Statistical Data Analysis 2 - Final Project - Report

Czapiewska Magdalena, Khodzina Anastasiya, Nechaieva Veronika, Znamierowski Mikołaj

January 22, 2026

Contents

Code is available on GitHub: https://github.com/quantumFeline/stats-group-proejct-1.git

1 Boolean Network representation and construction

In this section we describe the implementation of the code handling Boolean networks contained in `BooleanNetwork.py`. We divide it into four subsections for better readability.

1.1 Creation of the network

Each Boolean network is initiated with specified number of nodes, as, according to the project's description, everything else should be randomized. However, since part II requires the usage of a specific model, we also provide an additional option to specify the transitions (this also gives us a possibility to run more tests on previously created networks, if needed). The main usage of this feature is present in loading the networks from files, which will be described later. If no transitions are specified, the model creates them randomly, with at most 3 parent nodes for each node. The concrete representation of transitions (along with some theoretical issues concerning it) and the method of their creation can be found in the following subsection.

1.2 Transitions

Theoretically, each transition has a form of a logical formula (with negations, conjunctions and disjunctions) with nodes as boolean variables. However, each such formula can be rewritten in multiple ways, without changing the output. For example, the famous De Morgan's law states that $\neg(x_1 \wedge x_2)$ is equivalent to $\neg x_1 \vee \neg x_2$. To standardize this, the Conjunctive Normal Form (CNF) or the Disjunctive Normal Form (DNF) can be used. This however proves quite problematic to implement when creating a random **unbiased** logical formula (i.e. each transition can be achieved with equal probability). While implementing it, one can also easily run into problems similar to SAT, which are NP-hard.

To address the above issues, the code uses a binary representation of the transitions (in particular, it avoids all logical formulas). Namely: each transition is a tuple of two lists. The first is a list of (non-repeating) parents of length n (where $0 \leq n \leq 3$ is chosen randomly). The second one is a random binary sequence of length 2^n. It represents the transition in the following format: if the last parent is zero, we look at the first half of the sequence, otherwise, on the second half. We repeat it for all parents (from last to first), reducing our sequence to one element, which is the outcome of the transition. This way, all transitions are reachable, and **equally probable** (for given n), thus giving the most general networks.

To illustrate it with an example: suppose that the node x_2 has parents x_3, x_4, x_0 and the transition is $[0, 1, 0, 1, 1, 0, 1, 1]$. This means that the transition is described by the following table:

x_3	x_4	x_0	x_2
0	0	0	0
1	0	0	1
0	1	0	0
1	1	0	1
0	0	1	1
1	0	1	0
0	1	1	1
1	1	1	1

It's worth noting here, that if the node has no parents, by the above implementation, its transition vector has $2^0 = 1$ element. However, we don't want to overwrite it with that value, so, when updating the state, its transitions are always skipped (the actual transition applied to it is the identity).

1.3 Saving and loading the network

After creating the network, its original structure (i.e. the number of nodes and the transitions) is saved to a file for later reference (which is crucial for the following parts where we compare the ground truth network structure to the BNFinder results). The format used is the following:

- first line of the file is a single integer defining the number of nodes;
- each of the next lines defines a transition in the format `[list of node parents]; [transition]` (separated with commas) for the corresponding node.

A file with such format can be then loaded into the program, which allows for using custom Boolean networks, not just the random ones.

1.4 Creating and saving the datasets

After the creation of our model, we can create datasets by running a single command. We can specify the following arguments:

- **-d** the number of datapoints (i.e. transitions) in the dataset;
- **-l** the length of all trajectories in the dataset;
- **-s** whether the transitions should be applied synchronously or not;
- **-f** the sampling frequency;

We randomize the starting states for the trajectories.
Example:

```
CreateDatasets.py -o output.txt -g network.txt -n 10 -d 20 -l 15 -s
```

The datasets created in this way are saved to a file of format compatible with BNFinder2 for later inference.

2 Attractor detection implementation

One of the **key characteristics of a dataset** generated from a Boolean network is the **proportion of transient and attractor states** present in the observed trajectories. Since this proportion is explicitly investigated in the project, it is necessary to **identify attractors in the state space** of each generated network.

To enable control over this characteristic, dedicated classes were implemented that, for a given Boolean network, classify each possible network state either as an attractor state or as a transient state, based on the structure of the state transition graph.

The attractor detection strategy depends on the network update mode, as the structure of the state transition graph differs between synchronous and asynchronous update schemes. The following subsections describe the approaches used for attractor identification under synchronous and asynchronous dynamics.

2.1 Attractors in synchronous mode

The implementation of the `StateSpaceAnalyzer` class can be found in the `StateSpaceAnalysisSynchronous.py` file.

For each Boolean network, a **dictionary** is created that stores **information for every possible network state**. Specifically, for each state, it is recorded whether the state **belongs to an attractor**, the **identifier of the corresponding attractor**, and the **distance of the state to the attractor** (with distance 0 for attractor states).

The algorithm uses the fact that **trajectories in synchronous Boolean networks are deterministic**: the next state of the network is uniquely determined by its current state. Starting from each state that has not yet been analyzed, the network is iteratively updated following its synchronous dynamics until one of two situations occurs:

1. The trajectory reaches a state that **has already been visited within the current path**. In this case, a **cycle** (possibly consisting of a single state) has been detected, corresponding to a **new attractor**. All states in the cycle are marked as attractor states, and transient states leading to the cycle are assigned the corresponding attractor ID along with their distance to the attractor.

2. The trajectory reaches a state that **has already been analyzed in a previous trajectory**. In this case, the previously computed attractor information is propagated backward along the current path, assigning attractor IDs and distances to all transient states in the path.

This approach ensures that **each state in the network is analyzed exactly once**, and that all attractors and their basins of attraction are correctly

identified.

The distance from a transient state to its attractor is computed during the analysis. This information was originally intended to support a specific dataset generation strategy, but the approach to dataset generation has since changed. Nevertheless, the class retains the distance computation functionality, as it does not increase the computational cost and could potentially be used in alternative dataset generation strategies.

2.2 Attractors in asynchronous mode

Two versions of attractor detection in asynchronous mode were implemented. The helper class `AsyncAnalyzer` is provided in the `AttractorAsynchronous.py` file.
To detect attractors in the state transition graph under asynchronous dynamics, the `analyze` method can be used either from the `AsyncAnalyzerNX` class, located in `AttractorsAsynchronousNetworkx.py`, or from the `AsyncAnalyzerTarjan` class, located in `AttractorsAsynchronousTarjan.py`.

In asynchronous state transition system, the next state of the network is **not uniquely determined** by the current state, because only a single node is updated at a time and the choice of node is random. Consequently, the state transition graph can have multiple outgoing edges from a single state. In this situation, attractors are defined as **strongly connected components (SCCs) without outgoing edges**, meaning that each state in the component can reach every other state in the component, and no transitions leave the component. Identifying SCCs in the graph allows us to detect all asynchronous attractors and their basins of attraction.

The goal of the analysis is to assign **to each state of the network** whether it is an attractor or a transient state, and if it is an attractor, to assign the **identifier of the corresponding attractor**.

The two implemented approaches differ in the method used to find these SCCs:

1. **NetworkX-based approach (`AsyncAnalyzerNX`):** This approach constructs the full asynchronous state transition graph using the NetworkX library. Strongly connected components are computed with NetworkX's built-in algorithms. SCCs that have no outgoing edges are identified as attractors, and all other states are marked as transient. Each attractor state is assigned an attractor ID.

2. **Tarjan-based approach (`AsyncAnalyzerTarjan`):** This approach implements **Tarjan's algorithm** for finding strongly connected components directly, without relying on external libraries. Each SCC is checked for

outgoing edges, and those without outgoing edges are classified as attractors. Transient states are marked accordingly. Each attractor state is assigned an attractor ID.

In this project, we use the `AsyncAnalyzerNX` class. Since the NetworkX library was introduced during the course, we consider its use appropriate for this analysis.

3 Datasets generation

3.1 The way of generating a dataset with given characteristics

The dataset generation process is implemented in `GenerateTrajectories.py`. For each network size n in the range from 5 to 16, a Boolean network is created. For each network and for each update mode (synchronous or asynchronous), the following tasks are performed:

1. **State space analysis:** The full state space of the Boolean network is analyzed to identify attractor and transient states. In synchronous mode, the `StateSpaceAnalyzer` class is used, while in asynchronous mode, the `AsyncAnalyzerNX` class is employed.

