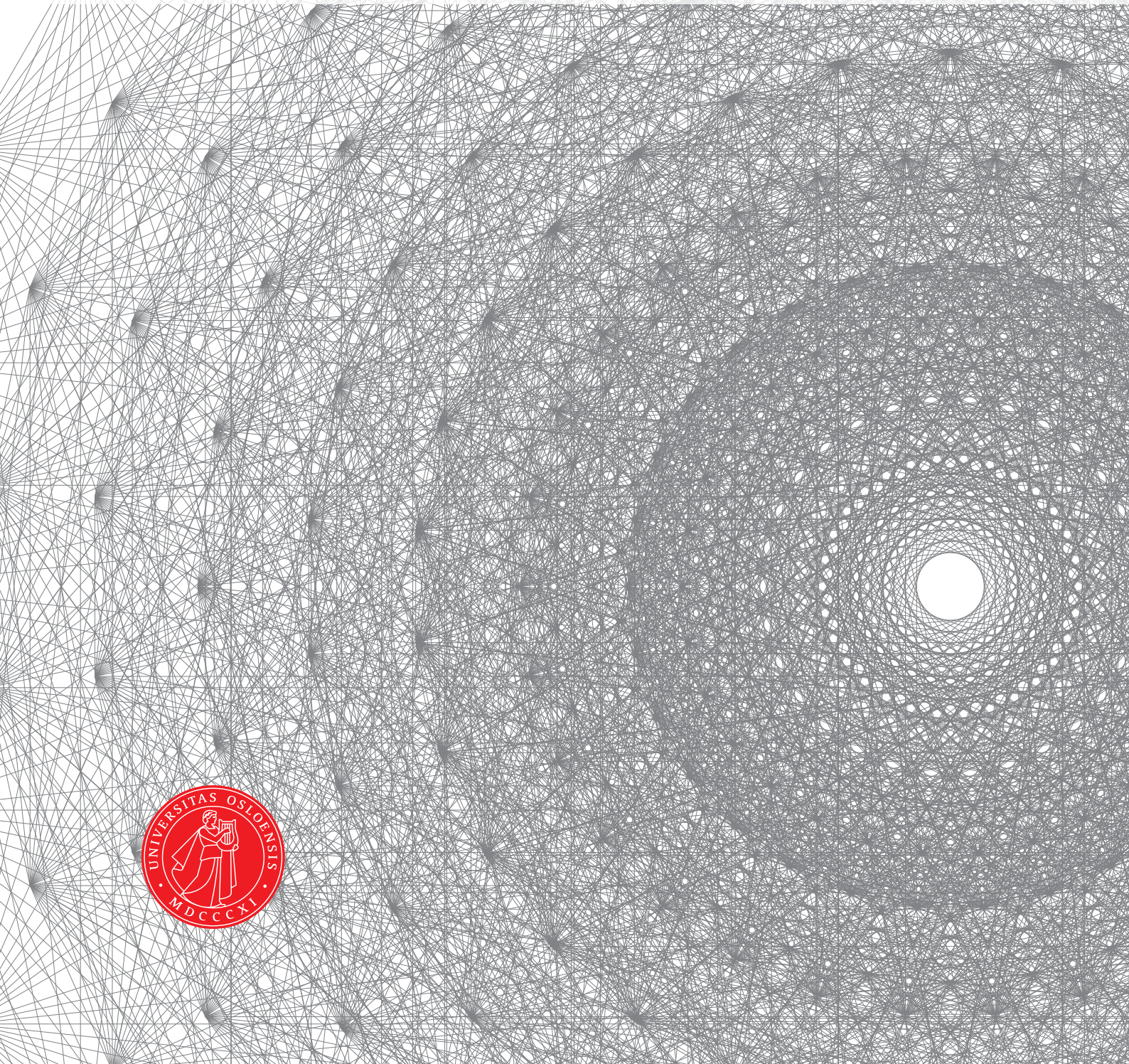
UiO : Department of Mathematics
University of Oslo

Causal discovery with Bayesian networks

Rayyan Syed

Master's Thesis, Spring 2023



Name of masterprogram: Data Science,Statistics and Machine Learning
Scope of the project: 60 study points

Acknowledgements

I want to start with tanking my supervisor Johan Pensar Professor of Data Science at the Department of Mathematical at UIO.This journey has been long.It would have been hard to be able to come through without his insight.

Abstract

One of the most widely used tools for causal discovery is based on causal models represented by the framework of Bayesian network. In the most challenging cases of causal discovery the underlying BN structure is not known and must be computed in a way that it takes into account the uncertainty that exist when trying to predict the underlying structure. The structure uncertainty can then be transformed into an uncertainty regarding a causal relationship between variables reflecting the strength of how likely a causal relationship is given data assumed to come from the underlying causal model. There are different methods account for such uncertainty. We will focus on Bayesian model averaging over structures implemented trough Markov Chain Monte Carlo(MCMC) and a state-the-art dynamic programming algorithm.The general way of expressing parameters for a causal model is through the use of conditional probability tables CPTs. It has been demonstrated that more expressive models that account for additional structures in each CPT may lead to improved predication over traditional causal models. We will represent the regularities within CPTs through more refined independency relations, defined according to the concept of context-specific independence(CSI), in the form of CSI-trees which are learned with a greedy algorithm. To identify plausible models, we use a score-equivalent Bayesian score. An optimal combination of these models will be found with the help of Bayesian model averaging in order to find the posterior distribution over the causal target of interest. These methodologies where tested on synthetic data generated from known benchmark Bayesian networks. A comparison between CPTs and CSI-trees with the help of AUC show that no significant improvement was made on the tested networks. However for some data sizes some improvement could be seen. One reason might be that no exact CSI-tree representation of the conditional distribution exist for these networks,since the true distributions are defined through CPD tables. Another reason might be that it was necessary to regulate the model fit with a model structure prior to avoid overfitting in the learning process. The prior used in this work might have been suboptimal. A comparison between MCMC and state-the-art dynamic programming algorithm shows that the result under AUC are similar,however the convergence of the MCMC over structure for some networks tested is slow.

Contents

CHAPTER 1

Introduction

Causality is an important concept in most scientific studies. While the optimal approach for inferring causal relationships is from controlled experiments, such experiments can often be hard to perform for various reasons. For this reason one would like to identify causal relations based on data generated through passive observation. One framework that has received a lot of attention is the framework of do-calculus.The typical assumption in this approach is that the causal structure over the variables is known and belongs to some class of (causal) Bayesian networks. In general this assumption cannot always be made with ease. In the most challenging setting the BN structure is completely unknown and must be learned from available data, a problem known in the graphical model framework as structure learning. This study will focus on development of a method for inferring the presence or absence of a causal relationship between a pair of variables without the help of experimental data. We will define causal relationships with help of Bayesian networks, in which the presence of a causal relationship from X to Y corresponds to the existence of a directed path from X to Y in BN structure G. Notation $X \rightsquigarrow Y$ will be used to denote such a path. We will phrase the problem of causal relationships this way so that the problem becomes a structure learning problem. Given the uncertainty involved in structure learning, we will compute the posterior of $X \rightsquigarrow Y$ through Bayesian model averaging:

$$p(X \rightsquigarrow Y|Data) = \sum_{G \in \mathcal{G}(X \rightsquigarrow Y)} p(G|Data), \tag{1.1}$$

where $p(G|Data)$ is the posterior probability of G given the data,and $\mathcal{G}(X \rightsquigarrow Y)$ are all DAGs that contain $X \rightsquigarrow Y$. The computational cost of calculating (1.1) is large. Using state-the-art dynamic programming the calculation can be done exactly for around 20 variables. Beyond that one must resort to other methods such as Markov Chain Monte Carlo(MCMC). One of the goals of our study will be to compare one state-the-art dynamic programming algorithm [Pen+20] against MCMC in terms of Bayesian model averaging. In addition we want to study the effect of the incorporation of local structure in the models when evaluating (1.1). By local structure,we refer to a structure describing local properties of the relationship between a node and its parents (or direct cause), in particular we will use tree-based structure that are able to capture a form of context-specific independence(CSI). Earlier research has shown that local structure learning can improve model accuracy in terms of estimation of the joint distribution that factorizes over the graph structures G. We aim to test

the hypothesis that local structures will prove themself to be beneficial for causal discovery as we hope they will aid in distinguishing between Markov equivalent graphs in terms of the conditional independence that they imply.
We will start our journey with the notion of conditional independence. Further we will look into necessary graph theoretical concepts. What follows after that is combining the notion of conditional independence with defined graph theoretical concepts. After that we will go through our chosen structured based learning scores, before proceeding to Bayesian model averaging and some theory on the exact algorithm. We will finish with a simulation study and a discussion about the results.

CHAPTER 2

Theory

2.1 Conditional Independence

In this work our interest is in modeling the joint distribution over a set of categorical random variables. We use the notation $P(X_1, ..., X_n)$ for the probability mass function of random variables $\mathcal{X} = \{X_1, ...X_n\}$. A particularly important concept in the considered framework is conditional independence.

Definition 2.1.1. [KF09, Definition 2.3] We say that an event α is conditionally independent of event β given event γ in P,denoted

$$P \models (\alpha \perp \beta|\gamma),\ if\ P(\alpha|\beta \cap \gamma) = P(\alpha|\gamma)\ orP(\beta \cap \gamma) = 0$$

Definition 2.1.2. [KF09, Definition 2.4] Let $\mathbf{X}, \mathbf{Y}, \mathbf{Z}$ is a set of random variables.We say that $\mathbf{X}$ is conditionally independent of $\mathbf{Y}$ given $\mathbf{Z}$ in a distribution P if P satisfies ($\mathbf{X}$=$\mathbf{x}$ $\perp$ $\mathbf{Y}$=$\mathbf{y}|\mathbf{Z}$=$\mathbf{z}$) for all $\mathbf{x} \in Val(\mathbf{X})$, $\mathbf{y} \in Val(\mathbf{Y})$ and $\mathbf{z} \in Val(\mathbf{Z})$. The variables in the set $\mathbf{Z}$ are often said to be observed. If the set $\mathbf{Z}$ is empty, then instead of writing $(\mathbf{X} \perp \mathbf{Y}|\emptyset)$, we write $(\mathbf{X} \perp \mathbf{Y})$ and say that $\mathbf{X}$ and $\mathbf{Y}$ are marginally independent.

As an alternative characterization of conditional independence we can also use the following statement.

Proposition 2.1.3. [KF09, Proposition 2.3]
The distribution P satisfies $(X \perp Y|Z)$ if and only if
$P(X, Y|Z) = P(X|Z)P(Y|Z)$

Definition 2.1.4. [KF09, Definition 3.2] Let P be a distribution over $\mathcal{X}$. We define $I(P)$ to be the set of independence assertions of form $(\mathbf{X} \perp \mathbf{Y}|\mathbf{Z})$ that holds in P.

2.2 Graphs

A graph is a structure made up of nodes denoted by $\mathcal{X}$ and edges denoted by $\mathbf{E}$. Every pair of nodes are either none-connected or connected by an edge, which is either undirected,$X_i - X_j$, or directed $X_i \rightarrow X_j$(or $X_i \leftarrow X_j$). Some particularly important attributes of a graph are listed below.

- **Parent/child:** X_j is parent to X_i in the graph whenever we have $X_j \rightarrow X_i$. Analogously, X_j is said to be the child of X_i. We denote all parents of X_i by Pa_{X_i}.
- **Degree of a node:** Number of nodes it is directly connected to through an edge.
- **degree of graph**: The maximal of all node degrees.
- **In-degree of a node:** The number of parents the node has,that is ,$|Pa_{X_i}|$.

In addition to these basic properties, there are additional properties that will be of relevance to this work. These will be listed in the following and they are taken from *Koller and Friedman* [KF09]. Some of the definitions are slightly modified.

Definition 2.2.1. [KF09, Definition 2.15] We say that $X_1, ..., X_k$ form a path in the graph $\mathcal{K} = (\mathcal{X}, E)$ if, for every $i = 1..., k-1$,we have either $X_i \rightarrow X_{i+1}$ or $X_i - X_{i+1}$. A path is directed if, for at least one i we have $X_i \rightarrow X_{i+1}$.

Definition 2.2.2. [KF09, Definition 2.16] We say that $X_1, ..., X_k$ form a trail in the graph $\mathcal{K} = (\mathcal{X}, E)$ if, for every $i = 1, ..., k-1$, we have $X_i \rightleftharpoons X_{i+1}$.

Definition 2.2.3. [KF09, Definition 2.17]
A graph is connected if for every X_i,X_j there is a trail between X_i and X_j.

Definition 2.2.4. [KF09, Definition 2.18] We say that X is an ancestor of Y in $\mathcal{K} = (\mathcal{X}, E)$, and that Y is descendant of X, if there exist a direct path $X_1, ..., X_k$ with $X_1 = X$ and $X_k = Y$.We use $Deschendants_X$ to denote $X's$ descendants, $Ancestetors_X$ to denote $X's$ ancestor, and $NoneDecsendants_X$ to denote the set of node in $\mathcal{X} - Descendants_X$.

The ancestors of a node X in a graph will be the parents of X plus the parents of the parents of X and so on, while the descendants of X is all nodes that has X as, ancestor.

Definition 2.2.5. [KF09, Definition 2.19]
Let $G = (\mathcal{X}, E)$ be a directed graph. An ordering of the nodes $X_1, ..., X_n$ is a topological ordering relative to G if $i < j$ whenever $X_i \rightarrow X_j \in E$.

Definition 2.2.6. [KF09, Definition 2.20] A cycle in K is a directed path $X_1, ..., X_k$ where $X_1 = X_k$.A graph is acyclic if it contains no cycles.

We will in this work focus on directed acyclic graphs (DAGs). In Figure (2.1) is an example of a directed graph that is not a DAG due to the existence of a direct cycle.

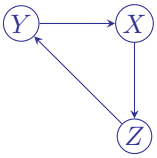


Figure 2.1

Definition 2.2.7. [KF09, Definition 2.21]
Let K be PDAG(partially directed acyclic graph) over $\mathcal{X}$. Let $K_1, ..., K_l$ be a disjoint partition of $\mathcal{X}$ such that:

- the induced subgraph over K_i contains no directed edges;
- for any pair of nodes $X \in K_i$ and $Y \in K_j$ for $i < j$, an edge between X and Y can only be a directed edge $X \to Y$

Each component K_i is called a chain component

Importantly, in graphical model framework, a graph is used to represent the dependence structure over the involved variables. The correctness of the implied independence statements with respect to some reference set is given by the following definition.

Definition 2.2.8. [KF09, Definition 3.3] Let K be any graph object associated with a set of independencies $I(K)$. We say that K is an I-map for a set of independencies I if $I(K) \subseteq I$

2.3 Bayesian Networks

Since we will work with Bayesian Networks we will call the DAG a BN structure. In this work, we will focus on Bayesian networks, for which the dependence structure is represented by a DAG. More specifically, the dependence structure encoded by a DAG can be characterized by the local Markov property.

Definition 2.3.1. [KF09, Definition 3.1]
A BN structure is a DAG G whose nodes represent random variables $X_1, ..., X_n$.The structure G encodes the following set of conditional independency assumptions:

$$X_i \perp NoneDesc_{X_i} | Pa_{X_i} \ , \ i = 1, ..., n$$

which are known as the local independencies in G and denoted by $I_L(G)$.

In addition to the local independencies there are additional independencies that are implied by $I_L(G)$ through the so called semi-graph axioms [KF09].

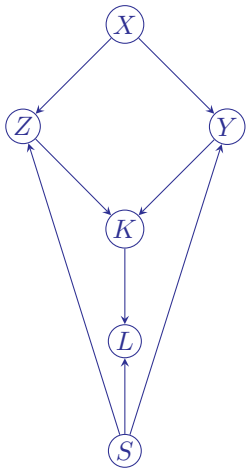


Figure 2.2

To explain the concept of using DAG as BN structure, we will use the example DAG in Figure 2.2. There are two nodes in Figure 2.2 that has an empty parent set. Here the local property still holds,$Pa_X = Pa_S = \emptyset$ and the local Marcov property implies marginal independence. We will now list up all the independencies in $I_L(G)$:
$X \perp S$
$S \perp X$
$L \perp X, Z, Y | K, S$
$K \perp X, S | Z, Y$
$Z \perp Y | S, X$
$Y \perp Z | S, X$

In addition to these independencies there are other so called global independence structures that can't be captured directly using definition 2.3.1. Global independence defines the conditional independencies for a DAG more generally. It is defined through the notion of d-seperation, which is a graph-theoretic criterion for reading off independence statements directly from the graph.

Definition 2.3.2. [KF09, Definition 3.6]
Let G be an BN structure,and $X_1 \rightleftharpoons .. \rightleftharpoons X_n$ a trail in G.Let $\mathbf{Z}$ be a subset of observed variables.The trial $X_1 \rightleftharpoons .. \rightleftharpoons X_n$ is active given Z if

- whenever we a v-structure $X_{i-1} \rightarrow X_i \leftarrow X_{i+1}$ then X_i or one of its descendants is in $\mathbf{Z}$;
- no other node along the trail is in Z.

Definition 2.3.3. [KF09, Definition 3.7]
Let **X,Y,Z** be three sets of nodes in G. We say that $\mathbf{X}$ and $\mathbf{Y}$ are d-separated given $\mathbf{Z}$, denoted $dsep_G(\mathbf{X}; \mathbf{Y}|\mathbf{Z})$, if there is no active trail between any node $X \in \mathbf{X}$ and $Y \in \mathbf{Y}$ given $\mathbf{Z}$. We use $I(G)$ to denote the set of independencies that correspond to d-separation:

$$I(G) = \{(\mathbf{X} \perp \mathbf{Y}|\mathbf{Z}) : d - sep_G(\mathbf{X}; \mathbf{Y}|\mathbf{Z})\}$$

Theorem 2.3.4. [KF09, Theorem 3.3] If a distribution P factorizes according to G,then $I(G) \subseteq I(P)$

Theorem 2.3.4. ensures soundness of d-separation.
We denote the independencies not captured by local independency by $I_p(G)$.Definition 2.3.3 can be used both to find $I_L(G)$ and $I_p(G)$. The union of $I_L(G)$ and $I_p(G)$ forms $I(G)$. Some of the independencies in $I_p(G)$ for DAG G are:

$K \perp X|Z, Y$
$X \perp L|Z, Y, S$
$Z \perp L|K, S$
$Y \perp L|K, S$
$S \perp K|Z, Y$

For two distinct DAGs $G_1 = G_2$ we might have that they encode the same dependence structure,that is $I(G_1) = I(G_2)$.This property is known as I-equivalence(or Markov equivalence).

Definition 2.3.5. [KF09, Definition 3.9]
Two graph structures K_1 and K_2 over X are I-equivalent if $I(K_1) = I(K_2)$. The set of all graphs over $\mathcal{X}$ is partitioned into a set of mutually exclusive and exhaustive I-equivalence classes, which are the set of equivalence classes induced by the I-equivalence relation.

Inspired by d-separation I-equivalence between two DAG's can easily be tested by a simple graph-based algorithm.

Definition 2.3.6. [KF09, Definition 3.10]
The skeleton of a Bayesian network graph G over $\mathcal{X}$ is an undirected graph over $\mathcal{X}$ that contains an edge $\{X, Y\}$ for every edge (X, Y) in G.

Definition 2.3.7. [KF09, Definition 3.11]
A v-structure $X \rightarrow Z \leftarrow Y$ is an immorality if there is no direct edge between X and Y if there is such an edge it is called a covering edge for the v-structure

Based on the concepts of skeleton and immoralities, we can then characterize I-equivalence as follows:

Theorem 2.3.8. [KF09, Theorem 3.8] Let G_1 and G_2 be two graphs over $\mathcal{X}$.Then G_1 and G_2 have the same skeleton and the same set of immoralities if and only if they are I-equivalent

Figure 2.3 and figure 2.4 shows the two I-equivalence classes for three nodes X, Y, Z that are connected as a chain.For the DAG's in Figure 2.3, we have the implied independency statement $X \perp Y|Z$ while for DAG in Figure 2.4 we have that $X \perp Y$.

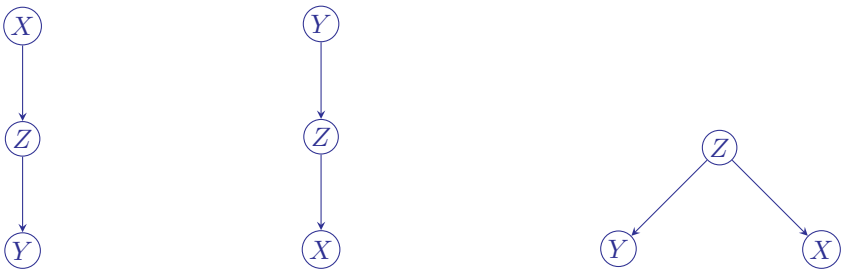


Figure 2.3

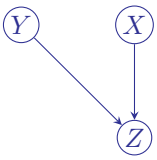


Figure 2.4

The I-equivalence class of the DAG's in figure 2.3 can be seen as:

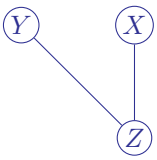


Figure 2.5

Figure 2.5 is the cPDAG of the I-equivalence class shown in Figure 2.3. A cPDAG is a PDAG that illustrates I-equivalent DAGs.
A Bayesian network is a tuple containing a DAG structure and a set of conditional probability distributions (CPDs) that define the joint distribution under the associated DAG. Generally one can write any joint distribution over a set of random variables $X_1, ..., X_k$ as a product using the chain rule: .

$$P(X_1, ..., X_k) = P(X_1)P(X_2|X_1)P(X_3|X_1, X_2)...P(X_k|X_1, ..., X_{k-1})$$

This decomposition holds for any factorization order. In the case of Bayesian networks, we have a similar type of factorization, which is known as the chain rule for Bayesian networks. There is a fundamental connection between the view of a DAG as a dependence structure and a specification of how to factorize a joint distribution.

Definition 2.3.9. [KF09, Definition 3.4]
Let G be a BN structure (i.e DAG) over the variable $X_1, .., X_n$. We say that a distribution P over the same space factorizes according to G if P can be expressed as a product

$$P(X_1, ..., X_n) = \prod_{i=1}^{n} P(X_i | Pa_{X_i}), \tag{2.1}$$

is called a chain rule of Bayesian Network

Definition 2.3.10. [KF09, Definition 3.5]
A Bayesian network BN is a pair $B = (G, P)$ where P factorizes over G, and where P is specified as a set of CPDs associated with the nodes in G.The distribution P is often annotated P_B

Thus, by specifying the CPD of each node given its parents, we specify the joint distribution under a specific DAG.

Theorem 2.3.11. [KF09, Theorem 3.1]
Let G be a BN structure over a set of random variables $\mathcal{X}$ and P be a joint distribution over the same space. If G is an I-map for P, then P factorizes according to G

Theorem 2.3.12. [KF09, Theorem 3.2]
Let G be a BN structure over a set of random variables $\mathcal{X}$ and let P be a joint distribution over the same space. If P factorizes according to G, then G is an I-map for P.

Theorem 1.3.13 and 1.3.14 looked at together ensures that $I(G) \subseteq I(P)$ whenever P factorizes over G.

Generally one assumes that a Bayesian network structure is an minimal I-map.

Definition 2.3.13. [KF09, Definition 3.13]
A graph K is a minimal I-map for a set of independencies I if it is an I-map for I,and if the removal of even a single from K renders it not an I-map.

A procedure for finding minimal I-maps given a variable ordering$X_1, ..., X_n$ is to use the definition of local independency iteratively by selecting minimal parent-sets for the nodes that follow the definition of local independency.

- Pick a minimal subset $U \subset (X_1, ...X_{i-1})$ for which

$$X_i \perp (X_1, ..., X_{i-1}) \setminus U | U$$

.

- set Pa_{X_i} to be U

The standard way of representing the CPDs of Bayesian network with categorical variables is to use conditional probability tables (CPD-tables or CPTs). A CPT is a lists the conditional distributions for each configuration on the parental variables. As an example consider a Bayesian network over the DAG in Figure 2.4 for the joint distribution factorizes as

$$P(X, Y, Z) = P(X)P(Y)P(Z|X, Y)$$

Now, assuming binary variables, we have would represent the above CPDs with CPTs in table 2.1-2.3,where every row represents a CPD under a parent configuration.

condition	z^1	z^0
x^0, y^0	p_1	$1 - p_1$
x^0, y^1	p_2	$1 - p_2$
x^1, y^1	p_3	$1 - p_3$
x^1, y^0	p_4	$1 - p_4$

Table 2.1: CPD table for $P(Z|X, Y)$

x^1	x^0
p_5	$1 - p_5$

Table 2.2: CPD table for $P(X)$

x^1	x^0
p_6	$1 - p_6$

Table 2.3: CPD table for $P(Y)$

An alternative way of representing CPDs of a variable is through CSI trees where CSI stands for context specific independence. This representation is useful when the certain CPDs within an CPT are identical. The main disadvantage with CPT is the number of parameters that has to be defined increases exponentially with the number of parents. Estimating the parameters accurately becomes harder when the number of parameters are large compared to the available data. This is one reason for the need to restrict the number of parameters. In many real life situation the need for a reduction in parameter size comes naturally and can be characterized through the notion of context-specific independence. CSI-trees follows something called CSI independence.

