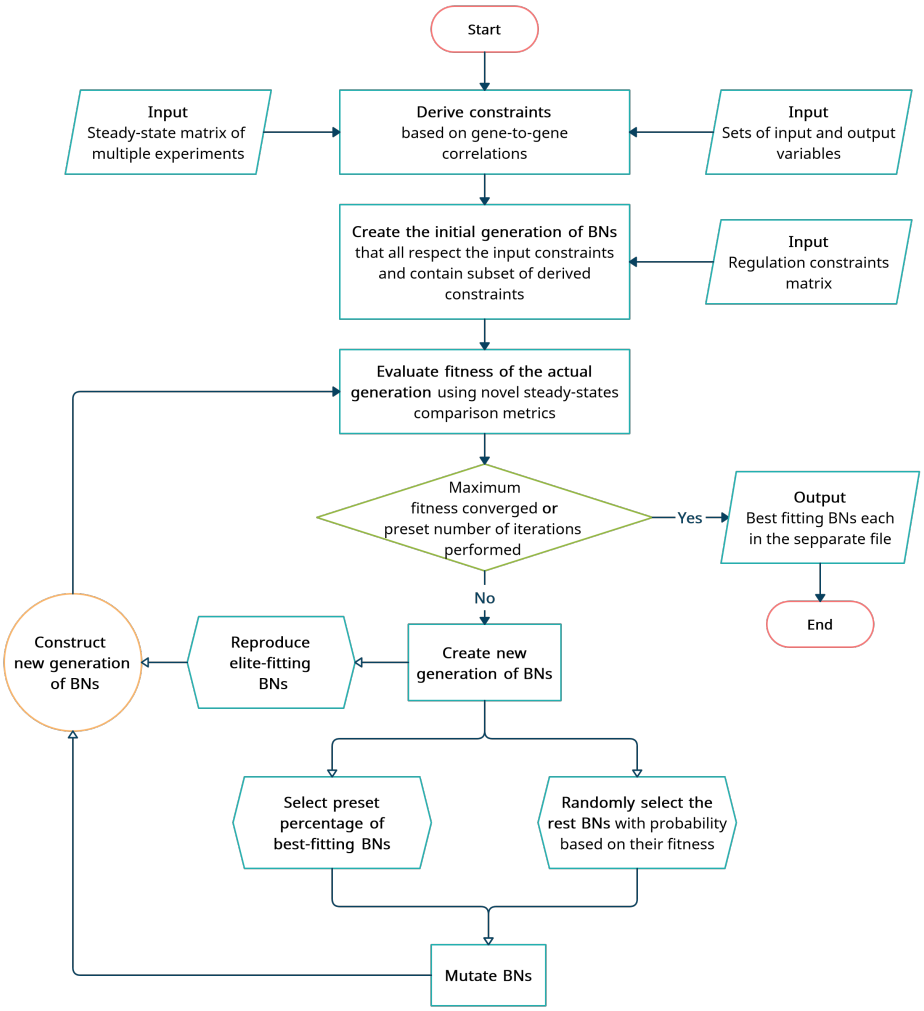

Figure 4.1: Overall framework of our genetic algorithm.

4.1.1 Steady-state matrix

Steady-state data are represented as a 2D matrix structure where each row represents one steady-state. Accordingly, each column corresponds to one Boolean variable, except the first column. The value in the first column specifies the type of experiment to which the steady-state in the row belongs. If the value is equal to -1, it indicates a wild-

type steady-state. Otherwise, the value specifies the perturbed gene index. Based on the expression value of such index in the row, we can then determine whether it is a knock-out (0) or over-expression (1) of this gene. The example of a steady-state matrix is shown in Figure 4.2. Blue values in the first column determine the perturbed gene index, and the red value in the corresponding column determines the type of experiment to which the steady-state belongs.

-1	0	1	0	1
-1	1	0	1	0
0	0	1	0	1
0	1	1	0	0
1	1	0	1	0
1	0	1	0	1
1	1	1	0	1
3	1	0	1	0
3	1	0	0	0
3	1	1	0	1

Figure 4.2: An example of a steady-state matrix. Each row corresponds to one steady-state. Consequently, each column corresponds to one gene, except the first column. The value in the first column (in blue) determines the perturbed gene index (-1 denotes wild-type). The value at the corresponding index (in red) determines the type of experiment to which the given steady-state belongs (0 for knock-out, 1 for over-expression). Having mutant-type experiments for every gene is not obligatory. Here, mutant-types of gene 2 are absent.

4.1.2 Regulation constraints matrix

The regulation constraints are also represented by a 2D matrix. Each matrix row corresponds to one *fixed* regulation. Such regulation cannot be altered nor removed from BN. The value in the first column of the matrix denotes the *source* gene index, the value in the second column corresponds to the *target* gene index, and the third column determines the sign of the regulation ('+' for the activation, '-' for the inhibition).

The regulation constraints matrix defines the *constraints* mapping included in $\mathcal{T}$. The presence of activation from a source gene index s to a target gene index t implies that $constraints((v_s, v_t)) = 1$, inhibition of such gene indices implies that $constraints((v_s, v_t))$ = -1. Otherwise, $constraints((v_s, v_t)) = 0$.

It is worth noting that the regulation constraints should not conflict with the input and output variables specification, as this will significantly affect the correctness of the inference. Therefore, if a variable is specified as input, it should not be the target of any regulation constraint and vice versa for output.

4.2 Derivation of regulations

In addition to the regulatory constraints defined in the regulation constraints matrix, we try to infer other potential regulations by computing gene-to-gene correlations. For all pairs of genes, we calculate the correlation of their expression vectors across all experiments, i.e., their corresponding columns from the input steady-state matrix. The potential regulation is added to the set of derived regulations if the correlation of the pair of genes is greater than or equal to a preset threshold and, at the same time, does not violate the requirements for input and output variables specified in $\mathcal{T}$. Specifically, a regulation is inconsistent with the requirements if its source and target are both identically input or output. Pearson's correlation is a symmetric metric. Therefore, it is unnecessary to calculate it in both directions. If the regulation is consistent with the requirements of the target network in only one direction (e.g., if one of the variables is defined as input or output), it is clearly added. However, if it is consistent with the requirements in both directions, the direction is randomly chosen in the constructor. The sign of the regulation is always consistent with

the sign of the Pearson's correlation value. The procedure is described in Algorithm 2.

Algorithm 2: Derivation of additional regulations

```
1 Function DeriveRegulations(𝒯, threshold)
2     derived_regs ← ∅
3     for i ← 0 to n do
4         for j ← i + 1 to n do
5             corr ← PearsonsCorrelation(𝒯, i, j)
6             if ‖corr‖ ≥ threshold ∧ IsConsistent(𝒯, i, j) then
7                 derived_regs ←
                    derived_regs ∪ {new Regulation(i, j, corr)}
8     return derived_regs
```

4.3 Representation of update functions

In Section 1.1, we have formally defined the high-perspective concept of NCFs, which we require to be respected by each $f_i \in \mathcal{F}$. It should be obvious that the form of the functions in $\mathcal{F}$ as well as the set of observable regulator indices of individual variables $\mathcal{V}$ can change due to network mutations in individual iterations of the genetic algorithm. In this section, we present a way of representing each function $f_i \in \mathcal{F}$ such that there always exists the equal function f_i' that is NCF in some permutation of $reg(v_i)$.

Consider a $\mathcal{B}^{(n)}$ and one of its variables v_i with its corresponding update function f_i. Let us assume k regulators of v_i ($k \leq n$) and consider one their permutation, denoted $\pi_i = (v_1, \ldots, v_k)$. Let us also consider vectors of canalyzing values $(i_1, \ldots, i_k) \in \mathbb{B}^k$ and canalyzed values $(o_1, \ldots, o_k) \in \mathbb{B}^k$. Then we represent the function f_i as a list of tuples $[(i_1, o_1, v_1), \ldots, (i_k, o_k, v_k)]$. Such representation implies the existence of an equal function f_i' that is an NCF in π_i (details in Subsection 4.3.1). The evaluation of such a represented NCF is described in the Algorithm 3. Suppose no variable is interpreted as its canalyzing value. In that case, the function returns the so-called *default*

value, which is equal to the negation of the canalyzed value of the last variable. Later in Section 4.6, we present how different types of mutations of NCFs are constructed.

Algorithm 3: Evaluation of NCF

```
1 Function EvaluateNCF(f, interpretation)
2     for i ← 0 to ‖interpretation‖ do
3         if interpretation[i] = Canalyzing(f, i) then
4             return Canalyzed(f, i)
5     return ¬ Canalyzed(f, ‖interpretation‖ − 1)
```

4.3.1 Representation of NCF

The previously defined sequence representation of the function encodes the following human-readable logical expression structure:

$$(polar[v_1]\ oper[v_1]\ (\ldots(polar[v_{k-1}]\ oper[v_{k-1}]\ (polar[v_k]))))$$

The following table shows how canalyzing, canalyzed value pairs encode a variable's literal polarity and operator. Except for that, the canalyzing, canalyzed values pair of the last variable encodes only literal as it has no corresponding operator. Note that equal values encode positive literal and non-equal encode negative literal, and also that the canalyzed value of 0 encodes $\wedge$ while 1 encodes $\vee$.

$$\begin{aligned}(0,0) &\rightarrow \textit{ positive literal }, \wedge\\ (0,1) &\rightarrow \textit{ negative literal }, \vee\\ (1,0) &\rightarrow \textit{ negative literal }, \wedge\\ (1,1) &\rightarrow \textit{ positive literal }, \vee\end{aligned}$$

4.4 Construction of the initial generation

Before the iterative phase of the genetic algorithm starts, it is necessary to construct the initial generation of BNs. Each generation, including the initial generation, contains exactly the preset number of networks m. Since the regulations specified in the constraints matrix are fixed, each of them must be contained in every network of the initial generation $\mathcal{G}_{init}$. Consequently, they stay preserved across all the following generations of the genetic algorithm. More formally, when considering variable indices s and t, then for each network in the generation $\mathcal{G}_{init}$ the following properties apply:

$$constraints(v_s, v_t) = 1 \implies f_t = [\,..., (i_s, o_s, v_s), ...\,] \text{ such that } i_s = o_s$$

$$constraints(v_s, v_t) = -1 \implies f_t = [\,..., (i_s, o_s, v_s), ...\,] \text{ such that } i_s \neq o_s$$

Next, a random subset of the derived regulations is individually selected for every network. Although these regulations are not fixed, they provide a reasonable way to ensure a more significant variance of the networks in the initial generation $\mathcal{G}_{init}$. By moving the value of the correlation threshold, we can statistically ensure a more extensive set of derived constraints and hence a more significant variance of the initial generation's networks $\mathcal{N}_{init}$. Finally, one random mutation is performed on one gene of each BN in $\mathcal{N}_{init}$. The initial generation thus created enters the first iteration of the genetic algorithm. The procedure is described in Algorithm 4.

Algorithm 4: Creation of the initial generation of Boolean networks

```
1 Function CreateInitGen(m, T, derived_regs)
2     G_init(m, N_init, ϱ_init) ← new Generation(m)
3     foreach net in N_init do
4         add all constraints to net as fixed regulations
5         add some randomly chosen derived_regs to net
6         perform 1 mutation on 1 randomly chosen gene from net
7     return G_init
```

4.5 Fitness evaluation

This section describes the procedure of evaluating the fitness of one particular experiment of a BN. The experiment is determined by a perturbed gene index and the type of the experiment.

We present a novel metric to evaluate the fitness of BN in the given experiment concerning the experiment's corresponding set of steady-states stored in $\mathcal{E}$. The metric compares attractor sets using a unbalanced minimum weighted assignment on a complete bipartite graph.
A Complete bipartite graph $\mathcal{BG}(V_t, V_c, E, \delta)$ is a tuple, such that:

- V_t is a set of vertices containing Boolean vectors representing a set of steady-states of a *target BN*,
- V_c a set of steady-states of the currently evaluated BN,
- $E = V_t \times V_c$ is a set of undirected edges such that there is an edge between every pair of vertices of mutually distinct sets,
- $\delta : E \to \mathbb{N}_0$ is a mapping that assigns weight to each edge such that the weight of the edge is equal to Hamming distance of the connected vertices.

4.5.1 Preparation for evaluation

The Boolean network model can contain so-called *isolated variables*, i.e., simultaneous input and output variables of the network. Such variables are excluded from BN's attractor states. Therefore, if the network contains isolated variables, the dimension of its steady-states will be reduced. For the fitness metric to work correctly, it is necessary to cut the isolated variables from the steady-state data $\mathcal{E}$ so that the dimensions of the Boolean vectors of the target and simulated steady-states are equal.

Algorithm 5 reflects the procedure of detecting isolated variables of given BN concerning the perturbed gene index of the current experiment (-1 denotes wild-type). If the gene is perturbed, its update function is set to *true* (for an over-expression) or *false* (for a knock-out). Therefore, it loses its original update function and, consequently, its regulations. That automatically makes such gene the input variable.

This fact is also reflected in line 11. If the currently investigated gene regulates only the perturbed gene, which regulations were formally removed, it also becomes an output variable.

After analysing whether the model contains isolated variables, we perform an attractor analysis of the current BN. The final computational model contains, among other attributes, a list of steady-states of the current network. Suppose there are isolated variables in the model. In that case, we reduce the dimension of the target steady-states $\mathcal{E}$ by cutting out the columns of the matrix with the indices of the isolated variables.

We then initialise the complete bipartite graph $\mathcal{BG}$, which serves as the computational structure for the novel fitness metric. The sets of the bipartite graph V_t and V_c are formed by the sets of target steady-states corresponding to the evaluated experiment, resp., steady-states of the currently simulated BN model. Thus initialised graph enters the metric. The whole preparation phase is described in Algorithm 6.

Algorithm 5: Detection of model's isolated variables

```
1  Function DetectIsolatedVars(B, pert_gene_index)
2      if pert_gene_index = −1 then
3          input_variables ← ∅
4      else
5          input_variables ← {pert_gene_index}
6      output_variables ← ∅
7      for gene ← 0 to ‖V‖ do
8          if Regulators(gene) = ∅ then
9              input_variables ← input_variables ∪ {gene}
10         regulates ← RegulatedBy(gene)
11         if regulates ⊆ {pert_gene_index} then
12             output_variables ← output_variables ∪ {gene}
13     return input_variables ∩ output_variables
```

4.5.2 Fitness metric

First, we compute the *unbalanced minimum weighted assignment* on the given instance of $\mathcal{BG}(V_t, V_c, E, \delta)$. As the cardinality of V_t and V_c may differ, we assume the unbalanced version of minimal weighted assignment $A_{min} \subset E$ such that $\|A_{min}\| = min(\|V_t\|, \|V_c\|)$. Additionally, A_{min} satisfies a requirement that each vertex from V_t resp. V_c is assigned to at most one vertex from V_c resp. V_t and the following total cost function is minimised.

$$\sum_{a \in A_{min}} \delta(a)$$

Finally, fitness is calculated as the ratio of *matching variables/total variables*. The number of matching variables is computed as the number of all matched variable pairs, i.e., the number of vertices in the smaller set times their dimension, minus the cost of the minimal weighted assignment A_{min}. The cost of A_{min} equals the number of variables in assigned vectors that do not match. The number of total variables then always refers to the number of variables in the V_t, independent of the relation between the sizes of the sets V_t and V_c. However, based on this relation, we distinguish two cases.