2. **Long trajectories generation:** From every possible initial state of the network, a long trajectory of states is generated by iteratively applying the network update rules. In total, 2^n trajectories of length 300 are generated, providing a base from which shorter dataset fragments can later be sampled.

3. **Loop:** For each combination of dataset parameters (number of trajectories, trajectory length, sampling frequency and transient fraction) the sampling process is applied to create dataset fragments with the requested characteristics.

    - **Fragment candidate identification:** For each long trajectory, multiple candidate fragments are considered by varying the starting offset (from 0 to `sampling_frequency`-1) and applying the user-defined sampling frequency. Each candidate fragment is checked to ensure it contains at least the required number of transient states and attractor states according to the desired fraction. Only fragments meeting these criteria are retained as candidates - each candidate is represented as a pair consisting of the long trajectory before applying the sampling frequency and the offset.
    - **Fragment shuffling, trimming, and uniqueness check:** Candidate fragments are first randomly shuffled. Then, for each candidate, if the dataset has not yet reached the desired number of trajectories, a fragment is selected by taking every `sampling_frequency`-th

state and deterministically picking the last required transient states and the first required attractor states. This ensures that each fragment has the desired length and maintains the intended proportion of transient to attractor states. After trimming, each fragment is represented as a tuple and checked against already selected fragments. If the fragment is already included, it is skipped. This process guarantees that all trajectories in the final dataset are distinct and that the dataset contains the requested number of trajectories.

- **Insufficient candidate handling:** If at any stage there are not enough fragments to satisfy the requested number of trajectories - either because the number of candidate fragments before trimming is too small, or because there are not enough unique fragments after trimming - the sampling function returns `None`. This guarantees that no dataset is created with fewer trajectories than requested.

3.2 Description of the datasets generated

The ground truth network representations are located in the `ground_truth_networks` directory. Example datasets (input files for BNFinder2) are available in the `datasets` directory. Only datasets for the 10-node network are included in the repository for demonstration purposes; the remaining datasets are stored locally.

Characteristics of the datasets:

- **Network size (n):** The number of nodes in the Boolean network, ranging from 5 to 16. For each n, a separate Boolean network is created.
- **Number of data points:** 13 values ranging from 3 to 39 (step size of 3).
- **Trajectory length:** 13 values ranging from 3 to 39 (step size of 3).
- **Update mode:** Synchronous and Asynchronous.
- **Sampling frequency:** 1, 2, and 3.
- **Fraction of transient states:** 0.2, 0.4, 0.6, 0.8.

In some cases, it was not possible to generate a dataset with the requested characteristics. This typically occurred for small networks (small n) and synchronous update mode, where the combination of a long required trajectory length and a high transient fraction made it impossible to find enough trajectory fragments that satisfied the criteria. In such cases, the dataset generation function returned `None` and no dataset was created for that parameter combination.

4 BNFinder2 usage

BNFinder2 was used to reconstruct networks from generated datasets (trajectories). Since the tool relies on the deprecated Python 2.7 and specific libraries (Scipy, FPConst), an isolated Conda environment was established to resolve dependency conflicts.

The execution of BNFinder2 on datasets derived from synthetic networks is implemented in the `RunBnfExperiments.py` script.

Example BNFinder2 results are available in the `results` directory. Only results for the 10-node network are included in the repository for demonstration purposes; the remaining result files are stored locally.

Due to a long computation time, we needed to reduce the number of parameters in a grid for bigger number of nodes.

For $n = 5..13$ we run bnf for the parameters:

- **Number of data points:** 13 values ranging from 3 to 39 (step size of 3).
- **Trajectory length:** 13 values ranging from 3 to 39 (step size of 3).
- **Update mode:** Synchronous and Asynchronous.
- **Sampling frequency:** 1, 2, and 3.
- **Fraction of transient states:** 0.2, 0.4, 0.6, 0.8.

For $n = 14..16$ we run bnf for the parameters:

- **Number of data points:** 3, 15, 27, 39.
- **Trajectory length:** 3, 15, 27, 39.
- **Update mode:** Synchronous and Asynchronous.
- **Sampling frequency:** 1, 2, and 3.
- **Fraction of transient states:** 0.2, 0.6.

In the repository we leave the version of a script for reconstructing networks from all datasets (the whole grid of parameters for all networks).

5 Evaluation of the accuracy of the reconstructed network

5.1 Evaluation metrics chosen

Boolean networks are directed graphs. To compute evaluation metrics, we represent the networks using adjacency matrices, where $A[i, j] = 1$ if and only if

there is a directed edge from x_i to x_j, and $A[i,j] = 0$ otherwise.

The accuracy of the reconstructed networks is assessed using five measures: recall, precision, F1-score, normalized Hamming distance and normalized Structural Hamming distance. Ideally, recall, precision, and F1-score should be high, whereas the normalized Hamming and Structural Hamming distances should be low.

All functions for computing metrics from adjacency matrices, as well as for generating adjacency matrices from ground truth files or SIF files (outputs of BNFinder), are implemented in the file `EvaluationMetrics.py`.
The evaluation of BNFinder results on datasets generated from synthetic networks is implemented in `EvaluateAllBnfResults.py`.

Description of metrics and justification of choice

We define the basic quantities as follows:

- TP (true positives) – the number of edges present in both the ground truth network and the reconstructed network
- FP (false positives) – the number of edges present in the reconstructed network but absent in the ground truth network
- TN (true negatives) – the number of ordered pairs of nodes without an edge in both the ground truth and reconstructed networks
- FN (false negatives) – the number of edges present in the ground truth network but missing in the reconstructed network

Recall

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}}$$

Recall allows measuring the fraction of dependencies (edges) present in the ground truth network that are also identified in the reconstructed network.

Precision

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}$$

Precision allows quantifying the proportion of edges inferred by the algorithm that are indeed present in the ground truth network.

F1-score

$$\text{F1} = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}$$

The F1-score is the harmonic mean of precision and recall. It reaches high values only if both precision and recall are high, making it a good indicator of the quality of the reconstruction, as it measures both whether the algorithm correctly reconstructed the edges that exist and whether it avoided reconstructing edges that should not be present.

Hamming distance

The Hamming distance counts the total number of differing entries between the adjacency matrices of the ground truth and reconstructed networks:

$$\mathrm{HD} = \sum_{i,j} \mathbf{1}[A[i,j] \neq B[i,j]],$$

where A and B are the adjacency matrices.

To compare the reconstruction accuracy across networks of different sizes, we define the normalized Hamming distance by dividing the Hamming distance by the total number of ordered pairs of vertices:

$$\mathrm{HD}_{\mathrm{norm}} = \frac{\mathrm{HD}}{n^2},$$

where n is the number of nodes in the network.

The normalized Hamming distance allows measuring the fraction of ordered pairs of vertices (i, j) for which either a directed edge from i to j exists in the ground truth but is missing in the reconstruction, or the edge is inferred by the reconstruction but does not exist in the ground truth.

Structural Hamming distance

The Structural Hamming distance (SHD) measures the number of unordered pairs of vertices for which the interactions in the reconstructed network need to be modified to match the ground truth network. Each inconsistency between a pair of vertices - whether it requires an edge addition, removal, or reversal - counts as one operation. In other words, we count whether a pair is incorrect, without distinguishing the type of correction needed (and without counting the number of corrections - addition of 2 edges counts as 1).

Consider a pair of vertices (i, j):

- For self-loops ($i = j$), a mismatch between the ground truth and reconstructed network counts as one operation.
- Now consider $i \neq j$. Let (p, q) denote the edge configuration in the network, where $p = 1$ if there is an edge $i \to j$ (otherwise $p = 0$) and $q = 1$ if there is an edge $j \to i$ (otherwise $q = 0$). The following table summarizes

the SHD contribution for each possible combination of ground truth and reconstructed configurations:

Ground truth (p, q)	Reconstructed (p, q)			
	(0,0)	(1,0)	(0,1)	(1,1)
(0,0)	0	1 (remove $i \to j$)	1 (remove $j \to i$)	1 (remove both edges)
(1,0)	1 (add $i \to j$)	0	1 (reverse $j \to i$)	1 (remove $j \to i$)
(0,1)	1 (add $j \to i$)	1 (reverse $i \to j$)	0	1 (remove $i \to j$)
(1,1)	1 (remove both edges)	1 (add $j \to i$)	1 (add $i \to j$)	0

To compare reconstruction accuracy across networks of different sizes, we define the normalized Structural Hamming distance by dividing the Structural Hamming distance by the total number of unordered pairs of vertices, including self-loops:

$$\mathrm{SHD}_{\mathrm{norm}} = \frac{\mathrm{SHD}}{\frac{n(n-1)}{2} + n},$$

where n is the number of vertices in the network.