Definition 2.3.14. [KF09, Definition 5.1]
Let **X,Y,Z** be pairwise disjoint set of variables,let **C** be a set of variables(that might overlap with $\mathbf{X} \cup \mathbf{Y} \cup \mathbf{Z}$),and let $\mathbf{c} \in Val(\mathbf{C})$.We say that **X** and **Y** are contextually independent given **Z** and the context **c** denoted ($\mathbf{X} \perp_c \mathbf{Y}|\mathbf{Z},\mathbf{c}$) if
$P(\mathbf{X}|\mathbf{Y},\mathbf{Z},\mathbf{c}) = P(\mathbf{X}|\mathbf{Z},\mathbf{c})$ whenever $P(\mathbf{Y},\mathbf{Z},\mathbf{c}) > 0$

For probability distribution $P(\mathbf{X}, \mathbf{Y}|\mathbf{Z} = z)$ some joint events of Z might be equal. If one uses a CPT to represent the CPDs one has to define all conditional distributions regardless of them being equal or not. This motivates the construction of other types of representations. One such representation is a CSI-tree. As an example the CSI-tree of Table 2.1 is:

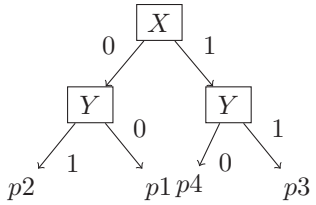


Figure 2.6

CSI can be seen as a generalization of conditional independence. If the CPD-table for $P(Z|X, Y)$ instead look like the one in Table 2.4 where we have identical CPDs on the third and fourth row,we could represent it more compactly using the CSI-tree in Figure 2.7.

condition	z^1	z^0
x^0, y^0	p_1	$1 - p_1$
x^0, y^1	p_2	$1 - p_2$
x^1, y^1	p_3	$1 - p_3$
x^1, y^0	p_3	$1 - p_3$

Table 2.4: CPD table for $P(Z|X, Y)$

For table 2.4 the new CSI-look like:

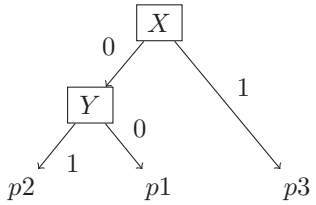


Figure 2.7

In Figure 2.7 we have that $P(Z|X = 1, Y) = P(Z|X = 1)$,representing a CSI of the form $(Z \perp_c Y|X = 1)$ that is, the context is specified by $X = 1$.Each branch represents a joint conditional configuration. If some variable is not included in a branch, it means that the value of that variable does not influence the conditional distribution regardless if one includes it or not. The leafs contains the parameters representing a CPD. In Figure 2.7 $p_1 = P(Z = 1|X = 0, Y = 0)$,$p_2 = P(Z = 1|X = 0, Y = 1)$ and $p_3 = P(Z = 1|X = 1)$.The reason for testing CSI-trees as an alternative

representation for the node-specific CPDs is to see if it is beneficial form a causal discovery point of view. For example,we want to study if the additional restrictions imposed by CSI can help in orientating additional edges within an I-equivalence class.

2.4 Bayesian networks as causal models

A causal model has the same form as a BN:A tuple consisting a BN structure and a set of CPD-tables grouping parameters on the basis of nodes conditioned on there parents. Each CPD table is called a causal mechanism.The causal mechanism has the same form as the CPD table but is the output of a stochastic function which changes and it describes the functional relation between a node and its parents.The difference between a BN and a causal model is the interpretation of the edges. In a causal model we assume that the edges between the node and its parents represent direct causal relationships with respect to the observed variable. With this interpretation the causal mechanisms will change when fixing specific values of the parental configurations and we will see changes of the output of the stochastic function. This is not possible to do with a standard BN,where the direction of the edges does not have a causal interpretation.
For BN, $X \to Y$, defined over two binary variables X, Y, we can answer $P(Y|X = 1)$ but we cannot reason about the effect that an intervention,where we set a value,e.g,$X = 1$, will have on the distribution. To denote that we are changing the CPD $P(Y|X)$ by setting $X = 1$ we write to $P(Y|do(X = 1))$, where $do(X = 1)$ means that X takes the value 1 with probability 1. The action of setting $X = 1$ we call an intervention on what value X takes. One cannot distinguished two BN $X \to Y$ and $X \leftarrow Y$ based observing specific values of the parent set. By interpreting $X \to Y$ as a causal model and setting $X = 1$ we have that:

$$P(Y|do(X = 1)) = P(Y|X = 1), \tag{2.2}$$

while the output of $P(X|do(Y = 1))$ will be:

$$P(X|do(Y = 1)) = P(X) \tag{2.3}$$

This is because $P(Y|X)$ is a causal mechanism,assuming $X \to Y$, while the CPD table $P(X|Y)$ is not. While we can always change the parameters in the CPT for a Bayesian network, this change those not have a causally interpretable meaning,and at the same time it defines a new Bayesian network. In a causal model, when the change in the CPD table(causal mechanism) happens, this has the interpretable meaning of intervening on a value of a parental configuration at the same time as the causal model remaining unchanged. More generally,as stated in the following definition, given a causal Bayesian network,we can answer intervention queries that involve any of the variables involved in the model.

Definition 2.4.1. [KF09, Definition 21.1] A causal model $\mathcal{C}$ over $\mathcal{X}$ is a Bayesian network over $\mathcal{X}$,which, in addition to answering probability queries,can also answer queries $P(\mathbf{Y}|do(z), x)$,as follows:

$$P_{\mathcal{C}}P(\mathbf{Y}|do(z), x) = P_{\mathcal{C}_{\mathbf{Z}=\mathbf{z}}}P(\mathbf{Y}|x)$$

CHAPTER 3

Structure learning

Constraint-bases vs score based

Structure learning is the task of learning the DAG structure of a BN based on some data that is assumed to have been generated by the model. Constraint based learning and score based learning are the main categories of methods used to learn Bayesian network structures. Constrained based methods utilize a sequence of independency test to learn the structure. Score based learning methods uses a scoring function together with a search algorithm to traverse the graph space in order to find a high-scoring DAG.The space of DAGs grows exponentially with the number of nodes. Therefore, one typically has to resort to heuristic methods that are not guaranteed to find the global optimum.

We will focus on score based learning methods. The reason for our choice is earlier experience in that constrained based methods tend to be less robust then score based methods. Constrained based learning is more sensitive to error in capture individual independency tests. Score based methods are more robust in the sense that they capture the overall graph structure. In this work we will consider two score based methods, one for standard CPD tables and one that also learn CSI-trees for each considered network.

3.1 CPT based score

We define $\Theta_G = \{\theta_{X_1|Pa_{X_1}}, ..., \theta_{X_n|Pa_{X_n}}\}$ to be all CPD tables P_B which factorizes over the BN structure G.That is,

$$\theta_{X_i|Pa_{X_i}} = (\theta_{X_i|u_i}) : u_i \in Val(Pa_{X_i}),$$

the CPD's of X_i,where $\theta_{X_i|u_i}$ specifies the conditional distribution of X_i given that the parents have taken on the configuration u_i. Our score will be the log-joint distribution $\log P(D, G)$.

$$\log P(D, G) = \log P(D|G) + \log P(G) \tag{3.1}$$

This is also called the Bayesian score and it consists of the log marginal likelihood given G ($\log P(D|G)$) and a graph prior ($\log P(G)$). For CPT scores we will use

a uniform prior $P(G) \propto 1$. Our goal is to approximate the posterior distribution $P(G|D)$.

$$P(G|D) = \frac{P(D|G)P(G)}{P(D)}) \propto P(D|G)P(G) \tag{3.2}$$

Where $P(D)$ is a constant for all G. The key component of the Bayesian score is marginal likelihood

$$P(D|G) = \int_{\Theta_G} P(D|\theta_G, G)P(\theta_G|G)d\theta_G \tag{3.3}$$

which is the expected likelihood $P(D|\theta_G, G)$, under some prior on the model parameters $P(\theta_G|G).P(D|\theta_G, G)$ is the likelihood given a Bayesian network structure G. Under certain assumptions, the marginal likelihood for a BN structure can be computed in closed form.
The likelihood can be written as:

$$L(\theta : D) = \prod_{m=1}^{M} P(x_1[m], ..., x_n[m] : \theta), \tag{3.4}$$

where $(x_1[m], ..., x_n[m])$ is the m'th joint instance of random variable $X_1, ..., X_n$ from a dataset generated with M instances. The product comes from the assumption that each instance was generated independently. We can express the likelihood under a BN structure as:

$$\begin{aligned} L(\theta : D) &= \prod_{m=1}^{M} P_G(x_1[m], ..., x_n[m] : \theta) \\ &= \prod_m \prod_i P(x_i[m]|pa_{X_i}[m] : \theta) \\ &= \prod_i \prod_m P(x_i[m]|pa_{X_i}[m] : \theta) \end{aligned} \tag{3.5}$$

We can then express the likelihood as a product of node-wise likelihoods:

$$L(\theta : D) = \prod_i L_i(\theta_{X_i|Pa_{X_i}} : D) \tag{3.6}$$

This decomposition is called the global decomposition property of the likelihood. This holds because of the assumption that the parameters $\theta_{X_i|Pa_{X_i}}$ are assumed to be disjoint from $\theta_{X_j|Pa_{X_i}}$ for all $j \neq i$. The global decomposition property ensures that the factorization of the likelihood can be done with respect to CPD-tables of the BN structure.
We can further decompose the likelihood by using the local decomposition property:

$$L(\theta : D) = \prod_i \prod_{u_i \in Val(Pa_{X_i})} L_i(\theta_{X_i|u_i} : D), \tag{3.7}$$

which makes it possible to look at each parameters of each row within a CPD-table separately.

Now the local likelihoods in Equation (3.6) will be in the form of Equation (3.8) when we use categorical likelihood:

$$L(\theta|D) = \prod_{k}^{K} \theta_k^{M[k]}, k = 1, ..., K \tag{3.8}$$

where $k = 1, ...K$ represents the different categories, θ_k denotes the probability of observing category k, and $M[k]$ denotes the number of times category k is observed in the data.
By putting a Dirichlet prior on the model parameters:

$$(\theta_1, ..., \theta_K) \sim Dirichlet(\alpha_1, ..., \alpha_K) \propto \prod_{k}^{K} \theta_k^{\alpha_k - 1} \tag{3.9}$$

$$\alpha = \sum_{k}^{K} \alpha_k,$$

we can compute the posterior distribution, and consequently,marginal likelihood in closed form.

Proposition 3.1.1. [KF09, Proposition 17.3]
If $P(\theta)$ is $Dirichlet(\alpha_1, ...\alpha_K)$ then $P(\theta|D)$ is $Dirichlet(\alpha_1 + M[k], ..., \alpha_K + M[K])$ where $M[k]$ is the number of occurrences of x^k

The Dirichlet prior is said to be a conjugate prior to the categorical likelihood,meaning that the posterior is in the same family of distributions as the prior. The general expression for the marginal likelihood with a categorical likelihood and Dirichlet prior is given as:

$$P(D|G) = \frac{\Gamma(\alpha)}{\Gamma(\alpha + M)} \prod_{k} \frac{\Gamma(\alpha_k + M[k])}{\Gamma(\alpha_k)} \tag{3.10}$$

We want the decomposition properties to hold for the marginal likelihood as well. The same type of decomposition holds if the prior satisfies global and local parameter independence in addition to the likelihood having global and local decomposition property.

Proposition 3.1.2. [KF09, Proposition 18.2]
Let G be a network structure,and let $P(\theta_G|G)$ be a parameter prior satisfying global parameter independence.
Then,

$$P(D|G) = \prod_{i} \int_{\Theta_{X_i|Pa_{X_i}}} \prod_{m} P(x_i[m]|pa_{X_i}[m], \theta_{X_i|Pa_{X_i}}, G) P(\theta_{X_i|Pa_{X_i}}|G) d\theta_{X_i|Pa_{X_i}}$$

Moreover if the prior $P(\theta_G|G)$ also satisfies local parameter independence

$$P(D|G) = \prod_{i} \prod_{u_i \in Val(Pa_{X_i}^G)} \int_{\Theta_{X_i|u_i}} \prod_{m, u_i[m]=u_i} P(x_i[m]|u_i, \theta_{X_i|u_i}, G) P(\theta_{X_i|u_i}|G) d\theta_{X_i|u_i}$$

Under the assumptions of likelihood decomposition and parameter independence, we can compute the marginal likelihood in closed form:

$$P(D|G) = \prod_i \prod_{Pa_{X_i}} \frac{\Gamma(\alpha_{X_i|pa_{X_i}})}{\Gamma(\alpha_{X_i|pa_{X_i}} + M[pa_{X_i}])} \prod_{x_i} \frac{\Gamma(\alpha_{x_i|pa_{x_i}} + M[x_i, pa_{x_i}])}{\Gamma(\alpha_{X_i|pa_{x_i}})}, \tag{3.11}$$

where $\alpha_{X_i|pa_{X_i}} = \sum_{x_i} \alpha_{x_i|pa_{X_i}}$ and Pa_{X_i} are the parent joint events of X_i. Note that the marginal likelihood is a product of node-wise factors, or a sum when log is used. Assuming a similar factorization of the graph prior, the score is thus a product(or sum) of node-wise scores.This property is particularly important when traversing the space of DAGs in the search phase since local modifications to a DAG will typically only change a few node-wise scores and the remaining can be reused from the previous iteration.
In addition to this decomposability, we want all DAGs in the same I-equivalence class have equal scores.

Definition 3.1.3. [KF09, Definition 18.4]
Let $score(G : D)$ be some scoring rule.We say that it satisfies score equivalence if for all I-equivalent networks G and G' we have $score(G : D) = score(G' : D)$ for all data set D.

This equality is important for us because the best output of our methods is the I-equivalence class of the BN structure for the true underlying causal model. For our prior, score equality within I-equivalence classes holds when one uses a Dirichlet parameter prior from the BDE family. In this work we are using such a prior. More specifically, we are using the BDEu prior for which the hyper-parameters are set according to:
We are using BDEU prior.

$$\alpha_{x_i|Pa_{X_i}} = \frac{N}{|Val(X_i)||Val(Pa_{X_i})|} \tag{3.12}$$

where $Val()$ is the outcome space of input variables and $|Val()|$ is simply the number of outcomes.

Algorithm 1. CPD-algorithm

Input: $data$,X_i,Pa_{X_i},$\{Val(X_i)\}_{i=1}^n$
assign $Val(Pa_{X_i})$
assign $\alpha_{x_i|par_{x_i}} = \frac{N}{|Val(X_i)|*|Val(X_{Pa_{X_i}})|}$
$Score_{val} = 0$
for each $pa_{x_i} \in Val(Pa_{X_i})$
count occurrence of each configuration i.e all $M[x_i, pa_{x_i}]$
assign $M[pa_{x_i}] = \sum_{x_i} M[x_i, pa_{x_i}]$
$Score_{val} = Score_{val} + \log \frac{\Gamma(\alpha_{X_i|pa_{X_i}})}{\Gamma(\alpha_{X_i|pa_{X_i}} + M[pa_{X_i}])} \prod_{x_i} \frac{\Gamma(\alpha_{x_i|pa_{x_i}} + M[x_i, pa_{x_i}])}{\Gamma(\alpha_{X_i|pa_{x_i}})}$

An overview of the algorithm for computing the family scores is given in Algorithm 1.

3.2 CSI score

A CSI-tree is a structure imposing a special kind of local parameter sharing. To understand local parameter sharing, it might be helpful to first understand global parameter sharing. Instead of structuring the parameters in Θ_G in terms of CPDs in a CPD table denoted by $P(X_i|Pa_{X_i}, \Theta_G)$ for nodes $X_i \in \mathcal{X}$ given its parent-set Pa_{X_i}, we will instead partition the set of all parameters in Θ_G into subsets $\theta^1, ..., \theta^k$ where each θ^k include parameters that are shared across the CPD tables [KF09, page 755]. For each of these subsets their is an accommodating set of variables L^k such that $L^1, ..., L^k$ form a partition of $\mathcal{X}$,that is,$L^1, ..., L^k$ are disjoint and there union is equal to $\mathcal{X}$. This ensures that one can then associate a disjoint set of parameters to a disjoint set of nodes, where one can find all parameters related to any node in L^k in the set of parameters θ^k. We have the following implication:

$$P(X_i|U_i, \theta) = P(X_i|U_i, \theta^k) \tag{3.13}$$

U_i are the parents of X_i, we also have that any pair of nodes X and Y in L^k. Further, we will assume that the CPDs for all $X, Y \in L^k$ are identical:

$$P(X|U_X, \theta^k) = P(Y|U_Y, \theta^k) \tag{3.14}$$

Note that above equality can only hold for X, Y where $Val(X) = Val(Y)$ at the same time as $Val(U_X) = Val(U_Y)$.
For notational convenience ,for any $X_i \in L^k$,let s_k denote the values of X_i and f_k denote the configurations of the parents Pa_{X_i}which we here denote by U_i. Based on this, one can now decompose the probability distribution to factorize over a specific network as:

$$P(X_1, .., X_n|\theta) = \prod_i^n P(X_i|Pa_{X_i}, \theta) \tag{3.15}$$

$$= \prod_{k=1}^{K} \prod_{X_i \in L^k} P(X_i|U_i, \theta) \tag{3.16}$$

$$= \prod_{k=1}^{K} \prod_{X_i \in L^k} P(X_i|U_i, \theta^k) \tag{3.17}$$

The first equality comes from chain rule of Bayesian network, the second equality comes from the disjoint sets L^k and third equality comes from independency of the set of parameters for variables in L^k to any other parameter set.
We assume now that every conditional distribution in the CPD-table follows a multinomial distribution. We use $\theta^k_{s_k|f_k}$ to denote the specific conditional probability $P(X_i = s_k|U_i = f_k, \theta^k)$ for some $X_i \in L^k$. We can then express the likelihood under global parameter sharing as:

$$L(\theta : D) = \prod_{k=1}^{K} \prod_{s^k, f^k} \prod_{X_i \in L^k} \theta^k_{s_k|f_k} \tag{3.18}$$

$$= \prod_{k=1} \prod_{X_i \in L^k} (\theta^k_{s_k|f_k})^{M_k[s_k, f_k]} \tag{3.19}$$

where

$$M_k[s_k, f_k] = \sum_{X_i \in L^k} I(x_i = s_k, u_i = f_k) \tag{3.20}$$

Thus,in the above expression we add up the counts of multiple variables in the network based on the parameter equality enforcement, due to global parameter sharing.

The parameters can not only be shared globally, but also locally in a single CPD table. One way of incorporating local parameter sharing is through the use of CSI-trees. The branches encodes a specific configuration of the conditioning set, where one is sharing parameters across different conditional distributions within a CPD table. We will again focus on the case where we assume that the CPDs of graph G defines a set of multinomial distributions. For each variable in G, together with a parent joint event $u_i \in Val(U_i)$, we have a multinomial distribution. We define the set $D = \cup_i^n \{P(X_i|u_i) : u_i \in Val(U_i)\}$ which contains all multinomial distribution in all CPDs of graph G.
We define a set of locally shared parameters $\theta^1, ..., \theta^k$ where each θ^k is associated with a set $D^k \subseteq D$. Similar to before, we assume that $D^1, ..., D^k$ form a partition of D and that all conditional distributions within D^k share the same parameters θ^k. Note that for this type of constraint we must have that all distributions in the same class,D^k, must have the same set of values.
As an example, for CSI-tree in figure (2.7) we have the sets D^1,D^2 and D^3.

$$D^1 = P(Z|x^0, y^0) \tag{3.21}$$
$$D^2 = P(Z|x^0, y^1) \tag{3.22}$$
$$D^3 = P(Z|x^1, y^1), P(Z|x^1, y^0) \tag{3.23}$$

where the CPDs in D^3 would be represented by the same parameters.Local parameter sharing can be seen in an analogous way as global parameter sharing by decomposing the likelihood.Starting with:

$$P(D|\theta) = \prod_i \prod_{u_i} \prod_{x_i} P(x_i|u_i, \theta)^{M[x_i, u_i]}, \tag{3.24}$$

when local parameter sharing is introduced, one can aggregate inner term according to the specific local parameter sharing set D^k thereby getting:

$$P(D|\theta) = \prod_i \prod_{u_i} \prod_{x_i} P(x_i|u_i,\theta)^{M[x_i,u_i]} \tag{3.25}$$

$$= \prod_{k=1}^{K} \prod_{\langle X_i,u_i\rangle \in D^k} \prod_j (\theta_j^k)^{M[x_i^j,u_i]} \tag{3.26}$$

$$= \prod_{k=1}^{K} \prod_j (\theta_j^k)^{\sum_{\langle X_i,u_i\rangle \in D^k} M[x_i^j,u_i]} \tag{3.27}$$

$\langle X_i, u_i\rangle$ refers to a conditional distribution.

For CSI-tree the set D^k is restricted to only include local branches where the set of parent configurations are generated from context involving a subset of the parental variables.

With the introduction of local sharing in the form of introducing a CSI-trees as, middle-step when computing the marginal-likelihood, the full CSI-tree that represents all full configurations of the parents as individual branches typically gets reduced such that shorter branches represent a set of CPDs. More specifically D^k is defined through branch k,and the parent configurations in D^k are identical for the variables specifying specific branch k.
Under local(and global) parameter sharing, we can still compute the marginal-likelihood in closed form using a similar approach and assumptions as in the standard case. The only difference is that instead of modelling each parent configuration separately, we must now have classes of parent configurations that result in the same CPD. Going back to our example D^1, D^2, D^3 represents the branches in the CSI-tree shown in Figure(2.7), the marginal likelihood becomes:

$$\prod_{k=1}^{3} \frac{\Gamma(\alpha_{Z|k,J})}{\Gamma(\alpha_{Z|k,J} + M[k])} \prod_z \frac{\Gamma(\sum_{j\in J}[M[z,k,j] + \alpha_{z|k,j}])}{\Gamma(\sum_{j\in J}\alpha_{z|k,j})}, \tag{3.28}$$

where J is the set of configurations of parent variables that are not used to specify the branch,that is, the variables Z is context-specifically independent of given the context if the branch.
For this example we have three D^k For D^1 , $J = \emptyset$.For D^2, $J = \emptyset$.For D^3, $J = \{y^0, y^1\}$. resulting in the counts:

$$M[1] = \sum_{z\in\{0,1\}} M[z, x = 0, y = 0] \tag{3.29}$$

$$M[2] = \sum_{z\in\{0,1\}} M[z, x = 0, y = 1] \tag{3.30}$$

$$M[3] = \sum_{z\in\{0,1\}} \sum_{y\in\{0,1\}} M[z, x = 1, y] \tag{3.31}$$

$$= \sum_{z\in\{0,1\}} M[z, x = 1] \tag{3.32}$$

$$\alpha_{z|k,j} = \frac{N}{|Val(Z)||Val(X,Y)|}. \tag{3.33}$$

$$\alpha_{Z|k,J} = \sum_{j \in J} \sum_{z \in Z} \alpha_{z|k,j} \tag{3.34}$$

Instead of now talking about DAGs in the same I-equivalence class having equal score, one can now talk about the score function returning the same score for DAGs in the same CSI-equivalence class. For a formal definition of CSI-equivalence we will refer to [Pen+13].

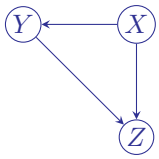


Figure 3.1: BN structure H

To illustrate the difference between I-equivalence and CSI-equivalence, we will first list all DAGs in the I-equivalence class for BN Network H in Figure 3.1, then we will put a label on one of the edges,representing a CSI, and see how the parameter constraint creates a CSI-equivalence class. All variable in H are binary.

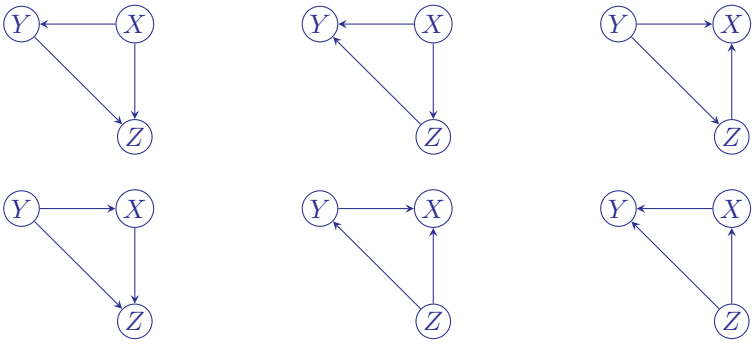
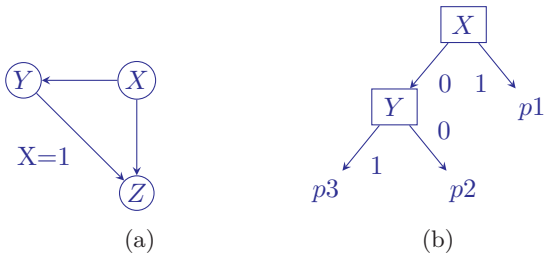


Figure 3.2: I-equivalence-class for BN H

Starting with the standard case the six I-equivalent DAGs are shown in Figure 3.2. Next, in Figure 3.3(a) we set $X = 1$ as a label on the edge going from Y to Z indicating that this edge is removed if $X = 1$, thus representing a CSI of the form $(Z \perp Y|X = 1)$. The corresponding CSI-tree in Figure 3.3(b) shows how the same CSI is captured by through the CPD structure of Z.



