POLYTECHNIC OF TURIN

Master's Degree in Computer Engineering

Master's Degree Thesis

A Fast Bayesian Artificial Intelligence Reasoning Engine For Modeling And Optimization Tasks

Supervisors

Prof. Alessandro SAVINO

Prof. Stefano DI CARLO

Candidate

Augusto PEDRON

04/2022

Summary

A Bayesian Network is a graphical model used to visually represent connections between variables. The main task performed on a Bayesian Network is the inference of a posterior probability distribution, of one or more nodes, which represents the probability that each variable's state has to occur. This probability distribution is used as a support to take decisions in the real world case modelled by the network. A factor to consider when taking decisions based on a posterior probability distribution is the influence that a parent node has on the node of interest. Based on the strength of influence measured, the user can alter its decision in accordance to the real world case.
Before the network can become an useful tool that can assist the user, its structure and the prior probability distribution of each node has to be defined. Both the structure and the probability distributions can be either defined by hand or learned through an input data set, making the modelling phase of the network easier.
When modelling complex scenarios, the Bayesian Network representing the case of interest can become extremely large and so the inference of each posterior probability distribution can become an expensive and time consuming task, specially if some observations are introduced in the network. With this in mind, our goal was to create a library where the most recent techniques are exploited in order to reduce the complexity of calculations and thus the time and memory required for solving inference in a network of any size, trying to completely exploit all the resources available in modern computers.
We also made possible to learn the structure of a network, with the possibility of adding constraints that have to be respected, and the prior probability distribution of each variable from an input data set and to save it for a later use.
In addition, we provide support for measuring the influence that a parent node has on the node of interest through different metrics.

Acknowledgements

ACKNOWLEDGMENTS

I would like to thank professors Savino, Alessandro and Di Carlo, Stefano, Department of Control and Computer Engineering, Polytechnic of Turin, for their support, guidance and advice that made me produce this thesis, enriching my studies with their ideas. They were always present and willing to clarify any doubts that raised since, at the start of this project, I was not familiar with the subject. Thanks to the opportunity they gave me, I had the possibility to take part in the decision process of a project since I had decision making control over it which is an invaluable experience at this point in my career.
I would also like to thank my family and my friends for their support and encouragement.

Table of Contents

List of Tables

List of Figures

Acronyms

BDs
bayesian-dirichlet sparse

CHC
constrained hill climbing

CPT
conditional probability table

DAG
directed acyclic graph

Lex
lexicographic

MCS
maximum cardinality search

MLE
maximum likelihood estimation

MMHC
max-min hill-climbing

MWCH
minimum weight clique heuristic

PCHC
parents children hill climbing

TP
thread pool

Chapter 1

Introduction

1.1 Overview

Bayesian Networks can be used as a support tool in many complex tasks such as prediction of an event, anomaly detection in manufacture, diagnostics of diseases, troubleshooting, time series prediction, decision making under uncertainty and decision support.
An expert of any of the above mentioned sectors could model a certain problem using a Bayesian Network, defining the relationships between causes and effects. The more accurate the model is, the more precise the prediction given by the network is, potentially replacing the human factor needed during the decision process. In reality the human factor is still very much needed since the operator may have knowledge not available to the network at that moment that should be taken into account when making a decision. Also, a useful information to the operator when making a decision is knowing the influence that each the parent has on the variable of interest, namely knowing which event influences the most the outcome.
Each task has its own requirements in terms of time available to obtain a result by the network, especially in time sensitive tasks as it can be in manufacture, and in terms of computational resources available to dedicate for solving inference, e.g. an office computer used by a doctor while diagnosing a disease to a patient.
Modelling a problem by hand can be a difficult task even for an expert due to the high complexity that the problem can have, not to mention that it can become a tedious task defining the relationships between causes and effects and their prior probability distribution. It is easier to collect data from previous occurrences of the interested events and let an algorithm learn the network by itself, which will define both the structure of the network and the prior probability distributions of the relationships. The expert can influence the learning of the network using

its knowledge of the problem to define which relationships are to be avoided and which must be included in the network.

1.2 Objectives

The topic of this thesis, considering every aspect of dealing with a Bayesian Network described before, is to create a library that allows the user to request any information, from a Bayesian Network, that he needs for his task.
The main objective of the library is to efficiently use all the computational resources available in a computer while reducing both the time and memory required to complete the requested operation. To achieve it, we implemented the latest algorithms developed to efficiently solve the requested operations, introducing our modifications to improve them. These algorithms will be described in detail in the next chapters.

1.3 Outline of the thesis

The discussion of the thesis will be subdivided with the following scheme:
Chapter 2 is dedicated to the explanation of the theory behind the thesis (what is a Bayesian Network and how inference is performed in the network, etc.) and the introduction of other famous libraries for operating with Bayesian networks that are available on the market.
Chapter 3 will give an overview of the library's functionalities, what our library can offer to the user. The details about the implementation of each functionality will be discussed in Chapter 4.
In chapter 5 a detailed comparison between our work and other libraries will be presented, discussing the improvements our library brings. In the end, Chapter 6 will present conclusions on the thesis and future plans for the library.

Chapter 2

Theory and notation

2.1 Graph Theory

The following graph theory is taken from [1]. A graph is defined as a set of nodes (or vertices) and a set of edges, where each edge connects two nodes X_i, X_j. If the edges do not have a direction the resulting graph is called undirected graph; if the edges have a direction they are called arcs and the graph created is a directed graph. A path is sequence of nodes where each node is connected by an edge. A cycle is a path where the start and end node are the same. A simple path is a path where each node appears only once in the path. A simple cycle is a cycle where each node appears only once in the cycle. A directed acyclic graph (DAG) is a directed graph without cycles.
In a directed graph, there exists a parent/child relationship between the two nodes connected by an arc. The parent is also called arc's tail while the child is the arc's head. Two nodes connected by edge (or an arc) are said to be adjacent (or neighbor). The parent/children relationship can be extended into an ancestors/descendants relationship: given two nodes X_i, X_j, where X_i is parent of X_j, then the parents of X_i and so on are the ancestors of X_j. The same rule can be applied to determine the descendants of a node but considering its children. The set of nodes composed of X_i and its parents is called a family.
A DAG in which each node has at most one parent is called forest. A forest where only one node has no parent (the root of the tree) is called tree. Trees, just like graphs, can have directed or undirected edges.
In a tree or graphs, the arcs can be considered pair wise and can be subdivided into 3 categories:

- Head-to-head (or converging connection): is a pair of arcs where both arcs' heads are the same node and the tails are different.
- Tail-to-tail (or diverging connection): is a pair of arcs where both arcs' tails

are the same node and the heads are different.

- Head-to-tail (or serial connection): is a pair of arcs that do not form a cycle and the head of one arc is the tail of the other.

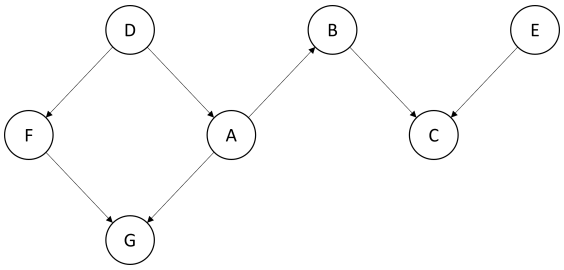

Figure 2.1: Example of DAG. The path D - A - B - C is also a simple path. The pair of arcs ((D,F),(D,A)) form a diverging connection; the pair of arcs ((D,A),(A,B)) form a serial connection; the pair of arcs ((B,C),(E,C)) form a converging connection.

A moral graph is obtained from a DAG by adding an undirected edge between all pair of parents of a node (this process is called marring the parents) and removing the direction of all arcs in the graph. In a cycle of four more nodes, a chord is an edge that connects two non-adjacent nodes and a graph with no cycles of four or more nodes exists is called chordal graph or triangulated graph.
A graph is said to be complete when each node is connected to every other node.
A clique is a complete sub-graph of the original graph.
A junction tree (or clique tree)is a tree composed of cliques obtained from the chordal graph of a DAG. Chordal graphs and junction tree will be discussed in details in Section 2.5.

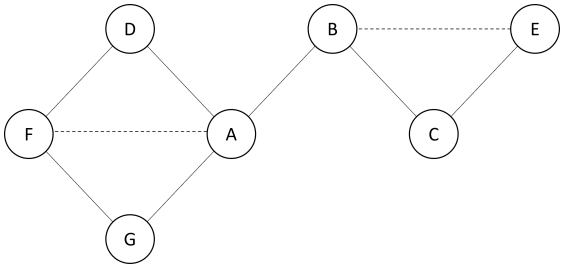

Figure 2.2: Example of chordal graph. The edges (B,E) and (F,A) are added to moralize the graph.

2.2 Graphical Model

One of the tools used to visually represent conditional dependencies among variables of a problem is a Graphical Model [1]. With a Graphical Model is possible to represent the problem by means of nodes and edges, where each edge can have an optional direction connecting the nodes. If the edge has a direction, it is called arc and its direction determines the relationship between the nodes: the source of the arc (arc's tail) is the parent node X_p, while the destination node is the child node (arc's head) X_c. The direction of the arc encodes the cause-effect relationship between nodes, meaning that the event represented by the parent node has its result represented by the child node.

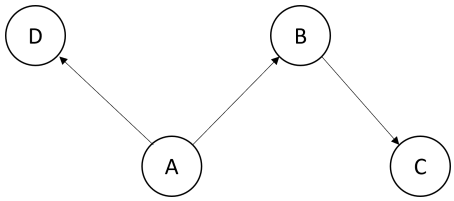

Figure 2.3: Example of graphical model with arcs

Figure 2.3 illustrates the concept of conditional independence. In this case C is conditionally independent of A given B meaning that to calculate the posterior probability distribution of C the formula is P(C|A,B) = P(C|B). Conditional independence does not mean that C is completely independent of A but that its posterior probability is influenced through the nodes, in this case only B, that connect A and C. A Graphical Model can be described as a set of variables (nodes) K = 1,2...n, a set of edges linking the nodes and for each variable a probability distribution function F.

2.3 Bayesian Network

A Bayesian Network [1] is a specific instance of Graphical Model where every edge is directed and no cycle is present in the network, meaning that it is impossible to start a path with a node and end it with the same node. This structure is called DAG (Directed Acyclic Graph) (an example is given in Figure 2.3). This property defines the joint probability distribution of the variables of the network which in the case of Figure 2.3 is

$$P(A, B, C, D) = P(A) \cdot P(B|A) \cdot P(C|B) \cdot P(D|A) \tag{2.1}$$

In general, given a set of nodes K = 1,2...n and their respective probability distribution functions, the joint probability distribution function of the network is:

$$P(K) = \prod_{i=1}^{N} P(K_i|parent(K_i)) \tag{2.2}$$

where the set of parent nodes is defined by the arcs in the network since, as mentioned in Section 2.2, the role of the arcs is to define the cause-effect relationship among variables. In a Bayesian Network, any joint probability distribution follows Bayes' rule meaning that Equation 2.1 does not represent the only way in which the joint of the network can be found.
Using Bayes' rule, we could rearrange the equation, and consequentially change the direction of the arcs in the graph, and get the same joint probability as Equation 2.1. For example:

$$P(A, B, C, D) = P(A) \cdot P(B|A) \cdot P(C|B) \cdot P(D|A) \tag{2.3}$$

$$= \frac{P(A|D) \cdot P(D)}{P(A)} \cdot P(A) \cdot P(B|A) \cdot P(C|B) \tag{2.4}$$

$$= P(D) \cdot P(A|D) \cdot P(B|A) \cdot P(C|B) \tag{2.5}$$

Equation 2.5 is the joint probability distribution of the graph in Figure 2.2

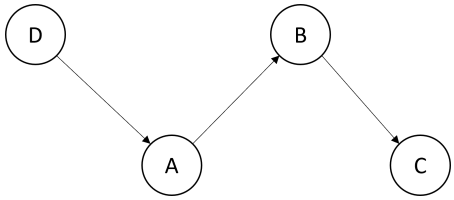

Figure 2.4: Example of graph with the same joint probability as the one in Figure 2.3

2.4 Inference in Bayesian Networks

Once the model is defined, each node's CPT has been filled with values and the observations (or evidences) have been introduced in the network, the first task and most common we want to perform is to find the posterior probability distribution of some nodes of interest in the network. This task is known as inference. There are many ways of performing inference, starting with the simplest Variable Elimination that works well for small networks. But, as size increases, more efficient algorithms are required such as the Lazy Propagation algorithm, which is exploited by our

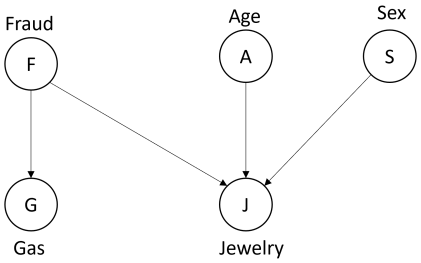

Figure 2.5: Example Bayesian Network from [2] representing credit card fraud.

library. Using the example network in Figure 2.5, taken from [2], we will describe how to calculate any posterior distribution of any node in the network. For example, in order to find the posterior distribution of F we do as follows:

$$P(F|A,G,J,S) = \frac{P(F,A,G,J,S)}{P(A,G,J,S)} = \frac{P(F,A,G,J,S)}{\sum_{F'} P(F',A,G,J,S)} \tag{2.6}$$

This approach is not feasible when dealing with large networks as it leads to unnecessarily big equations. To reduce the computational cost, when all probability distributions are discrete, we can use the conditional independencies induced by the arcs of the network, and in our example we obtain:

$$P(F|A,G,J,S) = \frac{P(F)\cdot P(A)\cdot P(S)\cdot P(G|F)\cdot P(J|F,A,S)}{\sum_{F'} P(F')\cdot P(A)\cdot P(S)\cdot P(G|F')\cdot P(J|F',A,S)} \tag{2.7}$$

$$= \frac{P(F)\cdot P(G|F)\cdot P(J|F,A,S)}{\sum_{F'} P(F')\cdot P(G|F')\cdot P(J|F',A,S)} \tag{2.8}$$

Many algorithms have been developed to solve inference more efficiently that also exploit conditional independencies along with other properties of Bayesian Networks. Some examples are the algorithms proposed by Shafer-Shenoy, Hugin and Lauritzen–Spiegelhalter. In our library we have chosen to use the Lazy Propagation algorithm proposed by Madsen, Anders L. and Jensen, Finn V. [3], which will be described in detail in the next section.

2.4.1 Inference with evidences

When dealing with Bayesian networks, commonly, experts want to see how the network behaves with evidences introduced. While the posterior marginals of the children of the observed nodes are easy to find, the same cannot be said for the parents, especially when there are multiple evidences. All the children of a node X which are observations are called *Evidences below X*, while the children of a parent of X, that are observations, are called *Evidences above X*. Given these

definitions of evidence, we will give an example of inference where both types of evidence are present. Given the graph in Figure 2.6, we want the posterior marginal of node C with node B = b and node E = e.

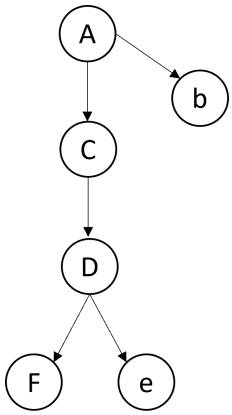

Figure 2.6: Example of Bayesian Network where node E = e and B = b

The operations needed to calculate the posterior of C are:

1. P(b) = $\sum_A P(A)P(b|A)$
2. P(A|b) = $\frac{P(A)P(b|A)}{P(b)}$
3. P(e|C) = $\sum_D P(D|C)P(e|D)$
4. P(e) = $\sum_C P(C)P(e|C)$
5. P(C|b) = $\sum_A P(A|b)P(C|A)$
6. P(C|b,e) = $\frac{P(C|b)P(e|C)}{P(e)}$

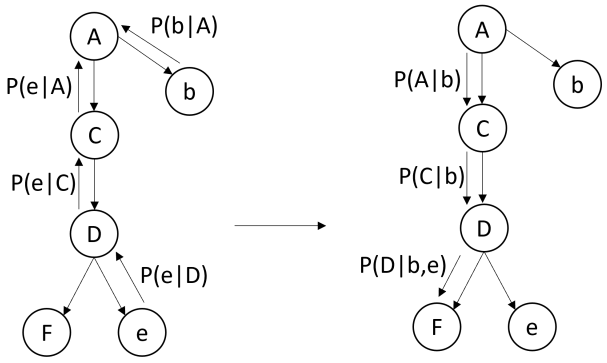

Figure 2.7: Operations needed to calculate any posterior in the network

These operations are the same defined in Pearl's belief propagation [4].
When propagating observations, two rules are applied to determine which observations influence which variables:

- When climbing the tree, to determine the list of ancestors influenced, the propagation stops at each observed ancestors, meaning that an observed variable do not influence the ancestors of another observed variable on its path.

- If a variable is influenced by multiple descendants and one or more of these are non-direct, meaning that there is at least one descendant that needs the updated parent to calculate its posterior, and one of the observed variables is his descendant, then the parent is updated using all the descendants except the descendant of the interested child. For example, considering the DAG of Figure 2.1 with A = a and G = g, then P(F|a,g) needs P(D|a) and not P(D|a,g).

2.5 Junction Trees

Junction trees [3] are a structure used to represent a Bayesian Network's DAG and is used by message passing algorithms, such as Lazy Propagation, to perform inference in a more efficient manner.
It is constructed with the following steps:

- Moralize the DAG: create an undirected graph removing all directions from the arcs and adding an edge between all pairs of parents of a node.

- Triangulate the moralized graph: add a chord connecting two non-consecutive nodes in a cycle of four or more nodes until no cycle of four or more nodes exists. The resulting graph is called chordal graph.

- Find the maximal cliques in the triangulated graph that will constitute the nodes of the junction tree. A clique is a maximal complete sub-graph of the triangulated graph. We will refer to a clique as C.

- Find the separators that connect the cliques and that will determine the junction tree. Each separator S_i that connects two cliques C_i and C_j, is associated with a subset of nodes that corresponds to $SN_i = C_i \cap C_j$. The set of neighbouring separators of a clique will be referred as ne(C).

- Find the probability distributions to insert into each clique. In order to choose a clique, that clique must contain all the parents and the variable itself. A probability distributions can only be contained into one clique. The set of

probability distributions contained in the clique will be referred as Φ_C and their combination forms the clique potential ϕ_C.

$$\phi_{\mathrm{C}} = \prod_{\phi \in \Phi_{\mathrm{C}}} \phi. \tag{2.9}$$

Given the following steps, any junction tree that can be built with these nodes must be a maximum-weight spanning tree, meaning that the sum of cardinalities of each set of nodes associated to each separator must be maximal.
The junction tree must also hold the running intersection property: given two non-consecutive cliques $\mathrm{C_i}$ and $\mathrm{C_j}$ if a node is present in both cliques, then it must also be present in any $\mathrm{C_k}$ within the path that connects the two cliques.

2.6 Lazy Propagation

The Lazy Propagation algorithm [3] bases its architecture on a junction tree where messages are passed between cliques and each message contains a set of potentials according to the nodes present in the separator connecting the cliques. This secondary computational structure is the key for reducing computational complexity of calculating inference for large networks: when a message is passed between cliques, only the potentials required for inference are combined instead of combining all the potentials within the clique. Any observation is incorporated into the junction tree by instantiation of the relative potential and then it has to be propagated to the cliques through messages.
The potentials contained in the messages are not combined until needed from the Lazy Propagation algorithm allowing to exploit conditional independence relations induced by evidence during inference identified by the d-Separation criteria and barren variables.

Definition 2.6.1 (d-Separation) *d-Separation [3] is the criteria used to determine which nodes are conditionally independent from others. Given two variables X_i and X_j, they are d-separated if for all paths connecting X_i and X_j there is a variable X_k that satisfies one of the following conditions:*

- *X_k is the middle variable in a serial or diverging connection and X_k is observed.*
- *X_k is the middle variable in a converging connection and neither X_k nor any of its descendants is observed.*

If two variables are not d-separated they are called d-connected. We will use the algorithm defined by Geiger [5] for finding the set R of variables which are d-connected to a set of J of variables given the observed variables L. Before we find

the set of d-connected variables, we need to find the reachable variables from J given a set of legal pairs of arcs (the definition of legal pair of arcs is given below):

Algorithm 1 Find reachable variables [3]. Given a directed graph G = (V,E), a set F of legal pairs of arcs, and J be the a set of variables, the following operations are performed:

```
1: procedure FIND REACHABLE VARIABLES
2:     R = ∅
3:     Add a temporary node S to V
4:     for X ∈ J do
5:         Add an arc from S to X and label it with 1.
6:         Add X to R.
7:     end for
8:     Label all other arcs with -1.
9:     i = 1.
10:    repeat
11:        for each unlabeled arc (Y,Z) adjacent to at least one arc (X,Y) labeled
       i such that ((X,Y)(Y,Z)) is a legal pair do
12:            Label the arc (Y,Z) with i+1.
13:            Add Z to R.
14:        end for
15:        i = i+1.
16:    until no legal pair of arcs ((X,Y),(Y,Z)) exists
17:    Return R.
18: end procedure
```

Algorithm 2 Find the set of d-connect nodes [3]. Given a directed graph G = (V,E), a of nodes J and a set of observed nodes L, in order to determine the set of nodes R = $\{X \mid X$ d-connected to Y $\in J$ given $L\}$, the following operations are performed:

```
1: procedure FIND THE SET OF D-CONNECT NODES
2:     Descendent = ∅
3:     for X ∈ V do
4:         descendent[X] = true if X is or has a descendent in L, false otherwise.
5:     end for
```

6: Create a directed graph G' = (V,E') where E' = E ∪ (X,Y)|(Y,X) ∈ E.
7: Be F the set of legal pairs of arcs where a legal pair of arcs ((X,Y)(Y,Z)) is such if and only if X ≠ Z and either of the following conditions holds:

1) The node Y is not a head-to-head node on the X - Y - Z in G and Y ∉ L

or

2) The node Y is a head-to-head node on the path X - Y - Z in G and descendent[Y] = true.

8: Use Algorithm 1 to determine the set R of reachable nodes from J by a legal path in G'.
9: Return R.
10: **end procedure**

The algorithm to find the set of d-connected nodes has time complexity linear in the number of arcs of the DAG.

Definition 2.6.2 (Barren variables) *Barren variables [6] are variables of a graphical model that have no observed descendants, are not observed itself, are not target variables of a query and that, since they do not provide any useful information to the posterior probability of interest, we do not want to calculate their posterior probability. For example, if we want calculate the posterior probability of B of Figure 2.3 we will not calculate the posteriors of both C and D as they are not required. A barren variable can be defined as follows:*
Let N = {X,E,P} be a Bayesian Network and let Q = (N,J ⊆ X,ϵ) be a query on N with J the nodes of interest and ϵ the observations. A variable K, with respect to Q, is a barren variable if and only if K ∉ J, X ∉ ϵ and all its descendants are barren themselves.

2.6.1 Message passing

The message passing phase, during which the evidence is propagate to the entire network, is divided into two steps: a collection and a distribution phase. During the collection phase, evidence is collected recursively starting from the predetermined root clique and calling the Collect Evidence algorithm on all its neighbours. When a clique C_i calls Collect Evidence on a neighbour clique C_j, then C_j calls Collect Evidence on all its neighbours until a leaf clique is reached. Once Collect Evidence has finished on C_j, the message from C_j is absorbed by C_i.

Algorithm 3 Collect Evidence [3]. Given two adjacent cliques C_i and C_j, when Collect Evidence is called on C_j from C_i, the following operations are performed:

1: **procedure** COLLECT EVIDENCE
2: C_j invokes Collect Evidence on all its adjacent cliques except C_i
3: C_i absorbs the message sent by C_j with
4: **end procedure**

After the Collect Evidence phase, the Distribute Evidence algorithm is recursively invoked starting from the root clique to distribute the evidence to rest of the network.

Algorithm 4 Distribute Evidence [3]. Given two adjacent cliques C_i and C_j, when Distribute Evidence is called on C_j from C_i, the following operations are performed:

1: **procedure** DISTRIBUTE EVIDENCE
2: C_j absorbs the message sent by C_i with
3: C_j invokes Distribute Evidence on all its adjacent cliques except C_i
4: **end procedure**

The message to be passed from clique C_j to C_i separated by S is equal to calculating a multiplicative decomposition Φ_S of the joint potential of S from the potentials in C_j and the messages received from the adjacent cliques except S. Let R_S be the potentials in C_j and the potentials received except those from S, then the potentials that will inserted into the message are all the potentials except those not present in S. The absorption algorithm works as follows:

Algorithm 5 Absorption [3]. Given two adjacent cliques C_i and C_j and a separator S between them, when absorption is invoked from C_i on C_j, the following operations are performed:

1: **procedure** ABSORPTION
2: Let $R_S = \Phi_{C_j} \cup \cup_{S' \in ne(C_j) \setminus S} \Phi'_S$
3: **for** $X \in \text{dom}(\phi) \mid \phi \in R_S$, $X \notin S$ **do**
4: Marginalize X
5: Let Φ^*_S be resulting the set of potentials, associate it with the separator connecting C_i and C_j
6: **end for**
7: **end procedure**

When passing a message from a clique C_i to C_j the potentials passed in the opposite direction (from C_j to C_i) are not included in the message, in this way we avoid sending back to the clique the same information we received from it.

