University of Castilla-La Mancha

A publication of the

Department of Computer Science

Incremental compilation of a Bayesian network

by

María Julia Flores Gallego

Technical Report **#DIAB-01-04-15** February 2001

DEPARTAMENTO DE INFORMÁTICA
ESCUELA POLITÉCNICA SUPERIOR
UNIVERSIDAD DE CASTILLA-LA MANCHA
Campus Universitario s/n
Albacete – 02071 – Spain
Phone +34.967.599200, Fax +34.967.599224

Abstract

When Bayesian networks are modified, for example by adding or deleting edges or nodes, or by changing probability tables, a recompilation of the model is usually required even though a partial (re)compilation could be sufficient.

Especially when considering dynamical models, where variables are frequently added and deleted, such recompilations use many resources, but also common model building, which is most often an iterative process, suffers from this lack of flexibility.

The project tries to investigate and implement methods for addition and deletion of nodes and edges and changes in potentials and develop methods for partial compilation in these situations.

CONTENTS

1. Introduction 5

2. Compilation of a Bayesian network 7
   2.1. A brief introduction to Bayesian networks 7
   2.2. What compilation of a Bayesian network is 8
   2.3. Process of compilation 9

3. Possible modifications in a Bayesian network 23
   3.1. Why we should look into it 23
   3.2. Systematic search of possible changes 23
   3.3. Potentials 23
   3.4. Graph 25
      3.4.1. Variables 25
         3.4.1.1. States 25
         3.4.1.2. Deletion 28
         3.4.1.3. Addition 43
      3.4.2. Edges 44
         3.4.2.1. Addition 44
         3.4.2.2. Deletion 52
   3.5. Discussion 58

4. Use of Maximal Prime Subgraph Decomposition in Incremental Compilation of Bayesian networks 59
   4.1. Purpose of using this method 59
   4.2. Presentation of Maximal Prime Subgraph Decomposition 60
   4.3. Procedure 62
   4.4. When and how we can apply it in incremental compilation 68
   4.5. Discussion 79

5. Implementation 81

5.1. r-Hugin tool 81
5.2. Programming with Visual C++ 81
5.3. Implemented examples 82
5.4. Discussion 83

6. Conclusion 85

Bibliography 87

A. r-Hugin details 89
A.1. Introduction to r-Hugin for Visual C++ 89
A.2. Class hierarchy 89
A.3. Program examples 91
A.3.1. *Compilation.cpp* 91
A.3.2. Changing potentials in A 96
A.3.3. Deleting X 106

Chapter 1. INTRODUCTION

A Bayesian network is a graphical model that we use to represent knowledge of a domain and the casual interaction between different variables in this domain. Once the Bayesian network for a problem has been modelled, a junction tree is created as a computational structure for reasoning about the domain.

This process is called compilation. Compilation consumes a significant amount of time. As we are going to see, it includes several steps: moralisation, triangulation and junction tree construction. If we are working with a big network the compilation time becomes really important. Then, the question that rises is the following one: if we do some modifications in a Bayesian network, is it necessary to recompile all of it again? We can guess that if this modification is not too comprehensive within the global network, it is very probable that we can find a way to save time.

We have an initial Bayesian network BN and its corresponding junction tree JT. But now we modify BN to the new one BN', for that JT will not be a valid junction tree, but another one we call JT'. The key is to find a faster way than a full recompilation of BN' to arrive from the new Bayesian network BN' to the new junction tree JT'. Therefore, we are trying to observe the differences between the two networks and find a method to reach JT' avoiding a new compilation (See *Figure 1-1*).

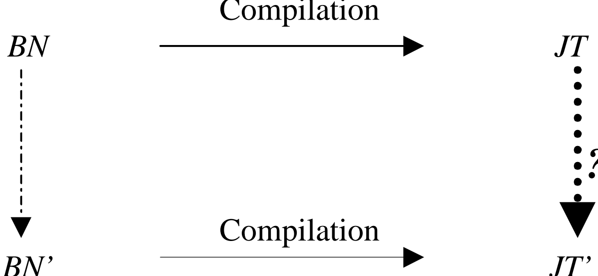


Figure 1-1. Drawing explaining the process of incremental compilation of a BN.

The purpose of this project is to analyse the modifications in a Bayesian network in order to answer to the question mark of *Figure 1-1*. For the development of this idea we have divided the work in a set of chapters.

The first one is about *Compilation of a Bayesian network*. To elaborate this chapter, the main reference has been [Jensen 1996]. Here we try to describe in a simple way how the compilation takes place. We find it important to undertake this point because the whole repetition of this process is exactly the one we want to avoid. So, we need to have some knowledge on how this process is carried out.

Afterwards, we start examining the *Possible modifications in a Bayesian network*. To accomplish this task we would use the method "learning by examples", since it is a good way both to study the subject and to explain it. As we will see, this analysis is not thoroughly accomplished, but we are going to look over most of the possibilities that include a reduced set of modifications. We touch upon most of the key questions and we present solutions to some of them.

For those cases where this analysis does not give a clear solution, we think about applying a new idea described in [Olesen and Madsen 1999] and we present the *Use of Maximal Prime Subgraph Decomposition in incremental compilation of Bayesian networks*. In this chapter we will try to put in practice this technique of decomposition to save time in recompilation.

In the next chapter we will implement some of the examples described in the previous chapters in order to illustrate the viability of using these ideas.

Finally we summarise and conclude that it is possible to make a partial compilation of a Bayesian network in a way that we can save quite a lot of computational time and work. We present some ideas and solutions, and we suggest future and deeper studies about this subject because we trust that they will lead to more attractive solutions and applications.

Chapter 2. COMPILATION OF A BAYESIAN NETWORK

1. A brief introduction to Bayesian networks.

A Bayesian network is a Directed Acyclic Graph (DAG) with some special characteristics. It lets us represent a domain, showing relationships between nodes, normally casual relationships. This is a graphical representation which will help us modelling the given domain.

To give a more formal definition [Jensen 1996], A *Bayesian network* consists of the following:

A set of variables and a set of directed edges between variables.

Each variable has a finite set of mutually exclusive states.

The variables together with the directed edges form a directed acyclic graph, i.e. a graph with no directed cycles.

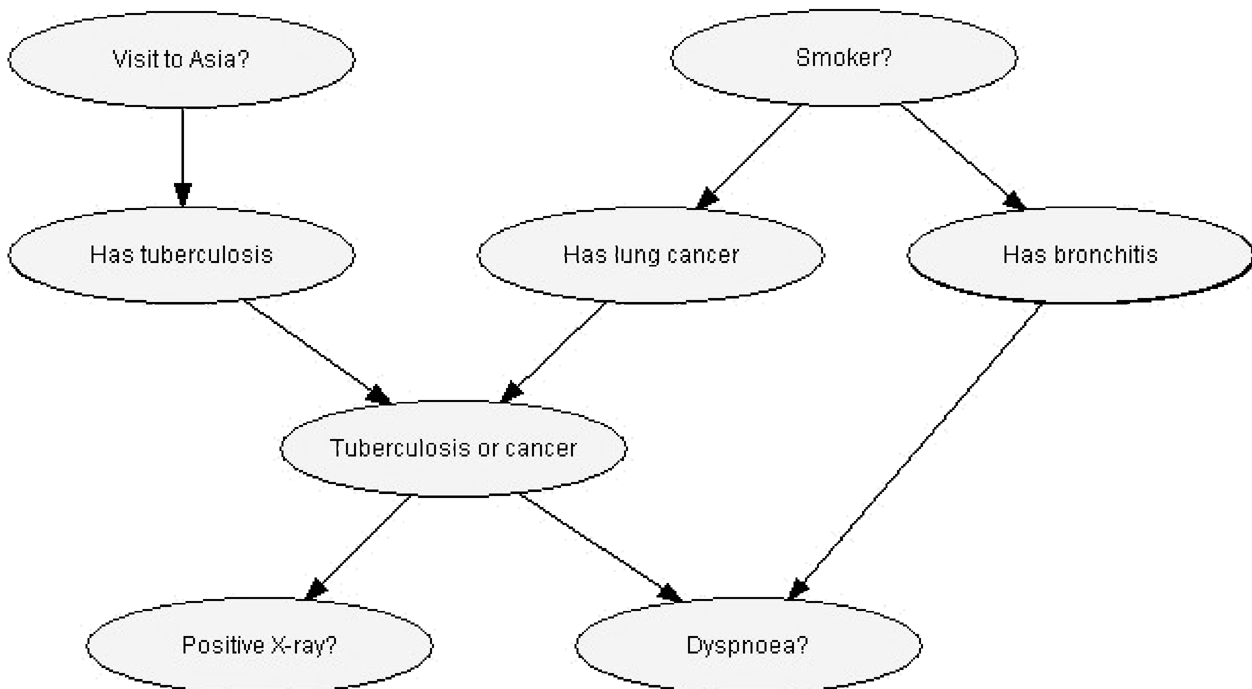


Figure 2- 1-. Example of a Bayesian network frequently used in the literature and called “Asia”. This network presents eight variables: “Visit to Asia?”, “Smoker?”, “Has tuberculosis”, “Has lung cancer”, “Has bronchitis”, “Tuberculosis or cancer”, “Positive X-ray?” and “Dyspnoea?”. All of them have escatly two possible states: yes (if it is true) or no (otherwise). The edges and their directions are visible in the network.

To each variable A with parents $B_1, \ldots, B_n$ there is attached a conditional probability table $P(A|B_1, \ldots, B_n)$.

In *Figure 2-1* we can see an example of a Bayesian network.

It can be demonstrated by the theorem called "the chain rule for Bayesian networks" that given a Bayesian network BN over the universe $U = \{A_1, \ldots, A_n\}$, the joint probability P(U) is the product of all conditional probabilities specified in the BN.

$$P(U)=\prod_i P(A_i| pa(A_i))$$

where $pa(A_i)$ is the parent set of A_i.

Without entering further into this subject we will only introduce that inside a Bayesian network there are these three aspects:

- Factored joint probability distribution as a directed graph:
- It is a structure for representing knowledge about uncertain variables.
- It is used as the basis for a computational architecture for calculating the impact of evidence on beliefs.

- Knowledge structure:
- Variables are depicted as nodes.
- Arcs represent direct probabilistic dependence between variables.
- Conditional probabilities encode the strength of the dependencies.

- Computational architecture:
- We can compute posterior probabilities given evidence about selected nodes.
- The goal is to exploit probabilistic independence for efficient local computation.

2. What the compilation of a Bayesian network is.

After this superficial introduction to Bayesian networks, it is the point to locate the role of compilation in them. As we have seen, initially in a Bayesian network we find nodes, which represent variables of a domain U, edges which indicate relations between them and probability tables. These will be the prior probabilities, P(X), for variables without parents and the conditional probabilities P(X|pa(X)) for variables with parents. So, these are our initial data. Now, we want to "translate" this information into a structure in which way we can compute marginal beliefs easily. We want a structure capable of giving numerical results about variables states after entering any kind of evidence. Compilation is the process followed to build this

structure. This is a systematic task, where in a first approach we find a deterministic number of steps that we will describe below.

3. Process of compilation.

Basically the compilation of a Bayesian network includes two parts:

① Junction tree construction

② Propagation of potentials along the tree

Below we present the initial situation (the network) and a detailed description of these two steps:

Let be a Bayesian network BN = {G, *P*}, where

- G is a directed acyclic graph, G = (V, E)

  V = set of vertices or nodes in the graph.

  E = set of edges which connect these nodes, and

- *P* is the set of probability tables required for this network ($P(A_i \mid pa(A_i))$).

1st Step. JUNCTION TREE CONSTRUCTION

Step 1.1.- Moralise the graph G.

In this step we must build the moral graph G^M, an undirected graph which will be based on G. To construct G^M, there are two actions. First, connect all nodes which have an edge pointing to the same node, i.e. those which have a child in common. From this terminology each of these new edges are also called *marriages*. We can call them moral edges as well. And afterwards keep the edges but dropping their directions.

Hence, if we had the graph G =(V,E), after moralising it we obtain G^M = (V,E^M), that is, it has exactly the same nodes but another set of edges E^M = 'Marriage edges' ∪ 'E without directions'.

Following the example of *Figure 2-1*, G^M would be as shown in *Figure 2-2.*

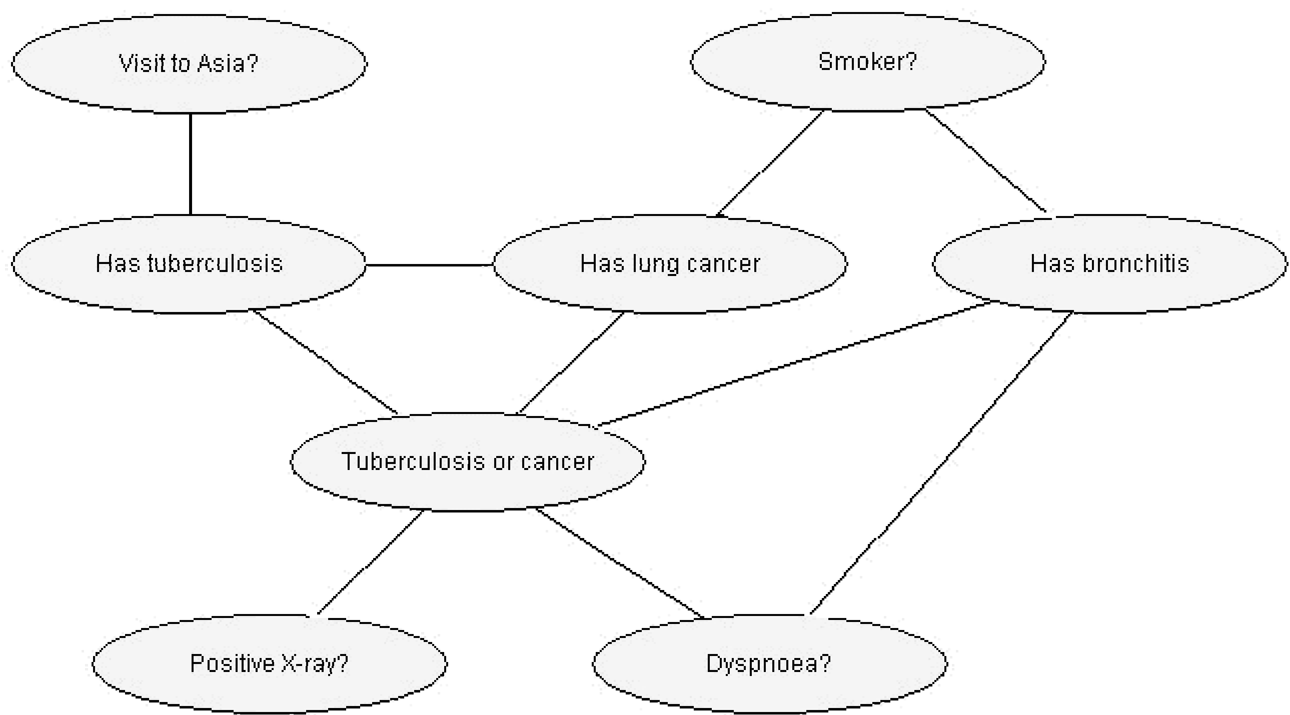


Figure 2- 2. Moral Graph for the one in *Figure 2-1.* Marriages between Has tuberculosis - Has lung cancer 0and Tuberculosis or cancer – Has bronchitis.

Step 1.2.- Triangulate G^M.

Now it is time for triangulation. The moral graph G^M is triangulated if every cycle of length greater than 3 has a chord. To triangulate the graph we add the so-called *fill-in edges* in order to satisfy this condition. One way to achieve a proper triangulation is to decide a sequence order to eliminate nodes. This order will give us the edges to add. This sequence order is called elimination ordering or deletion sequence, it is usually denoted by σ. If consist of a function which relates every node in the graph with a unique number between 1 and n, where n = |V|, that is, the number of nodes in the graph.

Finding an optimal ordering means giving an optimal triangulation, and this is a NP-hard problem. Thus, for solving it, heuristic methods are used. Some examples of them in the literature could be *minimum fill*, *minimum size* or *minimum weight* [Kjærluff 1993]. They are based on choosing one node first and after it, taking those whose elimination involve less fill-ins, less clique size or less clique weight.

Once we have decided about the heuristic method to use, and we have obtained a deletion sequence, then the triangulation is as follows:

Let be $\sigma = \{v_1, v_2, \ldots, v_n\}$

For i $\leftarrow$1 until i = n do

- Remove v_i from the graph and all his incident links.
- Add links between all his neighbours, if they did not exist before.

We will denominate the set of new links or edges (fill-in edges) added as T.

After triangulation our graph is $G^T = (V, E^M \cup T) = (V, E^T)$

In the example we are following we will use letters to make the expressions easier:

Visit to Asia=A, Smoker=Sm, Has tuberculosis=T, Has lung cancer=L, Has bronchitis=B, Has Tuberculosis or Cancer=ToC, Positive X-ray=X, Dyspnoea=D.

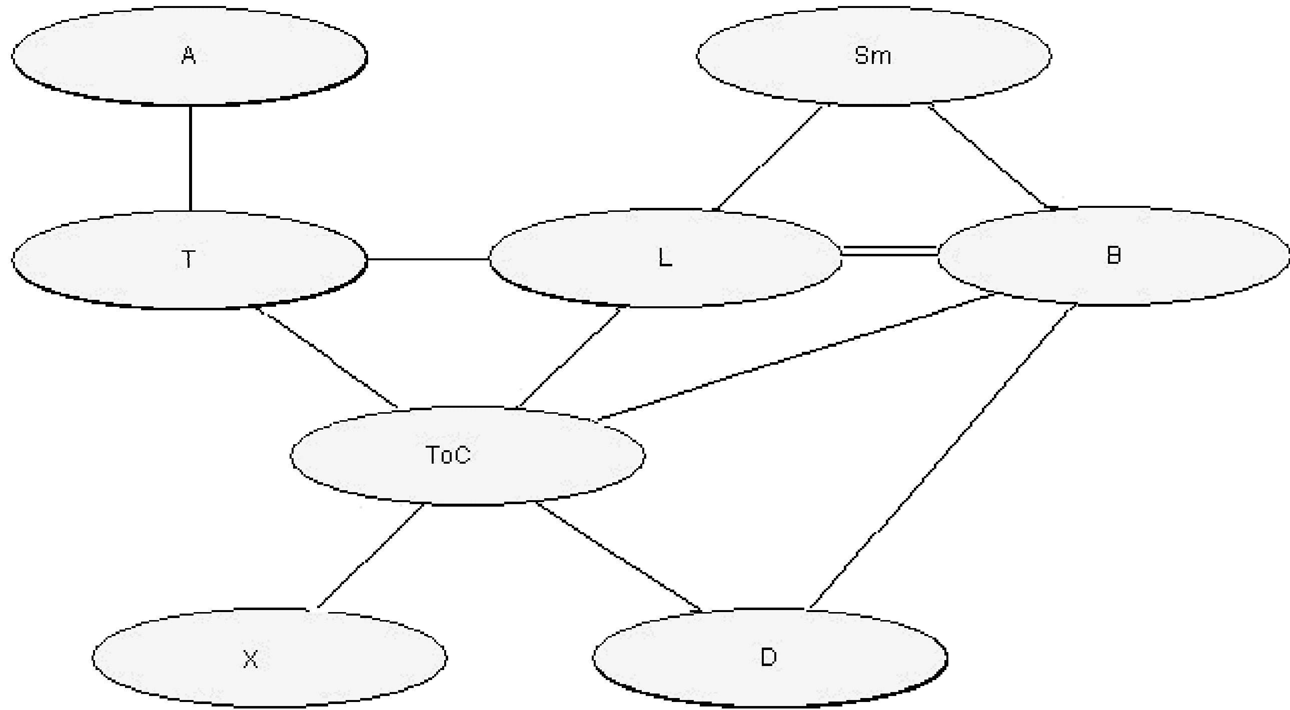


Figure 2- 3. Example of triangulation following the elimination order {A, T, X, D, Sm, B, L, ToC}; 1) A has only one neighbour, so no fill-in link is introduced; 2) T has his neighbours already connected; 3) X and D do not introduce new links either; 4) Elimination of Sm, forces us to introduce the fill-in link (L,B) and 5) {L,ToC,B} is already a complete subgraph. The triangulation is finished. The link added is drawn with a double line.