First, suppose the currently analysed model contains more or as many steady-states as there are in its corresponding experiment in $\mathcal{E}$. In that case, the number of total variables in the graph equals the number of variables in the smaller of sets V_t, V_c. Hence we do not penalize the overlap of the model's steady-states over the data since it is frequent that the data do not capture the complete behaviour of the experimented network.

The second case applies when the number of steady-states in the data is greater than in the model. In this case, the model does not adequately describe the network behaviour. For each overlapping steady-state, the resulting fitness is penalized because the number of total variables is now equal to the number of variables in V_t, which is currently larger than V_c. For each state from V_t that has not been matched by the assignment, additional attributes obtained by attractor analysis are used. Namely, the metrics tries to determine whether the unmatched state is included in a different type of attractor of currently analysed BN and thus has a higher potential to become

a steady-state. If so, such a state is penalized with only half of the weight. The described procedure of the fitness evaluation is described in Algorithm 7.

Algorithm 6: Evaluation of fitness of specific BN experiment with a focus on the steps of the preparation phase.

```
1 Function ComputeExpFitness(B, pert_gene_index, pert_type)
2     isolated_vars ← DetectIsolatedVars(B, pert_gene_index)
3     cpu_model ← perform attractor analysis of B
4     V_t ← GetTargetSinks(T, pert_gene_index, pert_type)
5     if isolated_vars ≠ ∅ then
6         reduce the dimension of V_t by removing isolated_vars
7     V_c ← DetectSteadyStates(cpu_model)
8     BG ← new BipartiteGraph(V_t, V_c)
9     return EvaluateMetric(B, BG, cpu_model, isolated_vars)
```

Algorithm 8 describes how the fitness of the whole generation is computed. The *total score* for each individual network is calculated as the average score across all the network experiments, which steady-states are stored in $\mathcal{E}$.

Algorithm 7: Evaluation of the fitness metrics

```
1  Function EvaluateMetric(B, BG, cpu_model, isolated_vars)
2      cost, pairs ← MinimalWeightedAssignment(BG)
3      dim ← ||V|| − ||isolated_vars||
4      matching_vars ← Min(||V_t||, ||V_c||) ∗ dim − cost
5      total_vars ← ||V_t|| ∗ dim
6      if ||V_t|| > ||V_c|| then
7          unmatched_states ← GetUnmatchedStates(BG, pairs)
8          foreach state in unmatched_states
9              if IsAttractorState(cpu_model, state) then
10                 matching_vars ← matching_vars + dim ∗ 1/2
11     return matching_vars / total_vars
```

Algorithm 8: Computation of the whole generation's fitness

```
1  Function ComputeGenFitness(G(m, N, ϱ))
2      for i ← 0 to m do
3          totalscore ← ComputeExpFitness(B_i, -1, None)
4          foreach ko in knockouted gene indices
5              totalscore ←
                totalscore + ComputeExpFitness(B_i, ko, false)
6          foreach oe in overexpressed gene indices
7              totalscore ←
                totalscore + ComputeExpFitness(B_i, oe, true)
8          ϱ_i ← AverageScore(totalscore)
```

4.6 Mutation of current generation

One of the fundamental parts of the genetic algorithm is the mutation of solutions of the current generation. The representation of update functions in $\mathcal{F}$, presented in Section 4.3, allows us to mutate functions easily. We distinguish between six types of mutations. In the following, we list them together with their design.

- *Canalyzing value reversion* - reverts canalyzing value of one certain gene:
$$f = [\ ..., (i, o, v), ...\ ] \rightsquigarrow [\ ..., (1 - i, o, v), ...\ ]$$
- *Canalyzed value reversion* - reverts canalyzed value of one certain gene:
$$f = [\ ..., (i, o, v), ...\ ] \rightsquigarrow [\ ..., (i, 1 - o, v), ...\ ]$$
- *Canalyzing and canalyzed values reversion* - reverting canalyzed and canalyzing values of one certain gene:
$$f = [\ ..., (i, o, v), ...\ ] \rightsquigarrow [\ ..., (1 - i, 1 - o, v), ...\ ]$$
- *Canalyzing and canalyzed values swapping* - by swapping the order of two regulators, we increase, resp., decrease the priority of such regulators. The earlier a regulator is in the sequence, the higher priority it has:
$$f = [\ ..., (i, o, v), ..., (i', o', v'), ...\ ] \rightsquigarrow [\ ..., (i', o', v'), ..., (i, o, v), ...\ ]$$
- *Canalyzing and canalyzed values insertion* - add a new regulator to the given variable update function alongside with corresponding canalyzing and canalyzed value:
$$f = [\ ..., ...\ ] \rightsquigarrow [\ ..., (i, o, v), ...\ ]$$
- *Canalyzing and canalyzed values deletion* - deletes certain regulator from the given variable update function:
$$f = [\ ..., (i, o, v), ...\ ] \rightsquigarrow [\ ..., ...\ ]$$

Based on this, we consider only a subset of meaningful mutations for each gene. The first two of the listed mutations change the sign of the regulation of the corresponding gene and, therefore, must be forbidden for fixed regulators. It is also intuitively forbidden to remove fixed regulators. The reversion of both canalyzing and canalyzed values at once does not change the regulation sign. Therefore, it is also allowed for fixed regulations. If the function's arity is less than two, swapping the order of two regulators is impossible. Finally, adding new regulators to a network that is too dense, i.e., contains too many regulations, is forbidden. The procedure of picking the meaningful mutations is shown in Algorithm 9.

Algorithm 9: Selection of meaningful mutations for specific regulator that will mutate f

```
1  Function SelectMutations(f, reg)
2      if reg ∈ Regulators(f) then
3          allow canalyzing and canalyzed values reversion
4          if Arity(f) ≥ 2 then
5              allow canalyzing and canalyzed values swapping
6          if reg is not fixed regulator of f then
7              allow canalyzing value reversion
8              allow canalyzed value reversion
9              allow canalyzing and canalyzed values removal
10     elif topology of the whole B is sparse enough
11         allow canalyzing and canalyzed values insertion
12     return list of allowed mutations
```

Algorithm 10 describes how the mutation of a randomly picked single update function of a BN is performed. The framework allows users to set the number of desired mutations for each update function globally. The corresponding number of regulators is then selected randomly. The set of meaningful mutations is evaluated for each selected variable, and one randomly picked is executed.

Algorithm 10: Mutation of single update function f

```
Function MutateUpdateFunction(f, num_of_muts)
    regs_to_mut ← select num_of_muts random non-output genes
    foreach reg in regs_to_mut do
        mutations ← SelectMutations(f, reg)
        perform 1 randomly chosen mutation from mutations on f
```

4.7 Construction of the next generation

After the fitness of the networks in the current generation $\mathcal{G}_{act}$ is evaluated, the selection to the next generation $\mathcal{G}_{next}$ is performed. The elite-fitting networks are reproduced for the next generation. This will ensure that the fitness of the best networks in each generation will have a non-decreasing tendency. Then, based on the preset value, a certain percentage of the best networks is selected for the next generation (including the previously selected elite networks), and a preset number of random mutations is performed on each of them. Finally, the next generation is completed with randomly selected networks, using fitness vector ϱ_{act} as the weight vector. To amplify the distribution of weights, we apply softmax to the vector of weights. Consequently, the randomly selected networks are also mutated. The described procedure is shown in Algorithm 11.

4.8 Output specification

The algorithm's iterative phase stops when either the preset threshold fitness is reached or if the preset number of iterations is completed.

As an output, the algorithm returns all Boolean networks from the last generation with the best fitness score. If multiple BNs reach the highest score, all of them are returned.

Here, the famous Occam razor principle should be considered. It is known that the number of regulators tends to be relatively small, and the whole GRN is typically rather sparse [33]. Therefore, the simpler

Algorithm 11: Creation of next generation of Boolean networks based on the current one

Function *CreateNewGeneration*($\mathcal{G}_{act}(m_{act}, \mathcal{N}_{act}, \varrho_{act})$)
- $\mathcal{G}_{next} \leftarrow$ **new** Generation(m_{act})
- reproduce elite BNs from $\mathcal{N}_{act}$ to $\mathcal{G}_{next}$
- select fixed percentage of the best BNs from$\mathcal{N}_{act}$
- $weights \leftarrow$ SoftMax(ϱ_{act})
- select the rest BNs based on *weights*
- mutate all the selected BNs and add them to $\mathcal{G}_{next}$
- **return** $\mathcal{G}_{next}$

the rules of the output network, the more likely this network is the desired one.

5 Implementation

We implemented the algorithm described in the previous chapter in Python. The implementation related to this thesis is freely available on `https://github.com/xsimurka/Constraint_steadystate_BN_inference/tree/bachelor_thesis`. This chapter discusses how we have implemented the procedures we formally described in the previous chapter.

5.1 Third-party libraries

The key part of our algorithm is the attractor analysis of the Boolean network, for which we used the *biodivine_aeon* package [34]. This package provides Python bindings for the AEON tool [35], originally written in Rust. In addition, we use the functionality provided by the author of this package to process and interpret the results of the attractor analysis. Namely, we wanted to enumerate all the explicit steady-states of the analysed BN model and determine if a particular state is part of some attractor of the studied BN model. The described functionality is implemented in the module *detect.py*.

In addition, we also used other python packages, namely: *scipy* for calculating unbalanced minimum weighted assignment and softmax, *numpy* and *pandas* for working with series and arrays, and *random* for random selection of gene indices for mutation.

5.2 Input file format and parsing

The input of the algorithm consists of two text files. The first contains a discretised matrix of steady-states. The second includes the input constraints matrix. The `DELIMITER` global constant specifies a character that separates values in both matrices. Full file paths to both files are passed as arguments of `main` function alongside sets of input and output variables.

Lines of both files are then individually parsed. Values are split based on a given delimiter, and composed steady-states are stored as tuples in lists or dictionaries in the target BN structure. Details are

described later in Subsection 5.3.1. The input parsing functionality is stored in module `parse_input.py`.

5.3 Data storing classes

We split the majority of the core functionality into five larger classes. Other classes contain only simple or marginal functionality. In this section, we describe the attributes and methods of five focal classes.

5.3.1 TargetBooleanNetwork

This class corresponds to the target Boolean network $\mathcal{T}$ defined in Section 3.2. It contains a number of desired variables (`num_of_vars`), two sets of input and output gene indices, and steady-states of multiple experiments. Attribute `wt_sinks` is a list of steady-states of wild-type experiments. Finally, `ko_sinks` and `oe_sinks` are two dictionaries whose keys correspond to perturbed gene indices and values to lists of steady-states of given gene knockout resp. over-expression.

The class contains methods for deriving additional regulations from gene-to-gene correlations alongside their validation and construction. The `create_regulation` method decides the direction of a derived regulation. The previous method `is_valid_regulation` ensures that the input regulation does not violate the requirements in both directions. If the derived regulation is consistent with requirements only in one direction, it is clearly added. This happens in cases where one of the genes is defined as input or output. However, if both directions are consistent, the method randomly chooses one of them and creates the regulation instance.

There is only one instance of this class during the entire algorithm run. Objects with reference to this object have their custom `deep_copy` method defined, ensuring that only the reference to this instance is copied.

5.3.2 UpdateFunction

This class represents the update function of a specific gene that is NCF in its regulators. The `gene` attribute specifies the index of the gene to which the update function applies, the three lists `canalyzing`,

`canalyzed` and `regulators` that copy the NCF representation described in Section 4.3.1. The `fixed` attribute corresponds to the set of fixed regulators of a given gene and ensures consistency with the regulatory constraints. More details are given in Section 5.4.

The class provides basic functionality for adding regulators to a specific position associated with generating canalyzing and canalyzed values. It also contains a method for finding meaningful mutations for a given update function, the design of which is described in 9. In addition, it provides a method for selecting a random, meaningful mutation and individual methods for performing the selected mutations. The types of mutations are discussed in Section 4.6. Finally, it contains a method for exporting the NCF into the update function format supported by AEON.

5.3.3 BooleanNetwork

The class represents a Boolean network $\mathcal{B}^{(n)}$. The `functions` attribute contains a list of n update functions such that `functions[i]` corresponds to $f_{i+1} \in \mathcal{F}$. This shift is because the Boolean variables are represented by integers ranging from 0 to $n-1$ in the implementation. The dimension of the network is stored in `target_bn` attribute.

The class contains the `total_num_of_regulations` property, which returns the current number of regulations in the entire network. This property is used in the selection of meaningful mutations. It also includes the `initialise_ncfs` method, which is used when initialising networks of the initial generation. This procedure is described in Section 4.4. The class also contains the `mutate`, `add_regulator`, and `to_aeon_string` methods, whose eponymous methods are then called on specific update functions. The `get_regulated_by` method returns a set of all genes regulated by a given gene. This set is used in the method which detects isolated variables. We describe this procedure in Section 4.5.1. The most important method of this class is the `compute_fitness` method which is used to compute the fitness of a given BN in its specific experiment concerning the target BN. This procedure is described in Section 4.5.

5.3.4 Generation

The class represents the generation of Boolean networks precisely as we defined it in Section 4.4. The `num_of_nets` attribute specifies the number of networks in the generation. The actual list of networks is contained in the `networks` attribute. The `scores` list contains the fitness score of the corresponding network such that `scores[i]` corresponds to ϱ_{i+1} (the shift is due to the fact that we index from 0 in the implementation).

The construction of the initial generation is located in the `utils.py` module. The class contains the method `create_new_generation`, which creates the next generation of networks from the current generation and then calls a method that mutates some networks. The implementation of creating the next generation copies the procedure from Section 4.7. It is worth noting that it is common for the same network from the current generation to be selected for the next generation multiple times. For this reason, a custom deep copy of each selected network is created when inserting it into the new generation. However, the target BN object is not replicated. Finally, the class contains the method `compute_fitness` that calculates the fitness of all the networks in the generation and sets the `scores` attribute. The resulting fitness of each BN is equal to the average fitness across all the network's experiments.

5.3.5 BipartiteGraph

An instance of this class represents a complete bipartite graph, defined in Section 4.5. The attributes `target` and `current` correspond to the vertex sets V_t and V_c from the definition. However, in the implementation, vertices are stored in ordered lists. The 2D matrix `distances` contains the Hamming distances between the vertices of the different sets. The matrix has size $m \times n$ where m is equal to $\|target\|$ and n to $\|current\|$. Consequently, the element `distances[m][n]` contains the Hamming distance between the m-th current and the n-th target steady-state.