(a) (b)

Figure 3.3: BN structure with label $X = 1$

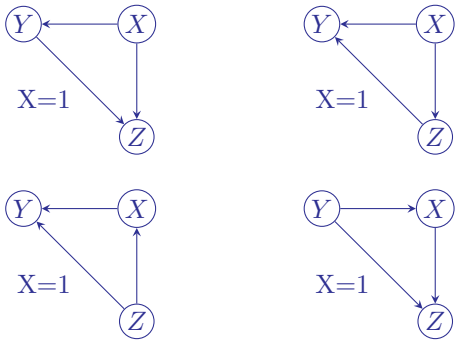


Figure 3.4: The CSI-equivalence class for the labeled DAG in 3.3(a) and csi-tree when the edge between Y and Z has label $X = 1$

When the edge between Y and Z is labelled according some value of X, one has to look at BN structures where X influences the probability relation between Y and Z. This means that X has to be a parent of at least Z or Y. The CSI-tree for Z and Y in BN structures where X is a descendant of both Y and Z does not include X. These structures can not capture the conditional edge existence between Y and Z based on a value of X.
From Figure 3.2 and Figure 3.4 we see that the DAGs in the CSI-equivalence class is a subset of these in the I-equivalence class. Furthermore,the orientation of the existing edges is half the times in either direction in the I-equivalence class.In the CSI-equivalence class we have the same thing for the edges between Y and Z, however,there is slightly higher support for $X \to Y$ and $X \to Z$ than $Y \to X$ and $Z \to X$, respectively.This way, a CSI can provide some additional information regarding orientation of some of the edges.

Graph Prior

When computing the scores for the CSI-method,we will use the sparsity-promoting graph prior from reference [Pen+15]:

$$log(P(G)) = -\sum_{j=1}^{d} (1+t)^{|pa(j)|} \cdot \log n, \tag{3.35}$$

where $t \geq 0$ is a tuning parameter of how much edges in G will be penalized. The factor $\log n$ is used to adapt the effect of penalization to the number of data samples.

The reason is that the marginal-likelihood alone has a tendency to overfit the data, resulting in overly complex models that include many none-true edges,that is,false positives, which again leads to BN structures learned by a CSI-score is of type of LDAG. [Pen+13] gives a detailed explanation of the overfitting problem. Rather then using prior of [Pen+13], we will instead go with the approach taken in [Pen+15],where the prior does not penalize the CSI-trees directly,but rather penalize the density of the global DAG structure.

CSI-Algorithm

The space of CSI-tree quickly becomes large as the number of parents is increased. For this reason the algorithm implemented for the construction of CSI-trees will be based on greedy hill climb, which tries to maximize the log marginal likelihood.The family score is then given by the log-marginal likelihood of the identified CSI-tree with the graph prior added. To further represent the search space we set an upper limit to the number of parents,K,such that,$\sum_{k=0}^{K} \binom{n-1}{k}$ possible parent sets are considered for each node. With a greedy hill climb on the space of CSI-tree one want to find the important joint configurations of the conditioning set that defines the unique CPDs within a CPD-table.
Our goal is to find the I-equivalence class of the underlying true BN structure. We are trying to find this based on a data set of limited sample size that will contain a considerable amount of noise. Therefore, we do not want our method to overfit on the data. One reason that CSI-scores might do better than CPT-scores is that it does not necessarily consider full parental configurations, but instead it tries to capture the most relevant CPDs through partially specified parental configurations. This counteract the exponential blowup og the parental outcome space, which is the main reason why it might be beneficial in terms of identifying the true global graph structure. Our algorithm works by reducing the parent set configurations of each node through choosing the nodes that noticeably outperform the others in every step of the CSI-tree building process. This is done by comparing the score of the current CSI-tree to a CSI-tree where all nodes that have not yet been added to a branch are added as a test of whether they increase the score or not.This procedure is done to all branches of the current CSI-tree state.The node resulting in the biggest improvement is selected.

The construction starts with the root. The root is selected based on computing all scores of parent set size 1, where the score is the sum of the log-marginal likelihood and the prior. After the root node has been added, the branch of the CSI-tree are iteratively extended,or grown, by splitting on parents that are not yet included in the considered branch. Each branch has its own score which is a sum of the log-marginal-likelihoods one gets by iterating the values for the node the tree is being built for, when the configuration of the parent set is held fixed. For the root selection, the scores one gets by iterating over the parent configurations is compared with the score that is calculated when no parent is included,.i.e instead of $M[x_i, pa_{x_i}]$ in expression (3.11) one uses the count

$M[x_i]$,and instead of $M[Pa_{X_i}]$, M is used.We compare these score by taking the score difference between a sum and the score for when no parent is included. This is done for all joint configuration and for all potentially addable parents. The greatest one that is greater then 0 gets added as root. Each component of its sum is the score for a branch. Once the first split has happened, the only configurations considered in the next iteration are those that include both the root and all the potentially addable parents that do not exist in a branch, for all branches.Again, the scores in the next iteration is compared with the previous values of the branches. The procedure of growing the branches of the tree is continued until no improvement is possible or the three is of full depth, meaning that each branch include all the parents.
A disadvantage with such an approach is that the algorithm only looks one step ahead comparing the current score against the new score obtained after a single split, even if the split that gives the biggest improvement now might lead to a suboptimal tree further ahead. However our goal is not to necessarily find the optimal CSI-tree. We mainly want to test whether the CSI-tree-score learned from a greedy hill climb result in an improvement over the standard CPT-score.

Algorithm 2. CSI-tree-algorithm

Input:$data$,X_i,$X_{Pa_{X_i}}$,$\{Val(X_i)\}_{i=1}^{n}$
assign $Val(X_i)$
while TRUE
  assign $init = 0$
  for each X_j in $X_{Pa_{X_i}}$ **do**
    for each branch k **do**
      if X_j notin branch k
        assign values of X_j, $Val(X_j)$
        count occurrence of each configuration ,i.e all $M[x_i, k, x_j]$
        assign $\alpha_{x_i|x_j,k} = \frac{N}{|Val(X_i)|\cdot|Val(k,X_j)|}$
        assign $\alpha_{X_i|k,x_j} = \sum_{x_i} \alpha_{x_i|x_j,k}$
        for each $x_j \in Val(X_j)$ **do**
          assign $M[k, x_j] = \sum_{x_i} M[x_i, k, x_j]$
          $Score(k'_{x_j}) = \log \frac{\Gamma(\alpha_{X_i|k,x_j})}{\Gamma(\alpha_{X_i|k,x_j}+M[k,x_j])} \prod_{x_i} \frac{\Gamma(\alpha_{x_i|k,x_j}+M[x_i,k,x_j])}{\Gamma(\alpha_{x_i|k,x_j})}$
        end for
        $diffscore = \sum_{x_j} Score(k'_{x_j}) - Score(k)$
        if $diffscore > init$
          $init = diffscore$
          $X_{choise} = X_j$
          $k_{choice} = k$
        end if
      end if
    end for
  end for
  if $init$ still is 0
    BREAK while
  end if
  else
    add X_{choice} to branch k_{choice} in tree
  end else
end while

Algorithm.2 shows the pseudo-code of the function used to calculate CSI-log-marginal likelihood.

3.3 MCMC over structures

If the goal is prediction and the sample size is large enough choosing one of the models that have a high score could give acceptable accuracy. Our goal is structure discovery and selecting out one high scoring DAG is less useful. It then make sense to consider averaging over multiple high scoring models to approximate the underlying BN structure. One way of doing this is using Markov Chain Monte Carlo (MCMC),more specifically Metropolis Hastings over structures to get samples from posterior distribution before averaging over these samples. Running the MCMC until convergence to the stationary distribution $P(G|D)$ and estimating the true DAG through averaging over the samples

DAGs, we get an approximation of the posterior probability of the existence of a causal path from X to Y.This procedure is called Bayesian model averaging.

We define a Markov chain over the space of DAGs,restricting the max parent set size to some upper limit. This Markov chain converges to the posterior distribution $P(G|D)$ if its designed carefully to be equal to the stationary distribution for the chain. This is ensured by the chain being irreducible, aperiodic, positively recurrent and the choice of the proposal distribution has to satisfy the detailed balance. Condition,which is satisfied when the probability of going from one DAG x to another DAG x' is equal to going from x' to x.The characteristic of the chain is ensured by the construction of a satisfactory proposal distribution, proposal distribution should not have the possibility of jumping to far from the current accepted value because such a proposal will fail to locate the correct neighbourhood depending on how big the jumps are.

In this article the proposal distribution $T()$ is defined based on the neighbourhood of the current BN structure which is constructed by adding,deleting or reversing a single edge in the considered DAG, ensuring that acyclicity is maintained.The proposal distribution for a given DAG is then defined as the uniform distribution over the neighbourhood:

$$T(G|G') = \begin{cases} \frac{1}{|nbhd(G')|} & if\ G \in nbhd(G') \\ 0 & else \end{cases} \tag{3.36}$$

The acceptance probability then becomes:

$$p = \min(1, \frac{T(G|G') \cdot P(G'|D)}{T(G'|G) \cdot P(G|D)}) \tag{3.37}$$

$$= \min(1, \frac{T(G|G') \cdot \frac{P(D|G')P(G')}{P(D)}}{T(G'|G) \cdot \frac{P(D|G)P(G)}{P(D)}}) \tag{3.38}$$

$$= \min(1, \frac{T(G|G') \cdot P(D|G')P(G')}{T(G'|G) \cdot P(D|G)P(G)}) \tag{3.39}$$

For the CPD-scores the graph prior is uniform over all graphs.The acceptance probability gets reduced to:

$$p = \min(1, \frac{T(G|G') \cdot P(D|G'))}{T(G'|G) \cdot P(D|G)}) \tag{3.40}$$

If the proposed state is accepted the current state is set to G', otherwise it is set to G.

For CPD-scores, when convergence is reached we can hope to sample from the I-equivalence class of the underlying BN structure that the data has been generated from. The DAGs in the class will have the same scores, and the chain would ideally circle around the neighbourhood of the equivalence class.

The deletes in the neighbourhoods ensures that there always is a probability for the proposed DAG being simpler than the current DAG. The reverse operator is there to make it easy to move back when the trajectory diverges from good paths towards convergence. Since our BN structure scores are decomposable this allows for only adding or subtracting one score from the currently accepted DAG score to attain the score of the proposed DAG, when the difference between the two DAGs is an add or delete. Reverse move has a delete and add but the same logic requires the update of two scores. We will start the chain with a "burn-in" period which is not only important for the trajectory to hit the target neighbourhood but also because every sample is correlated with the starting value. This correlation will never entirely disappear, but a long enough burn-in is necessary for this correlation to be small enough so that it does not influence the sample values to much. Once the samples are saved, one can compute approximate posterior probabilities of various DAG features by averaging over the sampled DAGs. In this case, we would like to find if there exist a directed path between a variable and another. By first transforming each DAG in the sample from G to $(I - G)^{-1}$ which the geometric series $\sum_{n=1}^{\infty} G^n$ converges to, one can find how many ways one could go from node j to node i, where i is row in an adjacency matrix representing a DAG and j is column. Here the interest is the existence of any such path. Therefore the next transformation needed is to convert all positive matrix elements of $(I - G)^{-1}$ to 1. Now by summing element-wise all the matrices and dividing by the number of samples, this information is attained. Each element in this matrix is approximated posterior probability of there existing a directed path from node(row) j to node (column) i.
The matrix we where referring to is known as the adjacency matrix. An Adjacency matrix is a representation used both for directed and undirected graphs. It is a one-zero matrix where each element represents the existence or non-existence of an edge between two nodes. Depending on preference, either one can interpret 1 in position (i, j) as the existence of an edge from i to j or an edge from j to i. Here one choose the first,since the exact algorithm being used to compare with uses this notation. A zero in place (i, j) of course means the none-existence of an edge from i to j.

Order MCMC has been shown to be superior when it comes to convergence and mixing compared to structure MCMC [GH08], the disadvantage being each DAG has multiple orders. The MCMC chain could therefore deviate substantially depending on the the chosen order in each iteration. The reason for this is using order as DAG representation fails to determine the prior of a DAG. Since one is sampling orders the prior is specified over orders. This is not a problem when one has a lot of available data. We will focus on testing our methods on relative small sample size and focus on implementing MCMC over structures. Empirically, structure MCMC seem to be slow in mixing. Since the moves in the space of DAGs is small,the sampler tends to get trapped in local maxima more easily. There are ways to mitigate such issues and one way is proposed in [GH08], where a new reversal move is introduced. The reason is that the conventional reversal move does not take into consideration if the reverse is useful or not in combination with the current parent set. For this reason, Grzegorczyk and Husmeier proposed a new move called the REV move.

However this will not be implemented here.The reason for bringing up this article is because it highlights some of the problems when implementing MCMC both over orders and over structures.

Score search

The MCMC implemented for this work takes in a scorefile where all scores are computed up to a max parent size. Each row in the scorefile includes the score followed by the size of the parent-set and the specific parentset. The scores are ordered and one can therefore find each score algorithmically.

This is the general pattern of score-file is illustrated in Figure 3.5.

```
20
1 1160
-301.566240 0
-301.671816 1 2
-306.835110 1 3
-307.910868 1 4
-315.266436 1 5
-304.565529 1 6
-303.612785 1 7
-307.730928 1 8
-307.658723 1 9
-304.792872 1 10
-314.790757 1 11
-301.798769 1 12
-304.627631 1 13
-306.891906 1 14
-304.493545 1 15
-301.110241 1 16
-303.878562 1 17
-306.715035 1 18
-307.364934 1 19
-302.212114 1 20
-313.147388 2 2 3
-314.959522 2 2 4
-328.981339 2 2 5
-307.976013 2 2 6
-307.363080 2 2 7
-314.481221 2 2 8
-314.703958 2 2 9
-308.491617 2 2 10
-329.907423 2 2 11
-319.686926 2 2 12
-308.371863 2 2 13
-313.611713 2 2 14
-308.026359 2 2 15
-310.772196 2 2 16
```

Figure 3.5: Example of scorefile

The pattern repeats for every node. The first row gives the node and number of values calculated for that node. After that, the scores are listed in the first column. The second column contains how many parents the score in a specific row is based on.The rest of every row contains the specific parent set.Every parent set is an ordered set such that 1 comes before 2 and so on. All parent sets for a specific node is ordered so that looking at any column in the columns containing parents they also have this order when the parent set size is fixed. One can find the index of each score by expressing the index as a sum of binomial coefficients, plus the node index, plus the index of the first value with a specific parent size. The nodes can be found by observing that in the first column only natural numbers excluding zero in the column are the nodes. The

next column represents the parent size. This includes only positive integers. Since all nodes have an equal number of parent set combinations, these indexes are the same no matter if we are searching for parent set of specific size for node 1, 5 or 14 and so on.

In addition, one needs a way to find the right index for a node within the set of all same size parent sets. Here one can utilize the fact that we can express the indexes as a sum of binomial coefficients. We use this method to find the correct parents up until the last one where we can map the last parent to an index between 1 to n i.e the number of nodes in the network. Since we exclude the last parent, our parent combination has become reduced by one. The total number of elements being chosen from has also been reduced by one because the node that we are searching for can not be included in its own parent set.

It will be helpful to define some variables like ,cardinality of parent-set subtracted by one Pa_X. Number of elements to choose from n subtracting one here as well. To find first parent we have to iterate j in expression:

$$\binom{n-j}{|Pa_X|}$$

j will give us the number of elements that have to be excluded in every step.Let say the first parent is 4 for node $X = 1$. The cardinality of the parent set is 5 so $|Pa_X| = 4$. To find the first element in the scorefile with this specification, we know that we have to jump over first parent being 2 or 3.

therefore we have to jump over :

$$\binom{n-1}{|Pa_X|} + \binom{n-2}{|Pa_X|}$$

when jumping over 2 we have to remember that since we are finding parent set for 1, 1 is not part of the parent set.When jumping over 3 we also have to remember that 1 and 2 can't be part of the any combination that starts with 3. When we have found 4, the number of elements to chose from n has been reduced by 2 for the next parent in the parent set and the size of the elements we pick out has been reduced by 1. Now $n = n - 2$ and $|Pa_X| = |Pa_X| - 1$. We continue like this until the next to last element in the parent set. For the last element we know the lower bound and the upper-bound of the elements that we have to search from. The upper bound is the number of nodes in the network. The lower bound is the next to last element in the parent-set plus 1. The sequence we get is the natural numbers between those bounds. By extracting the index of these elements in this interval,we have the last index. Finally we can sum all of these components up.This method seemed to be much more efficient then our function using general row search functions in R.

MCMC Algorithm

Algorithm 5. MCMC algorithm

$G_1 = inital\ BN\ Structure$
$Calculate\ initial\ P(G_1, D)$
$X_1 = P(G_1, D)$
$Calculate\ initial\ neighbourhood\ NB(G_1)$
$T(G'|G_1) = \frac{1}{|NB(G_1)|}$
for $t = 2\ to\ T$ **do**
  $Save\ samples\ G_t\ for$ **t** $after\ burn - in\ with\ thinning$
  $Choose\ one\ G'\ uniformly\ from\ NB(G_t)$
  $Calculate\ neighbourhood\ NB(G')$
  $T(G_t|G') = \frac{1}{|NB(G')|}$
    if: $X \rightarrow Y$ in G_t but not in G'
      Divide $P(X \rightarrow Y, D)$ to $P(G_t, D)$
      Multiply $P(\emptyset \rightarrow Y, D)$ to $P(G_t, D)$
      Set result equal to $P(G', D)$
    if: $X \rightarrow Y$ in G' but not in G_t
      Multiply $P(X \rightarrow Y, D)$ to $P(G_t, D)$
      Divide $P(\emptyset \rightarrow Y, D)$ to $P(G_t, D)$
      Set result equal to $P(G', D)$
    if : $X \rightarrow Y$ in G_t and $X \leftarrow Y$ in G'
      Divide $P(X \rightarrow Y, D)$ to $P(G_t, D)$
      multiply $P(X \leftarrow Y, D)$ to $P(G_t, D)$
      Set result equal to $P(G', D)$
  $AssignP(G'|D)$
  $p = min(1, \frac{T(G|G') * P(G'|D)}{T(G'|G) * P(G|D)})$
  $g = sample\ uniform\ number\ from\ (0, 1)$
  if $g \leq p$ **then**
    $X_t = P(G', D)$
    $G_t = G'$
    $T(G'|G_t) = \frac{1}{|NB(G_t)|}$
  end if

    else:
  $X_t = X_{t-1}$
end for
return samples

Algorithm.5 shows the pseudo-code of the MCMC algorithm.

Algorithm 3. scoremapfunction

Input:VEC(node,cardinality of parent set,parent set)
assign set1 1 to first parent in parent set excluding node.
assign set2, first node in parent set to last parent in parent set
assign set3 , 1 to first parent in parent set.
assign nr of parents except last l= |parent set|-1.
assign nr:nodes=nr of nodes except the node the score belongs to.
assign step=0.
assign n=1.
assign count=1.
while n!=0
    if n==cardinality of parent set
        BREAK
    end if
    if n=1
        if node in set3
            assign useset = set1
        end if
        else
            assign useset = set3
        end else
    end if
    else
        assign useset = set2
    end else
    if n>=2
        reduce useset by excluding the node in set2 from 1 to count
    end if
    assign count:2 = 0
    for j in 1 to |useset|
        if n>=2
            count = count+1
        end if
        update count:2 = count:2+1
        if useset[j] == VEC[(3+n-1)]
            nr:nodes = nr:nodes-count:2
            BREAK
        end if
        step= step + $\binom{nr:nodes-j}{l}$
    end for
    n=n+1
    l=l-1
end while
assign vector of elements from next to last node pluss one in parentset to nr of nodes in DAG and find which index of last element.
if parentset only contains one element the While loop will not be used.Therefore assign last parent index to index where parent is.
return nodeindex+ parentset size index+step+last parent index

Algorithm.3 shows the pseudo-code of the function that maps the parentsets to the corresponding indexs in the scorefile

Algorithm 4. neighborhoodfunction

Input:adjacency-matrix,max parent size
assign index set H for add indexes, indexes where adjacency matrix is 0
assign index set K for reverse, indexes where adjacency matrix is 1
assign index set S for delete, indexes of where adjacency matrix is 1
remove indexes from H and K that would create parentset larger then max parent size
assign $D = \{|H|, |K|, |S|\}$
for k in 1 to 3
    for a in 1 to $D[k]$
        if $D[k] == |H|$
            add 1 to each element $H[a]$ and check if the resulting graph is a DAG using topological sort
        if $D[k] == |K|$
            reverse an element $K[a]$ check if the resulting graph is a DAG using topological sort.
        if $D[k] == |S|$
            delete edge setting 1 to 0 for $S[a]$.
    end for
end for
save all element-wise changes to adjacency matrix

Algorithm.4 shows the pseudo-code of the function for calculating the neighbourhood of a DAG.

The basic idea of topological sort is to find an order containing all nodes in the DAG. This order must abide by the principal of "ancestor before descendants". Any node that is a descendant of some other node must come after this node. If this is not possible, the graph is not a DAG,.i.e, it contains one or more cycles. The algorithm ensures this by always marking the ancestors before descendants. If a cycle exist then a node that is already marked will be visited and thus failing the order principal.

3.4 Exact algorithm

The simulation study in this work will not only, compare the MCMC approximation of the two methods, but it will also compare the MCMC algorithm to the exact algorithm presented in the paper [Pen+20]. More specifically we will use the part of the algorithm that computes exact posterior probabilities of all pairwise ancestor relations, which corresponds to the existence of a causal relationship. Our interest is not to quantify how strong these effects are,only the existence. This algorithm can be applied up to a network of 20 variables. We are using it as a ground truth. This algorithm estimates the best result of our approach. In the following, we will give a brief overview of how it works.

We define $V = \{1, ..., n\}$ as the nodes in a BN structure G.We will used Pa_i to denote the parents of node $i \in V$. The algorithm calculates the posterior probability of a DAG G_i given some data, under standard assumptions ,by the formula:

$$P(Pa_i|D) = P(D)^{-1} \sum_{G:Pa_i} \prod_{v \in V} w_v(Pa_v)$$

$w_v(Pa_v)$ stands for :

$$w_v(Pa_v) = P(D_v|D_{Pa_v}, Pa_v)q_v(Pa_v)$$

$w_v(Pa_v)$ is the weight of node v when Pa_v is parent set of node v. $q_v(Pa_v)$ is

the node-wise contribution of node v to the prior $P(G)$. Since $P(G)$ is modular, $P(G)$ can be decomposed into a product where each element in the product is $q_v(Pa_v)$. Each term of the product of $P(G_i|D)$ is a weight contribution of node v to the posterior.
We define:

$$W_i(S) := \sum_{G:Pa_i=S} \prod_{v \in V} w_v(Pa_v),$$

to be the un-normalized posteriors for two distinct nodes $i, j \in V$. The set S is defined as $S \subseteq V \setminus i$ for any fixed i.
By rearranging the sums in the expression,one can divide the problem of computing the whole expression into smaller expressions, which is easier to handle.One can re-express $W_i(S)$ by partitioning the the sum product of weights into three sum-product of weights, and at the end multiply all of them together. The algorithm takes the advantage of this by re-expressing $W_i(S)$ as:

$$W_i(S) = \sum_{S \subseteq U \subseteq V \setminus i} f(U) w_i(S) b_i(V \setminus i \setminus U)$$

where $f(U)$ is the total weights of all DAGs of the non descendants of node i "forward weights", $b_i(V \setminus i \setminus U)$ is the total weights of all combinations of the parent-sets $Pa_v \subseteq V \setminus v$ "backward weights", where v refers to all nodes that in addition to being a parent to some other node, also is the descendant of i and at the same time does not contribute in creating a cycle. By combining the set in Pa_v with a specific non-descendant set U, one can construct all possible DAG's, having U as the non-descendants of i. One can do this for all possible U. The decomposition is done by the intuitive fact that for every i the nodes can either be a non-descendants of i or descendant, and if they are a descendant, they might be a parent of some node within the set of descendants which the nodes are part of,and this again can create different paths from i to some descendant j. For every none-descendant set U, of varying size, one sums out these nodes as they do not influence the path between i to j directly. By summing them out one has detached that part of the DAG which does not contribute directly to the path from i to j. One can write this mathematically as:

$$f(U) := \sum_{G \in G(U)} \prod_{v \in U} w_v(Pa_v),$$

$$b_i(T) := \sum_{G \in G(U)} \prod_{v \in U} w_v(Pa_v),$$

where $G(U)$ being all DAG's generated by U.We define the set $T = V \setminus \{i\} \setminus U$ containing the descendants of node i under the conditions that:

- $Pa_v \subseteq V \setminus v$ for each $v \in T$,
- The directed graph $(T, \cup_{v \in T} uv : u \in Pa_v \cap T)$ is acyclic,
- Every Pa_v intersect $T \cup i$,.i.e, is a descendant of i.

The details will be skipped.The key idea is that $f(U)$ and $b_i(T)$ can be computed efficiently through recursive recurrence relations [TH12].