Step 1.3.- Maximum Cardinality Search (MCS).

In fact, MCS is a technique that we will use for being able to finally build the junction tree. There are other possibilities. We can divide this step in three actions over G^T. The first one will be the basis for the other two.

❶ Numbering

❷ Identifying cliques

❸ Building the tree

Let us explain them:

❶ Numbering → We choose randomly one node, and give him number 1. Then, we go on numbering the rest with the condition "next node to number is the one with more already numbered neighbours". In case of draw, choose any of them.

❷ Identifying cliques → In the **inverse order** of the numbering we go through all the nodes. For each one we will take the clique made up of this node and all his neighbours with a smaller number, excepting the cliques included in another clique already found in this process.

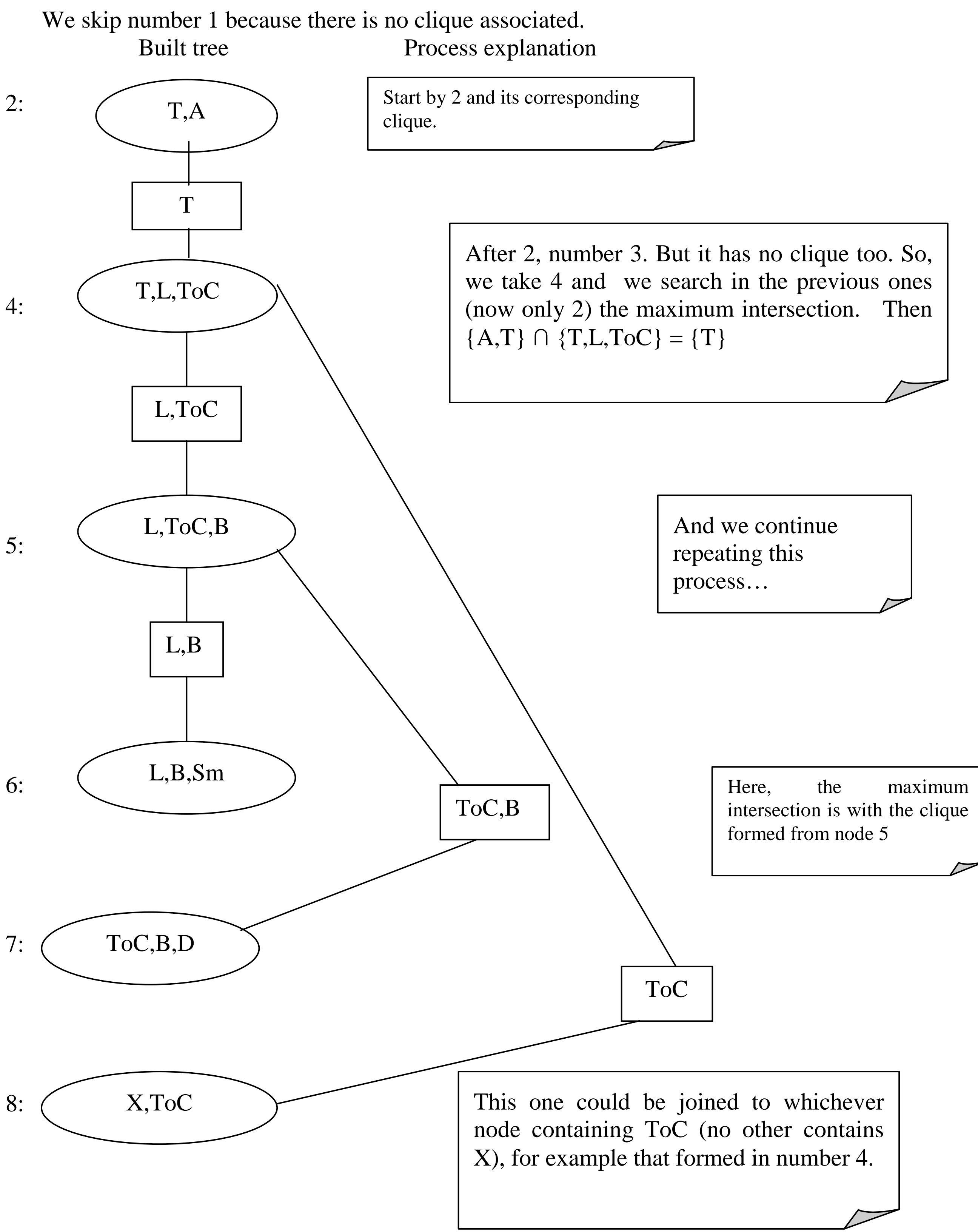
❸ Building the tree → Finally, we retake the initial numbering. Begin by 1, and take the clique associated (if any) as the first one. Next, take the second one, and look for one clique previously treated which implies the maximum intersection. If there are more than one, any of them can be chosen. This intersection will create a new object, also part of the tree, called a separator. As the name says it separates cliques in the tree.

Example:
Following the example of *Figure 2-3* we would do:
1.- Numbering : A←1, then: T ←2, L ←3, ToC ← 4, B ← 5, Sm ← 6, D← 7, X ←8.
2.- Cliques:

Number	Node	Clique
➢ 8	(X)	{X,ToC}
➢ 7	(D)	{ToC,B,D}
➢ 6	(Sm)	{L,B,Sm}
➢ 5	(B)	{L,B,ToC}
➢ 4	(ToC)	{T,L,ToC}
➢ 3	(L)	~~{L,T}~~
➢ 2	(T)	{T,A}
➢ 1	(A)	~~{A}~~

3.- Tree:

We skip number 1 because there is no clique associated.



2nd Step. PROPAGATION.

Before the propagation we must calculate the initial tables. This task is also included in the compilation itself, while the other two (2.2 and 2.3) are not really a part in this process, but as they are also relevant we find it convenient to explain them here.

Step 2.1.- Obtain the initial tables.

Let us remember the two components of a junction tree:

- Cliques: maximal sets of variables that form complete subgraphs.
- Separators: elements used for linking pairs of cliques, containing the set which results from the intersection between the cliques they relate.

All cliques and separators have a table initially composed only by ones. There will be a "1" in each entry which corresponds to the different combinations of the possible states in all the variables inside.

Since we are speaking about Bayesian networks, the conditional probability tables are available for us, and that will be the initial data to build the cliques tables necessary for this step. We will speak about potentials, whose notation is normally $\Phi_{\text{clique or separator}}$. So, the goal is to introduce the information we have in the junction tree we have just built. We must introduce the conditional probability tables in the adequate place and it is as we next explain.

If we remember the formula introduced in the introduction of this chapter, we had:

$$P(U)=\prod_i P(A_i| pa(A_i))$$

So, we are trying to represent this information in the tree. The systematic way to do this is to go through all potentials and find a clique and only one (because in the previous multiple product each potential appears also only once) to attach it to.

If we have a clique $C=\{A_1,\ldots,A_k\}$, its table will contain the product of all the associated potentials. To carry through this operation, we must take into account every possible combination of states in order to multiply them properly. If there are no related potentials the table will keep the initial ones.

Example:

For our initial network in *Figure 2-1*, the tree associated is shown in *Figure 2-4*.

Looking at *Figure 2-1*, we know which prior probabilities we shall treat:

P(A), P(Sm), P(T|A), P(L|Sm), P(B|Sm), P(ToC|T,L), P(X|ToC) and P(D|B,ToC). As we can see, there are eight, one for each node.

P(A) $\forall$ the only possibility is to introduce it in $\Phi_{A,T}$.

P(Sm) ∀ this one can only be introduce in $\Phi_{L,B,Sm}$, since it is the only place where the variable Sm appears.

P(T|A) ∀ in $\Phi_{A,T}$

P(L|Sm) ∀ in $\Phi_{L,B,Sm}$

P(B|Sm) ∀ in $\Phi_{L,B,Sm}$

P(ToC|T,L) ∀ in $\Phi_{T,L,ToC}$

P(X|ToC) ∀ in $\Phi_{ToC,X}$

P(D|B,ToC) ∀ in $\Phi_{ToC,B,D}$

Therefore, in the junction tree in *Figure 2-4* we will have the initial tables:

$\Phi_{X,ToC}$ = P(X|ToC)

$\Phi_{ToC,B,D}$ = P(D|ToC,B)

$\Phi_{L,B,Sm}$ = P(L|Sm) · P(B|Sm) · P(Sm)

$\Phi_{L,B,ToC}$ = All ones (there is no new information introduced)

$\Phi_{T,L,ToC}$ = P(ToC|T,L)

$\Phi_{A,T}$ = P(T|A) · P(A)

$\Phi_{L,B} = \Phi_{ToC,B} = \Phi_{L,ToC}$

1.00	1.00
1.00	1.00

$\Phi_{ToC} = \Phi_T$ = (1.00,1.00)

Let us take $\Phi_{A,T}$ as an example. Initially we have that:

P(A) = (0.01, 0.99)

P(T|A) =

T \ A	Yes	No
Yes	0.05	0.01
No	0.95	0.99

So, $\Phi_{A,T}$ will result to be:

T \ A	Yes	No
Yes	0.05 x 0.01 = 0.0005	0.01 x 0.99 = 0.0099
No	0.95 x 0.01 = 0.0095	0.99 x 0.99 = 0.9801

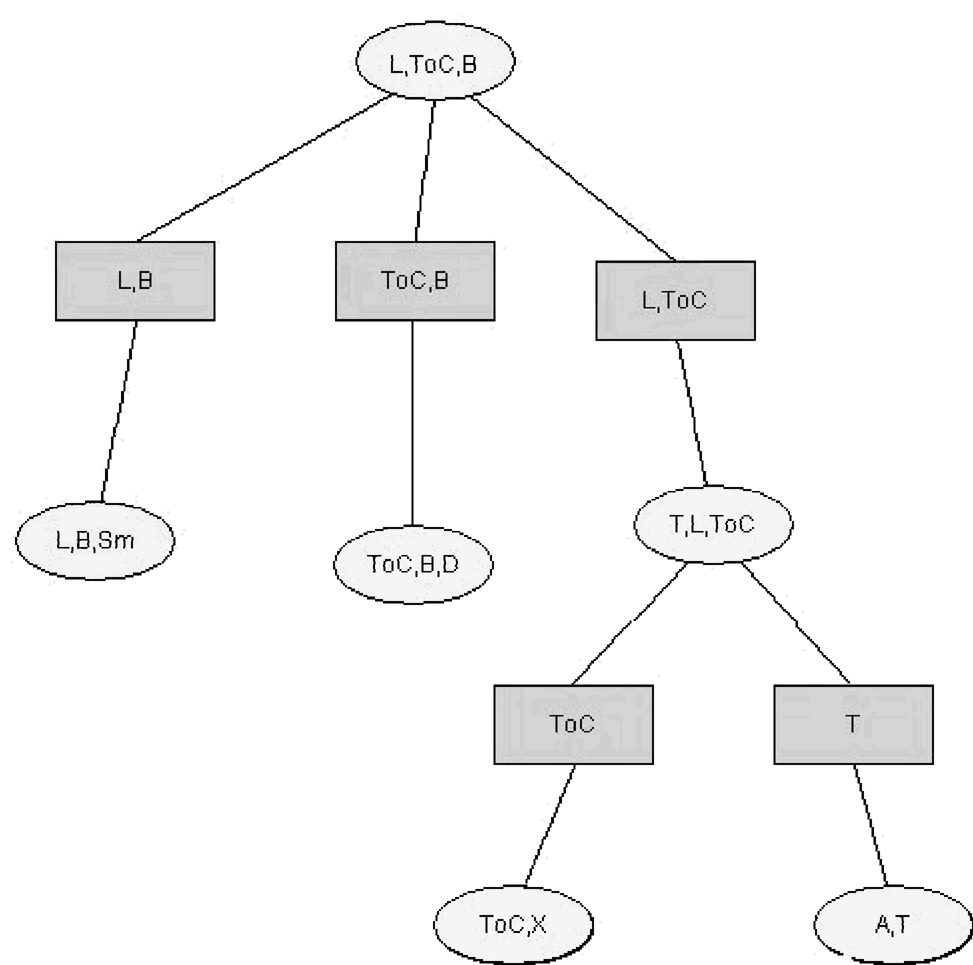


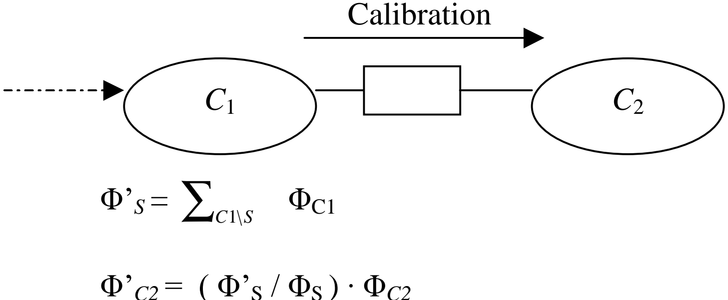
Figure 2- 4. Junction tree of the example built in step 1.3.

As we have commented before, at this point the compilation, as we understand it, has finished. Anyhow, it is interesting to describe propagation, because with this point it will be easier to clarify the necessity of a compilation process.

For the following two steps, we need to choose one of the cliques as the root one. Now, we are going to propagate all the original information through all the cliques. The purpose we pursue with steps 2.2 and 2.3 is a consistent tree. This consistence grants us to acquire the same marginal probability for each node, no matter from which clique or separator to which he belongs, we marginalise.

Step 2.2.- Collect phase.

One clique C_2 gets information from another one C_1 by means of the separator, S, between them. The separator has the nodes $C_1 \cap C_2$, and we update the potential using the operation illustrated bellow, which is commonly called *calibration*:



$\Phi'_S = \sum_{C1\backslash S} \Phi_{C1}$

$\Phi'_{C2} = (\Phi'_S / \Phi_S) \cdot \Phi_{C2}$

Once we have chosen the root node, we can think about the collect phase as a kind of survey which this node makes to all his neighbours, and these ones make it to their neighbours recursively, until finding a node without more neighbours, except the one who “asks” him.

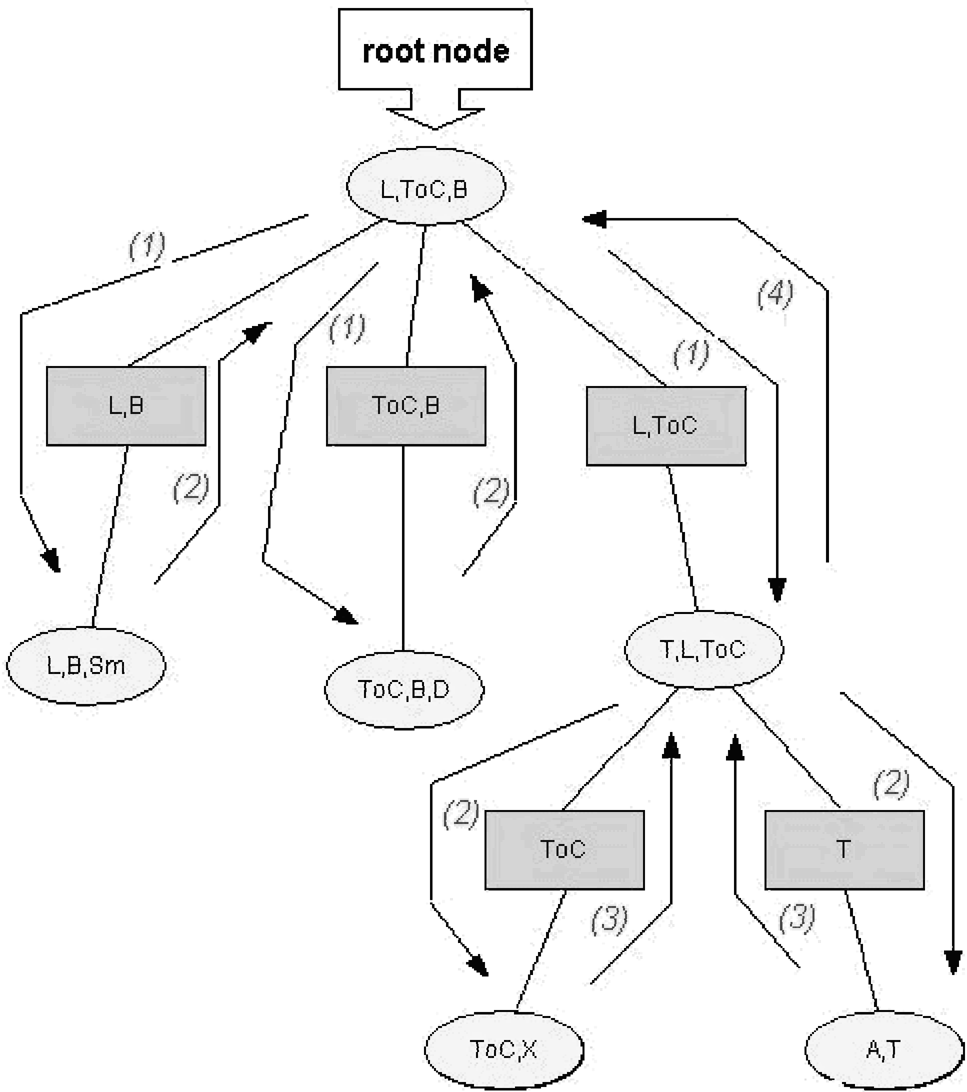


Figure 2- 5. Example of the collect phase development for tree in *Figure 2-4*. The download arrows are (recursive) calls to collect evidence and the upwards arrows is calibration.

Step 2.3.- Distribute phase.

Now, we can say that the root node "knows" everything about all cliques. So, he is going to communicate the rest. And performing a similar method we will distribute this information to his neighbours and so forth in a recursive way. Now what the nodes says to his neighbours is "calibrate from me", take what I know to have the complete information.

