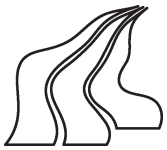
Learning Inference-friendly Bayesian Networks

Using Incremental Compilation

Dat 6, Spring 2008.

Martin Karlsen
Søren Pedersen



Aalborg University
Department of Computer Science

Aalborg University
Department of Computer Science

TITLE:

Learning Inference-friendly Bayesian Networks
Using Incremental Compilation

THEME:

Machine Learning

PROJECT PERIOD:

1/2/2008-5/6/2008

PROJECT GROUP:

d630a

GROUP MEMBERS:

Martin Karlsen
Søren Pedersen

SUPERVISOR:

Thomas D. Nielsen

NUMBER OF COPIES: 4

NUMBER OF PAGES: 96

CONCLUDED: 5/6/2008

SYNOPSIS:

This report describes a project with the aim of exploring structural learning of Bayesian networks. Specifically the complexity of the generated network, as a result of the chosen learning method.
We examine the junction tree method for doing propagation in Bayesian networks, describing the steps in compiling the junction tree from the network structure. From this analysis we learn that one cause of complexity in junction tree is the size of the cliques.
A score function which scored a network directly on the combined size of the cliques and the log-likelihood are proposed. This function uses a parameter to weight whether the complexity versus the likelihood. This function uses incremental compilation to avoid having to re-triangulate the entire junction tree for each candidate network.
This score function and regular BIC scoring (also augmented with a weighing parameter) was tested for precision and inference time. This analysis shows that there is a gain in using the size of the junction tree as a component in the scoring of learned nets, as these net scored using this function was most often faster when used for inference, and in some cases even able to produce usable networks where the networks learned with BIC-scoring proved too complex.

This report was written by:

Martin Karlsen	Søren Pedersen

Contents

Chapter 1

Introduction

In many areas within research and engineering, probabilistic models play a central task when it comes to modeling and reasoning about certain phenomenons, investigating influence, or predicting the outcome of certain events. The Bayesian network model class is one such probalistic model. It is a graphical language that can model the probabilistic and causal relationships presented in a given domain. It can be used for reasoning about any attribute in the domain, given observations about som eof the attribute variables. This has proven to be usefull within a number of areas: diagnosis [AWFA87], [BHP+92], [DH92], process optimization [POL07], [FN91], forecasting [Abr94], [GPC+94], [CSG04], automated vision [LAB90], classification [CS96], data mining [Hec97], and spam filtering [Gra03].

A Bayesian network consists of an acyclic directed graph and a set of local probability distributions. Each node in the graph represents a random variable. Variables models the various entities of a given domain, and may denote an attribute, feature, or hypothesis which we may be uncertain about. That is a variable has a set of possible values that it can take, and we are uncertain about which specific value it has. The relationships among the entities of a domain are modelled as directed links between the nodes in the graph. The links represent probabilistic dependencies, namely whether the occurnce of one event will change the belief about the occurence of another event. The semantics of the directed links may specifically represent causal relationships; (one entity is the cause of another entity) but this is not a requirement.

A Bayesian network can be constructed in a number of ways. One way is that the network is constructed by a human domain expert, such that the graphically structure as well as the parameters are determined by the expert. One might choose to construct the graphical structure by hand, and then use a dataset for learning parameters. If the dataset is complete the task

of learning the parameters can be done by merely counting frequencies, but even with incomplete data, good approximate estimations can be computed [DLR77].

A network may also be constructed through so called structural learning. There are several possibilities: score based learning, where the network structure as well as the parameters are "extracted" from the dataset, resulting in the structure and a set of parameters that is the most likely to have "generated the data". Or constraint based learning where the structure represents conditional independencies presented in the dataset based on a number of statistical tests. Structural learning is dealt with in greater detail in Chapter 4.

Learning a Bayesian network from data may serve many different purposes. The analyst may learn models with the purpose of exploratory analysis such as discovering dependencies in data. From an engineering perspective this analysis may be less interesting, and instead focus will be on producing models tailored towards inference that can be put to use in an application context.

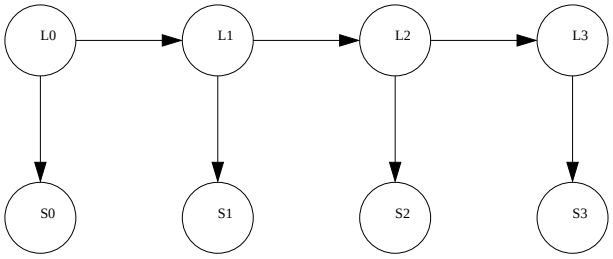
Regardless of the purpose, be it research or from an engineering perspective, the resulting model may prove to be impossible to use for probability updating, or too complex to visually comprehend. This may be because of model size or other factors.

In this project we work with a structural learning approach that may satisfy the requirements for both of these perspectives. The starting point of the work is the preliminary work described in the report [KP07], in which we explore structural learning of Bayesian networks with specific relation to the complexity of learned models. But let us first look a little more at some of the caveats of structural learning.

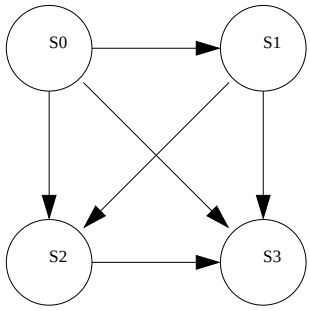
1.1 Structural Learning Caveats

Structural learning may become problematic if the domain that we are trying to learn contains a set of unobserved latent variables, where the presence and structure of these variables is unknown - or if the domain is simply too complex. To illustrate this problem consider learning a model of an industrial production plant from a dataset of sensor readings: We wish to learn the structure from a dataset of observations which is sampled from the true domain represented by the network in Figure 1.1(a). Only the the sample variables $S0, \ldots, S3$ are observed, and the latent variables $L0, \ldots, L3$ as well as their structure is unkown to us. Then the structure that we end up with is the very complex network in Figure 1.1(b). *(This is the case*

as none of the variables $S0, \ldots, S3$ block information and they all seem to be dependent on each other - the explanation for this can be found in the property of d-separation covered in Chapter 2). A complex structure such as the one in Figure 1.1(b), where all variables seems to be dependant, could be a strong indication that the actual domain we are trying to learn contain unobserved latent variables.



(a) The sampled domain. The structure as well as the latent variables $L0, \ldots, L3$ is unknown to us, and only variables $S0, \ldots, S3$ are observed.



(b) The network that we end up with after performing structural learning. The complexity of the network could indicate unobserved latent variables (in this case it does.

Figure 1.1: Unobserved latent variables makes the learned structure overly complex, as all variables seems to be dependant on each other.

There are several ways to deal with problems where the learned structure becomes too complex, one being to introduce a set of structural constraints such as naive Bayes [DP97], tree augmented naive Bayes [FGG97] or Chow Liu trees [CL68], or introduce additional variables to approximate the actual set of hidden variables that could account for the strong dependencies found [Hec95], [ELFK00]. Additionally it seems as the problem could also be dealt

with by introducing constraints in the form of an upper limit on the number of parents any given node can have.

Typically the term *complexity* refers to the density of the Bayesian network structure, which we will refer to as *network complexity*. The network complexity might not be the most accurate definition for our purposes. As the task of inference, for example inserting evidence and calculating marginal probabilities, is often performed in a secondary structure known as a junction tree *(junction trees are covered in more detal in Chapter 3)*, the computational effort required is thus determined by the complexity of the junction tree, rather than the complexity of the Bayesian network - which we will refer to as the *effective complexity*. This means that imposing constraints on the Bayesian network structure does not neccessarily result in a simpler junction tree, and in term a degration of effective complexity (except in the case where the constraints impose a tree-like structure of the Bayesian network).

Another way of dealing with the problem of minimizing effective complexity could be to have the domain expert cut away "less important" links, in order to simplify the model and reduce model complexity - which is in fact very similar to the structural learning method that we propose in this report!

1.2 Preliminary Work

The learning method we proposed in [KP07] makes it possible to explicit trade complexity for precision and vice versa. One could think of it as being some kind of "unsupervised re-configuration" of network links - just like when the domain expert cut away links. Therefore we think it might be better to use the effective complexity as the criterion for such a learning method. This is exactly what the learning method proposed in [KP07] does. Preliminary structural learning experiments suggest that by using this approach it is possible to trade an acceptable reduction in model precision in return for a network with a low effective complexity (which we refer to as an *inference-friendly* network). More details about the results from these experiments can be found in Appendix A. Although it seems that the method was better at producing inference-friendly networks, it suffered in performance - especially for large domains. The lack of performance can be explained by the numerous junction tree re-compilations required during a structural learning run.

1.3 Our Goal

The purpose of this project is to improve upon the learning method from [KP07] as to the lack of performance on bigger domains. This means we seek to propose a learning method that makes it possible to trade model precision in return for lower effective complexity in the learned model. In addition we wish that *the method should be compuationally affordable*, and that *the tradeoff between effective complexity and precision should be acceptible for inference-friendly networks*.

We achieved this goal in a number of steps which is covered in the different parts of this report.

Part 1 covers the basic foundations for Bayesian networks and inference.

We introduce Bayesian networks with a little more detail in Chapter 2. The problem of inference is covered in Chapter 3.

Part 2 introduce structural learning, explores incremental compilation and concludes by proposing the desired learning method which benefits from it.

Details of structural learning are explored in Chapter 4. In in Section 4.3 we propose the JTC score function, which motivates the need for junction tree decomposition and in turn incremental compilation.

We explore incremental compilation in Chapter 5, and in Chapter 6 we explore a number of possible optimizations that could benefit structural learning using using the proposed JTC score function.

Part 3 puts the proposed method to the test and concludes on the results.

In Chapter 7 we perform a number of structural learning experiments which we subsequent analyze and this leads us to the conclusion in Chapter 8.

Part I

Probalistic Graphical Models and Inference

Chapter 2

Bayesian Networks

A Bayesian network is a probabilistic model, and we need to introduce a few probabilistic concepts, namely *probability functions*, *independence* and the *chain rule* to fully appreciate the model. These sections are slightly modified copies of sections of [KP07].

2.1 Probability Functions

Probability functions expresses belief about variables. Variables can be though of as experiments or observations, where each state represents a possible outcome. Each variable has a *finite* number of *mutually exclusive* states, and these states should be *exhaustive*. A probability function is a function from two sets of variables to a real number: $P : V \times C \rightarrow [0; 1]$, where V is the set of variables about which we are expressing probabilities, and C is the set of conditioning variables (which might be empty). The probability ranges from 0, impossible - to 1, absolute certainty. Probability functions are written as $P(A)$ (read as *the probability of* A) if the there are no conditioning variables, and $P(A|B)$ (*"the probability of* A *given* B*"*) where B are the conditioning variables. The domain of $P(A|B)$ is denoted $dom(P(A|B)) = \{A, B\}$. The probability function $P(A)$ for a variable A with states $a_1, \ldots, a_n$ is a list of probabilities $x_1, \ldots, x_n$ where x_j represents the probability that A is in state a_j. An example table is shown in table 2.1. Such a function is called the *marginal* or *prior* probability. Since the states of A are exhaustive, all entries in the table $P(A)$ sum to 1, that is $\sum_{j=1}^{n} x_j = 1$.

For variables A with states $a_1, \ldots, a_n$ and B with states $b_1, \ldots, b_m$, $P(A|B)$ contains $n \cdot m$ conditional probabilities $P(a_i|b_j)$. This can be represented as a $n \times m$ table where cell j, k contains the probability that A is in state a_j given that we know B is in state b_k. Such a table is illustrated in table 2.2.

State	*Probability*
a_1	0.3
a_2	0.7

Table 2.1: Probability table for $P(A)$

states	b_1	b_2	$\cdots$	b_m
a_1	0.33	0.11	$\cdots$	0.46
a_2	0.37	0.25	$\cdots$	0.24
$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
a_n	0.21	0.32	$\cdots$	0.03

Table 2.2: Probability table for $P(A|B)$

The states for B is exhaustive: for each state b_k , all associated states of A must sum to one: $\sum_{j=1}^{n} P(A = a_j|B = b_k) = 1$ for all values of k.

The *joint probability* for variables A and B is written as $P(A, B)$. This expresses the probability of *both A and B* . Similarly to conditional probability, a joint probability can also be represented as a table with $n \cdot m$ entries for states $A_1, \ldots, a_n$ of A and states $b_1, \ldots, b_m$ of B, such a table is shown in Table 2.3. In a *joint probability table* all entries must sum to 1: $\sum_{j=1}^{n} \sum_{k=1}^{m} P(A = a_j, B = b_k) = 1$.

From a joint probability table we can "extract" probabilities for smaller sets of variables. If we want to find the probability for A, we calculate the total the probability for each state $a_1, \ldots a_n$ of A, by summing the probability for all entries in the table where A is in state a_i. In the example table in Table 2.3, this would means we would calculate

$$P(a_i) = \sum_{j=1}^{m} P(a_i|b_j)$$

where m is the number of states in B. When this calculation has been done for all states of A we have $P(A)$. We call this operation *marginalization* and say that B was *marginalized out*

2.2 Independence

Independence among variables indicate whether their states influence each other: If two variables are *independent* an occurrence of one event for one of the variables does not affect the probability for the events for the other

states	b_1	b_2	$\cdots$	b_m
a_1	0.2	0.04	$\cdots$	0.06
a_2	0.06	0.03	$\cdots$	0.0
$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
a_n	0.012	0.11	$\cdots$	0.03

Table 2.3: Joint probability table for $P(A, B)$

variables. For example, the probability for tails in consecutive coin-tosses (of a regular symmetric coin) are independent in that the result of the first toss does not impact the second. On the other hand, the probability for getting three consecutive tails are not independent of the result of the first toss, since if it is impossible to get three tails in a row if the first toss ends up a head. It might also be the case that information about one variable renders other variables independent. Formally two variables A and B are *conditionally independent* given a third variable C if $P(a_i|c_k) = P(a_i|b_j, c_k)$ for all values of i, j, k. This is a misuse of the notation in that the probability tables are not the same size, what it means is that if the state of C is known, the state of B does not influence A.

2.3 Bayesian Network

The property of independence among the variables within the given domain is exploited making the Bayesian network a compact representation, and is thus one of the key features of the language.

A Bayesian network is a graphical structure that represent the independencies among variables in a domain. The structure of the Bayesian network is represented as a *directed*, *acyclic* graph in which the nodes represent *variables* each with a number of *states*. The *edges* of a Bayesian network represents probabilistic relations among variables. The state of a variable is affected by the states of all its parents. Each variable in the network has an associated *probability table*, listing the probability that the node is in each of its states.

Formally a Bayesian network could be defined as follows:

Definition 1. *A Bayesian network* $BN = (G, \Theta)$ *for variables* $V = V_1, \ldots, V_n$ *consists of the following:*

- *A directed acyclic graph* $G = (V, E)$*, where node* $N_i \in V$ *represents a variable* V_i*.*

- *A set of conditional probabilities* Θ*, such that for each* V_i*,* Θ *contains* $P(V_i|pa(V_i))$ *specifying the conditional distribution for* V_i *given its parents.*

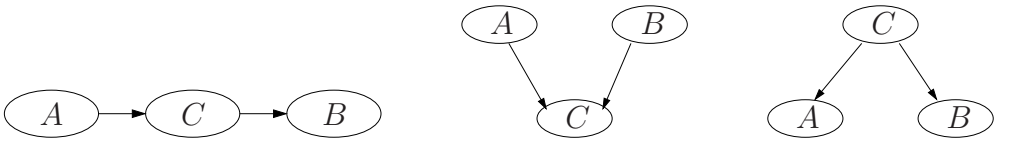
In this report we deal only with *discrete* variables, that is, variables with a finite number of *mutually exclusive* states (that is the variable is only in a single state at a time).

2.4 d-separation

An interesting property of Bayesian networks is *d-separation*. For $\mathbf{A}, \mathbf{B}, \mathbf{C} \subseteq \mathbf{V}$ where $\mathbf{A} \cap \mathbf{B} = \emptyset$, $\mathbf{A} \cap \mathbf{B} = \emptyset$ and $\mathbf{B} \cap \mathbf{C} = \emptyset$, that is 3 disjoint subsets of variables, $\mathbf{C}$ *d-separates* $\mathbf{A}$ from $\mathbf{B}$ if every path from a node $A \in \mathbf{A}$ to a node $B \in \mathbf{B}$ qualifies as one of the following:

1. a serial connection $A \rightarrow C \rightarrow B$ where C is instantiated.
2. a diverging connection $A \leftarrow C \rightarrow B$ where C is instantiated.
3. a converging connection $A \rightarrow C \leftarrow B$ where neither C nor any of its descendants are instantiated.

Rule 1 is illustrated in Figure 2.1(a), rule 2 in Figure 2.1(c) and rule 3 in 2.1(b)



(a) Serial connection (b) Converging connection (c) Diverging connection

Figure 2.1: d-separation.

When variables are d-separated, they are also conditionally independent. Independence among variables may be encoded in the associated probability tables, such that $P(A|B) = P(A)$ despite A and B not being d-separated. So d-separation implies independence, but independence does not (necessarily) imply d-separation.

2.5 Chain rule

Since all nodes in the network represents a variable with associated probability distribution over itself and its parents, summing these probability distributions for all nodes yields the joint probability distributions for the domain. Let $U = \{V_1, \ldots, V_n\}$ be a universe of variables in a given domain, and let $P(U)$ be the joint probability distribution over the variables of this domain.

From the Bayesian network the joint probability distribution $P(U)$ for a domain is given using the chain rule stated in Equation (2.1).

$$P(U) = \prod_{X \in U} P(X|pa(X)) \tag{2.1}$$

The chain rule states that a joint probability distribution can be obtained by multiplying the conditional probabilities for all variables in the network. In this way a Bayesian network is a compact representation of a joint probability distribution.

All queries regarding probabilities in the network can be answered using $P(U)$, however the size of $P(U)$ is exponential in the number of states of the variables, making it hard to represent the distribution of even a small network within the memory limitations of a computer.

Chapter 3

Inference

The definitions in Chapter 2 only describes the elements of a Bayesian network as a description of probabilistic independencies, but the model can also be used for *inference*.

Inference means propagating knowledge (called *evidence* about some variables throughout the model, with the purpose of seeing the effect on other variables.

There are different approaches to performing inference, even though the general problem has been shown to be NP-hard [Coo90]. Yet different methods may still be preferred over others. A widely used method for doing inference is to use a secondary structure, called a junction tree. Instead of working with the individual variables, the junction tree work with sets of variables. And instead of working with the probability functions from the Bayesian network, the general notion of *potentials* are used.

3.1 Junction tree inference

A junction tree is an effective structure used together with a *message-parsing* method for performing inference over a Bayesian network. Using the function tree for inference is discussed in Section 3.7, but before we get to that we will introduce the procedure used when compiling a Bayesian network into a junction tree.

In order to use the junction tree method, the variable sets (called *cliques*) and *potentials* must be constructed by compiling the Bayesian network to a junction tree.

Definition 2. *A potential ϕ is a real-valued function over a domain of a set of finite variables $\mathcal{V}$: $\phi : space(\mathcal{V}) \rightarrow \mathcal{R}$. A potential with a domain over variables A and B are denoted as $\phi_{(A,B)}$.*

Definition 3. *A clique is a complete set of nodes that is not a subset of another complete set.*

A complete set of nodes, is a set of nodes that are fully connected. These cliques represent the domains of the potentials we may need to calculate when doing belief updating. The process of compiling a Bayesian network to a junction tree structure involves a number of steps:

- Construct domain graph
- Triangulate domain graph
- Identify cliques
- Construct join tree
- Construct junction tree

In the following section this process will be explored. Throughout this chapter we will use the network shown in Figure 3.1, with probability functions $P(A)$, $P(B|A)$, $P(C|A)$, $P(D|B,C)$ and $P(E|C)$ as an example.

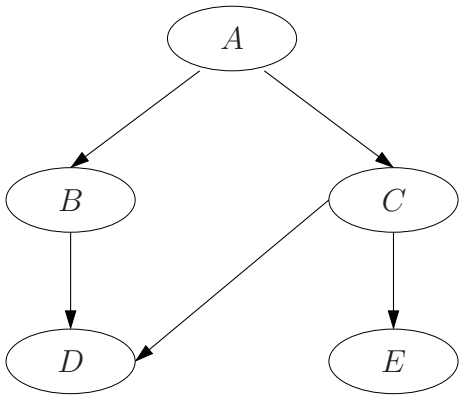


Figure 3.1: Example network

3.2 Domain graph

The first step in producing a junction tree from a Bayesian network is to construct a *domain graph* from the Bayesian network. The domain graph is a graphical representation of the domains of the probability *potentials* in the Bayesian network Φ. For our example graph we have the potentials $\phi_{(A)}$, $\phi_{(B,A)}$, $\phi_{(C,A)}$, $\phi_{(D,B,C)}$ and $\phi_{(E,C)}$.

A domain graph is an undirected graph where the nodes are variables from the potentials in Φ, and the edges signifies that the variables are member of the same domain.

Constructing a domain graph from a Bayesian network consists of adding edges between any pair of variables with a common child, and then remove the orientation of all edges.

Looking at our example graph, the potential $\phi_{(D,B,C)}$ which in this case is the probability function $P(D|B,C)$, has the variables D, B and C in its domain. An edge is therefore added between the two nodes B and C in the domain graph. Since the new edge connects ("marries") two nodes which share a child, this process is called *moralization*. We do this for all nodes sharing a child. Finally the orientation is removed from all edges, resulting in the graph shown in Figure 3.2.

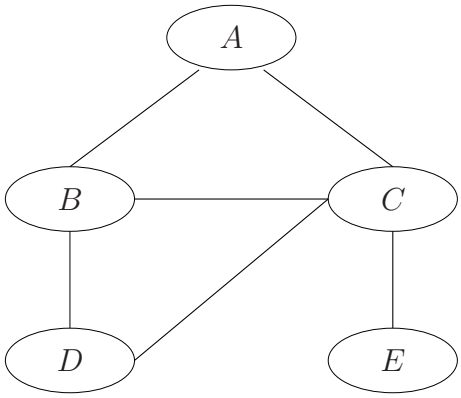


Figure 3.2: Moralized graph

3.3 Triangulation

Triangulation is an important step in the compilation process in that the structure of the resulting graph have great influence on the computational complexity of the junction tree, and that there are numerous ways to perform the triangulation.

To illustrate the first point we must draw parallels between the domain graph that we have established through moralization and the calculations needed to perform inference.

Notation. *The product of a set of potentials* $\Phi = \{\phi_1, ..., \phi_n\}$ *is denoted* $\prod \Phi$.

Assume $\mathcal{U} = \{A_1, \ldots, A_n\}$ and that we want to calculate $P(A_1)$. We would then have to marginalize out $A_2, \ldots, A_n$ from U. When a variable V is Marginalized out of Φ, all potentials $\Phi' = \{\phi | V \in \text{domain of } \phi\}$ will be removed from Φ and replaced by a single potential $\phi^* = \sum_V \prod_{\phi \in \Phi'} \phi$. How complex this operation is depends on the size of the involved domains. For a set $\mathcal{X}$ of variables $\sum_{\mathcal{X}} \prod \Phi$ is calculated by repeatedly eliminating $V \in \mathcal{X}$ from Φ. This means that the order in which variables are eliminated has a big impact on the size of the resulting potentials in Φ. For example: As previously mentioned, our example network have the set of potentials $\Phi = \{\phi_{(A)}, \phi_{(B,A)}, \phi_{(C,A)}, \phi_{(D,B,C)}, \phi_{(E,C)}\}$. If we were to marginalize out the variables A and B we could choose to first marginalize out A and then B or the other way around. We examine both options:

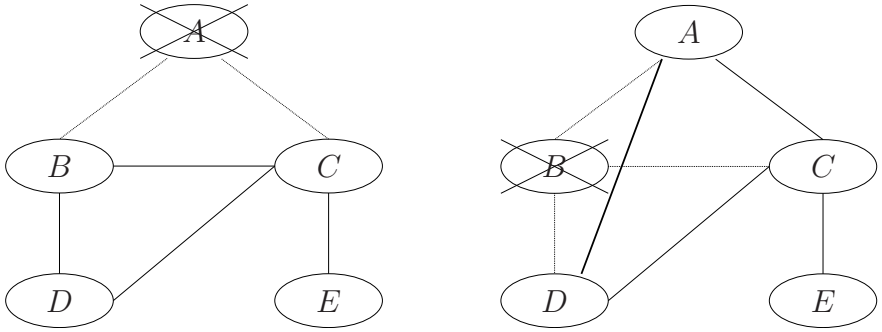
- If we start with A and do $\sum_A \prod \Phi$ we get $\Phi' = \{\phi_{(D,B,C)}, \phi_{(E,C)}, \phi_{(B,C)}\}$ where the last potential $\phi_{(B,C)}$, while new, is over (part of) the same domain as $\phi_{(D,B,C)}$. The next step is to do $\sum_B \prod \Phi'$ which yields $\Phi'' = \{\phi_{(E,C)}, \phi_{(D,C)}, \phi_{(C)}\}$ and again, the domains for the potentials are subsets of previously used domains.