The only functionality of this class is the calculation of a given graph's unbalanced minimum weighted assignment. The corresponding method returns a list of tuples of assigned vertices (steady-states)

and the total cost of this assignment, i.e., the sum of Hamming distances of all the assigned tuples of steady-states.

5.4 Mutations and constraints preservation

In this section, we discuss how mutations of individual networks are performed so that regulation constraints and input resp. output variables are preserved.

As we specify in Appendix A, in `main.py`, there are two global constants `NUM_OF_MUT_GENES` (how many genes of each network will be mutated) and `NUM_OF_MUTATIONS` (how many regulators of each selected gene will be mutated). Thus, the total number of mutations performed on each mutated network equals the product of these two constants. The higher the values we set, the more exhaustively the algorithm will move through the space of all solutions. Each `Generation`, `BooleanNetwork`, and `UpdateFunction` class then has its own `mutate` method, whose calls are propagated in the order shown. First, networks are selected from the current generation. As we mentioned, the elite-fitting networks, i.e., those whose fitness is the best among all, are reproduced for the new generation without mutations. This step is implemented in such a way that elite-fitting networks are placed on the first indices of the next generation's `networks` list. The `mutate` method is then called only on networks that are placed after the index of the last elite-fitting network in the `network` list.

The `mutate` method for `BooleanNetwork` then randomly selects a preset number of genes (`NUM_OF_MUT_GENES`) whose update functions will be mutated. However, it only selects among non-input genes. Thus, genes marked as inputs will never be chosen as mutation targets, ensuring compliance with the requirements. Consequently, the `mutate` method of selected genes' update functions is called to mutate them. For each update function, the preset number of regulators (`NUM_OF_MUTATIONS`) that will be mutated is randomly selected again. This time it is selected only among non-output genes. Thus, genes marked as outputs will never be chosen as regulation sources, ensuring compliance with the requirements. When a mutation's source and target genes are selected, a set of meaningful mutations is selected for them. This procedure is described in Section 4.6. This selection pro-

hibits the choice of mutations that would violate the requirements, e.g., the removal or modification of the sign of a fixed regulation. Finally, the randomly chosen meaningful mutation is performed.

6 Evaluation

We evaluated the proposed algorithm on six Boolean models. Two models were taken from *DREAM-3 In sillico challenge* dataset [36]. These models represent subnetworks of the transcriptional regulatory networks of E. coli and Yeast, respectively. Then we randomly generated two scale-free BNs using a scale-free BN generator from [37]. The last two are literature-based models taken from *Biodivine Boolean Models Benchmark Dataset*, available on [38].

We examined how the algorithm worked on each model when some inputs and parameters were changed. We tried different combinations of the number of mutated genes and the number of mutations. We found out that the algorithm achieves the best fitness when the number of mutated genes is 2, and the number of mutations is 2. We also found out that for such small models, the algorithm converges to the ideal solution within 30 iterations and the fitness rarely improves with further iterations. Finally, we found that a correlation threshold of 0.75 is appropriate because it derives the highest proportion of right edges on the considered models. We have not manipulated these parameters during the evaluation.

We tested the performance when given only subsets of true input and output variables or subsets of steady-states of the individual BN's experiments since we do not always know all the network properties from the data. We monitored both structural and dynamic accuracy concerning the target network.

The novel fitness metric defined in Section 4.5 was used to determine the dynamic accuracy. Consequently, we evaluated structural accuracy by monitoring the number of occurrences of individual regulations in the best-fitting networks of multiple algorithm runs.

We distinguished between four types of regulations when evaluating structural accuracy. (i) Regulations that were given as input constraints. They served as a positive control that the algorithm works correctly and preserves the input constraints in all networks. (ii) Regulations that are directly present in the target model. We denote such regulations as *direct* regulations of the model. (iii) Regulations that are not present in the target model. (iv) *Indirect* regulations of the

target model. The regulation is considered to be an indirect regulation of the model if one of the following 4 situations occurs in the model:

- two adjacent direct negative regulations $v_x \dashv v_y$ and $v_y \dashv v_z$ imply positive indirect regulation $v_x \rightarrow v_z$. The reason is that if v_x is significantly expressed, it inhibits the expression of v_y. v_y then does not inhibit v_z. This may indicate a positive regulation from v_x to v_z.

- two adjacent direct positive regulations $v_x \rightarrow v_y$ and $v_y \rightarrow v_z$ imply positive regulation $v_x \rightarrow v_z$. The reason is that if v_x is expressed, it activates v_y. v_y then activates v_z, which may indicate a direct activation from v_x to v_z.

- adjacent direct negative regulation $v_x \dashv v_y$ and direct positive regulation $v_y \rightarrow v_z$ imply indirect negative regulation $v_x \dashv v_z$. The reason is that if v_x is expressed, it inhibits v_y. v_y then does not activate the expression of v_z. This may indicate a negative regulation from v_x to v_z.

- adjacent direct positive regulation $v_x \rightarrow v_y$ and direct positive regulation $v_y \dashv v_z$ imply indirect negative regulation $v_x \dashv v_z$. The reason is that if v_x is expressed, it activates v_y. v_y then inhibits the expression of v_z. This may indicate a negative regulation from v_x to v_z.

Indirect regulations are also often reflected in the expression data. Therefore, we also consider the increased occurrence of these regulations in the inferred networks as a sign that our inference algorithm gives reasonable results.

We targeted smaller models with up to 20 variables to evaluate the algorithm prototype. The evaluation aimed to verify that the chosen inference method achieves meaningful results and that it makes sense to work on improving it. For this reason, we used networks of three different types to detect the algorithm's shortcomings.

In this chapter, we present the evaluated models and give the results of their inference when run in different configurations of the mentioned algorithm's parameters. Finally, we discuss the obtained results.

6.1 Models introduction

This section introduces six models used for the evaluation and lists their key characteristics based on which we decided to use them for evaluation. We used models with different properties to best reveal the shortcomings of our algorithm.

Escherichia coli regulatory subnetwork. This network is a subnetwork of the Escherichia coli transcriptional regulatory network stored in RegulonDB [39]. The network contains 10 variables and 12 regulations. Three variables are input, and five are output.

Yeast regulatory subnetwork. This network is a subnetwork of the yeast transcriptional regulatory network that was extracted from [40]. The network contains 10 variables and 11 regulations. Three variables are input, and three are output.

Only the network topologies of both models were available; update functions were not specified. Therefore, we randomly specified all the update functions such that they are in NCF. Both models are characterized by a relatively small total number of regulations and a small number of internal variables.

Scale-free random networks. We generated two scale-free Boolean networks using a random BN generator from [37]. The generator generates random scale-free topology and also NCF update functions. These models have a characteristic topology [41] and are relatively dense.

Body segmentation in Drosophila model. The first literature-based model originally collected from [42] describes a body segmentation in Drosophila melanogaster. The model contains 17 variables and 29 regulations. Three variables are inputs, and four are outputs.

Regulation of L-arabinose operon model. The second literature-based model collected from [43] describes a regulation of L-arabinose operon. The model contains 13 variables and 18 regulations. Four variables are inputs, and one is output.

Both literature-based models contain also update functions. However, some of them cannot be represented as NCFs. Therefore, we consider it beneficial to find out how well our algorithm can deal with the inference of a model whose update functions it is not able to represent.

To generate the expression data of target networks, we used the AEON sink state enumerator to obtain all *explicit steady-states* for all possible experiments of each network. We discuss the considered experiment types in Section 2.4. Explicit steady-states of a given experiment are obtained by analysing the model for all combinations of its input variables values. Details of generating the explicit steady-states are described in Apendix A.

6.2 Basic parameter setup

First, we performed 100 runs of the algorithm on a basic parameter setup to generally understand its performance. We recorded the highest fitness achieved within single runs. The concrete parameter setup is listed in the following.

- full dataset of network's explicit steady-states,
- number of specified regulations in constraints: 2
- specified inputs: all,
- specified outputs: all.

Figure 6.1 shows the obtained fitness distributions of all six evaluated models. We can see that the algorithm achieved satisfactory dynamic accuracy on all six models. However, the fitness of the other four relatively dense models is lower compared to sparse E. coli and Yeast models.

Regarding structural accuracy, we have also received satisfactory results for the first two models. Figure 6.2 and Figure 6.3 show the occurrence of individual regulations within 100 runs of the algorithm for the E. coli resp. Yeast model. The y-axis shows the individual regulations ("->" for activation and "-|" for inhibition). The x-axis shows the total occurrence of a given regulation in best-fitting networks within 100 runs of the algorithm. Plots show the 30 most frequently occurring regulations.

For the E. coli and Yeast models, many of the target model's direct or indirect edges are the most frequent ones. We can see that all 12 total direct regulations of the E. coli model were in the 24 most frequent

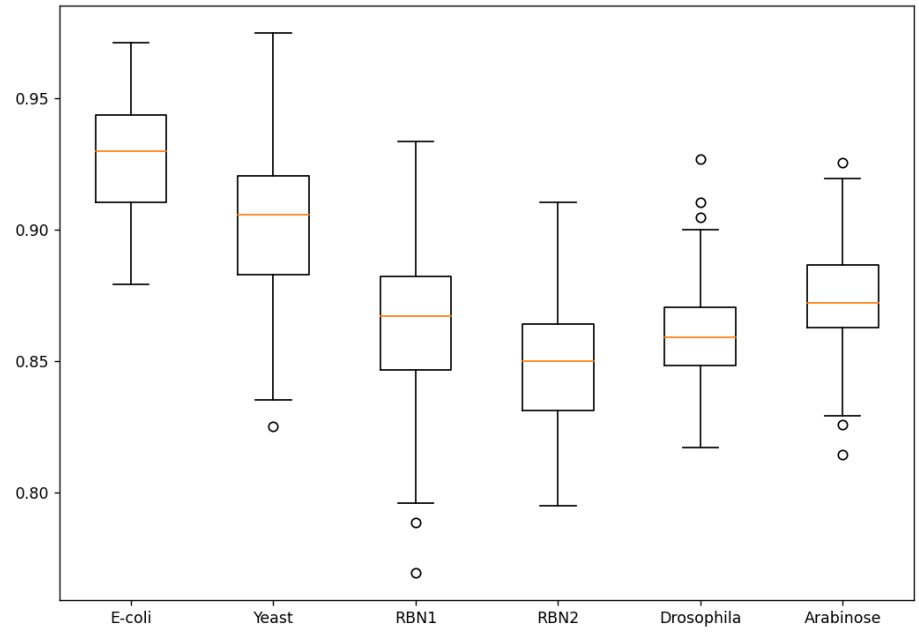

Figure 6.1: Dynamic accuracy of the basic parameter setup. The plot shows the distribution of the dynamic accuracy, including the best-fitting networks from individual runs for each model. The inference of sparse E. coli and Yeast models performed better than the other four dense models. However, we can see some outliers that exhibit satisfying dynamic accuracy for the four dense models.

inferred regulations. Additionally, the other 4 indirect regulations were inferred quite frequently too.

We can also see that 8 out of 11 total regulations of the Yeast model are statistically frequent. Five of them are even among the most frequently inferred regulations, except for constraints. Moreover, 8 other indirect regulations are also present in the plot.

The fact that some regulations that are not in the target model are inferred relatively often may be due to that we do not monitor the position of the regulator in the NCF update function. The position of the regulator in the NCF significantly affects its real effect. Regulators

located at the beginning of the NCF have a higher priority in the evaluation, and therefore their effect is more significant.

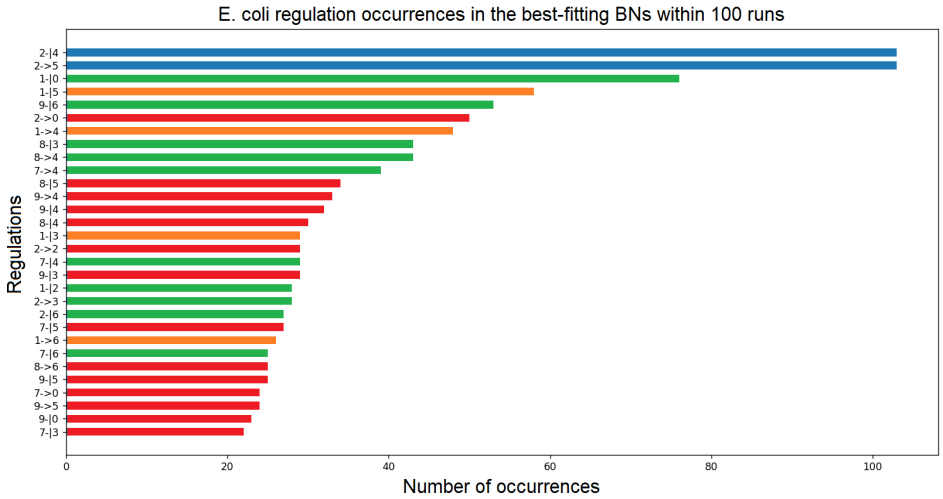

Figure 6.2: Structural accuracy of E. coli model while all the sinks specified. The plot shows the number of occurrences of the 30 most frequent regulations in best-fitting BNs within 100 runs of the algorithm according to the target model of E. coli. All the explicit sinks of the target E. coli model were specified on the input. The y-axis shows the most frequent inferred regulations, and the x-axis shows the number of their occurrences within all 100 runs of the algorithm. Ten out of twelve direct edges (in green) were inferred frequently, and the rest two were passed as the input constraints (in blue). Moreover, we can see that four other indirect regulations (in orange) are also quite frequent. The remaining regulations (in red) are not present in the target model.

We also created plots showing the structural accuracy of the remaining four models. However, analysis of the structural accuracy of the best-fitting BNs from each of the 100 runs showed no significant inference of any specific regulation. This can be explained by the fact that it is difficult to infer the regulations of such dense models, especially when our fitness metric is based on dynamics, not the structure. Another reason is that we do not possess experiments with multiple simultaneously perturbed genes, which better reveal the dependencies

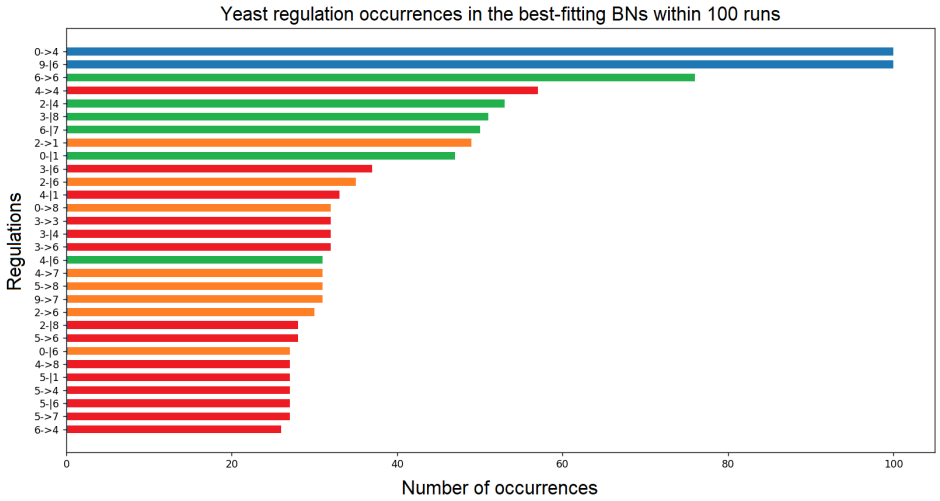

Figure 6.3: Structural accuracy of Yeast model while all the sinks specified. The plot shows the number of occurrences of the 30 most frequent regulations in best-fitting BNs within 100 runs of the algorithm according to the target model of Yeast. All the explicit sinks of the target Yeast model were specified on the input. The y-axis shows the most frequent inferred regulations, and the x-axis shows the number of their occurrences within all 100 runs of the algorithm. Six out of eleven direct edges (in green) were inferred frequently, and the other two were passed as the input constraints (in blue). Moreover, we can see that eight other indirect regulations (in orange) are also quite frequent. The remaining regulations (in red) are not present in the target model.

in such dense models. Plots of the frequency occurrence of regulations for the rest models are shown in Apendix B.1.