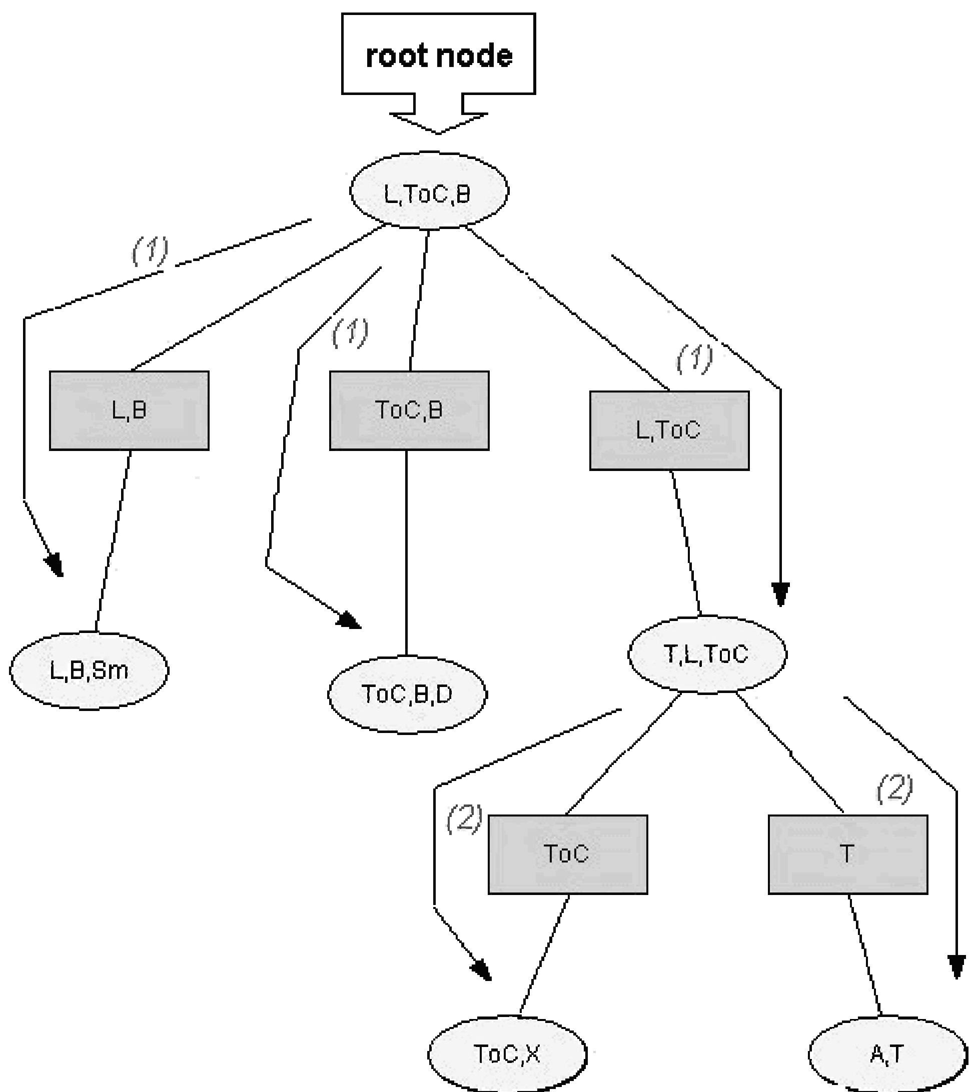


Figure 2- 6. Example of the distribute phase development for tree in *Figure 2-4*. The arrows symbolise (recursive) calls to distribute evidence, which results in calibration along the same paths.

Example

In the previous tree if we take, for instance, {L,ToC,B} as the root. We recommend following this sequence of steps watching *Figure 2-5* (Collect) and *Figure 2-6* (Distribute), numbers in brackets correspond with those in the drawings.

Collect phase → {L,ToC,B} “asks”

⇨{L,B,Sm}, he also asks his neighbours. *(1)*

In this case he has only {L,ToC,B} which sent him the message. So he can “answer” directly, and provokes the calculation of the separator:

$\Phi'_{L,B} = \sum_{Sm} \Phi_{L,B,Sm}$; *(2)*

⇨{ToC,B,D}, he also asks his neighbours. As before, he has not any. Then, *(1)*

$\Phi'_{ToC,B} = \sum_{D} \Phi_{ToC,B,D}$; *(2)*

⇨{T,L,ToC }, this one has neighbours, so he asks recursively: *(1)*

➔ {ToC,X}, no more neighbours: *(2)*

$\Phi'_{ToC} = \sum_{X} \Phi_{ToC,X}$; *(3)*

➔ {A,T}, no more neighbours *(2)*

$\Phi'_{T} = \sum_{A} \Phi_{A,T}$; *(3)*

↳ $\Phi'_{T,L,ToC} = (\Phi'_{ToC} / \Phi_{ToC}) \cdot (\Phi'_{T} / \Phi_{T}) \cdot \Phi_{T,L,ToC}$; $\Phi'_{L,ToC} = \sum_{B} \Phi'_{T,L,ToC}$; *(4)*

(3) *(3)*

↳ $\Phi'_{L,ToC,B} = (\Phi'_{L,B} / \Phi_{L,B}) \cdot (\Phi'_{ToC,B} / \Phi_{ToC,B}) \cdot (\Phi'_{L,ToC} / \Phi_{L,ToC}) \cdot \Phi_{L,ToC,B}$

(2) *(2)* *(4)*

Distribute phase → {L,ToC,B} says to his neighbours to calibrate from him. To do this, the separator must be updated [step a], and then cliques will use this new value of the separator combined with the previous one to update its potential [step b].

⇨{L,B,Sm}, he calibrates taking *(1)*

$\Phi''_{L,B} = \sum_{ToC} \Phi_{L,ToC,B}$ [step a] and $\Phi'_{L,B,Sm} = (\Phi''_{L,B} / \Phi'_{L,B}) \Phi_{L,B,Sm}$ [step b]

He has no more neighbours for telling them to calibrate from him.

⇨{ToC,B,D}, the same case as before. *(1)*

$\Phi''_{ToC,B} = \sum_{L} \Phi_{L,ToC,B,}$ and $\Phi'_{ToC,B,D} = (\Phi''_{ToC,,B} / \Phi'_{ToC,B})\ \Phi_{ToC,B,D}$

⇨{T,L,ToC }, we start calibrating him. *(1)*

$\Phi''_{L,ToC} = \sum_{B} \Phi_{L,ToC,B}$ and $\Phi''_{T,L,ToC} = (\Phi''_{L,ToC} / \Phi'_{L,ToC})\ \Phi'_{T,L,ToC}$

➔ {ToC,X}, he calibrates from the previous one: *(2)*

$$\Phi''_{ToC} = \sum_{T,L} \Phi'_{T,L,ToC}\ ;\ \Phi'_{ToC,X} = (\Phi''_{ToC} / \Phi'_{ToC})\ \Phi_{ToC,X}\ ;$$

➔ {A,T}, *(2)*

$$\Phi''_{T} = \sum_{ToC,L} \Phi'_{T,L,ToC}\ ;\ \Phi'_{A,T} = (\Phi''_{T} / \Phi'_{T})\ \Phi'_{A,T}\ ;$$

At this point the tree is globally consistent. With these steps the Bayesian network is initialised. Now we are ready to work with the network for example by introducing evidence in it.

To finish this chapter, we want to point out that its purpose is to introduce the method used for compiling a Bayesian network. We have tried to show a description easy to follow. For further details we recommend [Jensen 1996].

Chapter 3. POSSIBLE MODIFICATIONS IN A BAYESIAN NETWORK

2. Why we should look into it.

Since we want to study incremental compilation of a Bayesian network, we must start by analysing possible changes in it. These changes can arise at any time in the Bayesian network construction (the modelling phase) or maybe later in a revision task.

3. Systematic search of possible changes.

Looking at the definition of Bayesian network, given in chapter 2, we are able to examine the different parts of a Bayesian network that can be modified.

We have the structure BN = {G, P}, where G is a directed acyclic graph, G = (V, E). So, how can we go through each component of this structure? What we are going to do is that, for each element, we will try to identify the possible modifications and afterwards seek to deduce the implications in the compilation process as well as the cases where any computation work could be saved.

4. Potentials.

If we take the process of compilation, when do we use the potentials? Following the compilation steps described in chapter 2, we see that these data first appear in the construction of the initial tables in the junction tree. The potentials let us obtain every clique potential.

We commented that the real compilation ends by filling initial tables. Afterwards, we pass to propagation phase. Here, we should say that this discussion would be slightly different if we consider propagation inside compilation or not. For example, in this point about potentials if propagation is not included we will have to change only initial tables. Otherwise, we can imagine that making a change in an initial potential will imply changes in every clique potential table where the implicated variables participate. The junction tree is still the same, but the potentials are not.

- Possible consequences of changing potentials

If we change initial potentials for any variable we could expect that from step 2 (propagation), everything should be redone. It would be necessary to introduce the changed potentials in the corresponding tables. That would affect all the following process.

In order to do a better study, we are going to take the initial example of the Bayesian network Asia and we will try to make modifications with the purpose of observing the consequences in the compilation process. We know this is only a network example, and it will present particular characteristics that cannot necessarily be extended to general conclusions. Anyway, it can be interesting to see what happens in each of the described cases. Then, examples are taken from there.

Example 3.①.

Initially we had the values:

	Yes	*No*
A	*0.01*	*0.99*

Now we suppose changes and:

	Yes	*No*
A	*0.2*	*0.8*

Let study what happens with these new values:

It is clear that, as the qualitative structure of the Bayesian network does not change, neither does G^M, so neither does G^T, and finally the junction tree does not change either. But, in step 2.1, we start seeing that P(A) is not the same. That makes $\Phi_{A,T}$ different too.

Before we had:

P(A) = (0.01, 0.99)

P(T|A) =

T \ A	Yes	No
Yes	0.05	0.01
No	0.95	0.99

$\Phi_{A,T}$ →

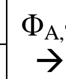
T \ A	Yes	No
Yes	0.05 x 0.01 = 0.0005	0.02 x 0.99 = 0.0099
No	0.95 x 0.01 = 0.0095	0.99 x 0.99 = 0.9801

And now, we would have:

P(A) = (0.2, 0.8)

P(T|A) =

T \ A	Yes	No
Yes	0.05	0.01
No	0.95	0.99



T \ A	Yes	No
Yes	0.05 x 0.2= 0.01	0.01x 0.8 = 0.008
No	0.95 x 0.2 = 0.19	0.99 x 0.8 = 0.792

As it seemed evident, tables have changed, that means that from step 2.1. the process must be retaken.

- Proposed solutions

In this case we find one solution quite easy. As in the example before, changing initial potentials implies changing potential tables in cliques, but not in a random way, what we actually do is substitute the old value by the new one. So, in the incremental compilation one possible method to cope with it would be to divide by the old value and multiply by the new one. That would be done in the table where this potential participated. We have to remember the step 2.1 in compilation (obtaining the initial table), where every potential had to be included in one and only one clique. Like that, we only detect one situation where this solution would not work, if the old value is 0 we cannot divide by it. But if the new value is also 0, then nothing has to be done, because the value remains the same.

5. Graph.

The next component to see is the graph. But this one will be more complicated to study. A graph is, composed of other elements. Let us remember its definition G=(V, E). The division is then immediate, variables and edges.

4.1.Variables.

It can be very reasonable to change one or more variables in a Bayesian network. Which possibilities can we find? Once more, we will try to look into them in a systematic way.

4.1.1. States.

To start with, we can find that the possible states in a variable are not the correct ones, too many or maybe too few states. It is not really part of the graph, we could consider it outside the graph, but it is clear that states are closely related to variables. For that reason we will discuss this case at this point.

- Possible consequences of changing the states at a variable.

What happens if a variable changes its states? Firstly, the number of states will cause changes in potentials values. Hence, that will provoke at least the same consequences as changes in potential values. But also, the states are very close to the potential tables, since they determine the size of these tables. So, incrementing or decrementing the number of states in one variable will modify tables. For example, assuming that there is a variable A without parents, its table will change. But even more, any variable child (B for instance) of A, will change its table as well.

Conditional probability for B would be P(B| pa(B)), if A is one of its parents, then for each new state of A, B will have to extend the table. And if a state of A disappears, the table of B will be reduced too.

Let B be a variable with *s* possible states. If this variable has *m* parents p_i (1≤i≤m) the number of entries in the probability table related to B will be *E*, where

$$E = s \quad * \quad \prod_{i=1}^{m} States \quad of \quad p_i$$

So, it is not hard to see that if A, one parent of B, increases the number of states then the number of entries for B is bigger. Or if one or more states in A are deleted then the number of entries is smaller.

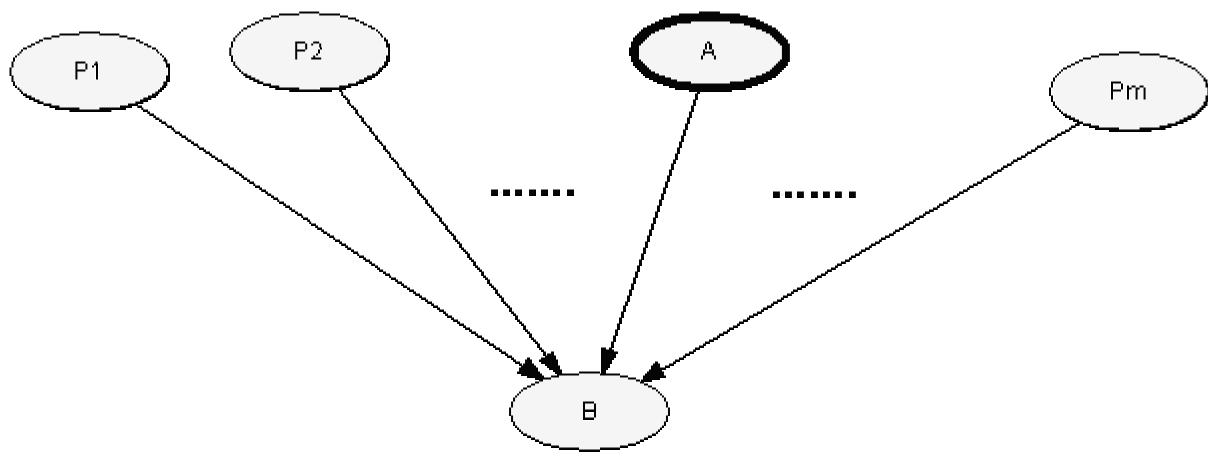


Figure 3- 1. Example of variable B with several parents p_i and, between them, variable A whose state set has been modified.

Then, changing the number of states in a variable will form new tables. These tables will provoke new clique sizes and therefore new clique tables. Like this, if for example we use the minimum clique size heuristic, the process will be already affected in triangulation step. Otherwise, propagation will be again the point from which we must begin the recompilation.

Example 4.1.1.①.

Let us imagine that now we reconsider the global problem and a new state for A (Visit to Asia?) is required. We have learnt that the time passed from the visit to Asia is quite important to determine that illness. Thus, we find that A ={yes in two months time, yes more than two months ago, no}.

→ Looking to the compilation chapter, we can see that nothing changes until step 2 (initialisation of junction tree). But then, we will have:

	- 2 months	+ 2 months	no	
P(A) = (	--- ,	--- ,	---	)

$\Phi_{A,T}$

T \ A	**- 2 months**	+ 2 months	No
Yes	----	----	----
No	----	----	----

We see that changing the sates of A will provoke changes in P(A) ans in potential table $\Phi_{A,T}$. A has now one more state and whatever table including it must consider this new state in the configurations.

Example 4.1.1.②.
*Changes in T = {no, **mild**, severe}.*

→ In this case in step 2.1 we will obtain:

	no	**mild**	severe	
P(T) = (	---,	--- ,	---	)

$\Phi_{T,L,ToC}$

L	Yes		No	
ToC	Yes	No	Yes	No
No	---	---	---	---
Mild	---	---	---	---
Severe	---	---	---	---

$\Phi_{A,T}$

T \ A	Yes	No
No	----	----
Mild	----	----
Severe	----	----

This is a similar example to the previous one. In this case a change in T potentials will affect in two tables, since it participated in these two clique potential tables.

- Proposed solutions

Here we have another change in tables at step 2.1, but this change is more serious, it does not only deals with values but also with table dimensions. So, this time we should find one dynamic way to change these table dimensions, and we could also use the previous method for those existing states, if their values are changed too.

4.1.2. Deletion.

But, going further, suddenly we can recognise that this variable is not really essential in our network and we simply want to delete it. Probably, making modifications in a variable will have some kind of impact in the edges set within the graph G. For that, the only thing we must think about is that deleting a variable will obviously involve deleting any edge containing it. We must remember that an edge is a pair of two variables, without one of them it would have no sense to keep it.

- Possible consequences of deleting an existing variable

This case starts to be a little more “annoying”. Deleting a variable and its incident edges as we told before produces a different moral graph G^M. Subsequently, this new G^M takes to a different triangulation and a different G^T, and finally it drives to another junction tree.

But, we can think in particular cases where the change in the junction tree provoked by a deleted variable might not be so serious from the point of view of compilation. When could it happen?

For example, let us think of those variables that only appear in one clique. Deleting one of them will only reduce this clique and the separators related, and maybe it could make the clique disappear (if it becomes a subset of another clique). That means our junction tree is reduced, but we will probably not have to do all the junction tree construction again. For this purpose, we will need some method to detect these cases.

Example 4.1.2.①.
Deleting variable X in Asia . The simplest case, deleting a variable that has only one parent and no children. This is the simplest one because deleting a child only affects itself and its potentials.

For the first time, I am going to explain the example in detail as in the compilation chapter, but afterwards we are going to simplify the examples in order to make it lighter to follow. So, the way of showing this first example coincides with the one taken in chapter 2 of compilation: initial network, moral graph, triangulated graph, MCS numbering that will take to identification of cliques. Then, finally we will see the junction tree, the structure we was looking for. But, since this chapter will present many examples, for the other ones we will show them in a more compact way: initial network (showing the changes from Asia), graph after being moralised and triangulated (fill-ins represented by double lines) with a certain elimination order. Then we will give the MCS numbering and finally we will show the junction tree.

So, the Bayesian network is:

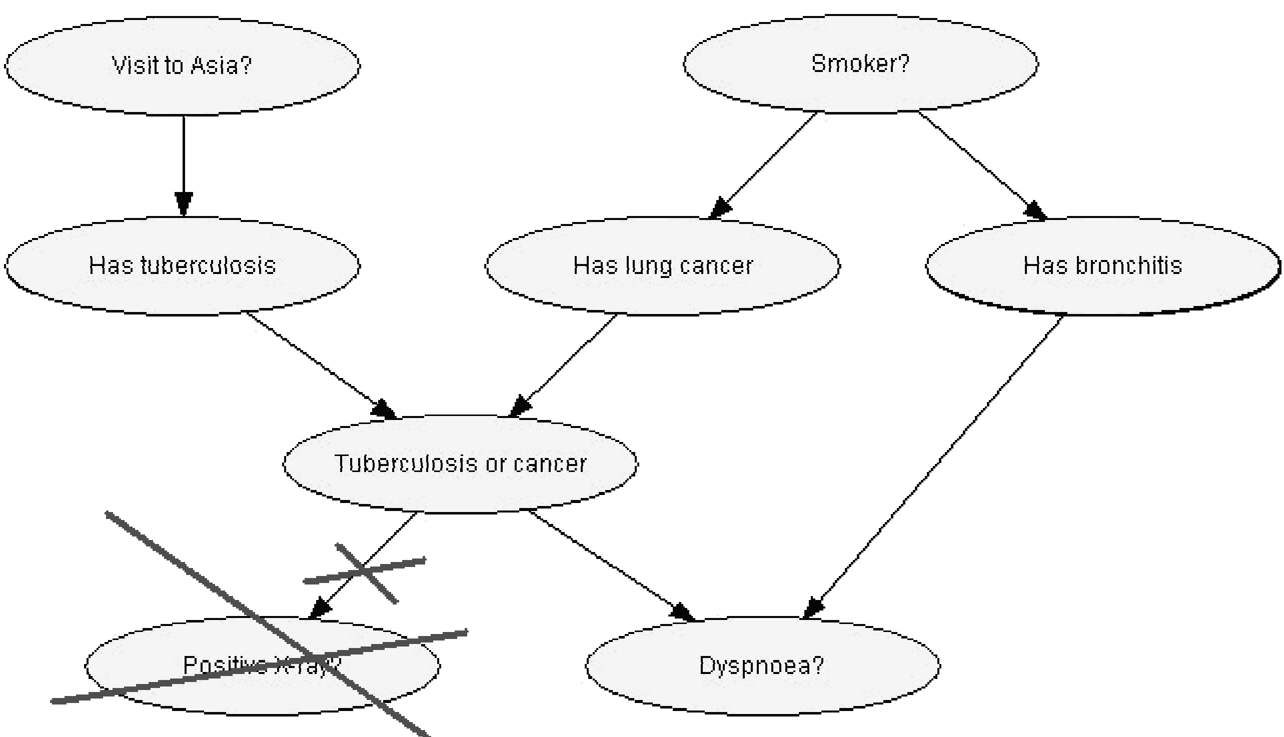


Figure 3- 2. Asia without variable X.