An analogous formula exist when one wants to calculate direct ancestral path between any node i and j in the DAG G.

$$P(i \rightsquigarrow j|D) = p(D)^{-1} \sum_{G: i \rightsquigarrow j} \prod_{v \in V} w_v(Pa_v)$$

Some of this bears resemblance to what we will do. We are also interested in $p(i \rightsquigarrow j|D)$. The weights is our un-logged family scores where each un-logged family score is multiplied together to attain the posterior of DAG's given data. However,our approach is different in the sense that we try to approximate the same posterior target using MCMC.

CHAPTER 4

Simulation Study

4.1 Simulation setup

In this simulation study we will use the BNs Survey,Asia,Sachs and Child from the BNlearn repository. Our first task will be to calculate CPT-scores and CSI-scores before adding the chosen prior for each score type. The number of parent sets increases rapidly with the number of nodes at the same time as the growth in parameters is due to an increase in the value that every variable take. Therefore, it is not feasible to calculate the scores for all parent sets of the nodes when number of nodes in the network when it is large enough and we will restrict the parent sets the max parent-size to 4 for the Sachs and 3 for the Child network. The Sachs network has 11 variables and its distribution consists 178 parameters and for an 11 variable network one can create 11264 unique parent set combinations, while the Child network has 20 variables, 10485760 unique parent sets combinations and the distribution for this network has 230 parameters. The rapid increase of the number of parent sets and parameters is especially a problem when building the CSI-tree. Secondly, we run MCMC and the exact algorithm based on the scores. We sample 20 data-sets of the same sample size for data samples sizes $ns = 200, 500, 1000$ for networks Asia,Sachs and Child. For Survey network the data sample size will range over $ns = 200, 500, 1000, 5000, 10000, 100000$. This is because both methods had problem in determining the underlying graph structure on sample sizes $ns = 200, 500, 1000$ and gave nearly identical results for the two different score types. The underlying graph structure for this network is the only member of its I-equivalence class.
For each MCMC estimation and Exact algorithm estimation of the ancestral paths an AUC will be calculated using the true direct ancestral paths as benchmark. AUC is the area under the ROC-curve. Each point in this curve arise from to coordinates (x=FPR,y=TPR).

$$TPR = \frac{TP}{TP + FN} \tag{4.1}$$

$$FPR = \frac{FP}{FP + TP}. \tag{4.2}$$

FPR is a ratio between the number of false positives compared to the number of positives that is predicted to be positive by the method i.e the sum of actual positives that the method correctly determined to be positive and the positives

that the method defined to be positive but actually is negative based on the threshold used to distinguish between positives and negatives. TPR is a ratio between, the number of true positives that was captured compared to how many that actually exist in reality based on the threshold used to distinguish between positives and negatives. If both x and y is equal it means that the ratios TPR and FPR are equal varying the threshold. One can denote this by drawing the line y=x. If this is the output of the method it is not good.[1]. In contrast,perfect accuracy would result in a ROC curve that goes through the points (0,0),(0,1),(1,1),which also would result in an area of 1.

The way we can generate the points of the ROC curve is by first flattening our estimate,which is in the form of a matrix, before sorting the elements in decreasing order. The true underlying ancestral path matrix is flattened and sorted in the same order. The flattened sorted estimate now provide the thresholds. For each posterior probability we count how many 0's exist above and including this probability in the true underlying ancestral path vector divide by total number of elements above an including the probability in the true vector. This is also called FPR. For the TPR we count how many 1's above the probability exist and divide by total number of $1's$ in the vector. The AUC is then calculated using a package in R called CARROT, using the flattened sorted matrices as input the AUC function. An example of a few ROC curves is shown in Figure 4.1.

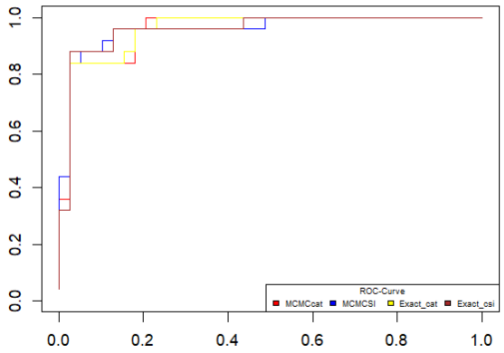


Figure 4.1: The following picture shows four ROC-curves on top of each other for one CSI-score and one CPT-score calculated for the Asia network when MCMC and the exact method is run on both scoretypes.The x-axis shows the FPR and the y-axis shows the TPR

[1]Our experiments are done on a ASUS TUF GAMING F15 computer

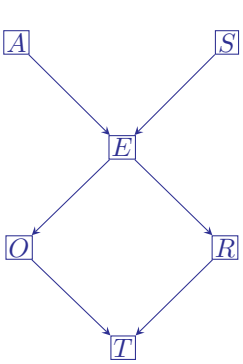
We use a uniform prior on the CPT scores because this prior has been shown to work well for these types of scores. We will use the prior mentioned in section 3.2 for the CSI-scores. The graph prior hyper-parameter will be varied between the values $t = 0, t = 0.5, t = 2$ for networks Asia,Survey and Sachs. One of the hyper-parameter values will be used when running the MCMC. For network Asia and Survey, $t = 0$ will be used to compare the MCMC AUC with the exact algorithm AUC, while for the Sachs network $t = 2$ will be used. The rest of the hyper-parameters will be tested on the networks with the exact algorithm. The exact algorithm is slow when applied on the scores for the Child network which contains 20 nodes. Therefore, we only run MCMC on this network together with hyper-parameter for the graph prior fixed to $t = 0.5$.[Pen+15] empirically found that $t = 0.5$ works well in the context of density estimation.We will run MCMC on Survey,Asia and Child networks with a burn-in of 150000. Every 10'th sample after that will be gathered until 200000 iterations. The Sachs network MCMC will be run with a burn-in of 250000.Every 10'th sample after that will be gathered until 300000 iterations. For larger networks it can sometimes take longer for the MCMC to converge. However due to computational cost of topological sort used in the computation of the neighbourhood of the proposed DAG for larger networks the MCMC,we fix the number of iterations with this in mind.
The cPDAGs will be shown for each network to illustrate the complexity of learning the correct edges in the networks illustrated in Figure 4.2,4.6,4.10 and 4.16.

4.2 Results

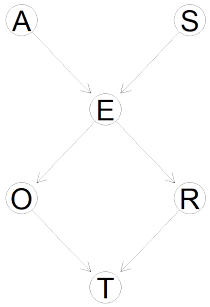
The following section will be dedicated for illustrating the results from the simulation study mostly in the form of box-plots. A small experimental error was made when running the MCMC for networks Asia and Survey. The CPT-score and CSI-score was tested on different data-sets. For these networks the $2X2$ grid plot in Figure 4.3,4.7, have to be looked at column-wise only. Each of the columns shows how close the MCMC results are compared to the exact algorithm. An additional box-plot is included using only the exact algorithm on both CPT-scores and CSI-scores on the same datasets in order to compare the result of the two scoring methods for these networks, shown in Figure 4.4 and 4.8. This is done for computational reasons.
Further Figure 4.5(Survey network) ,4.9(Asia network) shows plots using the exact algorithm with CSI-score tested on the hyper-parameters not used when comparing the CPT-score and CSI-score in Figure 4.4 for Survey network, and Figure 4.7 for the Asia network .

Figure 4.11 , 4.12 and 4.13 shows AUC results for the Sachs network. In Figure 4.11 the small error was corrected. The grid plot can be compared both column-wise and row-wise when comparing AUC for ancestral paths.Figure 4.12 show AUC for direct causal relations for the Sachs network, while Figure 4.13 shows the results of the AUC with CSI-score for the rest of the hyper-paramters not used in Figure 4.11. No Figure like Figure 4.12 is added for the Survey Network and Asia network because of the small error made. Figure 4.12 is still added for illustration. Figure 4.14 shows a convergence plot of the MCMC when compared to the exact algorithm for network Survey,Asia,Sachs network both

for CSI-score and CPT score.Figure 4.17 shows the AUC for ancestral paths of the Child network both when using CSI-scores and CPT-score with MCMC with the same datasets. Figure 4.18 shows the AUC for direct causal relations of the Child network both when using CSI-scores and CPT-score with MCMC.



(a) Survey Network



(b) Figure illustrates the CPDAG for Survey Network

Figure 4.2

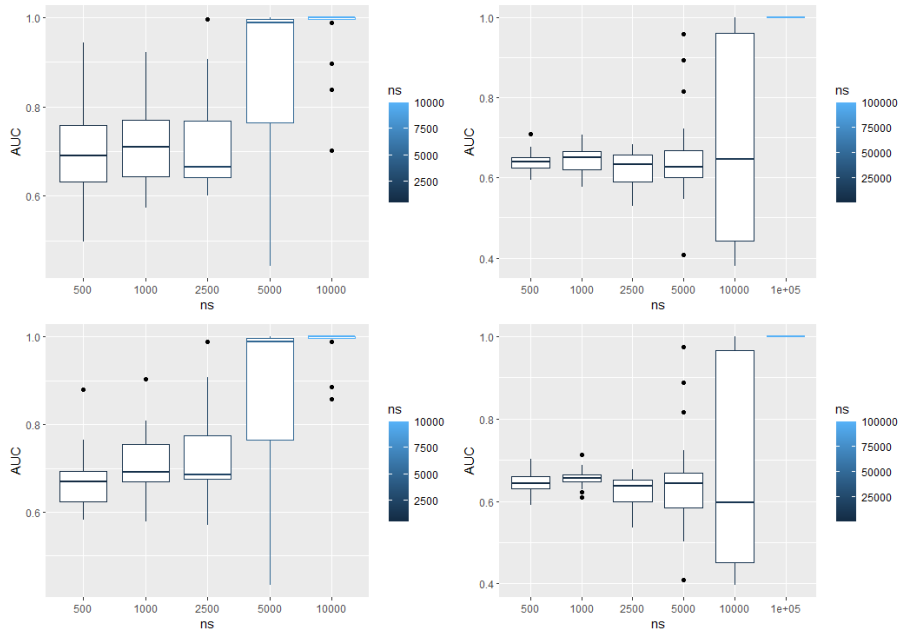


Figure 4.3: This plot is for the Survey network. The box-plot in upper left corner shows AUC ancestral path directions using MCMC with CSI-score.The box-plot on lower left corner shows AUC using the exact-algorithm with CSI-score.The sample sizes for these vary with ns=500,1000,2500,5000,10000.The box-plot in upper right corner shows AUC using MCMC with CPT-score.The box-plot in lower right corner shows AUC estimates using exact algorithm with CPT-score.The sample sizes for these vary with ns=500,1000,2500,5000,10000,100000.The CPT scores are calculated on different data-sets then the CSI-score.The comparison is done column-wise.

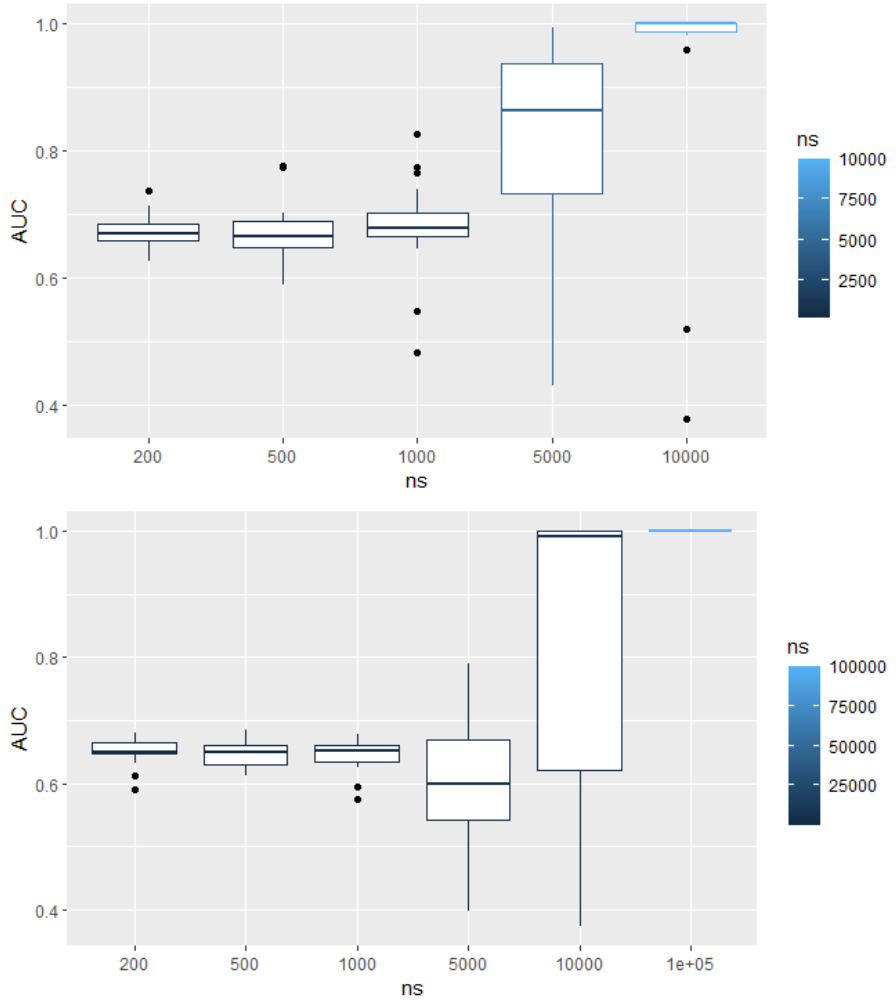


Figure 4.4: This plot is for the Survey network.Box plots showing AUC for prediction of ancestral paths using CSI-score and CPT-scores on the same dataset using the exact algorithm.This is done using the exact-algorithm.Upper box-plot shows the result for the CSI-score.The CSI-scores are calculated with the hyper-parameter of the prior set to t=0

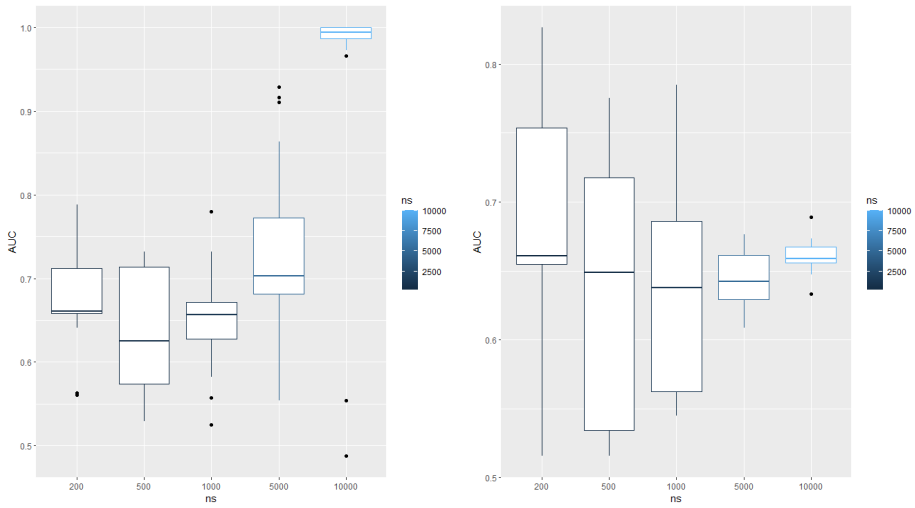
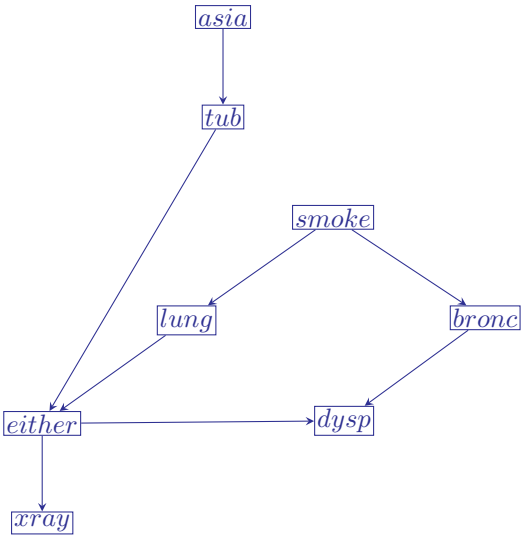
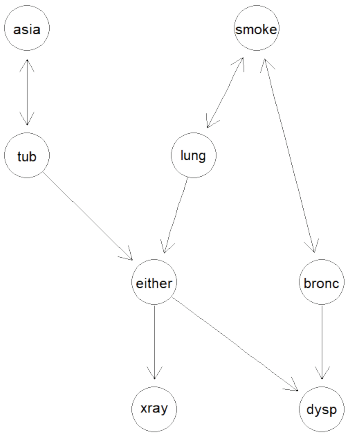


Figure 4.5: This plot is for the Surevy network.The plot on the right shows AUC for prediction of ancestral paths for the Survey network using prior hyperparameter t=0.5.On the left,The hyperparameter was set to t=2.This plot is generated using the exact algorithm with CSI-score.



(a) Asia network



(b) Figure illustrates the CPDAG for Asia Network

Figure 4.6

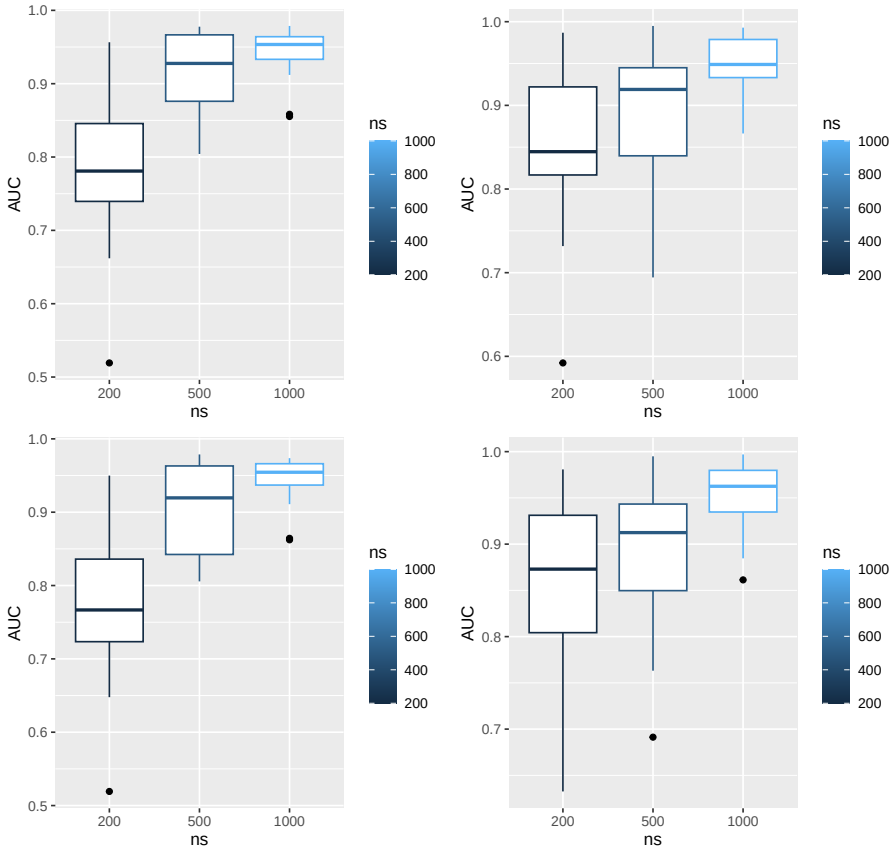


Figure 4.7: This plot is for the Asia network.The box-plot in upper left corner shows AUC for prediction of ancestral paths using MCMC with CSI-score.The box-plot on lower left corner shows AUC using the exact-algorithm on CSI-score.The sample sizes for these vary with ns=200,500,1000.The box-plot in upper right corner shows AUC using MCMC with CPT-score.The box-plot in lower right corner shows AUC using exact algorithm with CPT-score.The sample sizes for these vary with ns=200,500,1000.The CPT scores are calculated on different data-sets then the CSI-score.The comparison is done column-wise.

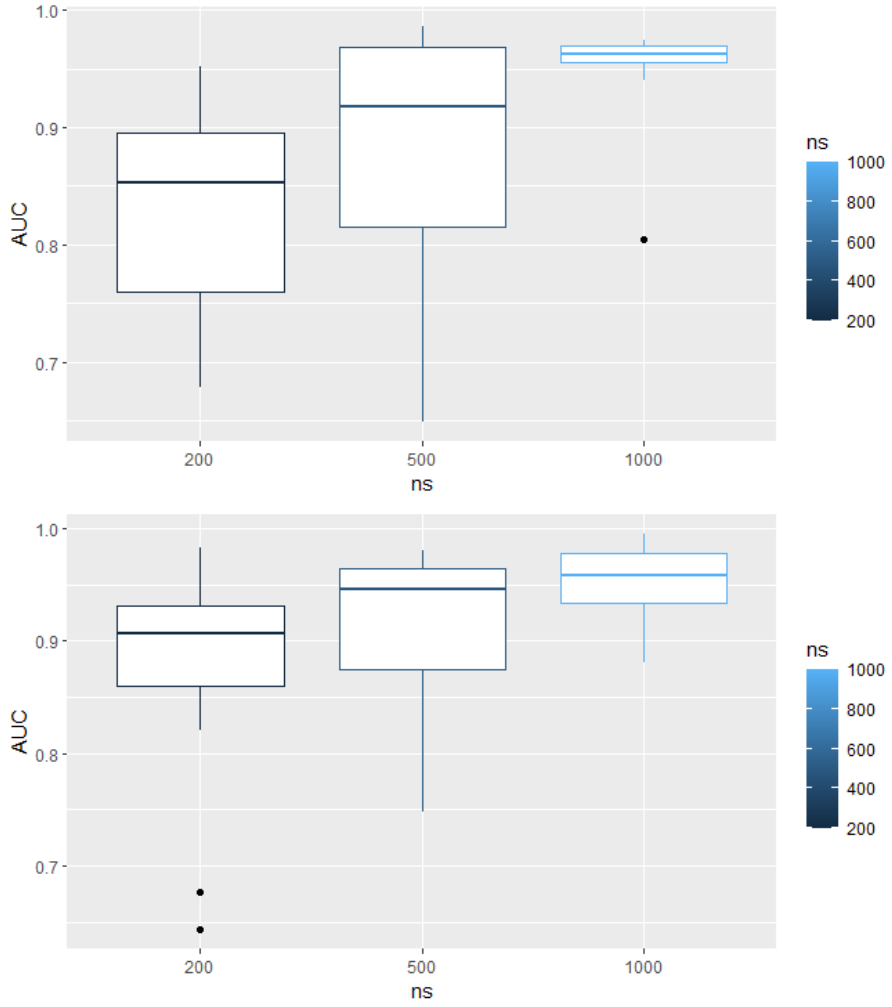


Figure 4.8: This plot is for the Asia network.Box plots showing AUC for prediction of ancestral paths using CSI-score and CPT-scores on the same dataset.This is done using the exact-algorithm.Upper box-plot shows the result for CSI-score.The CSI-scores are calculated with the hyper-parameter of the prior set to t=0

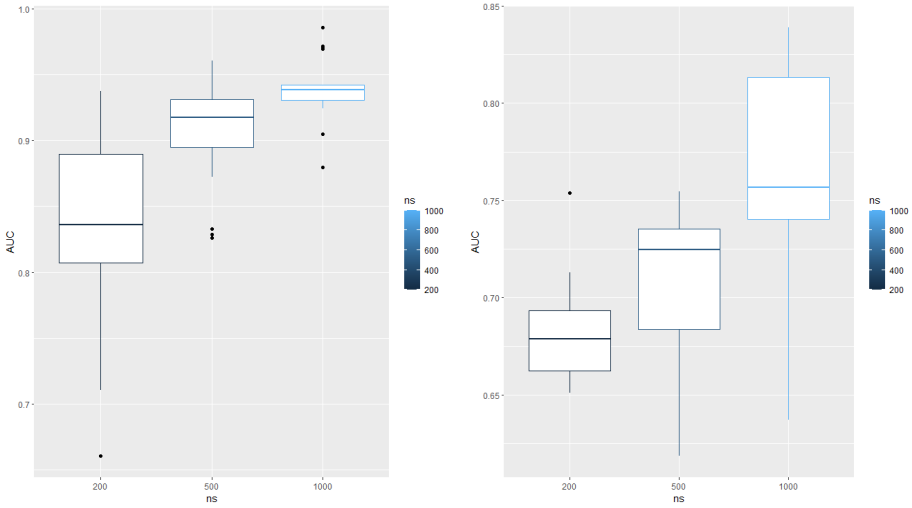
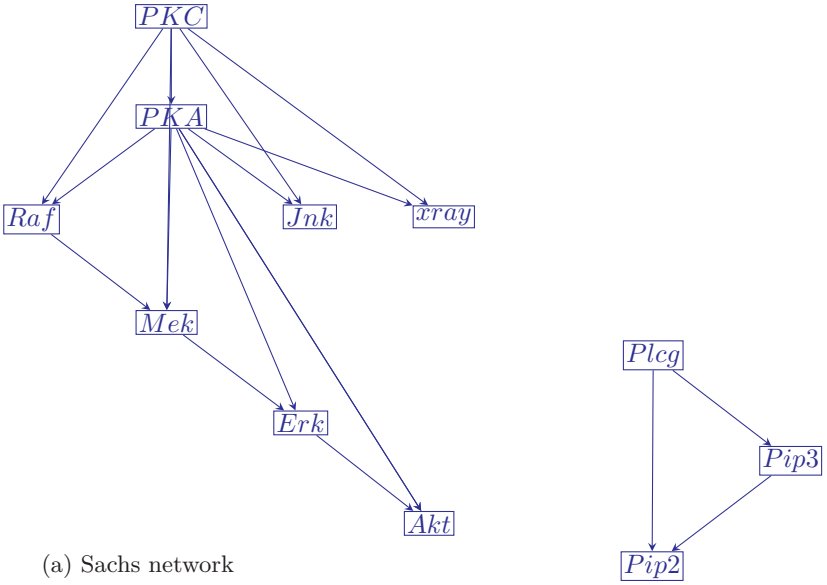
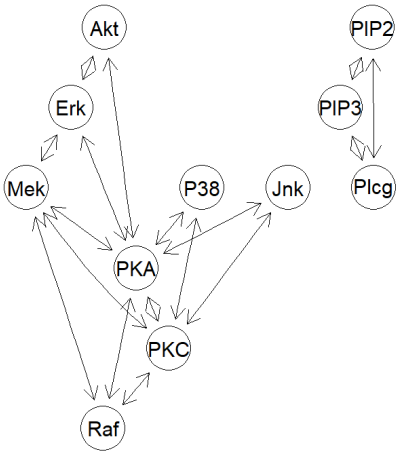


Figure 4.9: This plot is for the Asia network.Figure to the right shows AUC for prediction of ancestral paths prior hyper-parameter t=0.5.On the left,the hyper-parameter was set to t=2.This plot is generated using the exact algorithm.