- If we start with with B and do $\sum_B \prod \Phi$ we get $\Phi' = \{\phi_{(A)}, \phi_{(C,A)}, \phi_{(E,C)}, \phi_{(D,C,A)}\}$. This time the potential $\phi_{(D,C,A)}$ describes a completely new domain that has not been previously calculated. Then we can marginalize out A as $\sum_A \prod \Phi' = \Phi'' = \{\phi_{(E,C)}, \phi_{(D,C)}, \phi_{(C)}\}$. In this set of potentials no new domains are needed.

The order of marginalization is important for efficient calculation, but how do we know which order is a good one? For this we can use the domain graph. When talking about triangulation of graphs we have the notion of *elimination* of nodes. Eliminating a nodes means to mark the node as eliminated and then connect all non-eliminated neighbours of the node. The edges that are added are called *fill-in* edges. So that after the elimination the non-eliminated nodes constitute a complete subgraph.

Eliminating a variable from the graph corresponds to marginalizing out the variable from the set of potentials that the graph represents. The addition of *fill-in* edges signifies that calculations will include working with a potential over a domain that was not present initially. This is illustrated in Figure 3.3(a) which shows the domain graph after A has been eliminated, and Figure 3.3(b) which shows the domain graph after the elimination of B. In Figure 3.3(b) a fill-in has been added between A and D such that the nodes A,D and C constitutes complete subgraph. These nodes are also the ones who

constituted the new domain to be calculated if we chose to marginalize B out first.



(a) A eliminated

(b) B eliminated and a fill-in edge added

Figure 3.3: Examples of elimination of nodes and the produced fill-in edges.

An *elimination sequence* is a list of variables that denotes in which order the variables should be eliminated from the graph. An elimination sequence that results in no fill-in edges (called a *perfect elimination sequence*)have smaller complexity than an elimination sequence that introduce fill-ins, as new larger potentials will have to be created in the process. Thus it is advantageous to find an elimination sequence that introduces no fill-ins. A graph with a perfect elimination sequence is called a *triangulated* graph. A triangulated graph is also characterised by having no cycles of length ≥ 4 that is not cut by a *chord*. A chord is an edge between two non-consecutive nodes in the cycle. If a graph is not triangulated, then extra edges will have to be added until the graph satisfies the requirements. The process of converting the domain graph to a triangulated graph is called *triangulation*. Triangulation may be done by choosing an elimination sequence (or using an already known sequence) and then eliminate the variables from the domain graph in that order. A *triangulation* T for a graph G is a set of fill-in edges and the *triangulated graph* is then the domain graph, with the addition of the triangulation (that is, the set of fill-in edges produced during the elimination). Due to the high number of possible sequences, a heuristic approach is often used in finding a triangulation. We discuss triangulation heuristics in detail in Section 3.8.

3.4 Cliques

From the triangulated graph, a set of cliques can be identified. In our example, the graph is already triangulated and so no fill-in edges are added. The

cliques are $(ABC), (BCD), (CE)$ as shown in Figure 3.4.

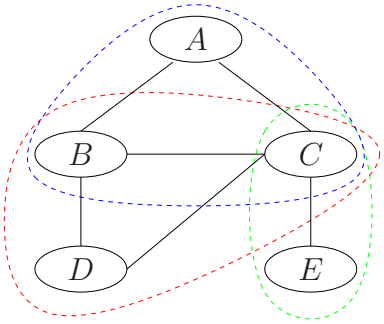


Figure 3.4: Cliques in the triangulated graph.

3.5 Join Tree

The cliques are arranged in a *join graph*. Two cliques $\mathbf{C_1}, \mathbf{C_2}$ are connected if $\mathbf{C_1} \cap \mathbf{C_2} \neq \emptyset$, that is if they share one or more variables. The join graph can be transformed into a *join tree* by constructing a *Maximum spanning tree* where the weight of an edge $e = (C_1, C_2)$ connecting $\mathbf{C_1}$ and $\mathbf{C_2}$, is defined as $w(e) = |\mathbf{C_1} \cap \mathbf{C_2}|$, namely the number of variables the connected cliques share.

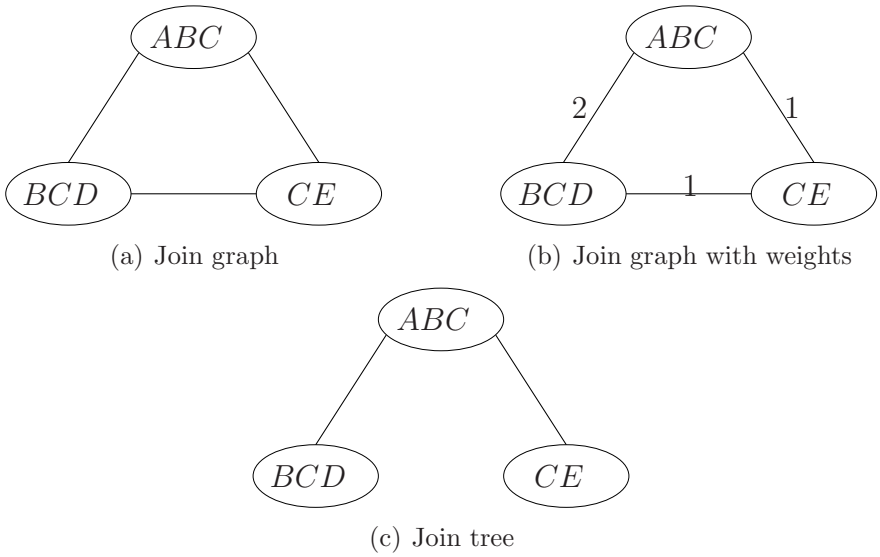
A maximum weight spanning tree is constructed by continuously selecting one of the edges with the highest weight until all nodes are connected to the tree. An example of this process is shown in Figure 3.5

3.6 Junction tree structure

A junction tree is a join tree, where edges are annotated with separators. Let T be a join tree for the domain graph G with potentials Φ. A junction tree for G consists of T with these additions:

1. Each potential $\phi \in \Phi$ is assigned to a clique containing $dom(\phi)$, where $dom(\phi)$ is the domain of ϕ
2. Each edge in T has a *separator* attached.
3. Each separator contains two mailboxes, one for each direction.

The separators acts as a mailboxes between the cliques, communicating potentials between them. The variables associated represents the domain of the potential communicated.



(a) Join graph (b) Join graph with weights

(c) Join tree

Figure 3.5: Transforming the join graph to a join tree

3.7 Exact inference using junction trees

Propagation in junction trees consists of two operations: *Collect evidence* and *Distribute evidence*, which work by projecting potentials associated with the cliques and passing them about as messages in the junction tree.

Definition 4. *For a set Φ of potentials whose domains are a subset of $\mathcal{V}$, the projection of $\Phi^{\downarrow\mathcal{W}}$ down to $\mathcal{W} \subseteq \mathcal{V}$ is defined as:*

$$\Phi^{\downarrow\mathcal{W}} = \{\phi \in \Phi | dom(\phi) \cap \mathcal{V}\backslash\mathcal{W} = \emptyset\} \cup \{\phi^*\}, \text{ where}$$

$$\phi^* = \sum_{\mathcal{V}\backslash\mathcal{W}} \prod \{\phi \in \Phi | dom(\phi) \cap \mathcal{V}\backslash\mathcal{W} \neq \emptyset\}$$

That is $\Phi^{\downarrow\mathcal{W}}$ is a set of potentials resulting from eliminating the variables in $\mathcal{V}\backslash\mathcal{W}$.

Definition 5. *Message passing: A clique V, with a set of potentials Φ_V and neighbouring separator $S_1, \ldots, S_k$, where each S_i have received a message ψ_i for V, can send the message $(\Phi_V \cup \psi_1 \cup \ldots \cup \psi_k)^{\downarrow S}$ to another neighbouring separator S. The direction V to S is then said to have been triggered. For a separator S, the messages ψ_S and ψ^S represent the two different messages (one for each direction) that can pass over the separator.*

Collection is done by selecting a node as temporary root, and passing messages towards that node. Distribution is done afterwards by passing messages the other way. If only a single marginal probability is needed, it is sufficient to collect evidence to a clique containing that node and no distribution will be needed.

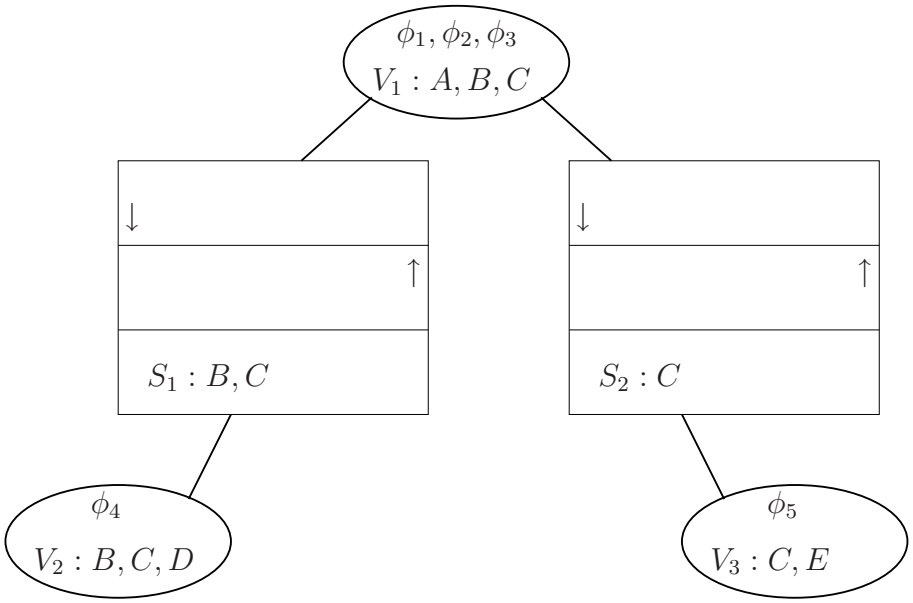


Figure 3.6: Junction tree

Inference Example

Lets consider our example network, and the associated junction tree (as shown in Figure 3.6). If we want to calculate $P(D)$ we would need to collect evidence in a clique containing D (e.g. V_2). So V_2 is made the temporary root and we need to send messages from the leaves towards it. In our network the only leaf is V_3, with the neighbouring separator S_2. We trigger the direction V_3 to S_2 by passing the message $\psi_2 = (\phi_5)^{\downarrow S_2} = (\phi_5)^{\downarrow C}$. Notice that the message is a reduction of the set of potentials in V_3 that "fits" in S_2. Then we create a message to pass from V_1 to S_1, this message is $\psi^1 = (\phi_1\phi_2\phi_3\psi_2)^{\downarrow S_1}$. Now the message has reached V_2 and the collection stages is completed (the state of the junction tree at this stage is shown in Figure 3.7). The message ψ^1 represents a perfect elimination order ending with the variables in V_2 and so the product $\phi_4\psi^1$ is the projection of the entire product down to B, C, D. We can now calculate $P(D)$ by eliminating $V_2 \backslash D$ from $\phi_4\psi^1$, namely $P(D) = (\phi_4\psi^1)^{\downarrow D}$.

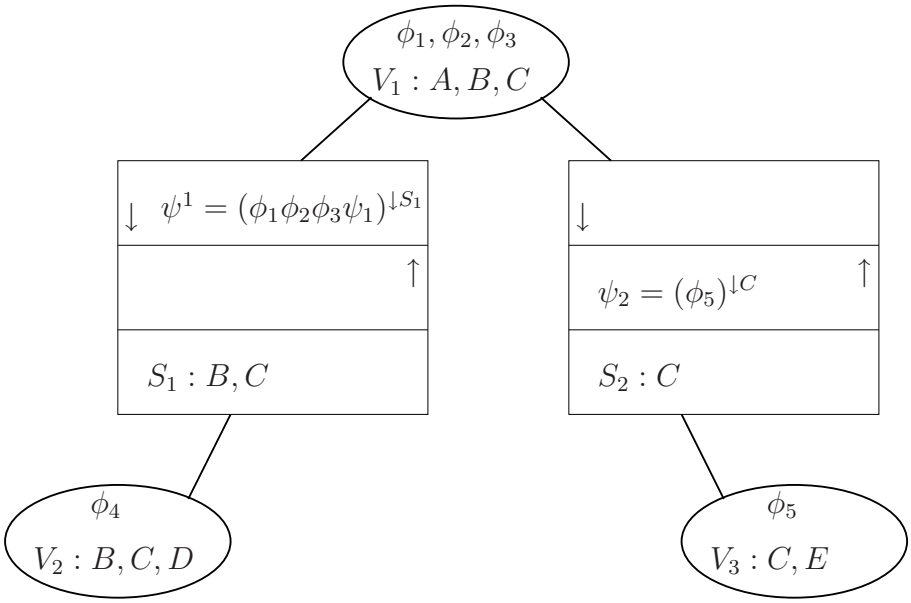


Figure 3.7: Junction tree after evidence collection

To be able to calculate the rest of the marginals in the network we need to distribute the evidence, so messages must be send towards the leaves. From V_2 to S_1 we send the message $\psi_1 = (\phi_4\psi^1)^{\downarrow S_1}$, and from V_1 to S_2 we send the message $\psi^2 = (\phi_1\phi_2\phi_3\psi_1)^{\downarrow S_2}$. The tree has now been "filled", and we have performed a *complete propagation*. The junction tree with all messages is shown in Figure 3.8.

Inferring probabilities with evidence in the network is a matter of placing the evidence, represented as a 0-1 potential, in the appropriate cliques and doing a full propagation. E.g. with evidence e that A is in state a_i, $P(X, e)$ is calculated by placing e in V_1 and then doing a full propagation.

3.8 Triangulation heuristics

As finding an optimal triangulation with regards to the size of potentials is NP-hard, heuristics are often employed in place of an exhaustive search. Many different types of heuristics can be imagined, a simple one could be to simply use a lexical ordering over the nodes as the elimination order. A heuristic search that often works quite well is to simple use a greedy algorithm with a one step lookahead. A generic algorithm is shown below. This algorithm greedily searches for a minimal value of a criterion $c(X)$ where X is a node in the graph. The return value is a set of fill-in edges and a

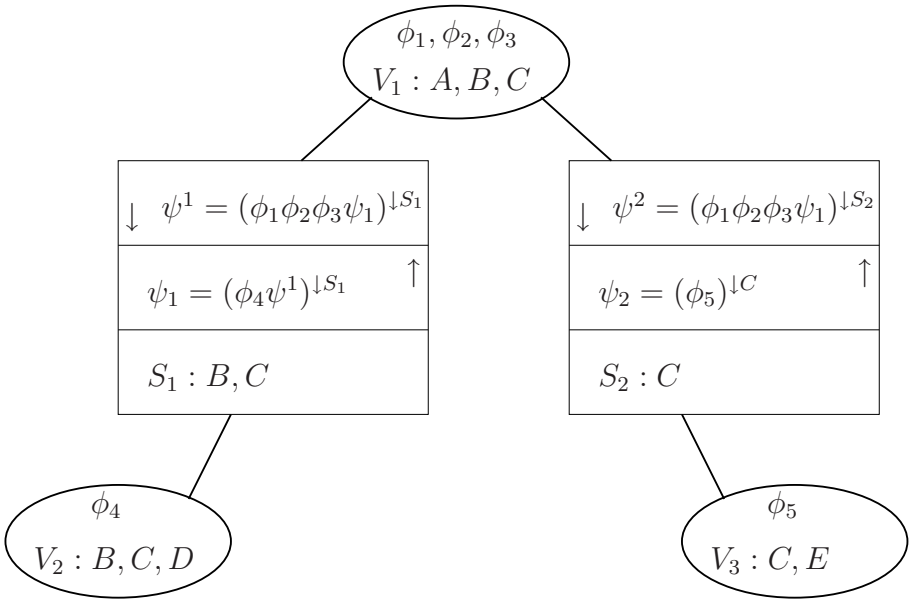


Figure 3.8: Junction tree after complete propagation

corresponding elimination order.

TRIANGULATION(Graph $G = (V, E)$,Optimisation criterion$c(X)$)

```
1  F ← {}
2  L ← {}
3  V' ← V
4  Repeat
5        do Select a node X ∈ V' such that c(X) is minimal
6           F ← F ∪ {{Y, Z}|Y, Z are neighbours of X, Y ≠ Z, {Y, Z} ∉ F}
7           Remove X from V'
8           Add X to the end of L
9  Until all nodes N ∈ V' has been eliminated
10 return F, L
```

In this algorithm the criterion $c(X)$ can be a swapped for different function if it is desirable. Possible functions could be *Fill-ins* (the number of added fill-in edges) and *clique-size* (the minimal combined size of the cliques), both of which are described in [Kjæ93] and repeated below. This makes the clique size heuristic a good choice for our project, since smaller cliques means smaller potentials and thus shorter inference time.

This algorithm simply returns the number of fill-in edges that would have to be added to the graph if the node in question where eliminated.

FILL-INS(node X)

1 $\mathcal{N} \leftarrow$ all neighbour of X
2 $E_{exist} \leftarrow \{(Y, Z)|Y, Z \in \mathcal{N}, Y \neq Z\}$
3 **return** $\binom{|\mathcal{N}|}{2} - |E_{exist}|$

The clique-size algorithm returns the size of the clique the node would be in after the triangulation.

CLIQUE-SIZE(node X)

1 $\mathcal{N} \leftarrow X \cup \{Y|Y$ is a neighbour of $X\}$
2 **return** $\prod_{Z \in \mathcal{N}} |Z|$

Some heuristics are more expensive to calculate than others, but this might be an acceptable tradeoff if the resulting junction tree is much faster for inference.

Since our example graph is already triangulated, we will use the graph in figure 3.9 where the variables A, B and C all have two states and the variables D and E has 5 states.

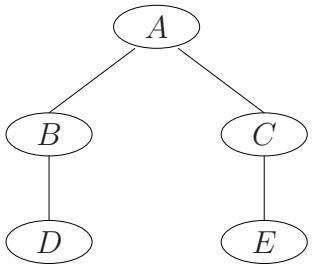


Figure 3.9: Example graph

Triangulation of this graph using the fill-ins heuristics would work like this: The first node to be added to the elimination order is either D or E, both of which produces no fill-ins. If we choose D, the next step adds either E or D which (in the current state of the graph) both introduces no fill-ins. The process continues in the same manner and all nodes can be added without introducing fill-ins. If we used the Clique-size heuristic, the process would be as follows:

First iteration The scores for the nodes are:

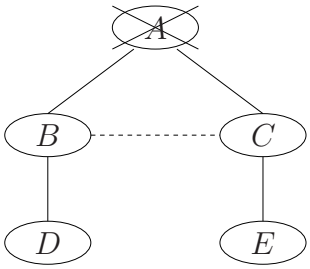
Node		*Score*
A	$2 \cdot 2 \cdot 2$	8
B	$2 \cdot 2 \cdot 5$	20
C	$2 \cdot 2 \cdot 5$	20
D	$2 \cdot 5$	10
E	$2 \cdot 5$	10

So the node A is added to the elimination order. This introduces the fill-in edge (B, C). The graph after this iteration is shown in Figure 3.10(a)

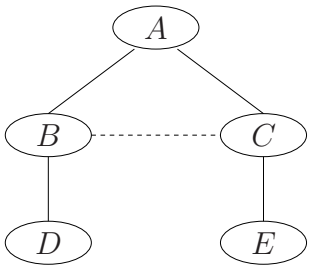
Second iteration the scores are:

Node		*Score*
B	$2 \cdot 2 \cdot 5$	20
C	$2 \cdot 2 \cdot 5$	20
D	$2 \cdot 5$	10
E	$2 \cdot 5$	10

At this point either D or E is added to the elimination order, followed by the rest of the nodes.From this point no more fill-ins are added. The final triangulated graph is shown in Figure 3.10(b).



(a) Example graph triangulated using the clique-size heuristic, after first iteration



(b) Example graph triangulated using the clique-size heuristic

Figure 3.10: Triangulation of the example graph using clique-size heuristic

3.8.1 Minimal triangulation

A minimal triangulation is one in which no fill-in edges are *redundant*, that is for edge e in triangulated graph $G^T = V, E$, $G^{T'} = (V, E \backslash \{e\})$ is also triangulated. Redundant fill-in edges can be product of the elimination sequence selected. An example of a redundant fill-in edge is the edge (B, C) in Figure 3.10(b)

While there exists triangulation algorithms for finding minimal triangulations (such as LEX M[RTL76]), another approach is to use the *recursive thinning* algorithm (described in [Kjæ93]) which can remove redundant fill-in edges from any triangulation. The algorithm is as follows:

THIN(graph $G = (V, E)$, triangulation T for G)

```
1  R' ← T
2  Repeat
3        do T' ← {(X,Y)|neighbours(X) ∩ neighbours(Y) is complete in G}
4           T ← T\T'
5           G ← (V, E ∪ T)
6           R' ← {e1|∃e2 ∈ T' such that e1 ∩ e2 ≠ ∅}
7  Until T' = ∅
8  return T
```

The algorithm works by finding the edges that connects two nodes which has fully connected common neighbours. In the network depicted in Figure 3.10(b) the edge (B, C) is such an edge since $neighbours(B) \cap neighbours(C)$ is the single node A. In the first iteration loop all edges are considered. In subsequent loops only edges that shares a node with an edge that was removed in a the previous loop are considered. The running time of the algorithm is $O(|T|^2)$.

Part II

Learning Inference Models

Chapter 4

Structural learning

Structural learning refers to the process of building a network structure from a dataset. A dataset is a database of cases, where each case represent a particular configuration of the variables. If the dataset contains no missing values the dataset is complete, and incomplete if it contains missing values. Complete datasets are easier to work with, but not an explicit requirement for performing structural learning [Fri98], [Nea03].

There are two different paradigms for performing structural learning. The first approach may be refferred to as constraint-based, where the learning process consists of building a graph by performing a set of independence tests on the dataset. (This is, for example, the structural learning method implemented in the Hugin tool [hug]). We have described the details of constraint based structural learning in Appendix B.

The other approach may be refferred to as score based learning. In score based learning, the learning problem becomes a search problem within the space of possible network structures. Then a search is performed using some scoring criterion for the search to maximize. A typical scoring criterion is the Bayesian information criterion (BIC) [Sch78], [LB94], [FG] that assigns a score to a network structure which combines the likelihood that this structure "generated the data" and how dense the structure is.

The differences between these two methods are apparent: The constraint based approach produces a network structure based on categorical information about conditional independencies derived by a number of statistical tests on the dataset. The score based approach ranks each network structure based on a computation of the conditional probability of the structure given the dataset. Consequently given small datasets, constraint based may make incorrect categorical decisions about conditional independencies. Score based can handle missing data items where constraint based typically throws out a case containing missing values. A longer discussion of differences can be

found in [Nea03].

For our purposes the score based approach has one main advantage over the constraint-based approach, namely that the scoring criterion offers an opportunity for better control over the search by specifying which properties the resulting network structure should maximize, and by what weights.

Althoug a similar effect could be approached in constraint based learning by changing the threshold value for when edges are considered significant enough to be added to the structure, it does not take into accout the factor of effective complexity. (As noted in section 3.7, a major factor on the inference complexity is the existence of undirected cycles, so one could perhaps preprocess the graph skeleton in terms of removing cycles. This could be done by considering each edge in a cycle and remove the one that represents the lowest mutual information).