After moralisation we have:

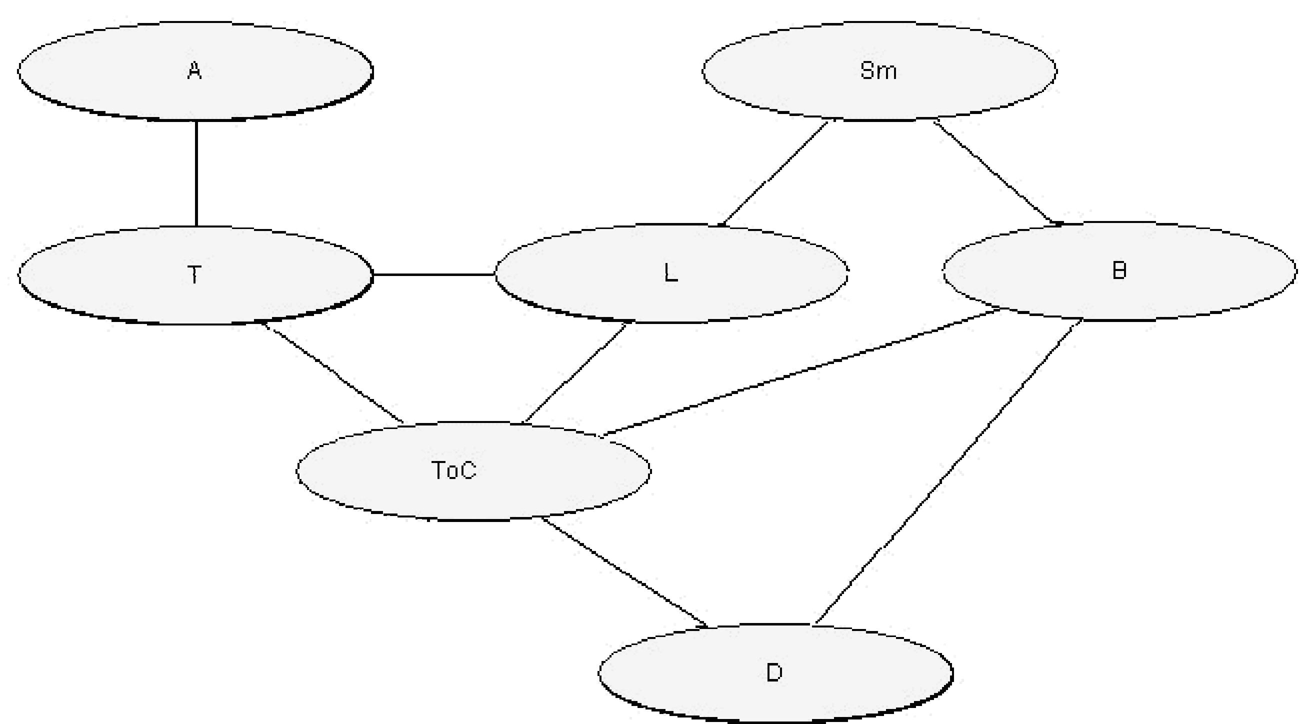


Figure 3- 3. Asia without X moralised.

Triangulation sequence: {A, T, D, Sm, B, L, ToC}, which introduces fill link {L,B}.

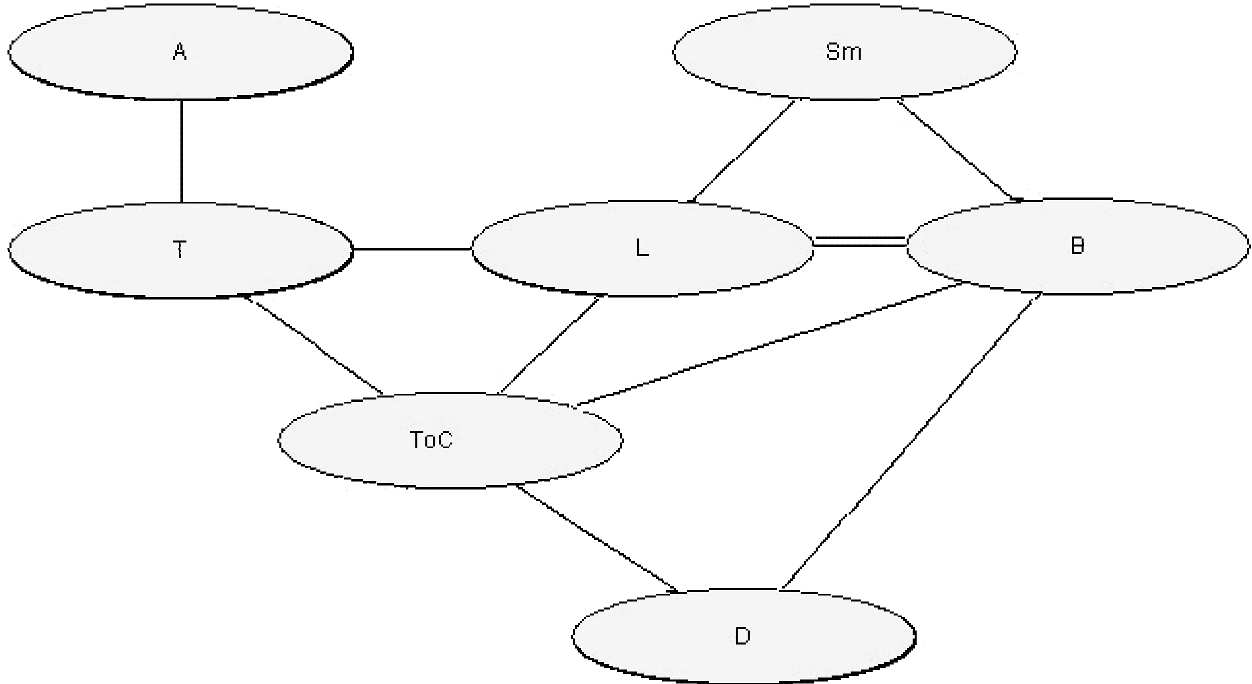


Figure 3- 4. Asia without X triangulated.

- Numbering: A←1, then: T ←2, L ←3, ToC ← 4, B ← 5, Sm← 6, D ←7.
- Cliques:

Number	Node	Clique
➢ 7	(D)	{ToC,B,D}
➢ 6	(Sm)	{L,B,Sm}
➢ 5	(B)	{L,ToC,B}
➢ 4	(ToC)	{T,L,ToC}
➢ 3	(L)	~~{L,T}~~
➢ 2	(T)	{T,A}
➢ 1	(A)	~~{A}~~

- Tree:

We will not show the tree construction. We think that the number of cliques is reduced enough to see this construction process directly in the final tree.

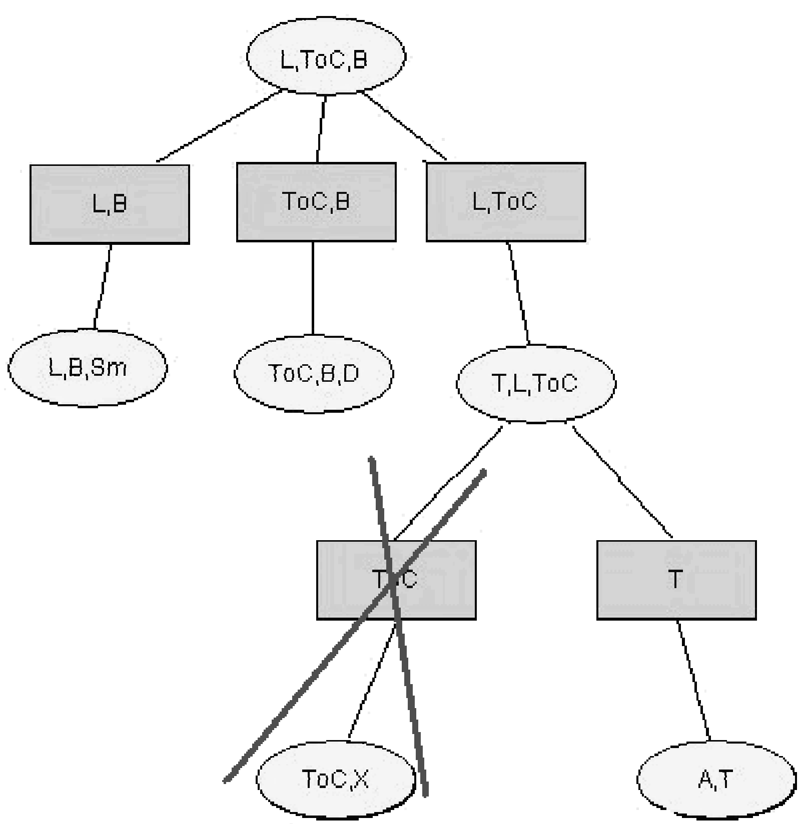


Figure 3- 5. Junction tree without X. We have already marked the differences from the original JT.

In this example deleting X takes us to delete one branch in the junction tree. This branch corresponds with the only clique related to it.

Example 4.1.2.②.
Delete Visit to Asia? This is a little more complicated in the sense that this time, since the deleted node is a parent there are more potentials implicated.

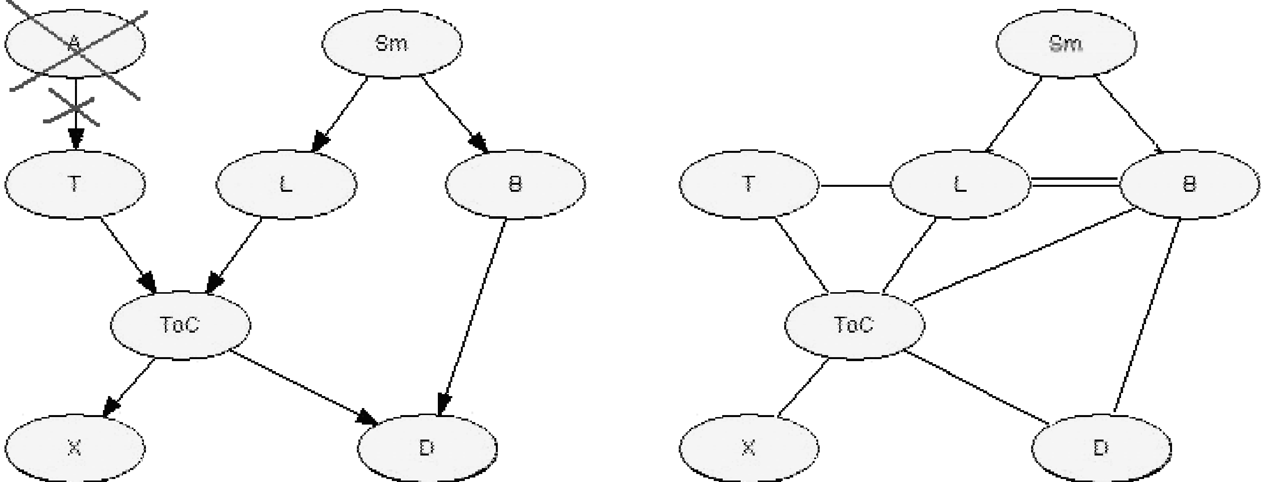


Figure 3- 6. Asia without A

Figure 3- 7. Asia without A moralised and triangulated (fill-in {L,B}) .Sequence order taken: {A,T,D,Sm,B,L,ToC}.

And with the numbering: T←1, then: L ←2, ToC ←3, B ← 4, Sm ← 5, D ← 6, X← 7, we obtain the tree shown in *Figure 3-8.*

In this example, we will "lose" only a branch in a similar way to 4.1.2.①. But, as we told before, this is not exactly the same case. Now deleting A will also affect on T potentials. If we see the original Bayesian network T is a child of A. So, its conditional probabilty will be P(T|A). So, if A is deleted the potential of T changes as well.

Once seen two cases of a variable father of only another one and a variable child of another one, it arrives to think about one variable linked to the rest of the network with more edges. To start with, we will consider two other edges. In which position? There are three possible ways: two incoming links (the simplest case following the same reasoning as for only one link), two outgoing, and one incoming and one outgoing.

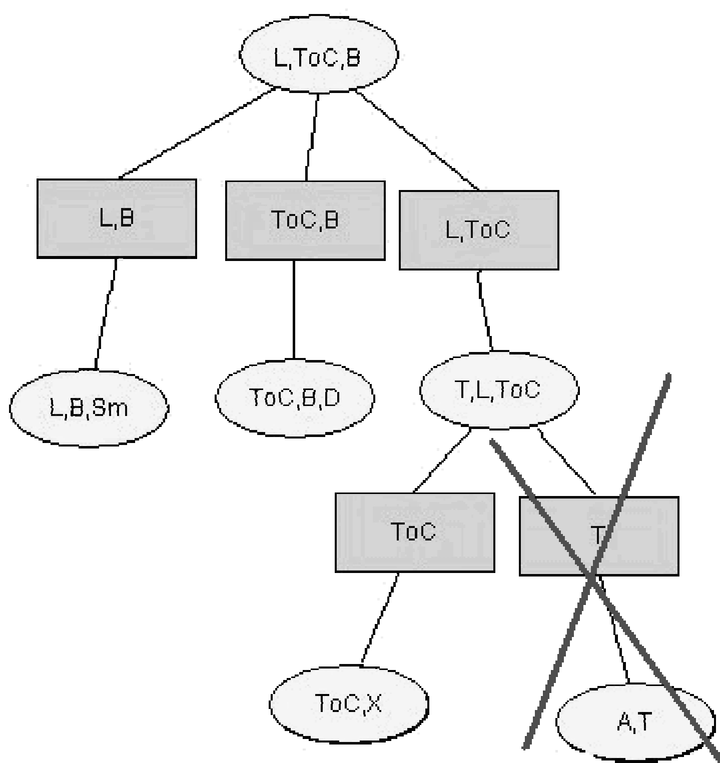


Figure 3- 8. Junction tree of Asia without A

Example 4.1.2.③.
One variable with two incoming links deleted. In Asia, we could delete D to see this point.
Taking the process as always we obtain

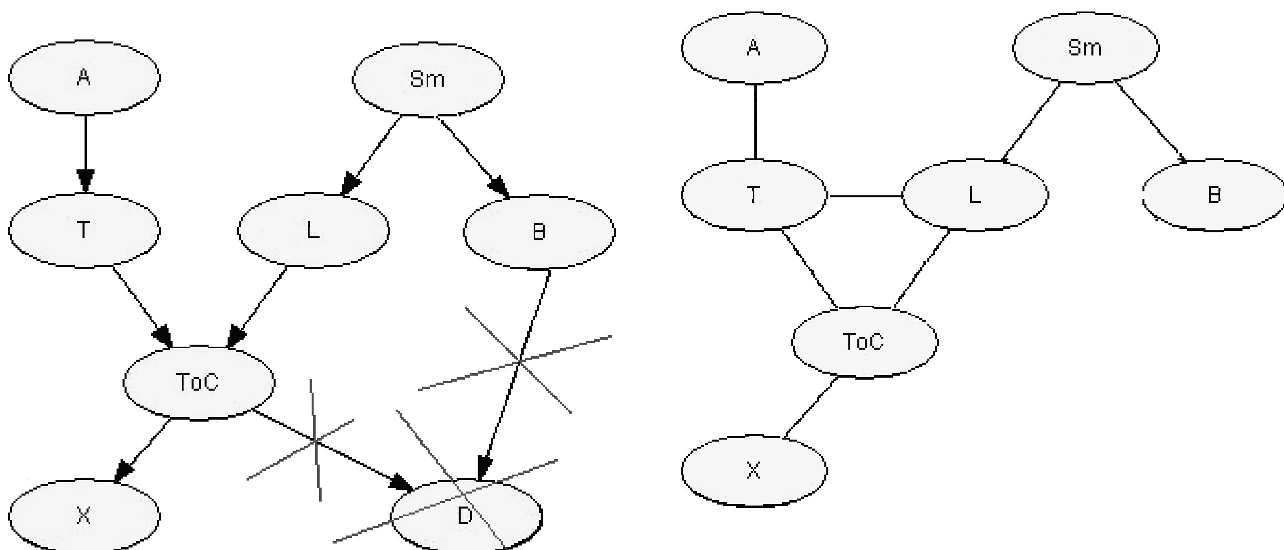


Figure 3- 9. Asia without D.

Figure 3- 10. Asia without D moralised and triangulated. Sequence: {A,T,X,B,Sm,L,ToC}

With the numbering A←1, and then T←2, L←3, ToC←4, Sm←5, B←6 , X←7, we obtain the tree:

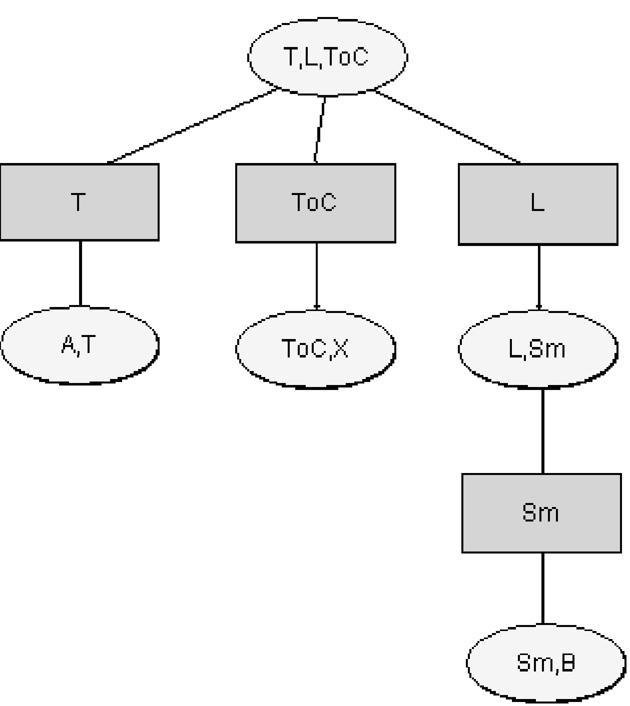


Figure 3- 11. JT for Asia without D following the usual compilation process.

Looking at this new tree we can see that this tree differs a little bit from the original one. It keeps the same the part {T,L,ToC} connected to {A,T} and {ToC,X}, but the rest turns to be slightly simpler.

Well, this was the first option, redoing everything. But this, recompilation, is precisely what we want to avoid. So, is there another way to do it? The easiest way to act will be taking the junction tree and delete the variable in the cliques where it appears. It is in fact what we did in the previous cases. Once the affected cliques are deleted we have to see if the tree is correct and absorb those cliques subset of others.

But, is it possible to do in this way? We are going to see what happens, in fact the point is that if D is deleted then the *marriage* between B and ToC is not necessary either. That means that we do not need the moral link between these two nodes. So acting directly on the junction tree will not consider the disappearing of this moral link. We realise that if this moral link continues there the tree is quite easier to reach, even there is a systematic method to obtain it from the previous junction tree. And this tree will be also valid, but no the most simplified one.

This validity can be justified from the principle of junction trees. That lies on the necessity of reducing the total number of configuration between variables of the Bayesian network to store. With moralisation step we can assure that parents of the same node are related,

since they are dependent in some way. And later on, triangulation is charged of splitting the moral graph up into smaller modules or subsystems that will provide smaller configurations than a global one.

So, deleting a link could never produce new dependencies to consider. And the worst thing to happen is having more tables (configurations) than we could, but the structure is still valid.

To illustrate this we are going to take again *Example 4.1.2.③.*(*Figure 3-9*), but this time without removing moral link between B and ToC.