6.3 Steady-state data reduction

On the inferred models, using basic settings of algorithm parameters and input data, we noticed an interesting anomaly. Since we included all explicit sinks in the input data, the inferred models tend to contain a large number of input nodes. The increasing number of inputs of the model causes an exponential increase in the number of its explicit

sinks. As a result, such a model can better cover a high number of steady-states in the input data since the fitness metric does not penalise the overhang of the currently evaluated model's steady-states over the target ones. To understand this, see Subsection 4.5.2. The total number of variables is always computed as the number of variables in target steady-states. Therefore, to try to prevent this phenomenon, we decided to reduce the number of explicit sinks in the input data to half since most explicit sinks have the same values for the non-input variables of the network and differ only in the setting of the input values. Apart from this change, we use the same parameters as before.

Figure B.5 (Appendix) shows no significant difference compared to the inference from the full dataset. We include also the structural accuracy plots of data-reduced inferences in Apendix B.2. However, they do not show any significant improvement too.

6.4 Increase of constraints for dense models

To increase the inference accuracy for dense models, we tried to increase the number of their constraints. For each model, we randomly selected approximately one-third of the total number of regulations of the target model. We put the selected regulations as the constraints.

Paradoxically, it turned out that a higher number of constraints lowered the average inference accuracy. We explain this fact by overly restricting the freedom of inference by static constraints, in particular, using a fitness metric based on dynamics.

Figure B.12 (Appendix) shows the dynamic accuracy of four dense models. We can see a decrease in dynamic accuracy compared to the previous two setups. The decrease in structural precision is even more evident. Individual graphs of structural accuracy are shown in Appendix B.3.

6.5 Input and output reduction

Finally, we tried to test the accuracy of inference with only partial specification of inputs and outputs. We likely know at least a fraction of the inputs and outputs in GRN inference. Therefore, we have reduced the number of inputs and outputs to only about half. The results do not

show a significant decrease in the dynamic accuracy of the algorithm shown in Figure B.17 (Appendix). However, the structural accuracy has significantly decreased. Individual graphs of structural accuracy are shown in Appendix B.4.

6.6 Discussion

In the evaluation of the algorithm, we focused on how the inference accuracy responds to changes in the settings of individual parameters.

In particular, we found out that when searching the space of Boolean networks that satisfy the constraints, it is appropriate to perform only a few mutations at once in each iteration. We then found out that for such small networks, the algorithm usually converged to a local maximum within 30 iterations and did not tend to improve further. We treated these parameters as constants and did not manipulate them further.

Then we individually investigated how the algorithm performs when changing some parameters or inputs. Specifically, we looked at the following:

- Reduction of the number of explicit steady-states by half to prevent the algorithm from inferring networks with a large number of input variables to increase fitness. However, this change did not have any observable effect. Therefore a change in the format of the input steady-states will be necessary for the future. In addition to dropping the explicit sinks, this change is likely associated with adding the dynamic properties mentioned in Section 3.1.2 and penalising the overlap of the analysed model's number of steady-states over the target BN.
- More constraints for dense models so that they are proportional to the number of total regulations of the target model. Paradoxically, this increase brought a decrease in inference accuracy. We explain this by the fact that we imposed too strong constraints on the resulting model. The algorithm apparently does not search the space in the right way and cannot find a suitable network to fit the data with such a strong constraint on the solution space. A change in the mutation-selection strategy will probably be necessary for the future.

- Specification of only a subset of the inputs and outputs of the network. This change produced almost no difference in the fitness metric value based on dynamics. However, the structural accuracy of the inferred networks was significantly reduced.

We consider the evaluation of the prototype algorithm to be useful. It revealed the fundamental shortcomings of the algorithm together with their reasons. On the basis of its results, we propose some fundamental changes that we would like to apply in the future.

7 Conclusion

The work aimed to identify the limitations of current automatic inference tools and to design a framework that would try to propose improvements to some of these limitations. In Chapter 1, we discussed the detailed definitions of key properties of Boolean networks. We used these insights to enumerate static and dynamic properties of BNs that we might be wanted to require from the target BN based on prior knowledge of the underlying network.

In Chapter 2, we have discussed different types of expression data and experiments. Based on the collected information, we concluded that it would be appropriate for the algorithm to work with single-cell data, which provides much more comprehensive information about network behaviour. Depending on this decision, we were motivated to construct an inference method working only on the basis of asynchronous semantics because it is closely related to the concept of single-cell data.

In Chapter 3 and Chapter 4, we then described the designed framework in detail. The presented framework adopts the popular concept of genetic programming together with attractor analysis of Boolean networks in AEON [35]. As part of the algorithm, we presented a novel fitness-evaluating metric based on the unbalanced minimum weighted assignment of a complete bipartite graph. The metric determines the similarity of two BNs based on their steady-states.

We implemented the prototype of the described framework in Python and performed its evaluation for six models with different properties to increase the chance of detecting the algorithm's shortcomings. We tested some chosen combinations of input data and algorithm parameters. We achieved satisfactory results on smaller and sparser models in both structural and dynamic accuracy concerning the target model. However, the algorithm had significant problems with inferring the underlying network structure for larger and denser models. However, it maintained a fairly solid dynamic accuracy to the target models.

Based on the evaluation results, we propose the following changes that might improve the asynchronous-based inference method in the future.

- To abandon the enumeration of explicit steady-states of the model but to introduce a different representation of steady-states. The current representation is inappropriate because it pushes the inference method to infer networks with a large number of inputs, which then cover the target steady-states. The new representation should reflect the reachability of attractors from specific states or presence in the basins of an attraction. Alternatively, we plan to extend the input data format to support multi-state attractors.

- To change the way mutations, as well as source and target variables, are selected. The current method does not properly traverse the space of available solutions and does not provide significant results even when a larger number of constraints are provided. The new method should first select the type of mutation and then select the source and target variables of the mutation based on the selected type.

- To improve the current fitness-evaluating metric to penalise the overhang of the currently evaluated model's steady-states over the target steady-states.

- To implement parallel fitness evaluation for individual networks of a given generation since individual fitness evaluations are independent. This step will significantly improve the scalability of the algorithm.

Bibliography

1. KITANO, Hiroaki. Systems biology: toward system-level understanding of biological systems. In: *Foundations of systems biology*. 2001, pp. 1–36.
2. TRINH, Hung-Cuong; KWON, Yung-Keun. A novel constrained genetic algorithm-based Boolean network inference method from steady-state gene expression data. *Bioinformatics*. 2021, vol. 37, no. Supplement_1, pp. 383–391.
3. MUNOZ, Stalin; CARRILLO, Miguel; AZPEITIA, Eugenio; ROSENBLUETH, David A. Griffin: A tool for symbolic inference of synchronous boolean molecular networks. *Frontiers in genetics*. 2018, vol. 9, p. 39.
4. SHI, Ning; ZHU, Zexuan; TANG, Ke; PARKER, David; HE, Shan. ATEN: And/Or tree ensemble for inferring accurate Boolean network topology and dynamics. *Bioinformatics*. 2020, vol. 36, no. 2, pp. 578–585.
5. ALTAY, Gökmen; EMMERT-STREIB, Frank. Inferring the conservative causal core of gene regulatory networks. *BMC systems biology*. 2010, vol. 4, no. 1, pp. 1–13.
6. SILVESCU, Adrian; HONAVAR, Vasant. Temporal boolean network models of genetic networks and their inference from gene expression time series. *Complex systems*. 2001, vol. 13, no. 1, pp. 61–78.
7. KAUFFMAN, Stuart A. Metabolic stability and epigenesis in randomly constructed genetic nets. *Journal of theoretical biology*. 1969, vol. 22, no. 3, pp. 437–467.
8. GAO, Shuhua; SUN, Changkai; XIANG, Cheng; QIN, Kairong; LEE, Tong Heng. Learning asynchronous boolean networks from single-cell data using multiobjective cooperative genetic programming. *IEEE Transactions on Cybernetics*. 2022, vol. 52, no. 5, pp. 2916–2930.

9. HEMEDAN, Ahmed Abdelmonem; NIARAKIS, Anna; SCHNEIDER, Reinhard; OSTASZEWSKI, Marek. Boolean modelling as a logic-based dynamic approach in systems medicine. *Computational and Structural Biotechnology Journal*. 2022, vol. 20, pp. 3161–3172.

10. KAUFFMAN, Stuart; PETERSON, Carsten; SAMUELSSON, Björn; TROEIN, Carl. Genetic networks with canalyzing Boolean rules are always stable. *Proceedings of the National Academy of Sciences*. 2004, vol. 101, no. 49, pp. 17102–17107.

11. KAUFFMAN, Stuart; PETERSON, Carsten; SAMUELSSON, Björn; TROEIN, Carl. Random Boolean network models and the yeast transcriptional network. *Proceedings of the National Academy of Sciences*. 2003, vol. 100, no. 25, pp. 14796–14799.

12. MURRUGARRA, David; LAUBENBACHER, Reinhard. Regulatory patterns in molecular interaction networks. *Journal of Theoretical Biology*. 2011, vol. 288, pp. 66–72.

13. KARLEBACH, Guy. A Novel Algorithm for the Maximal Fit Problem in Boolean Networks. *arXiv preprint arXiv:1505.07335*. 2015.

14. BENEŠ, Nikola; BRIM, Luboš; PASTVA, Samuel; POLÁČEK, Jakub; ŠAFRÁNEK, David. Formal Analysis of Qualitative Long-Term Behaviour in Parametrised Boolean Networks. In: *Formal Methods and Software Engineering*. Springer International Publishing, 2019, pp. 353–369.

15. GARG, Abhishek; DI CARA, Alessandro; XENARIOS, Ioannis; MENDOZA, Luis; DE MICHELI, Giovanni. Synchronous versus asynchronous modeling of gene regulatory networks. *Bioinformatics*. 2008, vol. 24, no. 17, pp. 1917–1925.

16. ZHENG, Desheng; YANG, Guowu; LI, Xiaoyu; WANG, Zhicai; LIU, Feng; HE, Lei. An efficient algorithm for computing attractors of synchronous and asynchronous Boolean networks. *PloS one*. 2013, vol. 8, no. 4, e60593.

17. BRAZMA, Alvis; PARKINSON, Helen; SARKANS, Ugis; SHOJATALAB, Mohammadreza; VILO, Jaak; ABEYGUNAWARDENA, Niran; HOLLOWAY, Ele; KAPUSHESKY, Misha; KEMMEREN, Patrick; LARA, Gonzalo Garcia; OEZCIMEN, Ahmet; ROCCA-SERRA, Philippe; SANSONE, Susanna-Assunta. ArrayExpress — a public repository for microarray gene expression data at the EBI. *Nucleic Acids Research*. 2003, vol. 31, no. 1, pp. 68–71.

18. CLOUGH, Emily; BARRETT, Tanya. The Gene Expression Omnibus Database. In: *Statistical Genomics: Methods and Protocols*. Springer, 2016, pp. 93–110.

19. PRATAPA, Aditya; JALIHAL, Amogh P; LAW, Jeffrey N; BHARADWAJ, Aditya; MURALI, TM. Benchmarking algorithms for gene regulatory network inference from single-cell transcriptomic data. *Nature methods*. 2020, vol. 17, no. 2, pp. 147–154.

20. SCHAFFTER, Thomas; MARBACH, Daniel; FLOREANO, Dario. GeneNetWeaver: in silico benchmark generation and performance profiling of network inference methods. *Bioinformatics*. 2011, vol. 27, no. 16, pp. 2263–2270.

21. KUKSIN, Maria; MOREL, Daphné; AGLAVE, Marine; DANLOS, François-Xavier; MARABELLE, Aurélien; ZINOVYEV, Andrei; GAUTHERET, Daniel; VERLINGUE, Loïc. Applications of single-cell and bulk RNA sequencing in onco-immunology. *European Journal of Cancer*. 2021, vol. 149, pp. 193–210.

22. HAQUE, Ashraful; ENGEL, Jessica; TEICHMANN, Sarah A; LÖNNBERG, Tapio. A practical guide to single-cell RNA-sequencing for biomedical research and clinical applications. *Genome medicine*. 2017, vol. 9, no. 1, pp. 1–12.

23. JEW, Brandon; ALVAREZ, Marcus; RAHMANI, Elior; MIAO, Zong; KO, Arthur; GARSKE, Kristina M; SUL, Jae Hoon; PIETILÄINEN, Kirsi H; PAJUKANTA, Päivi; HALPERIN, Eran. Accurate estimation of cell composition in bulk expression through robust integration of single-cell information. *Nature communications*. 2020, vol. 11, no. 1, pp. 1–11.

24. WANG, Xuran; PARK, Jihwan; SUSZTAK, Katalin; ZHANG, Nancy R; LI, Mingyao. Bulk tissue cell type deconvolution with multi-subject single-cell expression reference. *Nature communications*. 2019, vol. 10, no. 1, pp. 1–9.

25. WANG, Chunxiang; GAO, Xin; LIU, Juntao. Impact of data preprocessing on cell-type clustering based on single-cell RNA-seq data. *BMC bioinformatics*. 2020, vol. 21, no. 1, p. 440.

26. HERTEL, Stefanie; BRETTSCHNEIDER, Christian; AXMANN, Ilka M. Revealing a two-loop transcriptional feedback mechanism in the cyanobacterial circadian clock. *PLoS computational biology*. 2013, vol. 9, no. 3, e1002966.

27. ZHENG, Shijie C; STEIN-O'BRIEN, Genevieve; AUGUSTIN, Jonathan J; SLOSBERG, Jared; CAROSSO, Giovanni A; WINER, Briana; SHIN, Gloria; BJORNSSON, Hans T; GOFF, Loyal A; HANSEN, Kasper D. Universal prediction of cell-cycle position using transfer learning. *Genome biology*. 2022, vol. 23, no. 1, p. 41.