The normalized Structural Hamming distance measures the fraction of unordered pairs of vertices (i, j) for which the interaction in the reconstructed network differs from the interaction in the ground truth network.

5.2 Results

We used a custom Python script (`RunBnfExperiments.py`) to fully automate the reconstruction process and evaluation. The script iterates over all combinations of parameters considered (see the list in BNFinder2 usage) and saves results. Then a script `EvaluateAllBnfResults.py` is used to collect the metrics measuring the Bayesian network restoration for each combination. The results are stored in a .tsv table (`tables/evaluation_asynchronous.tsv`, `tables/evaluation_synchronous.tsv`).

After the .tsv results are created, a separate script - `NetworkAccuracyEvaluation.py` goes over the results and displays the aggregated, averaged-out metrics depending on the given parameters. The following graph types were used:

- Linear graph (metric against a parameter)
- Linear comparison graph (MDL vs BDE evaluation)
- Heatmap (metric against a pair of parameters)
- Bar plot (a collection different metrics against a parameter)

For most graphs presented we use aggregated metric, i.e. we consider all possible values of the parameters not analyzed in the given graph; we also created some graphs where the other parameters are fixed. In theory, this should be able to pick up on stronger correlation, as in situations where other

parameters are suboptimal (e.g. good trajectory length but too few data points or vice versa) there is more noise, which may make the positive trends less apparent. In practice, this results in having too little overall data to average out and, consequentially, too much noise. However, the plots from synchronous and asynchronous data were nevertheless separated for this section.

We begin the analysis by presenting a complete overview of all evaluation metrics for both update modes. For each mode (synchronous and asynchronous), we show grouped bar plots comparing all computed accuracy metrics (Hamming distance normalized, Structural Hamming distance normalized, precision, recall, and F1 score) as a function of key experimental parameters.

Synchronous update mode

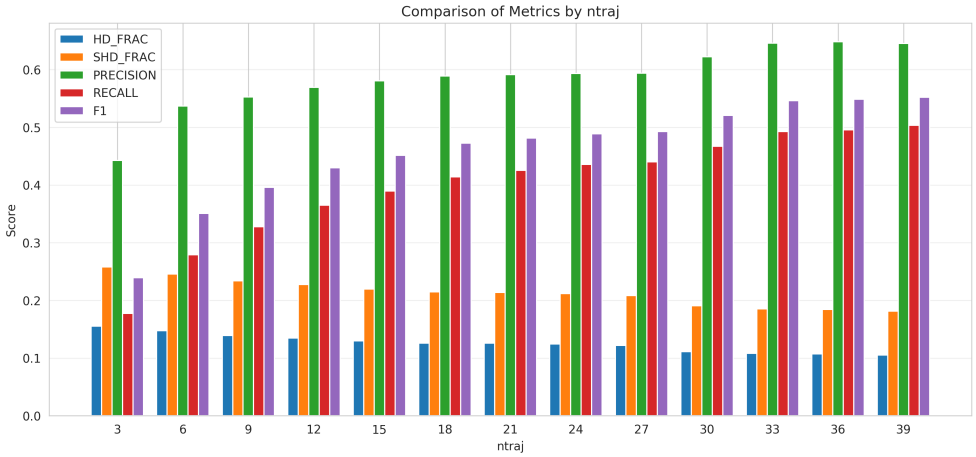


Figure 1: Comparison of all accuracy metrics as a function of the number of trajectories (synchronous mode).

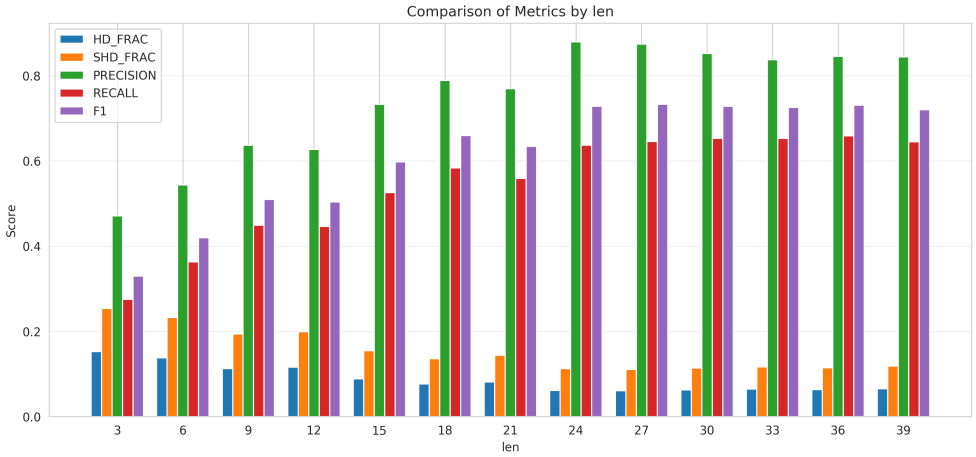


Figure 2: Comparison of all accuracy metrics as a function of trajectory length (synchronous mode).

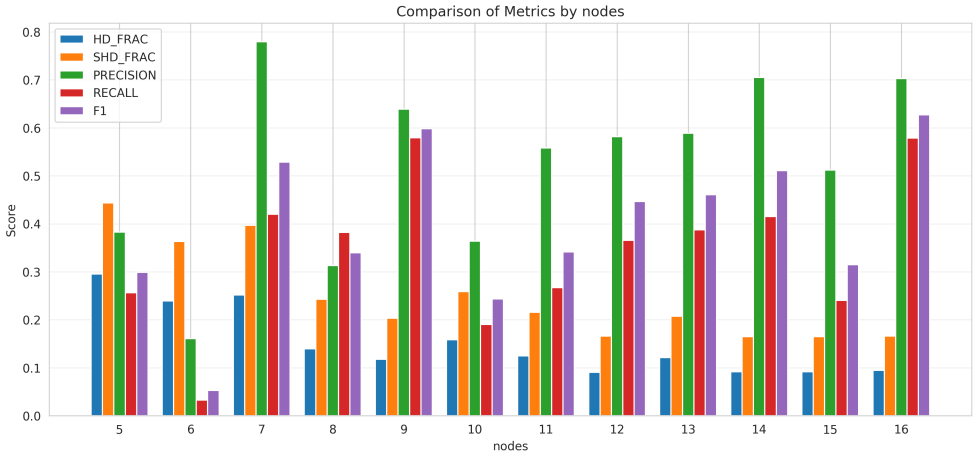


Figure 3: Comparison of all accuracy metrics as a function of network size (synchronous mode).

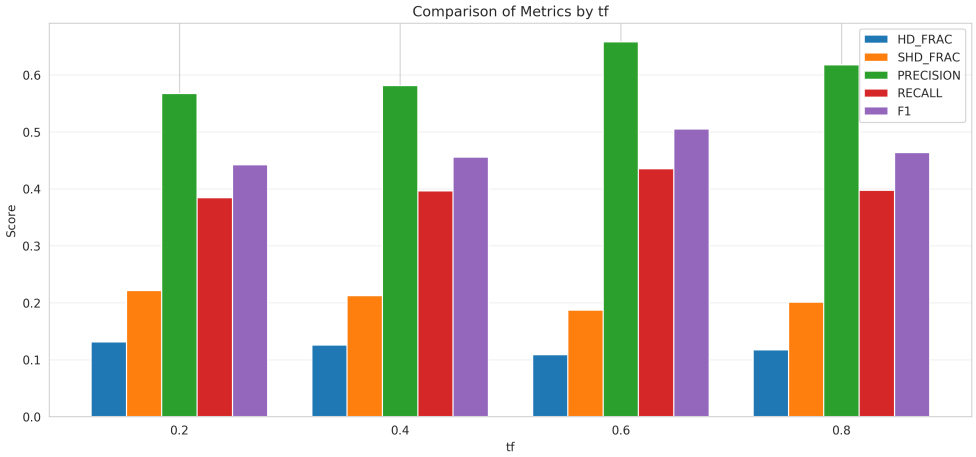


Figure 4: Comparison of all accuracy metrics as a function of transient fraction (synchronous mode).