2.6.2 Evidence

The Lazy Propagation algorithm has a method for incorporating evidence into the network in order to exploit independence relations. When a variable X has been observed to take value x, every clique containing X will receive evidence through the message passing phase and each potential that has X as parent will not have X in its domain. This process will be referred as instantiation of potentials ϕ_C.

2.6.3 Internal elimination

The junction tree on which is based the Lazy Propagation algorithm defines, as said before, a partial elimination order of the variables reducing the number of possible elimination orders during inference. This is because the separators S between two adjacent cliques C_i and C_j defines which variables will not be included into the message passed between the two cliques in either the collection or distribution of evidence phase. However, the separator is not able to define in which order the variables should be eliminated.
The phase in which the variables are eliminated from the message, when computing it, is called *internal elimination* [3] and takes place after the absorption algorithm since it is not able to determine which potential of R_S is really relevant to clique C_j. To further remove unnecessary potentials to clique C_j we can exploit barren variables, the unity-potential axiom and independence relations induced by evidence using a d-separation algorithm thus obtaining R'_S which is the set of relevant potentials to C_j.

Axiom 2.6.1 (Unity-potential axiom) *From [3] the unity-potential axiom is defined as follows:*
$\sum_H P(H|T) = 1_T$.
The axiom also applies if H is a set of variables.

Algorithm 6 Find relevant potentials to clique C_j [3]. Given a set of potentials Φ_C and a separator S, R'_S can be found with the following steps:

1: **procedure** FIND RELEVANT POTENTIALS
2: Let $R_S = \phi_C \in \Phi_C \mid \exists X \in \mathrm{dom}(\phi_C)$ such that X is d-connected to $Y \in S$
3: Using the unity-potential axiom remove potentials containing only barren head variables from R_S to obtain R'_S
4: Return R'_S
5: **end procedure**

2.6.4 Posterior marginals

Given any consistent junction tree obtained from the DAG of the network, the posterior marginal of variable X can be calculated from any clique or separator potential that contains X in its domain by marginalizing out all other variables from dom(ϕ) except X:

$$P(X, \epsilon) = \sum_{Y \in dom(\phi) \setminus X} \phi. \tag{2.10}$$

The formulation given above is the general case, but in the Lazy propagation algorithm the formulation is a bit different since the potentials are store as a multiplicative decomposition, they also need to be combined before marginalization.

Algorithm 7 Marginalization algorithm [3]. Given Φ a set of potentials and a variable X to be marginalized, marginalization is performed with the following steps:

1: **procedure** MARGINALIZATION ALGORITHM
2: Be $\Phi_X = \phi \in \Phi \mid X \in dom(\phi)$ the set of potentials with X in their domain.
3: Calculate $\phi_X^* = \sum_X \prod_{\phi \in \Phi_X} \phi$.
4: Let $\Phi^* = \phi_X^* \cup \Phi \setminus \Phi_X$.
5: **end procedure**

Φ^* is the resulting set of potentials where X is marginalized from Φ.

The algorithm for finding the relevant potentials can be used to determine which variables are not needed for the posterior marginal requested:

Algorithm 8 Posterior marginal [3]. Given Φ the set of potentials representing the joint distribution from which the posterior of X is to be calculated, the posterior marginal can be found with the following steps:

1: **procedure** POSTERIOR MARGINAL
2: Use the find relevant potentials on Φ to obtain R_X.
3: **for** Y in $\{Y \in dom(\phi) \mid \phi \in R_X, X \neq Y\}$ **do**
4: Marginalize Y.
5: **end for**
6: Be Φ_X the set of potentials obtained.
7: Calculate.

$$P(X|\epsilon) = \frac{\prod_{\phi \in \Phi_X} \phi}{\sum_X \prod_{\phi \in \Phi_X} \phi} \tag{2.11}$$

8: **end procedure**

2.7 Learning network structure and probability distributions from input data set

Learning the network structure and the probability distribution associated with each variable from an input data set is a common task when the user wants to recreate the causal relationships that generated to data set in fields such as economics, marketing etc. What is commonly done is using an algorithm to learn the causalities among variables in the data set and use those relationships to create a representation of the problem. This is also facilitates updating the network when more observations become available and the model needs to be refitted.
The learning task is divided into two phases:

- The first phase is dedicated to learn the structure of the network exploiting the causalities identified among variables. A few examples of algorithms used in this phase are: the PC (from the names of its authors Peter and Clark) algorithm [7], the Max-Min Hill-Climbing algorithm [8] and the PCHC (PC Hill-Climbing) algorithm [9].

- The second phase is dedicated to estimate the probability distributions using the relationships identified. Two most commonly used algorithms are: the Bayesian Estimation and the Maximum Likelihood Estimation (MLE).

In our library we used the PCHC algorithm for the structure learning phase and the MLE algorithm for the probability distribution estimation phase. Both algorithms will be described in details in the following sections.

2.7.1 PCHC Algorithm

The PCHC algorithm [9] is a recently proposed algorithm by Tsagris, M. for performing structural learning of a Bayesian Network. It is a hybrid algorithm meaning that it is composed of two phases: one phase is dedicated to the identification of the most relevant relationships among variables (in this phase the edges in the network are not directed yet), using conditional independence tests, and a subsequent phase is dedicated to determine the directions of the edges through a scoring metric and an Hill-Climbing algorithm, with the target to find the network structure with the highest score.
Compared to the MMHC algorithm, the PCHC has a higher computational efficiency. This is because the MMHC has been designed to be used on small data sets, as the ones frequently seen in bioinformatics. But as the sample size increases, the complexity of the algorithm makes the whole computation expensive in terms of time and electricity required from the computers to run the algorithm. The PCHC, having better performance than the MMHC, allows to reduce the time

required to have a result, and thus reducing the time required to decision making, but also reduces the monetary cost of electricity, ultimately reducing also carbon emission and ambient pollution. The better efficiency of PCHC allows to handle data sets of millions of observation in just a few minutes, maintaining or improving MMHC's accuracy. A nice property of PCHC is that its computational cost scales linearly with the sample size, meaning that if the sample size increases of a factor, the execution time increases of the same factor.

Conditional Independence Tests

Conditional independence tests (or CI tests) [9] are used to identify the skeleton of the network through the validation of the following statement: given two variables X and Y and a set of variables Z, possibly empty, X and Y are conditionally independent given Z ($X \perp\!\!\!\perp Y \mid Z$) if and only if $P(X,Y|Z) = P(X|Z) \cdot P(Y|Z)$ and holds for all values of X,Y and Z. CI tests are classified by the type of variables used in the network. A few examples of CI tests are: the Pearson correlation for continuous data and the G^2 independence test for categorical data. In our library we adopted the G^2 independence test since we only support categorical variables at the moment.

G^2 independence test

The G^2 independence test [9] is defined as:

$$G^2 = 2 \cdot \sum_k \sum_{i,j} O_{ij|k} \cdot log \frac{O_{ij|k}}{E_{ij|k}} \tag{2.12}$$

$O_{ij|k}$ are the observed frequencies when variable X is in its i-th value and variable Y is in its j-th value and the set of variables Z is in its k-th value, while $E_{ij|k}$ is the expected frequencies of those values. The G^2 test follows a X^2 distribution with (|X| - 1)(|Y| - 1)(|Z| - 1) degrees of freedom under the conditional independence assumption.

PCHC skeleton identification phase

Since the PCHC [9] is based on the PC algorithm, the skeleton identification phase uses the same algorithm defined in PC. The algorithm starts with a fully connected graph and through CI tests on pair of variables, edges that represent statistically insignificant associations are removed. At each iteration of the algorithm the conditioning set Z increases its size by one starting from zero. The algorithm continues until no edge is removed. In order to guarantee determinism, when

considering a variable V_i for CI test, it is tested against those variables V_j that are least probabilistically dependent, conditioned on those variables V_z that are most probabilistically dependent. This guarantees independence from the order in which variables are placed in the dataset.

Algorithm 9 Skeleton identification phase of the PCHC algorithm. Given an input dataset D with variables V, the skeleton is found with the following steps:

1: **procedure** SKELETON IDENTIFICATION
2: Let G = (V,E) be a fully connected graph with the variables contained in the dataset.
3: Let k = 0.
4: **repeat**
5: **repeat**
6: Select an ordered pair of variables V_i and V_j, such that |adj(G,V_i) \{V_j} | ≥ k and a subset S with |adj(G,V_i) \{V_j} | = k).
7: If(V_i ⫫ V_j | S) delete edge (V_i, V_j) from G.
8: **until** All ordered pairs of adjacent variables V_i and V_j with |adj(G,V_i) \{V_j} | ≥ k and all subsets S with |adj(G,V_i) \{V_j} | = k have been tested for conditional independence.
9: k = k + 1.
10: **until** For each ordered pair of adjacent variables V_i and V_j, |adj(G,V_i) \{V_j} | ≤ k
11: Return G.
12: **end procedure**

As stated by [10], the complexity of the PC algorithm is, in the worst case, $O(|V|^p)$, where V is the number of variables in the dataset and p is the size of the largest set of parents and children over all variables.

Hill-Climbing phase of PCHC

After having determined which edges represents independencies between the variables of the network and having removed them, the edges are oriented in order to create the DAG the will represent the Bayesian network. This is done through a scoring phase: the DAG is evaluated through a metric in order to determine which configuration of arcs represents better the relationships among variables.
The DAG is generated doing all kinds of transformations possible to the edges and determining which DAG has the best score. The possible transformations are:

- Add an arc to the DAG only if it was discovered by the skeleton identification phase and it does not introduce a cycle.

- Reverse an arc previously added to the DAG only if it does not introduce a cycle.
- Delete an arc from the DAG.

At each step the one transformation of the three above that brings the highest score increase is performed on the DAG.
This phase is a combinatorial problem and as such its time complexity makes it unfeasible to explore all possible combinations of arcs especially on networks with thousands of variables. In order to reduce the time required to obtain a good enough network, [8] and [9] use the Hill-Climbing search heuristic along with a tabu list that keeps track of the last 100 configurations explored. Being the Hill-Climbing Heuristic a local search heuristic type algorithm, it can get stuck in local maxima when exploring the space of possible transformations. In order to avoid getting stuck in a local maxima, the criterion to choose a transformation needs to be changed: instead of taking the best transformation overall, we take the best transformation that results in a network not in the tabu list. This can reduce the score of the network but avoids getting stuck in local maxima. When 15 consecutive transformations do not produce an improvement, the search stops.
Our implementation uses the Fast Constrained Hill-climbing (FastCHC) heuristic defined in [11]. The FastCHC is based on the CHC algorithm which introduces constraints on the neighborhood at each iteration, reducing the number of possible neighbours of each node. A list of forbidden parents (FP) is kept for each. A node X_j can be selected as parent of X_i if $\mathrm{X}_j \notin \mathrm{FP}[\mathrm{X}_i]$. Given a score function f, X_j is included in $\mathrm{FP}[\mathrm{X}_i]$ when the move considered does not increase the score of the structure and the FP sets are updated with the following rules:

- Adding arc (X_j, X_i). If $f(\mathrm{X}_i, \mathrm{pa}(\mathrm{X}_i) \cup \mathrm{X}_j)$ - $f(\mathrm{X}_i, \mathrm{pa}(\mathrm{X}_i)) < 0$, then X_j is added to $FP[\mathrm{X}_i]$ and vice versa.
- Deleting arc (X_j, X_i). If $f(\mathrm{X}_i, \mathrm{pa}(\mathrm{X}_i) \setminus \mathrm{X}_j)$ - $f(\mathrm{X}_i, \mathrm{pa}(\mathrm{X}_i)) > 0$, then X_j is added to $FP[\mathrm{X}_i]$ and vice versa.
- Reversal of arc (X_j, X_i). The operation can be seen as the union of deleting arc (X_j, X_i) and adding arc (X_i, X_j). The two cases above are then applied to update FP

The normal CHC does not guarantee that the resulting structure is an I-map; for this reason either the unconstrained HC can be applied to solve this problem or an iterated version of the CHC can be used as it has the same stopping condition of an unconstrained HC, i.e. it stops when no changes can be applied at the beginning of an iteration.
The advantage of the FastCHC is that it returns a minimal I-map without performing an iterated local search thanks to a relaxation of the constraints. Whenever an

arc is added to the DAG, all the nodes in the neighborhoods of head and tail nodes are removed from the other node's forbidden parents set. The faster execution time of the FastCHC comes at the cost of a reduction in quality of the resulting network. The FastCHC also has advantage of not requiring a tabu list saving a lot of space for large networks.

2.7.2 Bayesian-Dirichlet sparse (BDs)

The BDs score is a recently proposed metric by Scutari, Marco for scoring the structure of a Bayesian network. In this thesis we will give a brief overview of the metric and its formulation. For a detailed discussion of the metric read [12].
One of the most used metrics to score a Bayesian network is the Bayesian-Dirichlet uniform equivalent (BDeu) and its formulation is:

$$BD(G, D; \alpha) = \prod_{i=1}^{N} BD(X_i | \prod_{X_i}^{G}; \alpha_i) \tag{2.13}$$

$$= \prod_{i=1}^{N} \prod_{j=1}^{q_i} \left[\frac{\Gamma(\alpha_{ij})}{\Gamma(\alpha_{ij} + n_{ij})} \prod_{k=1}^{r_i} \frac{\Gamma(\alpha_{ij} + n_{ij})}{\Gamma(\alpha_{ijk})} \right] \tag{2.14}$$

where:

- Γ is the gamma function.
- r_i is the number of states of variable V_i.
- q_i is the number of possible configurations of values of $\prod_{X_i}^{G}$; if the variable X_i has no parents $\mathrm{q}_i = 1$.
- $\mathrm{n}_{ij} = \sum_{k=1}^{r_i} \mathrm{n}_{ijk}$.
- $\alpha_{ij} = \sum_{k=1}^{r_i} \alpha_{ijk}$.
- $\alpha = \{\alpha_1,...,\alpha_N\}$, where $\alpha_i = \sum_{j=1}^{q_i} \alpha_{ij}$ are the imaginary sample sizes associated with each X_i.

Each BD score uses a different α_{ijk} that produces different priors:

- The BDeu score uses $\alpha_{ijk} = \alpha \ / \ (\mathrm{r}_i \mathrm{q}_i)$, which is the most common choice, and uses $\alpha_i = \alpha$ for all X_i.
- The BDs score uses $\alpha_{ijk} = \alpha \ / \ (\mathrm{r}_i \ \tilde{q}_i)$, where $\tilde{q}_i$ is the number of $\prod_{X_i}^{G}$ such that $\mathrm{n}_{ij} > 0$.

However the DAGs learned with the BDeu score are highly dependent on the value of α: large values of α produce DAGs with more arcs with respect to lower values of α. This behaviour is the opposite to what is expected, that is a higher value of α should produce a DAG with fewer arcs. For a more in depth discussion of this is behaviour, please refer to [12].
The BDeu score is then defined as:

$$\begin{aligned} &BDeu(X_i|pa(X_i)_G;\alpha) \\ &\quad= \prod_{j:n_{ij}=0} \cancel{\left[\frac{\Gamma(r_i\alpha_{ijk})}{\Gamma(r_i\alpha_{ijk})} \prod_{k=1}^{r_i} \frac{\Gamma(\alpha_{ijk})}{\Gamma(\alpha_{ijk})}\right]} \prod_{j:n_{ij}>0} \left[\frac{\Gamma(r_i\alpha_{ijk})}{\Gamma(r_i\alpha_{ijk}+n_{ij})} \prod_{k=1}^{r_i} \frac{\Gamma(\alpha_{ijk}+n_{ij}k)}{\Gamma(\alpha_{ijk})}\right] \end{aligned} \tag{2.15}$$

In addition, the BDeu score presents a problem related to the imaginary sample size α_{ijk}, that is that in Equation 2.15 $\alpha_i jk$ does not simplify and the resulting imaginary sample size if $\alpha(\tilde{q}_i/q_i)$ instead of α losing the term $\alpha(q_i - \tilde{q}_i)/q_i$. This could make compare two marginal likelihoods computed from different priors, which is incorrect.
Scutari [12] proposes a change to α_{ijk} to prevent this problem:

$$\alpha_{ijk} = \begin{cases} \alpha/(r_i\tilde{q}_i) \text{ if } n_{ij} > 0, \text{ where } \tilde{q}_i \text{ is the number of } pa(X_i)_G, \text{ such that } n_{ij} > 0 \\ \\ 0 \text{ otherwise} \end{cases}$$

$$BDs(X_i|pa(X_i)_G;\alpha) = \prod_{j:n_ij>0} \left[\frac{\Gamma(r_i\alpha_{ijk})}{\Gamma(r_i\alpha_{ijk}+n_{ij})} \prod_{k=1}^{r_i} \frac{\Gamma(\alpha_{ijk}+n_{ij}k)}{\Gamma(\alpha_{ijk})}\right] \tag{2.16}$$

2.7.3 Maximum Likelihood Estimate Method

One of the most common methods used to learn the probability distributions of the network, from a complete dataset, is the Maximum Likelihood Estimate (MLE) [13]. The idea behind MLE is: testing randomly an event C may results in many different outcomes C^1, C^2, ... C^n. The estimated value $\hat{C}$ of event C will be represented at Θ if can maximize the likelihood function $(P|\Theta)$. The likelihood function is:

$$L(\theta : D) = P(D|\theta) = \prod_m P(D_m|\theta) \tag{2.17}$$

Applying this formula to a Bayesian network with n variables results in:

$$L(\theta : D) = \prod_m \prod_i P(x_{im}|pa(i)_m, \theta_i) \tag{2.18}$$

$$= \prod_i \prod_m P(x_{im}|pa(i)_m, \theta_i) \tag{2.19}$$

$$= \prod_i L_i(\theta_i : D) \tag{2.20}$$

If the probability $P(X_i|pa(i)_m, \theta_i)$ can be approximated by a polynomial distribution, the local likelihood can be simplified to:

$$L_i(\theta_i : D) = \prod_m P(x_{im}|pa(i)_m, \theta_i) \tag{2.21}$$

$$= \prod_m \prod_{pa(i)^j} \prod_{x_i'} P(x_{im}^k|pa(i)_m{}^j, \theta_i) \tag{2.22}$$

$$= \prod_{pa(i)^j} \prod_{x_i'} \theta_{x_i' pa(i)}^{N(x_i{}', pa(i)^j)} \tag{2.23}$$

Since this method is applied on complete datasets, as defined by [13], for each possible value $pa(i)^j$ of the parents' node pa(i), the probability distribution $P(x_i|pa(i)^j)$ is the independent polynomial distribution, which is not related to all other values of $(pa(i)^l (l \neq j))$. Taking this into consideration, the estimated parameter θ is calculated as follows:

$$\theta_{x_i^k} = \frac{N(x_i^k, pa(i)^j)}{N(pa(i)^j} \tag{2.24}$$

$$= \frac{\text{number of cases where } x_i = k \text{ and } pa(i) = j}{\text{number of cases where } pa(i) = j} \tag{2.25}$$

2.8 Strength Of Influence

Once the posterior probability distribution of interest has been obtained, often the user needs also information about what are the determining factors for the results, namely which events have influenced the most the result. This information can be expressed as "Strength Of Influence" which is a numerical value indicating which parent event influences the most the result, the higher the number the higher the influence. We followed the strategy defined by [14] for measuring the influence a node and its parents.
In [14] a dynamic approach is proposed in order to measure the strength of influence between nodes: firstly, the posterior marginal, without any observation added to the network, of the node of interest is calculated and then it is calculated considering

each parent observed singularly. This means that if a parent's node has n states, then n different probability distributions has to be calculated. If a node has N parents then N·n probability distributions will be calculated. The type of measures that can be requested are the average and the maximum of the distance between the two distributions. The function used to calculate the distance between the two distributions needs to be symmetrical, in other words:

$$D(P, Q) = D(Q, P). \tag{2.26}$$

To explain why this property is needed, we will use the example in [14]: consider a network like the one in Figure 2.8 where all nodes are binary and the following probability distributions are taken into consideration: P(B|A = 0) = [0.9, 0.1], P(B) = [0.5, 0.5], P(C|B = 0) = [0.5, 0.5] and P(C) = [0.9, 0.1]. The problem of using an asymmetric distance arises when considering the case:

$$D(P(B), P(B|A)) == D(P(C), P(C|B)) \tag{2.27}$$

$$\rightarrow D([0.5,0.5], [0.9,0.1]) == D([0.9,0.1], [0.5,0.5]) \tag{2.28}$$

If the distance function were asymmetric, then measuring the distance between the same two points would differ based on the direction considered. The distance

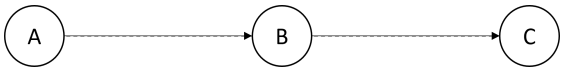

Figure 2.8: A Bayesian network with 3 nodes

measures that we have implemented are the same proposed by [14].

2.8.1 Euclidean distance

The Euclidean distance is defined as:

$$E(P, Q) = \sqrt{\sum_{i=1}^{n} (p_i - q_i)^2} \tag{2.29}$$

The value of distance is in the range $[0, \sqrt{2}]$. It is symmetrical and can be applied to any N-dimensional vector (or in our case probability distributions).

2.8.2 Hellinger distance

The Hellinger distance is defined as:

$$H(P, Q) = \sqrt{\sum_{i=1}^{n} (\sqrt{p_i} - \sqrt{q_i})^2} \tag{2.30}$$

This distance measure has the same properties as the Euclidean distance but is more sensitive the value approaches 0 or 1.

2.8.3 Kullback-Leibler distance

The Kullback-Leibler distance is defined as:

$$K(P,Q) = -\sum_{i=1}^{n} p_i \cdot log_2(q_i) + \sum_{i=1}^{n} p_i \cdot log_2(p_i) \tag{2.31}$$

$$= \sum_{i=1}^{n} (p_i \cdot log_2(\frac{p_i}{q_i})) \tag{2.32}$$

This distance has values in the range [0, inf). This distance has three problems: first, it is not symmetric, second, its range goes to infinity, and third, if $q_i = 0$ there is division by zero. However, this distance can be used to create a symmetric distance called J-divergence.

2.8.4 J-divergence

In order to make the Kullback-Leibler distance symmetric we can define the J-divergence as follows:

$$J(P,Q) = \frac{K(P,Q) + K(Q,P}{2} \tag{2.33}$$

This solves the symmetry problem but not the other two. To avoid its range from going to infinity we can rewrite 2.33 as:

$$J_norm(P,Q) = \frac{J(P,Q)}{\sqrt{J(P,Q)^2 + \alpha}} \tag{2.34}$$

where $\alpha \geq 0$ is a parameter controlling the degree of the normalization. There is still the problem regarding $q_i = 0$ to solve. To do that we can add a simple condition:

$$J_norm(P,Q) = \begin{cases} 1 \ \forall \ i \in [1, ..., n], \ q_i = 0 \\ \\ \frac{J(P,Q)}{\sqrt{J(P,Q)^2+\alpha}} \text{ otherwise} \end{cases} \tag{2.35}$$

2.9 Libraries used as comparison

2.9.1 aGrUM

aGrUM is a popular open-source library, written in C++17 and without a GUI, for manipulating graphical models such as Bayesian networks, influence diagrams and decision trees. aGrUM offers support to operations such as: exact or approximated

inference, structural learning and parameter learning.
We used this library as a comparison for the exact inference operations and our results will show that this library suffers from inaccuracy due to wrong marginalizations applied to variables in certain situations, affecting the whole inference task. We also used aGrUM as a benchmark to test the performance of our library in terms of time required for inference and RAM occupied during execution.

2.9.2 GeNIe

GeNIe [14] is a graphical software, specifically made for Windows, that uses the SMILE library to performs its operations on graphical models. SMILE is a platform independent library that supports operations on graphical models. Through the SMILE APIs, users can create, edit, save and load graphical models. Although GeNIe is said to support graphical models of any size and the only limitation is the capacity of the RAM installed on a computer, we will shows that this assumption is not true at least in the academic version of the software, which is the one we used as reference. GeNIe, just like aGrUM, suffers from inaccuracy due to wrongly marginalized variables.
We used GeNIe as a comparison for the exact inference task with observations as it was easier to manipulate the network compared to aGrUM.
The problems emerged during the work of this thesis in both GeNIe and aGrUM will be discussed in detail in Section 5.

2.9.3 bnlearn - R Package

bnlearn [15] is an R package, developed by Marco Scutari since 2007, that focuses on the learning of a graphical model, supporting algorithms such as MMPC and MMHC for structural learning and the MLE method for parameter learning.
We used this library as a support tool for the development of our library's learning features and for validating our results.