28. CUTILLO, Luisa; BOUKOUVALAS, Alexis; MARINOPOULOU, Elli; PAPALOPULU, Nancy; RATTRAY, Magnus. OscoNet: inferring oscillatory gene networks. *BMC bioinformatics*. 2020, vol. 21, no. 10, p. 351.

29. HUVAR, Ondřej. *Symbolic Model Checking of Hybrid CTL*. 2022. MA thesis. Masaryk University, Faculty of Informatics.

30. TRANCHEVENT, Léon-Charles; CAPDEVILA, Francisco Bonachela; NITSCH, Daniela; DE MOOR, Bart; DE CAUSMAECKER, Patrick; MOREAU, Yves. A guide to web tools to prioritize candidate genes. *Briefings in bioinformatics*. 2011, vol. 12, no. 1, pp. 22–32.

31. KAUFFMAN, Stuart A et al. *The origins of order: Self-organization and selection in evolution*. Oxford University Press, USA, 1993.

32. KOZA, John R; KEANE, Martin A; STREETER, Matthew J; MYDLOWEC, William; YU, Jessen; LANZA, Guido. *Genetic programming IV: Routine human-competitive machine intelligence*. Springer, 2006.

33. GAO, Shuhua; XIANG, Cheng; SUN, Changkai; QIN, Kairong; LEE, Tong Heng. Efficient Boolean modeling of gene regulatory networks via random forest based feature selection and best-fit extension. In: *2018 IEEE 14th International Conference on Control and Automation (ICCA)*. 2018, pp. 1076–1081.

34. FOUNDATION, Python Software. *biodivine-aeon 0.1.1* [`https://pypi.org/project/biodivine-aeon/`]. Python Package Index, 2022 [visited on 2022-12-05].

35. BENEŠ, Nikola; BRIM, Luboš; KADLECAJ, Jakub; PASTVA, Samuel; ŠAFRÁNEK, David. AEON: attractor bifurcation analysis of parametrised boolean networks. In: *International Conference on Computer Aided Verification*. 2020, vol. 12224, pp. 569–581. Lecture Notes in Computer Science.

36. MARBACH, Daniel. *DREAM network inference challenge* [`https://gnw.sourceforge.net/dreamchallenge.html`]. GeneNetWeaver, 2014 [visited on 2022-12-05].

37. STANKO, Oto. *Improving Efficiency of Boolean Networks Inference*. 2022. MA thesis. Masaryk University, Faculty of Informatics.

38. PASTVA, Samuel. *Biodivine Boolean Models (BBM) Benchmark Dataset* [`https://github.com/sybila/biodivine-boolean-models`]. GitHub, 2022 [visited on 2022-12-05].

39. TIERRAFRÍA, Víctor H; RIOUALEN, Claire; SALGADO, Heladia; LARA, Paloma; GAMA-CASTRO, Socorro; LALLY, Patrick; GÓMEZ-ROMERO, Laura; PEÑA-LOREDO, Pablo; LÓPEZ-ALMAZO, Andrés G; ALARCÓN-CARRANZA, Gabriel, et al. RegulonDB 11.0: Comprehensive high-throughput datasets on transcriptional regulation in Escherichia coli K-12. *Microbial Genomics*. 2022, vol. 8, no. 5, p. 000833.

40. SHEN-ORR, Shai S; MILO, Ron; MANGAN, Shmoolik; ALON, Uri. Network motifs in the transcriptional regulation network of Escherichia coli. *Nature genetics*. 2002, vol. 31, no. 1, pp. 64–68.

41. BARABÁSI, Albert-László; BONABEAU, Eric. Scale-Free Networks. *Scientific American*. 2003, vol. 288, no. 5, pp. 60–69.

42. MARQUES-PITA, Manuel; ROCHA, Luis M. Canalization and control in automata networks: body segmentation in Drosophila melanogaster. *PloS one*. 2013, vol. 8, no. 3, e55946.

43. JENKINS, Andy; MACAULEY, Matthew. Bistability and Asynchrony in a Boolean Model of the L-arabinose Operon in Escherichia coli. *Bulletin of mathematical biology*. 2017, vol. 79, no. 8, pp. 1778–1795.

A Repository content

The whole thesis package also contains the GitHub repository with the source code of the implemented algorithm, the additional script used for the data generation, and the models used for the evaluation. In this chapter, we provide an overview of the structure and the content of the repository, together with some additional technical details related to the implementation of the algorithm.

The repository has two main folders: *data* and *src*.

There are three more subfolders in data: (i) *target_networks* contains the six models in .aeon format used for the evaluation, (ii) *input_networks* contains two subfolders with files with explicit steady-states of the individual models used in the evaluation and files for specifying the input constraints, (iii) *out_dir* is the default directory for inserting outputs from individual computations.

The *src* directory contains: (i) the source code of the inference algorithm, (ii) the additional script *sink_state_enumerator.py* used for the explicit sinks generation, and (iii) the binary *sink_state_enumerator.exe*.

A.1 Parameters of the algorithm

Module *main.py* provides an entry point to the whole inference algorithm. This section provides the documentation of its included global constants and parameters.

Several global constants can be set for algorithm configuration, namely:

- `DELIMITER` - a character that separates input matrices' elements if set to `None`, whitespace characters are set as the delimiter,
- `THRESHOLD` - the real number in $[0; 1]$ that determines when the gene-to-gene correlation is considered to be strong enough to indicate potential regulation between these genes,
- `ELITE_RATIO` - the real number in $[0; 1]$ that determines a percentage of best-fitting BNs from an current generation to be automatically selected for the next generation,
- `NUM_OF_NETWORKS` - number of networks in each generation of the genetic algorithm,

- `NUM_OF_MUT_GENES` - number of randomly selected genes that are mutated in each BN of an current generation,
- `NUM_OF_MUTATIONS` - number of mutations performed on each selected mutated gene,
- `MAX_ITERATION` - number of iterations of the genetic algorithm to be performed,
- `MAX_FITNESS` - real number in $[0; 1]$ that determines a satisfying fitness. If such fitness is reached, the algorithm ends immediately.

In addition, more arguments have to be passed to the `main` function, namely:

- `steady_state_matrix_path` - total path to the file with steady-state data,
- `input_constraints_path` - total path to the file with the input constraints,
- `num_of_variables` - number of variables of the original BN,
- `output_path` - total path to the directory where the output files with the inferred BNs will be created,
- `inputs` - set of input variable indices,
- `outputs` - set of output variable indices,
- `net_index` - attribute for sequential execution that determines the ID of particular runs of the algorithm.

A.2 Explicit sinks enumerator

Script *explicit_sinks_enumerator.py* enumerates all explicit model sinks stored in the specified file in .aeon format and outputs them to the specified .tsv file. The sinks are enumerated for both wild-type and mutant-types of all genes. The entry function of the script is `explicit_sinks_enumerator`.

For the enumeration itself, the binary *sink-state-enumerator.exe* created by the authors of the AEON tool is used. The output is then parsed and stored in the output file as Boolean vectors with the values of the individual variables in a fixed order. This order is given by

the indices of the variables in the input model. The model must have variables named in the form "v_index" where the index ranges from 0 to the number of genes in the network - 1. The model also cannot contain any isolated variables.

To create a model with a perturbed gene, all regulations of this gene are set to *unobservable*. This ensures that the occurrence of the source variables of such regulations is not required in the update function of the perturbed gene. The update function is then set to "true" or "false", depending on the type of experiment.

In this way, the explicit sinks of the individual experiments are printed under each other to a single output file.

B Supplementary results

In this chapter, we show the additional plots supporting the evaluation results.

B.1 Basic parameter setup

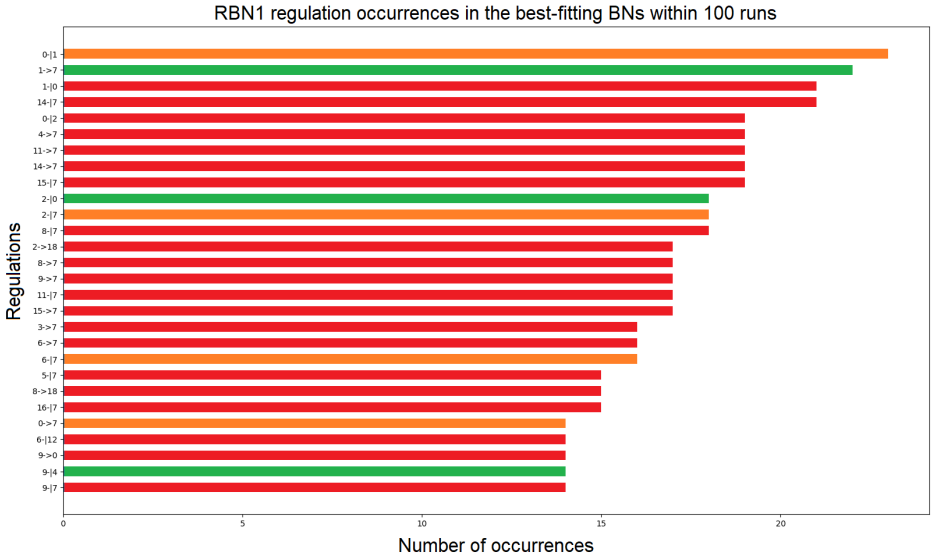

Figure B.1: Structural accuracy of RBN1 model with all the sinks specified. The plot shows the number of occurrences of the 28 most frequent regulations in best-fitting BNs from each of 100 runs of the algorithm according to the target model of RBN1. We omitted the constraints for the sake of the readability of the plot. All the explicit sinks of the target RBN1 model were specified on the input. The y-axis shows the most frequent inferred regulations, and the x-axis shows the number of occurrences within all 100 runs of the algorithm. Three direct edges (in green) were inferred frequently, and the other two were passed as input constraints. Additionally, we can see that 4 other indirect regulations (in orange) are also quite frequent. The remaining regulations (in red) are not present in the target model.

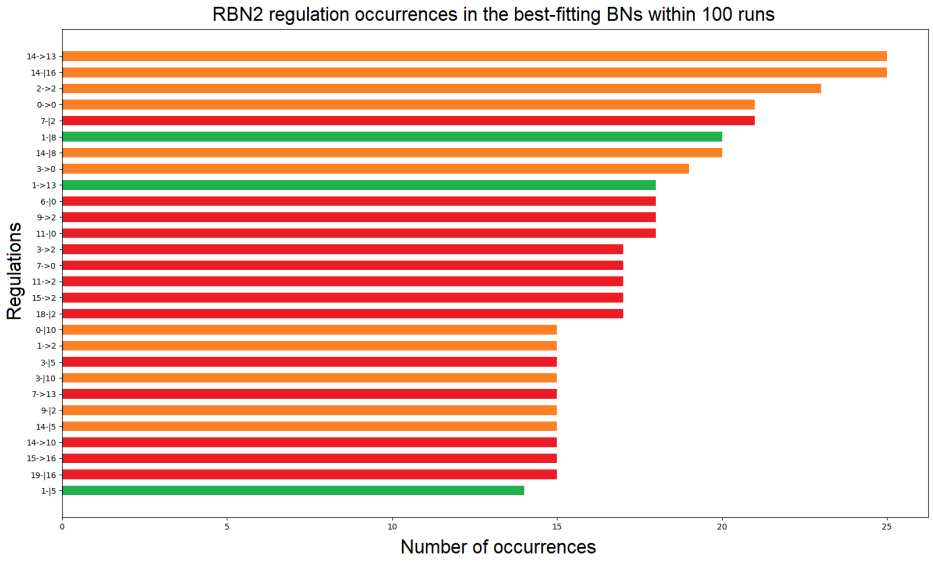

Figure B.2: Structural accuracy of RBN2 model with all the sinks specified. The plot shows the number of occurrences of the 28 most frequent regulations in best-fitting BNs from each of 100 runs of the algorithm according to the target model of RBN2. We omitted the constraints for the sake of the readability of the plot. All the explicit sinks of the target RBN2 model were specified on the input. The y-axis shows the most frequent inferred regulations, and the x-axis shows the number of occurrences within all 100 runs of the algorithm. Three direct edges (in green) were inferred frequently, and the other two were passed as the input constraints. Additionally, we can see that 11 other indirect regulations (in orange) are also quite frequent. The remaining regulations (in red) are not present in the target model.

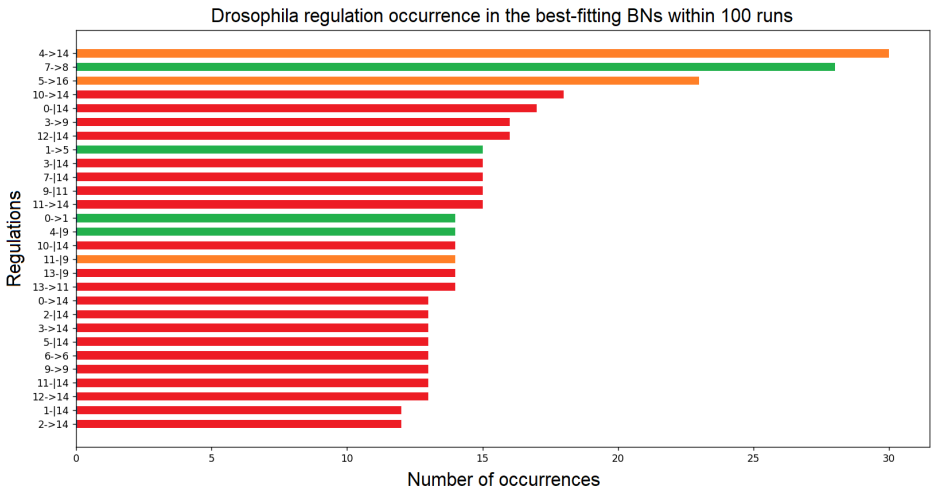

Figure B.3: Structural accuracy of RBN2 model with all the sinks specified. The plot shows the number of occurrences of the 28 most frequent regulations in best-fitting BNs from each of 100 runs of the algorithm according to the target model of RBN1. We omitted the constraints for the sake of the readability of the plot. All the explicit sinks of the target RBN1 model were specified on the input. The y-axis shows the most frequent inferred regulations, and the x-axis shows the number of occurrences within all 100 runs of the algorithm. Three direct edges (in green) were inferred frequently, and the other two were passed as the input constraints. Additionally, we can see that 11 other indirect regulations (in orange) are also quite frequent. The remaining regulations (in red) are not present in the target model.