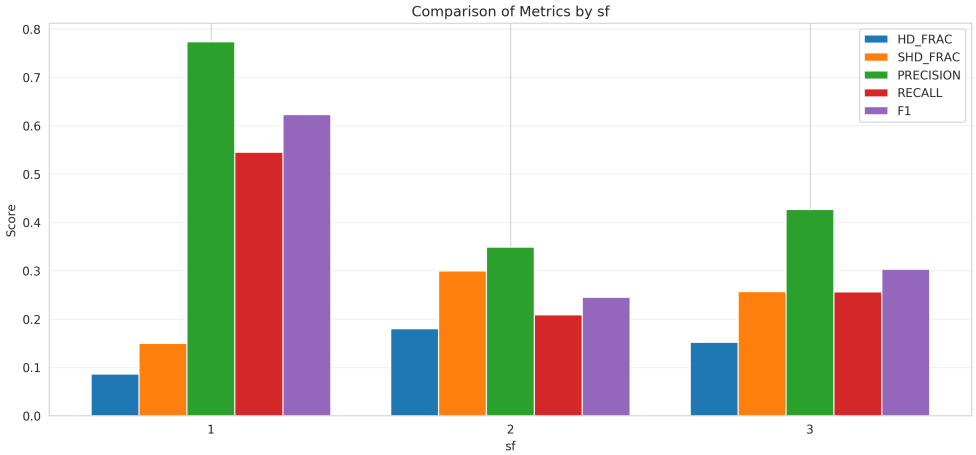


Figure 5: Comparison of all accuracy metrics as a function of sampling frequency (synchronous mode).

Asynchronous update mode

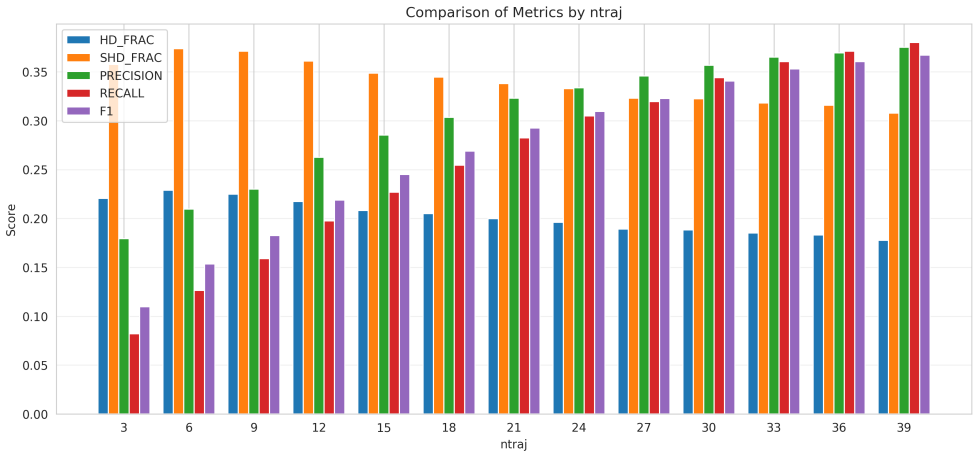


Figure 6: Comparison of all accuracy metrics as a function of the number of trajectories (asynchronous mode).

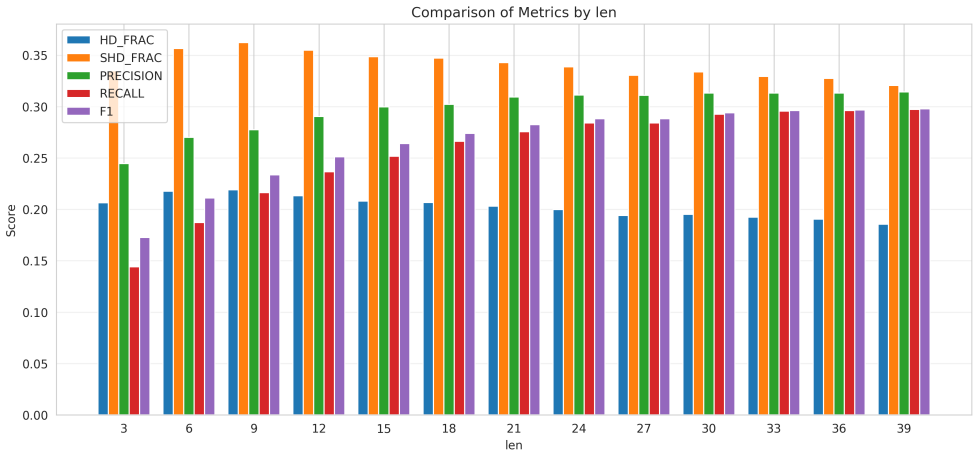


Figure 7: Comparison of all accuracy metrics as a function of trajectory length (asynchronous mode).

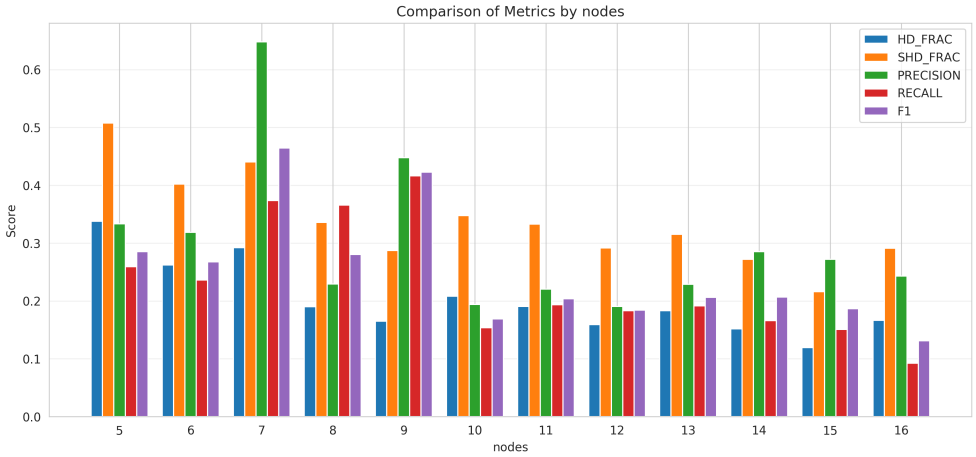


Figure 8: Comparison of all accuracy metrics as a function of network size (asynchronous mode).

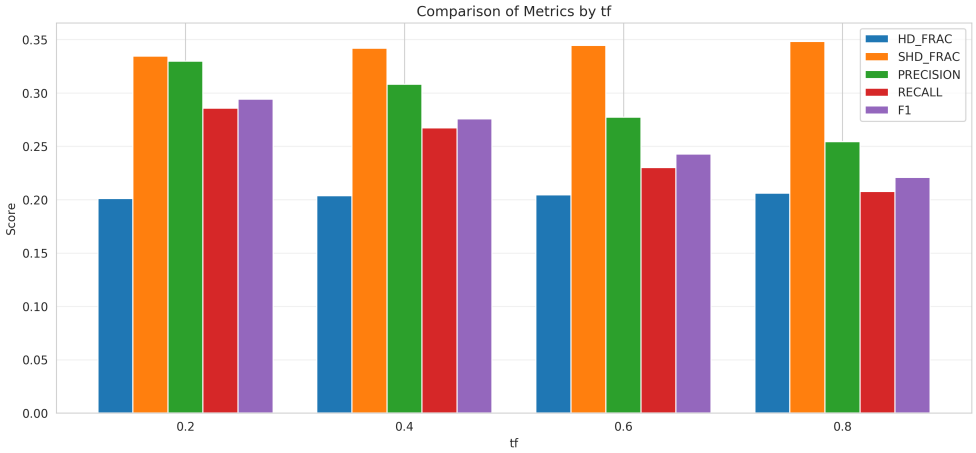


Figure 9: Comparison of all accuracy metrics as a function of transient fraction (asynchronous mode).