(a) Sachs network



(b) Figure illustrates the CPDAG for Sachs Network

Figure 4.10

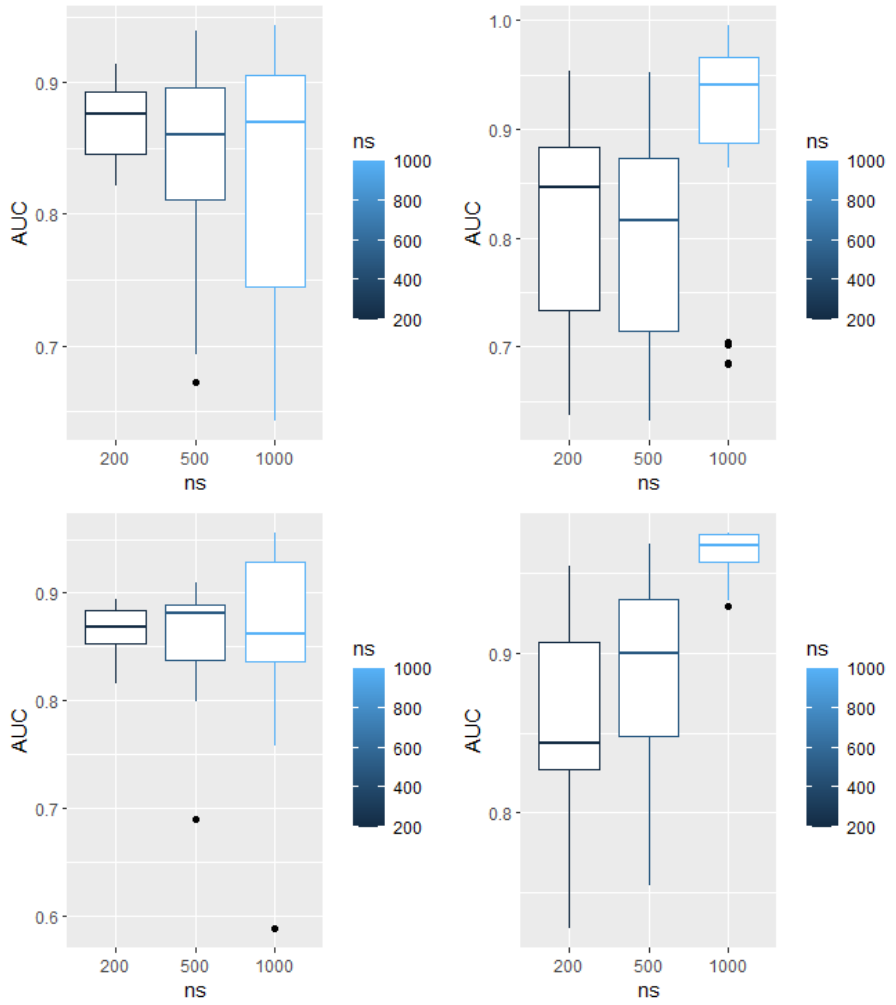


Figure 4.11: This plot is for the Sachs network.The box-plot in upper left corner shows AUC for prediction of ancestral paths using MCMC with CSI-score.The box-plot on lower left corner shows AUC using exact-algorithm with CSI-score.The sample sizes for these vary with ns=200,500,1000.The box-plot in upper right corner shows AUC using MCMC with CPT-score.The box-plot in lower right corner shows AUC estimates using exact algorithm with CPT-score.The sample sizes for these vary with ns=200,500,1000.The CPT scores were calculated on the same data-sets as the CSI-score.The CSI-scores were calculated with the hyper-parameter of the prior set to t=2.

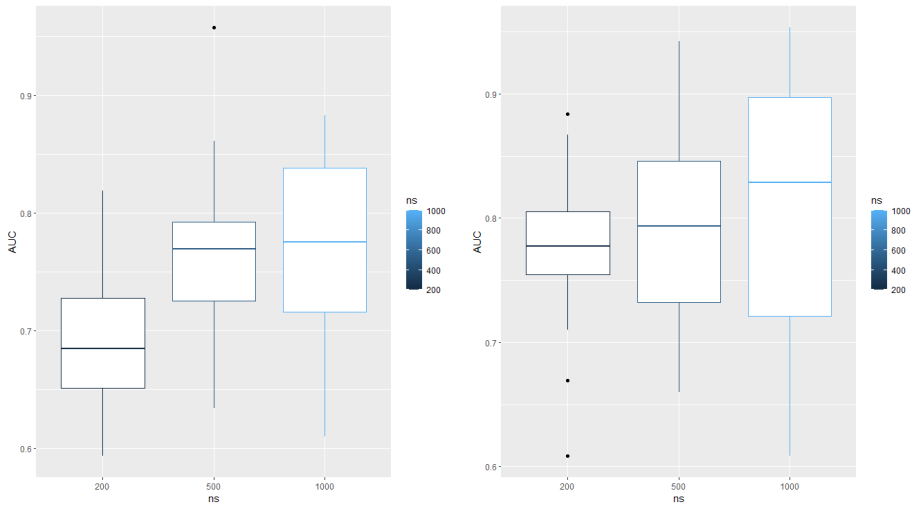


Figure 4.12: This plot is for the Sachs network.Figure to the right shows AUC for prediction of ancestral paths with the prior hyper-parameter t=0.On the left,the hyper-parameter was set to t=0.5.This plot was generated using the exact algorithm with the CSI-score.

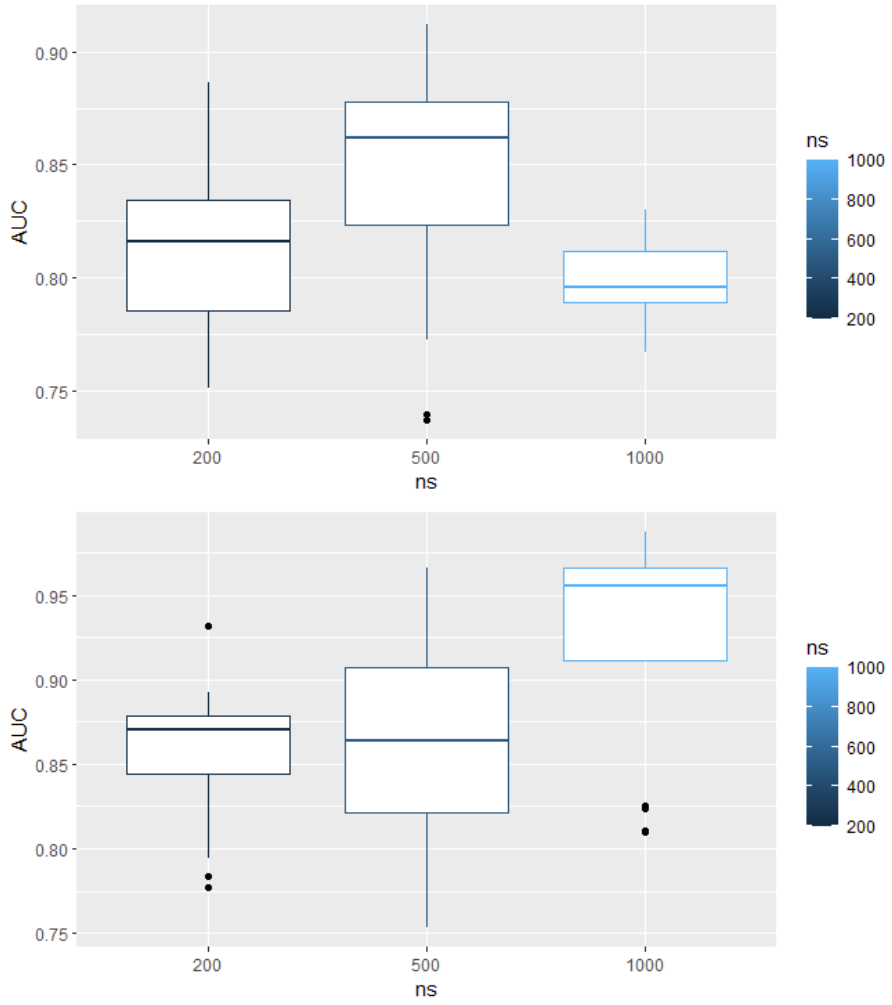
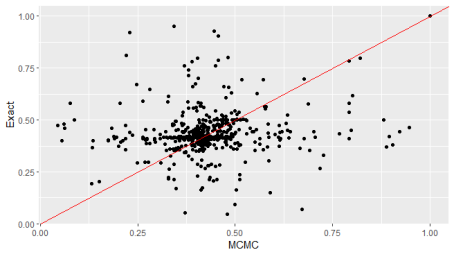
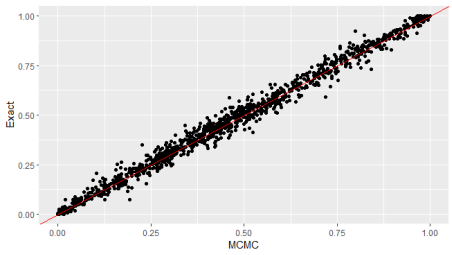


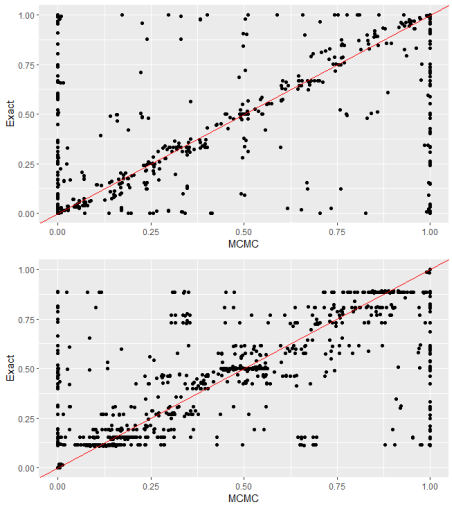
Figure 4.13: This plot is for the Sachs network.Box plots showing AUC for predicting direct causal relations using CSI-score and CPT-scores on the same data-set.This is done using MCMC. The upper box-plot shows the result for CSI-score which was calculated with t=2.



(a) Convergence plot for Survey Network



(b) Convergence plot for Asia Network



(c) Convergence plot for Sachs Network

Figure 4.14: Convergence plot for network Asia,Survey and Sachs.In each plot the upper figures compares the result of 20 MCMC runs compared to the exact algorithm for CSI-score while the plot below shows the comparison for CPT-scores

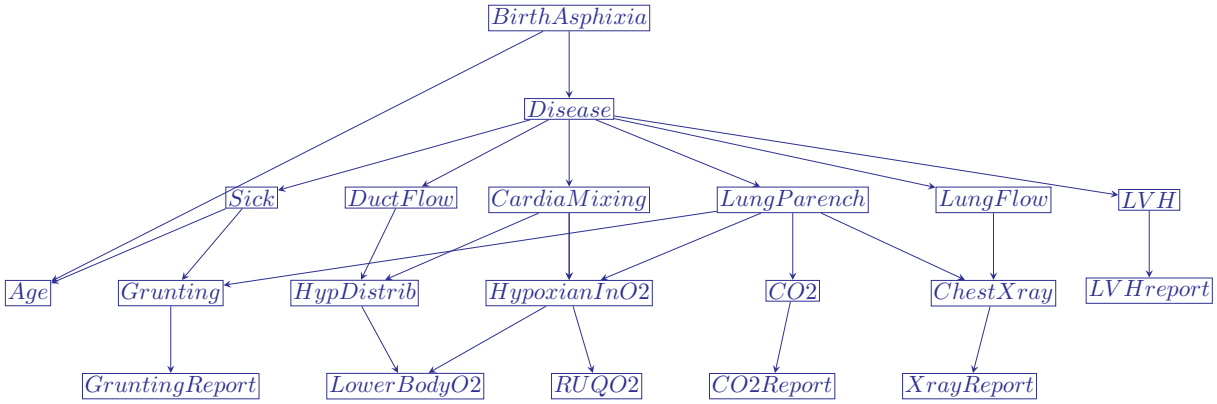


Figure 4.15: Child network

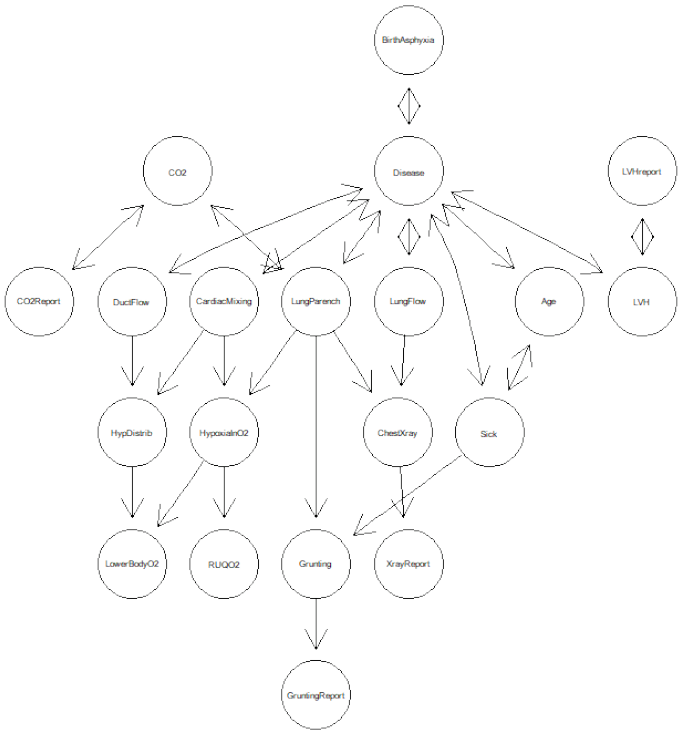


Figure 4.16: Figure illustrates the CPDAG for Survey Network

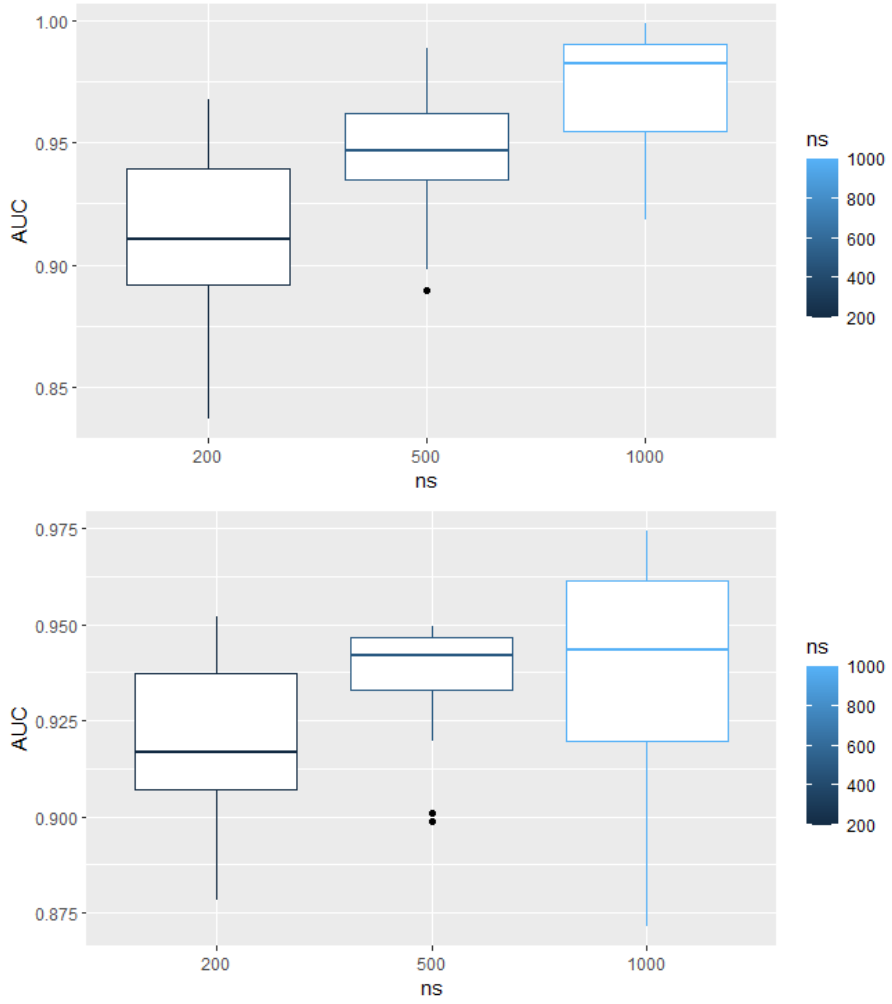


Figure 4.17: This plot is for the Child network.Box plots showing AUC for prediction of ancestral paths using CSI-score and CPT-scores on the same dataset.This is done using the MCMC.Upper box-plot shows the result for CSI-score.The CSI-scores are calculated with the hyper-parameter of the prior set to t=0.5.

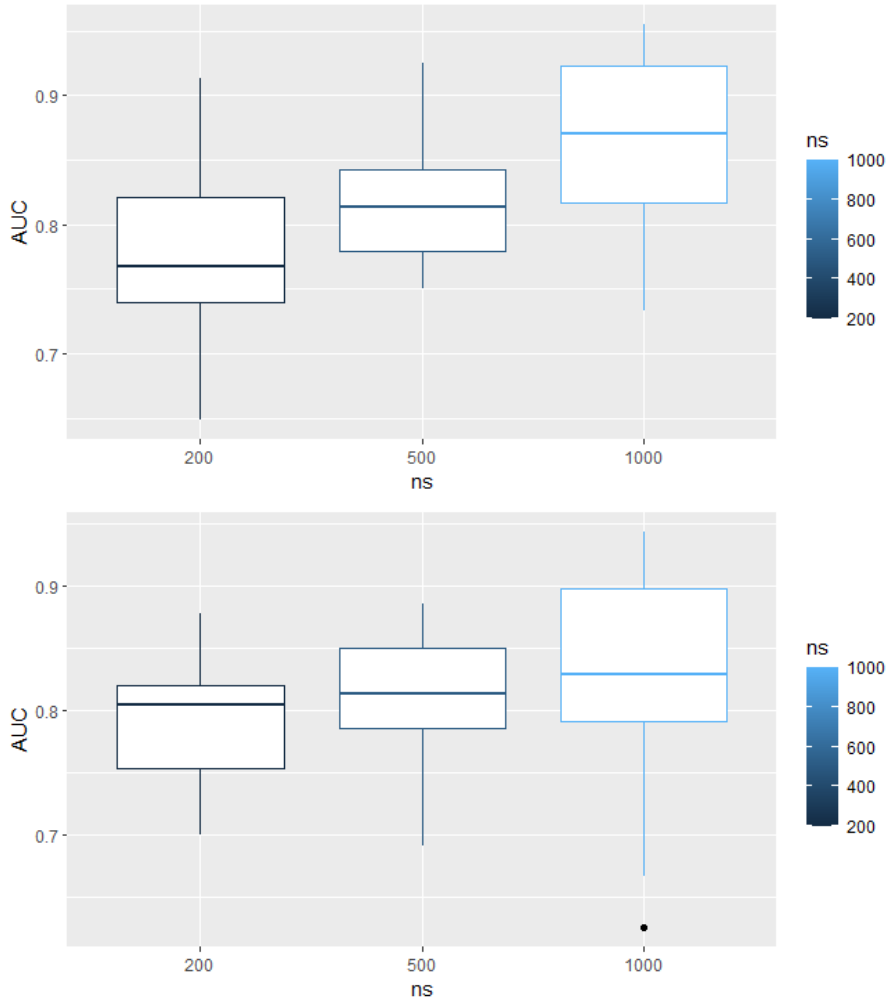


Figure 4.18: This plot is for the Child network.Box plots showing AUC for predicting direct causal relations using CSI-score and CPT-scores on the same data-set.This is done using MCMC. The upper box-plot shows the result for CSI-score which was calculated with t=0.5.

4.3 Discussion of results

From the plots comparing the AUC results of the MCMC compared to the exact method one see that they look similar, but there is a bit more variance for MCMC results. The convergence plots in Figure 3.16 of the Sachs network and the Survey network shows some tendency of bad mixing. This is apparent in the plot for the Sachs network to a much larger extent. The structure MCMC implemented is known for having this problem.One reason for this is the preposed moves in the space of DAGs being small. The convergence therefore can be expected to be slow especially for larger networks. The AUC plot shows that this problem is not a hinder for good AUC accuracy.
One see that the prior hyper-parameter has a big influence on the result for the CSI-scores. For each dataset and BN structure there is some optimal parameter. This hyper-parameter is not possible to find when the underlying BN structure is not known. We are testing our method on known structures in order to see which method performs better when applied to data-sets where the true BN structure is not known. Our method of comparing the methods require that the true BN is known. Therefore we do not try to pinpoint the optimal hyper-parameter through some iterative method. We choose the approach of testing a few values for the hyper-parameter in order to see whether the result changes.
One can see from the box-plot of the Asia network and the Survey network that t=0 (a uniform prior) in Figures 4.4 and Figures 4.8 does better then enforcing sparsity on the structures by increasing the value of t (Figure 4.5 and 4.9). The Sachs network is kind of the outlier here because the improvement is big when setting t=2 in Figure 4.11 compared to when t=0 or t=0.5(Figure 4.12).However, even when t=2, the CSI-score still does worse compared to the CPT scores when looking at sample sizes 500 and 1000. This observation can indicate that for some networks and for some dataset sizes it is best to treat more parameters as different instead of putting them equal.In a certain dataset the parameters might seem similar to each other.One should therefore be cautious. Depending on if the methods perform equally well or poorly the reasons are different. When the dataset is able to capture important parameter equalities that exist in the true distribution,this might lead to the result being improved or staying equal.The equality might come from that the direction of the edges implied by the data are irrelevant in the CSI-equivalence class when using CSI-score and I-equivalence class when using CPT-score.It can also mean that the CSI trees grows fully giving full CPD representations.For the case where the methods perform equally poorly, if the true parameters are almost equal, then in the empirical distribution they might become more equal when the data sample size is small. Parameter reduction in this case will not help to determine which parameters actually is equal within the set of parameters that seems to be equal from the data.In addition, when the dataset is to small it is hard to compute any of the parameters accurately. The CPD tables of the Survey network could be explained by this logic. The parameters that involve node E when E is both part of the parent set and when the parameter is a probability of a value E given its parent set, are almost equal in the true distribution, and 4 out of 7 edges in the BN network are attached to E.For $ns = 200, 500, 1000$, the two methods perform equally poorly which can be seen in Figure 4.4.

How close to equal some parameters within the CPTs for the true distribution are will determine the result of setting them explicitly equal based on the dataset generated from this distribution and at the same time determine how close the estimate of the two methods are. If the dataset make the parameters seem more equal then they are,the csi-method will underperform because it might set parameters equal when the minor difference in the empirical distribution is based on actual inequality in the true distribution. At some point the distribution starts being captured by the data-set when the sample size is increased. On these datasets setting equality when the parameters seem to be equal in the empirical distribution has the advantage of resulting in more data for each parameter estimation without having the downside of ignoring the parameters that are not equal in the true distribution. We are using a greedy algorithm which means that we will not be able to find the optimal parameter reduction. What the greedy algorithm does for us is ranking the parents in the parent set of a node through putting equality on some of the parameters based on the data while ignoring others. If the data reflect the distribution, one should be able to find some of the important parents.If all the parameter equalities given that the dataset reflect the true distribution had been found one would be able to find more important parents. By finding important parents, one has determined the edge direction of certain edges in global structure thus hopefully making it easier to find the edge directions of the true underlying BN structure.

When the data is large enough, the edge direction becomes apparent automatically at least for these networks where the underlying true distribution is known.We can see that for the network Asia and Child network for sample size ns=1000, there is a variance reduction and a small increase in mean value for the CSI based scores compared to the CPT based scores. For the Survey network, an improvement is made when $ns = 5000$. The Sachs network might show what is illuded to above,where the data-set does not reflect the distribution well enough, so that parameter equality does not have a positive effect. This is the case for all sample sizes. Setting parameter equal in this situation might be giving a disproportionate probability to the wrong edges. Here it seems that enforcing full CPD representations work better then estimating some of them more accurately.