Chapter 3

Library Design

Before diving into an in depth discussion on the implementation of the functionalities, we will give an overview of the design of our library.
The theories presented in Chapter 2 are a good foundation on which base an efficient library for Bayesian networks but they are not enough to obtain top-notch performance in terms of execution time and memory required. A solution to the first problem, by taking advantage of modern CPUs that have many cores and threads, is using cpu parallelization through the use of a threadpool and dividing algorithms, that would benefit from parallelization, in smaller tasks like the Lazy Propagation algorithm, the multiplication of two probability distributions and the learning of a network.
Regarding memory occupation, there are three major aspects of the library that have the highest impact on memory usage which are: storing probability distributions, duplication of CPTs due to the message passing phase and the input dataset for learning a network.
In order to reduce the memory required to store sparse probability distributions, we used a compact representation where the sparse value is not repeated. To avoid the duplication of the same probability distribution when a CPT is copied during the message passing phases or in case more CPTs have the same probability distribution, the copy-on-write paradigm has been used on the data structure containing the probability distribution. Lastly, a run length coding approach has been used on the records of the input dataset (from which the network is learned) to reduce the space occupied in case the same value of a variable is repeated across many consecutive records.
Since we expect this library to be used from anyone who deals with Bayesian networks, we allow the user to choose the precision of the calculations that he desires. For this reason the library is completely templated in order to support all standard types of floating point precision (float, double, long double) and any custom floating point precision.

3.1 CPT Class

The CPT class is dedicated to manage and store all the information relative to a variable (name, id, names of the resulting states, probability distribution, observations introduced, etc...). In this class are present all those functionalities needed to multiply CPTs, marginalize a variable from a CPT, instantiate the CPT and general functions to retrieve information relative to the CPT such as name, id, probability distribution, variables list, etc... The CPT class takes advantage of the COW technique to reduce the memory required to store its probability distribution when the same probability distribution is present in two or more CPTs.
After the CPT has been subject to any manipulation or after it has been read, the probability distribution is checked to determine if it sparse and if yes it is transformed in a more compact representation.
It is a fully templated class for supporting different operations precision.

3.2 Arcset Class

The Arcset class is the basis class used to store the structure of directed graphs. It uses two maps to store the parents and the children of each node and a set of arcs used to ease some of the operations in the class, although it is redundant. This class offers the basic functionalities needed for dealing with directed graphs, like adding, removing or checking if an arc exists in the graph, get list of parents, children, descendant, ancestors or neighbouring nodes of a given node, to other classes that will inherit the Arcset class (such as the DAG class).

3.3 Edgeset Class

The Edgeset class is the basis class used to store the structure of undirected graphs. It uses one map to store the connections between nodes and uses a set of edges some of the operations in the class. Like the Arcset class, it offers basic functionalities needed to deal with undirected graphs, like adding, removing or checking if an edge exists, obtain the common neighborhood of two nodes, check if the neighborhood is a clique etc.

3.4 DAG Class

The DAG class inherits from the Arcset class in order to store the structure of the Bayesian network, which can be either loaded by an input file or a defined by hand from the user. The functions contained in it are specific to the DAG use case,

like finding the d-connected nodes for the inference task, get the list of non barren variables and check if the graph is a DAG.

3.5 Bayesian Network Class

The Bayesian network class contains the structure of the network along with the list of CPTs. It can be seen as a wrapper class since most of the functionalities offered by this class are functions belonging to the DAG and the CPT classes. The class also assigns a unique id to each CPT in order to easily retrieve any information during execution.
It is a templated class since it has to handle the CPTs.

3.6 Undirectedgraph Class

The UndirectedGraph class inherits from the Edgeset class and is responsible of the triangulation of the DAG and identification of the cliques and separators that will constitute the junction tree. The triangulation methods available are three: the Minimum Weight Clique Heuristic (MWCH) [16], the MCS-M [17] and the Lex-M, which is just a variation of the MCS-M. In Chapter 5 we will compare how MCS-M and MWCH algorithms performs in different networks. Since they produce different triangulations, it is not obvious to assume which algorithm performs better on a network.
After the triangulation phase, the MCS algorithm, described in [18], is used to find the set of cliques and the separators of the junction tree. The peculiarity of this algorithm is that it finds all the cliques and separators in one iteration over all the nodes of the graph.

3.7 JunctionTree Class

The JunctionTree class handles the creation of the junction tree after having received the list of cliques and separators from the Undirectedgraph class. The separators received already contains the id of one of the two cliques to which it is connected. To other clique is easily found by searching which clique, with a lower id than the one already identified, generates the same intersection with the other clique as the one stored in the separator; the id of the clique is then associated to the separator. After the junction tree has been created, the CPTs of the Bayesian network need to be associated with one clique only. In order to determine to which clique each CPT will be associated, the clique has to contain the ids of all CPT's parents and the CPT's id itself. If there are more cliques suitable, only one of them

is chosen.
The idea behind this class is that of a more complex data structure on which the LazyPropagation class will operate: the LazyPropagation class is declared as friend of the JunctionTree class. This avoids the creation of a lot of getter methods, and consequentially a reduction of memory occupation due to less copies of information, and setter methods but still maintaining encapsulation of methods with respect to other classes that could potentially operate on the junction tree. This allows for a faster manipulation of the data related to the junction tree such as the messages passed between cliques and potentials present in any clique.

3.8 LazyPropagation Class

Within the LazyPropagation class are contained all functionalities needed to perform inference in a Bayesian network: it can be performed either by choosing a node as target or not; if a target node is selected, barren variables can be exploited to further reduce execution time; it is also possible to immediately retrieve any other posterior marginal from the network as long as it was not a barren variable of the last execution. If it was a barren variable then the messages of the Lazy propagation are updated to satisfy the requested posterior marginal. The algorithm in [3] does not explain some of the details regarding its definition, i.e. what it is intended for "multiplicative decomposition", so we will give our interpretation of the algorithm in Chapter 4, trying to clarify those gray areas of [3].
As anticipated, the Lazy propagation is a friend class of the JunctionTree class allowing to directly explore the tree as if it was a part of the class. The algorithm can be performed either in a serial or multi threaded fashion, depending on the available system resources.

3.9 Network Learning

The first step is to read the input dataset, containing the observations from which the network will be learned, stored in a "CSV" file. The CSVReader class will load the file and store the information in an instance of CSVDataStructure. The run length coding is applied to the information stored in order to reduce memory occupation. The learning of a network is divided into three major classes: the PCHC and HillClimbingScoringPhase classes, are responsible for the structural learning, and the MLE class is responsible for the probability distributions learning. The implementation of the PCHC algorithm is contained in the homonym class and is divided into two phases: the PC skeleton identification phase and the Hill-climbing score heuristic. The PC algorithm can be easily parallelized to exploit all cores of modern multi-core CPUs obtaining better performance as described in

[7]. The use of the parallel version of the PC algorithm is again dependant on the available resources of the system.
Once the PC algorithm has finished its execution, the resulting undirected graph will be oriented through the Hill-Climbing heuristic, which uses the BDs score to determine the best operation to be applied on the graph, obtaining the DAG representing the Bayesian network.
Before the PCHC class starts its execution, the user can introduce its knowledge of the problem that is being modelled by specifying constraints on the structural learning, that is which relationships between variables must be present and which has not to be considered.
The last step needed to learn a network is to estimate the probability distribution of each variable. This task is carried out by the MaximumLikelihoodEsitmation class using the formulation defined in Section 2.7.3.

3.10 Strength Of Influence

The last module is the related to measuring the strength of influence between variables, in particular the influence that each parent has on a child. To perform the task, the user has to specify which node he is interested in measuring the strength of influence, which measure is the most appropriate to its needs between the Euclidean distance, the Hellinger distance and the J-divergence and which kind of measure he wants: the average or the maximum influence.
The result is the influence that each parent has on the posterior marginal of the target node chosen.
The operations are carried out as defined in [14].

Chapter 4

Implementation Details

The goal of this chapter is to give details about the implementation of the previously described functionalities, explaining the changes made to some of the algorithms seen in Chapter 2 needed for their implementation. We will also discuss the work done to improve code efficiency from both CPU and memory point of view.
Throughout the discussion of the thesis, every vector used as parameter is passed by reference to reduce execution time. Furthermore, the position in which every vector is declared is fundamental to reduce execution time: there are many cases in which the vector could have been created inside the loop since it is were it gets used but this has the downside of deallocating and reallocating a lot of times the same vector. Declaring the vector before the loop and adjusting its size to the needs of the iteration is far more efficient. Lastly, when a vector/list, a set or a map is returned by a function it is always moved in the result variable, avoiding an unnecessary copy of the whole vector.
For each relevant algorithm we designed for our library we will provide time and space complexity. Those algorithms, although modified, which do not have such indication have the same time and space complexity defined in the document from which they are taken. They are included in order to describe the modification made.

4.1 CPT Class

Inside the CPT class is present a reference to an instance of CPTInfo, where all the CPT's information are stored. This brings benefits on execution time and memory usage since when a CPT is copied, only the reference is copied and not the whole data structure.
The CPTData structure, instead, is referenced through inheritance of the COW class. By doing so, we apply the COW technique to the CPTData instance achieving

both better execution time and reducing memory usage during execution. The execution time reduction comes from the use of a shared pointer that points to the data structure and in case of copy of the CPT, only the pointer is copied and not the whole data structure; this is particularly helpful when sending messages during the Lazy Propagation where a lot of CPTs can be copied. Furthermore, if the same probability distribution is present in more than one CPT, this mechanism can save memory by not duplicating the distribution. In order to identify CPTs with the same probability distributions, during the reading from a file of the network, each CPT is checked against all the previous read ones to see if they share the same distribution. If there is a CPT with the same distribution, then the pointer in the latter CPT is made reference the already present data.
The most logical and less computational taxing structure to use for storing the probability distribution of the CPT is a vector. While this reduces the cost of accessing each single element compared to using a map, for example, it is not the most efficient way of handling large distributions were the majority of values is one value repeated many times, leading to a waste of memory. To reduce the memory used by this vector, a check is performed to determine if the distribution is sparse. In the affirmative case, the repeated value is saved into a variable, in the CPTData structure and all other values are stored in the vector. For each one of those remaining values, its new index in the vector is associated with the old index and the association is stored in the map of CPTData in order to retrieve it if needed.
The CPT class can potentially handle huge probability distributions requiring that the operations are carried out as efficiently as possible. The most common operations are: multiplication of two CPTs, marginalization and instantiation of a variable in the CPT. The first operation is the one that can potentially have the highest computational demand if the number of parents of the CPT is high enough. Since this task consist mostly on iterating over a vector it can be easily parallelized in smaller tasks that can be sent to the thread pool. The parallelization applied to the multiplication of CPTs is explained in Algorithm 10 and Algorithm 11.

4.1.1 CPT Data Structures

VarStates

The information relative to the parents of a CPT, the number of states that each parent has and the number of states that this CPT has is stored inside a vector of VarStates. Each VarStates variable contains: the variable id, the number of states, the list of descendants of which the variable is its ancestor and a flag to check whether this parent has been already multiplied. We will refer to this vector as variablesOrder. During the discussion of the following algorithms, when referring to the list of variables present in the potential, we refer to a list of ids only of the

variables, ordered in descending order (the first id is always the potential's id).
The list of descendants is fundamental when we are in the following situation: considering the graph in Figure 4.1, one way to find the posterior P(C) is by first marginalizing the incomplete parent B and then multiplying A two times:

$$P(C) = P(C|B, A) \cdot P(B|A) \cdot P(A) \tag{4.1}$$
$$= P(C|A, A) \cdot P(A) \cdot P(A) \tag{4.2}$$

When doing so, we need to keep track of which A is which, i.e. one A is a direct parent of C and the other is the parent of B. During the development of our library, we have seen that this fundamental piece of information is lost in the inference carried out by GeNIe and possibly also in aGrUM, although the results are much similar to the real ones. The behaviour of both GeNIe and aGrUM will be discussed later with better examples.

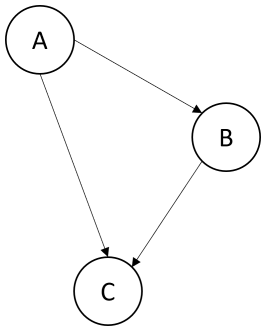

Figure 4.1: Bayesian network with joint distribution P(A) · P(B|A) · P(C|A,B)

CPTInfo

It is a simple data structure used to store the information identifying a CPT: the variable name and the names of the resulting states. This structure allows to save space and time during the message passing phase of the Lazy Propagation.

CPTData

It is a templated structure so that different accuracies for the distribution's values can be used. Inside the CPTData structure is stored the probability distribution of the CPT using a vector, the sparse value of the original distribution, the map with the indices of the remaining combinations after conversion of the distribution to its compact representation, the probability distribution size and the number of times the sparse value appeared in the original distribution.
The map used to store to association between the compact and the sparse distribution does the following association : <index in the compact vector, index in the sparse vector>.

4.1.2 CPT Class Functions

CPTs Multiplication

The multiplication of CPTs is divided into two phases: the first one determines whether the multiplication can be parallelized and the second one is the actual multiplication that can be divided into smaller tasks if parallelized.

Algorithm 10 EvaluateCPT. Given the multiplicand CPT CPTmult and the variablesOrder of the resulting CPT after the multiplication, the resulting CPT's probability distribution is calculated with the following steps:

```
1: procedure EvaluateCPT(CPTmult, newVariablesOrder)
2:     Let numOfCombinations ← the size of the new probability distribution.
3:     Be newProbDistribution the vector of size numOfCombinations that will
       contain the new probability distribution.
4:     Let nIterations ← the number of iterations in each task.
5:     Let minSize ← the minimum size of the new probability distribution
       beyond which the task is parallelized.
6:     if numOfCombinations >= minSize then
7:         numTasks ← numOfCombinations / nIterations.
8:         for l from 0 to numTasks do
9:             Let blockSize ← numOfCombinations / numTasks the size of the
               iteration.
10:            Let end ← 0.
11:            if l = numTasks - 1 then
12:                end ← numOfCombinations.
13:            else
14:                end ← (l + 1) · blockSize.
15:            end if
16:            Insert a task CalculateCPTData(CPTmult, newProbDistribution,
               newVariablesOrder, l · blockSize, end) into the thread pool to be
               executed in parallel.
17:        end for
18:    else
19:        CalculateCPTData(CPTmult, newProbDistribution, newVariablesOr-
           der, 0, numOfCombinations).
20:    end if
21: end procedure
```

EvaluateCPT is a wrapper function that decides whether the multiplication can be parallelized or not: the multiplication is parallelized if the resulting CPT is big enough to require parallelization. If the multiplication is parallelized then the

needed number of jobs of CalculateCPTData are submitted to the thread pool; if not CalculateCPTData is called directly for a non-parallelized execution.
CalculateCPTData computes the probability values of each combination in the interval of indices specified using the probability distribution of this CPT and the multiplicand CPT.

Algorithm 11 CalculateCPTData. Given the multiplicand CPT CPTmult, the vector in which store the new probability distribution, the variablesOrder of the resulting CPT after the multiplication, the task's first and last index, the new probability distribution is calculated with the following steps:

1: **procedure** CALCULATECPTDATA(CPTmult, distribution, newVariablesOrder, firstIndex, lastIndex)
2: Let *combination* ← The combination of parents' states and this variable's state corresponding to the first index of the block.
3: Be *partialCombination* the vector in which store the partial combination of variables of the multiplier or the multiplicand.
4: **for** i from firstIndex to lastIndex **do**
5: Let $newProb \leftarrow 1$;
6: **for** each variable *var* of this CPT's variablesOrder **do**
7: Be *position* the variable containing the position of *var* in newVariablesOrder.
8: Insert *combination*[*position*] in *partialCombination*.
9: **end for**
10: Let *prob* ← the probability of *partialCombination* in this CPT's distribution.
11: $newProb \leftarrow newProb \cdot prob$;
12: Clear *partialCombination*.
13: **for** each variable *var* of CPTmult's variablesOrder **do**
14: Be *position* the variable containing the position of *var* in newVariablesOrder.
15: Insert *combination*[*position*] in *partialCombination*.
16: **end for**
17: Let *prob* ← the probability of *partialCombination* in CPTmult's distribution.
18: $newProb \leftarrow newProb \cdot prob$;
19: distribution[i] ← *newProb*.
20: Update combination to the next combination.
21: **end for**
22: **end procedure**

4.2 DAG Class

4.2.1 DAG Data Structures

NodeSeparated

NodeSeparated is a simple data structure used to store if a node is d-Separated from a set of nodes. It is composed of an id representing the node and a boolean indicating whether the node is separated or not. This data structure is mostly used during the Lazy Propagation algorithm to identify the relevant potentials.

```
struct {
    NodeId id;
    bool separated;
} NodeSeparated;
```

4.3 UndirectedGraph Class

The first task the UndirectedGraph class is responsible for is triangulating the DAG. While the implementations of both MCS-M and Lex-M are straightforward, given the pseudo code in [17], the MWCH is only theoretically described by [16] so we will give our interpretation of the heuristic:

Algorithm 12 MWCH Triangulation. Given a moral undirected graph G and a vector containing the number of states of each variable in the Bayesian network variablesStatesNumber, the triangulated graph is obtained with the following steps:

1: **procedure** MWCH TRIANGULATION(G, variablesStatesNumber)

```
struct {
    int node, weight;
} NodeWeight;
```

2: Let $nNodes \leftarrow |V| \in G$.
3: Let $G_i \leftarrow G$.
4: Let $addedEdges \leftarrow \emptyset$.
5: Be $cliqueWeights$ a vector of size $nNodes$ of NodeWeight items.

```
6:    Let activeNodes ← ∅.
7:    for i from 0 to nNodes do
8:        activeNodes[i] ← 1.
9:        cliqueWeights[i].node ← i.
10:       Let weight ← 0.
11:       for each v ∈ ne(i) do
12:           weight += variablesStatesNumber[v].
13:       end for
14:       weight += variablesStatesNumber[i].
15:       cliqueWeights[i].weight ← weight.
16:   end for
17:   Order cliqueWeights in ascending order of weight.
18:   Let k ← nNodes - 1.
19:   for i from 0 to nNodes do
20:       Let v ← cliqueWeights[0].node.
21:       if ne(v)_{G_i} ≠ ∅ then
22:           Create a clique with the nodes in ne(v)_{G_i}, storing the newly added
              edges in addedEdges. They will be added at the end of the algorithm.
23:       end if
24:       Remove v from G_i.
25:       activeNodes[v] ← 0.
26:       cliqueWeights[v].weight ← ∞.
27:       Update the clique weight with the remaining vertices using the formula
          defined in the initialization of the vector and sort cliqueWeights in
          ascending order.
28:   end for
29:   for each e ∈ addedEdges do
30:       Add e to G.
31:   end for
32: end procedure
```

Time: $\mathcal{O}(n \cdot (e^2 + n \cdot e))$, Space: $\mathcal{O}(n \cdot e')$, where n is the number of nodes in the graph, e is the number of edges in the graph and e' is the number of added edges to the neighborhood of a node

The idea behind the algorithm is to identify, at each iteration, the clique that has the minimum weight generated by the active nodes. The weight is defined as the sum of the number of states of each variable in the clique. The algorithm starts calculating the weight of the neighbourhood of each node and stores it into a list. At each step, the node with the neighbourhood with the minimum weight is selected,

removed from the list of active nodes and the weights of the neighbourhoods of the other nodes are updated. When a node is selected to generate a clique, the edges needed to make its neighbourhood a clique are stored in a list; the edges are added to the graph at the end of the algorithm.

4.4 JunctionTree Class

4.4.1 JunctionTree Data Structures

MessagePotentialDependencies

In the message passing phase described in Section 2.6.1, the potentials inside the message are stored as a multiplicative decomposition, meaning that they are multiplied only when necessary, i.e. when calculating the posterior marginal of the requested node or anytime a potential is solved during the message passing phase, reducing time of subsequent calculations. The structure used to store the potentials needed is an ordered list. In this way, once all the needed potentials are present in the list, the requested potential is easily computed by multiplying the last potential in the list with all its preceding, the second-last with all its preceding, etc. At the end, the direct parents of the requested node are solved and can be used to compute the requested posterior.
Additionally, two sets of ids are required, one for keeping track of which potentials are already inserted into the list, and one for keeping track of which ancestor potentials are still missing. In this way it is easily determined which potentials to insert into the list of dependencies and which not. When a potential is inserted into potentialDependencies, its id is removed from pendingDependencies, added to solvedDependencies and its set of potentials required is added to pendingDependencies.
During the discussion of this thesis, we will refer to all the potentials needed to solve the potential in the structure as dependencies.

```
1 struct {
2     CPT potential;
3     CPT[] potentialDependencies;
4     Set pendingDependencies;
5     Set solvedDependencies;
6 } MessagePotentialDependencies;
```

Message

Message is the structure containing the potentials to pass during the message passing phase. Each message carries the information about which clique generated it, in this way it is easier to avoid sending the same potentials to clique that sent them. Additionally, each message uses isMessageNew to indicate whether it is new or not in case of multiple executions of the algorithm. If it is new, then the variable position indicates where this message will be inserted into the list of messages of the clique that receives it, substituting the old one. The potentials passed within a message are divided in two lists: potentialDependencies contains all non-instantiated potentials and instantiatedPotentialDependencies contains the instantiated ones.

```
struct {
    CliqueId sourceClique;
    MessagePotentialDependencies[] potentialDependencies;
    MessagePotentialDependencies[] instantiatedPotentialDependencies;
    bool isMessageNew;
    int position;
} Message;
```

4.5 LazyPropagation Class

4.5.1 LazyPropagation Class Functions

Lazy Propagation Algorithm

The Lazy Propagation algorithm in Section 2.6 discusses the operations at a high level without giving insights of how to efficiently implement the algorithm. We will give an in depth description of the adjustments made to the original algorithm in order to obtain better CPU and RAM performance.
Since the Lazy Propagation presents many modifications from the algorithm presented in [3], for each algorithm we will specify the time and space complexity. Below we will introduce the notation used in the calculations for time and space complexities:

- c: number of cliques in the junction tree
- c_p: number of potentials in the clique
- n: number of variables in the network
- m: number of messages received by the clique

- m_p: number of non instantiated potentials contained in the message
- m_i: number of instantiated potentials contained in the message
- mc_p: number of non instantiated potentials contained in the clique and in all the messages received by the clique
- p_v: number of variables contained in the potential
- p_a: number of ancestors of a variable
- p_i: number of instantiated potentials needed to update a potential

Algorithm 13 CollectEvidence. Given a clique C and its parent clique pa(C), CollectEvidence is recursively called on all its child nodes, performing the following operations:

1: **procedure** CollectEvidence(C, pa(C))
2: Be $MessageToSend$ the message to be sent to the parent clique.
3: Let $MessagePotentialsDependencies \leftarrow$ Potentials present in the clique and the potentials received from all the separators connected to the clique.
4: Let $MessagePotentialsDependencies\_S \leftarrow$ Potentials present in the clique and received from all the separators connected to the clique, except those received through S.
5: Let $InstantiatedPotentials \leftarrow$ Instantiated potentials present in the clique and those received from all the separators connected to the clique.
6: Be S the separator between C and pa(C).
7: **for** $S' \in$ ne(C) $\setminus$ S **do**
8: Be C_c the child of C connected to S'.
9: Let $MessageReceived \leftarrow$ CollectEvidence(C_c, C).
10: Save $MessageReceived$ in this clique's messages list.
11: **end for**
12: **if** pa(C) != -1 **then**
13: $MessageToSend \leftarrow$ CollectEvidenceMessageCreation(C, $MessagePotentialsDependencies$, $MessagePotentialsDependencies\_S$, $InstantiatedPotentials$, S).
14: **end if**
15: Return $MessageToSend$
16: **end procedure**

Time: $\mathcal{O}\left(p\_a^5 \cdot p\_i \cdot p\_v \cdot mc\_p \cdot c\right)$, Space: $\mathcal{O}\left(p\_a \cdot mc\_p \cdot c\right)$

The structure of our CollectEvidence is almost identical to the one in [3] (the absorption phase is inside the message creation phase), we added three vectors to facilitate the operations of locating the relevant potentials. The list of potentials used when creating a message is different from the list used to solve a potential, thus reducing the length of the iteration in the former case. These three lists are used in CollectEvidenceMessageCreation to create the message and subsequently to solve the potentials in the clique in case anyone is solvable.