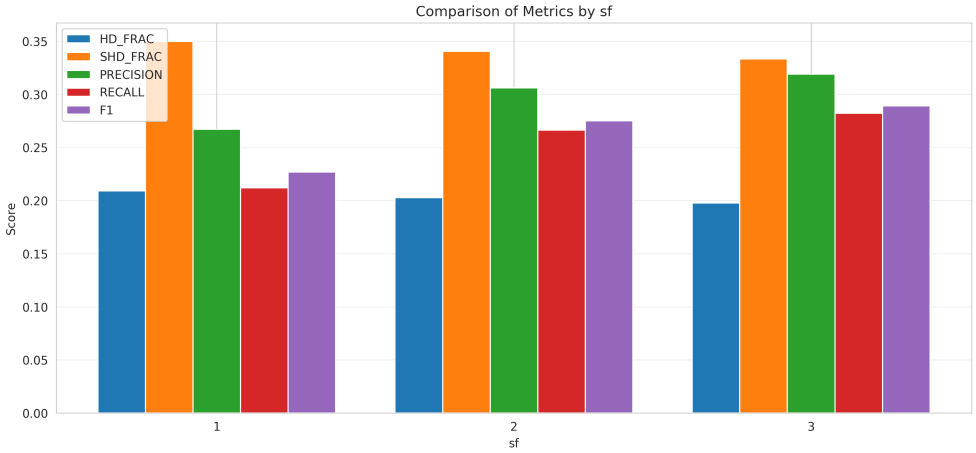
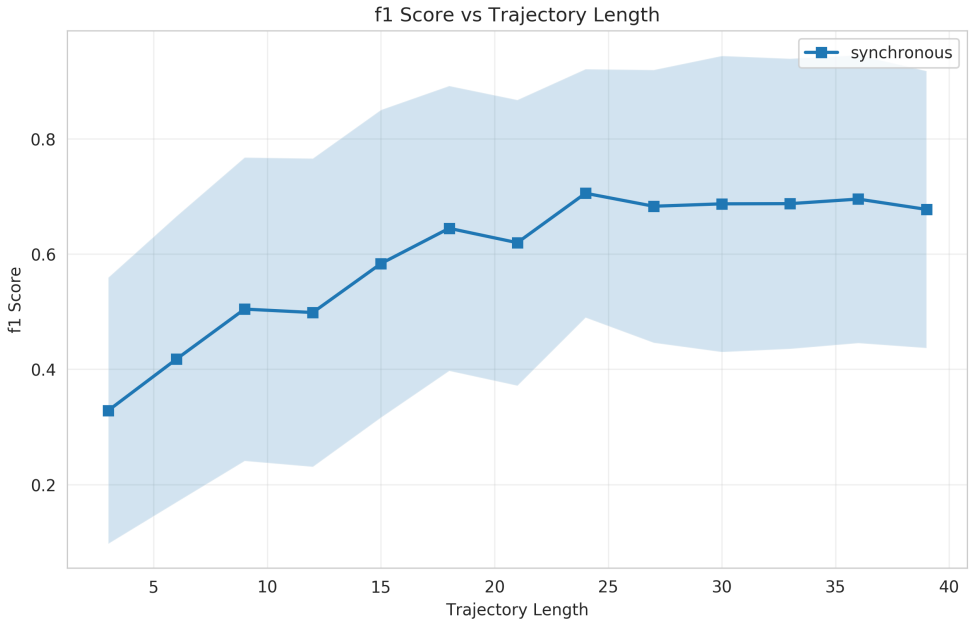
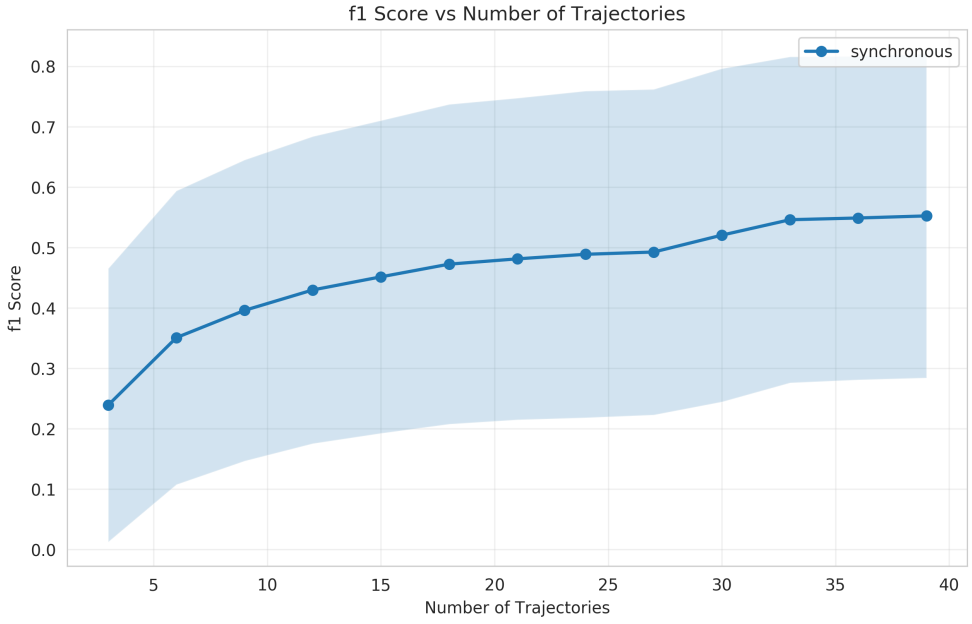


Figure 10: Comparison of all accuracy metrics as a function of sampling frequency (asynchronous mode).

From these graphs we can see that the F1 is the most responsive metric. Specifically, the change in F1 tracks the performance change most clearly in the sanity-check cases, such as increased data volume. Therefore, for the next graphs we will focus on it as our primary metric.

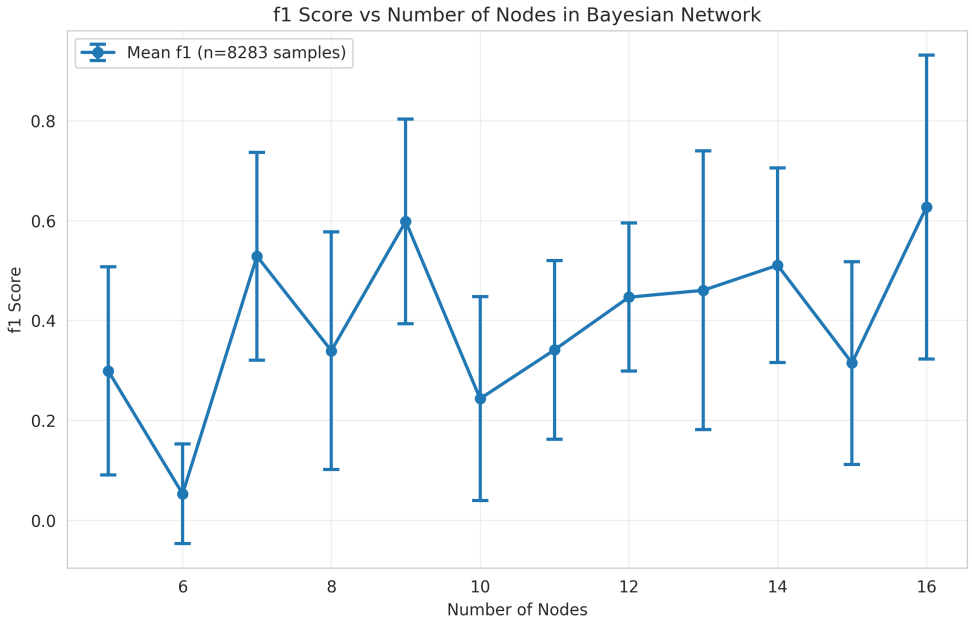
In the follow graphs we can observe that both increasing trajectory length and the number of trajectories improves the algorithm performance; however, increasing trajectory length has diminishing returns, capping at around 20. This is most likely because by this point the trajectory reached the attractor state, and cycling through it further does not provide any additional information.