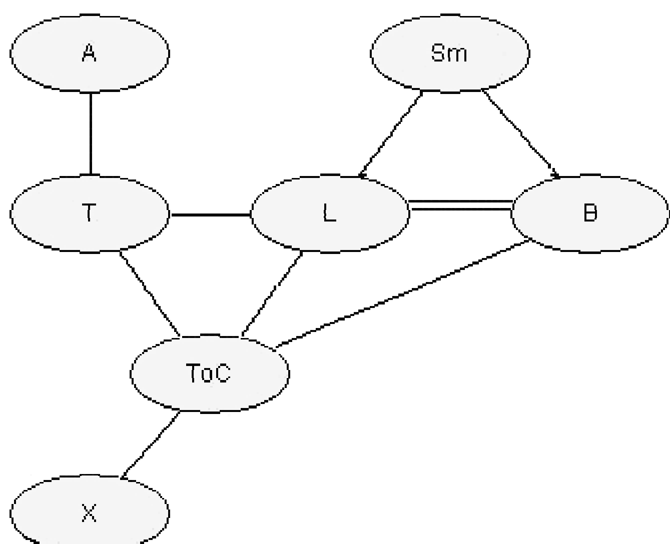


Figure 3- 12. Asia without D having kept previous moral links.

With the numbering A←1, and then T←2, L←3, ToC←4, B←5, Sm←6 , X←7, we obtain the tree:

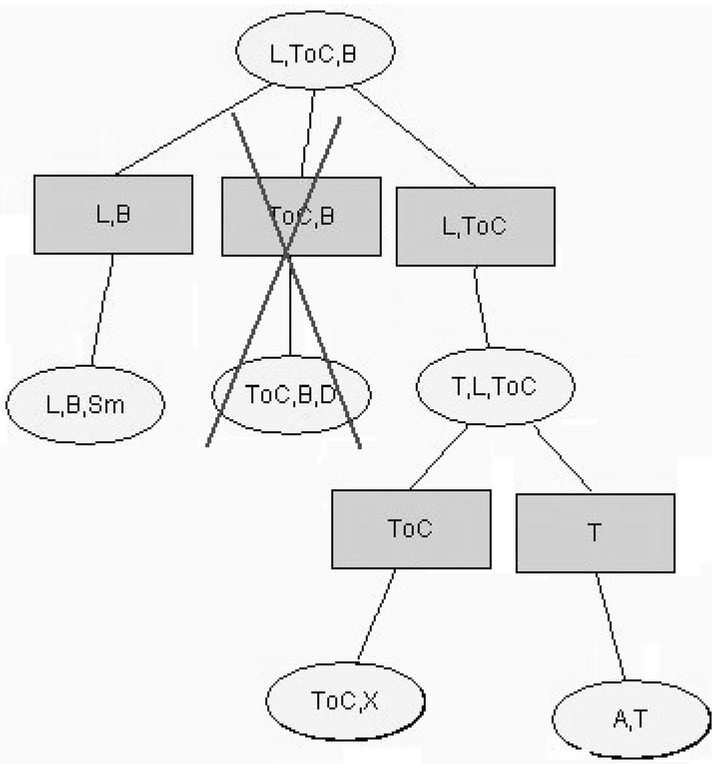


Figure 3- 13. JT of Asia without D having kept previous moral links.

In this case the branch where D was disappears. It makes us see that the method can be: take the clique (or cliques) where the variable deleted appeared in the original JT. So, if deleting this variable inside a clique provokes a situation where the clique coincides exactly with one separator linked to it, this clique has no more sense in the JT, since it is then a subset of another one (the one with the separator connects with). Then, in this example it is that what happens, we delete D from the only clique where we find it. Doing that the new clique stays as {Toc,B} which is just the same as the separator close to it. It is for that reason, that both of them disappear.

Then, for this example we have seen two possibilities:

1.- Recompile the network again (the one we are trying to avoid in this report). It gives a quite simpler tree, but it is computationally demanding.

2.- Delete the cliques that contains the variable. It is a simpler method and gives a valid tree, but not the best result.

We have passed from the “traditional” way with a good solution to a much faster way, but offering a worse solution. So, we wonder it maybe there is an intermediate point to solve the problem. It could be quite interesting if we are able to infer that the elimination of a variable will take us to the elimination of a certain moral link. And looking at it, we notice it is possible. We only need that in moralisation process we noted in some way the introduced moral links relating it to the nodes that provoked them. Like that, if a node whose parents were *married* is deleted then this/these moral link(s) between them will disappear.

So, we can guess that a third solution could be reached. And the obtained tree will be probably simpler than the one shown in *Figure 3-13*. We comment it because we find it is a way to follow the analysis, but let us leave the discussion at this point, without entering in more detail in this intermediate solution.

Anyway, wee are going to show it with more examples to see if these methods can be generalised.

Example 4.1.2.④.
One variable with two outgoing links deleted. In Asia, we could delete Sm to see this point.

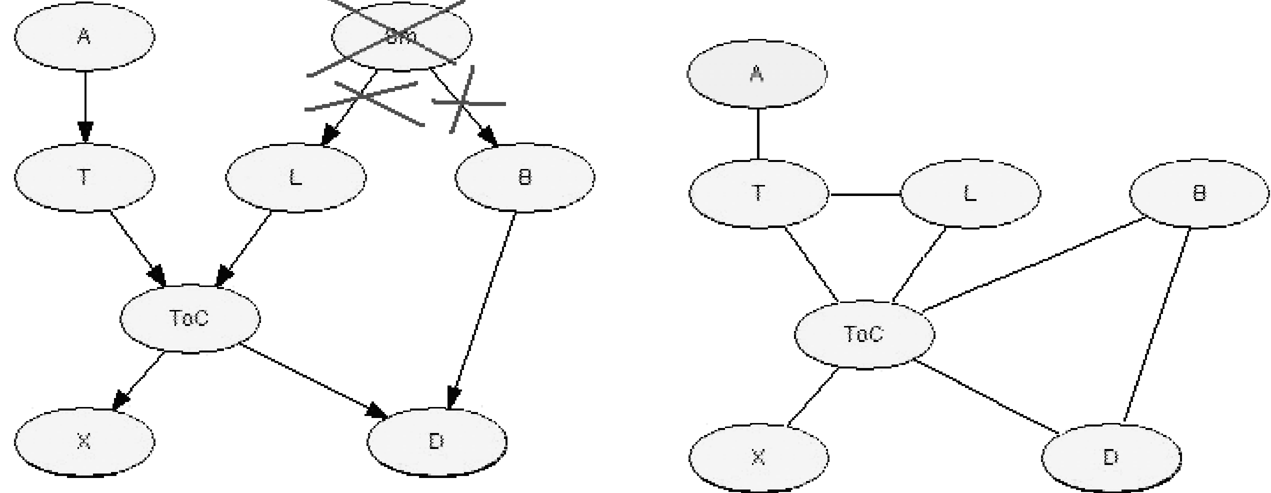


Figure 3- 14. Asia without Sm

Figure 3- 15. Asia without Sm moralised and triangulated. Sequence: {A,T,X,D,B,L,ToC}

With the numbering A←1, and then T←2, L←3, ToC←4, B←5, D←6, X←7 , we obtain the tree:

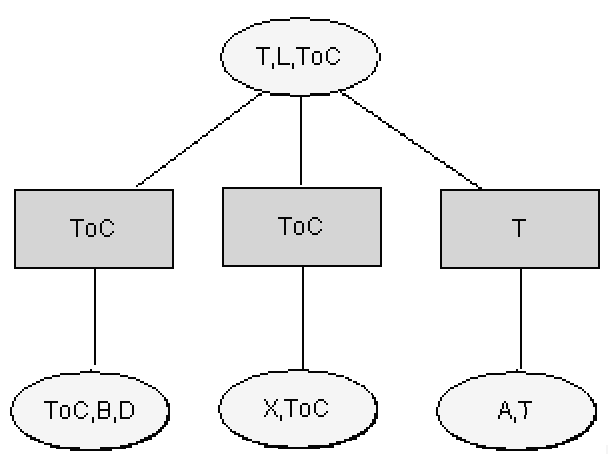


Figure 3- 16. JT of Asia without Sm.

If we compare with the original tree, we have “lost” two cliques {L,B,Sm} (quite normal because Sm has been deleted}, but also {L,ToC,B}. This one is not so obvious to see, but we have to bear in mind that the links related to Sm provoked the fill-in edge L-B, that now we have not. Again, the explanation is not so hard, but we do not really know how to mechanise this type of situations.

Maybe we can follow the same line of reasoning than in the previous example. We can delete the variable directly from the junction tree. We have just told that the key here is the fill-in between L and B, this link with the disappearing of Sm is no more necessary. If we keep it, the tree will be still valid taking the same justification as before.

Let see in the example. We delete Sm not from the original Bayesian network, but from the original G^T. And then we have as the new G^T:

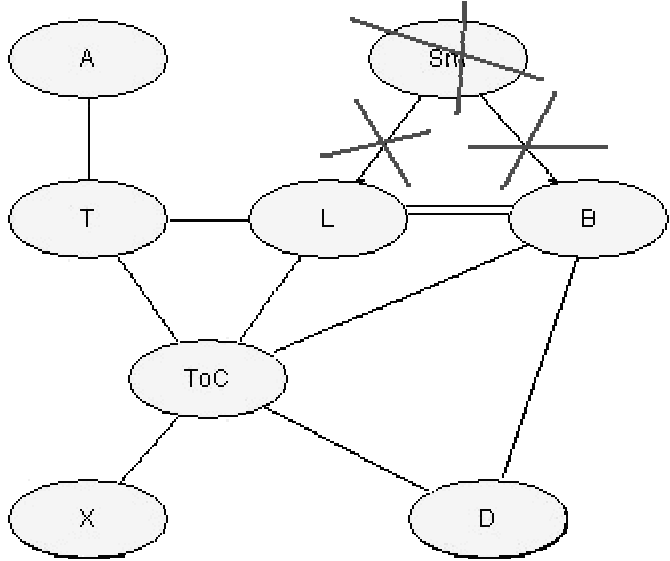


Figure 3- 17. Asia without Sm moralised and triangulated having kept previous fill-in links.

With the numbering A←1, and then T←2, L←3, ToC←4, B←5, D←6, X←7 , we obtain the tree:

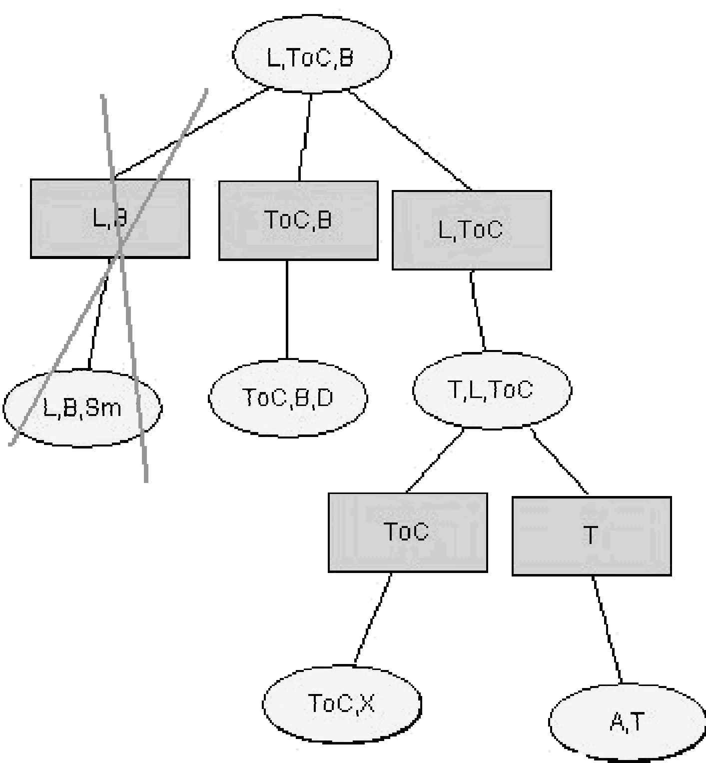


Figure 3- 18. JT for Asia without Sm having kept fill-in edges.

Here, we find that a similar situation has taken place. If we delete Sm from the original G^T, then it is quite simple to identify a new junction tree. We look for a clique containing Sm, then delete it, and the clique is then {L,B}, but as it is the same as the separator . So, both of them are absorbed by the tree.

Subsequently, that is very near to the previous case. First, we have the obvious but slow way to do it: do all the process from the Bayesian network to the junction tree again. Or, second, deleting the associated cliques in the old tree. This is quite easier, but less effective, since the tree is not the best one we could have. And what about the third solution?

Here it is even more complicated, because the link which was in the original G^T and which does not appear anymore is a fill-in. Keeping track of a moral link could be relatively easy, but it is not the same when we talk about triangulation links. Why? A moral link is associated to a child node that makes his parents *marry*. But a triangulation link can influence on several nodes deletion, not only on the one whose elimination introduced it. Here, according to the triangulation sequence we chose, elimination of both Sm and B could induce this link.

Still, we can distinguish one question to reach an intermediate solution. In this case, a cycle is broken. Cycles and triangulation links, fill-in links, are quite related (see definition in chapter 2, at step 1.2) as we will see later. And detecting cycles is a possible task to do in the graph that we have already. Thus, this is the hint we launch to attain a better solution to the second one and faster than the other one.

Let us go on with the next example: deleting a node with one outgoing edge and one incoming one.

Example 4.1.2.⑤.
Suppose we realise we do not need variable B (Bronchitis). A more complicated situation, since B is both child and parent.

→ That means we have change our graphical Bayesian network. Let see step 1.

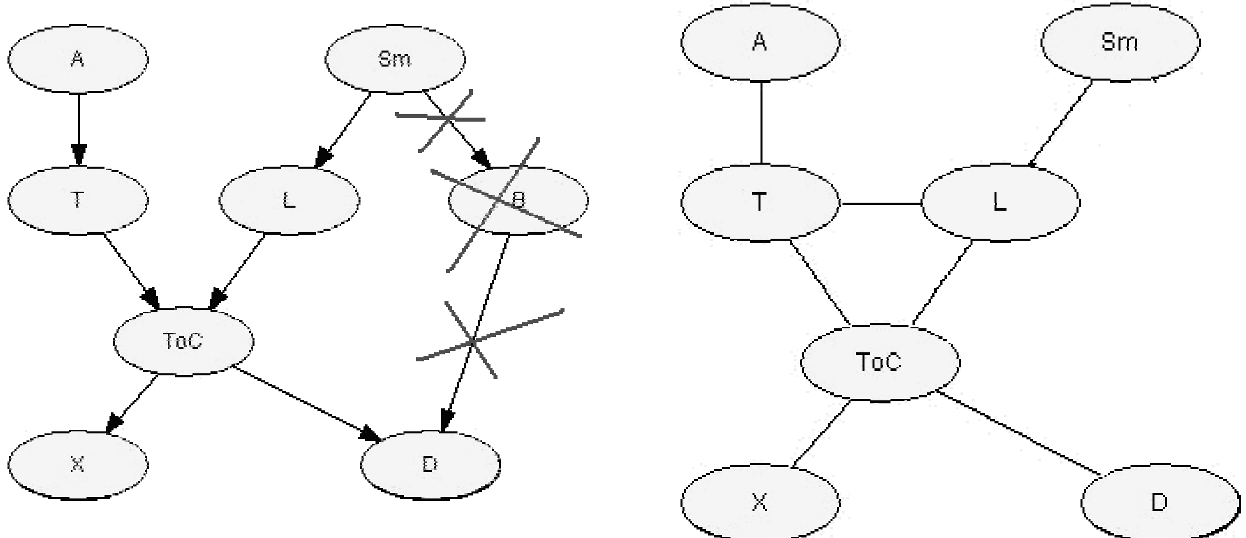


Figure 3- 19. Asia without variable B

Figure 3- 20. G^M from Asia without B,already triangulated, so it also coincides with G^T. We tried sequence {A,T,X,D,Sm,L,ToC}

And with the numbering: A←1, then: T ←2, L ←3, ToC ← 4, Sm ← 5, D← 6, X ←7, we obtain the tree:

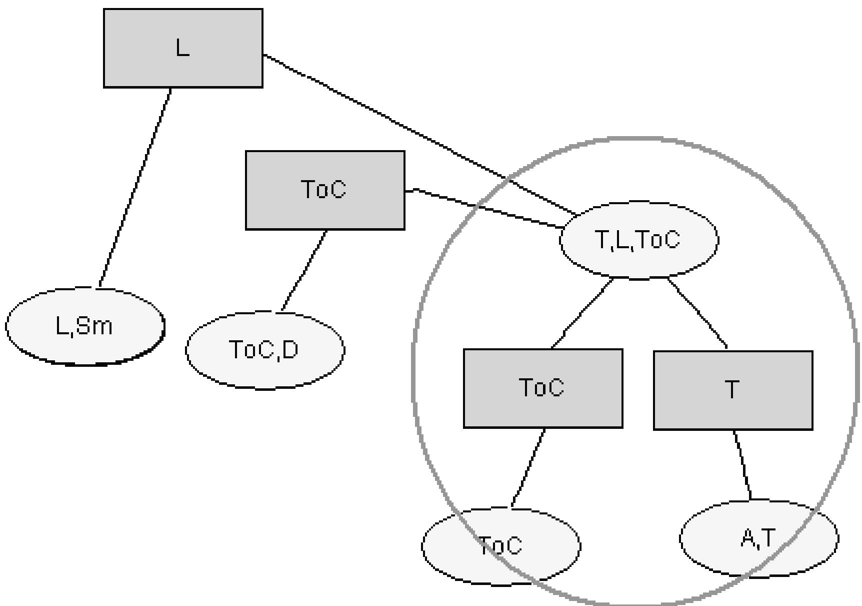


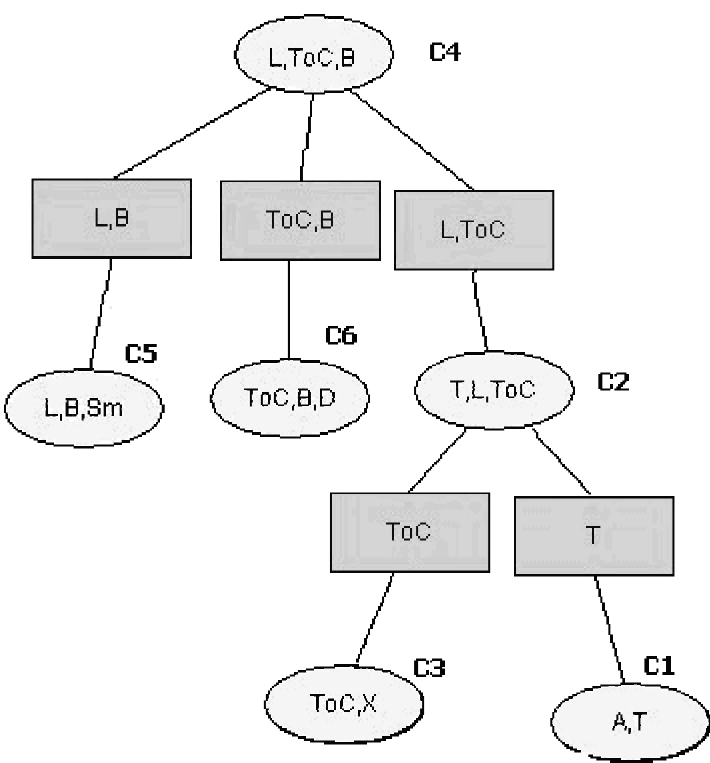
Figure 3- 21. Junction tree of Asia without B, marked parts are exactly the same as the initial tree.

We can see that the tree is not the same, but it is neither completely different. Marked part in *Figure 3-21* shows which ones are the same. Clique {L,Toc,B} has disappeared. {L,B,Sm} is reduced to {L, Sm}, {ToC,B,D} is reduced to {ToC,D} and since {L,ToC,B} does not exist yet, it is linked to {ToC,L,T} by L.

Even here where it seems to be quite different from the original tree, it deals with the disappearing of B in cliques (obviously, B is no more there) and the reduction of those cliques where B participated.