Algorithm 14 DistributeEvidence. Given a clique C, its parent clique pa(C), the list of evidence nodes L and the message M from pa(C), DistributeEvidence is recursively called on all its child nodes, performing the following operations:

1: **procedure** DISTRIBUTEEVIDENCE(C, pa(C), M)
2: Be $MessageToSend$ the message to be sent to the child clique.
3: Let $NotInstantiatedPotentials \leftarrow$ Not instantiated potentials present in the clique.
4: Let $InstantiatedPotentials \leftarrow$ Instantiated potentials present in the clique.
5: Be S the separator between C and pa(C).
6: Let $SeparatedNodes \leftarrow$ All the variables in this clique and those received in the messages.
▷ It is a vector of NodeSeparated.
7: Save M in this clique's messages list.
8: **for** $S' \in$ ne(C) $\setminus$ S **do**
9: Be C_c the child of C connect to S'.
10: IsDSeparated($SeparatedNodes$, C_c, L).
11: $MessageToSend \leftarrow$ DistributeEvidenceMessageCreation(C, $SeparatedNodes$, $NotInstantiatedPotentials$, $InstantiatedPotentials$, S).
12: DistributeEvidence(C_c, C, $MessageToSend$).
13: **end for**
14: **end procedure**

Time: $\mathcal{O}\left(p\_a^5 \cdot p\_i \cdot p\_v \cdot mc\_p \cdot c\right)$, Space: $\mathcal{O}\left(p\_a \cdot mc\_p \cdot c\right)$

Compared to the DistributeEvidence defined in [3], we compute which nodes, either contained in this clique or received through messages, are d-Separated from the nodes in the child clique before the absorption phase in order to reuse the same list for all the child cliques. This saves a lot of allocations and deallocations of memory improving code efficiency. The information about which nodes are d-Separated and which not is recalculated for each clique.
As vastly discussed, the Lazy Propagation algorithm is nothing more than a

recursive exploration of a tree done two times. Since we wanted to improve as much as possible the computational efficiency of the algorithm, the first improvement we introduced was parallelizing both the collect and distribute evidence functions, i.e. parallelizing a recursive exploration of a tree.
When parallelizing a recursive exploration of a tree using a thread pool, the first problem is handling the submission of jobs so to avoid dead-locks during the execution. Parallelizing a recursive algorithm can create a dead-lock if a job submits another job to the thread pool but no threads are available to perform the new job. In this situation the parent job waits indefinitely. A possible solution is to add new threads the thread pool when a new job is submitted and no threads are available; this solution, however, has two drawbacks: firstly, creating new threads is not immediate and requires time and memory by the operating system to allocate the resources. Creating too many threads could end up saturating system resources. The second drawback is context switching: creating too many threads increases the number of context switches the scheduler performs leading to poor performance.
Since this solution is not ideal for resources utilization, we will use a different approach: the recursive jobs will generate the list of jobs associated with each child; those jobs will be submitted to the thread pool until there are available threads. Once the thread pool is full, the jobs will be executed by the thread that created them. In this way dead-locks do not occur but the main thread is left unused. In order to utilize also the main thread we can leave it on wait over a queue of jobs. Then once the thread pool is filled, the main thread starts executing the first job in queue and continues until it is empty. In this way we are able to visit the junction tree in parallel, without any dead-lock, using as many resources are allocated to the thread pool to speed up the message passing phase in large trees. This solution still has margin for improvement and it will be discussed in Section 6.2.3.
In order to determine the end of the recursive algorithms we will use a variable *End*. *End* is set to True when CollectEvidenceParallel or DistributeEvidenceParallel has terminated its execution in the root clique. This will be used to end the while loop in the main thread that is extracting and executing jobs from the queue.
In the next algorithms, we will refer to the thread pool as TP.

Algorithm 15 LazyPropagationParallel. Given a root clique C_R the parallel Lazy Propagation algorithm is executed using two wrapper functions that performs the collect and distribute phase recursively on the junction tree:

```
1: procedure LazyPropagationParallel
2:     CollectEvidenceParallelWrapper(C_R).
3:     DistributeEvidenceParallelWrapper(C_R).
4: end procedure
```

Algorithm 16 CollectEvidenceParallelWrapper. Given a clique root C_R, a queue of jobs to be executed CollectEvidenceParallelJobsToPerform, a set containing the results of the jobs completed CollectEvidenceParallelJobsCompleted, a map containing the job being run by the main thread and its results CollectEvidenceParallelJobsRunning and a variable F indicating that the execution of CollectEvidenceParallel is completed, CollectEvidenceParallel is recursively called on all its child nodes performing the following operations:

```
1: procedure CollectEvidenceParallelWrapper(C_R)
2:     CollectEvidenceParallel(C_R, -1).
3:     while F = False do
4:         Wait until there is a job in CollectEvidenceParallelJobsToPerform or F
   is True.
5:         Let Job ← first job in CollectEvidenceParallelJobsToPerform.
6:         if Job ∉ CollectEvidenceParallelJobsCompleted then
7:             Execute Job.
8:             Put Job in CollectEvidenceParallelJobsRunning with its result.
9:         end if
10:    end while
11: end procedure
```

Time: $\mathcal{O}\left(p\_a^5 \cdot p\_i \cdot p\_v \cdot mc\_p \cdot c\right)$, Space: $\mathcal{O}\left(p\_a \cdot mc\_p \cdot c\right)$

Algorithm 17 DistributeEvidenceParallelWrapper. Given a clique root C_R, a queue of jobs to be executed DistributeEvidenceParallelJobsToPerform, a set containing the results of the jobs completed DistributeEvidenceParallelJobsCompleted, a map containing the job being run by the main thread and its results DistributeEvidenceParallelJobsRunning and a variable F indicating that the execution of DistributeEvidenceParallel is completed, DistributeEvidenceParallel is recursively called on all its child nodes performing the following operations:

```
1: procedure DistributeEvidenceParallelWrapper(C_R)
2:     DistributeEvidenceParallel(C_R, -1).
3:     while F = False do
4:         Wait until there is job in DistributeEvidenceParallelJobsToPerform or
   F is True.
5:         Let Job ← first job in DistributetEvidenceParallelJobsToPerform.
6:         if Job ∉ DistributeEvidenceParallelJobsCompleted then
7:             Execute Job.
```

```
            Put Job in DistributeEvidenceParallelJobsRunning with its result.
        end if
    end while
end procedure
```

Time: $\mathcal{O}(p\_a^5 \cdot p\_i \cdot p\_v \cdot mc\_p \cdot c)$, Space: $\mathcal{O}(p\_a \cdot mc\_p \cdot c)$

Both Algorithm 16 and Algorithm 17 are executed in the main thread of the library which is not part of the thread pool, meaning it cannot execute jobs and would sleep during the execution of the whole inference algorithm. By using this approach we are able to take advantage of the main thread's resources instead of wasting it. At each iteration of the while loop, the main thread checks whether there is any job available in the queue and extracts it. Once the extracted job is completed, its result is made available to the thread waiting for it, allowing it to continue its execution.

Algorithm 18 CollectEvidenceParallel. Given a clique C, its parent clique pa(C), a queue of jobs to be executed CollectEvidenceParallelJobsToPerform, a set containing the results of the jobs completed CollectEvidenceParallelJobsCompleted, a map containing the job being run by the main thread and its results CollectEvidenceParallelJobsRunning and a variable F indicating that the execution of CollectEvidenceParallel is completed, CollectEvidenceParallel is recursively called on all its child nodes, performing the following operations:

```
procedure CollectEvidenceParallel(C, pa(C))
    Be MessageToSend the message to be sent to the parent clique.
    Let MessagePotentialsDependencies ← Potentials present in the clique
    and the potentials received from all the separators connected to the clique.
    Let MessagePotentialsDependencies_S ← Potentials present in the clique
    and received from all the separators connected to the clique, except those
    received through S.
    Let InstantiatedPotentials ← Instantiated potentials present in the clique
    and those received from all the separators connected to the clique.
    Be S the separator between C and pa(C).
    Let LocalJobs ← ∅.                    ▷ Queue of jobs created by the current
CollectEvidenceParallel.
    for S' ∈ ne(C) \ S do
        Be C_c the child of C connect to S'.
        Let Job ← CollectEvidenceParallel(C_c, C).
```

```
11:         if AvailableThreadsInTP > 0 then
12:             Submit Job to TP.
13:         else
14:             Insert Job into CollectEvidenceParallelJobsToPerform.
15:             Insert Job into LocalJobs.
16:         end if
17:     end for
18:     for Job ∈ LocalJobs do
19:         if Job ∉ CollectEvidenceParallelJobsRunning then
20:             Insert Job into CollectEvidenceParallelJobsCompleted.
21:             Run Job and save the message in this clique's messages list.
22:         else
23:             Retrieve message from CollectEvidenceParallelJobsRunning and
                store it in this clique's messages list.
24:         end if
25:     end for
26:     if pa(C) != -1 then
27:         MessageToSend ← CollectEvidenceMessage-
            Creation(C, MessagePotentialsDependencies,
            MessagePotentialsDependencies_S, InstantiatedPotentials, S).
28:     else
29:         F = True.
30:     end if
31:     Return MessageToSend
32: end procedure
```

Time: $\mathcal{O}\left(p\_a^5 \cdot p\_i \cdot p\_v \cdot mc\_p \cdot c\right)$, Space: $\mathcal{O}\left(p\_a \cdot mc\_p \cdot c\right)$

Comparing the parallel version of the collect evidence to its serial version in Algorithm 13, the main difference is that the whole phase is divided into two sub-phases: in the first one, the CollectEvidenceParallel jobs are created for each children and it is determined whether they can be executed by the thread pool or not. In the second phase the messages of the jobs sent to the thread pool, and those executed by the main thread, are retrieved and the jobs that could not be executed by the thread pool are executed in the current thread. The same changes are applied also to DistributeEvidenceParallel but in this case there is not a message to retrieve from the job but we just wait for the completion of the job either from the main thread or the current thread.

Algorithm 19 DistributeEvidenceParallel. Given a clique C, its parent clique pa(C), the list of evidence nodes L, the message M from pa(C), a queue of jobs to be executed DistributeEvidenceParallelJobsToPerform, a set containing the results of the jobs completed DistributeEvidenceParallelJobsCompleted, a map containing the job being run by the main thread and its results DistributeEvidenceParallelJobsRunning and a variable F indicating that the execution of DistributeEvidenceParallel is completed, DistributeEvidenceParallel is recursively called on all its child nodes, performing the following operations:

1: **procedure** DISTRIBUTEEVIDENCEPARALLEL(C, pa(C), M)
2: Be $MessageToSend$ the message to be sent to the child clique.
3: Let $NotInstantiatedPotentials \leftarrow$ Non instantiated potentials present in the clique.
4: Let $InstantiatedPotentials \leftarrow$ Instantiated potentials present in the clique.
5: Be S the separator between C and pa(C).
6: Let $SeparatedNodes \leftarrow$ All the variables in this clique and those received in the messages.
7: Let $LocalJobs \leftarrow \emptyset$.
8: Save M in this clique's messages list.
9: **for** $S' \in$ ne(C) $\setminus S$ **do**
10: Be C_c the child of C connect to S'.
11: IsDSeparated($SeparatedNodes$, C_c, L).
12: $MessageToSend \leftarrow$ DistributeEvidenceMessageCreation(C, $SeparatedNodes$, $NotInstantiatedPotentials$, $InstantiatedPotentials$, S).
13: Let $Job \leftarrow$ DistributeEvidenceParallel(C_c, C, $MessageToSend$).
14: **if** AvailableThreadsInTP > 0 **then**
15: Submit Job to TP.
16: **else**
17: Insert Job into DistributeEvidenceParallelJobsToPerform.
18: Insert Job into $LocalJobs$.
19: **end if**
20: **end for**
21: **for** $Job \in LocalJobs$ **do**
22: **if** $Job \notin$ DistributeEvidenceParallelJobsRunning **then**
23: Insert Job into DistributeEvidenceParallelJobsCompleted.
24: Run Job.
25: **end if**
26: **end for**
27: **if** pa(C) = -1 **then**
28: F = True.
29: **end if**
30: **end procedure**

Time: $\mathcal{O}(p\_a^5 \cdot p\_i \cdot p\_v \cdot mc\_p \cdot c)$, Space: $\mathcal{O}(p\_a \cdot mc\_p \cdot c)$

Marginalization Process

The marginalization algorithm described in [3] is not a feasible solution, even for small sized networks, because it requires that the variables not present in the separator are marginalized when passing the message. The situation that makes this operation unfeasible is when an incomplete variable is marginalized. This is not a problem when that variable introduces only one other variable to the list of parents of the current variable; the problem arises when two ore more variables are added to the list, leading to an exponential increase of the size of the variable's distribution and ultimately causing an overflow of memory, if this process is repeated multiple times. We show an example of this problem using the junction tree of Figure 4.2.

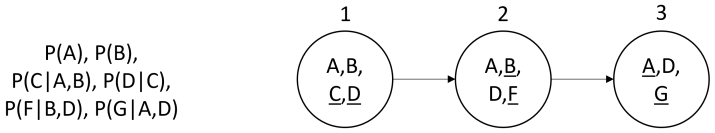

Figure 4.2: Example junction tree. The underlined letters represent the variables present in the clique. On the left, the variables' CPT definition.

Choosing clique 3 as the root clique, when passing the message from clique 1 to 2 during the collection of evidence, P(D|C) is subject to the following marginalization process:

$$P(D|C) \rightarrow P(D|C) \cdot P(C|A,B) = P(D|A,B) \tag{4.3}$$

In clique 2, variable D is conditioned on both A and B potentially increasing the size of the prior distribution of D. On small networks these operations are unlikely to create distributions too big to be stored in central memory but the size of the distribution could slow down the whole inference algorithm when multiplying or marginalizing variables. On bigger networks, if these operations are repeated enough times, the prior distribution will eventually become too big to be stored in RAM, slowing down the algorithm due to memory swapping, or in the worst case, the distribution could become too large to be indexed leading to an overflow of the index used to iterate over the distribution (the maximum size of a distribution is 2^{64} on a 64bit computer).
As described in Section 2.6, the messages used by the Lazy propagation algorithm contains a multiplicative decomposition of the potentials, i.e. the potentials are not multiplied until requested reducing memory usage and saving time during operations. This alone is not sufficient to obtain a result in reasonable time. Our solution is to use the MessagePotentialDependencies data structure defined in Section 4.4.1 to store the list of potentials needed to solve one potential and resolving it only when all dependencies have been satisfied.

Our marginalization process is reduced to just remove the potentials not in the separator from the list of potentials passed in the message but the actual marginalization on a variable is not carried out.

Caching Potentials

One of the property of the Lazy Propagation algorithm is the possibility to calculate the posterior marginal of any of the variables contained in the clique after the distribute evidence phase is concluded. This implies that the same variable can be calculated in many cliques. Considering this property from a computational cost point of view, it is possible that the same posterior is calculated multiple times during the message passing phase. To prevent this problem, we use two list of potentials (one list is used to store the potentials calculated without evidence introduced and the potentials that needs evidence from above; the second list is used to store the potentials calculated with evidence from below) to store each potential calculated during the message passing phase. Thus during the message passing phase only the potentials that are not already calculated and stored in the list are calculated. This list can also be used to retrieve parents needed for solving a potential, although it has not been received through a message yet, speeding up the algorithm.
Two normal lists are not enough for a parallel execution of the Lazy Propagation because they introduce race conditions when checking whether a potential is already calculated or not. To guarantee consistency in the execution we used an implementation of concurrent vectors that handles concurrent accesses to each element from multiple threads without requiring the use of an additional mutex.
The two lists will be referred as *ReadyPotentials* and *ReadyPotential_Evidence* respectively the list of potentials calculated without evidences or with evidences from above, and the list of potentials calculated with evidences from below.

Caching Evidence Information

One of the aspects that makes inferencing with observations so difficult is due to determine which observations have an impact on the posterior of a variable and which variables need parents that have been updated with observations. This boils down to perform tree searches on the DAG and determine:

- For each observed node the list of ancestors reached.
- For each variable reached by an evidence, the list of descendants that need the updated parent.

Then for each variable is stored the list of evidences from below and the list of evidences from above that influences them, facilitating the task of propagating evidence.

The lists will be referred as $EvidencesFromBelow$ and $EvidencesFromAbove$. In addition to these two lists, we use a list ($PotentialsToUpdate$) to easily keep track of which potentials need to be updated.

Message Creation

Our CollectEvidence and DistributeEvidence have two different message creation algorithms because, despite the operations performed in both phases are the same, the order in which they are performed is different, thus requiring two separate definitions.

Algorithm 20 CollectEvidenceMessageCreation. Given a clique C, a list of MessagePotentialDependencies RS, a list of MessagePotentialDependencies MessagePotentialsDependencies, a list of MessagePotentialDependencies MessagePotentialsDependencies_S, a list of MessagePotentialDependencies InstantiatedPotentials and the separator S, the list of evidence nodes L, the message to send is created performing the following operations:

1: **procedure** CollectEvidenceMessageCreation(C, RS, MessagePotentialsDependencies, MessagePotentialsDependencies_S, InstantiatedPotentials, S)
2: Let $SeparatedNodes \leftarrow$ All the variables in this clique and those received in the messages.
3: Let $RS \leftarrow$ MessagePotentialsDependencies_S.
4: IsDSeparated($SeparatedNodes$, C_c, L).
5: FindRelevantPotentials(RS, $SeparatedNodes$).
6: Order RS, MessagePotentialsDependencies, MessagePotentialsDependencies_S in descending order.
7: Return MessageCreation(C, RS, MessagePotentialsDependencies, MessagePotentialsDependencies_S, InstantiatedPotentials, InstantiatedPotentials, S).
8: **end procedure**

Time complexity: $\mathcal{O}(p\_a^5 \cdot p\_i \cdot p\_v \cdot mc\_p)$, Space complexity: $\mathcal{O}(p\_a \cdot mc\_p)$

Both CollectEvidenceMessageCreation and DistributeEvidenceMessageCreation are wrapper functions for finding relevant potentials and the message creation phase. The FindRelevantPotentials function uses the same algorithm defined in [3] for finding relevant potentials, removing potentials that are not required for the message from RS.

Algorithm 21 DistributeEvidenceMessageCreation. Given a clique C, a list of NodeSeparated NS, a list of MessagePotentialDependencies PotentialsNotInstantiated, a list of MessagePotentialDependencies PotentialsInstantiated, and the separator S, the message to send is created performing the following operations:

1: **procedure** DISTRIBUTEEVIDENCEMESSAGECREATION(C, NS, PotentialsNotInstantiated, PotentialsInstantiated, S)
2: Let $RS \leftarrow$ PotentialsNotInstantiated.
3: Let $MessagePotentials \leftarrow$ PotentialsNotInstantiated.
4: Let $MessagePotentials\_S \leftarrow$ PotentialsNotInstantiated.
5: Let $InstantiatedPotentials \leftarrow$ PotentialsInstantiated.
6: Let $InstantiatedPotentials\_S \leftarrow$ PotentialsInstantiated.
7: Add to $MessagePotentials$ all not instantiated potentials received through messages and add to RS and $MessagePotentials\_S$ all not instantiated potentials received through messages except those receive through S.
8: Add to $InstantiatedPotentials$ all instantiated potentials received through messages and add to $InstantiatedPotentials\_S$ all instantiated potentials received through messages except those receive through S.
9: FindRelevantPotentials(RS, NS).
10: Order RS, $MessagePotentials$, $MessagePotentials\_S$ in descending order.
11: Return MessageCreation(C, RS, $MessagePotentials$, $MessagePotentials\_S$, $InstantiatedPotentials$, $InstantiatedPotentials\_S$, S).
12: **end procedure**

Time complexity: $\mathcal{O}\left(p\_a^5 \cdot p\_i \cdot p\_v \cdot mc\_p\right)$, Space complexity: $\mathcal{O}\left(p\_a \cdot mc\_p\right)$

During the message creation phase, we decide whether a new message is needed (in case the Lazy Propagation algorithm is executed multiple times). Doing the check at this stage and not when the message is received allows to save time if the old message is still valid. The old message may become invalid when the target node is changed, because the list of barren variable changes, and when an evidence is introduced or removed from the network.
Function UpdateMessage checks whether the potentials of the current execution of the Lazy Propagation, after the marginalization process, will be the same as the old one. If yes, then checks whether also the instantiated potentials of the current execution are the same as those of the old one. If they are different then a new message is created otherwise the old message will be used.

Algorithm 22 MessageCreation. Given a clique C, a list of MessagePotentialDependencies RsPrime, a list of MessagePotentialDependencies MessagePotentialsDependencies, a list of MessagePotentialDependencies MessagePotentialsDependencies_S, a list of MessagePotentialDependencies InstantiatedPotentials, a list of MessagePotentialDependencies InstantiatedPotentials_S and the separator S, the message to send is created by performing the following operations:

1: **procedure** MessageCreation(C, RsPrime, MessagePotentialsDependencies, MessagePotentialsDependencies_S, InstantiatedPotentials, InstantiatedPotentials_S, S)
2: Let $C_c \leftarrow$ Child clique of C connected to S.
3: Let $MessageToSend = \emptyset$.
4: Let $OldMessage \leftarrow$ Old message contained in C_c sent by C.
5: Let $OldPotentials \leftarrow$ The list of potentials contained in $OldMessage$.
6: **if** $OldMessage = \emptyset$ Or ($OldMessage \neq \emptyset$ And UpdateMessage(RsPrime, InstantiatedPotentials_S, $OldMessage$, S)) **then**
7: UpdateParentPotentialsInMessages(C, InstantiatedPotentials).
8: SolvePotentials(C, RsPrime, MessagePotentialsDependencies, MessagePotentialsDependencies_S, InstantiatedPotentials, InstantiatedPotentials_S, S).
9: MarginalizePotential(C, RsPrime, S).
10: Insert RsPrime and InstantiatedPotentials_S into $MessageToSend$.
11: **if** $OldMessage \neq \emptyset$ **then** $MessageToSend$.position $\leftarrow$ $OldMessage$.position.
12: **end if**
13: **else**
14: $MessageToSend \leftarrow OldMessage$.
15: $MessageToSend$.isMessageNew $\leftarrow$ False.
16: UpdateParentPotentialsInMessages(C, InstantiatedPotentials).
17: **for** $\phi \in$ RsPrime **do**
18: **if** |MessagePotentialsDependencies| > 1 **then**
19: SolvePotential(C, MessagePotentialsDependencies, ϕ).
20: **end if**
21: **if** InstantiatedPotentials $\neq \emptyset$ **then**
22: Let $Id_\phi \leftarrow \phi$'s id.
23: UpdateAndStoreParentPotential(C, Id_ϕ, InstantiatedPotentials, True).
24: **end if**
25: **end for**
26: **end if**

27: Return $MessageToSend$.
28: **end procedure**

Time complexity: $\mathcal{O}(p\_a^5 \cdot p\_i \cdot p\_v \cdot mc\_p)$, Space complexity: $\mathcal{O}(p\_a \cdot mc\_p)$

If a new message is needed, the message creation phase starts by updating the posterior marginal of the variables, received through messages, with the evidences introduced using the process described in Section 2.4.1. This preliminary parents update has the purpose of updating parents that may be needed to calculate the message of this clique or to solve some of the potentials contained in the clique. If the old message is reused, the solving and updating potentials phases are repeated with the information contained in the old message since they are still valid.

Algorithm 23 UpdateParentPotentialsInMessages. Given a clique C and a list of MessagePotentialDependencies InstantiatedPotentials, the parents are updated by performing the following operations:

1: **procedure** UpdateParentPotentialsInMessages(C, InstantiatedPotentials)
2: **if** InstantiatedPotentials $\neq \emptyset$ **then**
3: **for** $M \in$ C's messages list **do**
4: **for** $\phi \in M$'s list of non-instantiated potentials **do**
5: UpdateAndStoreParentPotential(C, ϕ, InstantiatedPotentials, true).
6: **end for**
7: **end for**
8: **end if**
9: **end procedure**

Time: $\mathcal{O}(p\_a^2 \cdot p\_i \cdot p\_v \cdot m \cdot m\_p)$, Space: $\mathcal{O}(p\_a)$

UpdateParentPotentialsInMessages is a wrapper function that uses UpdateAndStoreParentPotential to update all potentials contained in the messages received by the clique.

Algorithm 24 UpdateAndStoreParentPotential. Given a clique C, a potential ϕ, a list of MessagePotentialDependencies InstantiatedPotentials and a boolean S, the parents are updated by performing the following operations:

1: **procedure** UPDATEANDSTOREPARENTPOTENTIAL(C, ϕ, InstantiatedPotentials, S)
2: Let $Id_\phi \leftarrow \phi$'s id.
3: **if** $|EvidencesFromBelow[Id_\phi]| > 0$ And $\phi \in ReadyPotentials$ And $\phi \notin ReadyPotential\_Evidence$ **then**
4: Let $\phi_C \leftarrow ReadyPotentials[Id_\phi]$.
5: Be $SolvedInstantiatedPotentials$ an empty list of potentials.
6: Be $InstantiatedPotentialsNeeded$ an empty list of MessagePotentialDependencies.
7: Let $VariablesNeeded \leftarrow EvidencesFromBelow[Id_\phi]$. ▷ $VariablesNeeded$ contains the list of observations from below that influences ϕ.
8: **for** $\phi_{inst} \in$ InstantiatedPotentials **do**
9: Let $Id_{\phi_{inst}} \leftarrow \phi_{inst}$'s id.
10: **if** $Id_{\phi_{inst}} \in VariablesNeeded$ **then**
11: Insert ϕ_{inst} into $InstantiatedPotentialsNeeded$.
12: Remove ϕ_{inst} from $VariablesNeeded$.
13: **end if**
14: **end for**
15: **if** $VariablesNeeded = \emptyset$ **then**
16: UpdateAndSolveInstantiatedPotentials(C, Id_ϕ, $InstantiatedPotentialsNeeded$, $SolvedInstantiatedPotentials$).
17: **end if**
18: **if** $|SolvedInstantiatedPotentials| = |EvidencesFromBelow[Id_\phi]|$ **then**
19: UpdateParentPotentialWithEvidence(ϕ, $SolvedInstantiatedPotentials$, S).
20: **end if**
21: **end if**
22: **end procedure**

Time: $\mathcal{O}(p\_a^2 \cdot p\_i \cdot p\_v)$, Space: $\mathcal{O}(p\_a)$

UpdateAndStoreParentPotential starts by selecting the observations that influence ϕ and then selects the instantiated potentials needed, corresponding to the list, from the list of instantiated potentials provided to the function. If all the observations needed are present, then the instantiated potentials' parents are

solved by UpdateAndSolveInstantiatedPotentials, i.e. after UpdateAndSolveInstantiatedPotentials, the instantiated potentials will depend only on ϕ. Then, if all the instantiated potentials are now depending on ϕ, it can be updated with the evidences. The last check is needed since, when propagating evidences upward, we need to marginalize all other potentials that are not ϕ. Hence, it can happen that some of the other potentials are not yet calculated and the operations cannot be executed.