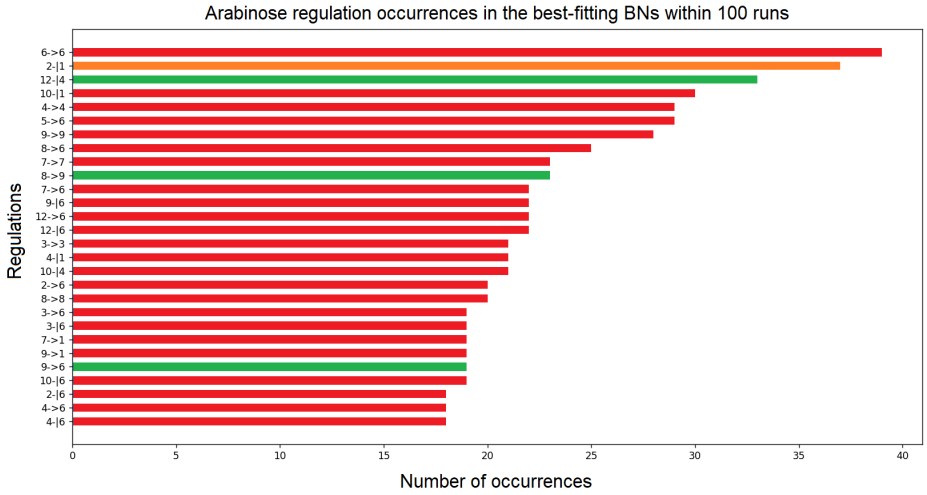

Figure B.4: Structural accuracy of Arabinose model with all the sinks specified. The plot shows the number of occurrences of the 28 most frequent regulations in best-fitting BNs from each of 100 runs of the algorithm according to the target model of regulation of L-arabinose operon. We omitted the constraints for the sake of the readability of the plot. All the explicit sinks of the target Arabinose model were specified on the input. The y-axis shows the most frequent inferred regulations, and the x-axis shows the number of occurrences within all 100 runs of the algorithm. Four direct edges (in green) were inferred frequently, and the other two were passed as input constraints. Additionally, we can see that 3 other indirect regulations (in orange) are also quite frequent. The remaining regulations (in red) are not present in the target model.

B.2 Steady-state data reduction

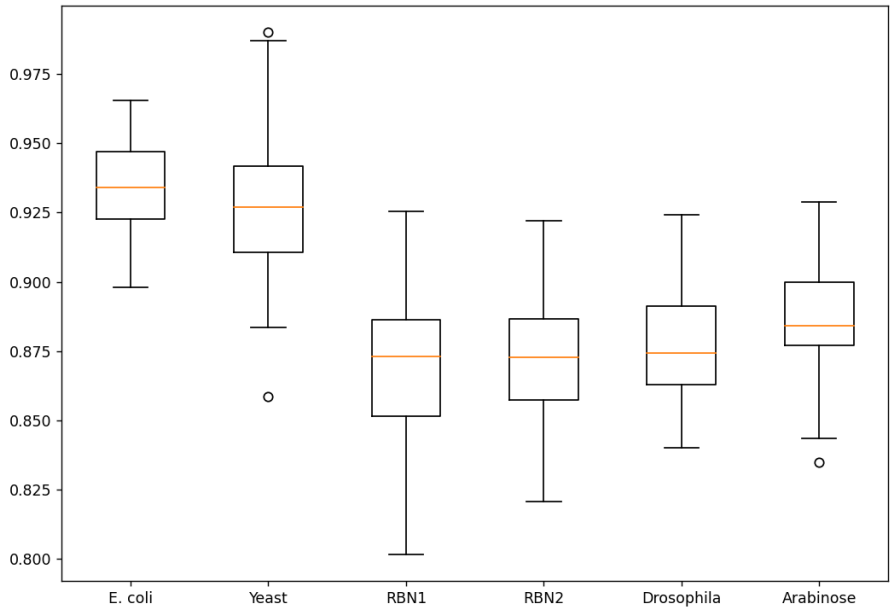

Figure B.5: Dynamic accuracy of the setup with 50% of sinks specified. The plot shows the distribution of the dynamic accuracy, including the best-fitting networks from individual runs for each model. The inference of sparse E. coli and Yeast models performed better than the other four dense models. We can see no significant difference from the dynamic accuracy of the basic setup.

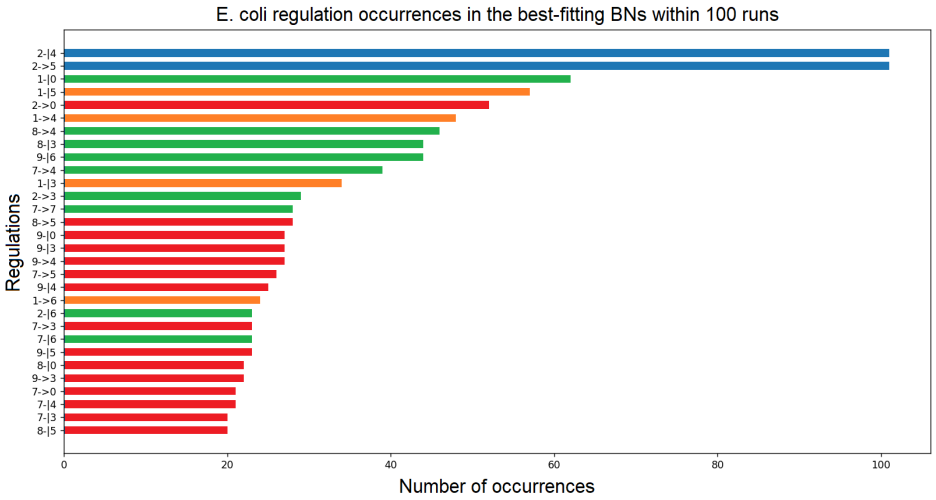

Figure B.6: Structural accuracy of E. coli model with 50% of sinks specified. The plot shows the number of occurrences of the 30 most frequent regulations in best-fitting BNs from each of 100 runs of the algorithm according to the target model of E. coli. Fifty percent of the explicit sinks of the target E. coli model were specified on the input. The y-axis shows the most frequent inferred regulations, and the x-axis shows the number of occurrences within all 100 runs of the algorithm. Nine direct edges (in green) were inferred frequently, and the other two were passed as the input constraints (in blue). Additionally, we can see that 4 other indirect regulations (in orange) are also quite frequent. The remaining regulations (in red) are not present in the target model.

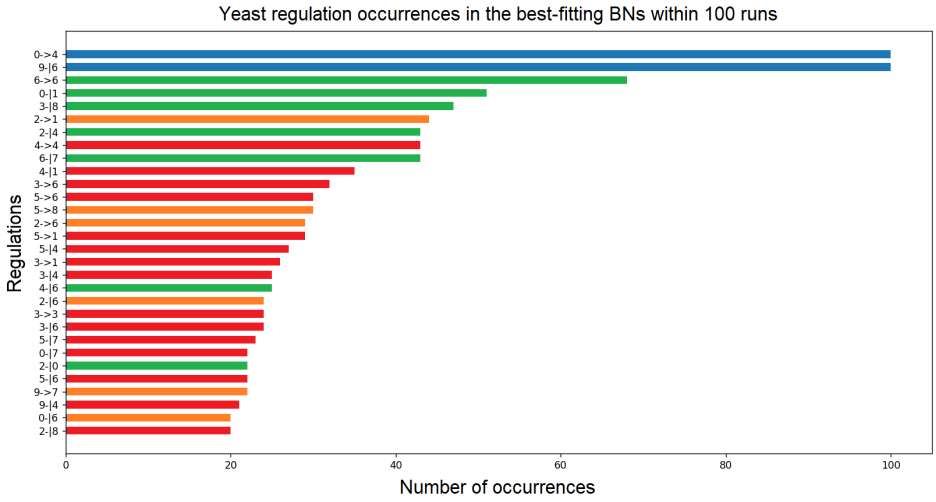

Figure B.7: Structural accuracy of Yeast model with 50% of sinks specified. The plot shows the number of occurrences of the 30 most frequent regulations in best-fitting BNs from each of 100 runs of the algorithm according to the target model of Yeast. Fifty percent of the explicit sinks of the target Yeast model were specified on the input. The y-axis shows the most frequent inferred regulations, and the x-axis shows the number of occurrences within all 100 runs of the algorithm. Seven direct edges (in green) were inferred frequently, and the other two were passed as the input constraints (in blue). Additionally, we can see that 6 other indirect regulations (in orange) are also quite frequent. The remaining regulations (in red) are not present in the target model.

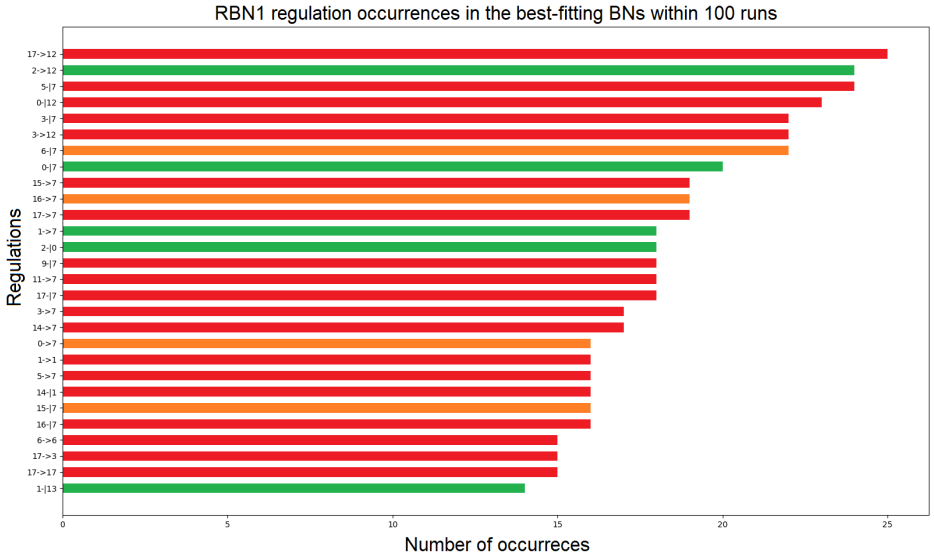

Figure B.8: Structural accuracy of RBN1 model while 50% of sinks specified. The plot shows the number of occurrences of the 28 most frequent regulations in best-fitting BNs from each of 100 runs of the algorithm according to the target RBN1 model. We omitted the constraints for the sake of the readability of the plot. Fifty percent of the explicit sinks of the target RBN1 model were specified on the input. The y-axis shows the most frequent inferred regulations, and the x-axis shows the number of occurrences within all 100 runs of the algorithm. Five direct edges (in green) were inferred frequently, and the other two were passed as the input constraints. Additionally, we can see that 4 other indirect regulations (in orange) are also quite frequent. The remaining regulations (in red) are not present in the target model.

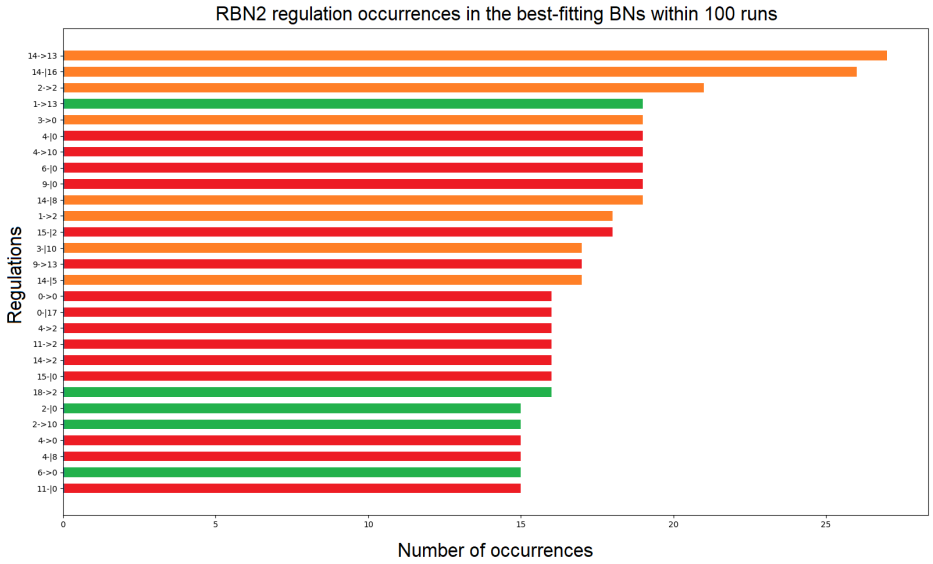

Figure B.9: Structural accuracy of RBN2 model with 50% of sinks specified. The plot shows the number of occurrences of the 28 most frequent regulations in best-fitting BNs from each of 100 runs of the algorithm according to the target RBN2 model. We omitted the constraints for the sake of the readability of the plot. Fifty percent of the explicit sinks of the target RBN2 model were specified on the input. The y-axis shows the most frequent inferred regulations, and the x-axis shows the number of occurrences within all 100 runs of the algorithm. Five direct edges (in green) were inferred frequently, and the other two were passed as input constraints. Additionally, we can see that 8 other indirect regulations (in orange) are also quite frequent. The remaining regulations (in red) are not present in the target model.

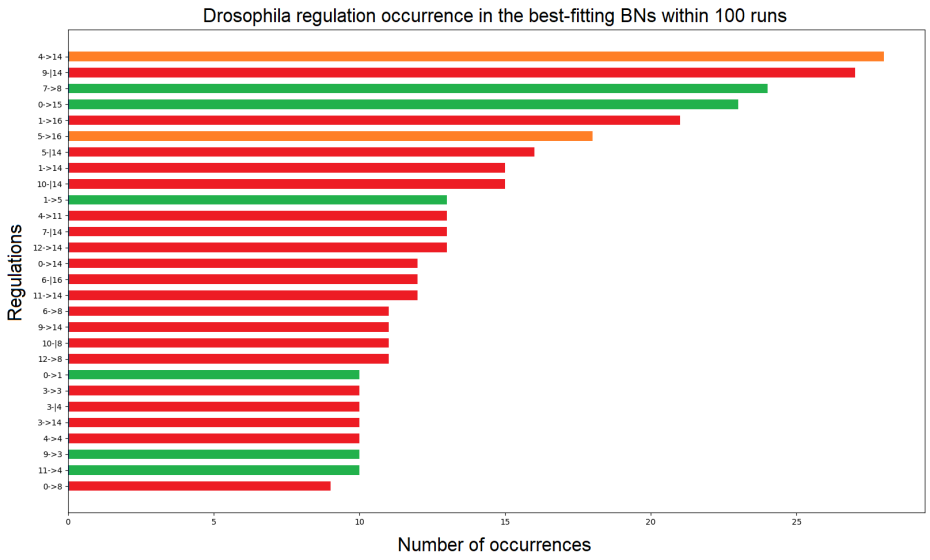

Figure B.10: Structural accuracy of Drosophila model with 50% of sinks specified. The plot shows the number of occurrences of the 28 most frequent regulations in best-fitting BNs from each of 100 runs of the algorithm according to the target model of the body segmentation in Drosophila. We omitted the constraints for the sake of the readability of the plot. Fifty percent of the explicit sinks of the target model of Drosophila were specified on the input. The y-axis shows the most frequent inferred regulations, and the x-axis shows the number of occurrences within all 100 runs of the algorithm. Six direct edges (in green) were inferred frequently, and the other two were passed as input constraints. Additionally, we can see that 2 other indirect regulations (in orange) are also quite frequent. The remaining regulations (in red) are not present in the target model.