CHAPTER 5

Conclusion

In this work we have considered the problem of inferring causal relationships from data. For this purpose, we presented the framework Bayesian networks in order to define the problem of doing causal discovery given data into a structure learning problem. We defined the concept of conditional independence seen through the graph structure of the framework together with how the independencies in an accommodating family of distributions (CPD Tables) can be captured by the graph structure.We then defined the relationship between a BN and a causal model, which is mainly that the edged of the BN entails a causal relation.In addition, we defined the concept of context-specific independence for CPD Tables in form of a CSI-tree, which is a more general way of looking at conditional independence.

We introduced the Bayesian score illustrated its decomposability properties, which we used further in order to be able to compute the score more efficiently, by computing its components scores separately before later putting them together. More specifically,we illustrated that the components scores are the node-wise contributions condition on its parents joint configurations. We defined the standard methodology of computing the Bayesian score, under standard CPT-tables, before defining our methodology of computing the score under CSI-trees learned with the help of greedy hill climb.

In order to take into account the uncertainty that exist in learning causal models, we considered two algorithms Marcov Chain Monte Carlo(MCMC) over structures and a state-of-the-art dynamic programming algorithm used together with Bayesian model averaging in order to estimate the models posterior distribution given data. We applied the considered approach in a simulation study.

The procedures was compared by generating data from known BNs, applying the different procedures and illustrating the comparisons with box-plots of AUC.Finally, we ended with some discussion of the results. A comparison between CPTs and CSI-trees show that no significant improvement was made on the tested networks .However for some data sizes some improvement could be seen. One reason might be that no exact CSI-tree representation of the conditional distribution exist for these networks, since the true distributions are defined through CPD tables. Another reason might be that it was necessary to regulate the model fit with a model structure prior to avoid overfitting in the learning process. The prior used in this work might have been suboptimal. A comparison between MCMC and state-the-art dynamic programming algorithm

shows that the result under AUC are similar, however the convergence of the MCMC over structure for some networks tested was slow.

Bibliography

ref8 [Gel+04] Gelman, A. et al. *Bayesian Data Analysis*. 2nd ed. Chapman and Hall/CRC, 2004.

ref3 [GH08] Grzegorczyk, M. and Husmeier, D. 'Improving the structure MCMC sampler for Bayesian networks by introducing a new edge reversal move'. In: *Machine Learning* vol. 71 (June 2008), pp. 265–305.

ref6 [HGC95] Heckerman, D., Geiger, D. and Chickering, D. 'Learning Bayesian Networks: The Combination of Knowledge and Statistical Data'. In: *Machine Learning* vol. 20 (Sept. 1995), pp. 197–243.

ref2 [KF09] Koller, D. and Friedman, N. *Probabilistic Graphical Models: Principles and Techniques*. Adaptive computation and machine learning. MIT Press, 2009.

ref5 [Pen+13] Pensar, J. et al. 'Labeled Directed Acyclic Graphs: a generalization of context-specific independence in directed graphical models'. In: *Data Mining and Knowledge Discovery* vol. 29 (Oct. 2013).

ref4 [Pen+15] Pensar, J. et al. 'The role of local partial independence in learning of Bayesian networks'. In: *International Journal of Approximate Reasoning* vol. 69 (Nov. 2015).

ref7 [Pen+20] Pensar, J. et al. 'A Bayesian Approach for Estimating Causal Effects from Observational Data'. In: *Proceedings of the AAAI Conference on Artificial Intelligence* vol. 34 (Apr. 2020), pp. 5395–5402.

ref1 [TH12] Tian, J. and He, R. 'Computing Posterior Probabilities of Structural Features in Bayesian Networks'. In: *CoRR* vol. abs/1205.2612 (2012). arXiv: 1205.2612.

APPENDIX A

R code

A.1 Main File

```
1
2
3
4
5  #import packages.
6  library(bnlearn)
7  library(MASS)
8  library(tidyverse)
9  library(CARRoT)
10 library(data.table)
11 library(gRbase)
12
13
14
15 source("runfile_apply.R")
16 source("run_apply_csi.R")
17 source("neede_functions_MCMC.R")
18
19
20
21 #load rda file of networks from bnlearn repository.
22
23 load("asia.rda")
24 load("survey.rda")
25 load("sachs.rda")
26 load("child.rda")
27 load("earthquake.rda")
28 load("alarm.rda")
29
30
31
32 #generate cpdag of loaded network.
33 d=cpdag(bn)
34
35
36 library(graph)
37 library(igraph)
38 #plot cpdag.
39 plot(as_graphnel(as.igraph(d)))
40
41
42 #detach packages because they had a conflict with Rbase library.
43 detach("package:gRbase", unload=TRUE)
44 detach("package:igraph", unload=TRUE)
45
46
47 #adjacency matrix of loaded network.
48 true_matrix=amat(bn)
49
50
51 #Direct ancestral relatoin matrix for loaded network.
52 true_matrix_1=solve(diag(11)-(true_matrix))
53 true_matrix_1=1*apply(true_matrix_1, 2, function(x) x>=1)
54
55
56 #nr of nodes in network.
57 nr_nodes=11
58 #maxparentsize for scorefile.
59 max_parent_size=4
60
61
62 #find how many values each variable in network takes by generating a large dataset.
63 data <- rbn(x = bn, n = 100000)
64
65 data_set_large=as.data.frame(map_df(data, as.numeric))
66
67 p=lapply(1:nr_nodes, function(x){unique(data_set_large[,x])})
68
69
```

```
70 #     ##########################################################################################################################
71 #Function:write_func
72 #     ##########################################################################################################################
73 #Input:
74 #read_this: path of scorefile containing P(D|G).
75
76 #nr_nodes:how many nodes in network.
77
78 #paramter:how many parentcombinations for each node.
79
80 #name:what name should be given to new scorefile containing P(G,D).
81 #t:tuning paramter for prior.
82 #nr_data:how many data samples is scorefile based on
83
84 #Output:scorefile containing scores P(G,D)
85
86
87 #This function adds prior for graph G added on the P(D|G) contained in scorefile , in
       order to estimate P(G,D).
88 #     ##########################################################################################################################
89 write_func=function(read_this,nr_nodes,parameter,name,t,nr_data){
90
91  # These rows in scorefile are excluded sinse these rows contains node together with how
       many parent-combination of that node.
92  seqq=c(seq(1,nr_nodes*parameter,parameter)+1:nr_nodes,1)
93
94  #Add prior on rest of the rows.
95  for(i in setdiff(1:nrow(read_this),seqq)){
96   component=as.numeric(strsplit(read_this[i,]," ")[[1]])
97   component[1]=component[1]-(1+t)^(length(component)-2)*log(nr_data)
98
99   read_this[i,]=gsub(",","",toString(component))
100
101  }
102
103  #convert scorefile back to type .score
104  colnames(read_this)=NULL
105
106
107
108  read_this_name=paste0('C:/Users/rasyd/Documents/gitrepo/master/score_folder/scores/csi_
       sachs/n200/t05/',"temp.",name,".score")
109  #write new scorefile
110  write.matrix(read_this,sep=" , ",file=read_this_name)
111
112
113
114
115 }
116
117
118
119 #     ##########################################################################################################################
120 #Function:run_write
121 #     ##########################################################################################################################
122 #Input:
123 #nr_of:how many scorefiles should be made.
124
125 #nr_nodes:how many nodes in network.
126
127 #paramter:how many parentcombinations for each node.
128
129 #t:tuning paramter for prior.
130
131 #nr_data:how many data samples is scorefile based on
132
133 #Output:scorefiles containing scores P(G,D)
134
135
136 #This function adds prior for multiple scorefiles where how many scorefiles are  specified
       by nr_of.
137 #     ##########################################################################################################################
138
139
140 run_write=function(nr_of,parameter,nr_nodes,t,nr_data){
141
142  for(j in 1:nr_of){
143   add=samp[j]
144   pas_string=toString(add)
145   if(k==1){
146    score_type=paste0("cat","type",pas_string)}
147   if(k==2){
148    score_type=paste0("csi","type",pas_string)
149   }
150   # define path of scorefiles
151   read_this=paste0('C:/Users/rasyd/Documents/gitrepo/master/score_folder/scores/csi_sachs/
       n200/',"temp.",score_type,".score")
152   read_this=read.csv(file = read_this, header = FALSE)
153
154   #calling write_func
```

```
  write_func(read_this,nr_nodes,parameter,score_type,t,nr_data )

 }


}
# Input variables for prior function write_func.
#Index names for score files.
samp=3:22
#If k=1 scorefile name is of cattype if k=2  scorefile name is of csitype.
k=1
#Number of parent combinations for each node.
parameter=386
#nr_nodes assigns how many nodes in network.
nr_nodes=11

#Tuning paramter of prior.
t=0.5
#Size of the data the scores are calculated from.
nr_data=200
#Add prior on 20 score files.
run_write(20,parameter,nr_nodes,t,nr_data)


#
      ##############################################################################################################

#Function:run_func_1
#
      ##############################################################################################################

#Input:
#max_parent_size:max parent size for scorefile.

#nr_nodes:how many nodes in network.

#paramter:how many parentcombinations for each node.

#j:iterator index.

#p:contains a list of lists where each list contains the values of a node in the network.

#k:is a vector used to set name on scorefile depending on if the scorefile is CSI type or
      CPT type.


#Output:scorefiles containing scores P(D|G).


#This function calculates  the scorefiles cotaining all P(D|G).
#
      ##############################################################################################################


run_func_1=function(max_parent_size,nr_nodes,j,p,k){

 #generate data of size ns from loaded network
 data <- rbn(x = bn, n = ns)
 #convert variables from factor to numeric
 data_set=as.data.frame(map_df(data, as.numeric))
 #convert to data.table
 setDT(data_set)


 add=j+samp
 pas_string=toString(add)

 #Set name of scorefile.based on k scorefile name changes
 if(k[1]==1){
  score_type_1=paste0("cat","type",pas_string)}
 if(k[2]==2){
  score_type_2=paste0("csi","type",pas_string)
 }

 #calulates and writes CPT based log-marginal likelihood to scorefile
 csitree_calc_parent_score_to_file_2(data_set,score_type_1 , max_parent_size, file_out="
       temp",p)
#calulates and writes CSI based log-marginal likelihood to scorefile
 csitree_calc_parent_score_to_file_3(data_set,score_type_2 , max_parent_size, file_out="
       temp",p)

}


#
      ##############################################################################################################

#Function:run_func_2
#
      ##############################################################################################################

#Input:
#max_parent_size:max parent size for scorefile.

#nr_nodes:how many nodes in network.

#true_matrix:some transformation of the adjacency matrix of the network

```

```
#j:iterator index.

#p:contains a list of lists where each list contains the values of a node in the network

#cpt_or_csi:which scoretype to run MCMC on.


#Output:AUC


#This runs MCMC over a scorefile
#
      ##########################################################################################################################


run_func_2=function(max_parent_size,nr_nodes,j,p,true_matrix,cpt_or_csi){

 add=j+samp
 pas_string=toString(add)
 if(cpt_or_csi==1){
  score_type=paste0("cat","type",pas_string)}
 if(cpt_or_csi==2){
  score_type=paste0("csi","type",pas_string)
 }

 #read scorefiles
 read_this=paste0('C:/Users/rasyd/Documents/gitrepo/master/score_folder/scores/csi_sachs/
      t2/',"temp.",score_type,".score")
 read_this=read.csv(file = read_this, header = FALSE)
 #run MCMC
 MCMC_AUC=func_MCMC(read_this,true_matrix,max_parent_size,j,nr_nodes,ns,cpt_or_csi)

 return(MCMC_AUC)
}


#Adding length of previous run.
samp=20

#Set data size.
ns=1000
#How many data files.
n=20
#Which score type.
k=c(1,2)

cpt_or_csi=1

#Run in parallel.
#Import packages needed for parallel runs.
library(parallel)
library(doSNOW)
#Divide processor into smaller clusters.
cl <- makeCluster(6,type = "SOCK",outfile="log.txt")
registerDoSNOW(cl)



#Feed clusters functions the input variables needed.
clusterExport(cl,c("func","func_2","samp","bn","k","cpt_or_csi","p","run_func_1","run_func
     _2","cat_calc_parent_score_to_fil_3_plane","CSI_tree_apply_imp_3_mat_B_3","csitree_
     calc_parent_score_to_file_3","csitree_calc_parent_score_to_file_2","func_MCMC","ns",
     "max_parent_size","nr_nodes","true_matrix_1","n"),envir = environment())



#import libraries for each cluster.
clusterEvalQ(cl, c(library(data.table),library(MASS)
,library(tidyverse)
,library(CARRoT),library(bnlearn),library(gRbase)))

#calculate scorefiles.
parLapply(cl,1:n,function(x){run_func_1(max_parent_size,nr_nodes,x,p,k)})

#Run MCMC on them and return AUC vector of all runs either for csi-score or for cpt score.
AUC_vec=unlist(parLapply(cl,1:n,function(x){run_func_2(max_parent_size,nr_nodes,x,p,true_
     matrix_1,cpt_or_csi)}))



# Stop cluster on master
stopCluster(cl)



#All plots are generated with a similar construction to the following code.
#Make an empty list where every element in the list will be a matrix.There will be four
     matrices each one containing the AUC of CSI score results using MCMC,
#CSI score AUC using  exact algortihm,CPT score AUC using  MCMC and CPT score AUC using
     exact algortihm.
#Every column in the matrices represents the result for a specific data sample size
Plot_list=list()
#we will generate 20 AUC for every data sample size
#set k=1
```

```
k=1
#set m=20
m=20
# ns  defines the domain of data sample sizes
ns=c(1000,500,200)
#define empty AUC vector
AUC_calc=matrix(0,20,1)
#for 4 different methods of computing AUC
for(j in 1:4){
 #for all sample sizes
 for(i in 1:length(ns)){
  #if CSI score posterior ansectral path matrices  using MCMC
  if(j==1){
   #extract names of all matrices
   matrix_name_extract=list.files(path="C:/Users/rasyd/Documents/gitrepo/master/score_folder/matrix/csi_sachs/direct_cause/joint", pattern=NULL, all.files=FALSE, full.names=FALSE)
   match_sring=grep(toString(ns[i]),matrix_name_extract)
   matrix_name_extract=matrix_name_extract[match_sring]



   matrix_name_extract=sort(matrix_name_extract,decreasing = TRUE)

   paste_to_each=paste("C:/Users/rasyd/Documents/gitrepo/master/score_folder/matrix/cat_sachs/direct_cause/joint/",matrix_name_extract)


   paste_to_each=lapply(1:length(paste_to_each),function(x)gsub(" ", "", paste_to_each[x]))
  }
  #if CPT score posterior ansectral path matrices  using MCMC
  if(j==2){


   #extract names of files containing matrices
   matrix_name_extract=list.files(path="C:/Users/rasyd/Documents/gitrepo/master/score_folder/matrix/cat_sachs/direct_cause/joint", pattern=NULL, all.files=FALSE, full.names=FALSE)


   match_sring=grep(toString(ns[i]),matrix_name_extract)
   matrix_name_extract=matrix_name_extract[match_sring]



   matrix_name_extract=sort(matrix_name_extract,decreasing = TRUE)

   paste_to_each=paste("C:/Users/rasyd/Documents/gitrepo/master/score_folder/matrix/cat_sachs/direct_cause/joint/",matrix_name_extract)


   paste_to_each=lapply(1:length(paste_to_each),function(x)gsub(" ", "", paste_to_each[x]))


  }
  #if CSI score posterior ansectral path matrices  using exact algorithm
  if(j==3){
   #extract names of files containing matrices
   matrix_name_extract=list.files(path="C:/Users/rasyd/Documents/gitrepo/master/score_folder/matrix/exact_csi_sachs/t0", pattern=NULL, all.files=FALSE, full.names=FALSE)


   paste_to_each=paste("C:/Users/rasyd/Documents/gitrepo/master/score_folder/matrix/exact_csi_sachs/t0/",matrix_name_extract)

   paste_to_each=paste_to_each[order(as.numeric(gsub("[^0-9]+", "", paste_to_each)))]

   paste_to_each=lapply(1:length(paste_to_each),function(x)gsub(" ", "", paste_to_each[x]))[k:m]


  }

  #if CPT score posterior ansectral path matrices  using exact algorithm
  if(j==4){
   #extract names of files containing matrices
   matrix_name_extract=list.files(path="C:/Users/rasyd/Documents/gitrepo/master/score_folder/matrix/exact_cat_survey", pattern=NULL, all.files=FALSE, full.names=FALSE)


   paste_to_each=paste("C:/Users/rasyd/Documents/gitrepo/master/score_folder/matrix/exact_cat_survey/",matrix_name_extract)

   paste_to_each=paste_to_each[order(as.numeric(gsub("[^0-9]+", "", paste_to_each)))]

   paste_to_each=lapply(1:length(paste_to_each),function(x)gsub(" ", "", paste_to_each[x]))[k:m]


```

```
424
425   }
426
427
428   #Extract matrices
429   Matrix_vec=lapply(1:length(paste_to_each),function(x)unname(as.matrix(read.csv(file =
          paste_to_each[[x]], header = FALSE))))
430
431   #if matrices is from exact algorithm  elements becomes strings.We have to convert them
          to numeric matrices
432   if(j>2){
433    Matrix_vec=lapply(1:length(paste_to_each),function(x)Matrix_vec[[x]][-1,])
434
435
436    Matrix_vec=lapply(1:length(paste_to_each),function(x){matrix(as.numeric(Matrix_vec[[x
          ]]),ncol = ncol(Matrix_vec[[x]]))})
437
438
439
440
441
442   }
443
444
445   #use class_prob to get matrix of two columns.First column reprsents estimated ancestral
          path probabilities sorted in a decending way.Second column
446   #represents true  ancestral vector matrix sorted in the same order as estimated
          probabilities.
447   AUC_input=lapply(1:length(Matrix_vec),function(x)class_prob(Matrix_vec[[x]]))
448
449   #concatinate zero vector with AUC vector of every estimated matrix for ns[i]
450   AUC_calc=cbind(AUC_calc,do.call(rbind,lapply(1:length(Matrix_vec),function(x)AUC(AUC_
          input[[x]][,1], AUC_input[[x]][,2]))))
451
452   k=k+20
453   m=m+20
454  }
455  #Fill in matrices into list
456  Plot_list[[j]]=AUC_calc
457  k=1
458  m=20
459  #redefine zero vector for each j
460  AUC_calc=matrix(0,20,1)
461
462 }
463
464
465
466
467
468
469
470 #remove zero vector from all matrices
471 Plot_list=lapply(1:length(Plot_list),function(x) as.matrix(Plot_list[[x]][,-1]))
472
473
474 #import ggplot2
475 library(ggplot2)
476
477
478
479 #Rechape every matrix in Plot_list to a vector and concatinate it with an indicator vector
        of which column in belonged to
480 box_matrix=lapply(1:4,function(z){Reduce("rbind",
481  Reduce("rbind",lapply(1:ncol(Plot_list[[z]]),function(y){lapply(1:nrow(Plot_list[[z]]),
482   function(x){cbind(Plot_list[[z]][x,y],y*as.numeric(x>0))})}))})
483 #give name to every element in box_matrix
484 lapply(1:length(box_matrix),function(x)colnames(box_matrix[[x]])<-c("AUC","ns"))
485
486
487 #if sorting was done wrong reshuffle an the scale x axis correctely
488 for(i in 1:4){
489  if(i==1){
490   ns=c(1000,500,200)
491
492
493
494
495   apply(matrix(1:nrow(box_matrix[[i]]),1,nrow(box_matrix[[i]])),2,function(x){box_matrix[[
          i]][x,2]<-ns[box_matrix[[i]][x,2]]})
496  }
497  if(i==2){
498   ns=c(1000,500,200)
499
500
501   apply(matrix(1:nrow(box_matrix[[i]]),1,nrow(box_matrix[[i]])),2,function(x){box_matrix[[
          i]][x,2]<-ns[box_matrix[[i]][x,2]]})
502  }
503
504  if(i==3){
505   ns=c(1000,500,200)
506
507   apply(matrix(1:nrow(box_matrix[[i]]),1,nrow(box_matrix[[i]])),2,function(x){box_matrix[[
          i]][x,2]<-ns[box_matrix[[i]][x,2]]})
508  }
509
510  if(i==4){
511   ns=c(500,200,1000)
512
513
514   apply(matrix(1:nrow(box_matrix[[i]]),1,nrow(box_matrix[[i]])),2,function(x){box_matrix[[
```

```
        i]][x,2]<<-ns[box_matrix[[i]][x,2]]})
515
516  }
517 }
518
519 #assign each plot to a variable and generate boxplot for each variable
520 for(i in 1:length(box_matrix) ){
521  token_seq=paste("token_",i,"")
522
523  assign(token_seq,ggplot(as.data.frame(box_matrix[[i]]), aes(x=(samples=factor(ns)), y=AUC
       ,color=ns)) + labs(y= "AUC", x = "ns")
524  +geom_boxplot(outlier.color="black"))
525 }
526
527 library(gridExtra)
528 #plot a grid-plot of all box-plots
529 grid.arrange(‘token_ 1 ‘,‘token_ 2 ‘,‘token_ 3 ‘,‘token_ 4 ‘)
530
531
532
533
534
535 #
       ##########################################################################################################

536 #Function:compute_roc
537 #
       ##########################################################################################################

538 #Input:
539 #compare:estimated ancestral path matrix .
540
541 #i:depending on i the color of the plot changes.
542
543
544
545
546 #Output:Roc curve
547
548
549 #This function for generating Roc curves
550 #
       ##########################################################################################################

551
552 compute_roc=function(compare,i){
553  #list of colors of plot
554  color=c("red","blue","yellow","brown")
555
556  #flatten matrix to vector
557  rechape_compare=c(compare)
558  #attain order of  rechape_compare
559  rechape_compare_order=order(rechape_compare,decreasing = TRUE)
560
561  #order rechape_compare in descending order
562  order_MCMC=rechape_compare[rechape_compare_order]
563
564  #flatten true ancestral path matrix
565  rechape_true=c(true_matrix)
566  #sort it in the same oder as rechape_compare
567  order_true=rechape_true[rechape_compare_order]
568
569  #concatinate vectors
570  compare_true_with_mcmc=cbind(order_MCMC,order_true)
571  #count how many 1’s and 0’s in order_true
572  how_many_total=table(order_true)
573
574  #count how many 1’s and 0’s exist every time one adds an element of  order_true
575  how_many_one_zero=lapply(1:nrow(compare_true_with_mcmc),function(x)table(order_true[1:x])
       )
576
577  #how_many_one_zero is a list, transforming it into a matrix
578  how_many_one_zero_rbind=do.call(rbind,how_many_one_zero)
579
580  #how many ones are in order_true
581  how_many_total=matrix(how_many_total)[,1]
582
583  #when number of 0’s is 0 in first rows in how_many_one_zero_rbind .On these rows in  how_
       many_one_zero_rbind number of 1’s get duplicated
584  #therefore one had to manually set 0 on these rows.
585  #In addition how_many_one_zero_rbind changes the placement of what is count of zero and
       what is count of 1 for these rows.The rest of the rows that has a sum smaller then
       how_many_total[2] stays unchanged.
586  how_many_one_zero_rbind=t(apply(how_many_one_zero_rbind, 1, function(x)if(sum(x)<= how_
       many_total[2]){ sort(replace(x, duplicated(x), 0))}else{x}))
587
588
589  #calculate TPR and FPR
590  how_many_one_zero_rbind_transform=apply(matrix(1:ncol(how_many_one_zero_rbind),1,ncol(how
       _many_one_zero_rbind)), 2,
591  function(x) how_many_one_zero_rbind[,x]/how_many_total[x])
592
593  #create Roc curve
594  lines(how_many_one_zero_rbind_transform[,1],how_many_one_zero_rbind_transform[,2],type =
       "l",col=color[i])
595
596 }
597 #import matrices
598 call_on_al_matrix=read.csv("C:/Users/rasyd/Documents/gitrepo/master/BIDA/matrix1.txt",sep=
       ",",header = TRUE)
```

```
599 call_on_al_matrix_2=read.csv("C:/Users/rasyd/Documents/gitrepo/master/BIDA/matrix.txt",sep
        =",",header = TRUE)
600 #plot empty plot
601 plot(NA, type="n", xlab="", ylab="", xlim=c(0, 1), ylim=c(0, 1))
602 par(new=TRUE)
603 #draw Roc curves
604 compute_roc(edge_matrix_1_cat,1)
605 compute_roc(edge_matrix_1,2)
606 compute_roc(as.matrix(call_on_al_matrix),3)
607 compute_roc(as.matrix(call_on_al_matrix_2),4)
608 legend("bottomright", cex=0.5, title="ROC-Curve",
609 c("MCMCcat","MCMCSI","Exact_cat","Exact_csi"), fill=c("red","blue","yellow","brown"),
        horiz=TRUE)
```