Algorithm 25 UpdateAndSolveInstantiatedPotentials. Given a clique C, a potential's Id_ϕ, a list of MessagePotentialDependencies InstantiatedPotentials and a list of potentials InstantiatedPotentialsSolved, the instantiated potentials are updated by performing the following operations:

```
1: procedure UpdateAndSolveInstantiatedPotentials(C, Id_φ, InstantiatedPotentials, InstantiatedPotentialsSolved)
2:     for φ_inst ∈ InstantiatedPotentials do
3:         UpdateInstantiatedPotential(C, Id_φ, φ_inst).
4:         if IsInstantiatedPotentialSolved(Id_φ, φ_inst) then
5:             SolveInstantiatedPotential(Id_φ, φ_inst).
6:             Insert φ_inst.potential into InstantiatedPotentialsSolved.
7:         end if
8:     end for
9: end procedure
```

Time: $\mathcal{O}\left(p\_a^2 \cdot p\_i \cdot p\_v\right)$, Space: $\mathcal{O}\left(p\_a\right)$

The operations inside UpdateAndSolveInstantiatedPotentials are divided into two phases: an updating phase and a solving phase. During the updating phase, the list of dependencies of ϕ is completed with the potentials contained in the messages received in this clique and the potentials contained in the clique. During the solving phase, the instantiated potential is updated using the dependencies in order to make it depend only on the variable which is influenced by the observations.

Algorithm 26 UpdateInstantiatedPotential. Given a clique C, a potential's Id_ϕ and an instance of MessagePotentialDependencies ϕ_{inst}, the instantiated potential's dependencies are updated by performing the following operations:

```
1: procedure UpdateInstantiatedPotential(C, Id_φ, φ_inst)
2:     Order φ_inst.potentialDependencies in descending order.
```

```
    Let φ_D ← GetCPTParentsInClique(C, Id_φ, φ_inst).
    while φ_D ≠ ∅ do
        Add φ_D to φ_inst.potentialDependencies.
        φ_D ← GetCPTParentsInClique(C, Id_φ, φ_inst).
    end while
end procedure
```

Time: $\mathcal{O}(p\_a \cdot m \cdot m\_p)$, Space: $\mathcal{O}(p\_a)$

During the updating phase, the potentials needed to update an instantiated potential are selected by GetCPTParentsInClique and then are added to ϕ.potentialDependencies for a later use, until there are no more potentials needed in this clique indicated by an empty potential returned by GetCPTParentsInClique.

Algorithm 27 GetCPTParentsInClique. Given a clique C, a potential's Id_ϕ and an instance of MessagePotentialDependencies ϕ_{inst}, the potentials needed to update one potential's dependencies are found by performing the following operations:

```
1: procedure GetCPTParentsInClique(C, Id_φ, φ_inst)
2:     Be φ_R an empty instance of MessagePotentialDependencies.  ▷ It will be the potential returned by the function.
3:     Let Id_φR ← -1.
4:     Let Id_φinst ← φ_inst.potential's id.
5:     for M ∈ C's messages list do
6:         for φ ∈ M do
7:             Let Id_φM ← φ.potential's id.
8:             if Id_φM ≠ Id_φinst And Id_φM ≠ Id_φ And Id_φM > Id_φR And Id_φM ∉ φ_inst.solvedDependencies And Id_φM ∈ φ_inst.pendingDependencies then
9:                 φ_R ← φ.
10:                Id_φR ← Id_φM.
11:            end if
12:        end for
13:    end for
14:    for φ ∈ C's potentials list do
15:        Let Id_φM ← φ's id.
16:        if Id_φM ≠ Id_φinst And Id_φM ≠ Id_φ And Id_φM > Id_φR And Id_φM ∉ φ_inst.solvedDependencies And Id_φM ∈ φ_inst.pendingDependencies then
17:            φ_R ← φ.
18:            Id_φR ← Id_φM.
19:        end if
20:    end for
```

```
21:     if φ_R.potential ≠ ∅ then
22:         if Id_φ ∉ anc(Id_φR) And Id_φ ∈ ReadyPotentials then
23:             φ_R ← ReadyPotentials[Id_φR].
24:         end if
25:     end if
26:     Return φ_R.
27: end procedure
```

Time: $\mathcal{O}(m \cdot m\_p)$, Space: $\mathcal{O}(1)$

GetCPTParentsInClique selects the potential with the highest id that has not already been added to ϕ_{inst}'s potentialDependencies list, and that is a potential required, from the potentials received through messages and the potentials contained in this clique. If a potential has been selected, we check whether ϕ is an ancestor of ϕ_R or not. If not it means that it will be marginalized during the solving phase and so we select, if available, the solved potential from *ReadyPotentials* reducing the time needed for calculations.

Algorithm 28 IsInstantiatedPotentialSolved. Given a potential's Id_ϕ and an instance of MessagePotentialDependencies ϕ_{inst}, ϕ_{inst} is deemed solvable by performing the following operations:

```
1: procedure IsInstantiatedPotentialSolved(Id_φ, φ_inst)
2:     Let PendingDependencies ← φ_inst.pendingDependencies.
3:     for V ∈ PendingDependencies do
4:         if V ≠ Id_φ And V ≠ φ_inst.potential's id And V ∈ ReadyPotentials
   then
5:             PendingDependencies = PendingDependencies \ {V}.
6:         end if
7:     end for
8:     if (|PendingDependencies| = 1 And Id_φ ∈ PendingDependencies) Or
   PendingDependencies = ∅ then
9:         Return True.
10:    else
11:        Return False.
12:    end if
13: end procedure
```

IsInstantiatedPotentialSolved checks whether ϕ_{inst} is ready to be used for updating the posterior of ϕ by performing some simple set operations. ϕ_{inst} is ready when,

after the marginalization process using the potentials added to its potentialDependencies list, the only variables remaining are ϕ and ϕ_{inst}. When all the instantiated potentials are ready then the parent's posterior marginal can be updated with the evidences.

Algorithm 29 SolveInstantiatedPotential. Given a potential's Id_ϕ, an instance of MessagePotentialDependencies ϕ_{inst}, ϕ_{inst} is solved by performing the following operations:

1: **procedure** SOLVEINSTANTIATEDPOTENTIAL(Id_ϕ, ϕ_{inst})
2: Order ϕ_{inst}.potentialDependencies in descending order.
3: Be $Variables$ a list of potentials id.
4: Let $N \leftarrow |\phi_{inst}.\text{potentialDependencies}|$.
5: Let $VarToKeep \leftarrow \text{Id}_\phi$.
6: Let $NextVar \leftarrow$ -1.
7: **for** i from N - 1 to 0 **do**
8: Be ϕ_i an empty potential.
9: Be ϕ_{iS} an empty potential.
10: Let $Id_{\phi_i} \leftarrow \phi_{inst}$.potentialDependencies[$i$]'s id.
11: **if** $Id_{\phi_i} \neq \text{Id}_\phi$ **then**
12: Let $Variables \leftarrow \phi_{inst}$.potentialDependencies[$i$]'s variables.
13: **for** $V \in Variables$ **do**
14: **if** $V \neq VarToKeep$ And $V \in ReadyPotentials$ **then**
15: ϕ_{inst}.potentialDependencies[i] $\leftarrow$ ϕ_{inst}.potentialDependencies[i] $\cdot$ $ReadyPotentials[V]$.
16: Marginalize out V from ϕ_{inst}.potentialDependencies[i].
17: **end if**
18: **end for**
19: Let $Variables \leftarrow \phi_{inst}$.potentialDependencies[$i$]'s variables.
20: **if** $|Variables| = 2$ And $\text{Id}_\phi \in Variables$ **then**
21: $NextVar \leftarrow \phi_{inst}$.potentialDependencies[$i$]'s id.
22: $\phi_{iS} \leftarrow \phi_{inst}$.potentialDependencies[$i$].
23: $\phi_{iS} \leftarrow \phi_{iS} \cdot ReadyPotentials[\text{Id}_\phi]$.
24: Marginalize out Id_ϕ from ϕ_{iS}.
25: **else if** $|Variables| = 1$ **then**
26: $\phi_{iS} \leftarrow \phi_{inst}$.potentialDependencies[$i$].
27: **end if**
28: **end if**
29: $\phi_i \leftarrow \phi_{inst}$.potentialDependencies[$i$].
30: **if** $\phi_{iS} = \emptyset$ **then**

$\phi_{iS} \leftarrow \phi_i$.
end if
Let $F \leftarrow$ False.
for j from 0 to i **do**
$Variables \leftarrow \phi_{inst}$.potentialDependencies[$j$]'s variables.
for $V \in Variables$ **do**
Let $Id_{\phi_i} \leftarrow \phi_i$'s id.
if $F =$ False And $V = VarToKeep$ And $Id_{\phi_i} = VarToKeep$ **then**
$F =$ True.
Break.
else if $V = Id_{\phi_i}$ And $F =$ False **then**
ϕ_{inst}.potentialDependencies[j] $\leftarrow$ ϕ_{inst}.potentialDependencies[j] ϕ_{inst}.potentialDependencies[j] $\cdot$ ϕ_i.
Marginalize out Id_{ϕ_i} from ϕ_{inst}.potentialDependencies[j].
$F \leftarrow$ True.
else if $V = Id_{\phi_i}$ And $F =$ True **then**
ϕ_{inst}.potentialDependencies[j] $\leftarrow$ ϕ_{inst}.potentialDependencies[j] $\cdot$ ϕ_{iS}.
Marginalize out Id_{ϕ_i} from ϕ_{inst}.potentialDependencies[j].
end if
end for
end for
$VarToKeep \leftarrow NextVar$.
end for
$Variables \leftarrow \phi_{inst}$.potential's variables.
Let $j \leftarrow 0$.
for i from 1 to $|Variables|$ **do**
if $Variables[i] \neq \mathrm{Id}_\phi$ **then**
while $j < |\phi_{inst}$.potentialDependencies$|$ And ϕ_{inst}.potentialDependencies[j]'s id $\neq Variables[i]$ **do**
$j \leftarrow j + 1$.
end while
if $j < |\phi_{inst}$.potentialDependencies$|$ **then**
ϕ_{inst}.potential $\leftarrow \phi_{inst}$.potential $\cdot$ ϕ_{inst}.potentialDependencies[j].
Marginalize out ϕ_{inst}.potentialDependencies[j]'s id from ϕ_{inst}.potential.
else
ϕ_{inst}.potential $\leftarrow \phi_{inst}$.potential $\cdot$ $ReadyPotentials[Variables[i]]$.

65: Marginalize out $ReadyPotentials[Variables[i]]$'s id from ϕ_{inst}.potential.
66: $j \leftarrow 0$.
67: **end if**
68: **end if**
69: **end for**
70: **end procedure**

Time: $\mathcal{O}(p\_a^2 \cdot p\_v)$, Space: $\mathcal{O}(1)$

As anticipated, the solving phase of the instantiated potential is a critical operation in terms of resources needed to complete it. There are two ways for performing it: marginalizing incomplete potentials starting by the parents of the instantiated potential or marginalizing the potentials starting by the ancestors of the instantiated potential. The first method can easily lead to probability distribution big enough that do not fit in RAM anymore or even they cannot be handled by a 64bit index causing overflows.
The most efficient way is to keep the variable to update in only one of the potentials in the list of dependencies and marginalize it in every other potential. Then using the potential where the parent is still present, marginalize only one of its child potential. In all other children use the completed potential and continue until all the potentials needed by the instantiated potential are solved except one that is still dependent on the variable to update. This approach greatly reduces memory and CPU utilization resulting in much smaller probability distributions.
We adopted the former approach and implemented it in SolveInstantiatedPotential. Starting from the last potential, we identify one potential, among those in ϕ_{inst}.potentialDependencies, that has $VarToKeep$ in its list of parents and marginalize every other variable (in the first iteration the variable to not marginalize is the variable we want to update). Then create a copy of that potential where also $VarToKeep$ is marginalized. Then we use both potentials to marginalize all other potentials in ϕ_{inst}.potentialDependencies. The potential where $VarToKeep$ is still present is used only once, in order to introduce the dependency on that variable on only one other potential. In every other potential, the solved potential is used during the marginalization process. At the end of each iteration we update $VarToKeep$ with the variable contained in $NextVar$. $NextVar$ is the id of the current potential being used in the marginalization process of every other potential. These steps are repeated at each iteration until all the parents of ϕ_{inst} are solved except one that is dependent on ϕ. At this point the parents in ϕ_{inst}.potentialDependencies are marginalized from ϕ_{inst}.potential and the instantiated potential is ready to be used for updating ϕ.

Algorithm 30 UpdateParentPotentialWithEvidence. Given a potential ϕ, a list of instantiated potentials *SolvedInstantiatedPotentials* and a boolean S indicating if the resulting potential is to be stored or not, ϕ is updated with the evidences introduced by performing the following operations:

```
1: procedure UpdateParentPotentialWithEvidence(φ, SolvedInstantiat-
   edPotentials, S)
2:     Let Id_φ ← φ's id.
3:     Let φ_U ← φ.
4:     for φ_I ∈ SolvedInstantiatedPotentials do
5:         φ_U ← φ_U · φ_I.
6:     end for
7:     Let φ_UM ← φ_U with Id_φ marginalized out.
8:     φ_U ← φ_U / φ_UM.
9:     for V ∈ φ_U's variables do
10:        if V ≠ Id_φ then
11:            φ_U ← φ_U with V marginalized out.
12:        end if
13:    end for
14:    if S then
15:        Insert φ_U into ReadyPotential_Evidence.
16:    end if
17:    Return φ_U.
18: end procedure
```

Time: $\mathcal{O}(p\_i)$, Space: $\mathcal{O}(1)$

Once all the instantiated potentials are ready for updating ϕ, we apply the formulas described in 2.4.1 to update ϕ. The resulting potential is stored in *ReadyPotential_Evidence* only when the potential is updated with all its evidences from below and not only a subset, as needed in case of calculating the posterior of a potential with mixed evidences.
After each potential received through messages has been updated with the evidences introduced, we calculate the potentials to be put inside the message.

Algorithm 31 SolvePotentials. Given a clique C, a list of MessagePotentialDependencies RsPrime, a list of MessagePotentialDependencies MessagePotentialsDependencies, a list of MessagePotentialDependencies MessagePotentialsDependencies_S, a list of MessagePotentialDependencies InstantiatedPotentials, a list of MessagePotentialDependencies InstantiatedPotentials_S and a separator S, the dependencies of the potentials to be sent are solved by performing the following operations:

1: **procedure** SolvePotentials(C, RsPrime, MessagePotentialsDependencies, MessagePotentialsDependencies_S, InstantiatedPotentials, InstantiatedPotentials_S, S)
2: Be $RsPotential$ an empty list of MessagePotentialDependencies.
3: Be ϕ_M an empty instance of MessagePotentialDependencies.
4: **for** $\phi \in$ RsPrime **do**
5: Let $Id_\phi \leftarrow \phi$'s id.
6: **if** ϕ.pendingDependencies $\neq \emptyset$ And ϕ.potential is instantiated **then**
7: **if** |MessagePotentialsDependencies_S| > 1 **then**
8: $\phi_M \leftarrow$ SolvePotentialDependenciesForMessage(C, ϕ, MessagePotentialsDependencies_S, S).
9: Order ϕ_M.potentialDependencies in descending order.
10: Insert ϕ_M into $RsPotential$
11: **else**
12: Insert ϕ into $RsPotential$.
13: **end if**
14: **if** |MessagePotentialsDependencies| > 1 **then**
15: SolvePotential(C, MessagePotentialsDependencies, ϕ_M).
16: **end if**
17: **else**
18: Insert ϕ into $RsPotential$.
19: **end if**
20: **if** InstantiatedPotentials $\neq \emptyset$ **then**
21: UpdateAndStoreParentPotential(C, Id_ϕ, InstantiatedPotentials, True).
22: **end if**
23: **end for**
24: **for** $\phi_{inst} \in$ InstantiatedPotentials_S **do**
25: **if** |RsPrime| > 0 **then**
26: Let $\phi_{inst} \leftarrow$ SolveInstantiatedPotentialDependencies(ϕ_{inst}, RsPrime, S).
27: Order ϕ_{inst}.potentialDependencies in descending order.
28: **end if**
29: **end for**

30: RsPrime ← RsPotential.
31: **end procedure**

Time: $\mathcal{O}(p\_a^5 \cdot p\_i \cdot p\_v \cdot mc\_p)$, Space: $\mathcal{O}(p\_a \cdot mc\_p)$

SolvePotentials identifies which potentials are needed by each potential in RsPrime and InstantiatedPotentials_S in order to be sent, i.e. each potential missing in the separator is added to the potentials that need it to solve its posterior distribution. The algorithm is divided into two steps: first the non instantiated potentials are solved for the message, and if possible are solved completely and stored into *ReadyPotentials*. The solving step is executed only if there is more than one potential in MessagePotentialsDependencies_S, i.e. there is at least another potential apart from the one we are solving that could be added to its list of dependencies. Once the potential has been solved for the message, the algorithm tries to solve it completely if there is at least another potential in MessagePotentialsDependencies.
The second step is dedicated to solve the instantiated potentials to send. InstantiatedPotentials_S contains the potentials that will be sent in order to have each instantiated potential only once in each clique. They are solved only if there is at least one potential in RsPrime, meaning that there is at least one potential that could be added to its list of dependencies.

Algorithm 32 SolvePotentialDependenciesForMessage. Given a clique C, an instance of MessagePotentialDependencies ϕ, a list of MessagePotentialDependencies MessagePotentialsDependencies_S and a separator S, the potentials for solving the dependencies of ϕ are selected by performing the following operations:

1: **procedure** SolvePotentialDependenciesForMessage(C, ϕ, MessagePotentialsDependencies_S, S)
2: Let $\phi_S \leftarrow \phi$.
3: Let $Id_\phi \leftarrow \phi$'s id.
4: Let $Variables \leftarrow \emptyset$.
5: **for** $V \in \phi$.pendingDependencies **do**
6: Insert V into $Variables$.
7: **end for**
8: Order $Variables$ in descending order.
9: Let $PotentialIndex \leftarrow 0$.
10: Let $VarIndex \leftarrow 0$.
11: **repeat**
12: **if** $Variables[VarIndex] \notin$ S **then**
13: Let $Ancestors \leftarrow Variables[VarIndex]$'s ancestors.

```
14:             Let UseSolvedPotential ← True.
15:             for Anc ∈ Ancestors do
16:                 if Anc ∈ S then
17:                     Let Found ← False.
18:                     for V' ∈ Variables do
19:                         if V' = Anc then
20:                             Found ← True.
21:                             Break.
22:                         end if
23:                     end for
24:                     if Found = False then
25:                         UseSolvedPotential ← False.
26:                         Break.
27:                     end if
28:                 end if
29:             end for
30:             AddNextPotentialToSolveParent(C, φ_S, VarIndex,
                PotentialIndex, Variables, MessagePotentialsDependencies_S,
                UseSolvedPotential).
31:             if PotentialIndex = |MessagePotentialsDependencies_S| then
32:                 PotentialIndex ← 0.
33:                 VarIndex ← VarIndex + 1.
34:             end if
35:         else
36:             VarIndex ← VarIndex + 1.
37:         end if
38:     until VarIndex < |Variables|
39:     Return φ_S.
40: end procedure
```

Time: $\mathcal{O}(p\_a^4 \cdot p\_i)$, Space: $\mathcal{O}(p\_a)$

Firstly, in order to find the potentials that satisfy the dependencies of ϕ, the set of pending potentials is converted into a vector ordered in descending order. Then each variable is checked whether it is present in the separator or not. If it is not present, then the ancestors of the node are checked whether they are present in the separator and are already present in ϕ_S.pendingDependencies. If all ancestors which are present in the separator are also present in ϕ_S.pendingDependencies it means that ϕ_S can already be used for propagating evidences to all those ancestors, if needed, and the next dependency to be chosen will be selected among

the already solved potentials reducing execution time during the solving phase. Then AddNextPotentialToSolveParent selects the correct potential to be added to ϕ_S.potentialDependencies, between the potentials in the messages, those already solved (with evidence from above or not) and stored in *ReadyPotentials* and those already solved with evidences from below and stored in *ReadyPotential_Evidence*.

Algorithm 33 AddNextPotentialToSolveParent. Given a clique C, an instance of MessagePotentialDependencies ϕ, an index VarIndex, an index PotentialIndex, a list of variables Variables, a list of MessagePotentialDependencies MessagePotentialsDependencies_S and a boolean UseSolvedPotential, the next potential to solve the dependencies of ϕ is selected by performing the following operations:

1: **procedure** AddNextPotentialToSolveParent(C, ϕ, VarIndex, PotentialIndex, Variables, MessagePotentialsDependencies_S, UseSolvedPotential)
2: Let $Id_\phi \leftarrow \phi$'s id.
3: Let $Id_{\phi_{PI}} \leftarrow$ MessagePotentialsDependencies_S[PotentialIndex].potential's id.
4: **if** $Id_{\phi_{PI}} \neq Id_\phi$ And $Id_{\phi_{PI}} \in \phi$.pendingDependencies **then**
5: **if** $EvidencesFromBelow[Id_\phi] \neq \emptyset$ And IsEvidenceFromBelowSubsetOf($EvidencesFromBelow[Id_\phi]$, $EvidencesFromBelow$[Variables[VarIndex]]) **then**
6: Let $EvidencesSubset \leftarrow$ GetEvidenceFromBelowSubsetOf($EvidencesFromBelow[Id_\phi]$, $EvidencesFromBelow$[Variables[VarIndex]]).
7: Let $\phi_{PIS} \leftarrow$ UpdateParentWithSubsetEvidences(C, Variables[VarIndex], $EvidencesSubset$).
8: **if** $\phi_{PIS} \neq \emptyset$ **then**
9: Add ϕ_{PIS} to ϕ.potentialDependencies.
10: **end if**
11: VarIndex $\leftarrow$ VarIndex + 1.
12: **else if** UsePotentialWithOrWithoutEvidencesFromAbove(Id_ϕ, VarIndex, Variables) **then**
13: **if** UseSolvedPotential = True And $ReadyPotentials$[Variables[VarIndex]] $\neq \emptyset$ **then**
14: Add $ReadyPotentials$[Variables[VarIndex]] to ϕ.potentialDependencies.
15: **else**
16: **while** PotentialIndex < |MessagePotentialsDependencies_S| And $Id_{\phi_{PI}} \notin \phi$.pendingDependencies **do**
17: PotentialIndex $\leftarrow$ PotentialIndex + 1.

```
18:                 end while
19:                 if PotentialIndex < |MessagePotentialsDependencies_S| then
20:                     Add MessagePotentialsDependencies_S[PotentialIndex] to
                        φ.potentialDependencies.
21:                     Let Variables′ ← ∅
22:                     for V ∈ φ.pendingDependencies do
23:                         Insert V into Variables′.
24:                     end for
25:                     Order Variables′ in descending order.
26:                     Variables ← Variables′.
27:                 end if
28:             end if
29:         else if UsePotentialWithEvidencesFromBelow(Id_φ, VarIndex, Variables)
    then
30:             Add ReadyPotential_Evidence[Variables[VarIndex]] to
                φ.potentialDependencies.
31:             VarIndex ← VarIndex + 1.
32:         else
33:             VarIndex ← VarIndex + 1.
34:         end if
35:     else
36:         PotentialIndex ← PotentialIndex + 1.
37:     end if
38: end procedure
```

Time: $\mathcal{O}(p\_a^3 \cdot p\_i)$, Space: $\mathcal{O}(p\_a)$

There are three possible outcomes when AddNextPotentialToSolveParent decides which potential is the right one to satisfied the next dependency of ϕ:

- Use a potential that has no evidences or just evidences from above. It can be either already solved or picked from MessagePotentialsDependencies_S if not. When a potential from MessagePotentialsDependencies_S is chosen, then the list of dependencies is updated with the newly introduced dependencies since the potential selected could be not solved.
- Use a solved potential that has evidences from below. The potential is added only when it has been already calculated in order to not repeat the same calculations, saving time.
- In case of mixed evidences, the parent has to be updated with only a subset of the evidences from below which do not include the evidence from below

of ϕ. UpdateParentWithSubsetEvidences calculates, if possible, the needed parent potential.