As the purpose of this project is to find a learning method that makes it possible to weight the relationship between effective complexity and precision, the score based approach seems a better fit than constraint-based. This chapter explores the score based learning approach and concludes by formulating a scoring criterion that offers this kind of control.

4.1 Search Space

As mentioned above, the problem of learning a Bayesian network structure from a dataset can be considered as a search problem. The search space is the set of possible network structures. The purpose of the search is to find the member that maximizes some scoring criterion (BIC scoring and our proposed scoring criteron is explored later in Sections 4.2 and 4.3). The scoring criterion most commonly incorporates finding a structure that serves as a good description of the dataset while satisfying some other constraints at the same time.

For a given structural learning problem, it is in theory possible to enumerate all members of the search space, and then evaluate each member using the scoring criterion. But in practice this approach is impossibe as the shear size of the search space makes such brute force evaluation impossible. This is evident at it has been shown that the number of member network structures $f(n)$ grows super exponentially in the number of nodes n:

$$f(n) = \sum_{i=1}^{n} (-1)^{i+1} \frac{n!}{(n-i)!n!} 2^{i(n-i)} f(n-1).$$

And it is the reason heuristics are used for directing the search. Structural search proceeds by using some candidate as a starting point, most commonly

representing the empty network, and then performing the search step by step. In the rest of this report we shall use the word 'candidate' to denote a member of the search space, which in turn represents a unique network structure - or, depending on the search strategy, a set of network structures.

An example of the search space over network structures with four nodes can be seen in Figure 4.1.

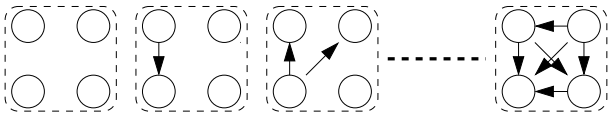


Figure 4.1: The space of network structures with four nodes (543 members.)

At each search step, which corresponds to a given point in the search space, the next set of possible candidates are evaluated. If one candidate is better than the best found so far, the search continues using that candidate. This procedure repeats itself until the scoring criterion yields that none of the possible candidates are better than the current candidate, thus a maximizing candidate has been found. This search can be depicted by a search tree as in Figure 4.2. In the search tree each vertex corresponds to a candidate, and the candidate pointed to by a dotted line represents the starting point of the structural search. The outgoing arrows from a candidate represents the set of candidates reachable from from that point.

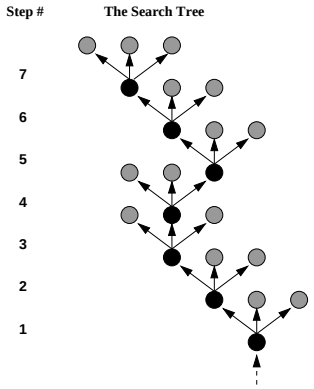


Figure 4.2: Representation of a search tree. Each vertex corresponds to a given structure candidate.

The search starts from the candidate pointed to by the dotted arrow, (this is step 1). Then the set of candidates reachable from that point is evaluated using the given scoring criterion. The candidate that maximizes the score

is chosen (indicated by a black vertice) and the remaining are discarded (indicated by shaded vertices). The next step of the search then continues from the maximizing candidate. This is repeated until no further candidates that maximize the scoring criterion are found (in the example the search ends at step 7, where the entire set of reachable candidates are discarded as they are all rated lower than the current candidate). This way we only explore a portion of all possible candidates, as only a particular branch is investigated using the scoring criterion as a heuristics.

How the next set of candidates are generated from a given point in the search space depends on the search strategy used, for example the commonly used simple greedy search. Greedy search explores the search space by incrementally modifying the current candidate structure using the following transformations:

- Add edge
- Remove edge
- Reverse edge

That is, the set of possible candidates reachable from a given candidate in one step, can be enumerated by repeatedly applying a single edge transformation to the candidate. *For the purpose of this discussion we denote the greedy search as the usual method associated with score-based structural learning.*

The type and size of the search space may have a big impact on the computational effort required. As the search space is defined by the type of search strategy used, in addition to greedy search additional strategies that offer different levels of performance exists. Two such strategies are *Equivalence class* search[JN07]pages 248-150, and *Ordering based* search[TK05].

4.2 BIC-scoring

A commonly used score function that takes into account both likelihood and complexity is the *Bayesian information criterion* (BIC). This section is from [KP07]. The BIC score consists of two terms, one regarding likelihood and the other regarding the complexity of the network. It is defined as:

$$\mathrm{BIC}(S|D) = \log_2 P(D|\hat{\Theta}_s, S) - \frac{size(S)}{2}\log_2(N) \tag{4.1}$$

where $\hat{\Theta}_s$ is an estimate of the maximum likelihood for the structure S, and N is the number of cases. $size(S)$ is the number of free parameters in the

model $\sum_{V in \mathcal{V}}(sp(V)-1)\cdot\sum_{W\in pa(V)} sp(W)$. When learning from complete data, we can use frequency counts for the maximum likelihood parameter estimates. A compelling feature of the BIC-score is that it is a decomposable score function, this means that the score can be expressed as the sum of local scores that is score for each node family in the structure. The BIC-score for a network with n variables can be calculated as

$$\mathrm{BIC}(S|D) = \sum_{i=1}^{n}\left[\sum_{j=1}^{q_i}\sum_{k=1}^{r_i} N_{ijk}\log_2\left(\frac{N_{ijk}}{N_{ij}}\right) - \frac{1}{2}q_i(r_i-1)\log N\right] \tag{4.2}$$

where r_i denotes the number of states for variable X_i, q_i denotes the number of configurations over the parents of X_i (or 1 if X_i has no parents), and N_{ijk} is the number of cases in the dataset that express configurations where variable X_i is in its i'th state and the parents of X is in the jth configuration. N_{ij} is the number of cases where the parents of X_i is in the jth configuration. The part of equation (4.2) within braces is the decomposable part and can be calculated individually for each node family.

A simple extension to the BIC-score model is to add an explicit parameter that balances the likelihood term and the network complexity penalty term. This λ-parameter is used as in equation (4.3) below:

$$\mathrm{BIC}(S|D,\lambda) = \sum_{i=1}^{n}\left[\lambda\left(\sum_{j=1}^{q_i}\sum_{k=1}^{r_i} N_{ijk}\log_2\left(\frac{N_{ijk}}{N_{ij}}\right)\right) - (1-\lambda)\left(-\frac{1}{2}q_i(r_i-1)\log N\right)\right] \tag{4.3}$$

The λ-parameter is set to a value between 1 and 0 and represents how much we value accuracy versus complexity. A higher λ-parameter adds weight to the part of the score that represents the maximum likelihood, whereas a lower value increases the weight of the part that represents network complexity. For a λ value of 0.5, $2\cdot\mathrm{BIC}(S|D,\lambda) = \mathrm{BIC}(S|D)$.

Score based learning offers the most direct way to balance network complexity and likelihood. We speculate that a highly complex network structure would result in a complex junction tree and thus slower inference. So it seems likely that at least some control over the complexity of the junction tree can be exercised by controlling the complexity of the network.

4.3 Junction tree scoring

A more direct approach for controlling the balance between accuracy and effective complexity would be to use a score function that compiles and measures a junction tree for the network in question. This is exactly the approach

we used in [KP07], which we seek to optimize with regards to performance (interesting preliminary results can be found in Appendix A). In the rest of this section we shall introduces the junction tree scoring criterion (JTC) as well as the disadvantages imposed. Then we shall, in pursuit of our project goal, explore a way of dealing with these issues.

If we consider a more abstract version of BIC from the last section, we get an expression on the following form:

$$\mathrm{SC} = LL - d \tag{4.4}$$

and similarly for the λ-version of BIC, we get an expression on the form:

$$\mathrm{SC} = \lambda \cdot LL - (1 - \lambda) \cdot d \tag{4.5}$$

Where LL is the log-likelihood of the network given the dataset, and d is the dimension of the structure. For the BIC score, the term that penalized according to network size was d, and in this way BIC serves as the inspiration for the JTC scoring criterion.

As we saw in Chapter 3 the computational effort required for performing junction tree inference is largely determined by the size of the cliques that make up the junction tree, as the size of cliques determines the size of the probability potentials that are operated upon. The larger the probability potentials, the more computational effort is required for performing inference. Thus the JTC criterion should score a candidate based on the state space of the resulting junction tree. As there does not exist a one-to-one correspondance between a Bayesian network and a junction tree, the score assigned by JTC will of course be in relation to the specific triangulation heuristic used for producing a junction tree.

Based on the expression in equation 4.5, we can formulate the JTC scoring criterion:

$$\mathrm{JTC}(S|D,\lambda) = \lambda \cdot \log_2 P(D|\hat{\Theta}_s, S) - (1-\lambda) \cdot \frac{statespace(\mathcal{T})}{2} \log_2(N) \tag{4.6}$$

JTC is almost identical to BIC. The first part of the term incorporates the log likelihood, which is the same for BIC - but the second part of the term differs as instead of using the size of the graph structure S, it incorporates the state space of a junction tree $\mathcal{T}$ for the graph structure S.

The structural search consist largely of iterative scoring candidates that share certain structural similarities. Similar to the decomposability of BIC, it would improve performance if JTC score could be expressed as a sum of local computations.

As for the BIC score, the log likelihood part of the JTC term are decomposable over the node families. This property has the effect that as a set of modifications to the graph structure changes the families of a subset of the nodes, likelihood must be recomputed for these nodes. The term need not be recomputed for nodes unaffected by the changes, as cached results from previous computations of these can be re-used. This gives a JTC scoring criterion with at least the log likelihood part of the expression being decomposable:

$$\mathrm{JTC}(S|D,\lambda) = \sum_{i=1}^{n} \lambda \cdot \left[\sum_{j=1}^{q_i} \sum_{k=1}^{r_i} N_{ijk} \log_2 \left(\frac{N_{ijk}}{N_{ij}} \right) \right] - (1-\lambda) \cdot \frac{statespace(\mathcal{T})}{2} \log_2(N) \tag{4.7}$$

The parts within braces are decomposable and can be calculated individually for each node family. The JTC in equation 4.7 is similar to the one we used in [KP07].

Chapter 5

Incremental Compilation

As described in the previous chapter, the problem with learning using the JTC-scoring method was the prohibitively long learning time. Two observations can be made regarding the performance of the method:

- The score function is not decomposable, so we are required to score the entire junction tree. As previously described one of the hallmarks of a good score functions is that is can be divided as a sum over divisions in the net (such as families of nodes in the case of BIB).

- The most costly, in terms of computer time, step in the JTC-scoring method is the re-triangulation of the entire tree. Since triangulation is such an expensive step on the construction of a junction tree, it is the most likely step to start with in search for optimisations.

Whereas little can be done about the decomposability of the score function, the notion of dividing the graph into smaller parts and only dealing with the part(s) that have changed can be employed. What we need is a division of the graph (or the junction tree) in such a way that a change in the set of edges can be associated with one or more parts which needs to be updated and the rest of the graph can be ignored with regards the change (in that these parts remain the same). The next section describes a structure that divides the graph into parts that can be triangulated individually and connected together to produce a junction tree for the graph. In [FGO03] a method for *incremental compilation* of Bayesian networks is presented. The method divides the graph into maximal prime subgraphs[OM02], which has the property were are looking for.

5.1 Maximal prime subgraph decomposition

This section introduces a method for decomposing Bayesian networks into their *maximal prime subgraphs* as described in [OM02]. At first we need a couple of definitions:

Definition 6. *A separator S, where $S \subset V$ of the set of variables in $G^M = (V, E)$, is complete if the subgraph g^M induced by S is complete.*

Consider the junction tree (Figure 5.1(b)) for the graph in Figure 5.1(a): The junction tree has two separators BC and CD. The separator BC is a complete separator as the induced subgraph in Figure 5.1(c) is complete, whereas the separator CD is incomplete which can be verified by looking at the induced subgraph in Figure 5.1(d) which is incomplete.

Definition 7. *A maximal prime subgraph is a sub-graph that is d-separated from its surroundings by complete separators.*

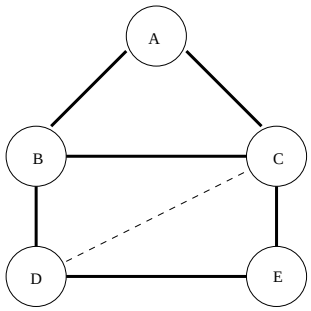
The *maximal prime subgraph decomposition* (MPD) can be used as computational structure for incremental construction of junction trees. The MPD is related to maximal complete subgraphs of the moral graph of the Bayesian network, and is also known as *decomposition by clique separators*. MPD concerns the graph theoretical part of Bayesian networks and does not consider information about the associated probability potentials. A Maximal prime subgraph decomposition can be constructed from the junction tree by recursively aggregating cliques connected by incomplete separators. The clusters in the resulting tree are the maximal prime subgraphs of the graph of the Bayesian network. It is a requirement that junction tree junction tree are based on a minimal triangulation.

A graph G is decomposed into its maximal prime subgraphs as follows: Let G^M be the moral graph of G, let $G^{T_{min}}$ be the graph corresponding to a minimal triangulation of G^M and let $\mathcal{T}_{min}$ be a junction tree correspoinding to the cliques induced by $G^{T_{min}}$. The maximal prime subgraphs of G^M are formed by aggregating adjacent cliques connected by a separator which is incomplete in G^M. This method is formalized to algorithm 1 below.

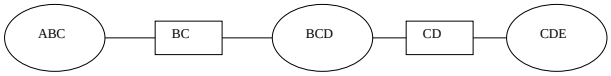
Algorithm 1.

ConstructMPDTree(join tree $\mathcal{T}$, moral graph G^M)

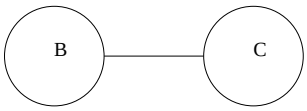
1 $\mathcal{T}^{\mathcal{MPD}} \leftarrow T$
2 Merge all adjacent cliques in $\mathcal{T}^{\mathcal{MPD}}$ with
an incomplete separator in G^M
3 **if** $\mathcal{T}^{\mathcal{MPD}}$ is a forest
4 **then** Connect trees in $\mathcal{T}^{\mathcal{MPD}}$ with empty separators
5 **return** $\mathcal{T}^{\mathcal{MPD}}$



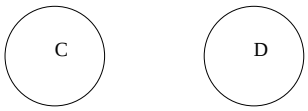
(a) Graph. Bold lines represent moral edges, dotted line represents fill-in



(b) Junction tree



(c) Subgraph of moral graph induced by separator BC



(d) Subgraph of moral graph induced by separator CD

Figure 5.1: Graph, junction tree and induced subgraphs of separators

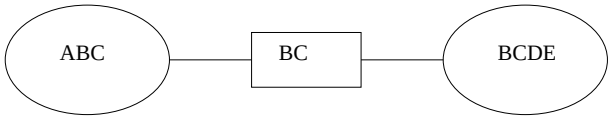


Figure 5.2: Maximal prime decomposition of junction tree from Figure 5.1(b)

The MPD for the junction tree in Figure 5.1(b) can be seen in Figure 5.2. The MPD contains a maximal prime subgraph ABC, which is based on the clique ABC from the junction tree. The MPS and the clique are identical as the clique is only connected to complete separators, namely BC. The MPS $BCDE$ does not correspond to any single clique in the junction tree. Figure 5.3 visualises which parts of the MPD corresponds to aggregated parts of the junction tree. The MPS $BCDE$ has been formed by aggregating cliques BCD and CDE as they are connected by an incomplete separator CD. This can be exploited as structural changes in the Bayesian network only affect the MPD structure locally. For example structural changes to the Bayesian network affecting B, D and E will be local to the MPS $BCDE$ and only the cliques corresponding to that MPS would have to be recompiled - not the entire tree.

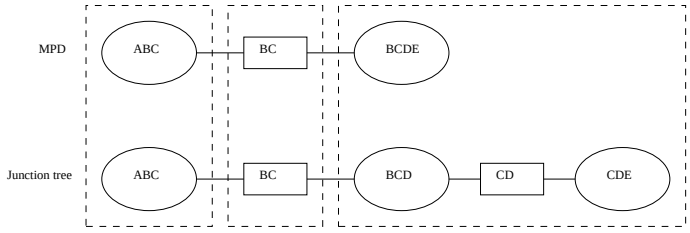


Figure 5.3: MPD and junction tree. Each MPS and separator represents a part of the junction tree, indicated by dotted rectangles.

5.2 Incremental compilation

As previously discussed the biggest reason for the long learning times when using junction tree scoring was that scoring each candidate required a triangulation of the entire graph. Decomposing the graph to a MPD allows us to only re-triangulate the MPS's that are affected by the changes. The method is proposed in [FGO03] and discussed in terms of speeding up the user feedback in a graphical environment (namely Elvira[Elv]). Incremental compilation should lower the time required to re-compile when modifying large Bayesian network structures, although naturally some amount of bookkeeping is required for maintaining a correspondance between the Bayesian network and junction tree. In [FGO04] test are presented that shows that the most performance is gained by incremental compilation when smaller changes are made in between each recompilation. This seems reasonable as smaller increments are more likely to affect a smaller part of the graph (and thus requires fewer MPS's to be retriangulated).

The process of incremental compilation is illustrated in figure 5.4 and explained in the following: The steps in the top (connected by arrows pointing right) represents the initial steps needed for producing a junction tree and its corresponding maximal prime subgraph decomposition. First the Bayesian network graph G is moralized to obtaion moral graph G^M, which is triangulated to obtain a triangulation G^T and construct the junction tree $\mathcal{T}$ and its maximal prime subgraph decomposition $\mathcal{T}^{\mathcal{MPD}}$.

Assume that some modifications are applied to graph G (e.g. add/remove/reverse arc) resulting in the updated structure G' (step 1 in Figure 5.4). To perform incremental compilation (step 2) the moral graph must be updated G'^M. In step 3 the minimal set of maximal prime subgraphs affected by the modifications are marked in $\mathcal{T}^{\mathcal{MPD}}$, and for each marked connected subgraph the following steps are performed: The subgraph g^M of G'^M are identified by the variables of a connected marked subtree of $\mathcal{T}^{\mathcal{MPD}}$ and triangulated g^T to produce a junction tree t and its corresponding maximal

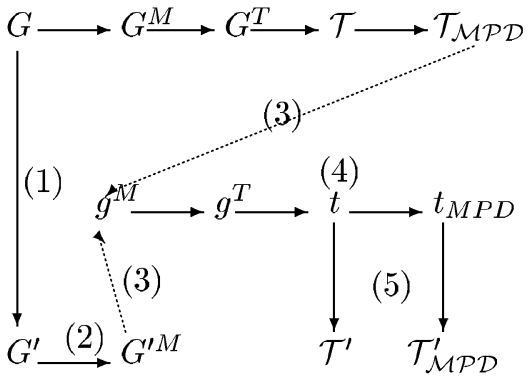


Figure 5.4: The process of incremental compilation. (the figure is from [FGO03])

prime subgraph decomposition t^{MPD} (step 4). The graphs t and t^{MPD} are subgraphs used to replace parts of the junction tree $\mathcal{T}$ and decomposition $\mathcal{T}^{\mathcal{MPD}}$ to produce an updated junction tree $\mathcal{T}'$ and decomposition $\mathcal{T}'^{\mathcal{MPD}}$ (step 5). These steps are formalised in algorithm 2 below:

Algorithm 2.

IncrementalCompilation(Modification list $ModList$, moral graph G^M, Maximal Prime subgraph Decomposition tree $\mathcal{T}^{MPD}$, junction tree $\mathcal{T}$)

```
1  for each modification mod ∈ ModList
2       do L ← ModifyMoralGraph(mod, G, G^M)
3          case mod of
4                  Delete link X → Y
5                       do let M_Y be a MPS ∈ T^MPD that contains both X and Y
6                          MarkAffectedMPSsByRemoveLink(M_Y, nil, L, T^MPD)
7                  Add link X → Y
8                       do MarkAffectedMPSsByAddLink(L, T^MPD)
9  for each connected marked subtree T_MPD ∈ T_MPD
10      do T ← subtree of T corresponding to T_MPD
11         V ← all variables included in T_MPD
12         g^M ← G^M(V)
13         t ← ConstructJoinTree(g^M)
14         t_MPD ← ConstructMPDTree(t, g^M)
15         Replace(T, t, T)
16         Replace(T_MPD, t_MPD, T_MPD)
```

Algorithm 2 performs the steps needed for incremental compilation. The task performed by the algorithm can be divided in two parts: 1) Identify

affected parts of the Bayesian network and the junction tree that must be re-compiled. 2) Perform re-compilation and replace relevant parts of the junction tree. Lines 1 - 8 incrementally modifies the moral graph and marks affected parts in the MPD tree. When all subtrees have been marked in the MPD, lines 9 - 16 identifies the corresponding parts of the junction tree, re-compiles each part and replace the subtrees of the MPD- and junction tree. The additional algorithm called from IncrementalCompilation are described in the next sections, and a few examples of incremental compilation of simple changes to a graph are presented in Section 5.3. IncrementalCompilation can handle single edge modifications (adding or removing a single edge), as well as a sequence of edge modifications. It is relevant to note that in the case of a single edge modifcation the loop in line 9 is only iterated once. For our purposes we are only considering adding or deleting links, which is why steps needed for adding or removing nodes have been ommitted.

5.2.1 Algorithms

This section presents supportive algorithms for doing incremental compilation. The algorithms are based on the work in [FGO03], in which algorithms for doing general purpose incremental compilation is presented (the algorithms from the paper, as well as our comments on each, can be found in appendix C). The algorithms have been designed with the purpose of using incremental compilation for scoring candidates in a structural search, and thus operations for adding or deleting nodes are omitted. Each algorithm is presented along with a description of its purpose.

5.2.2 ConstructJoinTree

This algorithm constructs a junction tree from a directed acyclic graph, through moralization and triangulation as described in Chapter 2. Invoking ConstructJoinTree on the entire Bayesian network yields the complete junction tree and invoking the algorithm on connected maximal prime subgraphs of a network yields the corresponding cliques. This algorithm only differs from the usual procedure (described in Chapter 2) by the requirement that the triangulation be minimal.

Algorithm 3.

ConstructJoinTree(DAG G)

1 Moralize G to obtain G^M.
2 Triangulate G^M to obtain minimal triangulation G^T.
3 Organize the clique decomposition induced by G^T
as a junction tree T.
4 **return** T

5.2.3 ModifyMoralGraph

This algorithm modifies the moral graph such that it represents the moral graph of the modified network. The return value is the set of edges that were removed or added as a result of the modification. The check made in lines 6-8 of the algorithm ensures that no moral edges from the moral graph are deleted if the involved nodes share children. Lines 9-12 check if there are any moral edges that needs to be deleted along with $X \rightarrow Y$. That is, if the only shared child between X and another of Ys parents Z is Y, and there is no edge between X and Z in the graph (the BN DAG), then the moral edge between X and Z can be deleted. In the case of an added edge the list returned contains a single directed link, where the case of a deleted edge the list returned contains one or more undirected links.

Algorithm 4.

ModifyMoralGraph(modification *mod*, DAG G, moral graph G^M)

```
 1  L ← ∅
 2  case mod of
 3          Add link X → Y
 4              do Add X → Y to L
 5                 G^M ← G^M ∪ { all links needed to make Y ∪ X ∪ parents(Y)
                   a complete subgraph in G^M}
 6          Delete link X → Y
 7              do if children(X) ∩ children(Y) = ∅
 8                    then Delete X − Y in G^M
 9                         Add X − Y to L
                   ▷ delete any moral edges no longer needed
10                 for each link X − Z ∈ G^M where Z ∈ parents(Y)\{X}
11                     do if X → Z, Z → X ∉ G
                          and children(X) ∩ children(Z) = {Y}
12                           then Delete X − Z in G^M
13                                Add X − Z to L
14  return L
```

5.2.4 MarkAffectedMPSsByRemoveLink

The purpose of this algorithm is to mark a MPS and any MPSs connected by separators that would be incomplete after the modification. The algorithm is a recursive depth first search, that use a list of links that have been deleted from the moral graph to guide the search. If two MPSs are connected by a separator affected by any of the links, the MPSs are marked and the search continues. It is also worth mentioning that the list L contains undirected links only, which is ensured by algorithm 4 ModifyMoralGraph.