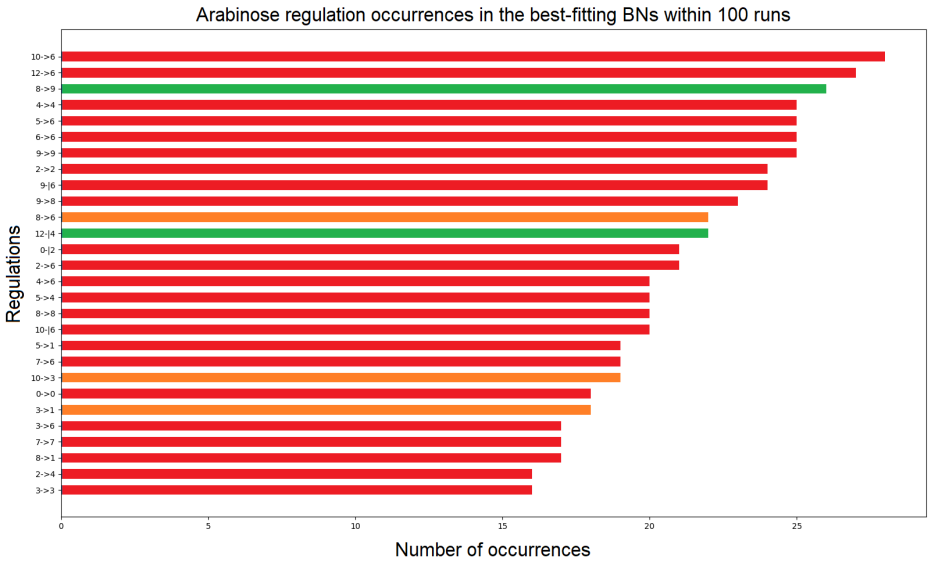

Figure B.11: Structural accuracy of Arabinose model with 50% of sinks specified. The plot shows the number of occurrences of the 28 most frequent regulations in best-fitting BNs from each of 100 runs of the algorithm according to the target model of regulation of L-arabinose operon. We omitted the constraints for the sake of the readability of the plot. Fifty percent of the explicit sinks of the target model of Arabinose were specified on the input. The y-axis shows the most frequent inferred regulations, and the x-axis shows the number of occurrences within all 100 runs of the algorithm. Two direct edges (in green) were inferred frequently, and the other two were passed as the input constraints. Additionally, we can see that 3 other indirect regulations (in orange) are also quite frequent. The remaining regulations (in red) are not present in the target model.

B.3 Increase of constraints for dense models

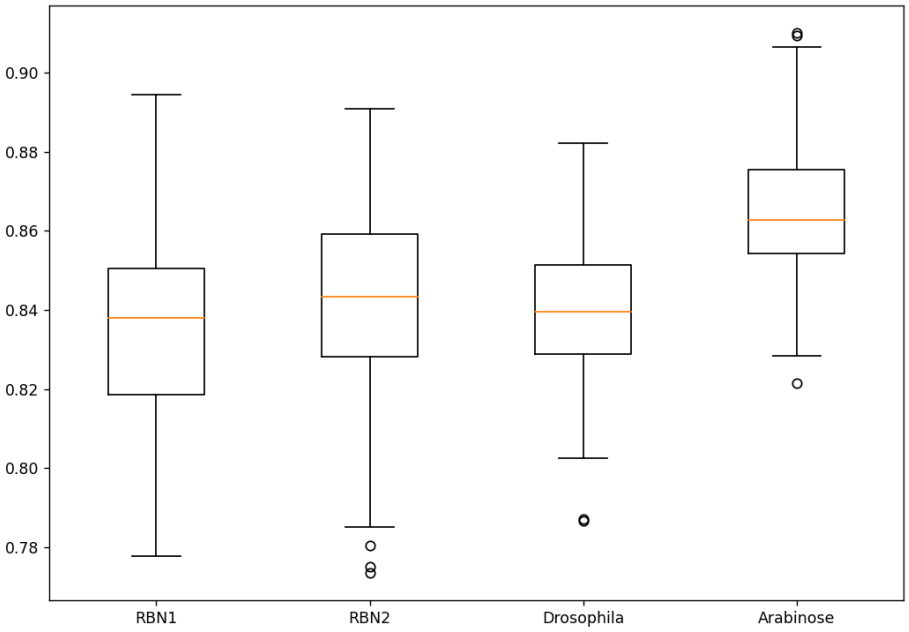

Figure B.12: Dynamic accuracy of the setup with 1/3 of the total number of regulations specified. The plot shows the distribution of the dynamic accuracy, including the best-fitting networks from individual runs for each of 4 dense models. Paradoxically, we can observe a decrease in dynamic accuracy for each of 4 dense models.

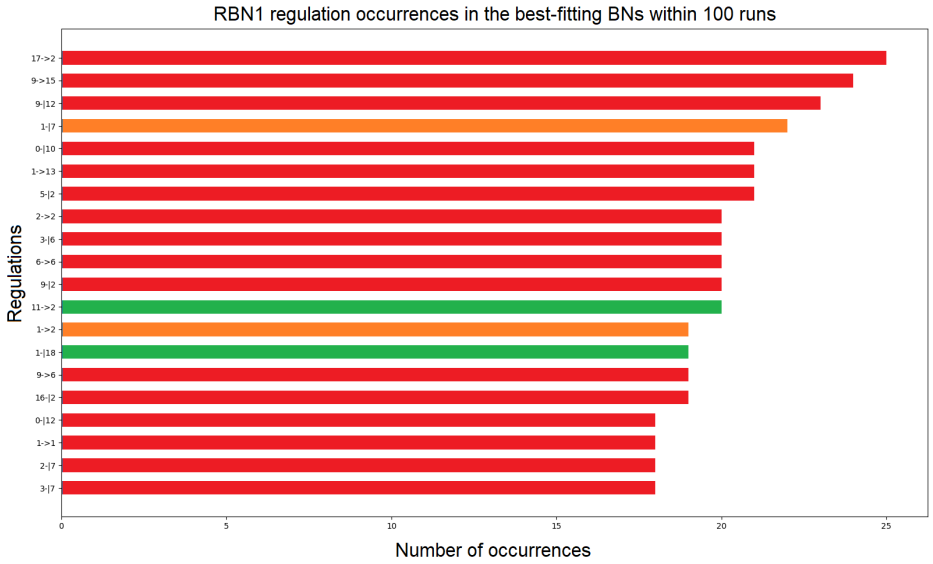

Figure B.13: Structural accuracy of RBN1 model with 1/3 of regulations specified in constraints. The plot shows the number of occurrences of the 20 most frequent regulations in best-fitting BNs from each of 100 runs of the algorithm according to the target RBN1 model. We omitted the constraints for the sake of the readability of the plot. The y-axis shows the most frequent inferred regulations, and the x-axis shows the number of occurrences within all 100 runs of the algorithm. Two direct edges (in green) were inferred frequently, and the other ten were passed as the input constraints. Additionally, we can see that 2 other indirect regulations (in orange) are also quite frequent. The remaining regulations (in red) are not present in the target model.

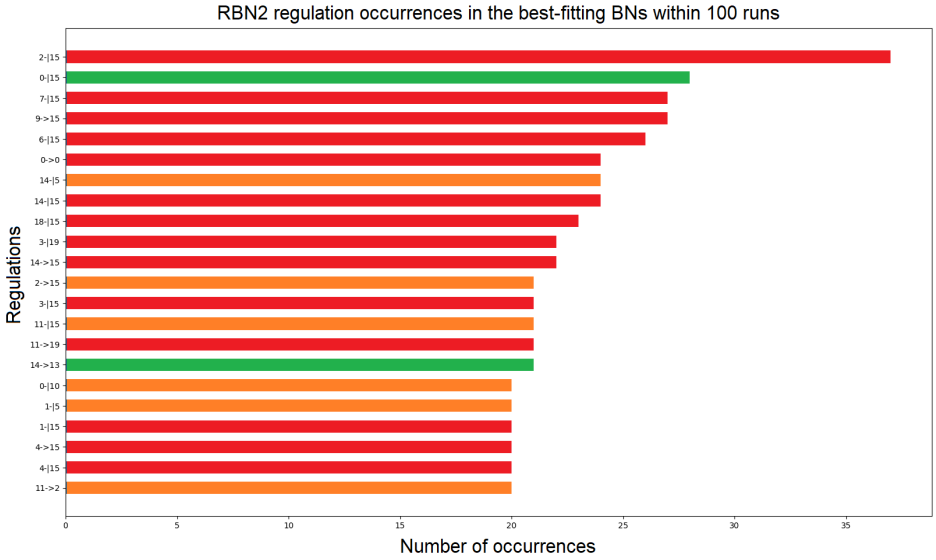

Figure B.14: Structural accuracy of RBN2 model with 1/3 of regulations specified in constraints. The plot shows the number of occurrences of the 22 most frequent regulations in best-fitting BNs from each of 100 runs of the algorithm according to the target RBN2 model. We omitted the constraints for the sake of the readability of the plot. The y-axis shows the most frequent inferred regulations, and the x-axis shows the number of occurrences within all 100 runs of the algorithm. Two direct edges (in green) were inferred frequently, and the other eight were passed as the input constraints. Additionally, we can see that 6 other indirect regulations (in orange) are also quite frequent. The remaining regulations (in red) are not present in the target model.

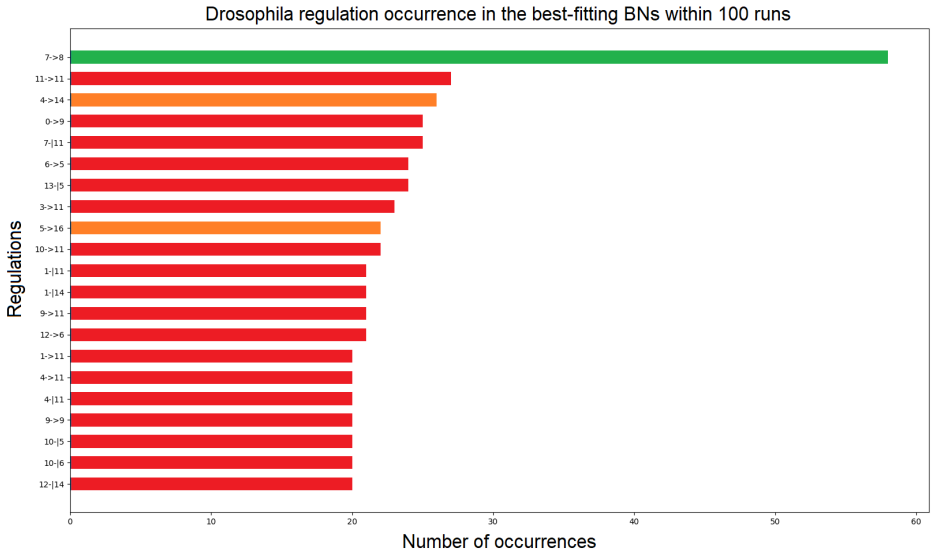

Figure B.15: Structural accuracy of Drosophila model with 1/3 of regulations specified in constraints. The plot shows the number of occurrences of the 21 most frequent regulations in best-fitting BNs from each of 100 runs of the algorithm according to the target model of body segmentation in Drosophila. We omitted the constraints for the sake of the readability of the plot. The y-axis shows the most frequent inferred regulations, and the x-axis shows the number of occurrences within all 100 runs of the algorithm. One direct edge (in green) was inferred frequently, and the other nine were passed as the input constraints. Additionally, we can see that 2 other indirect regulations (in orange) are also quite frequent. The remaining regulations (in red) are not present in the target model.

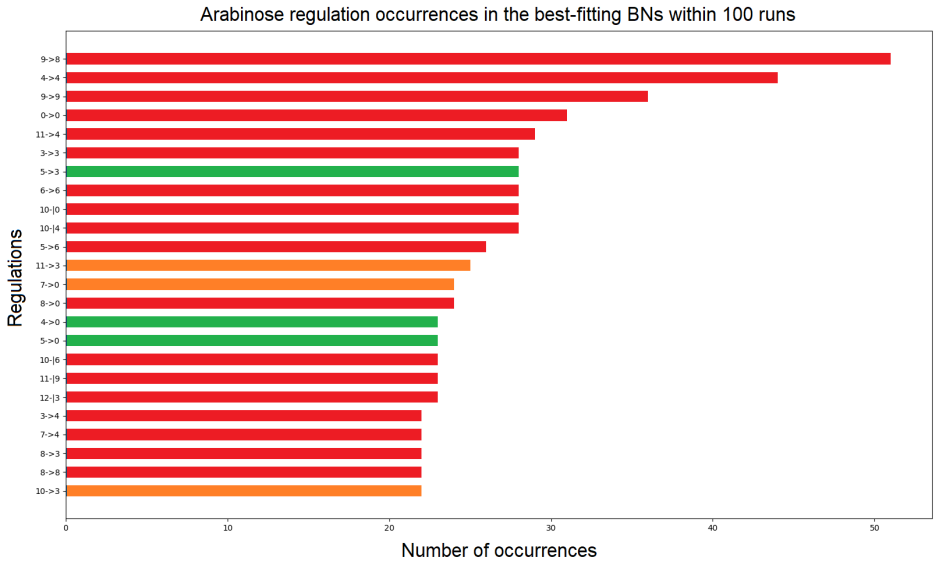

Figure B.16: Structural accuracy of Arabinose model with 1/3 of regulations specified in constraints. The plot shows the number of occurrences of the 24 most frequent regulations in best-fitting BNs from each of 100 runs of the algorithm according to the target model of regulation of L-arabinose operon. We omitted the constraints for the sake of the readability of the plot. The y-axis shows the most frequent inferred regulations, and the x-axis shows the number of occurrences within all 100 runs of the algorithm. Three direct edges (in green) were inferred frequently, and the other six were passed as input constraints. Additionally, we can see that 3 other indirect regulations (in orange) are also quite frequent. The remaining regulations (in red) are not present in the target model.