A methodical way to see it in this tree could be the following one:

Let see the original tree with the cliques numbered to be able to refer to them:



We start by one clique, for example C4. As we have deleted B the clique remains as L,ToC. But L,ToC is exactly the same as the separator $S_{2,4}$. So, we do not need it any more and we can remove both of them. But, if we remove C4 then C5 and C6 are automatically joined to C2. Doing that $S_{5,2}$ turns to be L and $S_{6,2}$ T. (B has to disappear also in C5 and C6). And then, the tree is the one shown in *Figure 3-21*.

So here we can conclude that the proposed solution of just deleting the node in the JT sometimes leads to the “best” tree (example 4.1.2.⑤), but that this is not always the case (earlier examples 4.1.2.③ and 4.1.2.④).

We have to say that here a cycle has been broken. Maybe that will have special consequences. We will see it in the point of edges deletion (4.2.2).

- *Proposed solutions*

In the examples we can realise that changing a variable which is a child or a father of only one other variable will simply prune the branch in the junction tree where it appears. And we can be sure that this clique will be a leaf in the junction tree, we know that the variable is only connected to one other variable.

Taking this idea, we can think that for these situations a possible solution can be deleting the clique in the junction tree that this variable forms if it is a binomial clique, or reducing it if the size is more than 2.

But, if we go further to a variable with two links (they can be incoming, outgoing or one of each), we have found a way to avoid the whole recompilation. It consists in eliminating the variable in the junction tree of the original network. In this manner we will save time of triangulation. But even more, we do not have to reconstruct again the whole tree. There is a solution to treat only the necessary part. We have already described it. The point is to delete it in all the cliques where it appeared. If that reduces a clique to his separator both of them are removed and the tree merges by this place. Actually, this solution is the same as the presented for the previous case. The reason is that when we eliminate a variable from a clique of size 2 the separator must be eliminated too, if it contained this variable, or absorbed, if it contained the other one. There is no other possibility because the separator must be a subset of the clique.

It is important to remark that this solution does not always give the optimal tree, as we have pointed out with the examples before. Although we do not say that a better solution could not be reached giving simpler trees, we do consider that this solution is satisfactory in the sense that it lets us do what we wanted at the beginning: obtain a valid junction tree for a modified Bayesian network without recompilation and assuming not too much time. An idea could be to inspect more exhaustively the different cases in the more simplified trees. But, maybe this will take us to a deeper study of the tree. This study can mean spending more time, when our main purpose is to be as fast as possible.

Looking ahead, a generalisation to one deleted variable with more than two associated edges can be interesting. In the line of this chapter, analyse by examples, Asia does not offer the possibility to do complete experiments further with deletion of edges. Although there exist nodes with more than two links, we cannot find all possibilities (3 parents or 3 children for example).

Anyway, with the progressive study we have done, we dare to guess that the solution will be quite similar to this one. Maybe then the resulting tree will be even less “optimal”, but as before, still valid, what is the most important point. And the lack of optimality will influence less in the global process if the network is large.

We cannot demonstrate it, but in this project we would want to introduce the subject and to encourage future studies and evaluations.

4.1.3. Addition.

Finally, here we have the last alternative for variables. Instead of finding that a variable is not necessary, the opposite can happen too. Going deeply into the problem to solve, or better, to model, we could notice we have forgotten one important aspect. This aspect could be represented by one or more new variables. It seems likely that adding a variable will mean adding new edges, we would want to relate in some way this new variable to the previous ones. At this point we can ask: when are we going to recompile? Just after adding a new variable or when all the new elements we find necessary are included. It appears that the second option is more reasonable, we will not want to recompile until we will have introduced every modification in the Bayesian network. Adopting this option, adding a new variable will lead us directly to the later point 4.2.1 (adding edges). However, we will try to examine the case of introducing a new variable alone.

- Possible consequences of changing variables adding a new variable

Although adding an isolate variable seems to be a trivial situation, it could be interesting looking at it. If this new variable is linked to no other one, it will form a clique itself. Like that, in the junction tree there will be one node disconnected to the rest. In propagation this node is not going to participate, since it has no mean (no separator, the intersection between it and the rest is the empty set) to communicate with the rest. This new clique (made up of only the new variable) will keep the initial belief probability, for it will never affected by the other ones.[1]

4.2. Edges.

An edge alteration seems to be quite similar to the variable one. It is due to the fact that both of them will probably take us to a different moral graph, or maybe a bit later to a different triangulation, and, like that, the junction tree will also change. Once more, the point is to identify which are the thinkable variations carrying to a set of diverse types of consequences.

4.2.1. Addition.

If we add an edge, E will have a new element. The first effect will be on G^M. If this new edge or link points to a variable with one or more other parents, then in the moral construction all these variables must be joined to the variable pointed to by the new edge. That will probably produce a new triangulation, but there may be a possibility that it does not: if the new edge and those provoked by the moralisation already belonged to the initial triangulation. So, there is another case we should be able to tell apart.

Let see the possibilities of adding one new edge in a more detailed way:

- New edges between existing variables:
  - If the new edge does not introduce other new edges in moralisation apart from itself (and it already belonged to G^T) then the result is trivial, because G^T will be the same, and so will be the junction tree. Or if the new edges changes the moral graph but keep the same G^T. In any case, we cannot forget that one new edge will always imply a change in its children, the conditional probabilities change.

Example 4.2.1.①.
Imagine that we put a new edge from “Has lung cancer” to “Has bronchitis”.

[1] That if we do not take into account the concept of dummy separators, used to avoid a disconnected graph. The only reason to introduce dummy separators would be the fact of a finding of it. By now, we are going to omit this.

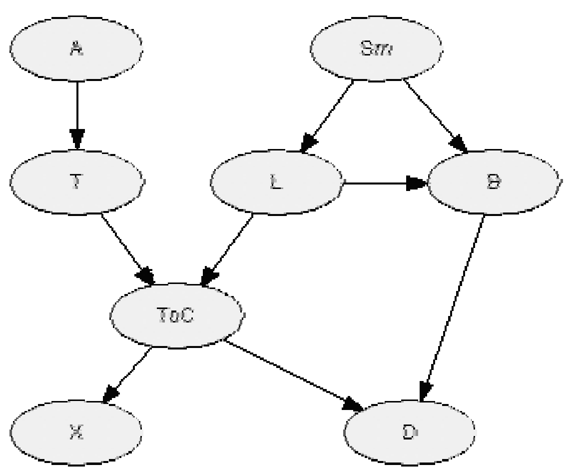


Figure 3- 22. Asia with link from L to B.

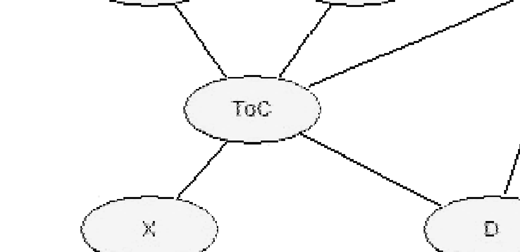


Figure 3- 23. Asia with link from L-B moralised and triangulated (the moral graph was already triangulated).

What happens then? In fact nothing happens to the compilation process, because G^T does not change and the junction tree is exactly the same. The new link will put a moral link, but this one was already in the graph. But we have to change conditional probability for B, because now it is P(B|Sm,**L**).

- If the new edge introduces other new edges at moralising time and some of them are not in G^T, then this triangulated graph may be different. That would imply different cliques and consequently a different tree. Studying this case can be quite difficult, it depends on the situation of the new edges. We are going to show one possible situation in Asia.

<u>*Example 4.2.1.②.*</u>

A new edge from T to L.

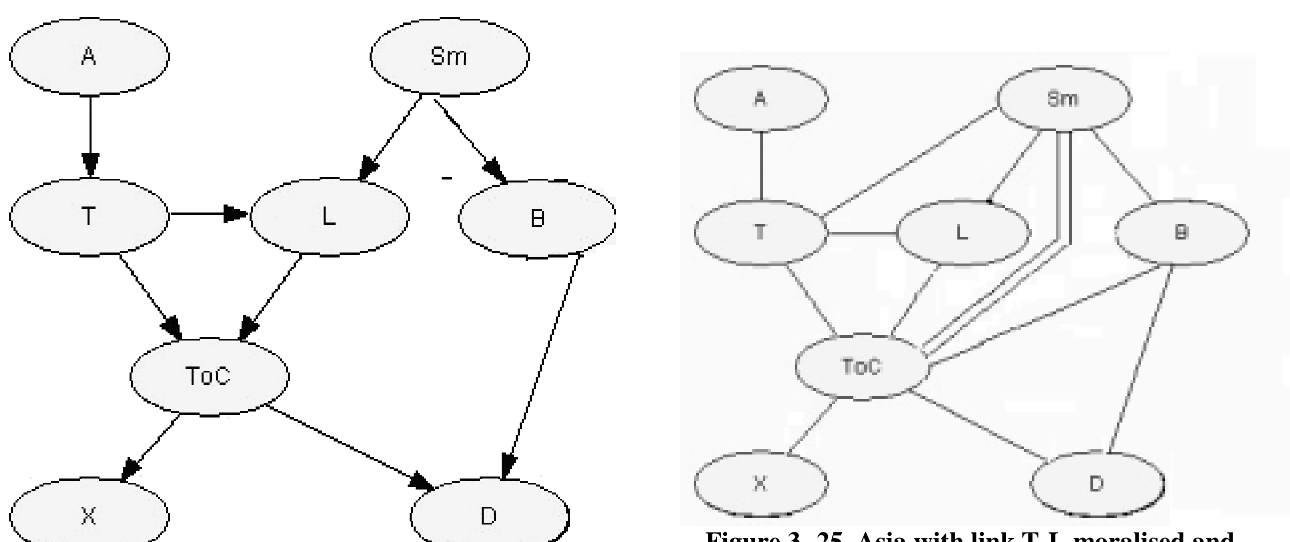


Figure 3- 24. Asia with link from T to L.

Figure 3- 25. Asia with link T-L moralised and triangulated. Sequence: {A,X,D,T,B,Sm,L,ToC}. The elimination of T will provoke the fill-in.

With the numbering A←1, T←2, L←3, ToC←4, Sm←5, B←6, D←7, X←8, we obtain the tree:

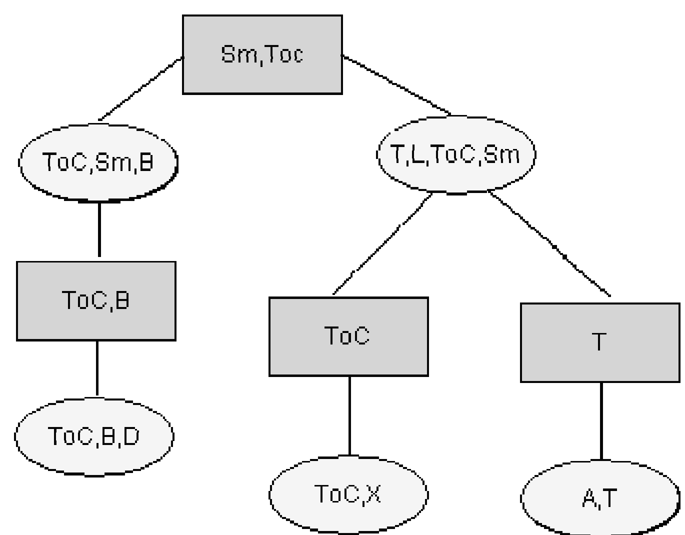


Figure 3- 26. JT of ASia with link from T to L.

The differences between this tree and the original one for Asia seem to be more difficult to analyse. One link will probably add new moral links, and these ones will give a quite different triangulation like in this example, where the fill-in {Sm, ToC} has never before appeared in the other cases, it could have appeared with another elimination order. So, we think that this is not at all very predictable. It seems that maybe this is one of the cases without a clear solution, that is, where incremental compilation cannot be used, and a new compilation should be completely done.

- Add an edge pointing from an existing variable to the new one. This is the simplest case again, since the new variable potential only will influence on itself.

Example 4.2.1.③.

Let suppose we will reconsider the possibility of a new variable whose father will be X.

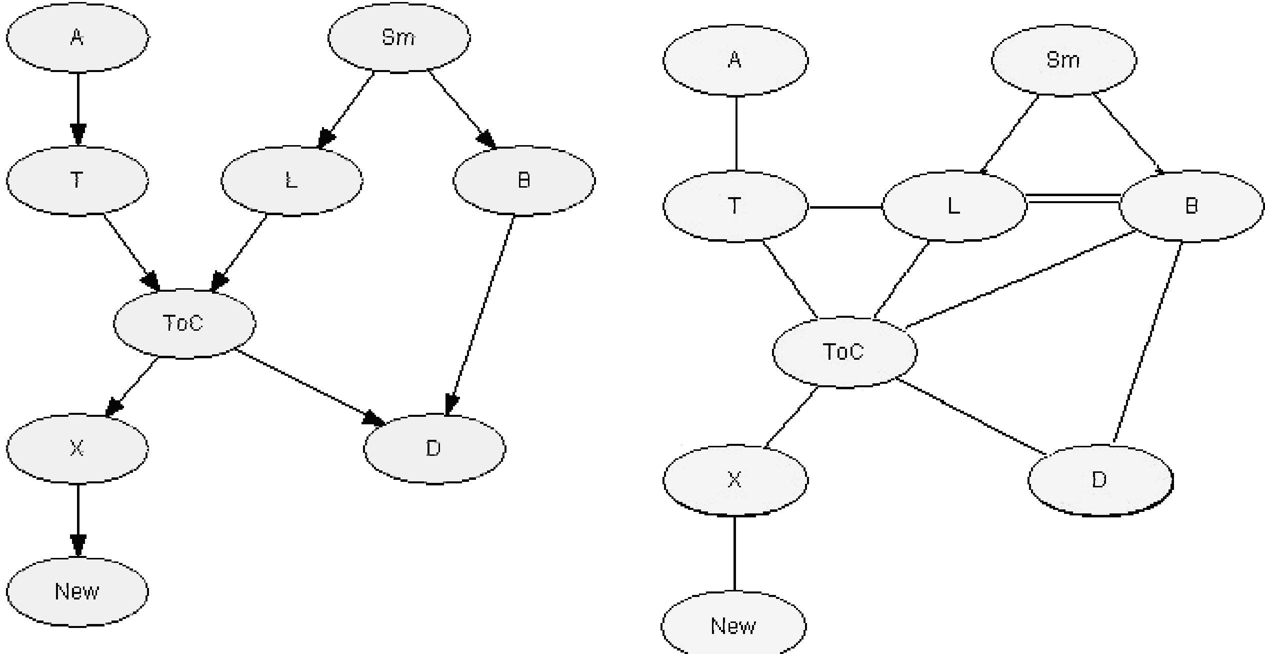


Figure 3- 27. Bayesian network for asia after adding a new variable New child of Positive X-ray?

Figure 3- 28. Asia with New child of X moralised and triangulated. Sequence: {A,T,New,X,D,Sm,B,L,ToC}

And with the numbering: A←1, then: T ←2, L ←3, ToC ←4, B←5, Sm ←6, D←7, X ←8, New←9, we obtain the tree:

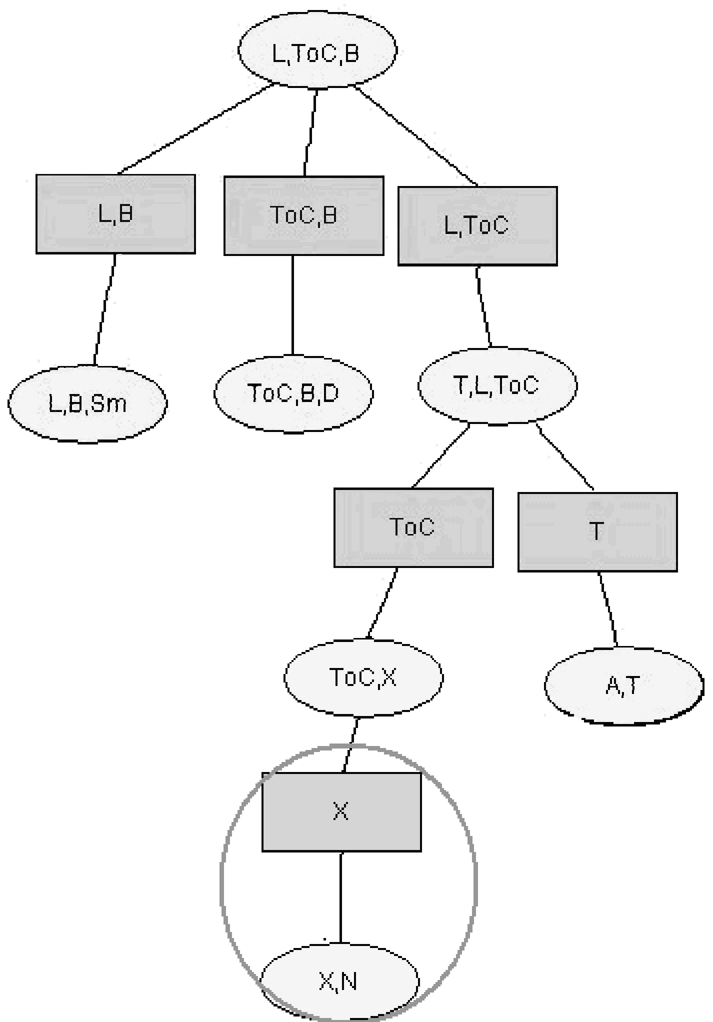


Figure 3- 29. Junction tree for Asia with the New variable pointed by X. (Here the name New is shortened to N)

So, after seeing this example, we could think that adding a new variable and making this one to be pointed at by an existing one makes that after recompilation our junction tree grow in one branch.

- Add an edge pointing from the new variable to an existing one.

Here we can distinguish two cases: 1.- the new one is the only father of this existing variable; and 2.-there are already other existing variables pointing to the same one, that will provoke a *marriage* between New and these other parents, altering G^M, and probably implying more changes in the junction tree. Let us see two other modification examples to illustrate these two cases.

<u>Example 4.2.1.④.</u>

We decide it is necessary a new variable and this one will have a causal relation to Asia? (A).