Algorithm 5.

MarkAffectedMPSsByRemoveLink(Maximal Prime Subgraph M_Y, M_Z, Link list L, Maximal Prime subgraph Decomposition tree $\mathcal{T}^{MPD}$)

1 Mark M_Y in $\mathcal{T}^{MPD}$
2 **for each** neighbour $M_K \neq M_Z$ of $M_Y \in \mathcal{T}^{MPD}$
3 **do** $S \leftarrow$ separator between M_Y and M_K
4 **for each** link $X - Y \in L$
5 **do if** $|S| \geq 2$ and $X \in S$ and $Y \in S$ **OR** either $X \in S$ or $Y \in S$
6 **then** MarkAffectedMPSsByRemoveLink($M_K, M_Y, L, \mathcal{T}^{MPD}$)
7 continue to next neighbour

5.2.5 MarkAffectedMPSsByAddLink

This algorithm marks two MPSs as well as the path between them. The link list L (produced by algorithm 4 ModifyMoralGraph) contains a single edge that has been added. If the path between the MPS's that contains families for the nodes of the link contains an empty separator, this separator is removed and a new separator is created, directly connecting the two MPS's. Otherwise all MPS's on the path are marked for re-triangulation. The family of a node X is defined as X and its parents in the Bayesian network prior to modification .

Algorithm 6.

MarkAffectedMPSsByAddLink(link list L , Maximal Prime subgraph Decomposition tree $\mathcal{T}^{MPD}$)

1 $(X \rightarrow Y) \leftarrow$ the single element from L
2 Let M_Y be a $MPS \in \mathcal{T}^{\mathcal{MPD}}$ containing the family of Y
and let M_X be some $MPS \in \mathcal{T}^{\mathcal{MPD}}$ containing the family of X
such that the path from M_X to M_Y is minimal in terms of
the number of separators.
3 **if** The path between M_X and M_Y contains an empty separator S
4 **then** Delete S
5 Connect M_X and M_Y directly, with a new separator
containing X
6 Mark M_X and M_Y in $\mathcal{T}^{MPD}$
7 **else** Mark M_X, M_Y and all M_Z on the path between them in $\mathcal{T}^{MPD}$

5.2.6 Replace

Algorithm 7 replaces a subtree with another one and connects the separators, possibly merging cliques as necessary. This is used to merge the newly compiled parts of both the junction tree and the MPS with the parts that are unaltered.

Algorithm 7.

Replace(Cluster tree t_{old}, Cluster tree t_{new}, Cluster tree T)

1 $\mathcal{S} \leftarrow$ all separators connecting a cluster in t_{old} to a cluster outside t_{old} in T
2 Delete t_{old} in T, but keep $\mathcal{S}$
3 $T \leftarrow T \cup t_{new}$
4 **for each** separator S in $\mathcal{S}$
5 **do** $C \leftarrow$ the cluster in T that S is associated to
6 let C_{new} be a cluster in t_{new} such that $C \cap C_{new}$ is maximal.
7 **if** $S = C_{new}$
8 **then** Merge C_{new} and C in T
9 **else** Connect S to C_{new} in T

5.3 Examples with incremental compilation

In this section we provide some examples of incremental compilation.

5.3.1 ASIA network

In our examples we will use the *asia* network[LS88] that is shown in figure 5.5. The moralized and triangulated graph are shown in figures 5.6(a). This leads to the junction tree shown in figure 5.7(a). From this structure we can construct the MPD-tree shown in figure 5.7 by using algorithm 1 ConstructMPDTree. One noteworthy detail is that the cliques LBS and LBE is merged in the MPS-tree (while separate in the junction tree). The reason is that the nodes (L and B) making up the separator between the cliques are not fully connected in the moral graph.

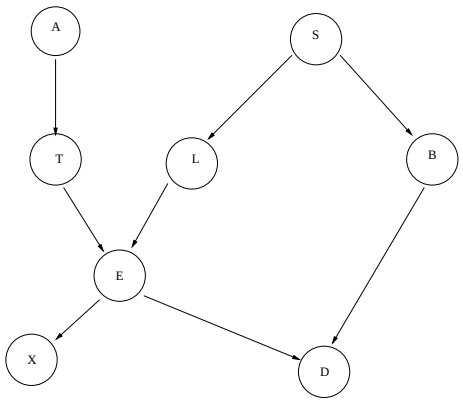


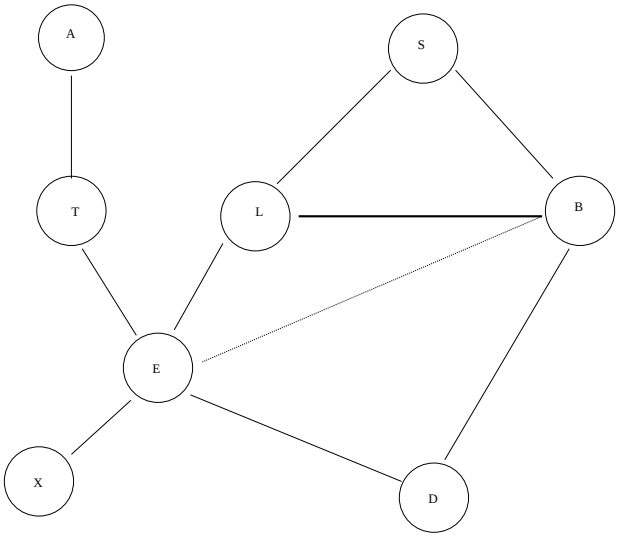
Figure 5.5: Asia network

5.3.2 Example 1 - Deleting a link

This example is based on the Asia example from [OM02]. The link $A \rightarrow T$ is deleted, splitting the Bayesian network in two graphs. In this example we will walk through the algorithm is some detail to show mechanics in play. The network is shown in figure 5.8

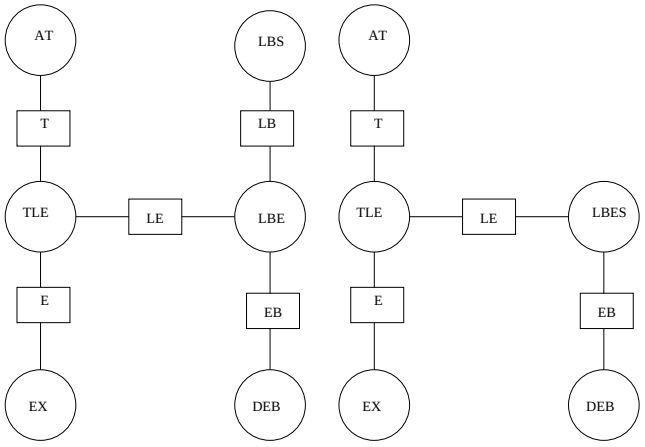
Having deleted the link, we start the IncrementalCompilation algorithm with the various moral graph, junction tree and MPD-tree for asia as parameters.

- The first order of business is the ModifyMoralGraph algorithm, that updates the moral graph and returns a list of changed edges.
- Since the nodes A and T have no common children and T has no other parents, the only action takes in the removal of the edge $A - T$ from the moral graph.
- ModifyMoralGraph returns the list $L = \{A - B\}$.



(a) Moralized and triangulated graph for asia

Figure 5.6: Asia network



(a) Junction tree for Asia network

(b) Maximal prime subgraph decomposition tree for Asia network

Figure 5.7: Junction tree and MPD for asia network

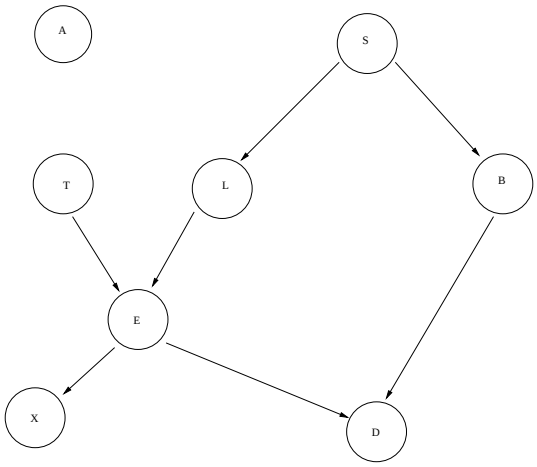
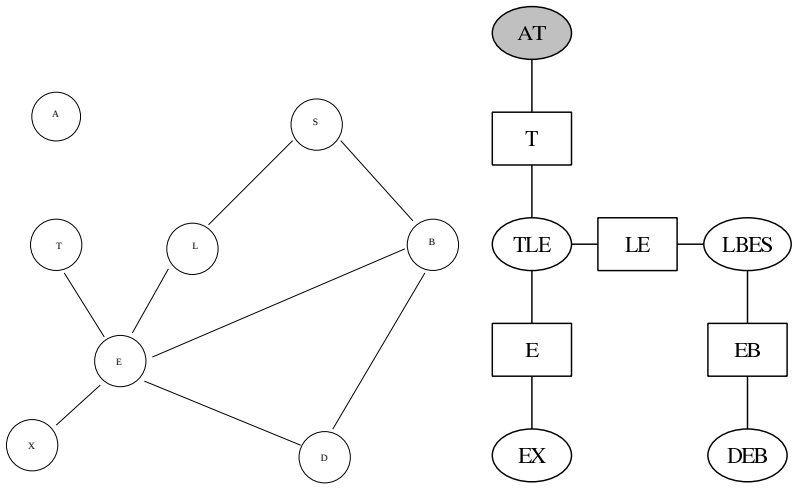


Figure 5.8: Asia network with link $A \rightarrow T$ removed

- As we are deleting edges we need to call MarkAffectedMPSsByRemoveLink with this list as a parameter. Another parameter is a MPS that contains both A and T, which can only be the MPS AT.
- The first thing that happens in MarkAffectedMPSsByRemoveLink is that AT is marked for recompilation. The next thing is to determine whether other MPSs need to be marked as well.
  - We check ATs neighbour TLE by checking if the separator between the two MPS (the separator T) contains both A and T. Since it does not TLE is not marked.
  - AT has no other neighbours, so we are done.
- By now we have the MPS AT marked in the MPD-tree and the moral graph updated to reflect the removal of the edge. The next step is to update the junction tree and the MPD-tree. This situation is shown in figure 5.9
- We extract the part of the moral graph that is included in the marked subgraph and use this to generate new junction trees and MPD-trees to replace the old parts.
- The junction tree consists simply of the two nodes A and T (as they are not connected).
- The MPD-tree has both of these nodes connected by an empty separator. The algorithm does not allow for more than one MPD-tree so empty separators are introduced at this step.



(a) Moral graph for asia without $A \rightarrow T$ (b) Marked subtrees in the MPD-tree for asia

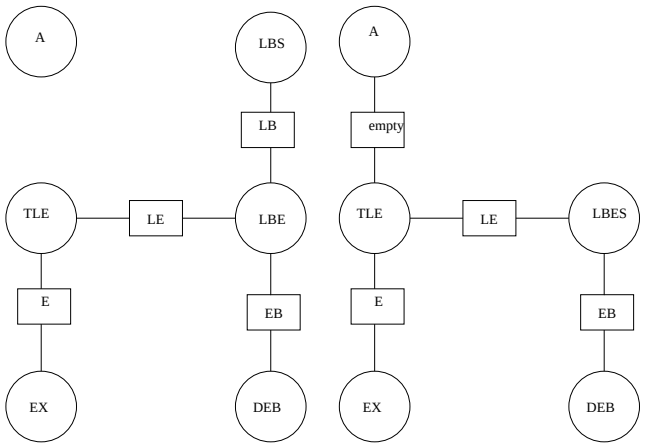
Figure 5.9: Asia network with marked MPSs and updated moral graph

- All that remains at this step is to use the REPLACE algorithm to replace the old AT cliques and MPS with the newly generated ones.
- In the case of the junction tree we have the separator T that was connected to the AT clique before the modification. This separator "contains" the same nodes as the T clique, so the T clique is merged into the TLE clique. The A clique remains separate.
- In the case of the MPD-tree, the situation is similar except that the A clique is still connected via the empty separator introduced previously.
- Having updated both the junction tree and the MPD-tree (to the states shown in figure 5.10) we are now done.

5.3.3 Example 2 - Adding a link

This example shows what happens when an edge is added between the nodes A and B, resulting in the graph shown in Figure 5.11. In this scenario the cliques AT, TLE, LBS and LBE, and the MPSs AT, TLE and $LBES$ needs to be recompiled and the new structure inserted with REPLACE.

- As Before, the first step is MODIFYMORALGRAPH. As A and S now has a common child, they are "married" by adding the edge $A - S$ to the moral graph.



(a) Junction tree (b) Maximal prime subgraph decomposition tree

Figure 5.10: Updated junction tree and MPD-tree

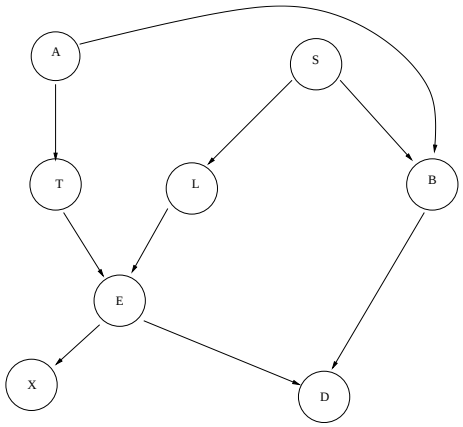


Figure 5.11: Asia network with link $A \rightarrow B$ added

- ModifyMoralGraph returns the list $\{(A \rightarrow B)\}$ and leaves the moral graph in the state shown in Figure 5.12.

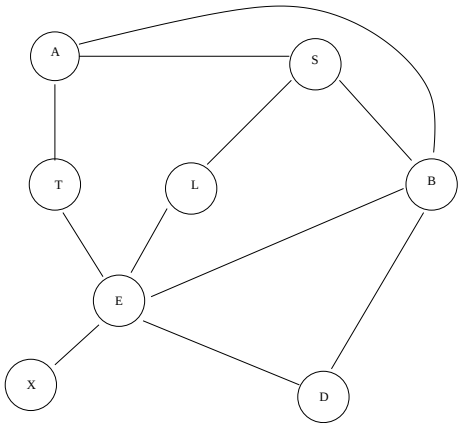


Figure 5.12: Moral graph for the asia network with $A \rightarrow B$ added

- The next step is MarkAffectedMPSsByAddLink. This algorithm First identifies two MPS, one that holds the family of A, which is the MPS AT; and another that holds the family of B. The family of B is the set $\{B, S\}$ as S is a parent in the graph previous to the modification, and the MPS that holds this set is the MPS $LEBS$.

- The path between AT and $LEBS$ contains the additional MPS TLE, and all of these MPS's are marked for re-triangulation as shown in Figure 5.13.

- The variables in the marked MPS's (that is A, T, L, E, B and S) are saved as the set V.

- The part of the moralized graph that holds the variables in V are extracted and stored as g^M(shown in Figure 5.14(a).

- A new junction tree is created from g^M, this junction tree is shown in Figure 5.14(c). Notice that the outcome of this step depends on the triangulation heuristic chosen, and as such there are different possibilities. The heuristic employed here is minimum fill-ins.

- From the newly create join tree a new MPD (shown in Figure 5.14(d))is created.

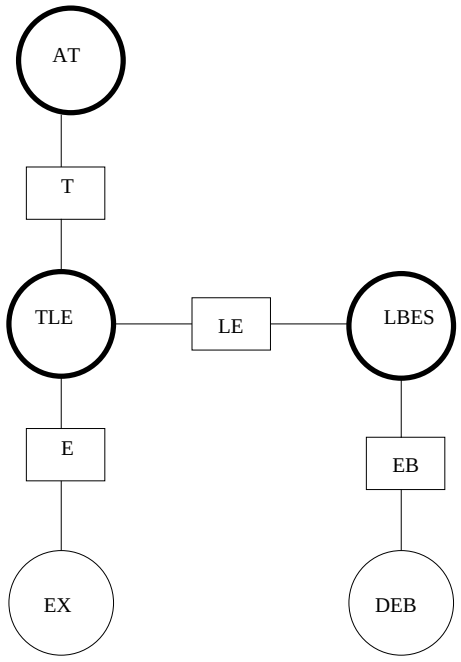
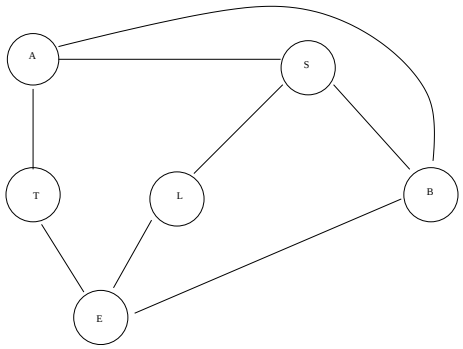
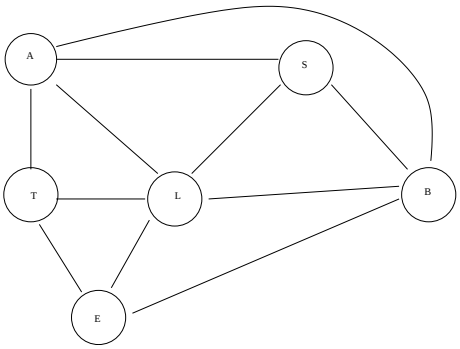


Figure 5.13: Marked MPS's in the asia MPD

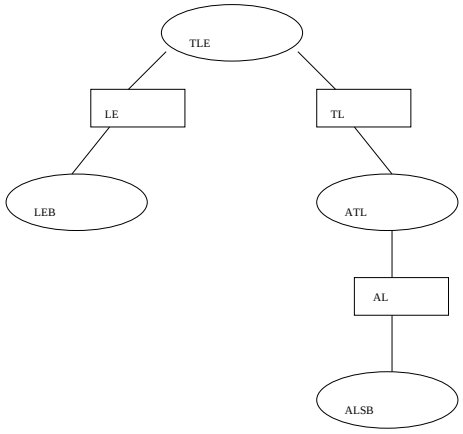
- All that remains now is to replace the obsolete parts of the junction tree and MPD with the newly created parts. This is done in REPLACE:
- The separators in the junction tree that needs to be updated are $S_1 = E$ and $S_2 = EB$:
  - For the separator S_1 connected to the clique $C_1 = \{EX\}$ in the unaltered part of the junction tree, the new clique that has the most nodes in common with C_1 is either LEB or TLE. TLE is chosen arbitrarily and S_1 is connected.
  - For the separator EB connected to the clique $C_2 = \{DEB\}$ in the unaltered part of the junction tree, the new clique that has the most nodes in common with C_2 is LEB and S_2 is connected.
  - We have now constructed the new join tree as shown in Figure 5.15(a).
- The separators in the MPD that needs to be updated are $S_1^M = E$ and $S_2^M = EB$
  - For the separator S_1^M connected to the cluster $C_1^M = \{EX\}$ in the unaltered part of the MPD, the new cluster that has the most



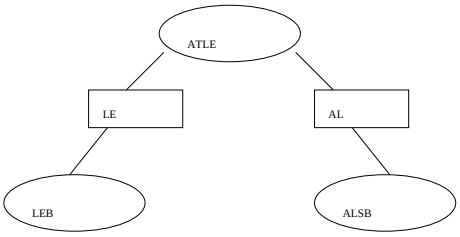
(a) Part of moral graph



(b) Part of moral graph triangulated

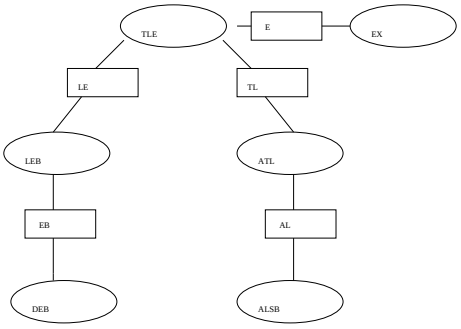


(c) New join tree to be included in the junction tree for asia

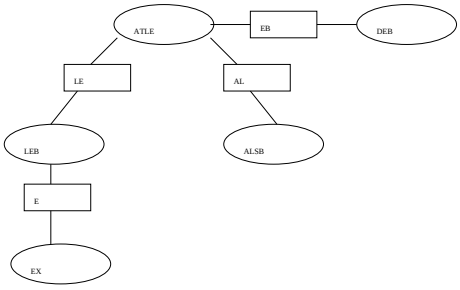


(d) New MPD to be included in the MPD for asia

Figure 5.14: Affected part of asia network when adding link



(a) Join tree



(b) MPD

Figure 5.15: Final MPD and join tree

nodes in common with C_1^M is either LEB or $ATLE$. $ATLE$ is chosen and S_1 is connected.

- For the separator S_2^M connected to the cluster $C_1^M = \{DEB\}$ in the unaltered part of the MPD, the new cluster that has the most nodes in common with C_2^M is LEB and S_1 is connected.
- We have now constructed the new MPD as shown in Figure 5.15(b).

- We have only a single connected marked subgraph so we only need to iterate through the loop once, and thus we are done.

Chapter 6

Incremental Compilation in JTC Scoring

In the previous chapter we introduced a way to do incremental compilation of Bayesian network and we hope to apply this method in combination with the JTC scoring to produce a learning method that allows balancing between accuracy and effective complexity as well as a speedier learning than those that were found in [KP07]. Bear in mind that the search strategy we are proposing is a simple greedy search as described in Section 4.1 and propose to use incremental compilation as part of the score function. The important distinction is that the point of doing incremental compilation of a (sub-)graph is to speed up the calculation of the junction tree score, not to produce a network. For this reason some of the algorithms have been modified for this purpose. These changes are described in Section 6.3. But before we get to that we describe a few observations that may allow us to discard candidates before without doing any compilation at all.

6.1 Establishing bounds on complexity

In this section we will show how knowing the score of a network can be used in determining bounds on "nearby" networks in the search. In the Figure 6.1 we have a selected network G_0 and three candidate networks G_1, G_2, G_3 that can be reached with a single arc-operation. For network G_0 we know that the score is:

$$Sc_0 = LL_0 - Comp_0$$

(as per junction tree scoring described in Section 4.3. The score is composed of the *Log-Likelihood* for the network (a negative number) and the size of the corresponding junction tree, which is calculated as the total sum of the size

of the cliques (a positive number). The two components of the score can be calculated and cached individually and, depending on the size of the database, the log-likelihood is usually faster to compute than the junction tree size. In this setup two different types of bounds are interesting: A bound on the total score and a bound on the size of the junction tree. The correlation of these two bounds is that a lower bound on the size of the junction tree imposes an upper bound on the total score, and vice versa. We denote bounds as $\overline{Sc}$ for an upper bound on the score function and $\underline{Sc}$ for a lower bound, and $\overline{Compl}$ for an upper bound on the junction tree size and $\underline{Compl}$ for a lower bound.

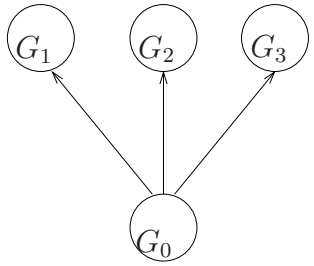


Figure 6.1: Network G_0 and candidates G_1, G_2 and G_3

These properties can be exploited to reason about the score for a candidate network without compiling it. A discriminating bound for the candidates would be an upper bound on the score. This type of bound can be produced when adding edges. The addition of a link will always produce a network of at least the same size as the one previous to the addition.

Theorem 1. *Given a network G_0 with score $Sc_0 = LL_0 - Compl_0$ and a candidate G_1 (for which the score is unknown) where the only difference is the addition of one link, it will always be the case that $Compl_0 \leq Compl_1$ if optimal triangulation is used.*

Proof:

A junction tree JT_1 for graph G_1 with moral graph M_1 and optimal triangulation T_1, cannot be smaller than the junction tree JT_0 for graph G_0 with moral graph M_0 and optimal triangulation T_0, when the difference from G_0 to G_1 is the addition of a single edge $X \rightarrow Y$.