Algorithm 34 UpdateParentWithSubsetEvidences. Given a clique C, a potential's id Id_ϕ and a set of evidences E_S, ϕ is updated with the evidences in E_S by performing the following operations:

```
1: procedure UpdateParentWithSubsetEvidences(C, Id_φ, E_S)
2:     Be InstantiatedPotentials a list of MessagePotentialDependencies.
3:     for M ∈ C's messages list do
4:         for φ_inst ∈ M.instantiatedPotentials do
5:             Let Id_φinst ← φ_inst's id.
6:             if Id_φinst ∈ E_S then
7:                 Insert φ_inst into InstantiatedPotentials.
8:             end if
9:         end for
10:    end for
11:    for φ_C ∈ C's list of potentials do
12:        Let Id_φC ← φ_C's id.
13:        if Id_φC ∈ E_S then
14:            Let φ_inst ← φ_C.
15:            Instantiate φ_inst with evidence.
16:            Insert φ_inst into InstantiatedPotentials.
17:        end if
18:    end for
19:    if InstantiatedPotentials ≠ ∅ then
20:        if Id_φ ∈ ReadyPotentials then
21:            Be InstantiatedPotentialsSolved a list of potentials.
22:            Let φ ← ReadyPotentials[Id_φ].
23:            if |InstantiatedPotentials| = |E_S| then
24:                UpdateAndSolveInstantiatedPotentials(C, Id_φ,
                   InstantiatedPotentials, InstantiatedPotentialsSolved).
25:            end if
26:            if |InstantiatedPotentialsSolved| = |E_S| then
27:                Return UpdateParentPotentialWithEvidence(φ,
                   InstantiatedPotentialsSolved, False).
28:            end if
29:        end if
30:        Return empty potential.
31:    end if
```

```
32:     Return empty potential.
33: end procedure
```

Time: $\mathcal{O}(p\_a^3 \cdot p\_i)$, Space: $\mathcal{O}(p\_a)$

UpdateParentWithSubsetEvidences creates a version of the parent potential ϕ which is updated with only a subset of all its evidences from below. It starts by creating the list of instantiated potentials needed selecting them from the instantiated potentials received through messages and the clique potentials. If all the potentials required are available, the parent is updated by performing the same operations described before.
Once the non-instantiated potential is ready to be sent, SolvePotential in Algorithm 31 tries to solve it and store the resulting posterior in *ReadyPotentials*.

Algorithm 35 SolvePotential. Given a clique C, a list of MessagePotentialDependencies MessagePotentialsDependencies and an instance of MessagePotentialDependencies ϕ, ϕ is solved by performing the following operations:

```
1: procedure SolvePotential(C, MessagePotentialsDependencies, φ)
2:     Let Id_φ ← φ's id.
3:     if Id_φ ∉ ReadyPotentials then
4:         Let φ_S ← φ.
5:         if MessagePotentialsDependencies ≠ ∅ then
6:             AddPotentialDependencies(C, MessagePotentialsDependencies, φ_S).
7:         end if
8:         if IsPotentialSolved(φ_S) then
9:             Order φ_S.potentialDependencies in descending order.
10:            SolvePotentialDependencies(C, φ_S).
11:            Insert φ_S into ReadyPotentials.
12:        end if
13:    end if
14: end procedure
```

Time: $\mathcal{O}(p\_a^5 \cdot p\_i \cdot p\_v)$, Space: $\mathcal{O}(p\_a)$

Before the potential can be solved, AddPotentialDependencies completes the list of dependencies of ϕ retrieving the needed potentials from those in MessagePotentialsDependencies, *ReadyPotential* or *ReadyPotential_Evidence*.

Algorithm 36 AddPotentialDependencies. Given a clique C, a list of MessagePotentialDependencies MessagePotentialsDependencies and an instance of MessagePotentialDependencies ϕ, the potentials for solving the dependencies of ϕ are selected by performing the following operations:

```
1: procedure AddPotentialDependencies(C, MessagePotentialsDependencies, φ)
2:     Let Variables ← ∅.
3:     for V ∈ φ.pendingDependencies do
4:         Insert V into Variables.
5:     end for
6:     Order Variables in descending order.
7:     Let PotentialIndex ← 0.
8:     Let VarIndex ← 0.
9:     repeat
10:        AddNextPotentialToSolveParent(C, φ, VarIndex, PotentialIndex,
           Variables, MessagePotentialsDependencies, True).
11:        if PotentialIndex = |MessagePotentialsDependencies| then
12:            PotentialIndex ← 0.
13:            VarIndex ← VarIndex + 1.
14:        end if
15:    until VarIndex < |Variables|
16: end procedure
```

Time: $\mathcal{O}(p\_a^4 \cdot p\_i)$, Space: $\mathcal{O}(p\_a)$

After all the needed potentials, from MessagePotentialsDependencies, have been selected to solve ϕ, IsPotentialSolved checks whether the potential can be solved or not by checking whether the list of pending dependencies is empty or not. If it is not empty it means that some of the potentials needed are potentials updated with evidences from below. If those potentials are all available then ϕ can be solved.

Algorithm 37 IsPotentialSolved. Given an instance of MessagePotentialDependencies ϕ, the potential is deemed solved by performing the following operations:

```
1: procedure IsPotentialSolved(φ)
2:     if φ.pendingDependencies = ∅ then
3:         Return True.
4:     end if
```

```
    for Id_φD ∈ φ.pendingDependencies do
        if Id_φD ∉ ReadyPotential_Evidence then
            Return False.
        end if
    end for
    Return True.
end procedure
```

The steps performed to solve a non-instantiated potential ϕ are similar to those described in Algorithm 29.

Algorithm 38 SolvePotentialDependencies. Given a clique C and an instance of MessagePotentialDependencies ϕ, the potential is solved by performing the following operations:

```
procedure SolvePotentialDependencies(C, φ)
    Let Id_φ ← φ's id.
    Be Variables an empty list of potentials id.
    for i from |φ.potentialDependencies| - 1 to 0 do
        Variables ← φ.potentialDependencies[i]'s list of variables.
        for j from 1 to |Variables| do
            if EvidencesFromBelow[Id_φ] ≠ ∅ And
IsEvidenceFromBelowSubsetOf(EvidencesFromBelow[Id_φ],
EvidencesFromBelow[Variables[j]]) then
                Let EvidencesSubset ← GetEvidenceFromBelowSubsetOf(Evi-
                dencesFromBelow[Id_φ], EvidencesFromBelow[Variables[j]]).
                Let φ_S ← UpdateParentWithSubsetEvidences(C, Variables[j],
                EvidencesSubset).
                φ.potentialDependencies[i] ← φ.potentialDependencies[i] · φ_S.
            else
                φ.potentialDependencies[i] ← φ.potentialDependencies[i] ·
                ReadyPotential_Evidence[Variables[j]].
            end if
            Marginalize out Variables[j] from φ.potentialDependencies[i].
        end for
        for j from 0 to i - 1 do
            Variables ← φ.potentialDependencies[j]'s list of variables.
            for V ∈ Variables do
```

```
19:                 if V = φ.potentialDependencies[i]'s id. then
20:                     φ.potentialDependencies[j] ← φ.potentialDependencies[j] ·
                        φ.potentialDependencies[i].
21:                     Marginalize out V from φ.potentialDependencies[j].
22:                 end if
23:             end for
24:         end for
25:     end for
26:     Variables ← φ.potential's list of variables.
27:     Let j ← 0.
28:     for i from 1 to |Variables| do
29:         while j < |φ.potentialDependencies| And φ.potentialDependencies[j]'s
    id ≠ Variables[i] do
30:             j ← j + 1.
31:         end while
32:         if j < |φ.potentialDependencies| then
33:             φ.potential ← φ.potential · φ.potentialDependencies[j].
34:             Marginalize out φ.potentialDependencies[j]'s id from φ.potential.
35:         else
36:             j ← 0.
37:         end if
38:     end for
39: end procedure
```

Time: $\mathcal{O}(p\_a^5 \cdot p\_i \cdot p\_v)$, Space: $\mathcal{O}(p\_a)$

As in Algorithm 29, SolvePotentialDependencies starts solving the last potential in potentialDependencies and it is marginalized out from every other potential in the list and continues backwards from the last potential. The potentials are solved by choosing the right potential from those available in *ReadyPotential* or *ReadyPotential_Evidence* or the parent is updated with only a subset of evidences. The next step is to use the solved ϕ.potential's parents from potentialDependencies to solve ϕ.potential.
During Algorithm 31, each potential in RsPrime is then updated by Algorithm 24 with evidences from below, if any is present, and the updated potential is stored in *ReadyPotential_Evidence*.
The last step needed to create the message is to add the dependencies to the instantiated potentials that will be sent within the message.

Algorithm 39 SolveInstantiatedPotentialDependencies. Given an instance of MessagePotentialDependencies ϕ_{inst}, a list of MessagePotentialDependencies RsPrime and a separator S, the dependencies needed by ϕ_{phi} are selected by performing the following operations:

1: **procedure** SolveInstantiatedPotentialDependencies(ϕ_{inst}, RsPrime, S)
2: Let $Id_{\phi_{inst}} \leftarrow \phi_{inst}$.potential's id.
3: **for** i from |RsPrime| - 1 to 0 **do**
4: Let $Id_{\phi_i} \leftarrow$ RsPrime[i].potential's id.
5: **if** (S $\neq \emptyset$ And $Id_{\phi_i} \notin$ S) Or S $= \emptyset$ **then**
6: **if** $Id_{\phi_i} \notin \phi_{inst}$.solvedDependencies And $Id_{\phi_i} \in \phi_{inst}$.pendingDependencies **then**
7: Add RsPrime[i] to ϕ_{inst}.potentialDependencies.
8: **end if**
9: **end if**
10: **end for**
11: **end procedure**

Time complexity: $\mathcal{O}(p\_a)$, Space complexity: $\mathcal{O}(p\_a)$

When both list of potentials, the instantiated and non-instantiated potentials, are ready, they are put inside the message and sent to the parent/child clique.

Posterior Marginal Retrieval

After each update of the network performed by the LazyPropagation, following a change of the observations introduced in the network or the target variable is changed resulting in a change of the barren variables, the main task performed by the user is to visualize the posterior marginal of any variable he needs.

Algorithm 40 PosteriorMarginal. Given the id of a potential Id_ϕ and the list of evidences L, the posterior marginal of Id_ϕ is obtained by performing the following operations:

1: **procedure** PosteriorMarginal(Id_ϕ)
2: UpdateNetworkIfNeeded(Id_ϕ).
3: Let $C \leftarrow$ the clique's id containing ϕ.
4: Be *Potentials* an empty list of MessagePotentialDependencies.
5: Be $Potentials_{inst}$ an empty list of MessagePotentialDependencies.
6: Be ϕ an empty instance of MessagePotentialDependencies.

```
7:  if Id_φ ∈ ReadyPotential_Evidence then
8:      Return ReadyPotential_Evidence[Id_φ].
9:  end if
10: if Id_φ ∈ ReadyPotential then
11:     Return ReadyPotential[Id_φ].
12: end if
13: for φ_C ∈ C's potentials list do
14:     Let φ_inst ← φ_C.
15:     for E ∈ L do
16:         φ_inst ← φ_inst instantiated to E.
17:     end for
18:     if φ_inst is not an evidence then
19:         if φ_inst's id = Id_φ then
20:             φ ← φ_inst.
21:         else
22:             Insert φ_inst into Potentials.
23:         end if
24:     else
25:         Insert φ_inst into Potentials_inst.
26:     end if
27: end for
28: for M ∈ C's messages list do
29:     for φ_M ∈ M.potentialDependencies do
30:         Insert φ_M into Potentials.
31:     end for
32:     for φ_M ∈ M.instantiatedPotentialDependencies do
33:         Insert φ_M into Potentials_inst.
34:     end for
35: end for
36: Be SeparatedNodes an empty set of NodeSeparated.
37: for φ_P ∈ Potentials do
38:     Insert φ_P.potential's variables into SeparatedNodes.
39: end for
40: IsDSeparated(SeparatedNodes, C, L).
41: FindRelevanPotentials(Potentials, SeparatedNodes).
42: Order Potentials in descending order.
43: SolvePotential(C, Potentials, φ).
```

```
44:     if EvidencesFromBelow[Id_φ] ≠ ∅ then
45:         Order Potentials in ascending order.
46:         for φ_inst ∈ Potentials_inst do
47:             if Potentials ≠ ∅ then
48:                 SolveInstantiatedPotentialDependencies(φ_inst, Potentials).
49:                 Sort φ_inst.potentialDependencies in descending order.
50:             end if
51:         end for
52:         if Potentials_inst ≠ ∅ then
53:             UpdateAndStoreParentPotential(C, Id_φ, Potentials_inst, True).
54:         end if
55:         Return ReadyPotential_Evidence[Id_φ].
56:     else
57:         Return ReadyPotential[Id_φ].
58:     end if
59: end procedure
```

In order to be able to retrieve the requested posterior marginal, PosteriorMarginal starts the execution of the Lazy Propagation algorithm if the evidences introduced changed or if barren variables are used and need to be changed. Once the network has been updated, it is first checked whether the requested posterior marginal is already available or not. If not it is calculated by performing the same operations done during the Lazy Propagation algorithm and then returned.

4.6 Network Learning Input Dataset

4.6.1 CSVDataStructure

In our library, the input dataset format supported for learning networks is the CSV. It is one of the most widespread and supported format for datasets. Since datasets can have many records (thousands if not hundreds of thousands) it is crucial to store them efficiently in memory once read. We adopted a run length encoding in our library to reduce the memory used by the dataset when a variable takes the same value across multiple consecutive records; this has the effect of speeding up the read of the records during the learning phase.

```
1 struct {
2     int value;
3     int count;
4 } RunLengthItem;
```

```
struct {
    String[] variableNames;
    Map<String, int>[] mappedValues;
    RunLenghtItem[][] CSVData;
    int sampleSize;
} CSVDataStructure;
```

RunLengthItem contains for each value of the column in the CSV file the number of consecutive times the same value appeared. CSVDataStructure contains the list of columns inside the CSV file in the list variableNames. The mapping between the state of a variable and its int value is stored in mappedValues. This association is stored for each variable. CSVData data contains for each variable the list of RunLenghtItem. sampleSize represents the number of total records contained in the dataset.

4.7 PCHC Class

4.7.1 PCHC Data Structures

NodeAssociation

NodeAssociation is a data structure used to keep track of the associations between a node and every other node of the network. It is composed of an Id, to identify the node, and a floating point variable to store the value of association. The associations will be measured by a pValue.

```
struct {
    int id;
    double pValue;
} NodeAssociation;
```

4.7.2 PCHC Class Functions

PCHC Algorithm

As anticipated, the PCHC algorithm is a combination of the PC algorithm and an Hill-climbing scoring phase. Throughout the discussion of the algorithm's implementation, the dataset contained in CSVDataStructure will be referenced as D.

Algorithm 41 PCHCLearning. Given a list of ArcConstraint C, the network's structure is identified by performing the following operations:

```
1: procedure PCHCLearning(C)
2:     Be NodesAssociations an empty matrix of NodeAssociation
3:     InitializeAssociationsMatrix(NodesAssociations).
4:     PCPhase(NodesAssociations).
5:     CheckNeighborhoodConsistency(NodesAssociations).
6:     Be UndGraph and empty instance of UndirectedGraph Class.
7:     for i from 0 to |NodesAssociations| do
8:         for j from 0 to |NodesAssociations[i]| do
9:             Add edge (i, NodesAssociations[i][j].id) to UndGraph.
10:        end for
11:    end for
12:    Let DAG ← HillClimbingScoringPhase(UndGraph, C).
13:    Let NewNodesOrder ← DAG.ReorderNodes(). ▷
       After the Hill-climbing scoring phase, it is possible to obtain a DAG where
       the node parent has an id higher than the child's id. This breaks the DAG's
       consistency and the nodes' ids have to be reordered to avoid these situations.
       The new nodes' ids order is used to reorder the nodes in D since the dataset
       is needed during the parameter estimation phase.
14:    D.ReorderNodes(NewNodesOrder).
15:    Return DAG.
16: end procedure
```

The version of the PC algorithm we implemented is the stable-PC as introduced in [7]. Compared to the original PC, the output of stable-PC does not depend on the order in which the variables appears in the dataset, i.e. it is a deterministic algorithm.
InitializeAssociationsMatrix initializes the matrix of nodes associations, associating each node with every other node with a pValue equal to zero. Once the matrix is ready, the PC algorithm is executed. PCPhase is a wrapper function that decides whether the PC algorithm is executed in a serial or parallel fashion; the matrix *NodesAssociations* is used not only as a support for the calculations but also to return the skeleton of the undirected graph identified. It is possible that during the PC algorithm some associations are not removed from both nodes, for example node B is removed from the list of associations of A but not vice versa. CheckNeighborhoodConsistency makes the matrix consistent by adding the missing associations that are present in reverse form (considering the example above, B is added to the associations of A since A is present in the list of associations of B). Once the undirected graph corresponding the matrix is created, the Hill-climbing

scoring phase will determine the orientation of each arc of the network and whether the presence or absence of an arc constitutes a better network while respecting the constraints introduced by C. At the end, the nodes' ids are reordered to keep the network consistent.

Algorithm 42 PCSerial. Given a matrix of NodeAssociation NodesAssociations, the network's skeleton is identified by performing the following operations:

```
1: procedure PCSerial(NodesAssociations)
2:     Let Depth ← 0.
3:     repeat
4:         for i from 0 to |NodesAssociations| do
5:             FindEdgesToRemove(i, NodesAssociations[i], Depth).
6:         end for
7:         for i from 0 to |NodesAssociations| do
8:             Order NodesAssociations[i] in ascending order.
9:         end for
10:        RemoveNodesAssociations(NodesAssociations).
11:        Depth ← Depth + 1.
12:    until PCStoppingCondition(NodesAssociations, Depth) = True
13: end procedure
```

FindEdgesToRemove calculates for each nodes associated to node i the pValue, given a certain depth. With each depth increment, increases also the size of the conditioning set of variables that will be tested in FindEdgesToRemove. The associations of each node are then ordered in ascending order of pValue so that at the next iteration the variables with a stronger association are used as conditioning variables before those with a weaker association. Then RemoveNodesAssociations removes from each node's associations list the nodes that are deemed not associated. PCStoppingCondition checks whether the stopping condition of the PC algorithm has been reached, i.e. the number of nodes associated with each node minus 1 is less than the depth reached by the algorithm. If yes the algorithm is interrupted.

Algorithm 43 PCParallel. Given a matrix of NodeAssociation NodesAssociations, the network's skeleton is identified by performing the following operations:

```
1: procedure PCParallel(NodesAssociations)
2:     Let Depth ← 0.
3:     repeat
```

```
4:        for i from 0 to |NodesAssociations| do
5:            Let Job ← FindEdgesToRemove(i, NodesAssociations[i], Depth).
6:            Send Job to TP.
7:        end for
8:        for i from 0 to |NodesAssociations| do
9:            Order NodesAssociations[i] in ascending order.
10:       end for
11:       RemoveNodesAssociations(NodesAssociations).
12:       Depth ← Depth + 1.
13:   until PCStoppingCondition(NodesAssociations, Depth) = True
14: end procedure
```

As proposed in [7], the results of the independence tests performed at each level of the PC algorithm directly impact the results of the next level. This makes infeasible to parallelize the PC algorithm across multiple levels but the tests performed at each level can be parallelized. Compared to the solution proposed in [7], we implemented a sub-optimal solution where we parallelized the tests at nodes level: for each node N we send a FindEdgesToRemove task to the thread pool where the pValues of every nodes still associated to N are calculated. This solution has the disadvantage of not uniformly spreading the independence tests across the threads, i.e. a thread could have thousands of tests to perform while all other threads have only hundreds of tests to perform.
The main thread waits for the completion of all tasks in order to proceed with the removal of associations that are not valid anymore.

Algorithm 44 FindEdgesToRemove. Given a node N, a vector of NodeAssociation NodesAssociations and the depth D, the associations between N and every node in NodesAssociations are updated by performing the following operations:

```
1: procedure FindEdgesToRemove(N, NodesAssociations, D)
2:     Be Nodes a vector of size 2.
3:     Nodes[0] ← N.
4:     if D > 0 And |NodesAssociations| >= D + 1 then
5:         Be Zx a vector of size D.
6:         Increment Nodes' size to 2 + D.
7:         for i from 0 to D do
8:             Zx[i] ← i.
9:         end for
10:        Be NodesToTest a vector of size |NodesAssociations| - D.
11:        repeat
12:            if CheckIfAllNodesAreValid(NodesAssociations, Zx) then
```

```
13:                 for k from 0 to |Zx| do
14:                     Nodes[k + 2] ← NodesAssociations[Zx[k]].id.
15:                 end for
16:                 j ← 0.
17:                 k ← 0.
18:                 for i from 0 to |NodesAssociations| do
19:                     if k < |Zx| And NodesAssociations[i].id =
    NodesAssociations[Zx[k]].id then
20:                         k ← k + 1.
21:                     else
22:                         NodesToTest[j] ← i.
23:                         j ← j + 1.
24:                     end if
25:                 end for
26:                 for K ∈ NodesToTest do
27:                     if NodesAssociations[K].pValue <= 0.05 then
28:                         Nodes[1] ← NodesAssociations[K].id.
29:                         NodesAssociations[K].pValue ←
                            IndependenceTest(Nodes).
30:                     end if
31:                 end for
32:             end if
33:         until NextCombination(Zx, NodesAssociations, D) = False
34:     else
35:         for i from 0 to |NodesAssociations| do
36:             Nodes[1] ← NodesAssociations[i].id.
37:             NodesAssociations[i].pValue ← IndependenceTest(Nodes).
38:         end for
39:     end if
40: end procedure
```

When deciding whether the link between two nodes is necessary or not, the independence test has to be performed for each possible permutation of variables in the conditioning set of size D. If for every possible permutation the independence test returns a value less than or equal to 0.05 then the link is kept. Vice versa, for a link to be removed it needs only one test with result higher than 0.05. NextCombination calculates every possible permutation of nodes from NodesAssociations taking D nodes from it. When all possible permutations have been tested the loop ends. CheckIfAllNodesAreValid checks whether all pValues of the nodes selected by Zx are less than or equal to 0.05. If one node has a higher pValue the combination is

not valid and discarded for the test. The following one is selected.

4.8 HillClimbingScoringPhase Class

4.8.1 HillClimbingScoringPhase Data Structures

ArcConstraint

ArcConstraint is used to introduce constraints for the presence or absence of arcs in the resulting DAG. It contains two strings indicating the names of the head and tail nodes of arc and the type of constraint: 0 = the arc must be present, 1 = the arc has not to be present.

```
struct {
    string head;
    string tail;
    int type;
} ArcConstraint;
```

Move

Move contains the information relative to the move identified by the Hill-climbing algorithm. Action represents the action to perform on the arc: 0 = add arc, 1 = remove arc, 2 = reverse arc. Arc is the arc relative to the move. ScoreDiff contains the difference of score between before and after the move is applied on the DAG.

```
struct {
    int action;
    Arc arc;
    double scoreDiff;
} Move;
```

4.8.2 HillClimbingScoringPhase Class Functions

FastCHC Algorithm

The HillClimbingScoringPhase Class, as the name suggests, is dedicated to the search of a DAG representing the network through an hill-climbing heuristic. It is applied on the undirected graph resulting from the PC algorithm. The hill-climbing algorithm starts from an empty DAG and performs one of three possible actions at each iteration: add an arc, if it was present in the undirected graph and does not

introduce a cycle in the DAG, remove an arc from the DAG or reverse an arc in the DAG if it does not introduce a cycle. At each iteration the move that generates the highest positive increase of score is applied to the DAG. If none of the moves generates a positive increase of score then the algorithm is stopped.
The initial score of each iteration will be referenced as IS, the resulting DAG as DAG, the best moves selected at each iteration are stored in the list $Moves$ and the list of forbidden parents for each node in FP.

Algorithm 45 FastCHC. Given an undirected graph UG and a list of ArcConstraint ACs, the DAG of the network is obtained by performing the following operations:

```
1: procedure FastCHC(UG, ACs)
2:     Be ArcsToAdd an empty set of arcs.
3:     Let ArcsNotToAdd an empty set of arcs.
4:     Be DAG and empty DAG.
5:     for AC ∈ ACs do
6:         Convert the names of the nodes in AC into node ids and create an arc
           a(head, tail).
7:         if AC.type = 0 then
8:             Add a to DAG.
9:             Create an edge e(head, tail).
10:            Remove e from UG.
11:            Add a to ArcsToAdd.
12:        else
13:            Add a to ArcsNotToAdd.
14:        end if
15:    end for
16:    Calculate the initial score of DAG through BDs and store it in IS.
17:    if TP is initialized then
18:        FastCHCParallel(UG, ArcsToAdd, ArcsNotToAdd).
19:    else
20:        FastCHCSerial(UG, ArcsToAdd, ArcsNotToAdd).
21:    end if
22:    Return DAG.
23: end procedure
```

The main function of the HillClimbingScoringPhase Class is just a wrapper function that converts the ArcConstraint nodes' names into actual arcs and applies the constraints to the DAG and the starting undirected graph. The arcs that has to be present are removed from the undirected graph because in this way they are not tested for removal or reversal. The function then decides whether the parallel or serial FastCHC is used.