The Bayesian network would be as follows:

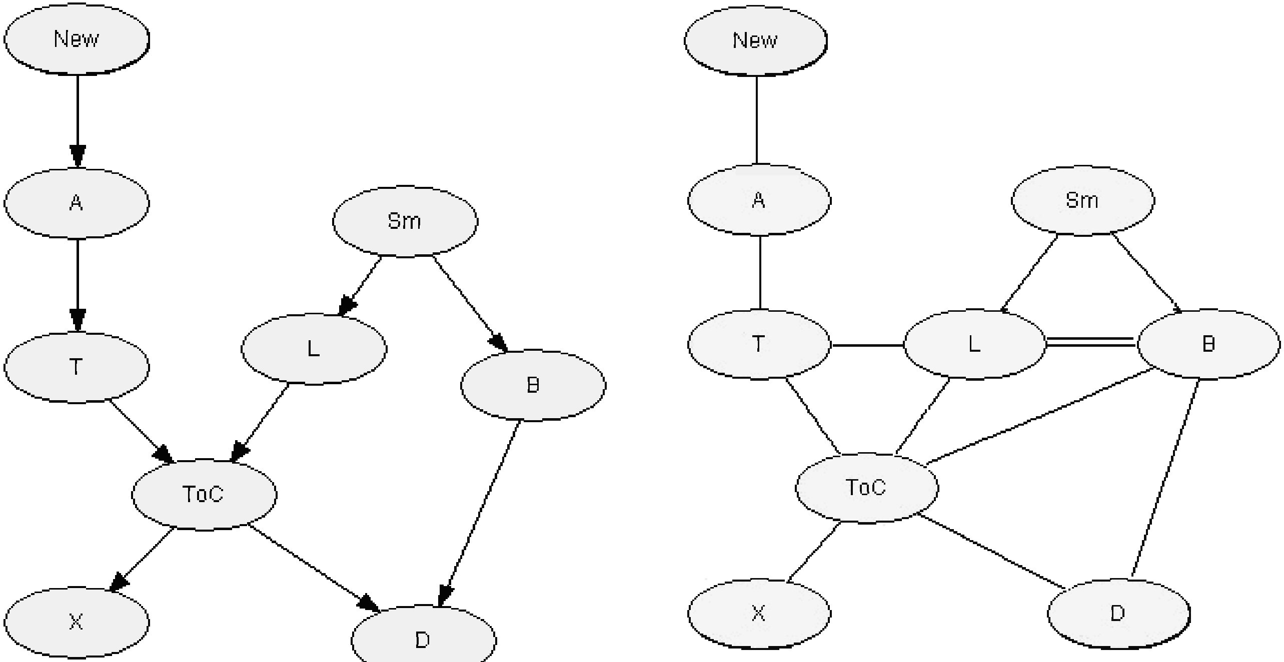


Figure 3- 30. Asia with a New variable pointing to A

Figure 3- 31. Asia with a New variable pointing to A moralised and triangulated. Sequence: {New,A,X,D,Sm,B,L,ToC}

And with the numbering New←1, then: A←2, T ←3, L ←4, ToC ← 5, B← 6, Sm ← 7, D← 8, X←9, we obtain the tree:

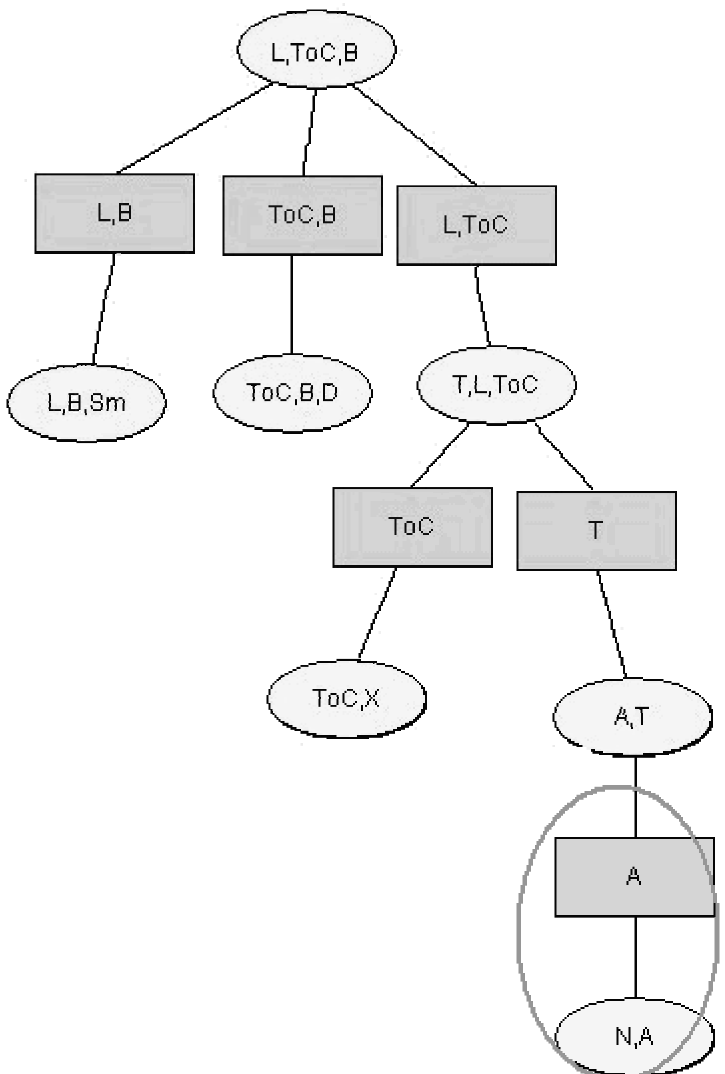


Figure 3- 32. Junction tree of Asia with New pointing to A.
(New is shortened to N)

The fact that there is a new leaf is observed. A appeared before in one leaf, since it was connected only to T. As the new one is also only connected to A, it should be joined to its clique by a new separator. So, in this case, we could say the addition of a new variable pointing to an existing one, when this is the only father will conduce to an extension of one extreme of the tree.

But now, we are going to complicate it a little bit. In the next example, the new variable will point to B, which is already child of Sm. Let us show what happens.

Example 4.2.1.⑤.
We decide it is necessary a new variable and this one will be pointing to B.

So, this time our network looks like:

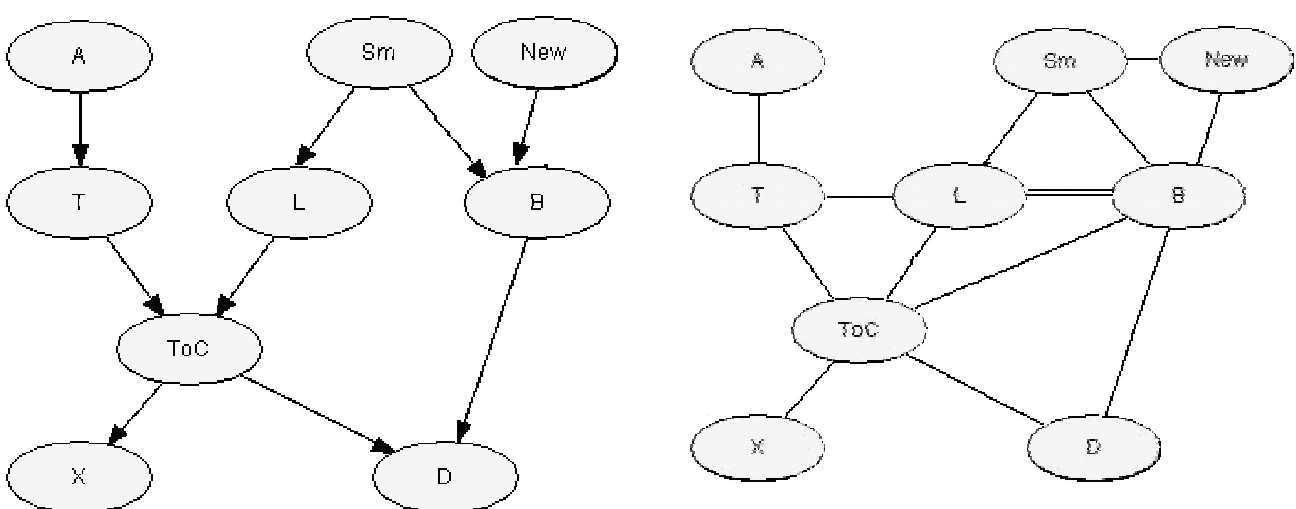


Figure 3- 33. Asia network with a new variable New pointing to B.

Figure 3- 34. Asia network with new parent to B moralised and triangulated. Sequence: {A,T,X,D,N,Sm,B,L,ToC}

With the numbering: A←1, then: T ←2, L ←3, ToC ← 4, B← 5, Sm ← 6, D← 7,N←8, X ←9, we obtain the tree:

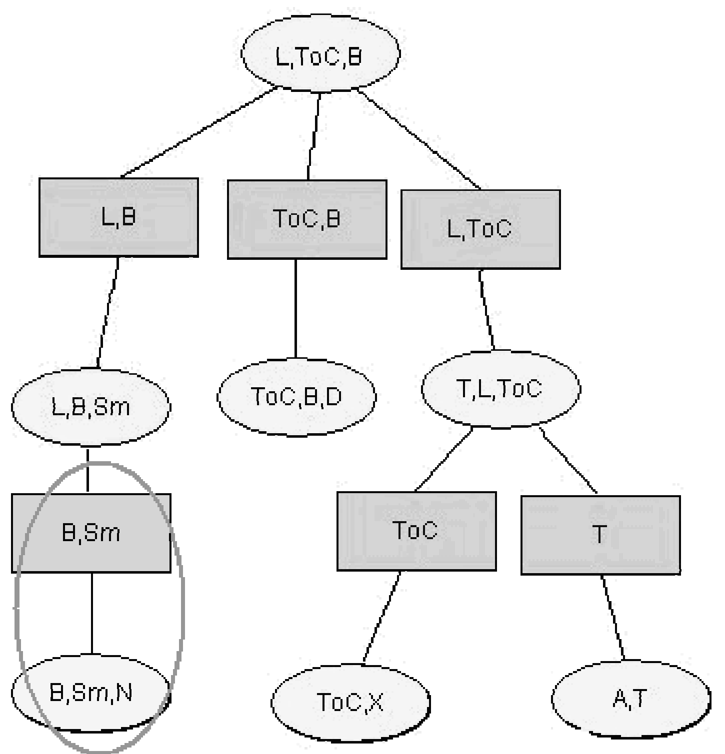


Figure 3- 35. Junction tree Asia network with a new variable

The result is quite similar to the case before, but now, the new clique is bigger, that is because in addition to the new edge between N and B, we will add the moral edge Sm and N.

- *Proposed solutions*

In the first case (new edge already in G^T that does not introduce new edges in G^M) the solution is very simple, we only have to update the potentials of the new child and those in the junction tree.

In the second case (new edge between existing nodes that leads to a different G^T), we have concluded we recommend recompiling, maybe other methods could be studied, but we have not reached them.

And finally, a new variable with a new edge. As we have seen, adding a new edge that joins an existing variable to a new one will imply one new clique in the tree. This clique will be formed by the new one, by the one connected to it, and maybe by other variables if moral edges have been added.

This is quite similar to deleting a variable, but in the opposite direction, that is, now we should add a new clique to the tree. This clique will also be a leaf in the tree. We need some method to detect which clique and where to locate it.

As a child will always be connected to its other parents there will always be a clique with all of these nodes and the new one cannot be connected to others in G^M. Therefore, this will always result in exactly one new clique that should be connected to an existing one. However this existing clique might be a subset of the new one, resulting in an extension of the existing rather than the addition of a new one.

So, in general adding a new variable with only one new link will take us to the creation of a new clique or to add one component to an existing clique.

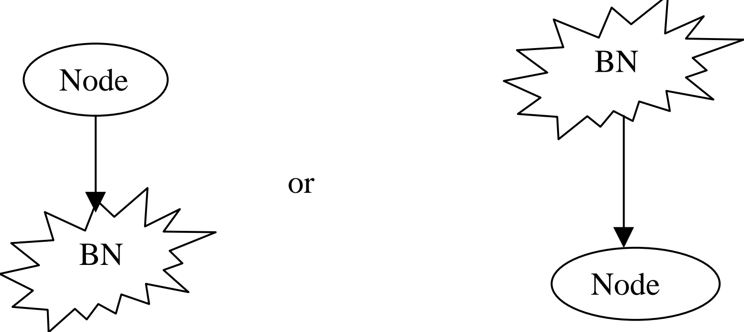
4.2.2. Deletion.

Here something similar occurs. If we decide to remove one edge, it is very probable that the moral graph will change, since the moral graph only contributes with new edges between variables with a common child. If so, once again triangulation can change or not. G^T will be equal if in the process of triangulation this edge reappears and the deleted edge did not take to any other fill-in edge.

It can produce an isolate variable if it is the only edge containing this variable. In this case, it would be a similar case to add a new variable without edges associated. Now this variable will be lost in all the previous cliques where it appeared before.

We will try to see first what happens if we remove one variable only related to another one. The systematic way to take here starts to be a challenging task. Anyway, we propose one method to do it:

- Delete a link with connects one node to the rest of the network. It seems to be the easiest one:



Both cases are alike, but once more, considering that deleting the link from the node "outside" will have more implications in potential tables due to conditional probabilities.

Example 4.2.2. ①.
Deleting the edge from ToC to X

Bayesian network would be:

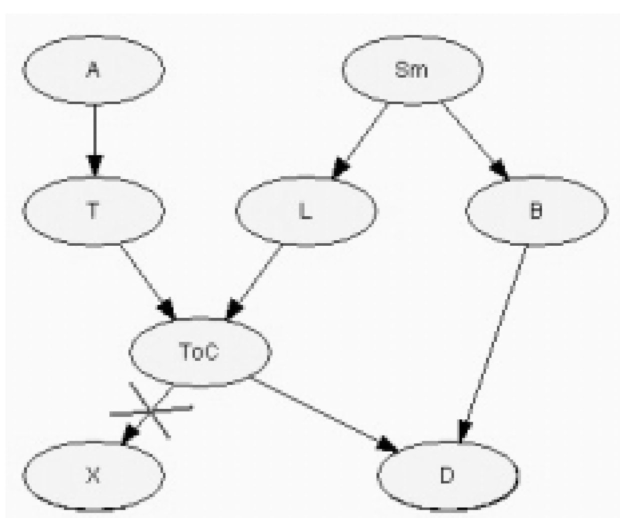


Figure 3- 36. Asia without edge from ToC to X

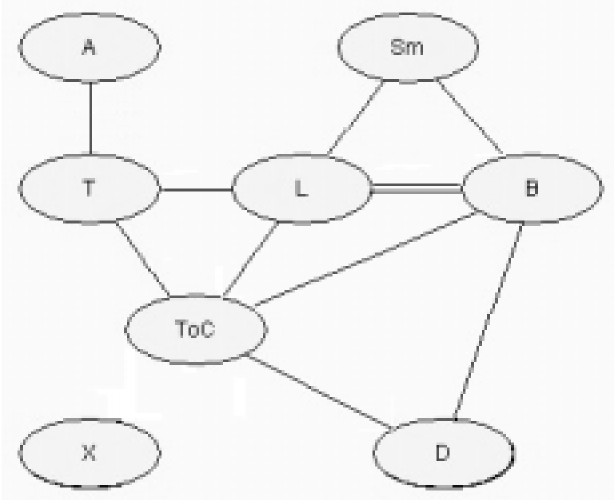


Figure 3- 37. Triangulated graph of Asia without edge from ToC to X

→ Isolated variable.

Now, the other possibility is to delete a link "inside" the BN, that is a link which will not isolate one node from the network. In that case, we distinguish between those breaking a cycle or not.

The reason for choosing this election is quite simple and after analysing some cases it fitted for the results. We know that a graph is triangulated if and only if every cycle of length greater than 3 has a chord. So cycles will provoke fill-ins in the triangulation process. For that, removing a link which participated in a cycle will probably suppose changes in triangulation and in fill-ins.

But even further, the bigger the broken cycle is the more this deleted edge will affect on the new G^T and therefore on the junction tree.

- Those which do not break a cycle

We are going to take the edge from T to ToC, one that fulfils the condition.

<u>*Example 4.2.2.②.*</u>
Asia without edge from T to ToC.

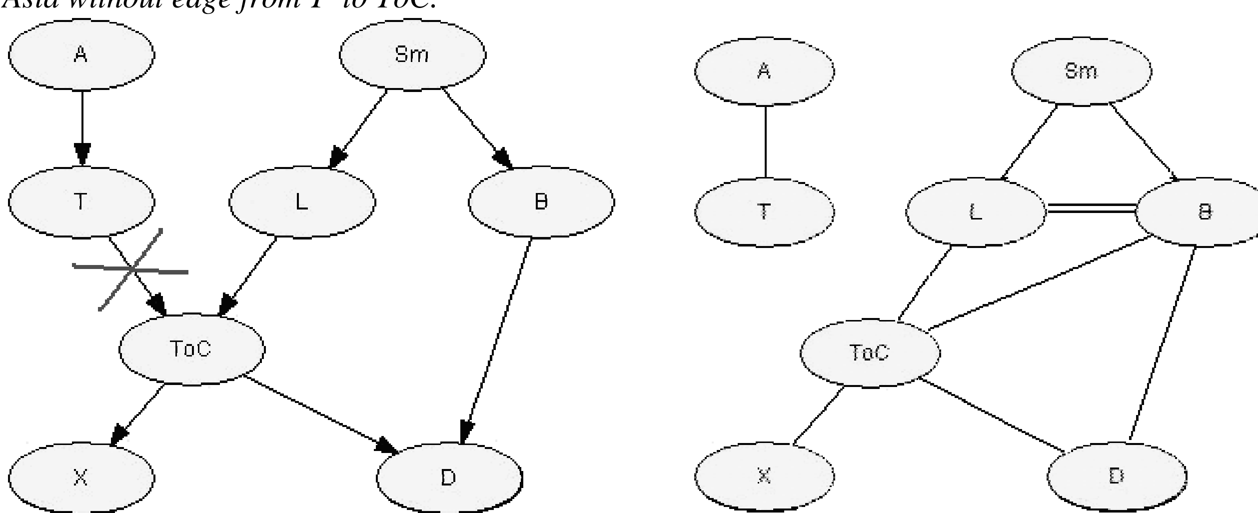


Figure 3- 38. Asia without edge from T to ToC.

Figure 3- 39. Asia without edge from T to ToC moralised and triangulated. Sequence: {A,T,X,D,Sm,B,L,ToC}

With the numbering A←1, and then T←2, L←3, ToC←4, B←5, Sm←6, D←7, X←8, we obtain the tree:

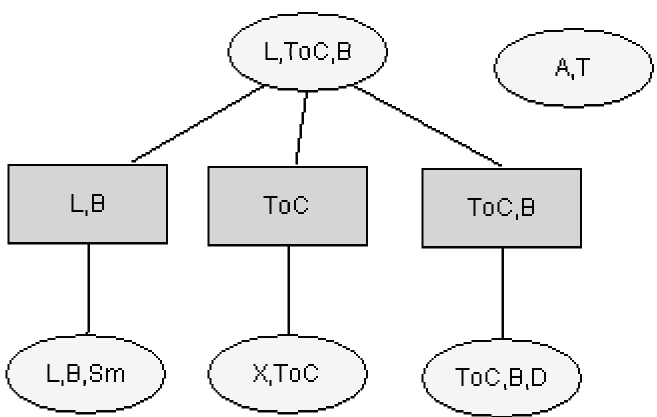


Figure 3- 40. JT for Asia without edge from T to ToC.

As we can see it differs from the original in several cliques, and besides there is an isolated one, A,T. Here it can arrive to think about doing the same thing as we did in elimination of nodes, that is, keep the moral and fill-in links from the original network. But we have not found an easy way to reach the new JT either. It could be detecting cliques with the two extremes of the edge implicated, but this will also affect on other leafs hanging from this one.

- Breaking a cycle.

Example 4.2.2.③.
Asia without edge from B to D.

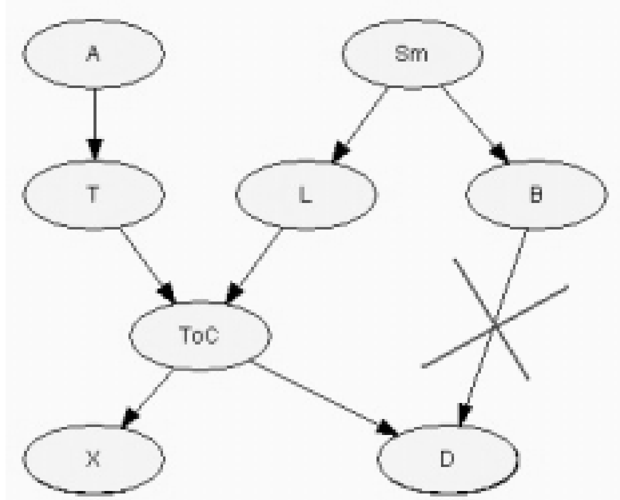


Figure 3- 41. Asia without edge from B to D.