To show that this is the case, a valid triangulation T' for M_0 can be constructed from T_1 as $T' = \mathcal{E}_{T_1} \cup (\mathcal{E}_{M_1} \setminus \mathcal{E}_{M_0})$. The triangulation T' may not be the optimal triangulation T_0 for M_0, thus we have that $\overline{size_1} = size' \geq size_0$, as for any triangulation of a moral graph M, we know that $size_{non} \geq size_{optimal}$ comparing the junction tree for the optimal triangulation $T_{optimal}$ with any non-optimal triangulation T_{non}.

□

Theorem 1 can be used in establishing the upper bound for the score of new network G_1 using only the log-likelihood:

Corollary 1. *Given a network G_0 with score $Sc_0 = LL_0 - Compl_0$ and a candidate G_1 (for which the score is unknown) where the only difference is the addition of one link, then $\overline{Sc_1} = LL_1 - Compl_0$*

The upper bound is useful in that it describes the maximum score a particular network can archive. If the upper bound for a candidate is lower than the score for the best score so far, the candidate can be ignored.

Another way to use the two part nature of the score is to compare the candidates directly and discriminate based on log-likelihood alone.

Proposition 1. *Given a current best score Sc_0 and a candidate G_1 (for which the score is unknown, but the Log-Likelihood LL_1 is known), it is the case that if $LL_1 < Sc_0$ the G_1 can be ignored since the junction tree size can only reduce the score further.*

The combination of the observations regarding the upper bound when adding a link and the possibility to complete ignore certain networks based only on their log-likelihood should allow us to save quite a few junction tree size measurements (with associated triangulations) and thus speed up the search. The problem is that Theorem 1 demands that the triangulation is optimal. In practice this can not be guaranteed as we employ heuristics in out triangulation.

6.2 Predicting junction tree state space with minimal re-compilation

The purpose of our work is to use incremental compilation for speeding up structural learning, using likelihood and the junction tree state space as the score metric. Acknowledging that the most expensive step in the process will be the triangulation part, additional steps should be taken such that as few unnecessary triangulations will be performed. This involves identifying the special cases where triangulation can be omitted to either score or reject a candidate network structure, as well as measures for only performing "enough" triangulation to be able to reject certain candidates.

We have the following ideas:

- When adding or deleting an edge such that no changes are implied in the moral graph, the state space of the junction tree remains unchanged and just calculating a new likelihood will be sufficient for scoring the candidate.

- Maintain the product of the state space for each nodes family. When adding a link $X \rightarrow Y$ it should be checked if $|X| \cdot |Y| \cdot \sum_{Z \in pa(Y)} |Z|$ alone would make the state space so large that new score is worse than the best so far.

- If the scoring of a candidate reaches the point at which triangulation is performed, the score of the state space should be incrementally updated as cliques are identified, and triangulation should be aborted if it reaches a point where the induced state space makes the total score worse than best so far.

6.2.1 Special cases

Predicting junction tree state space of a modified Bayesian network without compiling a new junction tree is possible when the modification is an instance of some special cases. One such case is deleting the single edge of a node.

Theorem 2. *If X only has a single edge $X \rightarrow Y$, and the edge were deleted, then the state space S_1 of a resulting junction tree can be calculated without compilation like this*

$S_0 \leftarrow$ *size of junction tree JT,* $\sum_{C \in JT} \prod_{Z \in C} |Z|$

$C_Y \leftarrow$ *the single clique* $\in JT$ *where* $X \in C$

$S_1 \leftarrow S_0 - \prod_{Z \in C_Y} |Z| + \prod_{Z \in C_Y \setminus \{X\}} |Z| + |X|$

Proof: As the family for Y (including X) becomes a fully connected set after moralization, it will also be fully contained in a single clique C_Y. Removing X from the set of parents of Y would result in two new cliques: C_X containing only X and C'_Y containing C_Y without X. The size of a resulting JT would be the same as the size of the JT before the modification except that the contribution of C_Y would have been replaced by the sum of contributions from the two new cliques C_X and C'_Y. □

Another special case of modifications is edge additions where the new edge connects two disjoint subgraphs.

Theorem 3. *When adding an edge $X \rightarrow Y$ such that two disjoint subgraphs of a Bayesian network becomes connected, then the state space S_1 of a resulting junction tree can be calculated as*

$S_0 \leftarrow$ *size of junction tree JT,* $\sum_{C \in JT} \prod_{Z \in C} |Z|$

$C_Y \leftarrow$ *clique that contains* $Y \cup pa(Y)$

$C'_Y \leftarrow C_Y \cup \{X\}$

$S_1 \leftarrow S_0 - \prod_{Z \in C_Y} |Z| + \prod_{Z \in C'_Y} |Z|$

Proof: When adding edge $X \rightarrow Y$, X becomes a member of the family for Y, and as the two subgraphs were disjoint before, triangulation would not introduce undirected cycles either. This means that the clique C_Y with the family of Y will be replaced by a clique C'_Y which is C_Y extended with a single new member X, and a separator will be introduced connecting this clique to some clique containing X. The size a resulting JT would be the same as the size of the JT before the modification except that the contribution of C_Y would have been replaced by the contribution from the new clique C'_Y. □

If structural search was performed using a full triangulation of each candidate structure to compute the junction tree size, then these two cases would improve the time needed to score a candidate network as the costly triangulation could be omitted.

But when using incremental compilation these two cases of modifications become trivial, as the moral subgraph is already triangulated in terms that no fill-in edges are needed. Thus the before costly triangulation becomes less expensive and the method described in these cases might become unnecessary. This leaves us to believe that predictions about junction tree state space is most interesting when the modifications performed are non-trivial (cases there the moral subgraph does need triangulation). Such special cases would have to be very carefully explored to ensure that they are worth the effort to detect them.

6.3 Learning process

The purpose of using incremental compilation is to produce a junction tree complexity score for a candidate quicker than using a complete compilation. This means that only the operations needed to produce a complexity score is required to be performed and remaining operations should be postponed and only performed for the chosen candidate. Structural learning is performed

as a regular greedy search, with the score function being Algorithm 8 described below in Section 6.3.1. When all candidates have been scored and one selected, Algorithm 10 (COMMIT described in Section 6.3.3) is then used to update the graph to the chosen candidate. The data structures that are maintained in between search steps and for each candidate are described in Section 6.3.2

6.3.1 Scoring

This algorithm is similar to Algorithm 2 INCREMENTALCOMPILATION in that is utilises the other algorithm to perform a partial triangulation of a graph. The different is that this algorithm does not alter the graph, instead it returns a junction tree score(Δ_{size}) and enough information that the actual changes can be performed later. This is very important in the context of learning as we do not know which changes to perform until we have evaluated all candidates.

Algorithm 8.

SCORE(Modification list $ModList$, moral graph G^M, Maximal Prime subgraph Decomposition tree $\mathcal{T}^{MPD}$, junction tree $\mathcal{T}$)

```
1  for each modification mod ∈ ModList
2       do L ← ModifyMoralGraph(mod, G, G^M)
3          case mod of
4                  Delete link X → Y
5                       do let M_Y be a MPS ∈ T^MPD that contains both X and Y
6                          MarkAffectedMPSsByRemoveLink(M_Y, nil, L, T^MPD)
7                  Add link X → Y
8                       do MarkAffectedMPSsByAddLink(L)
9  Δ_size ← 0
10 for each MPS M ∈ M_marked
11      do S_MPD ← ∅
12         V ← ∅
13         size_deleted ← IdentifyMPDSubtree(M, S_MPD, V, null, size_deleted)
14         S_JT ← JT separators corresponding to MPS separators in S_MPD
15         g^M ← G^M(V)
16         C_JT ← ConstructJoinTree(g^M)
17         size_new ← GetNewSize(S_JT, C_JT)
18         Δ_size ← Δ_size + size_new − size_deleted
19 return ⟨Δ_size, S_JT, S_MPD, C_JT, ModList, g^M⟩
```

The IdentifyMPDSubtree algorithm performs a depth-first recursive search of the MPD to identifies the clusters that make up a particular subtree. As it moves along it uses the marks left by MarkAffectedMPSsByAddLink and MarkAffectedMPSsByRemoveLink and removes those that it finds to ensure that MPDs are only added to the list $\mathcal{S}_{MPD}$ once. At the same time it performs this walk it collects the state space of the junction trees corresponding to the MPSs it ads to the list. In this way the state space for the part of the junction tree that are to be replaced are collected and this is the final value to be returned. This value can then be used together with the score for the new parts of the junction tree to calculate the total score for a candidate.

Algorithm 9.

IdentifyMPDSubtree(MPS M, List of MPD separators $\mathcal{S}_{MPD}$, List of variables V, MPD Separator S_{ignore}, State space of deleted cliques $size_{deleted}$)

```
1  V ← V∪ variables in M
2  Z_MPD ← list of separators in M
3  for each separator S ≠ S_ignore in Z_MPD
4        do M' ← other MPS in S
5           S_JT ← separator in JT that S represents
6           if M' is marked
7              then size_deleted ← size_deleted+IdentifyMPDSubtree(M', S_MPD, V, S, size_deleted)
8              else  S_MPD ← S_MPD ∪ S
9                    C ← list of cliques from JT associated with M
10                   for each C ∈ C
11                         do size_deleted ← size_deleted+ state space of C
12                   if M has pointer to location in M_marked
13                      then if S_ignore ≠ null
14                                 then in M_marked: remove M
15                   return size_deleted
```

Replace (Algorithm 7) substitutes the obsolete parts of the junction and MPD trees with new substructures which means that the algorithm enforces a legal tree structure. Calculating the complexity of the junction tree only requires a valid set of cliques, and as such a legal tree structure is not a requirement for our purpose. Therefore the algorithm has been replaced by GetNewSize and ReplaceFinal (Algorithm 11).The difference between Replace and GetNewSize is that operations that connects the cluster tree t_{new} to T producing a valid tree structure, has been omitted. The only task

that GetNewSize performs is to calculate the new junction tree score. To do this it need to consider that any pair of clusters where the members of one is a subset of the members of the other should merged. The algorithm does not do the actual merge, it simply calculates the correct score considering this special case as the score would not be accurate if both clusters in any such pairs contributes to the total state space.

6.3.2 Data structures

In this section we describe the data structures that are involved in scoring with incremental compilation. These come in two categories: Some structures are generated for each candidate when scored and used if the candidate was the one with the best score. The other variety are structures that are updated each time a candidate is selected (that is, once for each step in the search) and are reused in the scoring of the next set of candidates.

The structures that are only updated once per step requires a some bookkeeping but on the other hand allows for speedier lookup in the scoring of all the candidates. The structures are:

- A list of parents for each node in the network. By caching this information this list can be created once instead of each time it is needed.
- The log-likelihood for the family of each node in the network, like in regular BIC scoring.

The structures that represents a candidate and are returned from a call to Score are:

- $\mathcal{S}_{JT}$ is a list of junction tree separators that should be disconnected from an obsolete part of the junction tree and reconnected to the new junction tree.
- $\mathcal{S}_{MPD}$ is a list of junction tree separators that should be disconnected from an obsolete part of the MPD and reconnected to the new MPD, when it is created later.
- $\mathcal{C}_{JT}$ is a new junction tree made from the variables that were affected by the change to the network.
- *ModList* is the list of arc operations that transforms the original graph into the candidate.
- g^M is a moral graph over the variables that were affected by the change to the network.

If a candidate is selected, these structures are passed to COMMIT which is described below.

6.3.3 Applying changes

When a candidate is selected the graph needs to be updated to represent it. This is the duty of Algorithm 10. It is not until COMMIT has been called that any changes are made to the graph and the junction tree.

Algorithm 10.

COMMIT($\langle \Delta_{size}, \mathcal{S}_{JT}, \mathcal{S}_{MPD}, \mathcal{C}_{JT}, ModList, g^M, G^M \rangle$)

```
 1  C_MPD ← ConstructMPDTree(C_JT, g^M)
 2  ReplaceFinal(S_JT, C_JT)
 3  ReplaceFinal(S_MPD, C_MPD)
 4  for each modificiation mod ∈ ModList
 5        do case mod of
 6                  Add link X → Y
 7                        do in children(X): add Y
 8                           in parents(Y): add X
 9                  Remove link X → Y
10                        do in children(X): remove Y
11                           in parents(Y): remove X
```

REPLACEFINAL is used in the COMMIT algorithm to unite a newly compiled part of the junction tree or MPD into the old structure. It checks for empty separators to ensure the MPD is never a forest and is responsible that the resulting structures are valid.

Algorithm 11.

```
ReplaceFinal(S, C)
 1  for each separator S in S
 2       do if |S| = 0            ▷ empty separator. connects anywhere.
 3             then in S: overwrite null-pointer with a pointer to first element in C
 4                  continue to next separator
 5          C ← the cluster that S is associated to
 6          C_new ← null
 7          max ← 0
 8          for each cluster C' in C
 9               do max_current ← 0
10                  for each node A in C'
11                       do for each node B in C
12                                do if A = B
13                                     then max_current ← max_current + 1
14                  if max_current > max
15                     then max ← max_current
16                          C_new ← C'
17          in S: overwrite null-pointer with a pointer to C_new
```

ReplaceFinal uses a procedure called MergeInto for merging a clique C_1 into another clique C_2. This is done by adding separators from C_1 to C_2 and updating any pointers. When C_1 has been merged into C_2, C_1 is deleted.

Algorithm 12.

```
MergeInto( Cluster C_1, Cluster C_2)
1  S ← list of separators from C_1
2  for each separator S in S
3       do in S: replace C_1 with C_2
4          in C_2: add S to list of separators
5  delete C_1
```

This concludes the modifications of incremental compilation neccessary for speeding up the JTC scoring method.

Part III

Results

Chapter 7

Experiments

To investigate the practical usability of using the proposed method of the JTC scoring criterion in combination with incremental compilation, a number of experiments have been performed. The experiments consists of the following:

1. Perform structural learning from training data to produce networks that can be used in the remaining experiments.

2. Submit trained networks to an inference bencmarking to measure effective performance.

3. Submit trained networks to a classification benchmarking in relation to the set of training data and a set of test data.

Experiments will be conducted using both the proposed method for doing structural learning as well as using the BIC scoring criterion.

7.1 Test setup

The basis for the experiments are sets of training data. Most of these data sets we have generated from a set of networks, which we will denote the "gold standard" networks. Additionally some experiments were performed on training data for which there exists no gold standard network, and for which traditionally learning methods has been problematic to use. The networks we have used as our gold standard networks are listed in Table 7.1.

The first network, Asia, is the well known toy-network often used as an example in education when introducing Bayesian networks. The network is rather small so we do not expect the significant results to come from this. The Alarm network is a research network, often used in connection with structural

Network name	# Nodes	Dimension (sum of CPT sizes)
Asia	8	36
Alarm	37	752
Hut	74	8148
Big_dense	25	5084

Table 7.1: The original (Gold-standard) networks used for generating the training data

Name	# Training cases	# Test cases
Asia	1000	1000
Alarm	10000	10000
Hut	10000	10000
Big_dense	10000	10000
Elsam	4053	3246

Table 7.2: The data sets available. (all data sets except for Elsam are generated from the respective gold-standard networks)

learning and classification. The Hut network models a real-world domain, and the network produced interesting results in the preliminary work. The network Big_dense was automatic generated with the tool presented in [ICR], and did also produce interesting results in the preliminary work.

In addition we have a large data set from a research project [var04] on establishing a monitoring system giving early warnings when a production plant may be heading into serious production disturbances. The data set proved to be problematic for traditionally learning methods as the networks produced were too complex. It would be interesting to see the performance of the proposed method on these data. The data provided from that project have been split in a training data set and a test data set.

In total we have ten data sets available for the experiments (two sets for each problem), the data sets are summarized in Table 7.2.

7.1.1 Test Setup for Structural Learning

All networks produced during structural learning were learned using a number of different λ-values, to test whether the values represents a balance of accuracy and effective complexity. As the two scoring criteria in question, namely JTC and BIC, differs the networks learned using either is not directly comparable by their λ-values. Learned networks for different score functions should instead be compared on common properties such that the

BIC score or log-likelihood for a given network. Another reason for using a number of different λ-values for training, is that it should produce a bigger span of learned networks that might overlap on comparable features, making comparisons easier.

The following statistics were collected during **structural learning:**

- The total time it took to learn the network.
- The number of steps the algorithm took (which is the height of the search tree as depicted in Figure 4.2 on page 33).
- The BIC score for the final network given the training data (also for the networks scored using IC-scoring).
- The log-likelihood of the network given the training data.
- Total size of the CPTs (conditional probability table) of learned network
- Size of Junction tree (sum of clique sizes) produced using clique size triangulation heuristic on the learned network.

7.1.2 Test Setup for Inference Benchmarks

For the inference benchmarking the set of training data with 40% MCAR (40% of the values missing completely at random [JN07] page 200) were propagated, by randomly withhelding evidence for 40% of the variables. The inference benchmark has two purposes. One purpose is to give a picture of the relationship between the λ-values and effective complexity, namely the inference time. The other purpose is to measure the accuracy of the learned networks. The accuracy is measured in terms of the Kullback-Leibler divergence (also known as the relative entropy) [KL51], [SJ80] from the generating distribution to the induced distribution, that is from the gold-standard network to the learned network.

For proability distributions P and Q of a discrete random variable, the Kullback-Leibler divergence from P to Q is defined to be:

$$\mathrm{D_{KL}}(P||Q) = \sum_{X} P(x_i) log_2 \frac{P(x_i)}{Q(x_i)} \tag{7.1}$$

Note that the Kullback-Leibler divergence is not symmetric, which means that $D_{KL}(P||Q) \neq D_{KL}(Q||P)$. Another property of the Kullback-Leibler divergence is is that $D_{KL}(P||Q) \geq 0$, and $D_{KL}(P||Q) = 0$ if and only if

$P = Q$. Thus it implies that when seeking a network to approximate the true distribution P, one should select the network presenting the distribution Q such that $D_{KL}(P||Q)$ is minimized.

For our purposes we modify the Kullback-Leibler divergence such that an average is calculated for the entire set of cases. For the distributions P and Q we use the gold-standard network for P as the true distribution and the learned network to be the approximated distribution Q. Only the divergence for variables for which evidence has been withheld are considered, so the average Kullback-Leibler divergence for the inference benchmark can be expressed as:

$$\mathrm{D_{KL}}(P||Q) = \frac{1}{N}\sum_{i=1}^{N}\frac{1}{|V_i|}\sum_{X\in V_i} P(x_j|c_i)log_2\frac{P(x_j|c_i)}{Q(x_j|c_i)} \tag{7.2}$$

Where $c_1, \ldots, c_n \in \mathcal{D}$ is a set of cases, P is the distribution presented by the gold-standard network, Q is the distribution presented by the learned network, N is the number of cases in $\mathcal{D}$, V_i is the number of query variables for case i for which evidence is withheld in both P and Q.

Thus the following information was collected during the inference benchmarks for learned networks for which a gold-standard network exists:

- Time taken to propagate all training data cases in the learned network
- Accuracy of the learned network in terms of the average Kullback-Leibler divergence

7.1.3 Test Setup for Classification Benchmarks

Classification is a basic task in data analysis and pattern recognition that requires the construction of a classifier [FGG97], [JN07] Ch. 8, [CS96]. A classifier is a function that assigns a class label to instances described by a set of features. That is you have a set of feature variables $\{F_1, \ldots, F_n\}$ and a class variable C, where the states of C corresponds to the possible classes. Bayesian networks learned from at data set of observations over feature variables and class variable, $\{F_1, \ldots, F_n\} \cup C$, can be used for classification. Classification is then done by calculating $P(C|f_1, \ldots, f_n)$ for a given instance $\{f_1, \ldots, f_n\}$, and predicting the class with the highest posterior probability.

A way to distinguish the quality of a Bayesian classifier is by its ability to classify not-yet-seen cases. For the classification benchmark a set of test data have been generated from the gold-standard networks. In the case where we do not have such a network, we have retained a portion of the

training data such that the remaining portion can be used as a test set. The learned networks are then used as Bayesian classifiers to classify the instances represented in the test data. For each case in the test data, 40% of the variables are randomly selected as the feature variables, and one of the remaining is selected as the class variable. For a given case, only the selected feature variables $F_1, \ldots, F_n$ will receive evidence. The classification will be ranked true or false by comparing the predicted class to the the class observed in the given case. The accuracy of the classifier can then be expressed as:

$$ACC = \frac{\#Correct}{\#Wrong} \tag{7.3}$$

where $\#Correct$ is the number of correctly classified cases and $\#Wrong$ is the number of wrongly classified cases. Thus the following information is collected for the classification benchmarks:

- The accuracy of the network as a classifier for the test data.
- The accuracy of the network as a classifier for the training data.

The results from the experiments can be found in Section 7.2

7.2 Experimental Results

The following pages presents the experimental results of using the JTC score function in combination with incremental compilation for structural learning in comparison to using the BIC score function. The information collected from structural learning and the inference benchmarks can be found in Tables 7.4 p. 78 (Alarm), 7.5 p. 79 (Hut), 7.6 p. 80 (Big_dense), 7.7 p. 81 (Elsam) and 7.3 p. 77(Asia). The colums of the tables are:

- λ the lambda value used in the score function.
- **Learn. Time** the time it took to learn the networks in seconds.
- **Steps** The number of steps taken by the algorithm (as described in Section 7.1.1).
- **CPT** dimension of the resulting network (measured as the sum of the size of each CPT)
- **JT size (RH)** junction tree size of the junction tree used for structural learning with incremental compilation. (measured as the sum of state space for the cliques).

- **Inf. time** the time it took to propagate the set of training data with 40% MCAR (measured in seconds).
- **KL** Average Kullback-Leibler divergence from the gold-standard network to the learned network in question (as described in Section 7.1.2).
- **BIC** the BIC score of the resulting network given the training data.
- **LL** the log-likelihood of the resulting network given the training data.
- **JT size (H)** junction tree size of the junction tree used for the inference and classification benchmarks. Note: JT size (RH) and JT size (H) might differ for some of the results, as the sequence for eliminating equally good candidates might differ between RHugin (RH) used for structural learning and Hugin (H) used for inference and classification benchmarks.

The information collected from the Bayesian classifier experiments can be found in Tables 7.9 p. 82 (Alarm), 7.10 p. 83 (Hut), 7.11 p. 83 (Big_dense), 7.12 p. 84 (Elsam) and 7.8 p. 82 (Asia). The columns of the tables are:

- λ the lambda value used in when learning the network.
- ACC_{test} accuracy of the network as a Bayesian classifier on the set of test data, measured in terms of $\frac{\#Correct}{\#Wrong}$ as described in Section 7.1.3.
- $ACC_{training}$ accuracy of the network as a Bayesian classifier on the set of training data used when learning the network.

A number of observations can be made. Many of the observations are supported by the preliminary results of [KP07], summarized in Appendix A.

1. The λ values for JTC seems to be a good indicator of the junction tree complexity in the learned network (that is, low values produce less complex junction trees than higher values), as opposed to the λ value in BIC where the junction tree complexity seems unpredictable as it rapidly explodes. This can be seen in Figure 7.2 and by inspecting *JT size* and λ columns in data for all networks, Tables 7.4 (Alarm), 7.5 (Hut), 7.6 (Big_dense), 7.7 (Elsam).
2. Networks with low inference time are generally concentrated about the low end of the λ values. This can be seen by inspecting the λ and *Inf. time* columns in data for all networks, Tables 7.4 (Alarm), 7.5 (Hut), 7.6 (Big_dense), 7.7 (Elsam).

3. Higher λ values favor more densely connected networks which can be seen in Figure 7.3, and higher λ values generally also results in higher log-likelihood scores which can be seen in Figure 7.5. Especially networks learned with BIC was susceptible to a rapid increase in network size for higher λ values. For networks learned with JTC only Elsam and Big_dense seems to exhibit a rapid increase in network size for the highest λ values (but the rapid increase for Big_dense with JTC and $\lambda 0.95$ this is somewhat expected, as with such a high λ value learning will be almost entirely based on likelihood).

4. Some of the high λ value networks learned with BIC scoring produced networks which were impossible to compile due to memory limitations or intractable to use for inference, and as a result some benchmarking information have not been collected for them. This relates to the following networks: Big_dense $\lambda 0.9$ and Elsam $\lambda 0.5$ for which inferrence became intractable as a single case took about 10 seconds or more to propagate and Elsam $\lambda 0.8$ and Hut $\lambda 0.9$ which were impossible to compile due to memory limitations.