A.2 Log-marginal likelihood computation for CPT based score

```
1
2  #
       ##########################################################################################################
3  #Function:cat_calc_parent_score_to_fil_3_plane
4  #
       ##########################################################################################################
5  #Input:
6  #data:data used.
7
8  #node:which node to calculate log-marginal likelihood from.
9
10 #parent_comb:Specific parent combination
11
12
13
14 #p:contains a list of lists where each list contains the values of a node in the network
15
16
17
18
19 #Output:Return log-marginal-likelihood
20
21
22 #This function calculates the CPT based log-marginal-likelihood
23 #
       ##########################################################################################################
24
25
26 cat_calc_parent_score_to_fil_3_plane <- function(data,node,parent_comb,p){
27
28
29
30
31  # N  in the BDEU prior is set to 1
32  N <-1
33
34
35  #Function for counting specific configuration in data
36  M_xi_parent_count=function(data,col_2){
37
38   M_xi_parent=data[, .(n = .N), by = col_2]
39   return(M_xi_parent)
40  }
41
42
43  #function for comparing vector with row in matrix
44  comparetorow=function(x,y){
45   nbr=nrow(x)
46   nbc=ncol(x)
47   ret=!vector("logical",nbr)
48   for(i in 1:nbr){
49    for(k in 1:nbc){
50     if(x[i,k]!=y[k]){
51      ret[i]=FALSE
52      break
53     }
54    }
55   }
56   return(ret)
57  }
58
59
60
61
62
63
64
65
66
67  #List of unique values of node
68  uniq_Xi_value=matrix(p[[node]])
69
70
71  #nr of unique values for node
72  nr_uniq_Xi_value1=nrow(uniq_Xi_value)
73
74  #parents of node
75  par=parent_comb
76
77  #columnname of node and nodes parents in dataset
78  col_2=names(data[,.SD,.SDcols=c(node,par)])
79
80  #part of data with nodes parents as columnname
81  parent_set_entries=data[,.SD,.SDcols=c(par)]
82
83
84  #unique configurations in dataset for parent_set_entries
85  uniq_par_value=as.matrix(unique(parent_set_entries))
86
87
88  #number of unique parent_set_enteries in dataset
89  nr_uniq_par_value=nrow(unique(parent_set_entries))
90
```

```
91
92  #list of values of parents to node
93  parent_set_entries_to_alpha=p[c(par)]
94
95  #multiplication of length of all element in list  parent_set_entries_to_alpha
96  in_between_move=lapply(1:length(parent_set_entries_to_alpha),function(x){length(parent_
        set_entries_to_alpha[[x]])})
97  nr_uniq_par_value_to_alpha=Reduce('*',in_between_move)
98
99  #calculate BDEU prior
100 alpha_node_parnode=N/(nr_uniq_Xi_value1*nr_uniq_par_value_to_alpha)
101
102 #Vector containing BDEU prior for all values of node
103 alpha_node_parnode_vec=rep(alpha_node_parnode,nr_uniq_Xi_value1)
104
105 #Sum of all element in alpha_node_parnode_vec
106 alpha_sum_parnode=sum(alpha_node_parnode_vec)
107
108
109 #count how many times every configuration in data set reduced to entries for columnnames
        col_2
110 a=unname(as.matrix(M_xi_parent_count(data,col_2)))
111
112 # CPT score set to 0
113 src=0
114
115
116 #For every parent configuration of node
117 for(parent in 1:nr_uniq_par_value){
118
119
120
121
122  #extract part of matrix "a" that contain the parent configurations of node
123  if(nrow(a)==1){
124
125   b=matrix(a[,-c(1,ncol(a))],nrow = 1)
126
127  }else{
128   if(is.null(nrow(a[,-c(1,ncol(a))]))){
129    b=matrix(a[,-c(1,ncol(a))],ncol = 1)}else{
130    b=(a[,-c(1,ncol(a))])
131   }
132
133  }
134
135  #find index in matrix "a" that matches specific parent configuration.This returns a
        logical vector for indexes of counts where values of node appear
136  #for the specific parent config
137  w=a[comparetorow(b,uniq_par_value[parent,] ),]
138
139
140  #if w contains only one row and is a vector
141  if(is.null(nrow(w))){
142
143   #make zero vector
144   fill=rep(0,nr_uniq_Xi_value1)
145
146   #collect counts from w
147   M_X_split=(w[c(ncol(a))])
148
149   # collect values of node in data from w
150   con=w[c(1)]
151
152
153   #code below is written to have fixed length on count vector no matter how many counts
        exist for node
154
155   #match with theoretical values of node
156   where_in_total=match(uniq_Xi_value,con)
157   #remove NA from match
158   where_in_total=which(where_in_total>0)
159
160   #put counts in zero matrix
161   fill[where_in_total]=M_X_split
162   #rename zero vector
163   M_X_split=fill
164   #sun all elements in M_X_split
165   M_parent_count=sum( M_X_split)
166
167
168  }else{
169   #Else if w contains more then one row and is a matrix.Same procedure is done as when w
        is a vector
170
171   M_X_split=(w[,c(ncol(a))])
172
173   con=w[,c(1)]
174
175
176   fill=rep(0,nr_uniq_Xi_value1)
177
178   where_in_total=match(uniq_Xi_value,con)
179
180   where_in_total=which(where_in_total>0)
181
182
183   fill[where_in_total]=M_X_split
184
185   M_X_split=fill
186
```

```
    M_parent_count=sum(M_X_split)

   }

   #if some counts are different from zero exist in zero vector fill
   if(sum( M_X_split)!=0){


    #add to CPT log-marginal likelihood

    src=src+(lgamma(alpha_sum_parnode)-lgamma(alpha_sum_parnode+M_parent_count)+sum(lgamma
          (alpha_node_parnode_vec+(M_X_split))-lgamma(alpha_node_parnode_vec)))

   }
  }


  #return CPT log-marginal likelihood
  return(src)
 }
```

A.3 CSI based log-marginal likelihood computation

```
#
     ##########################################################################################################
#Function:CSI_tree_apply_imp_3_mat_B_3
#
     ##########################################################################################################
#Input:
#data:data used.

#idented_for:which node to calculate log-marginal likelihood from.

#parent_comb:Specific parent combination


#p:contains a list of lists where each list contains the values of a node in the network



#Output:Return log-marginal-likelihood


#This function calculates the CSI-log-marginal-likelihood
#
     ##########################################################################################################


CSI_tree_apply_imp_3_mat_B_3 <- function(data, parent_set,intended_for,p){
 #set N in BDEU prior to 1
 N <-1

 #set containing node=intended_for together with its parentset
 #set <- c(parent_set,intended_for)

 #function for counting number of time configurations occure in dataset
 M_xi_parent_count=function(data,cols_2){


  M_xi_parent=data[, .(n = .N), by = cols_2]
  return(M_xi_parent)
 }

 #function for comparing vector with row in matrix
 comparetorow=function(x,y){
  nbr=nrow(x)
  nbc=ncol(x)
  ret=!vector("logical",nbr)
  for(i in 1:nbr){
   for(k in 1:nbc){
    if(x[i,k]!=y[k]){
     ret[i]=FALSE
     break
    }
   }
  }
  return(ret)
 }

 #this function is used when selecting the root
 indicator_function_2=function(row,ma,a){
  row_ma=ma[row,]
  row_config=row_ma[!is.na(row_ma)]

  #This part is same as CPT based log-marginal-computation


  if(nrow(a)==1){

```

```
 b=matrix(a[,-c(1,ncol(a))],nrow = 1)

}else{
 if(is.null(nrow(a[,-c(1,ncol(a))]))){
  b=matrix(a[,-c(1,ncol(a))],ncol = 1)}else{
  b=(a[,-c(1,ncol(a))])
 }

}

w=a[comparetorow(b,row_config ),]

M_X_split=0

if(is.null(nrow(w))){

 fill=rep(0,length(val_intended_for))


 M_X_split=(w[c(ncol(a))])
 con=w[c(1)]
 where_in_total=match(val_intended_for,con)
 where_in_total=which(where_in_total>0)

 fill[where_in_total]=M_X_split

 M_X_split=fill


}else{


 M_X_split=(w[,c(ncol(a))])
 con=w[,c(1)]


 fill=rep(0,length(val_intended_for))

 where_in_total=match(val_intended_for,con)
 #where_in_total=where_in_total[!is.na(where_in_total)]
 where_in_total=which(where_in_total>0)
 fill[where_in_total]=M_X_split
 #print(fill)
 M_X_split=fill


}


return( M_X_split)
}
#this function is used after root is selected.Difference between indicator_function_3 and
      indicator_function_2 is the first line in code when defining row_ma
indicator_function_3=function(row,ma,a,next_element_con){
 row_ma=ma[row,-1]


 #This part is same as indicator_function_2
 row_config=row_ma[!is.na(row_ma)]
 row_config=as.numeric(c(row_config,next_element_con))




 if(nrow(a)==1){

  b=matrix(a[,-c(1,ncol(a))],nrow = 1)

 }else{
  if(is.null(nrow(a[,-c(1,ncol(a))]))){
   b=matrix(a[,-c(1,ncol(a))],ncol = 1)}else{
   b=(a[,-c(1,ncol(a))])
  }

 }



 w=a[comparetorow(b, row_config),]


 M_X_split=0

 if(is.null(nrow(w))){

  fill=rep(0,length(val_intended_for))


  M_X_split=(w[c(ncol(a))])
  con=w[c(1)]
  where_in_total=match(val_intended_for,con)
  where_in_total=which(where_in_total>0)

  fill[where_in_total]=M_X_split

  M_X_split=fill
```

```
168
169
170   }else{
171
172
173    M_X_split=(w[,c(ncol(a))])
174    con=w[,c(1)]
175
176
177    fill=rep(0,length(val_intended_for))
178
179    where_in_total=match(val_intended_for,con)
180
181    where_in_total=which(where_in_total>0)
182    fill[where_in_total]=M_X_split
183
184    M_X_split=fill
185
186
187   }
188
189
190
191   return(M_X_split)
192  }
193
194
195
196
197
198  #indicator for root selection
199  indicator=0
200  # n is set to 0
201  n=0
202
203
204
205
206  #CSI-log-marginal-likelihood is set to 0
207  sco=0
208
209
210
211
212  #list of theoretical values of node
213  val_intended_for=p[[intended_for]]
214  #how many values in val_intended_for
215  len_val_intended_for=length(val_intended_for)
216
217  #while TRUE
218  while (n<1 ) {
219
220
221
222   #if indicator is 0
223   if(indicator==0){
224
225    #sort element of parentset
226    elements=parent_set
227
228    #Renaming of len_val_intended_for
229    no_split_uniq=len_val_intended_for
230
231    #count how many entries of value of node exist in dataset
232    M_X_no_split=table(data[,.SD,.SDcols=c(intended_for)])
233
234    # The specific values of node that exist in dataset
235    con=as.numeric(names(M_X_no_split))
236    #Create zero vector for storing counts of occurrences of value of node
237    fill=rep(0,length(val_intended_for))
238
239    #this is done to have fixed length on count vector no matter how many counts exist for
          node
240
241
242
243
244
245
246
247
248
249    #match with theoretical values of node
250    where_in_total=match(val_intended_for,con)
251    #remove NA from where_in_total
252    where_in_total=where_in_total[!is.na(where_in_total)]
253    #put counts in zero matrix
254    fill[where_in_total]=as.numeric(M_X_no_split)
255    #rename zero vector
256    M_X_no_split=fill
257
258
259
260    #calculate BDEu prior when no parent is included
261
262    alpha_no_split= N/(no_split_uniq)
263    alpha_no_split_vec=rep(alpha_no_split,len_val_intended_for)
264
265    aplha_sum_no_split=sum(alpha_no_split_vec)
266
267
268    M_sum_no_split= sum(M_X_no_split)
269    #calculate log marginal likelihood for when node has no parent
```

```
270     src_no_split=lgamma(aplha_sum_no_split)-lgamma(aplha_sum_no_split+M_sum_no_split)+sum(
              lgamma(alpha_no_split_vec+M_X_no_split)-lgamma(alpha_no_split_vec))
271
272     #if input parentset is empty break whileloop return CSI-score
273     if(is.null(parent_set)){
274      sco=src_no_split
275
276      break
277     }
278
279
280
281
282     #Set comparing value to 0
283     split_t=0
284     #Set CSI-score to 0
285     scores=0
286
287     #for each element(parent node) in parentset of node
288     for(element in (elements)){
289
290      # list of columnnames for node=intended_for and element in dataset
291      cols=names(data[,.SD,.SDcols=c(intended_for,c(element))])
292
293
294
295
296
297      #Compute how many theoretical values does element have
298      parent_set_entries_to_alpha=p[c(element)]
299
300      #calculate number of unique values for element
301      in_between_move=lapply(1:length(parent_set_entries_to_alpha),function(x){length(parent
               _set_entries_to_alpha[[x]])})
302      nr_uniq_par_value_to_alpha=Reduce('*',in_between_move)
303
304
305
306
307
308      # unique values of element in dataset
309      split_uniq_par_val=unname(as.matrix(unique(data[,.SD,.SDcols=c(element)])))
310      #number of unique values of element in dataset
311      split_uniq_parent=nrow( split_uniq_par_val)
312      con_2=split_uniq_par_val
313
314      #initialize empty score vector
315      src_split=rep(0,split_uniq_parent)
316      #count how many instances of different configurations for node intended_for and
               element that exist in dataset
317      c=unname(as.matrix(M_xi_parent_count(data,cols)))
318
319
320      #for each value of element
321      for(row in 1:nrow(split_uniq_par_val)){
322       #element value held fixed vary values of nodes,put each count into a vector M_X_split
323       M_X_split=indicator_function_2(row,split_uniq_par_val,c)
324
325       #if at least one configuration exist
326       if (sum(M_X_split)!=0){
327
328
329        alpha_split=N/(len_val_intended_for*nr_uniq_par_value_to_alpha)
330        alpha_split_vec=rep(alpha_split,len_val_intended_for)
331
332        alpha_sum_split=sum(alpha_split_vec)
333        M_sum_split=sum(M_X_split)
334        #calculate log marginal likelihood for node when element's value is held fixed
335        src_split[row]=lgamma(alpha_sum_split)-lgamma(alpha_sum_split+M_sum_split)+sum(
                lgamma(alpha_split_vec+M_X_split)-lgamma(alpha_split_vec))
336
337       }
338
339
340
341
342     }
343
344     #if at least one log marginal likelihood exist
345     if(sum(src_split)!=0){
346      #if the sum of log-marginal likelihood is greater then split_t
347      if((sum(src_split)-sum(src_no_split))>split_t ){
348       #save element
349       choice=element
350
351       #Which value of element has  log-marginal likelihood 0
352       w_zeo=which(src_split==0)
353       if(length(w_zeo)!=0){
354        #exclude this score
355        scores=src_split[-c(w_zeo)]
356        #Exclude the value related to that score
357        con_3=c(con_2[-c(w_zeo),])
358       }else{
359
360        #else all values has a score
361        scores=src_split
362
363        con_3=c(con_2)
364       }
365       #set positive difference to be new comparing value
```

```
366       split_t=(sum(src_split)-sum(src_no_split))
367
368      }
369     }
370    }
371
372
373
374
375
376    #if all element and there values have been looked at and score still is zero no root is
           found
377    if(sum(scores)==0){
378
379     sco=src_no_split
380
381     break
382
383    }else{
384
385     #Else root is found define tree
386
387
388     #chosen elements value in data
389     choise_uniq=con_3
390     #number of values of element in data
391     choise_unique_nr=length(choise_uniq)
392
393     #matrix containing score for every branch(value)
394     mat=matrix(c(scores,choise_uniq),ncol = 2)
395     #element  corresponding value in mat
396     al_matrix=matrix(rep(choice,choise_unique_nr),ncol = 1)
397
398     #set indicator to 1
399     indicator=1
400
401    }
402   }else{
403
404
405    #continue building tree in the same way
406    scores=0
407    split_t=0
408    #for element in parentset
409    for(element in elements){
410
411     #for every row in mat
412     for(row in 1:nrow(mat)){
413
414
415
416
417      #go through every row in al_matrix
418      parent_elements_row=al_matrix[row,]
419
420      parent_elements_row= (parent_elements_row[!is.na( parent_elements_row)])
421
422      #add element in branch(row)
423      cols=names(data[,.SD,.SDcols=c(intended_for,parent_elements_row,element)])
424
425      #if element is not in row of al_matrix continue
426      '%!in%' <- Negate('%in%')
427      if(element%!in%parent_elements_row){
428
429       #do the same procedure as described above to calculate log-marginal-likelihood of
             branches adding element to each row in al_matrix on every branch(row)
430       #that does not contain element(parent node) look for which element added to which
             branch gives the greatest improvement.Grow mat(tree ) with the values of that
             element
431
432       parent_set_entries_to_alpha=p[c(parent_elements_row,element)]
433
434
435       in_between_move=lapply(1:length(parent_set_entries_to_alpha),function(x){length(
              parent_set_entries_to_alpha[[x]])})
436       nr_uniq_par_value_to_alpha=Reduce('*',in_between_move)
437
438
439       split_uniq_parent_config=nrow(unique(data[,.SD,.SDcols=c(parent_elements_row,element
              )]))
440
441       c=unname(as.matrix(M_xi_parent_count(data,cols)))
442
443
444       next_el=unname(as.matrix(unique(data[,.SD,.SDcols=element])))
445       con_2=next_el
446
447       s_vec=rep(0,nrow(next_el))
448
449
450       for(s in 1:length(s_vec)){
451
452        M_X_split=indicator_function_3(row,mat,c,next_el[s])
453
454
455
456        if(sum(M_X_split)!=0){
457
458
459         M_sum_split=sum(M_X_split)
460
461         alpha_split= N/(len_val_intended_for* nr_uniq_par_value_to_alpha)
```

```
462          alpha_split_vec=rep(alpha_split,len_val_intended_for)
463
464          alpha_sum_split=sum(alpha_split_vec)
465          s_vec[s]=lgamma(alpha_sum_split)-lgamma(alpha_sum_split+M_sum_split)+sum(lgamma(
                 alpha_split_vec+M_X_split)-lgamma(alpha_split_vec))
466
467         }
468
469
470        }
471
472
473        if(sum(s_vec)!=0){
474         src_no_split=mat[row,1]
475
476         diff=sum(s_vec)-src_no_split
477
478
479
480
481         if(diff>split_t){
482          w_zeo=which(s_vec==0)
483          if(length(w_zeo)!=0){
484           scores=s_vec[-c(w_zeo)]
485
486           con_3=c(con_2[-c(w_zeo),])
487          }else{
488           scores=s_vec
489
490           con_3=c(con_2)
491          }
492
493          which_element_choosen=element
494
495          which_row_branch=row
496          #update split_t the same way as before
497          split_t=diff
498
499
500         }
501        }
502
503       }
504
505      }
506     }
507
508
509     #if no element added to each branches give any improvement break
510     if(all(scores==0)){
511
512      break
513     }else{
514
515      #else add the element to the branch that gave the best improvement
516
517      #values of chosen element
518      choosen_element_uniq_val=con_3
519
520      #number of values of element in dataset
521      choosen_element_uniq_val_nr=length(choosen_element_uniq_val)
522
523
524
525
526      #if more then one value exist for element in dataset, expand  the rows with the number
             of values of element-1
527      if(length(con_3)>1){
528       v=rep(1,length(mat[,(ncol(mat))]))
529       v[which_row_branch]=choosen_element_uniq_val_nr
530
531
532       mat=(mat[rep(1:nrow(mat), times = v),])
533
534
535       al_matrix=al_matrix[rep(1:nrow(al_matrix), times = v),]
536
537      }
538
539      #else only one value exist for element in dataset and expand only the columns
540
541      #new configurations to add to mat
542      new_node_value=matrix(rep(NA,nrow(mat)),ncol=1)
543
544      #put values of element into new_node_value
545      new_node_value[which_row_branch:(which_row_branch+choosen_element_uniq_val_nr-1),]=
             choosen_element_uniq_val
546
547
548
549      #new configurations to add to al_matrix
550      new_node_to_al=matrix(rep(NA,nrow(mat)),ncol=1)
551
552
553      #put in chosen element into al_matrix
554      new_node_to_al[which_row_branch:(which_row_branch+choosen_element_uniq_val_nr-1),]=
             which_element_choosen
555
556      #concatinate these vectors with mat and al_matrix
557      mat=cbind(mat,new_node_value)
558      al_matrix=cbind(al_matrix,new_node_to_al)
559
```

```
560
561     #change scores of mat
562     mat[which_row_branch:(which_row_branch+choosen_element_uniq_val_nr-1),1]=scores
563
564
565
566
567    }
568   }
569  }
570
571  # If while-loop stopped after root was added to tree redefine sco from 0 to the sum of
         the branch scores in mat
572  if(sco==0){
573   sco=sum(as.numeric(mat[,1]))
574  }
575
576  #return log-marginal likelihood
577  return(sco)
578 }
```

A.4 Function for computing and writing marginal-likelihoods multiple parentsets

```
1
2
3  #       ##########################################################################################################

4  #Function:func
5  #       ##########################################################################################################

6  #Input:
7  #data:data used.
8  #k:parentsize.
9  #node:which node to calculate log-marginal likelihood from.
10
11 #parent_comb:Specific parent combination
12
13 #fid:path of scorefile.
14
15 #set_of_intrest:contains all node in the network .
16
17 #p:contains a list of lists where each list contains the values of a node in the network
18
19
20
21
22 #Output:Writes score for all combinations of set_of_intrest for all parentsets for
       specific parentsize.This function will be put into
23 #function below csitree_calc_parent_score_to_file_3.
24
25
26
27 #       ##########################################################################################################

28 #import CSI algorithm
29 source("csi_tree_imp_2.R")
30 func=function(data,node,k,fid,set_of_intrest,p){
31
32
33  #calculate log-marginal likelihood for empty parent
34  if(k==0){
35   N=1
36
37   writeLines(paste(node, nps), con = fid, sep = "\n")
38
39
40   uniq_Xi_value=matrix(p[[node]])
41
42
43   nr_uniq_Xi_value1=nrow( uniq_Xi_value)
44
45
46   alpha_node_parnode1=N/(nr_uniq_Xi_value1)
47   alpha_node_parnode1=rep(alpha_node_parnode1,nr_uniq_Xi_value1)
48   alpha_sum_parnode1=sum( alpha_node_parnode1)
49   M_X_i=table(data[,..node])
50
51   con=as.numeric(names(M_X_i))
52   fill=rep(0,nrow(uniq_Xi_value))
53
54
55
56   where_in_total=match( uniq_Xi_value,con)
57   where_in_total=which(where_in_total>0)
58
59   fill[where_in_total]=as.numeric(M_X_i)
60
61   M_X_i=fill
62
63   M_sum_count=sum(M_X_i)
```

```
64
65
66    src=(lgamma(alpha_sum_parnode1)-lgamma(alpha_sum_parnode1+M_sum_count)+sum(lgamma(alpha_
          node_parnode1+M_X_i)-lgamma(alpha_node_parnode1)))
67
68    writeLines(paste(trimws(format(round(src, 6), nsmall=6)),k, sep = " "), con = fid, sep =
          "\n")
69    src=0
70   }else{
71
72    #Else for all parentsets of node with cardinality k
73    parent_set=setdiff(set_of_intrest,node)
74    parent_comb=combn(parent_set,k)
75
76    #calculate all CSI-log-marginal-likelihoods
77
78    lapply(1:ncol(parent_comb),function(x){ writeLines(paste(trimws(format(round(CSI_tree_
          apply_imp_3_mat_B_3(data,c(parent_comb[,x]),node,p),6), nsmall=6)),k,paste(parent_
          comb[,x],collapse = " "), sep = " "), con = fid, sep = "\n")})
79
80
81   }
82  }
83
84
85
86  #csitree_calc_parent_score_to_file_2 is simlar to csitree_calc_parent_score_to_file_3
87
88
89  #
        ##########################################################################################################################
90  #Function:csitree_calc_parent_score_to_file_3
91  #
        ##########################################################################################################################
92  #Input:
93  #data:data used.
94  #score_type: type of score.
95  #file_out:Name of score file
96
97  #max_parent_size:bound on parentsize
98
99
100 #p:contains a list of lists where each list contains the values of a node in the network
101
102
103
104
105 #Output:Writes score for all combinations of set_of_intrest for all parentsets for all
        parentsizes up to the bound max_parent_size.
106
107
108
109 #
        ##########################################################################################################################
110
111
112 csitree_calc_parent_score_to_file_3 <- function(data, score_type, max_parent_size, file_
        out,p){
113
114  #N=1
115
116
117  #number of nodes
118  numcol= ncol(data)
119  #list of nodes
120  set_of_intrest=1:numcol
121
122
123
124
125  #calculate and write to file  scores of all parent combinations of all parentsets for all
         nodes for all parent cardinalities smaller or euqal to max_parent_size
126  fid <- file(paste0("C:/Users/rasyd/Documents/gitrepo/master/score_folder/scores/csi_
       survey/n5000/",file_out, ".", score_type, ".score", sep = ""),"wt")
127
128  writeLines(toString(numcol), con = fid, sep = "\n")
129
130  #call on func and iterate over all nodes in network and all parent sizes
131
132  apply(matrix(set_of_intrest,1,length(set_of_intrest)),2,function(x)apply(matrix(0:max_
       parent_size,1,(max_parent_size+1)),2,function(y) func(data,x,y,fid,set_of_intrest,p
       )))
133
134  #close file
135  close(fid)
136
137
138
139 }
```