B.4 Input and output reduction

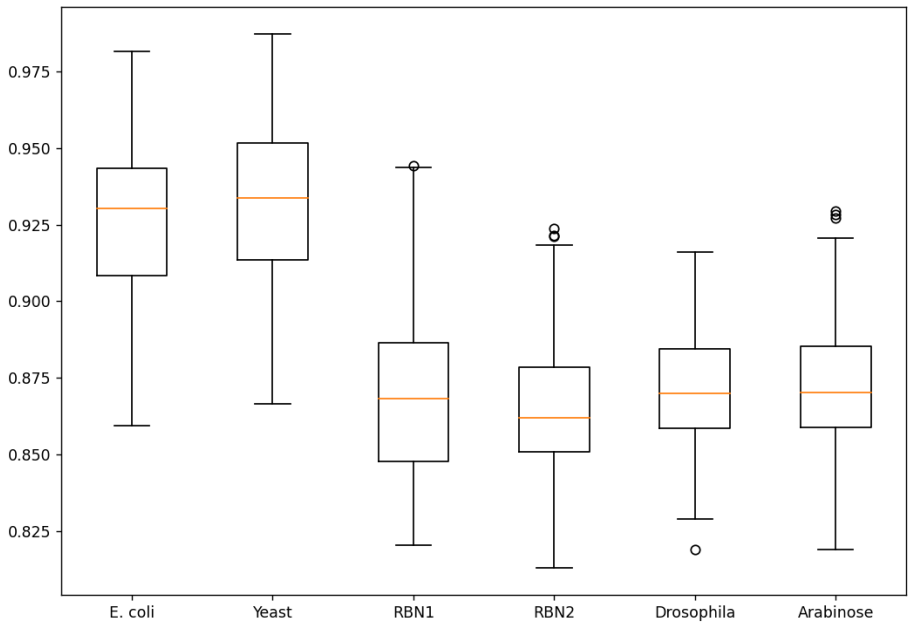

Figure B.17: Dynamic accuracy of the setup with 50% of input and output variables specified. The plot shows the distribution of the dynamic accuracy, including the best-fitting networks from individual runs for each model. We can see no significant difference from the dynamic accuracy of the basic setup.

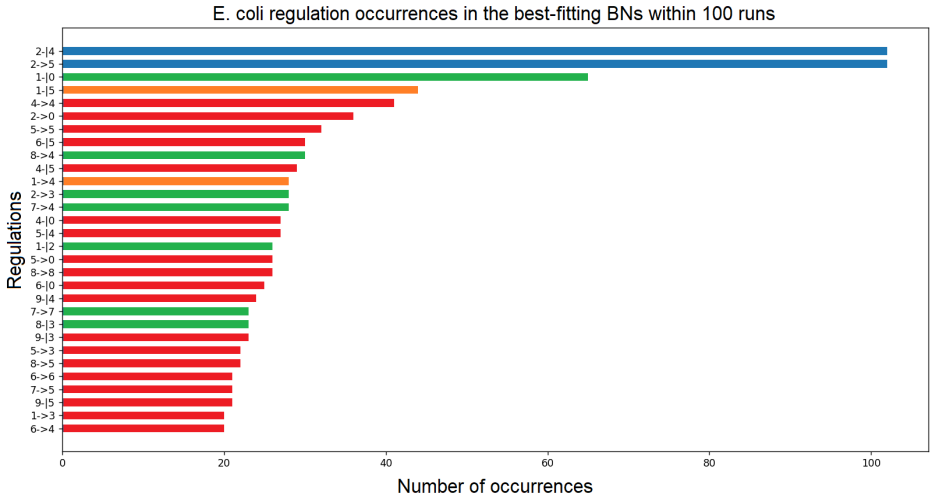

Figure B.18: Structural accuracy of E. coli model with 50% of inputs and outputs were specified. The plot shows the number of occurrences of the 30 most frequent regulations in best-fitting BNs from each of 100 runs of the algorithm according to the target model of E. coli. Fifty percent of the input and output variables of the E. coli model were specified. The y-axis shows the most frequent inferred regulations, and the x-axis shows the number of occurrences within all 100 runs of the algorithm. Seven direct edges (in green) were inferred frequently, and the other two were passed as the input constraints (in blue). Additionally, we can see that 2 other indirect regulations (in orange) are also quite frequent. The remaining regulations (in red) are not present in the target model.

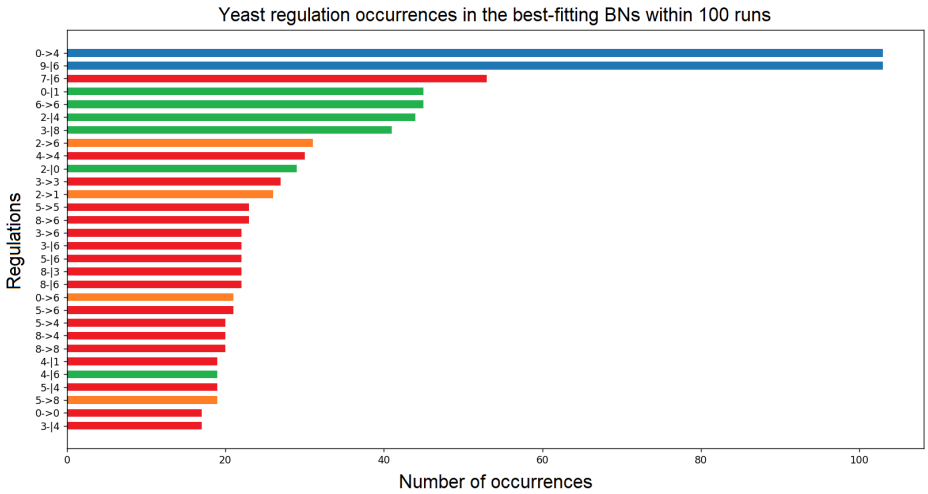

Figure B.19: Structural accuracy of Yeast model with 50% of inputs and outputs were specified. The plot shows the number of occurrences of the 30 most frequent regulations in best-fitting BNs from each of 100 runs of the algorithm according to the target model of Yeast. Fifty percent of the input and output variables of the Yeast model were specified. The y-axis shows the most frequent inferred regulations, and the x-axis shows the number of occurrences within all 100 runs of the algorithm. Six direct edges (in green) were inferred frequently, and the other two were passed as the input constraints (in blue). Additionally, we can see that 4 other indirect regulations (in orange) are also quite frequent. The remaining regulations (in red) are not present in the target model.

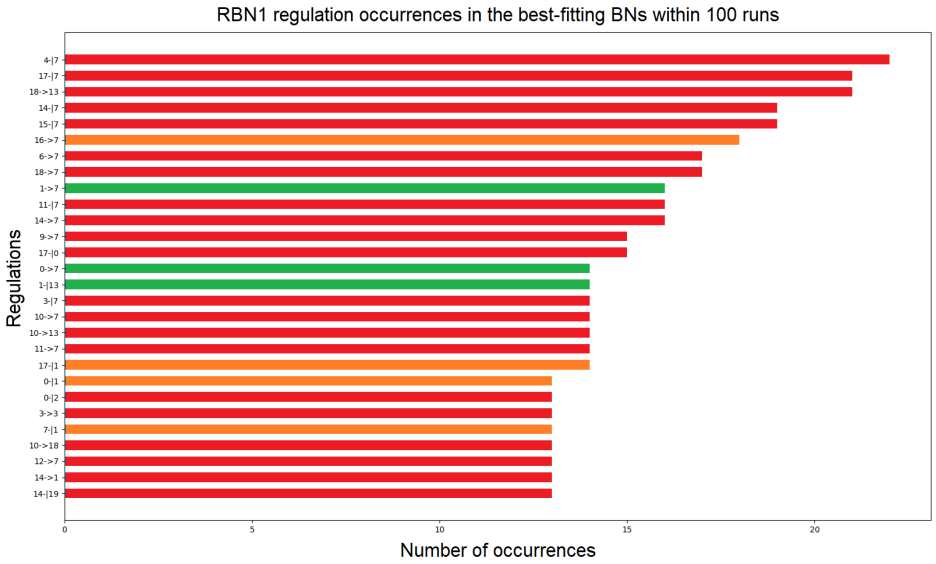

Figure B.20: Structural accuracy of RBN1 model with 50% of inputs and outputs were specified. The plot shows the number of occurrences of the 28 most frequent regulations in best-fitting BNs from each of 100 runs of the algorithm according to the target model of RBN1. We omitted the constraints for the sake of the readability of the plot. Fifty percent of the input and output variables of the RBN1 model were specified. The y-axis shows the most frequent inferred regulations, and the x-axis shows the number of occurrences within all 100 runs of the algorithm. Three direct edges (in green) were inferred frequently, and the other two were passed as input constraints. Additionally, we can see that 4 other indirect regulations (in orange) are also quite frequent. The remaining regulations (in red) are not present in the target model.

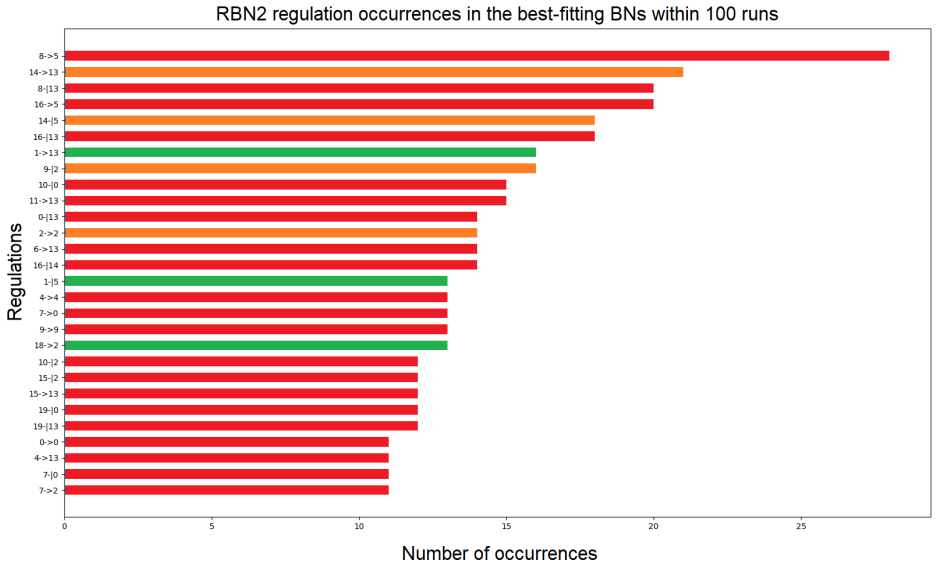

Figure B.21: Structural accuracy of RBN2 model with 50% of inputs and outputs were specified. The plot shows the number of occurrences of the 28 most frequent regulations in best-fitting BNs from each of 100 runs of the algorithm according to the target model of RBN2. We omitted the constraints for the sake of the readability of the plot. Fifty percent of the input and output variables of the RBN2 model were specified. The y-axis shows the most frequent inferred regulations, and the x-axis shows the number of occurrences within all 100 runs of the algorithm. Three direct edges (in green) were inferred frequently, and the other two were passed as input constraints. Additionally, we can see that 4 other indirect regulations (in orange) are also quite frequent. The remaining regulations (in red) are not present in the target model.

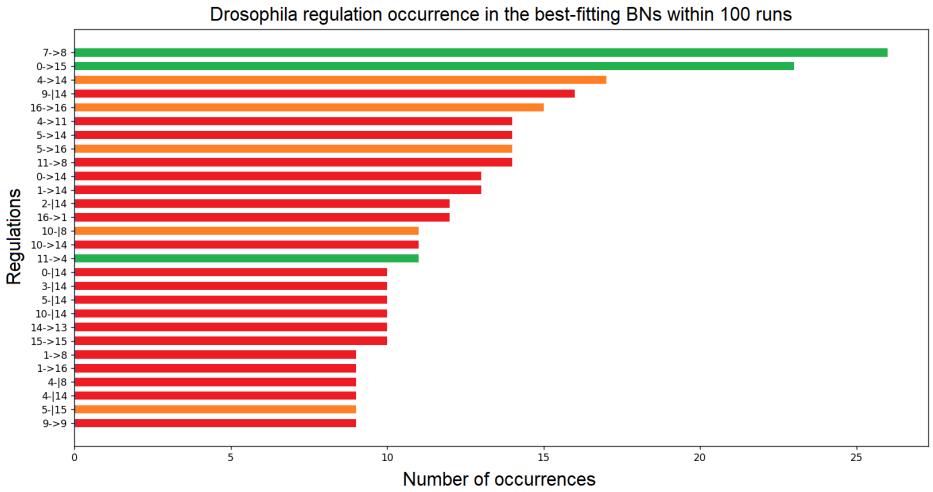

Figure B.22: Structural accuracy of Drosophila model with 50% of inputs and outputs were specified. The plot shows the number of occurrences of the 28 most frequent regulations in best-fitting BNs from each of 100 runs of the algorithm according to the target model of body segmentation in Drosophila. We omitted the constraints for the sake of the readability of the plot. Fifty percent of the input and output variables of the Drosophila model were specified. The y-axis shows the most frequent inferred regulations, and the x-axis shows the number of occurrences within all 100 runs of the algorithm. Three direct edges (in green) were inferred frequently, and the other two were passed as input constraints. Additionally, we can see that 5 other indirect regulations (in orange) are also quite frequent. The remaining regulations (in red) are not present in the target model.

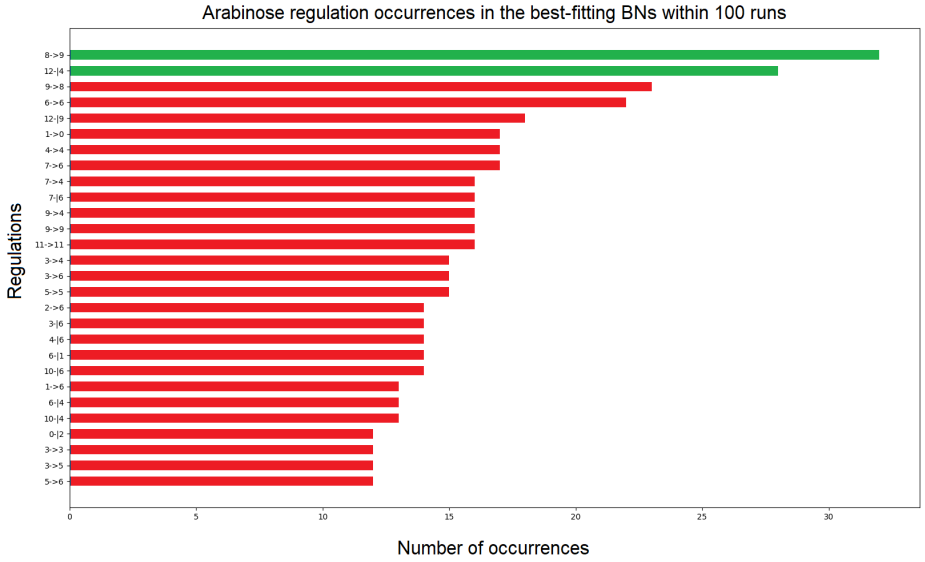

Figure B.23: Structural accuracy of Arabinose model with 50% of inputs and outputs were specified. The plot shows the number of occurrences of the 28 most frequent regulations in best-fitting BNs from each of 100 runs of the algorithm according to the target model of regulation of L-arabinose operon. We omitted the constraints for the sake of the readability of the plot. Fifty percent of the input and output variables of the Arabinose model were specified. The y-axis shows the most frequent inferred regulations, and the x-axis shows the number of occurrences within all 100 runs of the algorithm. Two direct edges (in green) were inferred frequently, and the other two were passed as the input constraints. We can see no other indirect regulations. The remaining regulations (in red) are not present in the target model.