5. For λ values ≤ 0.5, the learning time for JTC is comparable to the BIC learned networks with $\lambda 0.5$ (which exhibits the same performance as the "unmodified BIC score function" as $2 \cdot \mathrm{BIC}(S|D, \lambda = 0.5) = \mathrm{BIC}(S|D)$, refer to Section 4.2), except for the Elsam network. The JTC learning time is comparable in the sense that it is close to twice the time for BIC $\lambda 0.5$. This can be seen by inspecting the λ and *Learn. Time* columns.

6. Larger λ-values may increase the learning time significantly for JTC scoring, this can be seen in Figures 7.4 and by inspecting the λ and *Learn. Time* columns. This behavior is expected as larger parts of the MPD structure needs to be recompiled when networks grow denser, as more cliques will be collected in the same MPSs. Thus we suspect that for the higher λ values for JTC there is little to gain from using incremental compilation, as almost the entire junction tree will have to be recompiled during the search.

7. For the classification benchmarking all networks performed nearly as good (or bad) as the gold-standard networks with regards to predicting the training data, and with regards to predicting the test data. The BIC learned networks seems to have a slightly higher accuracy as classifiers compared to the JTC learned networks. This can be seen by inspecting the *Acc. Test* and *Acc. Train* columns for in Tables 7.9 (Alarm), 7.10 (Hut), 7.11 (Big_dense), 7.12 (Elsam) and 7.8 (Asia).

8. For the Elsam networks as classifiers there is a big difference between how the network performes on the training data and on the test data, which can be seen in Table 7.12 (Elsam). Elsam learned with BIC $\lambda 0.5$ was intractable to use for the classification benchmark as propagating a single case took about 10 seconds, and Elsam learned with BIC $\lambda 0.8$ could not even be compiled because of memory limitations.

9. From the experiments with Asia there does not seem to be any advantage to using either BIC of JTC based learning. This can be seen by inspecting the data from the learning and inference benchmark in Table 7.3 and the classification benchmark in Table 7.8. This behavior was expected and just verifies an observation in the preliminary work [KP07] that there is nothing to gain by junction tree scoring for learning of small networks.

Table 7.3: Combined results for Asia

λ	Learn. Time	Steps	CPT	JT size (RH)	Inf. time	KL	BIC	LL	JT size (H)
Gold-standard Asia									
			36		0.0666361916666667	0	-2236.72389569369	-2174.55409818285	40
IC									
0.2	0.047000	5	22	18	0.0439679683333333	0.0417475306464872	-2363.30625654305	-2325.31360250865	18
0.3	0.047000	6	28	20	0.048539248	0.0315765173111843	-2349.66107736961	-2301.30679041673	20
0.4	0.094000	11	42	28	0.0531099203333333	0.0215569060202409	-2317.91069948263	-2245.37926905331	28
0.5	0.078000	8	30	22	0.049630451	0.0136662378531017	-2282.82018804326	-2231.01202345089	22
0.6	0.047000	7	30	22	0.0500115066666667	0.0136653226885139	-2282.93870737952	-2231.13054278715	22
0.7	0.047000	7	30	22	0.0499158756666667	0.0136653226885139	-2282.93870737952	-2231.13054278715	22
0.8	0.141000	14	54	36	0.056849284	0.00367301259636479	-2269.3323399935	-2176.07764372724	36
0.850000	0.140000	14	54	36	0.056489667	0.00367301259636479	-2269.3323399935	-2176.07764372724	36
0.9	0.188000	17	70	48	0.060078517	0.00386193392388671	-2294.19272912768	-2173.30701174549	48
0.910000	0.235000	19	94	56	0.05803429	0.00456971154071699	-2337.72656061488	-2175.3943115588	56
0.950000	0.219000	19	94	56	0.0586373193333333	0.00456971154071699	-2337.72656061488	-2175.3943115588	56
0.980000	0.250000	19	94	64	0.0591725456666667	0.00429642881309366	-2335.77621690665	-2173.44396785057	64
BIC									
0.2	0.031000	12	28		0.058566785	0.0158114987130239	-2278.64767350091	-2230.29338654803	26
0.3	0.016000	13	34		0.05343531	0.00985483121550303	-2269.21649726711	-2210.50057739576	26
0.4	0.015000	10	40		0.0584814963333333	0.00307875297217731	-2243.87490374487	-2174.79735095504	42
0.5	0.031000	11	40		0.0576749543333333	0.0030663067466805	-2243.73720550072	-2174.6596527109	42
0.6	0.032000	10	40		0.057478776	0.00310510678158019	-2243.86357375244	-2174.78602096262	42
0.7	0.031000	11	42		0.0621152626666667	0.0032384410672974	-2245.6855189576	-2173.15408852829	44
0.8	0.031000	19	46		0.060329551	0.00713432655661742	-2276.4057776065	-2196.9665918982	56

Table 7.4: Combined results for Alarm

λ	Learn. Time	Steps	CPT	JT size (RH)	Inf. time	KL	BIC	LL	JT size (H)
Gold-standard Alarm									
			752		3.58694955333333	0	-102791.178581756	-100447.146957088	1020
IC									
0.2	16.750000	44	456	375	2.81542774833333	0.0721813778436179	-116510.256706381	-115174.757352444	375
0.5	17.329000	43	483	493	2.97292725833333	0.0259569820598821	-107772.625036896	-106271.339556264	484
0.8	30.063000	50	567	624	3.19081988166667	0.0163685501195988	-105555.459835449	-103768.653803286	615
0.850000	34.656000	52	676	824	3.39956981933333	0.0106891443962561	-104589.66349357	-102452.864527271	815
0.9	31.484000	49	641	831	3.38531768666667	0.0110576194736628	-104388.258582775	-102352.773360569	822
BIC									
0.2	6.234000	65	682		3.58115708166667	0.00767525694215299	-103813.631937629	-101768.93637505	855
0.5	7.172000	73	882		3.75870486466667	0.00567840386194616	-103730.194871646	-101105.247865633	986
0.8	8.328000	84	1192		4.82674396533333	0.00198875398722218	-103911.16895489	-100374.398252051	1999
0.85	9.141000	92	1186		8.73980455266667	0.01014186009447	-105419.369622272	-101785.890345527	9161
0.9	11.344000	115	1803		37.1335960703333	0.00324254308432525	-105640.93652712	-100266.702920072	73740

Table 7.5: Combined results for Hut

λ	Learn. Time	Steps	CPT	JT size (RH)	Inf. time	KL	BIC	LL	JT size (H)
Gold-standard Hut									
			8148		51.3532609866667	0	-356921.644640181	-325832.140714576	93114
IC									
0.2	106.625000	53	965	914	5.92443802133333	0.115954207279515	-378046.423933517	-374615.572144956	914
0.5	153.000000	69	1201	1160	6.06402668466667	0.0793024455672249	-362952.000708101	-358752.08549848	1160
0.8	198.000000	73	1623	1504	6.26566336933333	0.0657893875754167	-358636.053230167	-353119.059347353	1504
0.85	226.969000	78	2252	2038	6.72770770066667	0.05847941175558	-357798.980144513	-349836.640892939	2038
0.9	245.922000	80	2390	2268	6.727917862	0.0577479860752004	-357895.511018971	-349541.732301589	2268
BIC									
0.2	60.485000	120	1457		7.60311296933333	0.0676503291862309	-357483.27268195	-352620.212965547	3095
0.5	83.156000	154	2994		14.485069882	0.0289707564722859	-346684.586157293	-336622.28930091	16591
0.8	118.594000	198	7731		86.2742590893333	0.017322991427133	-357030.407060691	-330476.995768284	179481
0.9	213.359000	283	20923		ERROR	ERROR	ERROR	ERROR	ERROR

Table 7.6: Combined results for Big_dense

λ	Learn. Time	Steps	CPT	JT size (RH)	Inf. time	KL	BIC	LL	JT size (H)
Gold-standard Big_dense									
			5084		2352.071843194	0	-249928.070180471	-233455.376425191	2379136
IC									
0.2	1.531000	10	173	150	1.999047253	0.206108345468155	-262977.789783019	-262420.564190515	150
0.5	4.984000	27	425	394	2.402551444	0.139357874721931	-254426.47470897	-253026.502972429	394
0.8	8.172000	31	486	583	2.80545502133333	0.12147482176031	-251960.683000799	-250362.688946261	583
0.85	18.406000	38	858	1176	3.34010421433333	0.0978880647828956	-249515.195643236	-246798.145233503	1176
0.9	61.313000	66	3842	3355	5.135321235	0.0706170407447025	-253727.335254266	-241763.103111069	3355
0.91	64.563000	68	3826	3355	5.07886026066667	0.0706195347443623	-253726.984622994	-241762.752479796	3355
0.95	840.969000	103	57944	49088	40.8214111046667	0.106163660947062	-422495.335255218	-233632.70075766	49088
BIC									
0.2	2.594000	49	622		3.87288484266667	0.0993473859107273	-249772.354738193	-247732.2643458	1729
0.5	3.875000	66	1261		12.090671341	0.069700778425427	-246576.397269579	-242883.050780417	15506
0.8	5.797000	87	3278		820.498713125333	0.0547506511565102	-250266.750430877	-240038.667447797	956864
0.9	10.610000	113	14998				-281065.944849188	-231864.306582091	12130176

Table 7.7: Combined results for Elsam

λ	Learn. Time	Steps	CPT	JT size (RH)	Inf. time	BIC	LL
IC							
0.2	1405.547000	84	1706	1682	5.46121662666667	-244809.257673303	-239185.274725076
0.5	433.500000	93	2540	2479-	6.73545138766667	229648.550854843	-221262.419708262
0.8	778.297000	122	7102	6692	8.37966471733333	-229555.787021231	-206008.992831053
BIC							
0.2	50.515000	98	2488		7.691395364	-218975.035359967	-210755.048465919
0.5	118.781000	165	6234			-212877.616289629	-192246.653731398
0.8	144.734000	202	22303			MEM	MEM

lambda	Acc. test	Acc. Train
Gold-standard	0.8620	0.858
IC		
0.2	0.816	0.826
0.3	0.816	0.826
0.4	0.847	0.833
0.5	0.846	0.851
0.6	0.846	0.851
0.7	0.846	0.851
0.8	0.858	0.861
0.85	0.858	0.861
0.9	0.857	0.86
0.91	0.858	0.861
0.95	0.858	0.861
0.98	0.858	0.862
BIC		
0.2	0.849	0.852
0.3	0.856	0.859
0.4	0.858	0.861
0.5	0.858	0.861
0.6	0.858	0.861
0.7	0.858	0.861
0.8	0.858	0.86

Table 7.8: Classification results for Asia

lambda	Acc. test	Acc. Train
Gold-standard	0.9106	0.9092
IC		
0.2	0.8939	0.9038
0.5	0.9013	0.8964
0.8	0.9058	0.9079
0.85	0.9068	0.9089
0.9	0.9069	0.9094
BIC		
0.2	0.9075	0.9097
0.5	0.9082	0.9107
0.8	0.909	0.9113
0.9	0.9091	0.9113

Table 7.9: Classification results for Alarm

lambda	Acc. test	Acc. Train
Gold-standard	0.8136	0.8086
IC		
0.5	0.7826	0.7849
0.2	0.7654	0.7712
0.8	0.7876	0.7904
0.85	0.7894	0.7938
0.9	0.7883	0.7941
BIC		
0.2	0.7928	0.7939
0.5	0.8043	0.8068
0.8	0.8078	0.8141
0.9	Error	Error

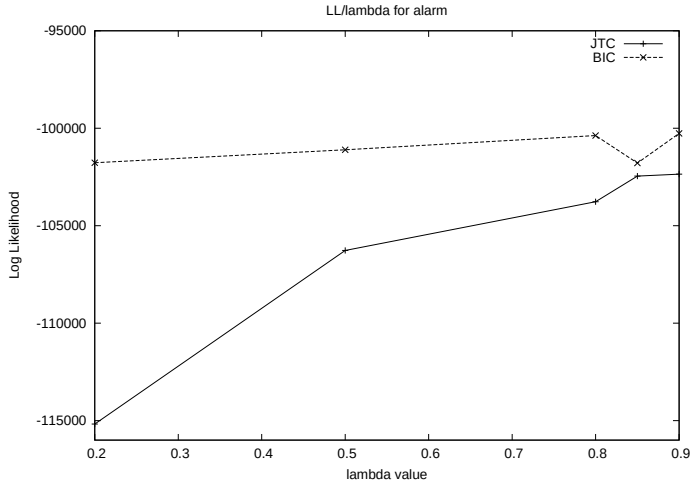
Table 7.10: Classification results for Hut

lambda	Acc. test	Acc. Train
Gold-standard	0.5078	0.505
IC		
0.5	0.4531	0.4551
0.2	0.429	0.4222
0.8	0.4615	0.4643
0.85	0.4693	0.4683
0.9	0.4799	0.4866
0.91	0.4798	0.4859
0.95	0.4762	0.494
BIC		
0.5	0.482	0.4858
0.2	0.4665	0.4786
0.8	0.487	0.4952
0.9	Time	Time

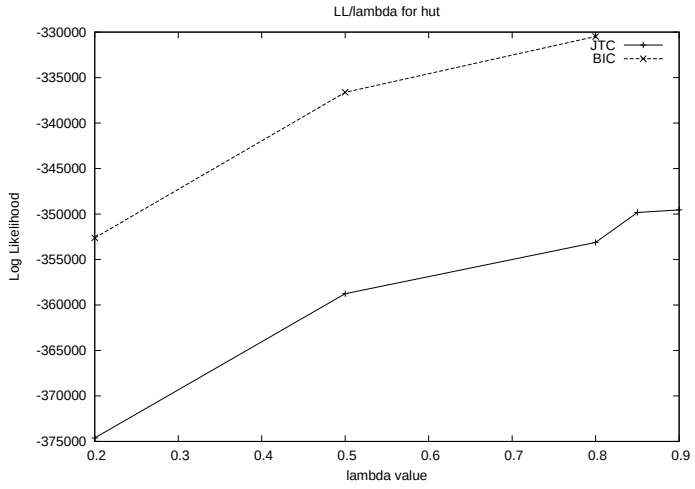
Table 7.11: Classification results for Big_dense

lambda	Acc. test	Acc. Train
IC		
0.2	0.429583975346687	0.708610905502097
0.5	0.415100154083205	0.734270910436714
0.8	0.428043143297381	0.754502837404392
BIC		
0.2	0.430816640986133	0.749568221070812

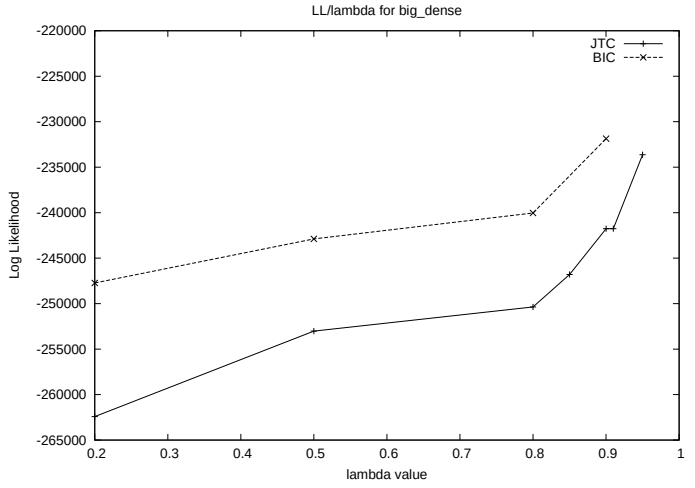
Table 7.12: Classification results for Elsam. *Elsam with BIC and* $\lambda 0.5$ *is missing as running the classification benchamrk proved to be intractable, as propagating a single case took* 10 *seconds.*



(a) alarm

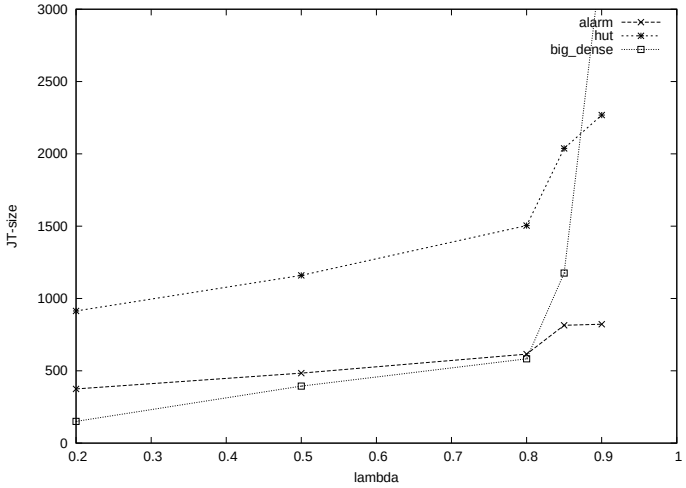


(b) hut

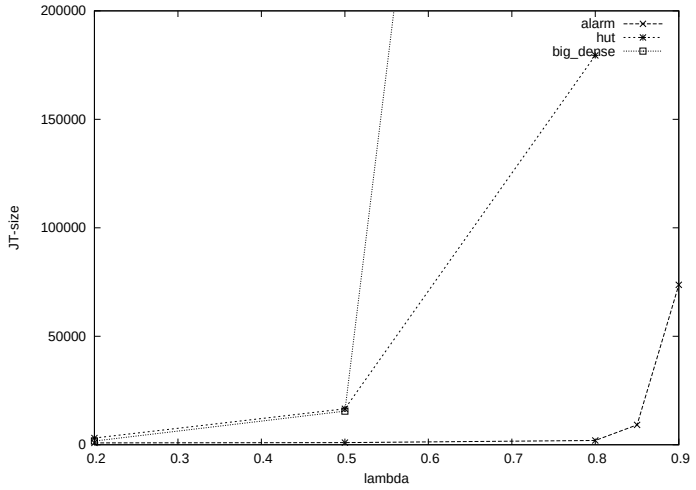


(c) big_dense

Figure 7.1: In all networks the likelihood of learned networks increases when learned with higher λ value

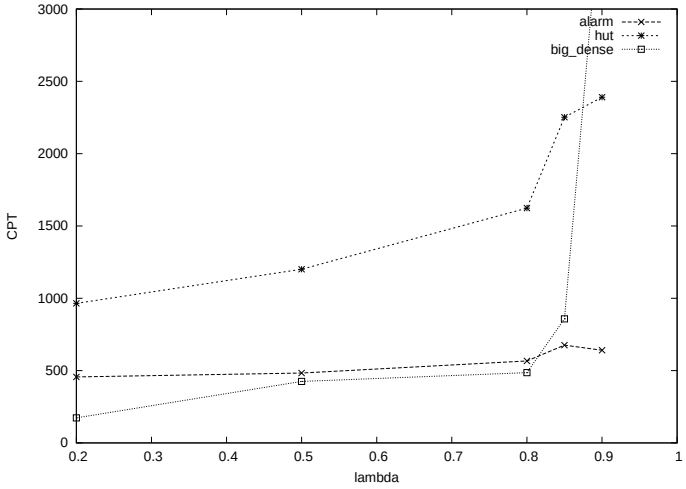


(a) JTC scored networks

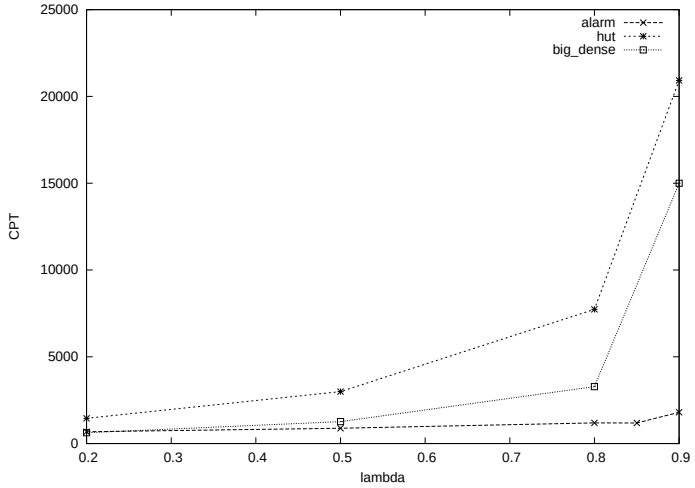


(b) BIC scored networks

Figure 7.2: In all networks junction tree size increases when learned with higher λ values. Junction tree size for BIC learned networks rapidly explodes with higher λ values.

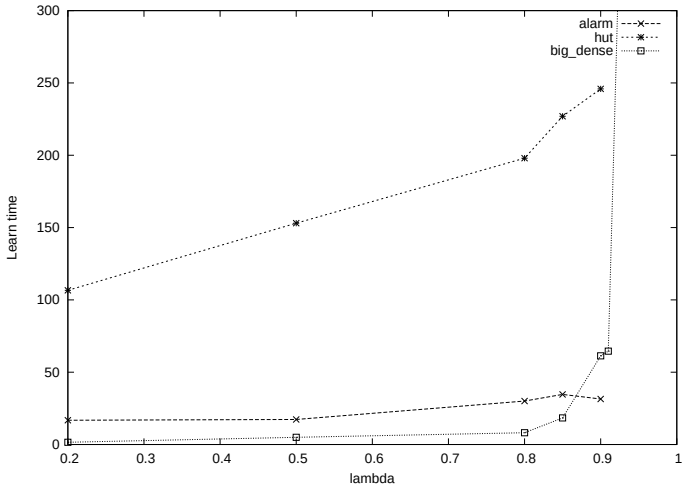


(a) JTC scored networks

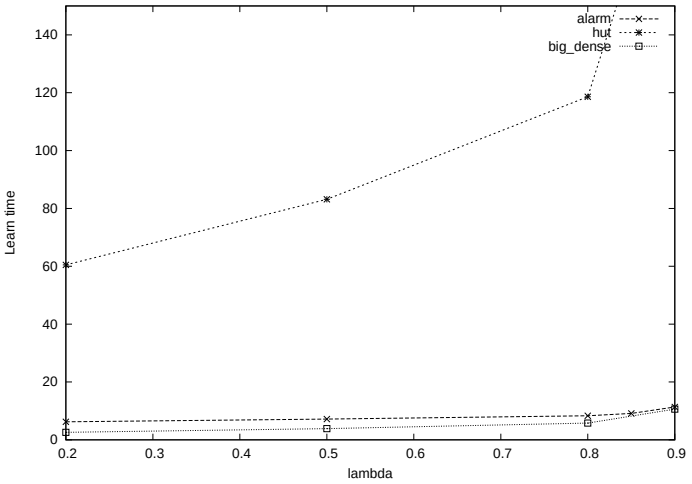


(b) BIC scored networks

Figure 7.3: Network size (CPT) increases when learned with higher λ value.