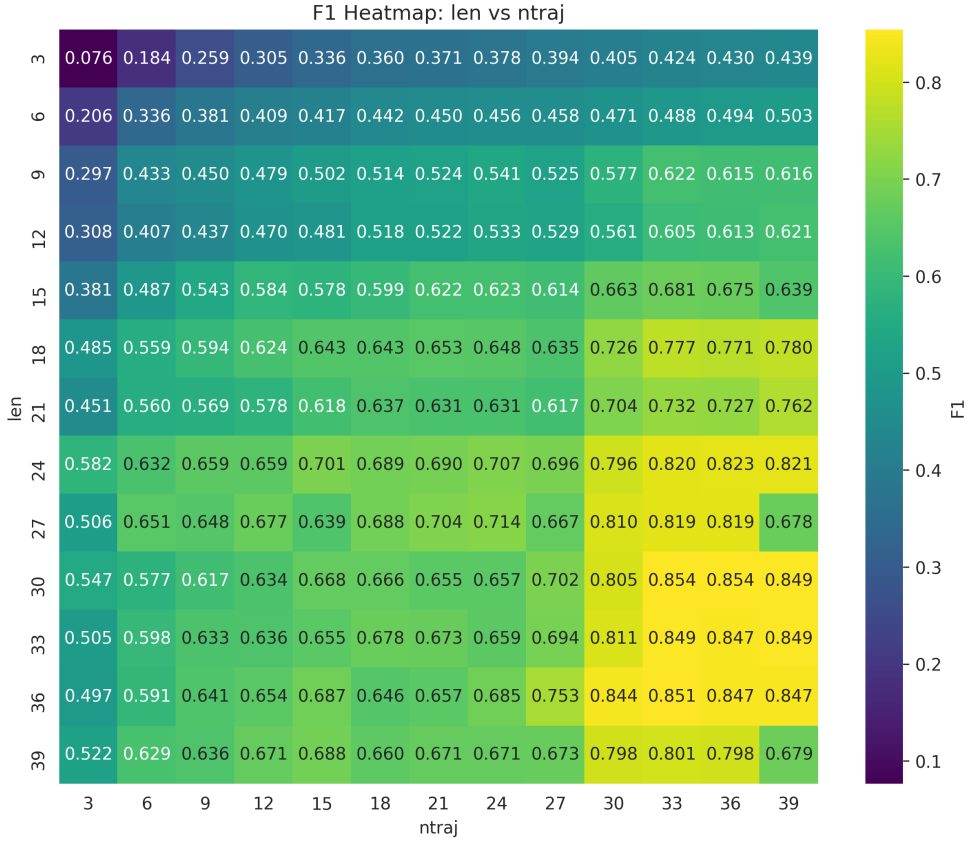


The graph size does not have significant effect on performance either way

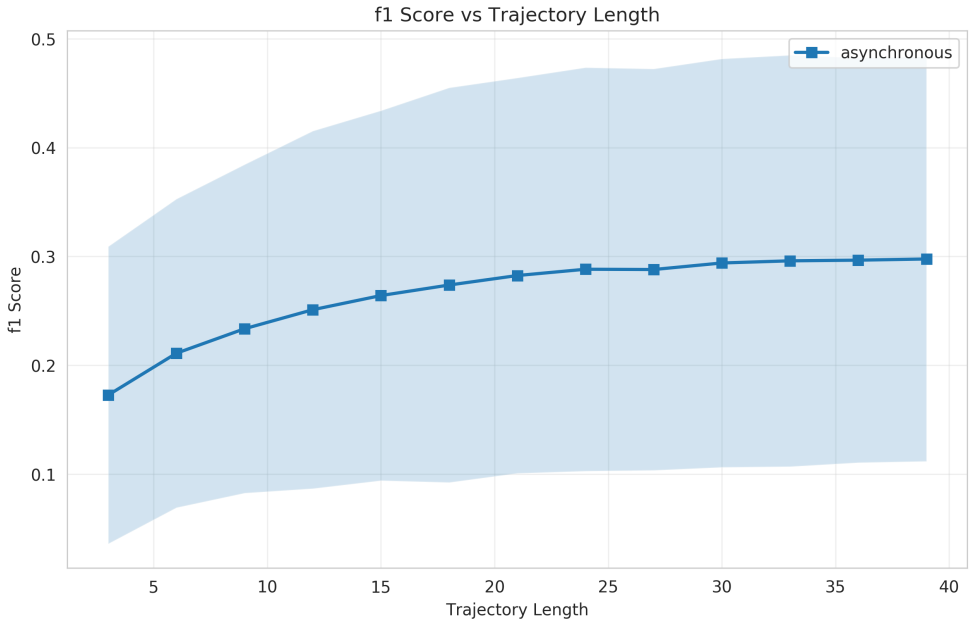
- it is more data to analyze but also a more complicated network structure to derive.

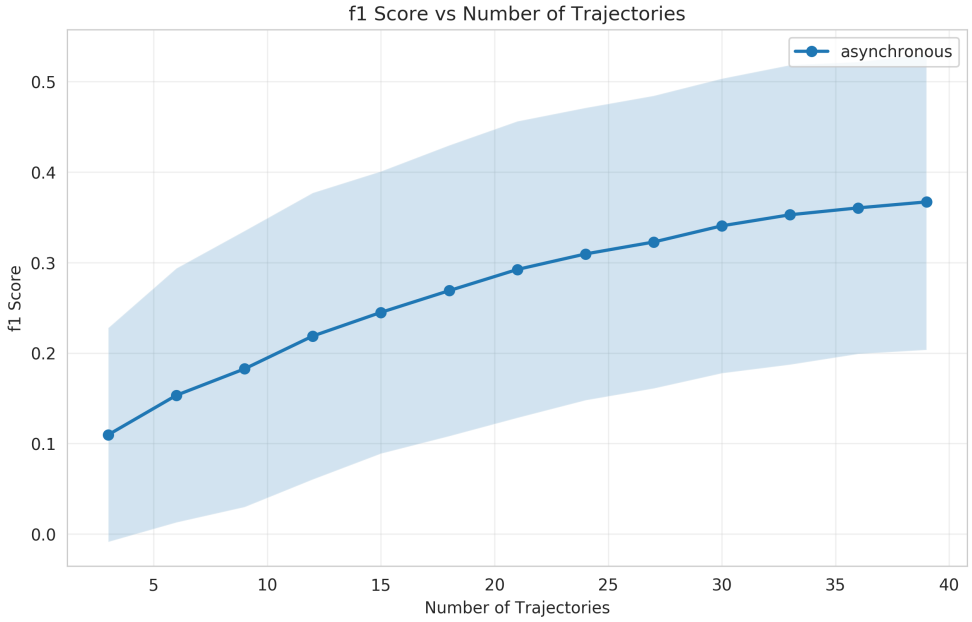


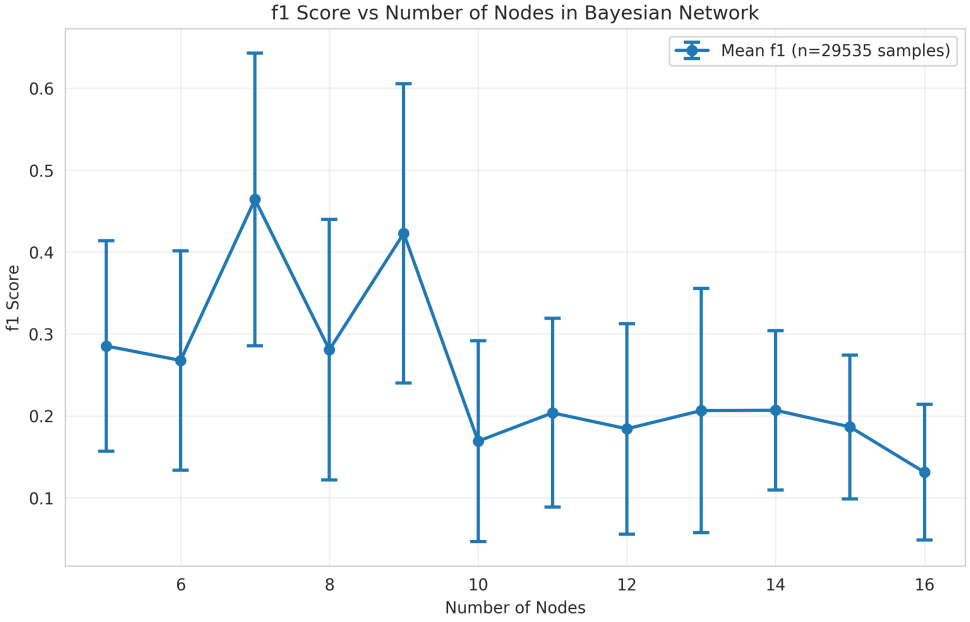
When plotting against both trajectory length and dataset size, there is no significant correlation between the two - they both contribute positively to the result independently.