A.5 MCMC algorithm

```
#
     ##########################################################################################################
#Function:run_func_2
#
     ##########################################################################################################
#Input:

#read_this:path of scorefile
#max_parent_size:how many scorefiles should be made.

#nr_nodes:how many nodes in network.

#true_matrix_2:some transformation of the adjacency matrix of the network

#j:iterator index.


#ns_i:data sample size.
#khh:which score type, CSI or CPT

#Output:AUC


#This runs MCMC over a scorefile
#
     ##########################################################################################################



func_MCMC=function(read_data,true_matrix_2,max_parent_size,j,nr_nodes,ns_i,khh){
 #change .score file to .txt file
 func_score=function(read_data){
  library("MASS")
  add=paste0("Score_csi_",toString(j),".txt")

  write.matrix(read_data,sep=" , ",file=add)


  score_csi_2=read.csv(file = add, header = FALSE,sep=" " )

  return(score_csi_2)
 }

 #run func_score on score-file
 score_csi_2=(func_score(read_data))
 # remove 2 first rows containing only NA values
 score_csi_2=score_csi_2[-c(1,2),]
 #list of nodes
 nodes=1:nr_nodes
 #index of nodes
 integer=(as.numeric(score_csi_2[,1]) - abs(floor(as.numeric(score_csi_2[,1])))) == 0


 integer_true=which(integer==TRUE)

 #index of first occurrence of parentset with specific parentset cardinality.These indexes
       are the same for all nodes in list nodes
 seq_3=rep(0,max_parent_size)
 for(i in 1:max_parent_size){
  seq_3[i]=choose(nr_nodes-1,i)
 }

 #Function for finding score with empty parentset
 integer_zero=function(score_csi_2,int,node){

  index=int[node]+1
  return(as.numeric(score_csi_2[index,1]))
 }



 #Function for finding score of  parentset of a specific node
 integer_map=function(score_csi_2,input_row,integer,nodes,seq_3,nr_nodes){
  #node is in first place in input_row
  node=input_row[1]

  #parentset cardinality of parentset of node is in third place in input_row
  nr_parent=input_row[3]

  #set_1 is a set of node from 1 to first parent in parentset excluding node
  set_1=setdiff(1:input_row[4],node)
  #set_3 is a set of node from 1 to first parent in parentset
  set_3=1:input_row[4]
  #set_2 is a set of node from  first parent to last parent excluding node
  set_2=setdiff(input_row[4]:nodes[length(nodes)],node)


  #step will be used as the number of indexes that has to be jumped over
  step=0

  #Count for while loop
  n=1
  #count will be used to denote how many times one is in the for loop after n>=2
```

```
91  count=1
92  #number of parents excluding last parent
93  node_1_len=nr_parent-1
94  #number of nodes excluding node
95  nr_nodes_1=nr_nodes-1
96  #while true
97  while(n!=0){
98
99   #if n=number of parents
100  if(n==length(input_row[4:(length(input_row))])){
101
102   break
103  }
104
105  #if n is 1
106  if(n==1){
107   #if node is between 1 to first parent
108   if(node%in%set_3){
109    #use set_1
110    set_4=set_1
111
112
113   }else{
114    #else use set_3
115    set_4=set_3
116   }
117  }else{
118
119   # for n>1 use set_2
120   set_4=set_2
121
122  }
123  #if n>=2 reduce set_4 by indexes 1 to how many times in forloop after n>=2
124  if(n>=2){
125
126   set_4=set_4[-c(1:(count))]
127  }
128
129
130  #count for how many times in forloop
131  count_4=0
132
133  #for j from 1 to length of set_4
134  for(j in 1:(length(set_4))){
135   #if n>=2 find number of elements to exclude from first element in constant set set_2
136   if(n>=2){
137    count=count+1
138   }
139
140   #number of times in forloop
141   count_4=count_4+1
142
143   #if element j in set_4 equal parentset element n-1 in input_row
144   if(set_4[j]==input_row[(4+n-1)]){
145
146    #subtract number of parent with how many times inside forloop before the if statement
          is activated denoted by count_4
147    nr_nodes_1= nr_nodes_1-count_4
148    #break forloop
149    break
150   }
151   #Add how many rows to jump over in scorefile
152   step=step+choose((nr_nodes_1-j),node_1_len)
153
154
155  }
156  #subtract element from number of elements
157  node_1_len=node_1_len-1
158
159  # update n
160  n=n+1
161
162 }
163
164 #find last parent in parentset.If parentset contains one element then the whileloop will
      not be used  else last
165 #parent will be between next to last parent+1 and nr_nodes
166 if(nr_parent==1){
167
168  #define seq_4
169  seq_4=setdiff((1:nr_nodes),node)
170
171 }else{
172  #else we have to find the last element based on next to last element
173  what=input_row[(length(input_row)-1)]
174  seq_4=setdiff((what+1):nr_nodes,node)
175
176 }
177 #find index of last parent in parentset
178 add=which(seq_4==input_row[length(input_row)])
179 #add to step
180 step=step+add-1
181
182 #calculate index of node parent combination
183 extract_row=integer[node]+sum(seq_3[1:(nr_parent-1)])+step+2
184 if(nr_parent==1){
185  extract_row=integer[node]+step+2
186 }
187
```

```
188   #return score of node parent combination
189   return(as.numeric(score_csi_2[extract_row,1]))
190  }
191
192
193
194
195
196  #Function for calculating neighbourhood of a DAG
197  func_nabour_imp_large_B_2=function(adj){
198
199   #index how many potential adds
200   index_add=which(adj==0,arr.ind = TRUE)
201   # how many 1's in every column
202   nr_of_par_each=apply(adj,2,sum)
203
204   #remove edge from node to itself
205   index_add=index_add[index_add[,1] !=index_add[,2],]
206
207   #number of deletes
208   index_delete_rev_1=which(adj==1,arr.ind = TRUE)
209
210   #number potential reverse
211   index_delete_rev_2=which(adj==1,arr.ind = TRUE)
212
213   #if any column has more then max_parent_size 1's
214   if(any(nr_of_par_each>=(max_parent_size))){
215    #find which column
216    which_add_greater_then=which(nr_of_par_each>=(max_parent_size))
217    #match these elements with column vector in index_add matrix
218    match_val=match(index_add[,2],which_add_greater_then)
219    #remove NA
220    match_val=which(match_val>0)
221    #delete indexes corresponding to element in second column in index_add being equal to
          match_val
222    if(length(match_val)!=0){
223     index_add=index_add[-match_val,]
224    }
225   }
226
227
228   #If potential reverse index matrix is none empty
229   if(length(index_delete_rev_2 )!=0){
230
231    # column 1 represents rows in adjecency matrix.If a reverse happens these elements will
          represent columns in the adjecency matrix
232    current_switch=index_delete_rev_2[,1]
233    # sort first column in reverse index matrix
234    which_true=sort(unique(current_switch))
235
236    #how many 1's exist in the columns in the adjecency matrix that will get an extra 1 by
          reversing
237    larger_then_max=apply(matrix(adj[,c(which_true)],ncol=length(which_true)),2,function(x)
          sum(x))
238
239
240
241
242
243    #if any of these columns already have more then max_parent_size 1's
244    if(any( larger_then_max>=(max_parent_size))){
245     #which index in which_true has this characteristic
246     which_reverse_greater_then=which(larger_then_max>=max_parent_size)
247     # find which column in which_true
248     which_true= which_true[which_reverse_greater_then]
249     what_to=match(1:nr_nodes,which_true)
250     mmacth=which(what_to>0)
251
252     match_val=match(index_delete_rev_2[,1],mmacth )
253
254     match_val=which(match_val>0)
255
256
257     #exclude the adjacency matrix indexes contained in first column in index_delete_rev_2
           that matches match_val
258     if(length(match_val)!=0){
259      index_delete_rev_2=index_delete_rev_2[-match_val,]
260     }
261
262    }
263   }
264
265   #Set bounds for number of add,delete and reverse
266
267   if(length(index_add)==0){
268    index_add=c(1,2)
269
270    u=0
271   }else{
272    u=nrow(index_add)
273   }
274   if(length(index_delete_rev_1)==0){
275    index_delete_rev_1=c(1,2)
276    l=0
277   }else{
278    l=nrow(index_delete_rev_1)
279   }
280
281
```

```
282  if(length(index_delete_rev_2)==0){
283   index_delete_rev_2=c(1,2)
284   pp=0
285  }else{
286   pp=nrow(index_delete_rev_2)
287  }
288
289  #define a zero matrix
290  zero_matrix=array(0,c(nrow(adj),ncol(adj)))
291  #Define a two dimentional storing matrix
292  B_matrix=array(0,c(nrow(adj)*(u+l+pp),ncol=ncol(adj)))
293
294  #every k:m row is a matrix in B_matrix
295  k=1
296  m=ncol(adj)
297
298  #for j=1 add, j=2 delete and j=3 reverse
299  for(j in 1:3){
300   if(j==1){
301    #set bound for number add checks
302    f=u
303   }
304
305   if(j==2){
306    #set bound for number delete checks
307    f=l
308   }
309   if(j==3){
310
311    #set bound for number reverse checks
312    f=pp
313   }
314   for(i in 1:f){
315
316    #if add
317    if(j==1){
318
319     if(is.null(nrow(index_add))){
320
321      next
322     }
323
324     #set 1 on index index_add[i,] in adj(adjacency matrix)
325     zero_matrix=adj
326     zero_matrix[array(index_add[i,],c(1,2))]=1
327
328
329    }
330
331    #if delete
332    if(j==2){
333     if(is.null(nrow(index_delete_rev_1))){
334
335      next
336     }
337     #set 0 on index index_add[i,] in adj(adjacency matrix)
338     zero_matrix=adj
339     zero_matrix[array(index_delete_rev_1[i,],c(1,2))]=0
340
341
342
343    }
344    #if reverse
345    if(j==3){
346     if(is.null(nrow(index_delete_rev_2))){
347
348      next
349     }
350     #set reverse index_add[i,] in adj(adjacency matrix)
351     zero_matrix=adj
352     rev=array(index_delete_rev_2[i,],c(1,2))
353     zero_matrix=reverse_oper_adjacent(zero_matrix,rev[1],rev[2])
354
355    }
356    #check whether the change made adj cyclic
357    ind=is_dag(zero_matrix)
358
359    #if not
360    if(ind==TRUE){
361
362     #if add
363     if(j==1){
364
365      #add to B_matrix
366      B_matrix[k:m,1:ncol(adj)]=zero_matrix
367      #change k and m
368      k=k+ncol(adj)
369      m=k+ncol(adj)-1
370     }
371     #if delete
372     if(j==2){
373      #add to B_matrix
374      B_matrix[k:m,1:ncol(adj)]=zero_matrix
375
376
377      #change k and m
378      k=k+ncol(adj)
379      m=k+ncol(adj)-1
380
```

```
381
382
383       }
384
385       #if reverse
386       if(j==3){
387        #add to B_matrix
388        B_matrix[k:m,1:ncol(adj)]=zero_matrix
389
390        #change k and m
391        k=k+ncol(adj)
392        m=k+ncol(adj)-1
393
394
395
396       }
397
398      }
399
400
401     }
402
403
404    }
405
406
407    #remove zero rows in last portion of B_matrix
408    B_matrix=B_matrix[1:(k-1),]
409
410
411    return(B_matrix)
412   }
413
414
415
416
417   #Function for reversing edge of adj
418   reverse_oper_adjacent=function(adj,i,node){
419    #make copy of adj
420    adj_copy=adj
421    #set value of  matrix on index node,i to value of matrix on index i,node
422    adj[node,i]= adj[i,node]
423
424    #set value of  matrix on index i,node to value of matrix on index node,i
425    adj[i,node]=adj_copy[node,i]
426
427    return(adj)
428   }
429
430
431
432   #Function for indexing B_matrix in neighbourhood function into matrices of adj size
433   sec_func=function(stack_matrix,adj_row){
434    k=1
435    m=adj_row
436    nr_matrix=nrow(stack_matrix)/adj_row
437
438    seq_mat=array(0,c(nr_matrix,2))
439    seq_mat[1,]=c(k,m)
440    if(nr_matrix>1){
441     for(i in 2:nr_matrix){
442
443      k=k+adj_row
444      m=k+adj_row-1
445      seq_mat[i,1]=k
446
447      seq_mat[i,2]=m
448     }
449    }
450
451    return(seq_mat)
452   }
453
454
455   #Function returning columns where adj_1 is different from adj_2 together with the columns
         of both adj_1 and adj_2 where they differ
456   diff_adj=function(adj_1,adj_2){
457
458
459
460
461    ind=which(adj_1!=adj_2,arr.ind = TRUE)
462
463
464    return(list(a=ind[,2],b=adj_1[,ind[,2]],c=adj_2[,ind[,2]]))
465   }
466
467   #Function for converting node,parentsize, parentset represented by 0 1 vector into
        parentset represented with natural numbers and finding the score for this vector
468   from_adjecency_row_2=function(input_diff1,input_diff2){
469
470    matr_desing_node=input_diff1
471
472    nr_parent=sum(input_diff2)
473
474
475    parent=which(input_diff2==1)
476
477
478
479    if(nr_parent>0){
```

```
480     score=integer_map(score_csi_2,c( matr_desing_node,123,nr_parent,parent),integer_true,
            nodes,seq_3,nr_nodes)
481   }else{
482    score=integer_zero(score_csi_2,integer_true,matr_desing_node)
483   }
484
485
486   return(score)
487
488  }
489
490
491
492
493
494  #Function for converting adjacency matrix  into a set of (node,parentsize,parentset)
         represented with
495  #natural numbers in order to find the score for the adjacency matrix.
496
497  from_adjecency_to_matrx=function(adj){
498
499
500   matr_desing=matrix(0,ncol(adj),ncol(adj)+3)
501
502   matr_desing[,1]=1:ncol(adj)
503   matr_desing[,2]=1
504   matr_desing[,3]=apply(adj,2,sum)
505
506   v=apply(adj,2,function(x) which(x==1))
507
508
509   for(j in 1:nrow(matr_desing)){
510    if(matr_desing[j,3]==0){
511     next
512    }
513
514    parent=v[j][[1]]
515
516    len_par=length(parent)
517
518    matr_desing[j,4:(4+len_par-1)]=sort(parent)
519   }
520
521
522   for(i in 1:nrow(matr_desing)){
523    if(matr_desing[i,3]==0){
524
525     matr_desing[i,2]= integer_zero(score_csi_2,integer_true,matr_desing[i,1])
526
527    }else{
528
529
530     none_zero=match(0,matr_desing[i,])-1
531
532
533
534
535     matr_desing[i,2]= integer_map(score_csi_2,matr_desing[i,1:none_zero],integer_true,
            nodes,seq_3,nr_nodes)
536
537
538    }
539   }
540
541   return(matr_desing[,2])
542
543  }
544
545
546  #number of iterations
547  N=600000
548  #thinning
549  lag=10
550  #how many MCMC chains
551  m=1
552  #start collecting samples after burn in
553  start=550000
554
555  #stop collecting samples
556  stop=600000
557
558  #sample collection sequence
559  save_every=seq(start,stop,lag)
560
561  #set initial DAG for chains
562  inital_value_adj=matrix(0,m*nr_nodes,nr_nodes)
563
564  #Slice inital_value_adj depending on m
565  slice_intial=sec_func(inital_value_adj,nr_nodes)
566
567  #calculate initial score
568  inital_value_scores=matrix(0,nr_nodes,m)
569  for(i in 1:dim(slice_intial)[1]){
570
571
572   inital_value_scores[,i]=(from_adjecency_to_matrx(inital_value_adj[slice_intial[i,1]:
         slice_intial[i,2],]))
573
574  }
575
576
577  inital_value_score=colSums(inital_value_scores)
```

```
578
579 #define storage matrix for traversal of chain in score space
580 X_t_matrix=matrix(0,m,N)
581 #put initial score as first column in storage matrix X_t_matrix
582 X_t_matrix[,1]=inital_value_score
583
584
585 #matrix for saving samples
586 save_array=array(0,c(nr_nodes*(length(save_every)),nr_nodes,m))
587
588 #slice save matrix
589 slice_save_array=sec_func(save_array,nr_nodes)
590
591 #iterator for slice_save_array
592 acc_vec=rep(0,m)
593
594
595 w=0
596
597 #Save currently accepted matrix in matr_current
598 matr_current=array(0,c(m*nr_nodes,nr_nodes))
599 #Save currently proposed matrix in matr_prop
600 matr_prop=array(0,c(m*nr_nodes,nr_nodes))
601
602 start=Sys.time()
603
604 #for 2 to N
605 for(i in 2:(N)){
606  #for 1 to m chains
607  for(k in 1:m ){
608
609   # set initial DAG to matr_current for chain k and calculate neighbourhood of inital DAG
610   if(w==0){
611    matr_current[slice_intial[k,1]:slice_intial[k,2],]=inital_value_adj[slice_intial[k,1]:
          slice_intial[k,2],]
612    neighbour_candate=func_nabour_imp_large_B_2(matr_current[slice_intial[k,1]:slice_
          intial[k,2],])
613
614   }
615
616   #slice neighbour_candidate
617   neighboors_cand_slice=sec_func(neighbour_candate,nr_nodes)
618   #number neighbours in neighbour_candidate
619   nr_of_neighboors=nrow(neighboors_cand_slice)
620
621   #sample uniform one DAG from neighbour_candidate
622   matr_prop_nr=sample.int(nrow(neighboors_cand_slice),1)
623
624   #put it equal to matr_prop
625   matr_prop[slice_intial[k,1]:slice_intial[k,2],]=neighbour_candate[neighboors_cand_slice
        [matr_prop_nr,1]:neighboors_cand_slice[matr_prop_nr,2],]
626
627   #Collect matr_current at state i
628   if(i%in% save_every){
629
630    #go to next index in slice_save_array
631    acc_vec[k] = acc_vec[k]+1
632    # save matr_current in state i in position acc_vec[k],1]:slice_save_array[acc_vec[k
          ],2],,k in save_array
633    save_array[slice_save_array[acc_vec[k],1]:slice_save_array[acc_vec[k],2],,k]=matr_
          current[slice_intial[k,1]:slice_intial[k,2],]
634
635   }
636
637   #calculate neighbourhood of current state of prop_matr
638   prop_neighbor=func_nabour_imp_large_B_2(matr_prop[slice_intial[k,1]:slice_intial[k
        ,2],])
639
640   #number of neighbours in prop_neighbor
641   prop_neighbor_nr=nrow(sec_func(prop_neighbor,nr_nodes))
642
643   #find difference between current state of matr_current and current matr_prop
644   diff_c_p=diff_adj(matr_current[slice_intial[k,1]:slice_intial[k,2],],matr_prop[slice_
        intial[k,1]:slice_intial[k,2],])
645
646   #If one column is returned it means the difference is an add or delete
647   if(length(unlist(diff_c_p$a))==1){
648
649    #Define which  column differ
650    extract_1=as.numeric(diff_c_p$a)
651    #column in matr_current
652    extract_2=diff_c_p$b
653    #column in matr_prop
654    extract_3=diff_c_p$c
655    #score of column in matr_current
656    what_to_delete=from_adjecency_row_2(extract_1,extract_2)
657    #score of this column in matr_prop
658    what_to_add=from_adjecency_row_2(extract_1,extract_3)
659
660    #compute score of matr_prop by subtracting score of column in matr_current and adding
          score of this column in matr_prop
661    prop_score=X_t_matrix[k,(i-1)]+what_to_add-what_to_delete }else{
662
663    #else the difference is an reverse if the neigbourhood function is correct.
664
665    #Define which  column differ
666    extract_1=t(diff_c_p$a)
667
668    #column in matr_current
```

```
669     extract_5=diff_c_p$b
670     #column in matr_prop
671     extract_6=diff_c_p$c
672
673     #score of column in matr_current
674     what_to_delete=apply(matrix(c(1,2),1,2),2,function(x) from_adjecency_row_2(extract_1[,
            x],extract_5[,x]))
675     #score of this column in matr_prop
676     what_to_add=apply(matrix(c(1,2),1,2),2,function(x) from_adjecency_row_2(extract_1[,x],
            extract_6[,x]))
677
678     #compute score of matr_prop by subtracting score of column in matr_current and adding
            score of this column in matr_prop
679     prop_score=X_t_matrix[k,(i-1)]+sum(what_to_add)-sum(what_to_delete)
680    }
681
682    #calculate acceptance ratio
683    R = min(1,exp((prop_score+log(1/prop_neighbor_nr))-(X_t_matrix[k,(i-1)]+log(1/nr_of_
           neighboors))))
684
685
686
687    #Sample random uniform number and check if its smaller or equal to acceptance rate
688    if(runif(1)<=R)
689    {
690     #if this is the case add matr_prop score to X_t_matrix
691     X_t_matrix[k,(i)]=prop_score
692     #Change matr_current to matr_prop
693     matr_current[slice_intial[k,1]:slice_intial[k,2],]=matr_prop[slice_intial[k,1]:slice_
            intial[k,2],]
694     #calculate neighbourhood of matr_current
695     neighbour_candate=func_nabour_imp_large_B_2(matr_current[slice_intial[k,1]:slice_
            intial[k,2],])
696
697
698
699     #increase w
700     w=w+1
701
702    }else{
703     #else do not change matr_current and set score t equal to score t-1
704     X_t_matrix[k,(i)]=X_t_matrix[k,(i-1)]
705    }
706
707
708
709   }
710
711
712
713  }
714
715
716  #transform the samples matrixes where each element in the matrix represent how many ways
         irrelevant of path length one can go from parent(column) to node(row)
717  gemetric_s_sum=lapply(1:nrow(slice_save_array),function(x)solve(diag(nr_nodes)-save_array
       [slice_save_array[x,1]:slice_save_array[x,2],,1]))
718
719
720  #transform matrices in gemetric_s_sum into 1 0 matrices indicating if there is at least 1
         way to go from parent(column) to node(row).
721  one_path=lapply(1:length(gemetric_s_sum), function(x) gemetric_s_sum[[x]]>=1)
722
723  #take an average of matrices in one_path.
724  edge_matrix_sum=Reduce('+', one_path)
725
726
727  edge_matrix=edge_matrix_sum/length(one_path)
728
729
730  edge_matrix[row(edge_matrix)==col(edge_matrix)]=1
731
732
733  #take an average of the samples.This average shows the estimate for direct causal
         relation between parent(column) to node(row).
734  direct_cause=lapply(1:nrow(slice_save_array),function(x)save_array[slice_save_array[x,1]:
       slice_save_array[x,2],,1])
735  direct_mat=Reduce('+', direct_cause)
736  direct_mat=direct_mat/length(one_path)
737
738
739
740
741
742  #this function returns random string  based on vector vec
743  randstr <- function(vec) {
744   characters=vec[1]
745   numbers=vec[2]
746
747   lowercase=vec[3]
748   uppercase=vec[4]
749   ASCII <- NULL
750
751   if(numbers>0)    ASCII <- c(ASCII, sample(48:57, numbers,replace = TRUE))
752   if(uppercase>0)  ASCII <- c(ASCII, sample(65:90, upperCase,replace = TRUE))
753   if(lowercase>0)  ASCII <- c(ASCII, sample(97:122, lowerCase,replace = TRUE))
754   if(characters>0) ASCII <- c(ASCII, sample(c(65:90, 97:122), characters,replace = TRUE))
755
756   string=rawToChar(as.raw(sample(ASCII, length(ASCII))))
757   return( string )
```

```
758  }
759
760  samp_1=sample(4:6,1,replace = TRUE)
761
762  samp_2=sample(1:samp_1,4,replace = TRUE)
763
764  uniq=randstr(samp_2)
765
766  if(khh==1){
767   keep_adding="cat"}
768  if(khh==2){
769   keep_adding="csi"
770  }
771
772  string_to_add=paste0("C:/Users/rasyd/Documents/gitrepo/master/score_folder/matrix/csi_
       sachs/causal_mechanism/plot2/",toString(j),".txt")
773
774  write.matrix(edge_matrix,sep=" , ",file=string_to_add)
775
776
777
778  samp_1=sample(4:6,1,replace = TRUE)
779
780  samp_2=sample(1:samp_1,4,replace = TRUE)
781
782  uniq=randString(samp_2)
783
784  if(khh==1){
785   keep_adding="cat"}
786  if(khh==2){
787   keep_adding="csi"
788  }
789
790  #string_to_add=paste0("C:/Users/rasyd/Documents/gitrepo/master/score_folder/matrix/cat_
       child/direct_cause/n1001/",keep_adding,toString(ns_i),uniq,toSting(j),".txt")
791  #toSting(j+1)
792  #write.matrix(direct_mat,sep=" , ",file=string_to_add)
793
794
795
796
797
798
799  # Function for sorting estimate probabilities and true ancestral relationship matrix in
       same order and concatinating them
800  class_prob=function(compare){
801
802
803   rechape_compare=c(compare)
804   rechape_compare_order=order(rechape_compare,decreasing = TRUE)
805
806   order_MCMC=rechape_compare[rechape_compare_order]
807
808
809   rechape_true=c(true_matrix_2)
810   order_true=rechape_true[rechape_compare_order]
811
812
813   compare_true_with_mcmc=cbind(order_MCMC,order_true)
814
815
816
817
818   return(compare_true_with_mcmc)
819
820  }
821  r=c("edge_matrix")
822
823
824
825  #calculate AUC
826  AUC_calc=AUC(class_prob(get(r))[,1],class_prob(get(r))[,2])
827
828  #return AUC
829  return(AUC_calc)
830
831 }
```