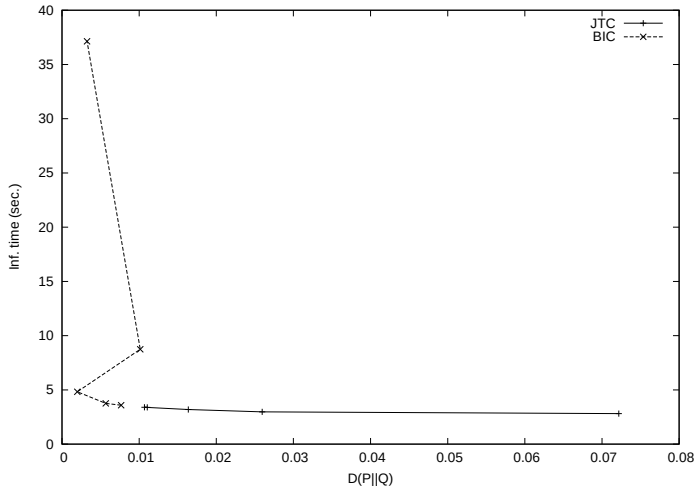


(a) JTC scored networks

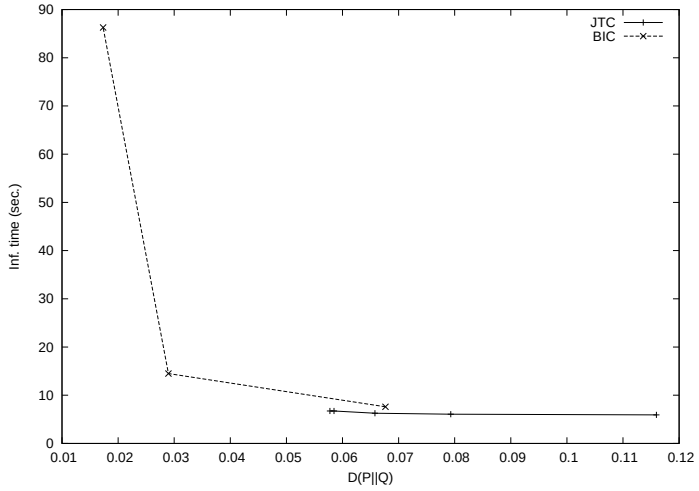


(b) BIC scored networks

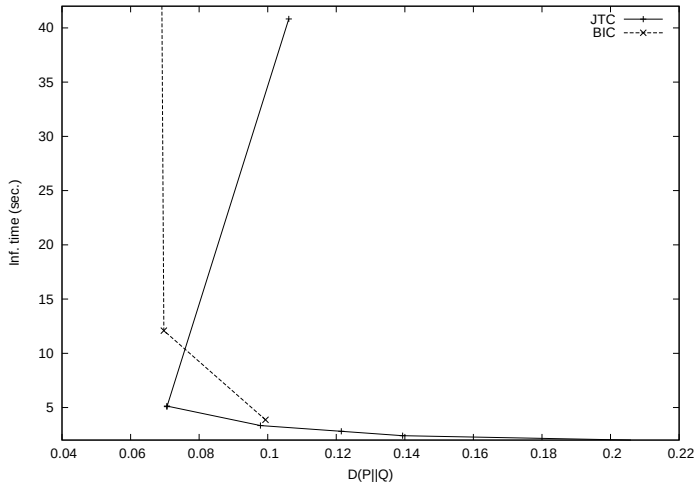
Figure 7.4: In all networks learning time increases when learned with higher λ values.



(a) alarm

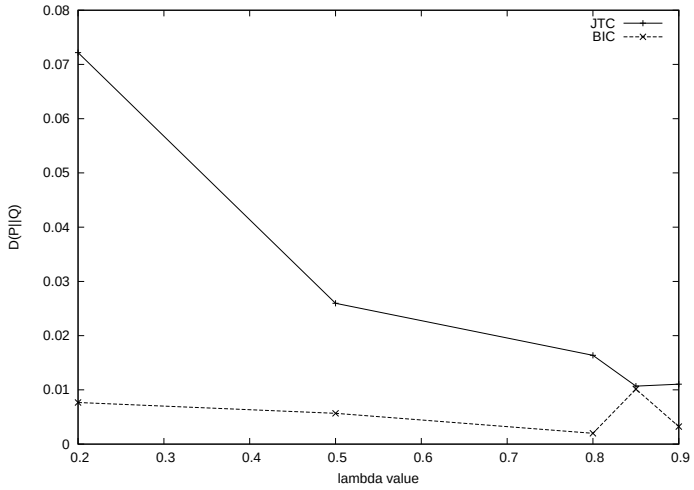


(b) hut

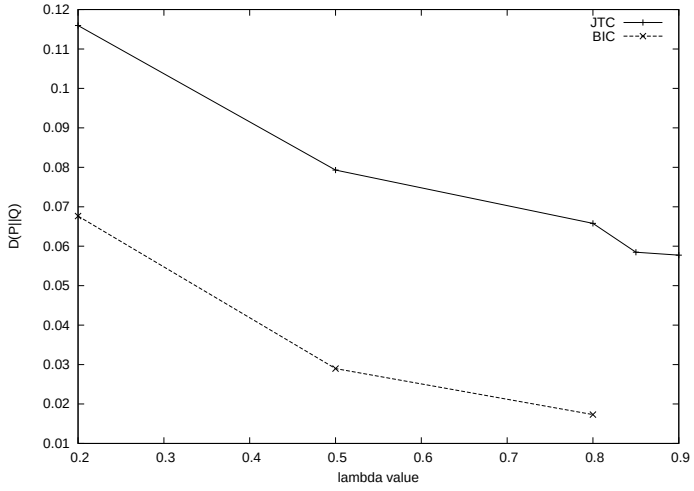


(c) big_dense

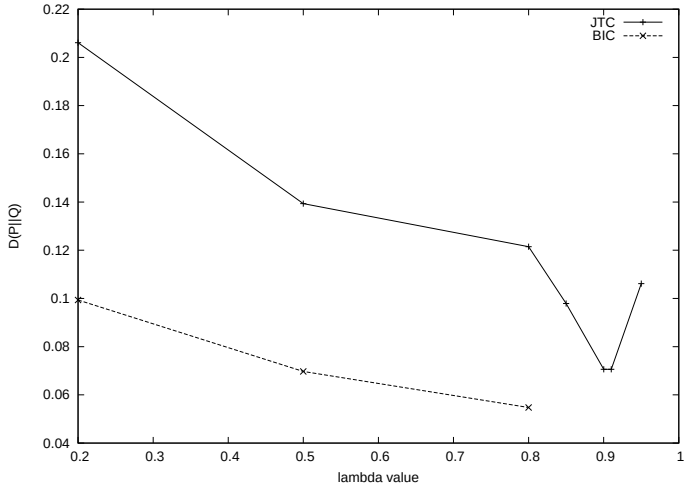
Figure 7.5: Inference time for BIC learned networks are much langer than inference time for JTC scored networks



(a) alarm



(b) hut



(c) big_dense

Figure 7.6: In most cases a higher λ value results in a better Kullback-Leibler divergence.

Chapter 8

Conclusion

In this project we have explored how control of the effective complexity of Bayesian networks learned from data can be exercised. This was done in the light of improving upon the learning method from our previous work [KP07], which exhibited poor learning times for large networks. We have examined the process of constructing junction trees from Bayesian networks and the process of performing inference in the network using these junction trees. From this evaluation the conclusion was drawn that a major contributor to the complexity of inference is the size of the cliques in the junction tree. The cliques are constructed from the triangulated domain graph for the network. We explored common triangulation heuristics to get an insight into how cliques-size can be limited. Then the score-based method of structural learning from data was explored. The BIC scoring function was found (by our previous work) to be inadequate in describing the computational complexity for the resulting network, even when augmented with a λ parameter that allows for explicit weighing of the log-likelihood and size estimation. One answer would be to apply a score function directly on the junction tree of the candidate network. In our previous work we presented a score function that simply triangulates each candidate to crate a junction tree and then measures the clique-size of that tree (together with the log-likelihood and weighed by a λ-parameter) to give the candidate a score. In this report we propose a similar score function denoted JTC scoring. Unfortunately this score function is not decomposable. Scoring a complete junction tree with a score function that is not decomposable is possible, yet impractical in that the learning becomes prohibitively slow. As the score function could not easily be made decomposable, another option was to try to speed up the triangulation of the network. One way which has proven to do just that is incremental compilation of the junction tree. This method relies on a third structure (the maximal prime subgraph decomposition) which has the property that the

individual clusters it is comprised of can be triangulated individually. This method fits well with structural learning as it is likely only a small part of the network are changed at each step in the structural search. The incremental compilation method was modified to better fit in the process of structural learning. The major difference is that the goal of the method is no longer to produce a junction tree, but simply to provide enough of a tree (namely the cliques) to allow for a scoring to take place, the rest of the process can be skipped until a candidate is selected.

A number of experiments were done with a number of networks and λ values. The results of these experiments shows that networks scored with JTC scoring are generally less complex in terms of junction tree size (triangulated using clique size heuristic) than junction trees for those networks scored with BIC, which in turn allows for faster inference in these networks. In some cases the networks learned with BIC scoring proved practically unusable. With regards to the balancing of complexity and accuracy, the results also indicate that some measure of complexity control can be exercised through the λ parameter. This supports the results from the preliminary work.

With regards to improving on the performance with regards to learning time reguired, speeding up JTC scoring using incremental compilation improves significantly upon this. This can be verified by inspecting the learning time for networks Hut and Big_dense (learned with LL-JT) from the preliminary results in Appendix A to the learning times for the same networks (learned with JTC) in Chapter 7.2 (note comparison cannot be made directly on the lambda values they are implemented in slightly different ways between the preliminary work and in this report). It is evident that the learning time for most of the Hut and Big_dense learned with JTC scoring has been significantly improved in comparison with the best learning times for Hut and Big_dense networks in the preliminary results.

Results indicate that structural learning using JTC seems to be computationally affordable, as the learning time is comparable to that of using BIC scoring (except for some of the higher λ values). This effect degrades for the higher λ values, but this is expected as the learning will almost be based purely on log-likelihood, which favor denser structures, resulting in larger parts of the MPD to be retriangulated for the scoring of each candidate. The precision (Kullback-Leibler and classification benchmark) of networks learned with BIC scoring is slightly better than for those learned with JTC scoring, but the JTC scored networks has a much better effective complexity and thus the degrade in precision seems acceptable.

References

[Abr94] Bruce Abramson. The design of belief network-based systems for price forecasting. *Comput. Electr. Eng.*, 20(2):163–180, 1994.

[AWFA87] Steen Andreassen, Marianne Woldbye, Björn Falck, and Stig K. Andersen. Munin - a causal probabilistic network for interpretation of electromyographic findings. In *IJCAI*, pages 366–372, 1987.

[BHP+92] J Breese, E Horvitz, M Peot, R Gay, and G Quentin. Automated decision-analytic diagnosis of thermal performance in gas turbines. In *In Proceedings of the International Gas Turbine and Aeroengine Congress and Exposition*, 1992.

[CL68] C. Chow and C. Liu. Approximating discrete probability distributions with dependence trees. *Information Theory, IEEE Transactions on*, 14(3):462–467, May 1968.

[Coo90] Gregory F. Cooper. The computational complexity of probabilistic inference using bayesian belief networks (research note). *Artif. Intell.*, 42(2-3):393–405, 1990.

[CS96] Peter Cheeseman and John Stutz. Bayesian classification (autoclass): Theory and results. In *Advances in Knowledge Discovery and Data Mining*, pages 153–180. 1996.

[CSG04] Rafael Cano, Carmen Sordo, and José M. Gutiérrez. Applications of bayesian networks in meteorology. *Advances in Bayesian Networks*, pages 309–327, 2004.

[DH92] BN Nathwani DE Heckerman, EJ Horvitz. Toward normative expert systems: Part i. the pathfinder project. In *Methods of inf. Med. 32*, pages 90–105, 1992.

[DLR77] A. P. Dempster, N. M. Laird, and D. B. Rubin. Maximum likelihood estimations from incomplete data via the em algorithm. *Journal of the Royal Statistical Society*, 39(1):1–38, 1977.

[DP97] Pedro Domingos and Michael J. Pazzani. On the optimality of the simple bayesian classifier under zero-one loss. *Machine Learning*, 29(2-3):103–130, 1997.

[ELFK00] Gal Elidan, Noam Lotner, Nir Friedman, and Daphne Koller. Discovering hidden variables: A structure-based approach. In *NIPS*, pages 479–485, 2000.

[Elv] Elvira. Elvira. `http://leo.ugr.es/elvira/`.

[FG] Nir Friedman and Moises Goldszmidt. Learning Bayesian networks with local structure. pages 252–262.

[FGG97] Nir Friedman, Dan Geiger, and Moises Goldszmidt. Bayesian network classifiers. *Machine Learning*, 29(2-3):131–163, 1997.

[FGO03] M. Julia Flores, José A. Gámez, and Kristian G. Olesen. Incremental compilation of bayesian networks. In *Proceedings of the 19th Annual Conference on Uncertainty in Artificial Intelligence (UAI-03)*, pages 233–240, San Francisco, CA, 2003. Morgan Kaufmann.

[FGO04] M. Julia Flores, José A. Gámez, and Kristian G. Olesen. Incremental compilation of Bayesian networks in practice. In *Proceedings of the 4th International Conference on Intelligent Systems Design and Applications (ISDA-2004)*, pages 843–848, Budapest (Hungría), 2004.

[FN91] DA Hodges F Nadi, AM Agogino. Use of influence diagrams and neural networks in modeling semiconductor manufacturing processes. In *IEEE Transactions on Semiconductor Manufacturing*, volume 4, pages 52–8, 1991.

[Fri98] Nir Friedman. The Bayesian structural EM algorithm. In *UAI*, pages 129–138, 1998.

[GPC+94] Yiqun Gu, D.R. Peiris, J.W. Crawford, J.W. NcNicol, B. Marshall, and R.A. Jefferies. An application of belief networks to future crop production. *Artificial Intelligence for Applications, 1994., Proceedings of the Tenth Conference on*, pages 305–309, Mar 1994.

[Gra03] Paul Graham. A plan for spam. `http://www.paulgraham.com/spam.html`, 2003.

[Hec95] D. Heckerman. A tutorial on learning with bayesian networks, 1995.

[Hec97] David Heckerman. Bayesian networks for data mining. *Data Mining and Knowledge Discovery*, 1(1):79–119, 1997.

[hug] Hugin expert a/s. `http://www.hugin.com`.

[ICR] Jaime S. Ide, Fabio G. Cozman, and Fábio T. Ramos. Generation of random bayesian networks with constraints on induced width.

[JN07] Finn V. Jensen and Thomas D. Nielsen. *Bayesian networks and Decision Graphs*. Springer, second edition, 2007.

[Kjæ93] U. Kjærulff. *Aspects of Efficiency Improvements in Bayesian Networks*. PhD thesis, Aalborg University, 1993.

[KL51] S. Kullback and R. A. Leibler. On information and sufficiency. *The Annals of Mathematical Statistics*, 22(1):79–86, 1951.

[KP07] M. Karlsen and S. Pedersen. Learning inference-friendly bayesian networks - initial analysis. Dat5 project from Aalborg University, 2007.

[LAB90] Tod S. Levitt, John Mark Agosta, and Thomas O. Binford. Model-based influence diagrams for machine vision. In *UAI '89: Proceedings of the Fifth Annual Conference on Uncertainty in Artificial Intelligence*, pages 371–388, Amsterdam, The Netherlands, The Netherlands, 1990. North-Holland Publishing Co.

[LB94] W. Lam and F. Bacchus. Learning bayesian belief networks: An approach based on the mdl principle, 1994.

[LS88] S. L. Lauritzen and D. J. Spiegelhalter. Local computations with probabilities on graphical structures and their application to expert systems. *Journal of the Royal Statistical Society*, 50:157–224, 1988.

[Nea03] Richard E. Neapolitan. *Learning Bayesian Networks*. Prentice Hall, April 2003.

[OM02] K.G. Olesen and A.L. Madsen. Maximal prime subgraph decomposition of bayesian networks. *Systems, Man, and Cybernetics, Part B, IEEE Transactions on*, 32(1):21–31, Feb 2002.

[POL07] Artem Parakhine, Tim O'Neill, and John Leaney. Application of bayesian networks to architectural optimisation. In *ECBS '07: Proceedings of the 14th Annual IEEE International Conference and Workshops on the Engineering of Computer-Based Systems*, pages 37–44, Washington, DC, USA, 2007. IEEE Computer Society.

[RTL76] Donald J. Rose, R. Endre Tarjan, and George S. Lueker. Algorithmic aspects of vertex elimination on graphs. *SIAM Journal on Computing*, 5(2):266–283, 1976.

[Sch78] Gideon Schwarz. Estimating the dimension of a model. *The Annals of Statistics*, 6(2):461–464, 1978.

[SGS01] Peter Spirtes, Clark Glymour, and Richard Scheines. *Causation, Prediction, and Search, Second Edition (Adaptive Computation and Machine Learning)*, chapter 5. The MIT Press, January 2001.

[SJ80] J. Shore and R. Johnson. Axiomatic derivation of the principle of maximum entropy and the principle of minimum cross-entropy. *Information Theory, IEEE Transactions on*, 26(1):26–37, Jan 1980.

[TK05] Marc Teyssier and Daphne Koller. Ordering-based search: A simple and effective algorithm for learning bayesian networks. In *UAI*, pages 548–549, 2005.

[var04] Värmforsk, 2003-04. http://vbn.aau.dk/research/surveillance_of_power_plants_varmforsk(16266).

Part IV

Appendix

Appendix A

Preliminary Results

This appendix goes through some of the testing results from the preliminary work in [KP07]. The tests consists of learning a number of networks using structural learning and subsequently submitting the networks to an inference benchmark. The training data as well as the test data was generated from a set of networks, which we will denote the "gold standard" networks, representing the models that we wished to approximate through structural learning.

Structural learning was conducted using a greedy search and two different score functions: The BIC score function, and a score function (denoted as LL-JT) based on a combination of log-likelihood and junction tree size (sum of state space for the cliques). Junction trees were constructed using the clique size heuristic. The score functions took as a parameter a λ-value for ballancing log-likelihood against model complexity.

The inference benchmark consisted of propagating the a set of test data with 40% MCAR for both the gold standard and the learned network. Then the difference between marginal probabilities between these two were measured.

Two gold standard networks produced the most interesting results, namely the Hut and the Big_dense network. The dimension of the Hut network is 8148, and the dimension of the Big_dense network is 5048. (these are the same networks used in the experiments for this project, described in Chapter 7). The preliminary tests were conducted using a number of different λ-values and the results are presented on the following pages. Table A.1 shows results from testing based on the Hut network, Table A.2 shows results based on the Big_dense network. The colums of the tables are:

- λ the lambda value used in the score function.
- **BIC** the BIC score of the resulting network given the training data.

- **LL** the log-likelihood of the resulting network given the training data.
- **JT size** junction tree size of the resulting network (measured as the sum of state space for the cliques).
- **CPT** dimension of the resulting network (measured as the sum of the size of each CPT)
- **Learn. Time (H:M:S)** the time it took to learn the networks (Hours:Minutes:Seconds).
- **Inf. time** the time it took to propagate the set of test data (measured in seconds)
- **Accuracy** differnce between marginals for the gold standard network and the learned network given the data set. The closer the value is to 0, the closer the marginals of the learned network comes to the marginals of the gold standard network. These numbers should be taken with a grain of salt as the one in Table A.1 is a summation of euclidian distance between marginals of the gold standard and learned for each test case, and the one in Table A.2 is a summation of an approximation of the Kullback-Leibler divergence between marginals of the gold standard and learned for each test case - despite this both measures has the property that the closer the value gets to 0, the closer the the learned network is to the gold standard network.

Based on the preliminary experiments a number of observations were made:

1. Networks with low inference time are generally concentrated about the low end of the λ values. This can be seen by inspecting the λ and *Inf. time* columns. This effect was somewhat expected, as high λ values for both BIC and LL-JT should favour densely connected networks, which might not be fast for inference. Dense networks generally also produce larger junction trees, which is why the Hut network learned with BIC scoring and $\lambda = 0.9$ does not participate in the inference benchmark, as the test computer ran out of memory when compiling the network (indicated by "unknown" in Table A.1).
2. Taking the junction tree size into account we can see that the inference time increase as the JT size increase, this is effect can be seen in Figure A.1.
3. Learning using LL-JT score can be several orders of magnitude slower than using BIC score. This can be seen by inspecting the *Learn.time* columns of BIC and LL-JT. Most learning runs for for LL-JT took at

least twice the time or more, as did most of the learning runs using BIC.

4. LL-JT seems to have a smaller accuracy difference for about the same inference time compared to BIC. This is evident in Figure A.6 where LL-JT scored networks are generally closer to the point (0; 0) than BIC scored networks. Thus it seems there is something to gain by choosing LL-JT as the score function with regards to resulting inference time.

5. Taking junction tree size into account has a deep impact on the learning time comparing it to the learning time when using BIC. But this behaviour is somewhat expected as the LL-JT score is far more complex than the BIC, as LL-JT involves performing a large number of triangulations.

6. The λ values for LL-JT seems to be a good indicator of the junction tree complexity in the learned network (that is, low values produce less complex junction trees than higher values), as opposed to the λ value in BIC where the junction tree complexity seems to fluctuate a lot more. This can be seen in Figure A.2.

7. It seems that as the network increase in complexity, the Log-Likelihood increases as well. This can be seen in Figure A.4.

8. The BIC-score of a network does not seem to be a good prediction of the resulting junction tree, Figure A.5.

Table A.1: Results of preliminary testing on Hut data sets.

λ	BIC	LL	JT size	CPT	Learn.Time (H:M:S)	Inf. time	Accuracy
LL-JT							
0.1	-356283,250	-349103,7905	2334	2135	00:33:35	7,214	2,837
0.15	-358792,766	-351364,627	2423	2192	00:35:03	7,156	2,987
0.2	-359094,104	-349105,4901	2948	2876	00:38:25	7,323	2,789
0.25	-362140,830	-345718,7933	4504	4948	00:50:19	8,534	2,658
0.3	-364434,399	-349605,7516	4210	4119	00:48:06	8,4	2,962
0.4	-364926,364	-343466,2716	6466	5854	01:00:41	9,587	2,448
0.5	-372233,362	-339743,8868	10596	9411	01:07:54	11,874	2,241
0.6	-379974,007	-344044,4693	13513	10421	01:08:27	13,321	2,717
0.7	-433340,178	-342176,2294	25510	25232	02:19:29	25,093	3,038
0.8	-552056,361	-341986,9186	70557	62034	04:55:45	51,445	3,474
0.9	-1107137,707	-343752,4614	218794	205514	14:17:38	167,441	4,211
BIC							
0.1	-357774,637	-353464,1983	1474	1233	00:07:21	6,567	3,100
0.15	-354254,706	-349317,9645	2900	1463	00:08:51	7,694	2,934
0.2	-358455,809	-353979,5842	2503	1296	00:07:56	7,334	3,284
0.25	-354104,917	-347772,8086	6089	1821	00:10:28	8,971	2,875
0.3	-354949,261	-348165,8457	5398	2008	00:12:09	9,265	2,816
0.4	-351266,764	-342899,1699	10284	2514	00:14:29	11,505	2,322
0.5	-355235,624	-345684,5012	8830	2740	00:15:26	10,418	2,652
0.6	-353607,269	-342780,514	14241	3297	00:18:20	13,032	2,425
0.7	-359811,846	-345121,3534	39278	4246	00:22:09	22,357	2,657
0.8	-354628,095	-335056,1224	3804706	5840	00:31:13	3419,098	1,936
0.9	-429882,043	unknown	unknown	30939	01:54:24	unknown	unknown

Table A.2: Results of preliminary testing on Big_dense.

λ	BIC	LL	JT size	CPT	Learn.Time (H:M:S)	Inf. time	Accuracy
LL-JT, *big_dense,net*							
0,1	-249714,3495	-246965,0629	839	841	0:0:52,078	2,810147794	0,099595098
0,2	-248390,3589	-244397,6763	1700	1180	0:0:59,609	3,584115142	0,082003285
0,3	-250487,5892	-243644,3063	2280	2103	0:01:33,859	4,189805606	0,079441753
0,4	-248685,7264	-242477,957	2456	1898	0:01:23,516	3,967183702	0,070612631
0,5	-249529,8191	-242635,8793	2592	2029	0:02:07,5	4,07486707	0,073128371
0,6	-270307,5795	-242022,6242	8032	8338	0:03:32,313	8,662986495	0,089077271
BIC, *big_dense,net*							
0,1	-253912,6414	-252793,5851	1109	359	0:0:14,375	2,923148461	0,137587847
0,2	-248736,0726	-246847,9529	2248	583	0:0:29,375	3,823394573	0,096405933
0,3	-247371,1798	-244870,5724	5628	773	0:0:35,047	5,385194663	0,082074183
0,4	-247316,7123	-244742,4221	11104	799	0:0:36,453	8,533886394	0,08141089
0,5	-244999,3888	-239063,3244	144336	1858	0:01:10,969	63,27850802	0,044747685
0,6	-247240,3474	-242948,3288	35672	1384	0:0:52,140	25,75348437	0,070742416

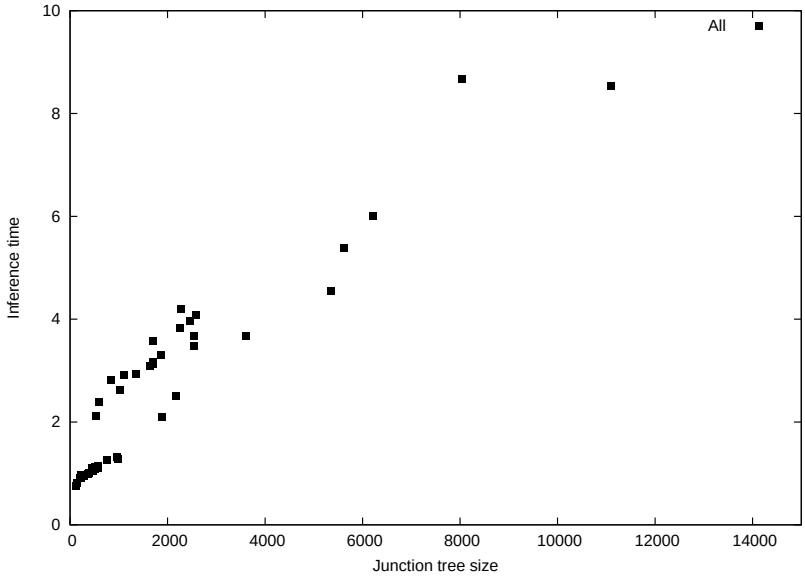


Figure A.1: There seems to be a dependence between inference time and junction tree size. (plot includes data from networks learned with both BIC and LL-JT for both Hut, Big_dense and some additional networks).