Algorithm 46 FastCHCSerial. Given an undirected graph UG, a list of arcs to not change ANC and a list of arcs to not add ANA, the DAG of the network is obtained by performing the following operations:

```
1: procedure FastCHCSerial(UG, ANC, ANA)
2:     Let N ← the number of nodes in UG.
3:     Let Imp ← True.
4:     repeat
5:         Imp ← False.
6:         for i from 0 to N do
7:             AddArcMove(i, N, UG, ANA).
8:         end for
9:         for i from 0 to N do
10:            RemoveOrReverseArcMove(i, N, UG, ANC, ANA).
11:        end for
12:        Imp ← ApplyBestMove().
13:    until Imp = True
14: end procedure
```

The main slow down of the FastCHC algorithm is the scoring of the DAG during the selection of the best move. This part of the algorithm can take advantage of the parallelization so that multiple scoring of the DAG can be performed simultaneously, speeding up significantly the FastCHC algorithm for large networks. Like in the case of the PC algorithm, we parallelized the FastCHC algorithm at nodes' level, i.e. we submit to the thread pool a job of AddArcMove and RemoveOrReverseArcMove per node at each iteration.

Algorithm 47 FastCHCParallel. Given an undirected graph UG, a list of arcs to not change ANC and a list of arcs to not add ANA, the DAG of the network is obtained by performing the following operations:

```
1: procedure FastCHCParallel(UG, ANC, ANA)
2:     Let N ← the number of nodes in UG.
3:     Let Imp ← True.
4:     repeat
5:         Imp ← False.
6:         for i from 0 to N do
7:             Let Job ← AddArcMove(i, N, UG, ANA).
```

```
8:            Send Job to TP.
9:        end for
10:       for i from 0 to N do
11:           Let Job ← RemoveOrReverseArcMove(i, N, UG, ANC, ANA).
12:           Send Job to TP.
13:       end for
14:       Imp ← ApplyBestMove().
15:   until Imp = True
16: end procedure
```

Although the parallelization introduces a significant speed-up for the algorithm, our solution introduces indeterminism in the output due to operations performed in different order with respect to the serial solution. This is not necessarily a downside since this indeterminism produces different outputs, i.e. DAGs slightly different from each other. The user can then use the DAG with the highest score among those obtained or use some criterion to determine which suits his needs best.

Algorithm 48 AddArcMove. Given a node's id Id, the number of nodes in the graph N, an undirected graph UG and a list of arcs to not add ANA, the best move for adding an arc is obtained by performing the following operations:

```
1: procedure AddArcMove(Id, N, UG, ANA)
2:     for i from 0 to N do
3:         if i ≠ Id then
4:             Be e the edge (i, Id).
5:             if e ∈ E And Id ∉ pa(i) And i ∉ FP[Id] then
6:                 Be a the arc (Id, i).
7:                 if a ∉ ANA And a does not introduce a cycle in DAG then
8:                     Let NewScore ← DSM(DAG, 0, a).
9:                     Store the move in Moves if it improves the score.
10:                    if NewScore - IS < 0 then
11:                        Add i to FP[Id].
12:                        Add Id to FP[i].
13:                    end if
14:                end if
15:            end if
16:        end if
17:    end for
18: end procedure
```

AddArcMove tries to add an arc only if the edge that generates it exists in the

undirected graph coming from the PC algorithm. Then move is stored in $Moves$ only if it improves the score of the DAG and is the best move found so far for adding an arc. The forbidden parents sets are updated accordingly to the rules defined in 2.7.1. DSM (DagScoringMetric) scores the dag with the BDs metric. A value is assigned to each operation to score (addition = 0, removal = 1, reversal = 2). In this way we avoid applying the actual operation to the DAG itself during the scoring.

Algorithm 49 RemoveOrReverseArcMove. Given a node's id Id, the number of nodes in the graph N, an undirected graph UG, a list of arcs to not change ANC and a list of arcs to not add ANA, the best move for removing or reversing an arc is obtained by performing the following operations:

```
1: procedure RemoveOrReverseArcMove(Id, N, UG, ANC, ANA)
2:     Let Parents ← pa(Id).
3:     for P ∈ Parents do
4:         Be a the arc (P, Id).
5:         if a ∉ ANC then
6:             Let NewScore ← DSM(DAG, 1, a).
7:             Store the move in Moves if it improves the score.
8:             if NewScore - IS > 0 then
9:                 Add P to FP[Id].
10:                Add Id to FP[P].
11:            end if
12:            Be a_R the reverse of a.
13:            if a_R ∉ ANA And does not introduce a cycle in DAG then
14:                Let NewScore2 ← DSM(DAG, 2, a).
15:                Let Diff ← NewScore + NewScore2 - 2 · IS.
16:                Store the move in Moves if it improves the score.
17:                if NewScore - IS > 0 Or NewScore2 - IS < 0 then
18:                    Add P to FP[Id].
19:                    Add Id to FP[P].
20:                end if
21:            end if
22:        end if
23:    end for
24: end procedure
```

RemoveOrReverseArcMove identifies the best move for both removal and reversal of already present arcs. The best moves that generates an increase of score are saved in $Moves$.

ApplyBestMove then picks the move with the highest score difference between before and after the move is applied. If an adding move is applied, the forbidden parents set are relaxed increasing the number of alternative moves available at the next iteration. When no move is applied, the algorithm is stopped by ApplyBestMove returning false.

Algorithm 50 ApplyBestMove. The best move is applied by performing the following operations:

```
 1: procedure ApplyBestMove
 2:     Be BM an empty instance of Move.
 3:     Let Imp ← False.
 4:     for M ∈ Moves do
 5:         if BM.scoreDiff < M.scoreDiff then
 6:             BM ← M.
 7:         end if
 8:     end for
 9:     if BM.scoreDiff > 0 then
10:         Imp ← True.
11:         if BM.action = 0 then
12:             Add BM.arc to DAG.
13:             Let Neighborhood ← nb(BM.arc's head).
14:             for N ∈ Neighborhood do
15:                 Remove BM.arc's tail from FP[N].
16:                 Remove N from FP[BM.arc's tail].
17:             end for
18:             Let Neighborhood ← nb(BM.arc's tail).
19:             for N ∈ Neighborhood do
20:                 Remove BM.arc's head from FP[N].
21:                 Remove N from FP[BM.arc's head].
22:             end for
23:         else if BM.action = 1 then
24:             Remove BM.arc from DAG.
25:         else
26:             Reverse BM.arc in DAG.
27:         end if
28:         IS ← IS + BM.scoreDiff.
29:     end if
30:     Return Imp.
31: end procedure
```

4.9 StrengthOfInfluence Class

In our library we implemented functionalities to measure the strength of influence that each parent has on a predetermined variable, following the operations described in [14].

4.9.1 StrengthOfInfluence Class Functions

Parent Strength Of Influence

In order to measure the influence that each parent has on a variable V, we need to measure the "distance" between the posterior marginal of V and the posterior marginal of V where a parent is instantiated to each possible state of its n states, for example given a distance measure D, we want to measure the average of the distances: Avg(D(P(V), P(V|A=a_1)), D(P(V), P(V|A=a_2), ..., D(P(V), P(V|A=a_n))).

Algorithm 51 CalculateParentInfluence. Given a potential ϕ, a distance measure DM (0 = Euclidean distance, 1 = Hellinger distance, 2 = J-Divergence) and the type of measure TM (0 = average, 1 = maximum), the strength of influence of each parent on a variable V is calculated by performing the following operations:

```
1: procedure CalculateParentInfluence(φ, DM, TM)
2:     Let ParentsInfluence ← ∅.
3:     Let φ_S ← The posterior marginal of φ.
4:     Let Variables ← φ's variables list.
5:     Be ParentsPotential_S be a list of potentials containing the posterior
       marginal of each parent.
6:     for V ∈ Variables do
7:         Let φ' ← φ.
8:         Let Results ← ∅.
9:         for φ_paS ∈ ParentsPotential_S do
10:            if φ_paS's id ≠ V then
11:                φ' ← φ' · φ_paS.
12:                Marginalize out φ_paS from φ'.
13:            end if
14:        end for
15:        Let Res ← 0.
16:        for S ∈ φ_paS's states do
17:            Let φ'_inst ← φ' where parent V is instantiated to state S.
```

```
            Res ← 0.
            if DM = 0 then
                Res ← EuclideanDistance(φ_S, φ'_inst).
            else if DM = 1 then
                Res ← HellingerDistance(φ_S, φ'_inst).
            else
                Res ← J-Divergence(φ_S, φ'_inst).
            end if
            Insert Res into Results.
        end for
        Res ← 0.
        if TM = 0 then
            Res ← the average of the values in Results.
        else
            Res ← the maximum value in Results.
        end if
        Insert Res into ParentsInfluence.
    end for
    Return ParentsInfluence.
end procedure
```

4.10 Inference With Lazy Propagation - Example

Considering the Bayesian network BN in Figure 4.3 and the corresponding junction tree JT in Figure 4.5, we want to calculate P(F).

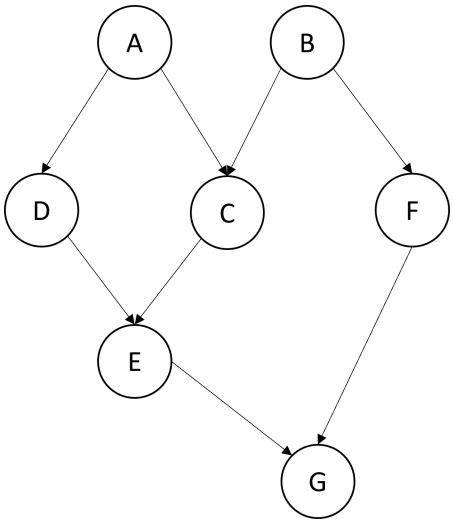

Figure 4.3: Example of Bayesian Network

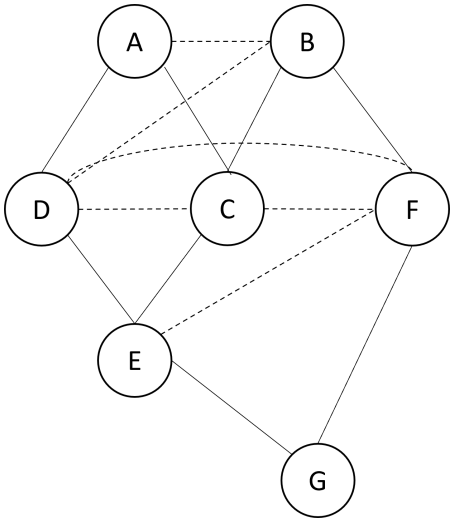

Figure 4.4: The chordal graph obtained after moralization and triangulation, using MCS-M algorithm, of the Bayesian network in Figure 4.3. The edges added during the two transformations are the dotted ones.

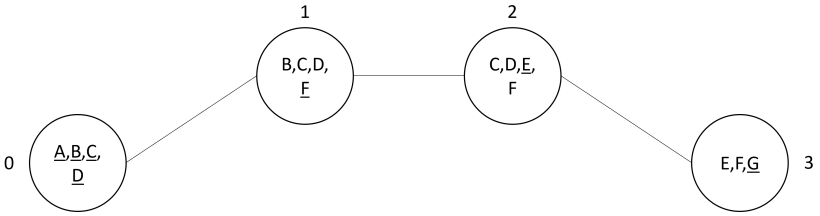

Figure 4.5: The junction tree obtained from the Bayesian network in Figure 4.3. The underlined letters indicate that the relative potential is associated to that clique.

Using the data structure defined in Section 4.4.1, Figure 4.6 and Figure 4.7 shows the contents of the messages during collect and distribute evidence phases. The names used in Section 4.4.1 are abbreviated for simplicity: PotentialDependencies = PoD, PendingDependencies = PeD, SolvedDependencies = SD.

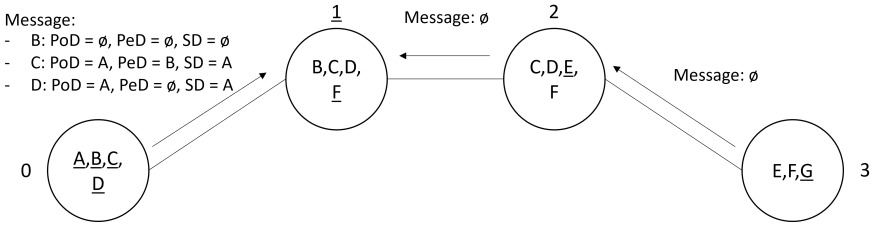

Figure 4.6: Messages sent during the Collect Evidence phase. Clique 1 is the root clique

During the Collect Evidence only the message $0 \rightarrow 1$ contains potentials. In particular, recalling Algorithm 33, ϕ_C gets potential A added to its list of dependencies as A is not present in the separator and B is still a pending dependency as it is present. The same goes for ϕ_D.

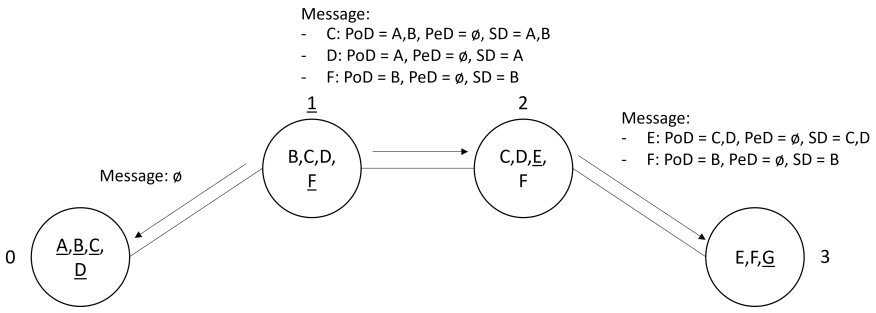

Figure 4.7: Messages sent during the Distribute Evidence phase. Clique 1 is the root clique

The same rules are applied during the Distribute Evidence: during the creation of message $1 \rightarrow 2$, potential C is completed with the addition of B, D is taken from the message received from clique 0 and F is completed with B. When creating message $2 \rightarrow 3$, potential E is completed with C and D solved and potential F is taken from clique 1's message.
At this point F's list of potential dependencies is already completed and its posterior marginal can be easily computed and returned to the user.

4.11 Third Party Libraries

In this section we will present third party libraries used during the implementation of our library.

4.11.1 Thread-Pool Library - Shoshany, Barak

One of the key point of our library is the parallelization of the algorithms to improve execution time. There are many thread pool libraries available like OpenMp, that could have satisfied our needs. Although high-level libraries are easier to use, they have the major downside of not permitting modifications to the implementation, i.e. we cannot modify the already implemented functionalities or add new ones to suit our needs.
The thread-pool implementation provided by Shoshany [19] is a fast, lightweight, single header and low-level implementation that can be easily modified to introduce optimizations to our use case. It offers many functionalities present in other libraries like loop parallelization and task submitting but also the possibility of changing the number of threads available on-the-fly.
We complemented this library with a wrapper class ThreadPoolManager. Its purpose is to allow the use of the thread-pool anywhere in the code using the singleton pattern, avoiding to pass a reference to the thread-pool to every class. Through ThreadPoolManager we also added functionalities to the original implementation of thread-pool:

- Try submitting job: during the LazyPropagation algorithm, as discussed previously, the exploration of the tree is a recursive task and thus the collect and distribute evidence tasks have to be submitted only if there are threads available. TrySubmitJob allows the submission of a job only if at least one thread is available.

- Reserving threads for parallel loop: when handling large probability distributions it is convenient to split the iteration in multiple smaller tasks but before doing it we check whether it is possible to use the thread-pool or not.

If any number of threads are available then ReserveTasksForLoop reserves the number of threads required to perform the computation. After the reservation has been made, the loop is parallelized and once completed the reserved threads are freed.

4.11.2 RapidXML - XML reader/writer library - Kalicinski, Marcin

The XDSL format file used to load and write networks is based on the XML format so any XML reader/writer library can handle it. RapidXML [20] is one of many libraries available offering good performance while being easyt f use and integrate with the project.

4.11.3 Copy-On-Write - Shao Voon Wong

The copy-on-write paradigm is a key point of our library in order to save space and time during execution. The implementation offered by Shao Voon Wong [21] implements the COW through inheritance of the templated COW class. The COW class handles the data structure that we do not want to duplicate until a modification occurs. The inheriting class can access the data structure through the pointer provided by the COW class.

4.11.4 RapidCSV - Kristofer Berggren

Using CSV to load datasets for network learning, we needed an easy to use and fast library to parse CSV files. RapidCSV [22] satisfies our needs being a single header library with reasonable performance. Its easiness of use was a determining factor for its adoption in our project.

4.11.5 Chi-squared pValue - Jacob F. W.

In our independence tests we used the chi-squared function provided by Jacob F. W. [23] to obtain the pValues needed to estimate the associations.

4.11.6 DAG Cycle Test - Techie Delight (web site)

During the Hill-climbing phase of the network learning we need to know whether the addition of an arc or the reversal of one already present introduces a cycle in the DAG. The implementation provided by Techie Delight [24] is easy to use and has the lowest possible time complexity.

4.11.7 Parallel Patterns Library (PPL) - Microsoft

We used the concurrent vector implementation contained in PPL [25] for storing information which can be accessed simultaneously by different threads, either for write or read operations, during the Lazy Propagation algorithm.

Chapter 5

Comparison with other libraries

In this section we will discuss the efficiency of our library comparing it against the well known aGrUM library. All experiments have been performed on a desktop computer equipped with a Ryzen 7 3700x@3.6 GHz (8 cores/ 16 threads) CPU and 16GB@3200 MHz of RAM using Windows 10 as operating system.

5.1 Inference Task Validation

We selected some networks from those available at `https://www.bnlearn.com/bnrepository/` (the network Tsunami was provided by our department) in order to give a representation of how our library behaves in different scenarios.

We chose two configurations to use for testing our library against aGrUM: the first one where the Lazy Propagation does not use the parallelization and the 16 threads thread pool is used just to perform the multiplications of CPTs, and the second one where a thread pool of 200 threads is used and the Lazy Propagation uses the parallelization. The first configuration is used to resemble as closely as possible how aGrUM works, while the second one serves to show how the library behaves when handling large networks on CPUs with a higher number of threads, although this is just representative as in our case this introduces a lot of context switching.

These two configurations show how different networks behaves when parallelization is enabled or not in the Lazy Propagation algorithm, demonstrating that in certain scenarios the parallelization brings little to no benefit or even slows down the execution.

Following the comparison with aGrUM, we will also demonstrate the scalability of

Networks List			
Network Name	Number of nodes	Number of arcs	Number of parameters
Alarm	37	46	509
Andes	223	338	1157
Barley	48	84	114005
Diabetes	413	602	429409
Hailfinder	56	66	2656
Hepar2	70	123	1453
Mildew	35	46	540150
Munin	1041	1397	80592
Tsunami	1277	2123	11260193

Table 5.1: List of networks used to test the performance of our library. For each network it is specified the number of nodes, the number of arcs and the number of parameters. Note that the number of parameters refers to the number of combinations of variables' states needed to represent the joint probability distribution of the network.

our library showing how it behaves when 16, 50, 100 and 200 threads are available to the parallel Lazy Propagation. All graphs are in log-10 scale.

5.1.1 Execution Time

One of the main objectives of our work was to create a more efficient library in terms of execution time compared to those already available. In this test, we used the MWCH triangulation method in our library which is the same used by aGrUM. The execution time values in the following graphs are calculated as the mean over 30 iterations of the whole inference task (reading the network, triangulation, etc...). Figure 5.1 shows how our implementation is more efficient in every execution compared to aGrUM, reducing by half the time required by execution in almost every network and some even more. In the Tsunami case, aGrUM was not able to finish execution crashing before the end, while in the Mildew case the execution finished in about 26 seconds but the result was completely wrong so we did not consider it valid for the test, although this is not the only case that happens that aGrUM returns the wrong result. However, the correctness of the results will be discussed later in the Chapter.
The graph also shows that generally the serial Lazy Propagation is faster by a few milliseconds (up to 6 seconds in the Mildew case) on small networks (up to few hundreds of nodes). On large networks, with hundreds of nodes and more, the parallel Lazy Propagation is much faster, for example in the Munin case the time required by the parallel configuration is less than half the time required by the

serial configuration.

Figure 5.1: Execution time comparison between aGrUM and our library. In parenthesis the name of the variable requested. No barren variables used.

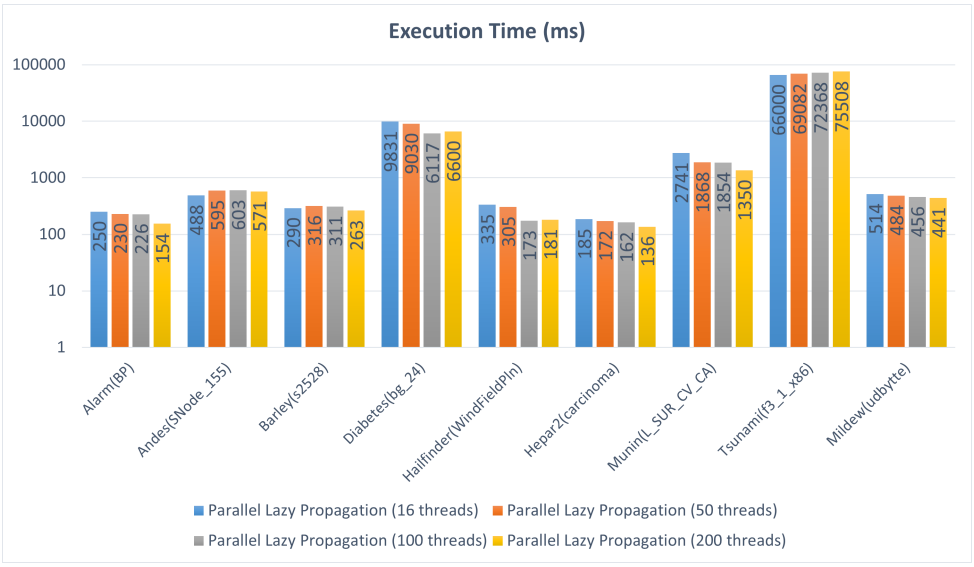

Figure 5.2: Execution time comparison between parallel Lazy Propagation with 16, 50, 100 and 200 threads using the MCS-M triangulation method. In parenthesis the name of the variable requested. No barren variables used.

The reason behind this difference in behaviour between the serial and parallel Lazy Propagation is mainly due to the size of the networks in accordance to our testing: smaller networks benefit less from a higher number of threads during the recursive task of the Lazy Propagation leading to too many context switches. On the other hand, on bigger networks the parallel Lazy Propagation benefits more from a bigger thread pool during the recursive task allowing for a faster exploration of the junction tree. But, depending on the structure of the network, also big networks can incur in a bottleneck caused by too many context switches if the junction tree is not ramified enough.

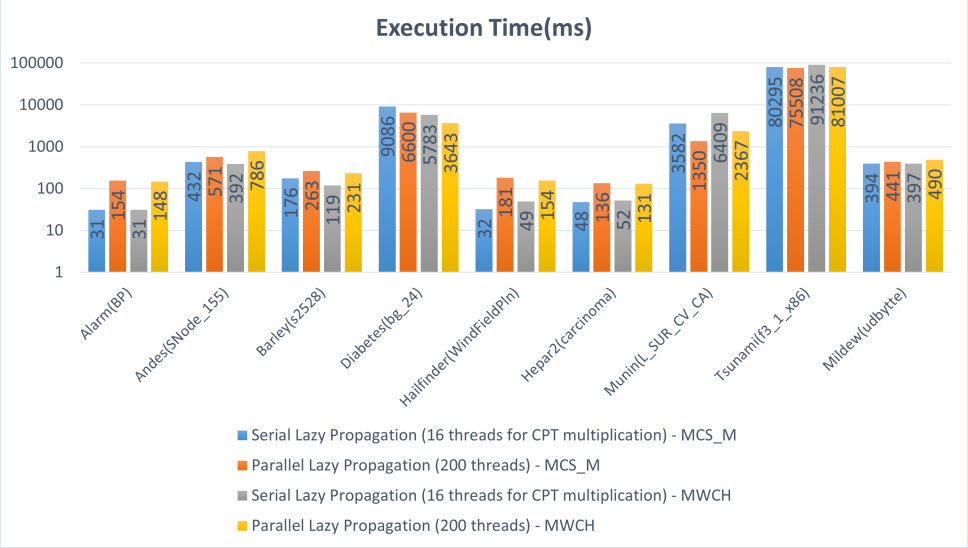

Figure 5.3: Execution time comparison between MCS-M and MWCH triangulation methods in our library. In parenthesis the name of the variable requested. No barren variables used.

Figure 5.2 shows how our library scales with different thread pool sizes. It can be noticed that increasing the number of threads on smaller networks reduce to some extent the execution time, probably because the higher thread counts of the program fakes a higher load to operating system resulting in more resources allocated to its execution but this is still not enough to beat the serial execution. On the other hand, bigger networks shows different trends: Diabetes and Munin sees a reduction in execution time with a bigger thread pool while Tsunami has slightly worse results with 200 threads compared to 16 threads because the obtained junction tree has only a few ramifications. Compared to the serial execution with 16 threads for CPT operations, only Tsunami and Munin have an improvement while Diabetes has a slower execution time. This shows that to obtain good performance

with the inference task it is important to configure the library correctly and this can be done by dedicating only a limited number of threads to the recursive task of the Lazy Propagation. This feature will be discussed in Section 6.2.3 as a future improvement to the library. Figure 5.3 compares the execution time required when using either the MCS-M or the MWCH triangulation method (the Lex-M algorithm has not been considered for brevity, but the results would be similar to those of the MCS-M since its just a variant). On small networks the triangulation method has little to no impact at all on the performance of the Lazy Propagation, either serial or parallel, while on larger networks it is not obvious which triangulation to use. With the Tsunami network the MCS-M triangulation reduced execution time by about 10% compared to the MWCH triangulation, while with the Diabetes network the MWCH triangulation almost halved the execution time compared to the MCS-M. In order to determine which triangulation suits one case the best, the user should test all three of them, Lex-M, MCS-M and MWCH, and use the faster one.