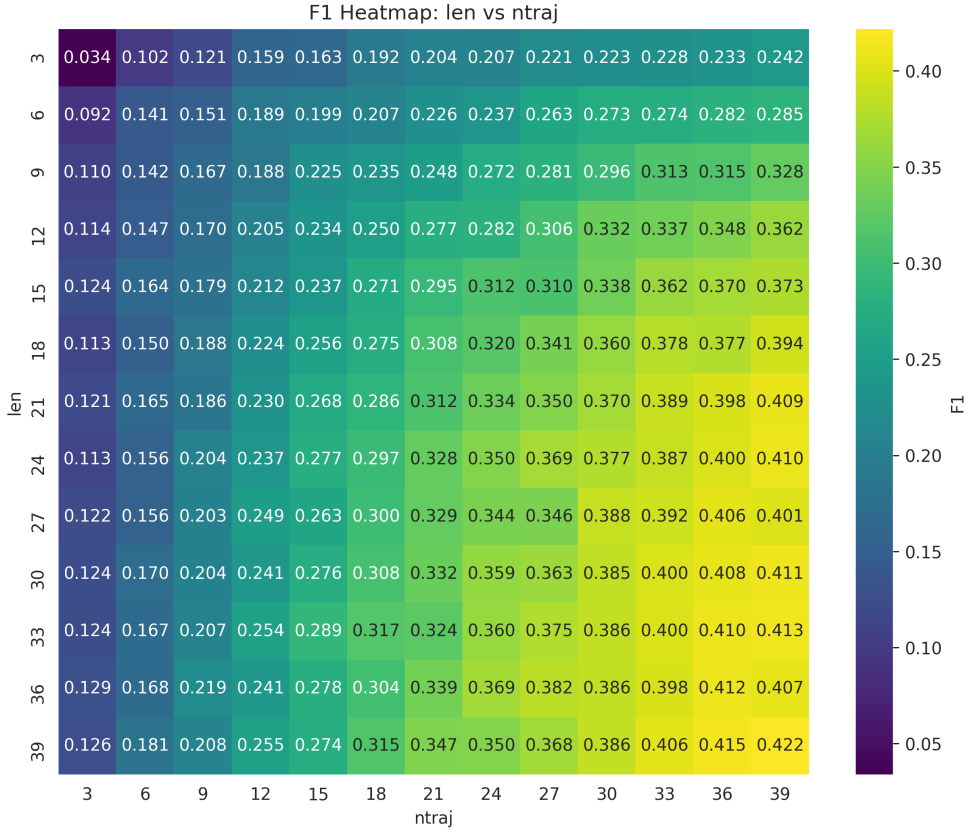


The asynchronous datasets results are similar. The main difference we can notice is that the trajectory length caps out more gradually - likely because for a given trajectory it is not guaranteed when the next step will get picked up. The correlations are also more clearly observes, likely due to large total data volume as synchronous dataset omits some parameter combinations that are possible when in asynchronous mode.

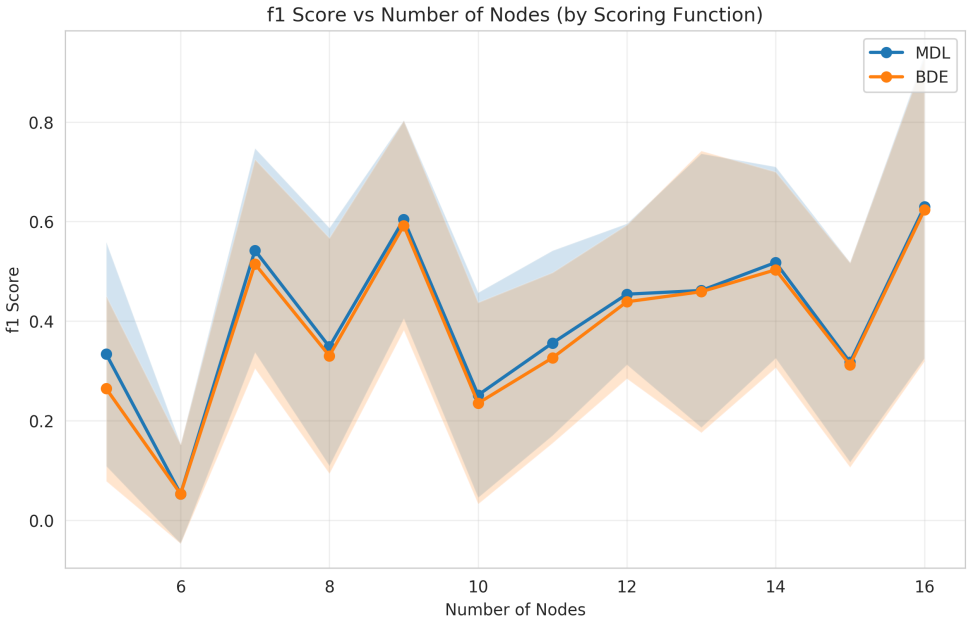

f1 Score vs Trajectory Length
asynchronous
f1 Score
Trajectory Length
0.1
0.2
0.3
0.4
0.5
5
10
15
20
25
30
35
40


f1 Score vs Number of Trajectories
asynchronous
f1 Score
Number of Trajectories
0.0
0.1
0.2
0.3
0.4
0.5
5
10
15
20
25
30
35
40

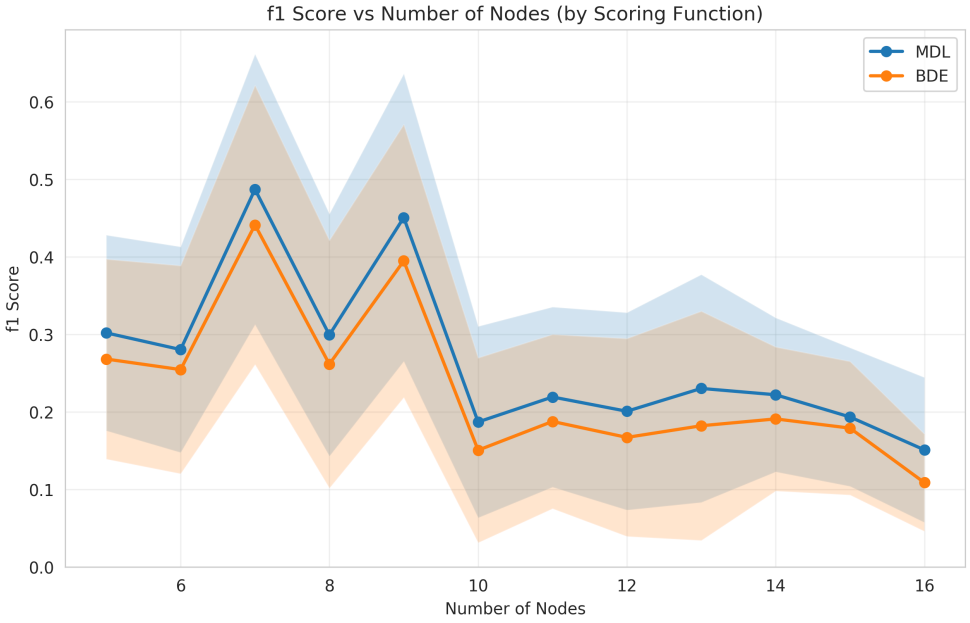

f1 Score vs Number of Nodes in Bayesian Network
Mean f1 (n=29535 samples)
f1 Score
0.6
0.5
0.4
0.3
0.2
0.1
6
8
10
12
14
16
Number of Nodes



As for the scoring method, we can see that while the performance is generally comparable, MDL stably gives slightly better results than BDE. There are no significant gaps in performance that would clearly favour one method over enough - MDL is just stably slightly better across the board.



Similar picture can be observed for asynchronous graphs, with advantage somewhat more apparent.

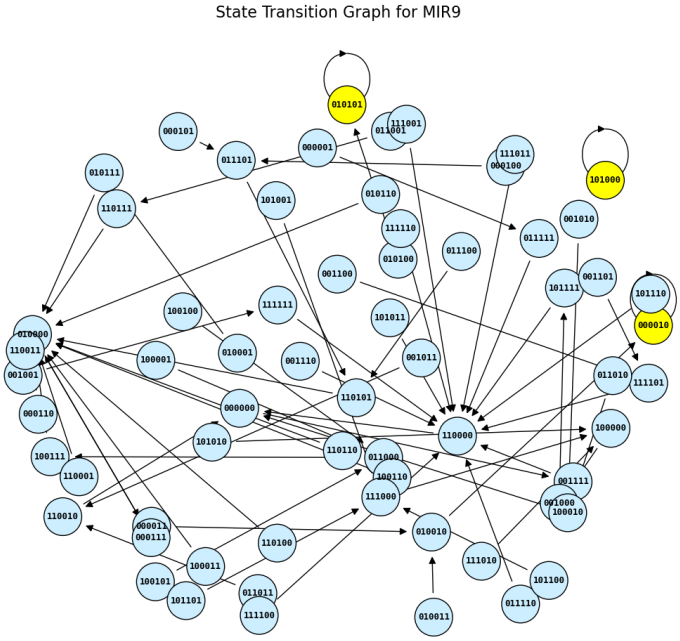


In the next part we therefore choose MDL as our only scoring method.