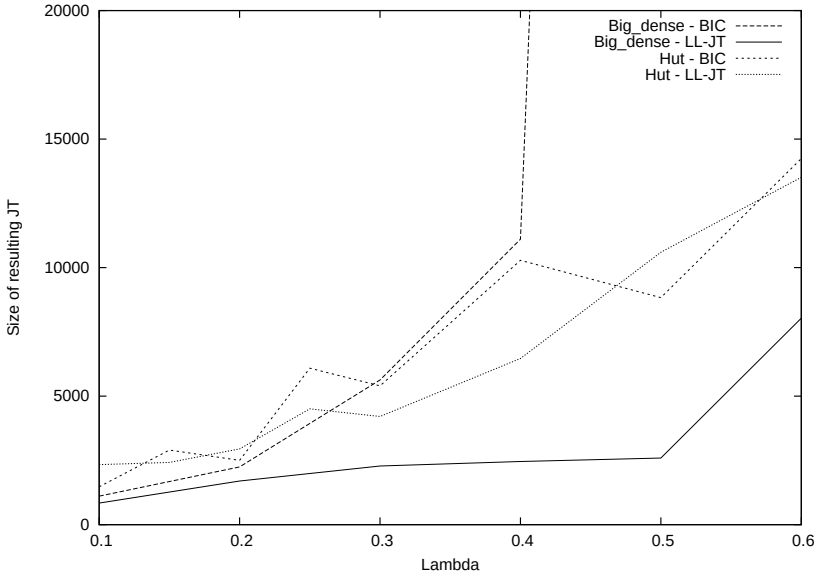


Figure A.2: Junction tree size for Hut and Big_dense networks using both LL-JT and BIC scoring with varying λ values

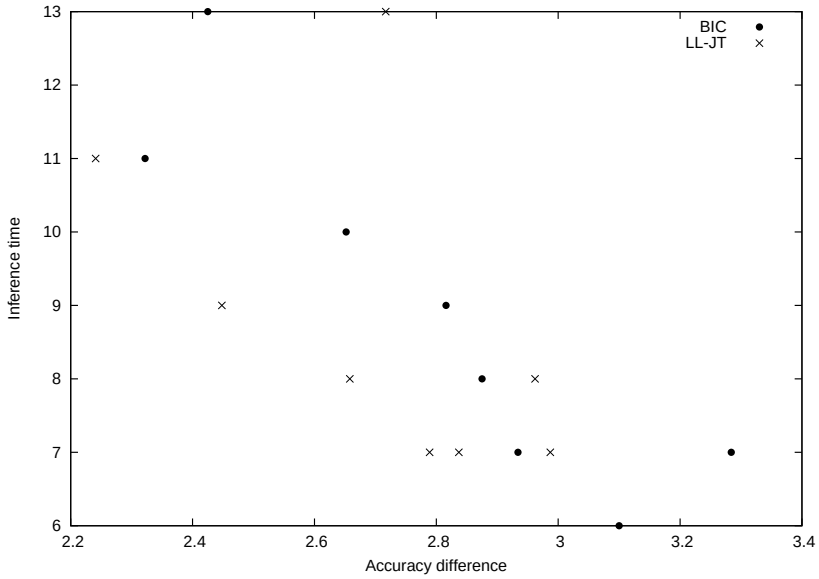


Figure A.3: Results from inference benchmark for Hut.

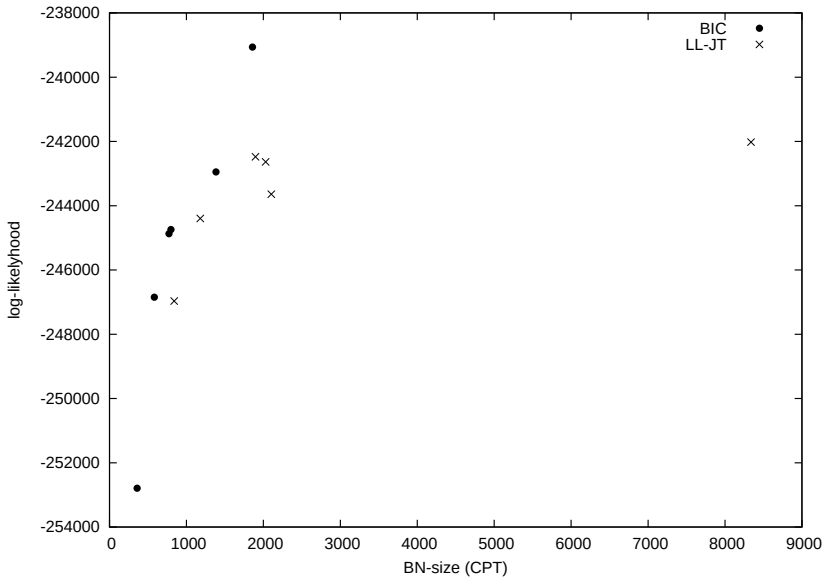


Figure A.4: Plot of Log-Likelihood over the network size (sum of CPT sizes) for networks learned from Hut. *Plot includes data from networks learned with both BIC and LL-JT in* big_dense.net.

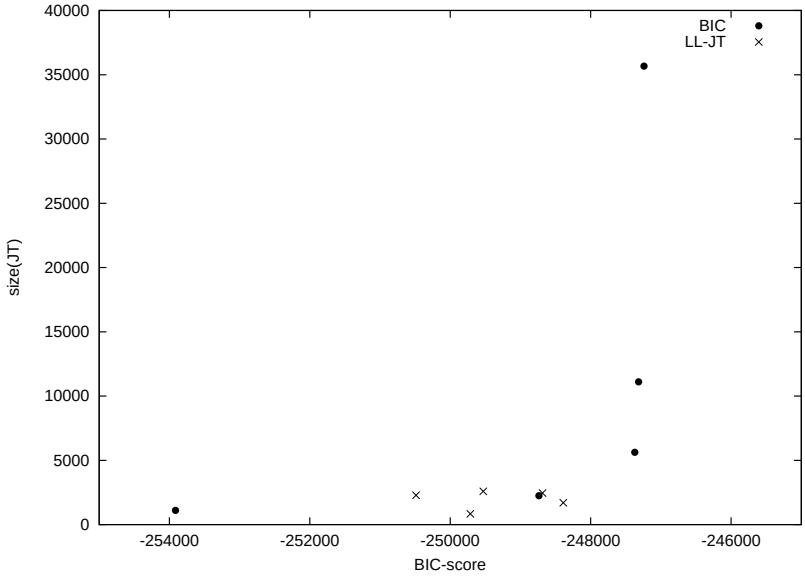


Figure A.5: It seems that the BIC-score of a network cannot predict the size of the resulting junction tree. *(Plot includes data from networks learned with BIC and LL-JT for Big_dense).*

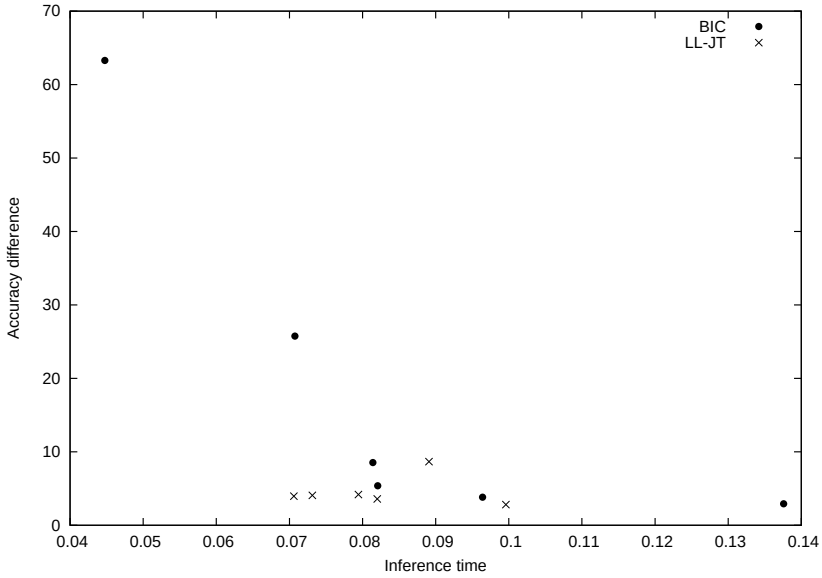


Figure A.6: Kullback-Leibler divergence between learned network and the generative network, compared with inference time in the learned network (results from learned networks with BIC and LL-JT for *big_dense.net*.)

Appendix B

Constraint-Based Learning

The constraint-based learning method *(described in [Nea03] ch. 10, [JN07] ch. 7.1)* creates the structure of the graph so that it models the independencies in the database. The process can be divided into two phases:

1. Build a graph skeleton by applying independence tests to the observations

2. Apply a number of rules to direct the edges of the skeleton to produce a directed acyclic graph (DAG)

B.1 Independence tests

Independence tests are used for determining dependencies between variables. A test on two variables A and B will evaluate to *true* if the variables are independent. The notation for independence tests has the form $I(A, B)$ when testing on two variables, or $I(A, B|\mathcal{X})$ when conditioning on variables $\mathcal{X}$.

Dependencies are determined by analyzing the set of observations used for learning. This is done by calculating the conditional mutual information and use the result in an independence test:

$$CMI(A, B|\mathcal{X}) = \sum_{\mathcal{X}} P^{\#}(\mathcal{X}) \sum_{A,B} P^{\#}(A, B|\mathcal{X}) \log_2 \frac{P^{\#}(A, B|\mathcal{X})}{P^{\#}(A|\mathcal{X})P^{\#}(B|\mathcal{X})}$$

Where $P^{\#}(v)$ means the relative count of how many times the configuration $V = v$ is observed in the data.

The calculated value is then used in a statistical test on the hypothesis $I(A, B|\mathcal{X}) = true$, called the *null hypothesis*. The statistical test is based on a χ^2 distribution, and use the notion of *degrees of freedom*. In the statistical

test, degrees of freedom can be thought of as the number of different outcomes of an event, which corresponds to the number of parameters needed for determining $P(A)P(B)P(\mathcal{X})$.

The degrees of freedom in the test is thus defined as

$$(s_A - 1)(s_B - 1)\prod_{\mathcal{X}}(s_X - 1)$$

where S_Z is the number of states in variable Z.

For the statistical test we decide on a significance level. The significance level represents the maximum acceptable level of uncertainty about the null hypothesis. With a significance level of 0.05 we will accept a 0.5% probability that the null hypothesis is false. Thus if the χ^2 test determines that the probability of the null hypothesis being false is greater than 0.5%, we reject the null hypothesis and accept the alternative hypothesis, namely that A and B is not independent given $\mathcal{X}$.

In practice the value for $I(A, B|\mathcal{X})$ is calculated and compared against a *critical value*. The critical value represents the maximum acceptable CMI, given the specific degrees of freedom, such that the probability of the null hypothesis being false is less than or equal to the significance level.

The critical value for different significance levels can be looked up in precalculated tables like the one shown in Table B.1. Common significance levels are $0,01$ and $0,05$ and the higher the level is set, the fewer independencies are found.

Performing the independence tests is basically about counting observations, thus the cost of performing an independence test linear in the number of observations in the database.

B.2 Building graph skeleton

The skeleton for a Bayesian network is an undirected graph over the same set of variables and edges. The first part of the learning process is to construct a graph skeleton, which ultimately can be transformed into a Bayesian network structure by directing the edges.

We start with a fully connected undirected graph. By performing independence tests on the observations, edges between the nodes may iteratively be removed in accordance to their independence properties. Thus a edge may remain between two nodes A and B if for all X: $\neg I(A, B|X)$. This is because two connected nodes cannot be d-separated, and under the assumption that the database is faithful the nodes can only be independent if they are d-separated.

DF	0,01	0,05
1	3,84	6,64
2	5,99	9,21
3	7,82	11,35
4	9,49	13,28
5	11,07	15,09
6	12,59	16,81
7	14,07	18,48
8	15,51	20,09
9	16,92	21,67
10	18,31	23,21
11	19,68	24,73
12	21,03	26,22
13	22,36	27,69
14	23,69	29,14
15	25,00	30,58

Table B.1: Critical values for the χ^2 Distribution for different significance levels (Degrees of Freedom [DF] in the first column)

Explicitly enumerating all independencies, by making a full run through the database for each hypothesis, may be too expensive. Thus edges should be removed from the graph skeleton by using as few tests as possible. The number of tests can be reduced by performing test $I(A, B|X)$ only when $X \in neighbours(A) \cap neighbours(B)$. This is the case since an edge between A and B, in the "real" Bayesian network, cannot exist if they are d-separated given their parent nodes. Since the skeleton is undirected we do not know which part of each nodes neighbourhood is its parents or children, so we have to consider the entire neighbourhood.

The PC-algorithm converts the complete undirected graph into the graph skeleton by removing edges in an efficient way, by restricting independence queries to node neighbourhoods.

1. Start with the complete graph
2. $i := 0$
3. **while** a node has at least $i + 1$ neighbours

   for all nodes A with at least $i + 1$ neighbours

   for all neighbours B of A

for all neighbour sets χ such that $|\chi| = i$ and $\mathcal{X} \subseteq (\text{nb(A)}\backslash\{\text{B}\})$
if $I(A, B, \mathcal{X})$ **then** remove the link $A-B$ and store $I(A, B, \mathcal{X})$
$i := i + 1$

The worst-case complexity of the PC-algorithm is $O(\frac{n^2(n-1)^{k-1}}{(k-1)!})$ where k is the maximum degree for any node and n is the number of nodes. According to [SGS01] the worst-case scenario is very rare, although they present no expected average complexity analysis.

B.3 Produce DAG by directing edges

The edges of the graph skeleton may be directed using the following four rules until all edges have been directed. A v-structure in a DAG G is an ordered triple of nodes (X, Y, Z) such that G contains the edges $X \rightarrow Y$ and $Z \rightarrow Y$, and X and Z are not adjacent in G.

Introduction of V-structures For any connections $A-B-C$, direct edges $A \rightarrow B \leftarrow C$ if for all X where $B \notin X$: $I(A, C|X)$ or $I(A, C)$.

Avoid further V-structures When no further V-structures can be introduced based on independence tests, if possible direct any connections $A \rightarrow B - C$ such that V-structures are avoided i.e. $A \rightarrow B \rightarrow C$.

Avoid Cycles If directing any of the edges introduces a directed cycle in the graph, reverse the edge.

Random When all rules have been exhausted and undirected edges still exists, chose a random direction for the remaining edges without violating the cycle condition.

This method may produce different graphs (on account on some of the rules chooses randomly), but all generated DAGs have the same d-separation properties.

Appendix C

Incremental compilation algorithms

This appendix is a short presentation of the incremental compilation algorithms found in [FGO03], as well as our comments.

ConstructMPDTree (JoinTree $\mathcal{T}$, Graph G^M)

1. Aggregate all adjacent cliques in $\mathcal{T}$ with an incomplete separator in G^M to obtain $\mathcal{T}_{MPD}$
2. Return $\mathcal{T}_{MPD}$

We assume that this is the function referred to as AggregateCliques in the IncrementalCompilation algorithm of the paper. The maximal prime subgraph decomposition tree must be a single structure, so it seems that the paper does not consider the case when the junction tree is a junction forest (we have added a step that transforms a forest to a single structure in the version we use).

IncrementalCompilation (Modification list *ModList*)

1. For each modification *mod* in *ModList* do
   (a) L ← ModifyMoralGraph(*mod*)
   (b) Case *mod* of
      i. Add node X: AddNode(X)
      ii. Delete node X: RemoveNode(X)
      iii. Delete link $X \rightarrow Y$: MarkAffectedMPSsByRemoveLink(M_Y, nil, L)

iv. Add link $X \rightarrow Y$:
MarkAffectedMPSsByAddLink(L)

2. For each connected marked subtree, T_{MPD}, of $\mathcal{T}_{MPD}$ do

   (a) Mark all cliques in subtree T of $\mathcal{T}$ corresponding to T_{MPD}

   (b) Let C be any cluster of T and let M be any cluster of T_{MPD}

   (c) $V \leftarrow \{$all variables included in $T_{MPD}\}$

   (d) $g^M \leftarrow G^M(V)$

   (e) $t \leftarrow$ ConstructJoinTree(g^M)

   (f) $t_{MPD} \leftarrow$ConstructMPDTree(t, g^M)

   (g) $\mathcal{T} \leftarrow$ Connect(t, C, nil)

   (h) $\mathcal{T}_{MPD} \leftarrow$ Connect($t_{,MPD}, M, nil$)

   (i) Delete T and T_{MPD}

The algorithm as presented in the paper contains a few typos and minor mistakes, which has been corrected in the above version. Our algorithm differs from the one presented in the paper by omitting the additional marking of clusters in junction tree and maximal prime subgraph decomposition tree, as we use our own REPLACE algorithm instead of the CONNECT algorithm from the paper. All of the algorithms mentioned will be presented below.

ModifyMoralGraph(Modification *mod***)**

1. $L \leftarrow \emptyset$

2. case *mod* of

   - Add node X: add a new (isolated) node X to G^M
   - Delete node X: remove X from G^M
   - Add link $X \rightarrow Y$: add $X \rightarrow Y$ to L together with all new links needed to make $Y \cup parents(Y)$ a complete sub-graph
   - Delete link $X \rightarrow Y$:
     (a) if $(children(X) \cap children(Y) = \emptyset)$ then
         – delete (X, Y) from G^M
         – add (X, Y) to L
     (b) for all $Z_i \in parents(Y) \setminus \{X\}$ do

   i. if $((children(Z_i) \cap children(X) = \{Y\})$ and $Z_i \rightarrow X$ in G and $X \rightarrow Z_i$ not in $G)$ then
      – delete (X, Z_i) from G^M
      – add (X, Z_i) to L

3. Return L

We have made these observations about *ModifyMoralGraph* in addition to *MarkAffectedMPSsByAddLink*:

- Should the links in the link list have orientation or be undirected?

  In the case of **Add link** $X \rightarrow Y$, the link $X \rightarrow Y$ is added to the link list. In addition any links needed to make $Y \cup pa(Y)$ a complete subgraph is also added, but these links must naturally be without orientation, as this must refer to the process of moralization of Y's parents. However, careful reading suggests that these undirected edges should only be added to the moral graph, not the link list

- should it say "$Z_i \rightarrow X$ **not** in G" instead of "$Z_i \rightarrow X$ in G" in the *delete link* part?

  As we have interpreted the purpose of line *(b)*, it is to: delete moral edges between X and any $Z \in pa(Y)$, where Y is the only common child between X and Z, and there does not exist a link between X and Z in the BN DAG.

  Therefore we suspect that the line should have said "$Z_i \rightarrow X$ **not** in G" instead of "$Z_i \rightarrow X$ in G", as this would be part of the condition that no links existed between the two nodes in the BN DAG in order to delete the moral edge.

In our version we have omitted the parts that deals with added or deleted nodes, since the structural search only involves adding and deleting edges. Based on our observations we have rewritten the add link part such that it is clear which edges are added to the list. The delete link part of the algorithm has been re-written as well, to clarify what should happen.

MarkAffectedMPSsByRemoveLink(MPS M_Y, MPS M_Z, LinkList L)

1. Mark M_Y

2. For all neighbours $M_K \neq M_Z$ of M_Y do

   (a) $S_{YK} \leftarrow$ separator between M_Y and M_K

   (b) if $L \cap links(S_{YK}) \neq \emptyset$ then
   MarkAffectedMPSsByRemoveLink(M_K, M_Y, L)

The way to identify separators that will become incomplete after modification is to find separators that includes both nodes of a link in the list of deleted edges. The version presented in the paper uses an undefined operator *links* when performing this check. We have replaced this operator with the procedure just described.

MarkAffectedMPSsByAddLink(LinkList L)

For each Link $X \rightarrow Y$ in L do

1. Let M_X be the nearest neighbour to M_Y containing X
2. If there is an empty separator S on the path between M_X and M_Y then
   (a) Disconnect $\mathcal{T}_{MPD}$ and delete S
   (b) Connect M_X to M_Y by an (artificial) separator containing X
3. Mark M_X, M_Y and all M_Z on the path between them

At some point in the paper, the notation of M_X is introduced as "*[...] M_X will identify the MPS in $\mathcal{T}_{MPD}$ which has the family of X associated.*" Where we assume that the family of X refers to the set of nodes $X \cup pa(X)$ in the Bayesian network.

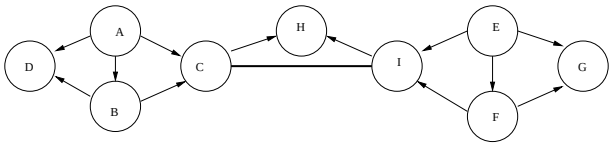
We have made these observations:

- Is M_X a MPS containing $X \cup pa(X)$, or is it the MPS that contains the node X and is closest to M_Y in the MPD tree?

  It seems like the notation has been overloaded, as we are very certain that M_Y should be interpreted as an MPS that contains $Y \cup pa(Y)$. However, the elvira source code documentation seems to indicate that M_X is just a cluster containing X, and it is the closest such cluster to M_Y

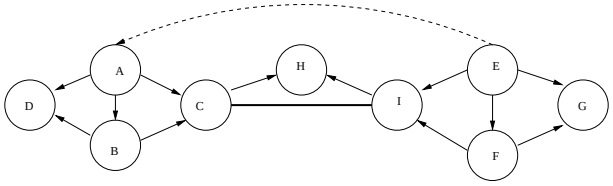
- Which M_Y should be chosen if $Y \cup pa(Y)$ is contained in several MPSs?

  The problem is illustrated with the Bayesian network and MPD tree in Figure C.1. In which case the link $A \rightarrow E$ is added and M_E can be any of the two MPSs IEF and EFG as both contains $E \cup pa(E)$. **???**



(a) DAG (Graph is also triangulated. Bold line is moral edge.)

(b) Corresponding MPD tree (also junction tree)



(c) Link $E \rightarrow A$ added

Figure C.1: Which of the MPSs should be marked by MarkAffectedMPSsByAddLink ABD, ABC, IEF and EFG?

- Since the link list returned by ModifyMoralGraph, when dealing with added links, contains only a single oriented edge, the *for each* statement seems unnecessary.

The version presented in the paper is ambiguous with regards to which MPSs should be marked as the endpoints of the path constituting the connected subtree. Also it seems that M_X should be the MPS which has the family of X associated, but it is not always the case that the family of a node is associated with the only one MPS. It is also worth mentioning that the list L contains a single directed link, which is ensured by ModifyMoralGraph.

Connect(Cluster tree t, Cluster C_i, Cluster C_J)

For each separator S between C_i and $C_K \neq C_j$ do
If C_k is unmarked then

1. locate cluster $C \in t$ such that $C \cap C_K$ is maximal
2. Connect C with C_K by S
3. If $S = C$ then amalgamate C and C_K

else Connect(t,C_K,C_i)