5.1.2 Memory Usage

Keeping the memory usage to a minimum during the execution of the Lazy Propagation was one of our main objectives for this project since, as already discussed, we want this library to be used on as many devices as possible.

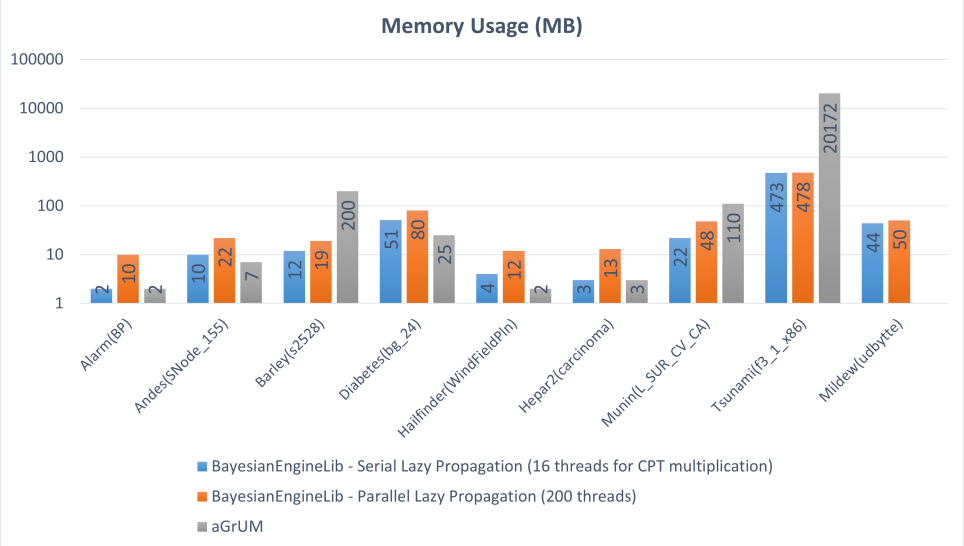

Figure 5.4: Memory usage comparison between aGrUM and our library. In parenthesis the name of the variable requested. No barren variables used.

The memory usage values are obtained through the diagnostic tools of Visual Studio executing the library in release mode.
Figure 5.4 shows the memory used by our library with the MWCH triangulation and compares it with the memory used by aGrUM (the MWCH triangulation method has been used in our library since it is the same used by aGrUM). The memory used by the Lazy Propagation serial and aGrUM is comparable on small networks, with the exception of the Barley network, were aGrUM uses almost 200MB, and Munin were aGrUM exceeded 100MB. In small and medium sized networks our library ties with aGrUM but it is in large networks, like the Tsunami network, that our library outperforms aGrUM where, although aGrUM was not able to finish its execution, the memory used at the time it crashed peaked at 19.7GB while our library used at maximum 535MB throughout the whole execution. The result of the Mildew network produced by aGrUM has been ignored for the same reason stated in Section 5.1.1.
The parallel Lazy Propagation uses more memory due to the higher number of threads in the thread pool, and thus a higher number of thread instances to keep track of, and the use of additional structures required for the correct functioning of the algorithm.
The memory usage using either the MCS-M or the MWCH triangulation is almost identical with some advantage towards the MWCH in larger networks.

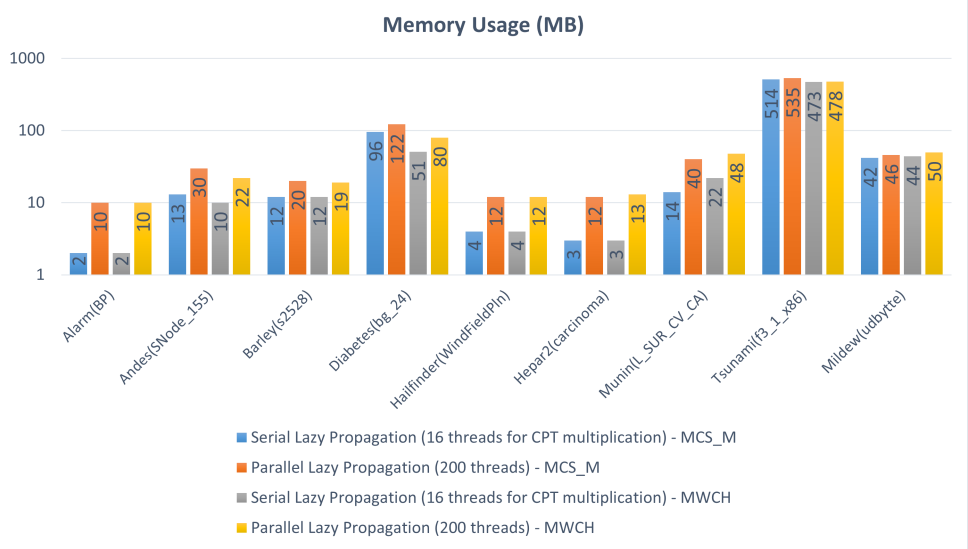

Figure 5.5: Memory usage comparison between MCS-M and MWCH triangulation methods in our library. In parenthesis the name of the variable requested. No barren variables used.

These results show the effectiveness of using the COW paradigm and a compact

representation for sparse CPTs, allowing the library to be used on devices with limited amount of memory, like IoT devices.
When looking at the scalability of our network with different thread counts, as expected, the memory usage increases. This is due to a higher number of thread pointers used by the thread pool and the increased size of the data structures used by the parallel Lazy Propagation. This behaviour is shown in Figure 5.6.

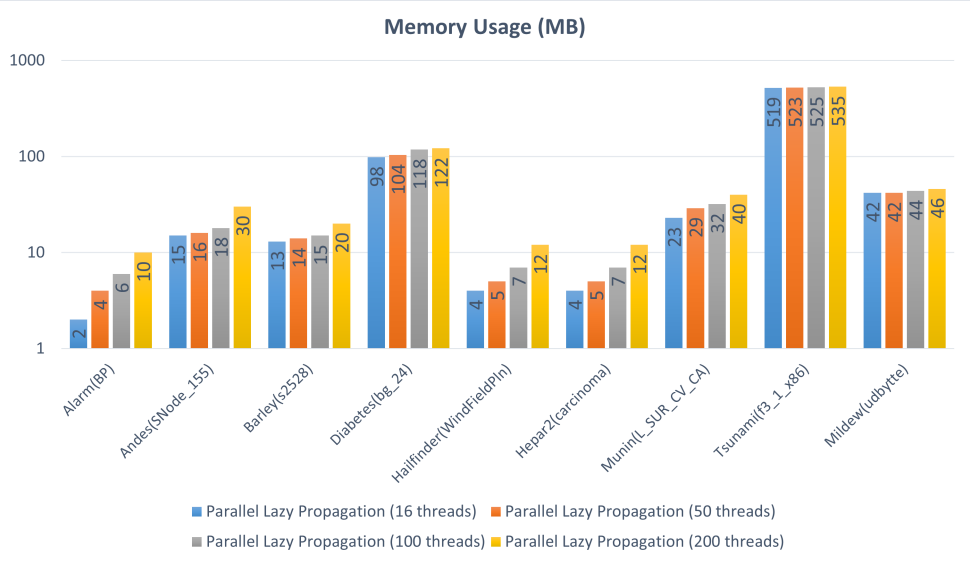

Figure 5.6: Memory usage comparison between parallel Lazy Propagation with 16, 50, 100 and 200 threads using the MCS-M triangulation method. In parenthesis the name of the variable requested. No barren variables used.

5.2 Network Learning Task Validation

In order to compare the quality of our network learning algorithm, we used the datasets available at `https://www.bnlearn.com/documentation/man/index.html` involving categorical variables since, at the moment, is the only type of variables we support. The datasets used are: Alarm, Asia, Coronary, Insurance and Lizards. The comparison, however, is not completely fair since bnlearn provides different algorithms (for this test we used the MMHC with the BIC score for the Hill-climbing phase in the bnlearn library) from those we implemented in our library. Although, as discussed in Section 2.7.1, the FastCHC we implemented in the Hill-climbing phase generates networks with overall lower scores, in this small set of datasets the networks we generate have a quality close to those generated by bnlearn, with the exception of a couple of cases where bnlearn outperforms our implementation.

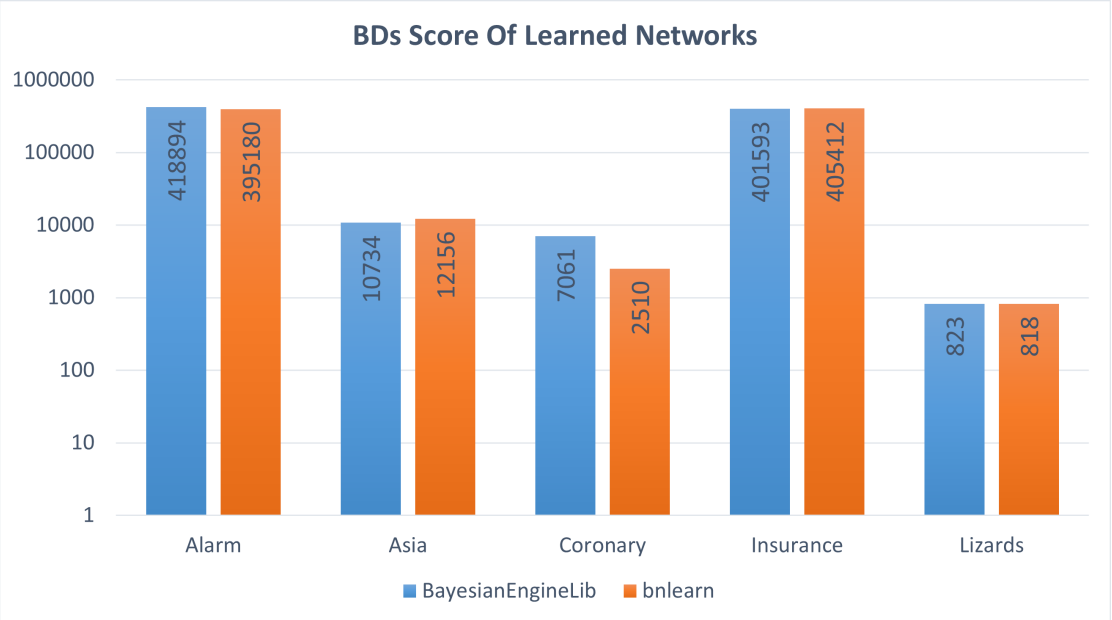

Figure 5.7: BDs scores of networks learned from our library and from the package bnlearn(R). The scores are negated in order to be visualized in log scale. The lower the score the better the network is.

5.3 Result Correctness

We did not consider GeNIe in the previous tests since it is a desktop application and could not be tested in the same way we have done for both aGrUM and our library, but, like aGrUM, it is one of the most known applications for operating on Bayesian networks and as such the user expects that the application gives correct results. During the development of our project we noted that in many cases both aGrUM and GeNIe gave different results so it became hard to say which was the correct one. After a lot of testing in order to individuate the possible cause of these differences, we believe that both programs perform their marginalization process in a wrong manner at one point or another. In the GeNIe case, using as example the Barley network and considering the posterior marginal of the variable "jordn", the result given by GeNIe can be replicated by multiplicating all three parents of the variable together, "potnmin", "nmin" and "aar_mod", obtaining the potential Φ_{PNA} and then multiplicating only once the variable "jordtype" to Φ_{PNA} but marginalizing it three times. Our opinion is that the library is not able to keep track correctly of the parents of a potential, especially when n variables with the same parent are multiplied together and thus n instances of that parent variable should be present in the list of parent variables of the resulting potential,

but instead only one instance of the variable is kept at a time. We believe this happens because they start solving their potentials by multiplying each parent with the target variable, marginalize them and proceed like this with the ancestors until finished. This problem could have been easily avoided using an approach of marginalization like our or by designing a data structure for correctly storing the variables needed by a potential.
We believe that the same problem is present in the aGrUM library but we were not able to replicate their results so it is only an hypothesis. In their case the results where much closer to our in some instances where in others their values were completely off. Although we considered the results of aGrUM as valid for time and memory comparisons, in general both libraries are not 100% reliable.
We want to point out also that we used the academic version of GeNIe so we do not know if the business version suffers from the same problem or not.

Chapter 6

Conclusions And Future Work

6.1 Conclusions

In this thesis we introduced a new library able to operate on Bayesian networks and perform tasks such as inferencing, learning networks and measuring the influence between nodes. Basing our work on some of the most recent researches conducted in the sector and using modern C++ programming techniques, we were able to create an efficient and scalable library which is capable of handling very large networks in reasonable time and can even be used on IoT devices.
Starting from the theories introduced in Chapter 2, we gave a brief introduction to the topic of Bayesian networks and on which theories our library is based. Additionally we also nominated a few examples of Bayesian networks library which have been taken as reference for our work. In Chapter 3 we described the structure of our library defining for each class, or group of classes, the features contained giving, in Chapter 4, a detailed discussion on the features we implemented, explaining also the differences with the solution proposed in the paper from which they have been taken. Additionally, we also described the data structures needed to perform specific tasks and their usage during the execution of the task. In Chapter 5 we validated the quality of our work against some of the most well known open source libraries. Our inference capabilities outperforms those of the aGrUM library while giving more precise results, which we discovered being one major downside of GeNIe. Our network learning algorithm produces results that are comparable with those produced by bnlearn which is again a well known R package in the sector.
During the work of this thesis we achieved many of our goals, such as efficiency and scalability, but we already have plans and ideas to test for improving the library and its set of features in areas like network learning with continuous variables,

GPU based inferencing and strength of influence measures.

6.2 Future Work

This section is dedicated to introduce the future changes and features that will be brought to the library.

6.2.1 Extending Support To Other Bayesian Network File Formats

At the moment the only file format supported is the XDSL, which is based on the XML format. In future we will extend the support to other formats such as BIF, NET and DSL which are some of the more commonly used. This saves the user from converting its file to the XDSL format.

6.2.2 Introducing Parallelism To CPT Operations

CPTs multiplication is the only parallelized operation. The check performed to determine if a CPT is sparse and its subsequent transformation, marginalization and instantiation of CPT can also benefit from parallelization when the probability distribution is big enough. The easiest way of parallelizing these tasks is to create many tasks where each one operates on a smaller part of the vector, for example we can assume that each task will operate on 5000 elements or more. Obviously the degree of parallelization of the tasks is bound to the number of threads allocated to the thread pool. This will be a benefit for very large networks handled by computers with hundreds of threads if not super computers.

6.2.3 Improving Lazy Propagation Parallelism

As discussed in Section 4.5.1 the parallel version of Lazy Propagation is not optimal and there are a few aspects that can be improved. The first one is the way in which the parallelization is handled. In the current implementation, the main thread of library starts the recursive algorithm and then each parallel job decide whether to submit a new job to the thread pool or not and the main thread executes the jobs that are not submitted to the thread pool. The problem present in this solution is that both the main thread and the thread that creates the job do not check whether the thread pool can receive new jobs and so all the jobs that were not submitted are executed in either the main thread or the thread that created them, potentially leaving threads doing nothing. Therefore a check should be performed to determine whether jobs that were not previously submitted to the thread pool can now be submitted.

Another way of improving the parallelism could be dedicating only a limited number of threads to the LazyPropagation, leaving more threads available for performing CPT operations (multiplications, marginalizations and instantiations) which in general are the more demanding tasks. This partitioning can be done either in a static way, i.e. the user can specify the partition, or dynamically estimating which operations would benefit more from having more threads available.

6.2.4 Improving PC Algorithm Parallelism

Our implementation of the PC algorithm uses the idea presented in [7] for parallelization but the parallelization is done at nodes level. This solution has the downside of leaving threads inactive when all the independence tests of one node have been executed and other threads are still performing tests on the assigned node. We will improve the current implementation by fully exploiting the idea of splitting the independence tests in homogeneous groups independent from which node they are generated from. Additionally the option of indicating the maximum number of tests for a thread will be added in order to reduce memory usage on systems with limited amount of memory.

6.2.5 Improving FastCHC Algorithm

In Section 2.7.1 we introduced the FastCHC algorithm used in the Hill-climbing scoring phase of the PCHC and anticipated that the base implementation of the FastCHC produces networks of lower quality compared to those produced by a classical Hill-climbing heuristic. In [11] a few modifications are proposed in order to increase substantially the score of the networks learned by the FastCHC retaining the same execution time as the normal FastCHC. From the comparison between our library and the bnlearn package we established that the normal FastCHC already obtains results in par with a well known library, but we believe that these modifications can generate even better networks.

6.2.6 Improving Support To Networks Learning

The focus of our work for this thesis was to introduce support to network learning with discrete variables. In future we will extend our learning features also to continuous networks by adding independence tests and scoring metrics specific to them.

6.2.7 Improving Support To Strength Of Influence

In Sections 2.8 and 3.10 we discussed which strength of influence measures we implemented, i.e. the influence that each parent has on a child. In future we will

expand the possible measures implementing those described in [14] like measuring the influence that a child has on its parents and the influence, in both cases, in presence of observations in the network.

6.2.8 Adding Support To GPU

In the modern era, along with the increase in computing capacity of CPUs, GPUs have become much more powerfull with thousands of cores. By adding support to inference through GPUs we want to improve the performance of our library on large networks (>1000 nodes) ultimately reducing the time required to perform inference compared to performing it with the CPU.

Bibliography

[1] Todd Andrew Stephenson. «An Introduction to Bayesian Network Theory and Usage». In: (2000). URL: http://infoscience.epfl.ch/record/82584 (cit. on pp. 3, 5).

[2] David Heckerman. «A Tutorial on Learning With Bayesian Networks». In: *CoRR* abs/2002.00269 (2020). arXiv: 2002.00269. URL: https://arxiv.org/abs/2002.00269 (cit. on p. 7).

[3] Anders L. Madsen and Finn V. Jensen. «Lazy propagation: A junction tree inference algorithm based on lazy evaluation». In: *Artificial Intelligence* 113.1 (1999), pp. 203–245. ISSN: 0004-3702. DOI: https://doi.org/10.1016/S0004-3702(99)00062-4. URL: https://www.sciencedirect.com/science/article/pii/S0004370299000624 (cit. on pp. 7, 9–11, 13–15, 29, 39, 41, 47, 49).

[4] Judea Pearl. *Probabilistic reasoning in intelligent systems : networks of plausible inference / Judea Pearl*. eng. 2nd rev. printing. The Morgan Kaufmann series in representation and reasoning. San Francisco: Kaufmann, 1988. ISBN: 1-55860-479-0 (cit. on p. 9).

[5] Dan Geiger, Tom S. Verma, and Judea Pearl. «d-Separation: From Theorems to Algorithms». In: *CoRR* abs/1304.1505 (2013). arXiv: 1304.1505. URL: http://arxiv.org/abs/1304.1505 (cit. on p. 10).

[6] Uffe B. Kjærulf and Anders L. and Madsen. *Probabilistic Networks — An Introduction to Bayesian Networks and Influence Diagrams*. May 2005. URL: http://people.cs.aau.dk/~uk/papers/pgm-book-I-05.pdf (cit. on p. 12).

[7] Thuc Duy Le, Tao Hoang, Jiuyong Li, Lin Liu, Huawen Liu, and Shu Hu. «A Fast PC Algorithm for High Dimensional Causal Discovery with Multi-Core PCs». In: *IEEE/ACM Trans. Comput. Biol. Bioinformatics* (Sept. 2015), pp. 1483–1495. URL: http://arxiv.org/abs/1502.02454 (cit. on pp. 16, 30, 75, 77, 103).

[8] Ioannis Tsamardinos, Laura E. Brown, and Constantin F. Aliferis. «The max-min hill-climbing Bayesian network structure learning algorithm». In: (2006). DOI: 10.1007/s10994-006-6889-7. URL: https://doi.org/10.1007/s10994-006-6889-7 (cit. on pp. 16, 19).

[9] Michail Tsagris. «A New Scalable Bayesian Network Learning Algorithm with Applications to Economics». In: (2020). DOI: 10.1007/s10614-020-10065-7. URL: https://doi.org/10.1007/s10614-020-10065-7 (cit. on pp. 16, 17, 19).

[10] Markus Kalisch and Peter Buehlmann. «Estimating high-dimensional directed acyclic graphs with the PC-algorithm». In: (Oct. 2005). URL: https://arxiv.org/abs/math/0510436 (cit. on p. 18).

[11] Jacinto Arias, José A. Gámez, and José M. Puerta. «Structural Learning of Bayesian Networks Via Constrained Hill Climbing Algorithms: Adjusting Trade-off between Efficiency and Accuracy». In: *Information Processing Letters* (2014). DOI: https://doi.org/10.1002/int.21701. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/int.21701 (cit. on pp. 19, 103).

[12] Marco Scutari. *Dirichlet Bayesian Network Scores and the Maximum Relative Entropy Principle*. 2018. arXiv: 1708.00689 [math.ST]. URL: https://arxiv.org/abs/1708.00689 (cit. on pp. 20, 21).

[13] Zhiwei Ji, Qibiao Xia, and Guanmin Meng. «A Review of Parameter Learning Methods in Bayesian Network». In: *Advanced Intelligent Computing Theories and Applications*. Ed. by De-Shuang Huang and Kyungsook Han. Cham: Springer International Publishing, 2015, pp. 3–12. ISBN: 978-3-319-22053-6. URL: https://link-springer-com.ezproxy.biblio.polito.it/book/10.1007%2F978-3-319-22053-6 (cit. on pp. 21, 22).

[14] J.R. Koiter. «Visualizing Inference in Bayesian Networks». MA thesis. University of Pittsburgh: Faculty of Electrical Engineering, Mathematics, and Computer Science, Department of Man-Machine Interaction, 2006. URL: http://www.kbs.twi.tudelft.nl/docs/MSc/2006/JRKoiter/thesis.pdf (cit. on pp. 22, 23, 25, 30, 85, 104).

[15] Scutari Marco. *bnlearn - An R Package For Bayesian Network Learning And Inference*. 2022. URL: https://www.bnlearn.com/ (visited on 01/21/2022) (cit. on p. 25).

[16] Uffe Kjærulff. *Triangulation of Graphs – Algorithms Giving Small Total State Space*. Tech. rep. 1990. URL: http://cse.unl.edu/~choueiry/Documents/Kjaerulff-TR-1990.pdf (cit. on pp. 28, 36).

[17] Anne Berry, Jean R. S. Blair, and Pinar Heggernes. «Maximum Cardinality Search for Computing Minimal Triangulations». In: *Graph-Theoretic Concepts in Computer Science*. Ed. by Gerhard Goos, Juris Hartmanis, Jan van Leeuwen, and Luděk Kučera. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 1–12. ISBN: 978-3-540-36379-8. DOI: 10.1007/s00453-004-1084-3. URL: https://link.springer.com/article/10.1007/s00453-004-1084-3 (cit. on pp. 28, 36).

[18] Anne Berry and Romain Pogorelcnik. «A simple algorithm to generate the minimal separators and the maximal cliques of a chordal graph». In: *Information Processing Letters* 111.11 (2011), pp. 508–511. ISSN: 0020-0190. DOI: https://doi.org/10.1016/j.ipl.2011.02.013. URL: https://www.sciencedirect.com/science/article/pii/S0020019011000536 (cit. on p. 28).

[19] Barak Shoshany. *bshoshany/thread-pool: thread-pool v2.0.0*. Version v2.0.0. Aug. 2021. DOI: 10.5281/zenodo.5202857. URL: https://doi.org/10.5281/zenodo.5202857 (cit. on p. 89).

[20] Marcin Kalicinski. *RapidXML*. Version v1.13. 2009. URL: http://rapidxml.sourceforge.net/index.htm (cit. on p. 90).

[21] Shao Voon Wong. *Copy-On-Write Base Class Template*. Version v1. 2020. URL: https://www.codeproject.com/Tips/5261583/Cplusplus-Copy-On-Write-Base-Class-Template (cit. on p. 90).

[22] Kristofer Berggren. *RapidCSV*. Version v1. 2021. URL: https://github.com/d99kris/rapidcsv (cit. on p. 90).

[23] Jacob F. W. *Chi-squared pValue*. Version v1. 2012. URL: https://www.codeproject.com/Articles/432194/How-to-Calculate-the-Chi-Squared-P-Value (cit. on p. 90).

[24] Techie Delight (web site). *DAG Cycle test*. Version v1. URL: https://www.techiedelight.com/check-given-digraph-dag-directed-acyclic-graph-not/ (cit. on p. 90).

[25] Microsoft. *Parallel Patterns Library (PPL)*. Version v1. URL: https://docs.microsoft.com/en-us/cpp/parallel/concrt/parallel-patterns-library-ppl?view=msvc-170 (cit. on p. 91).