6 Reconstructing the model of a real-life biological mechanism

6.1 Chosen Model

MIR-9-NEUROGENESIS (**ID = 88**) Boolean Network model was chosen from Biodivine repository, because it has small number of nodes(6), which reduces computational complexity and should give better accuracy, and has zero inputs, which eliminates the need to simulate external environmental changes, allowing for a straightforward simulation of the network's internal dynamics. Below you can see graph for MIR9 model where yellow nodes are attractors.



Variables: $V = \{x_0, x_1, x_2, x_3, x_4, x_5\}$
Predictor functions: $f_0(x_2) = x_2$

$f_1(x_3, x_5) = (\neg x_3 \wedge x_5) \vee x_3$ (equivalent to $x_3 \vee x_5$)
$f_2(x_1, x_4) = \neg x_4 \wedge \neg x_1$
$f_3(x_0, x_4) = \neg x_4 \wedge \neg x_0$
$f_4(x_0, x_3) = \neg x_3 \wedge \neg x_0$
$f_5(x_0, x_4) = \neg x_4 \wedge \neg x_0$

6.2 Datasets Generation

A total of 1014 datasets were generated using BooleanNetwork class method create_dataset from `BooleanNetwork.py` to assess the algorithm's robustness. The datasets covered a combination of the following parameters:

- **Number of data points:** 13 values ranging from 3 to 39 (step size of 3).
- **Trajectory length:** 13 values ranging from 3 to 39.
- **Update mode:** Synchronous and Asynchronous.
- **Sampling frequency:** 1, 2, and 3.
- **Initialization:** Random starting state for all simulations.

The total number of combinations is calculated as $13 \times 13 \times 2 \times 3 = 1014$ datasets.

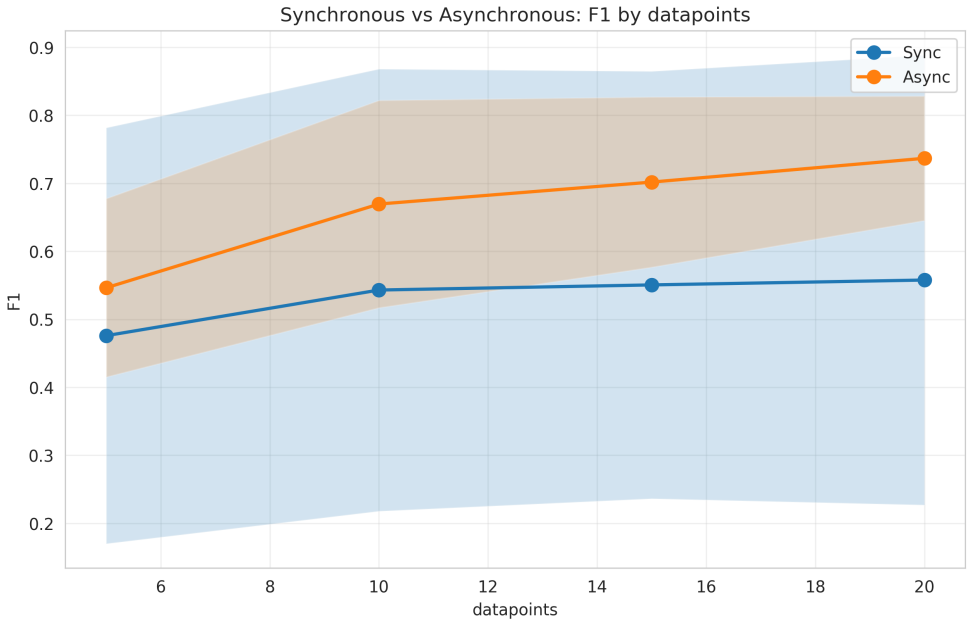
6.3 Reconstructing

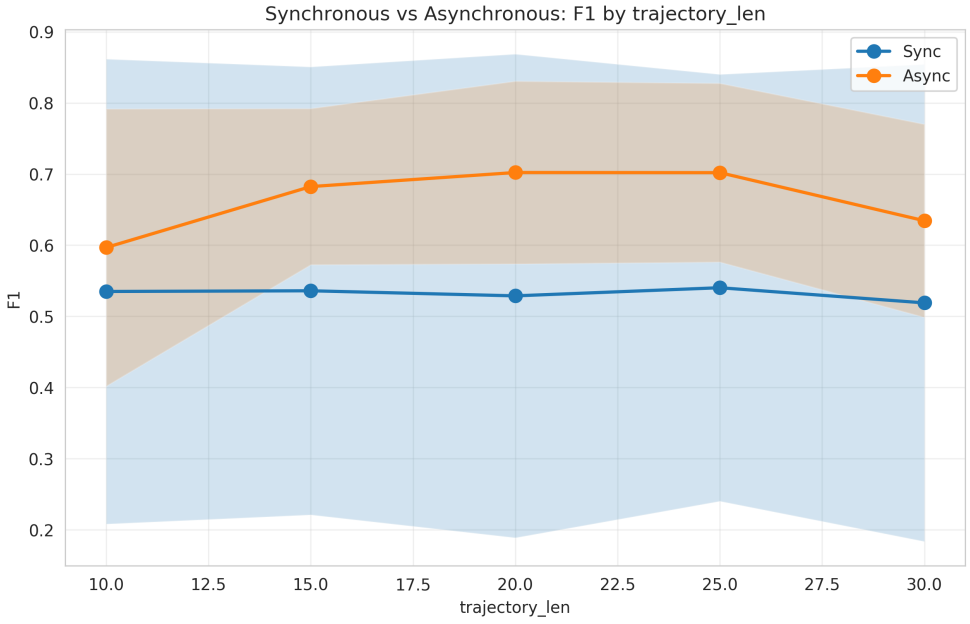
The next step was to reconstruct Bayesian Network with MDL scoring function, which was chosen based on the insights from the first Part. This allowed for the batch processing of all 1014 datasets and the automatic calculation of all metrics defined in Part 1.

The evaluation process was analogous to evaluation in part 1. While some technical modifications were needed in order to account for a different data format (.csv file, column names, number formatting), the general approach was the same for both evaluations, and the script could be easily extended. A function was added to compare the results on sync and async datasets.

6.4 Results

The results on MIR9 dataset are less clear in comparison to artificial data. However, some positive correlation with the dataset size can be nevertheless observed:





This is likely due to the real-life being noisier.

7 Contributions

Authors (given in alphabetical order) and their contributions to the project:

- Czapiewska Magdalena:
  - Implementing attractor detection in the state space for synchronous update mode
  - Implementing attractor detection in the state space for asynchronous update mode (two approaches)
  - Implementing functions for generating datasets, taking into account update mode, number of data points, trajectory length, sampling frequency and fraction of transient states
  - Implementing a script for generating datasets for synthetic networks
  - Implementing a script to run BNFinder2 on datasets derived from synthetic networks
  - Choosing, describing and implementing evaluation metrics
  - Creating a script for evaluating BNFinder2 results
  - Report: Attractor detection implementation (2), Datasets generation (3), Evaluation of the accuracy of the reconstructed network (only description of the metrics in 5.1, without results in 5.2)
- Khodina Anastasiya(whole Part 2):
  - Choosing the appropriate model in Part 2
  - Interpreting/Processing chosen model code
  - Creating inference of BN class for the model
  - Generating trajectories for the model
  - Reconstructing the BN from created trajectories
  - Calculating metrics for the model reconstruction from Part 2
  - Creating graphical representation for the model
  - Describing Part 2 in the report (6.1, 6.2, 6.3, without results in 6.4)
- Nechaieva Veronika:
  - Organisational activity (Github etc.)
  - Reworking the code for Boolean networks; readability
  - Saving and loading the network ground truth structure
  - Implementing the dataset save/load in a proper format for BNFinder2
  - Converting the Python scripts to work with command-line arguments
  - Testing BNF base functionality
  - Processing and converting of the aggregated BNFinder results

  - Implementing the graphs plotting (single-argument plots, heatmaps, metrics comparison)
  - Evaluating the network-finding accuracy according to the implemented metrics
  - Report: result analysis (for subcases as well as the general cases) for Part 1 (5.2) and Part 2 (6.4).
  - Editing and reviewing

- Znamierowski Mikołaj:
  - Gathering the group and initiating the work.
  - Finding and sharing crucial resources regarding BNFinder2.
  - Implementing the main code for Boolean network class: setting proper structure, functionalities, transitions, etc.
  - Implementing functions that create the datasets and save them in a file.
  - Explaining to the rest of the group how to use the code.
  - Describing the most important parts of the above implementations in the report (i.e. writing the first section).