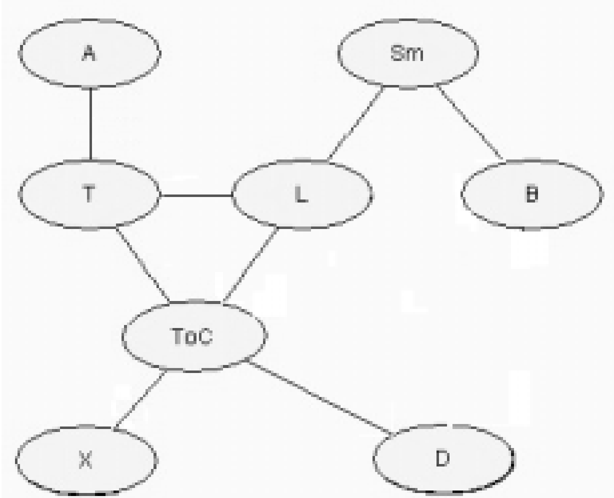


Figure 3- 42. Moral graph for Asia without edge from B to D. It was already triangulated. Sequence:{A,T,X,D,B,Sm,L,ToC}

With the numbering: A←1, then: T ←2, L ←3, ToC ← 4, Sm← 5, B← 6, X←7, D ←8, we obtain the tree:

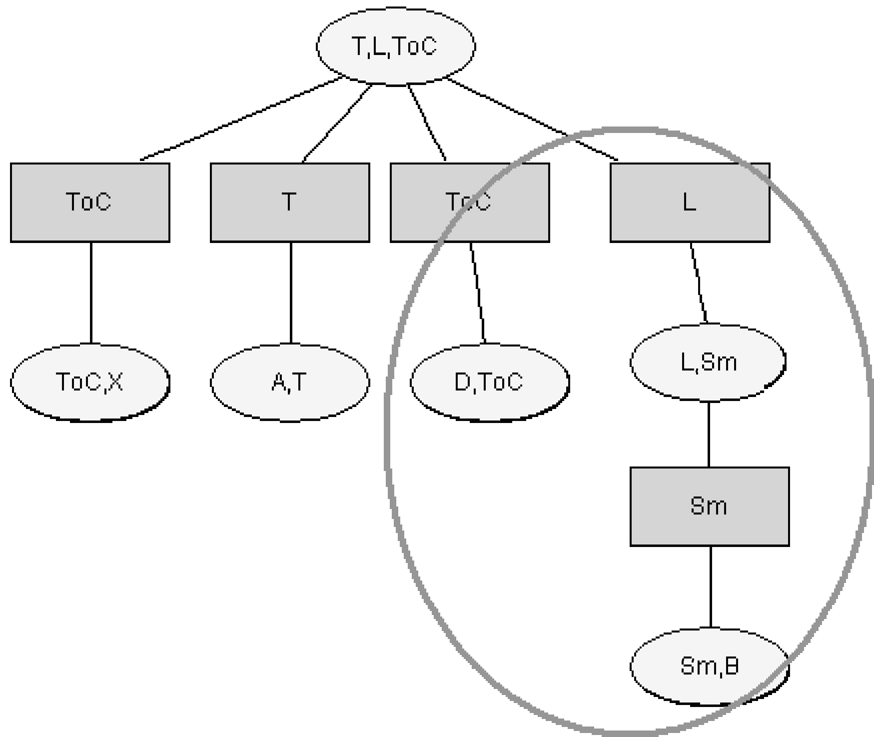


Figure 3- 43. Junction tree for Asia without edge from B to D.

The branch marked is the one that changes. We have deleted the edge from B to D. Then, clique {Toc,B,D} does not exist any more. But, in addition , there is not a fill-in from L to B, and that is the resulting junction tree.

Example 4.2.2.④.
Asia without edge from Sm to B.

We have the following network:

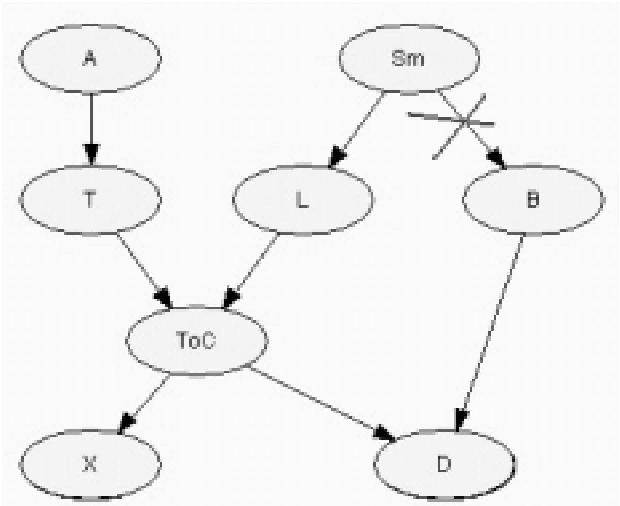


Figure 3- 44. Asia without edge between Sm and B.

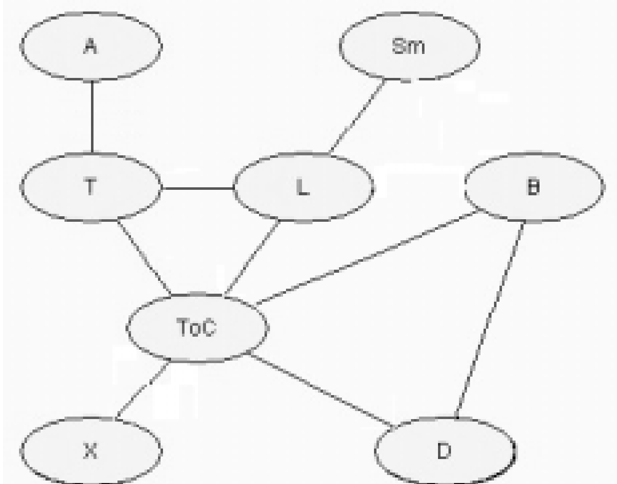


Figure 3- 45. Moral graph from Asia without edge from Sm to B moralised and triangulated (it was already). Sequence: {A,T,X,D,Sm,B,L,ToC}

With the numbering: A←1, then: T ←2, L ←3, ToC ← 4, B← 5, D← 6, Sm←7, X ←8, we obtain the tree:

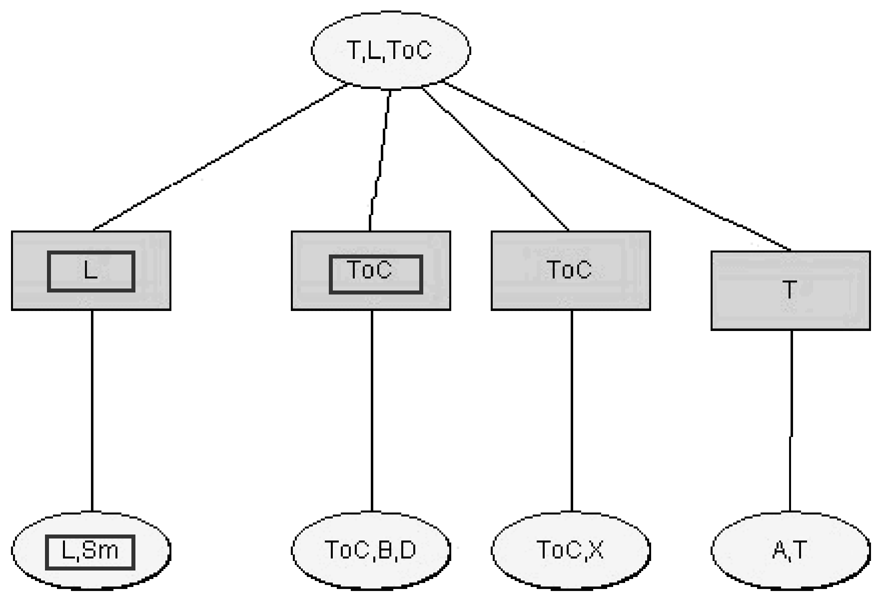


Figure 3- 46. Junction tree of Asia without edge from Sm to B.

Here, for example, we can think that if link from Sm to B is removed, then there will not be a fill-in (L,B) what leads to the loss of cliques {L,B,Sm} and {L,ToC,B}.

- *Possible consequences of deleting an edge*

This case is maybe one of the most difficult. If one edge disappears the changes are bigger because probably this will mean important changes in G^M. The only two simple cases we can think about is the first one where a variable turns to be isolated (this modification seems to be little probable, for what do we want a variable if it is not related to any other one?) and the second one, deleting one edge which will reappear afterwards in the triangulation process. Then the compilation will turn to be the same.

If none of these two cases take place, then the conclusion starts to be a bit confused. The key is to be able to determine in what fill-ins and later in what cliques this edge participated. Like this, we can probably know which cliques disappeared and what other cliques can raise. This study starts to be a hard work.

- *Proposed solutions:*

As we have just discussed, apart from the two easiest cases, the other seem to be quite difficult to solve here. So, we will try to see if we can treat these cases with a different method: Maximal Prime Subgraph Decomposition of Bayesian Networks. We will deal with it in the following chapter.

As we told before (section 4.1.2.), a deeper analysis can be done looking into the number of deleted edges and their influence on the G^M and G^T graphs. Once again, we cannot attack all the possibilities in this project. At least we have introduced a representative set of them and a hint about tricks to execute an incremental compilation.

5. Discussion.

In this chapter, we have elaborated a methodological study about the main alterations inside a Bayesian network that can occur in a revision task. These alterations go from the easiest one and deal with every component inside a network.

This study has shown a progressive inspection of the possible modifications, the consequences that each of them can provoke and finally the most important question, in some of the cases we have been able to state a practicable solution to tackle the problem of recompiling, which is the purpose of this work.

To try to study some of the unsolved cases here next chapter will show a different method that we may use in order to do the partial compilation we are looking for.

Chapter 4. USE OF MAXIMAL PRIME SUBGRAPH DECOMPOSITION IN INCREMENTAL COMPILATION OF BAYESIAN NETWORKS

1. Purpose of using this method.

In the previous chapter we have tried to solve our problem from the original junction tree. Repeating the figure of the introduction, we made an attempt to obtain the new junction tree using as a basis the original one:

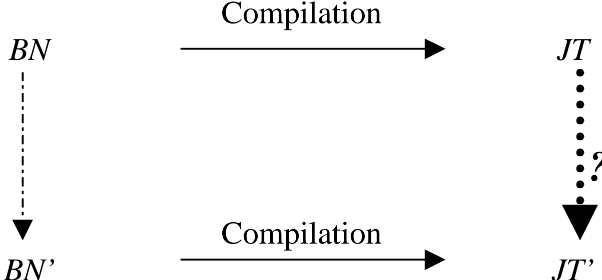


Figure 4- 1. Drawing explaining the process of incremental compilation of a BN.

Nevertheless, we found some cases where this solution was not practicable. But, we can think about using another mechanism, probably a bit more sophisticated. In the article about Maximal Prime Subgraph Decomposition of Bayesian networks [Olesen and Madsen 1999] the idea was launched.

In the said article the concept of a Maximal Prime Decomposition tree (Γ_{MPD}) is explained. In section 2 of this chapter we will give a more detailed description about it. At this introductory level we would only say that it is an intermediate structure we place between one Bayesian network and its associated junction tree. As we will see this Γ_{MPD} can be extracted from the "normal" junction tree. So, the idea is to go one step backwards to reach Γ_{MPD} from the original junction tree JT. And then, taking Γ_{MPD}, we are going to determine a way of "translating" it to the new $\Gamma_{MPD'}$. So, with the new $\Gamma_{MPD'}$ and a certain knowledge (that we have) about the changes in the new Bayesian network BN', we can arrive to the new junction tree JT'.

Then, as means to illustrate this new strategy for partial compilation we will employ a similar scheme to the original one in *Figure 4-1*. Like that we can perceived the path we follow to avoid a total recompilation of the network.

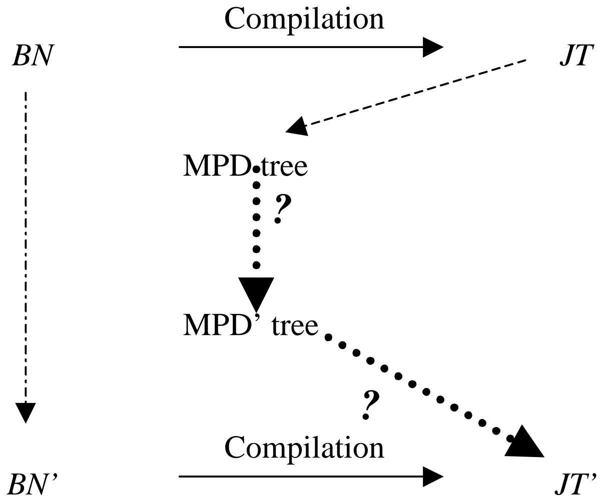


Figure 4- 2. Drawing explaining the process of incremental compilation of a BN by means of the MPD.

As the quoted article says, and as we will explain in the following points, the tree of maximal prime subgraphs will allow us to re-triangulate only determined parts of the network, and this feature is quite attractive in order to save computational time as well as effort, specially if the Bayesian network is large.

2. Presentation of Maximal Prime Subgraph Decomposition.

To use this new structure, we should first present it and its most important characteristics. Let G=(V, E) be a graph. A subgraph G(V') is *maximal* with respect to the property p if V´ is not a subset of some larger set V'' that induces a subgraph G(V'') with the property p. A maximal complete subgraph is called a *clique*. If the nodes V of a graph G can be partitioned into a triple (V',S,V'') of non-empty sets, where S is a clique of V' and V'' in G such that every path from a node Y' ∈ V' to a node Y'' ∈ V'' includes a node in S, then G is *decomposable* (or *reducible*) otherwise G is *prime* (or *irreducible*). S is called a *separator* of V' and V''.

A Maximal Prime Sugbraph Decomposition of G is then the identification of all maximal prime subgraphs in G.

After defining what we understand by prime subgraph we will seek to give the principal idea of the Maximal Prime Subgraph Decomposition (MPSD). First of all, taking the previous definition, we would say that the property p we are interested in is primeness.

A MPSD tree is a tree with the same structure as the junction tree. It will also have separators, except that now the cluster nodes are not cliques but maximal prime subgraphs within the moral graph. The reason for choosing the moral graph is that this one will contain the edges that must necessarily belong to the graph, whereas we could find some different ways to triangulate it, that is, several different G^T all of them valid.

It is known that a Bayesian network BN is constituted of a directed acyclic graph G and a set of potentials related to each variable in the graph BN = (G, *P*). So, in compilation process we did:

BN = (G, *P*)   G ⟶ G^M ⟶ G^T

We have just said that Γ_{MPD} is obtained from G^M, for it is formed by the maximal prime subgraphs of the moral graph. Anyway, the process needed to attain to Γ_{MPD} from G^M is not at all clear at this moment. There are some proposed methods in the literature and all of them include some kind of triangulation step.

In [Olesen and Madsen 1999] article the alternative to produce Γ_{MPD} is an easy way that goes through a minimal triangulation T_{min} of the graph G (the triangulated graph will be denoted then by G^{Tmin}). Given a graph G, a triangulation T is minimal if and only if there is no other triangulation $T' \subset T$ such that $|T'| < |T|$. After achieving a minimal triangulation Γ_{MPD} could be obtained from the corresponding junction tree JT. Let see it in the explicative *Figure 4-3*.

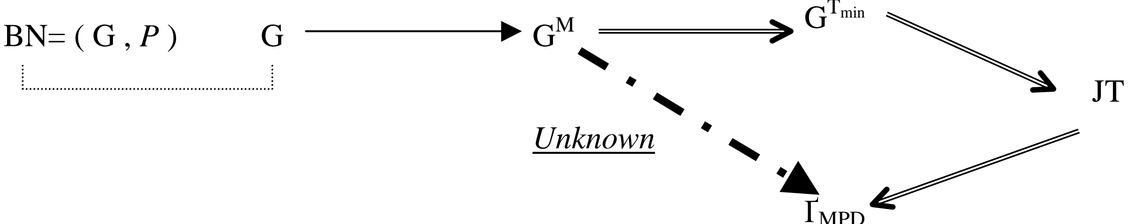


Figure 4- 3. Graphic process to reach Γ_{MPD}. Dashed line shows the unknown direct method from G^M to Γ_{MPD} and the double one represents the method proposed in [Olesen and Madsen 1999] .

For constructing the Γ_{MPD} we will need to identify the different maximal prime subgraphs of G^M. The step from a junction tree to the associated Γ_{MPD} is what [Olesen and Madsen 1999] described. It consists basically in aggregating the cliques in the JT whose separators are not complete. A more concrete explanation will be presented in the next section.

3. Procedure.

Having defined the structure of the Γ_{MPD} and explained a little bit about how to obatin it, we go on describing the algorithm of this method.

The idea is that the MPD junction tree produced can be stored as an intermediate data structure that fits in between the moral graph and the triangulated graph. And the method proposed in [Olesen and Madsen 1999] to identify these maximal prime subgraphs inside a Bayesian network uses the junction tree. For that we presented *Figure 4-2.*

Until now, the style used to describe a process has been a bit informal, trying to give a detailed explanation, but using natural language. However, in this part we will use an algorithmic form due to the significance of this new method in the development of this project. We really think that precision is required to show how it works exactly.

To reach the maximal prime subgraph decomposition tree Γ_{MPD}, we are going to construct it by aggregation of cliques connected by separators that are incomplete in the moralised graph.

[Algorithm 1] Construction of a MPD Junction Tree

- Input: A junction tree Γ obtained from a minimal triangulation T_{min} of a Bayesian network BN= (G, *P*)
- Output: a MPD Junction tree Γ_{MPD}

Step 1.- *T' ← T*
Step 2.- *Repeat*

*(a) Take a separator **S** of T' connecting C' and C''.*
(b) If $G^M(S)$ is not complete then aggregate C' and C'' in T'.

Until no separators S of T' such that $G^M(S)$ is incomplete exists.

Step3.- *Return T'*

In the article there are some properties of this new structure with their demonstration. We are not going to reproduce them, but if we use any of these properties the associated point will be referred.

As we mentioned before, a precondition to use this method is a minimal triangulation of G^M. There are several methods for finding minimal triangulation, for example the LEX-M Algorithm. Following the method of the aforementioned article we will use the recursive thinning method to assure minimal triangulation when, as in our case, the triangulation algorithm is not guaranteed to produce minimal triangulations.

First of all we describe:

[Algorithm 2] Recursive Thinning:

- Input: An undirected graph G =(V,E) and a triangulation T of him.
- Output: A minimal triangulation T_{min} of G.

Step 1.- $G \leftarrow (V, E^M \cup T)$, $R \leftarrow T$
Step 2.- $R' \leftarrow \{e_1 \in T \mid \square e_2 \in R \text{ s.t. } e_1 \cap e_2 \neq \emptyset\}$
Step 3.- $T' \leftarrow \{\{X,Y\} \in R' \mid G(adj(X) \cap adj(Y)) \text{ is complete in } G\}$
Step 4.- *If* $T' = \emptyset$ *then return T else*
(a) $T \leftarrow T \setminus T'$
(b) $G \leftarrow (V, E \cup T)$.
(c) $R \leftarrow T'$
(d) *Goto* ***Step 2***.

(So, the technique is to keep only fill-ins whose variables adjacency sets have a non-complete intersection.)

And now, we will write the final algorithm to follow:

[Algorithm 3] Constructing the Maximal Prime Subgraph Decomposition Junction Tree.

- Input: A Bayesian network BN= (G, *P*)
- Output: A maximal prime subgraph decomposition junction tree $Γ_{MPD}$

Step 1.- *Moralise G to obtain G^M.*
Step 2.- *Triangulate G^M to obtain G^T.*
Step 3.- *Thin out redundant fill-in edges with Recursive Thinning*
Step 4.- *Organise the clique decomposition induced by G^T as a junction tree Γ.*
Step 5.- *Construct the MPD junction tree $Γ_{MPD}$.*

To see better the problem we are going to show an example. And to pursue the same line along all this project Asia is the best choice. First, we are going to construct its maximal prime subgraph decomposition junction tree. Let us take the algorithm step by step.

Asia Example

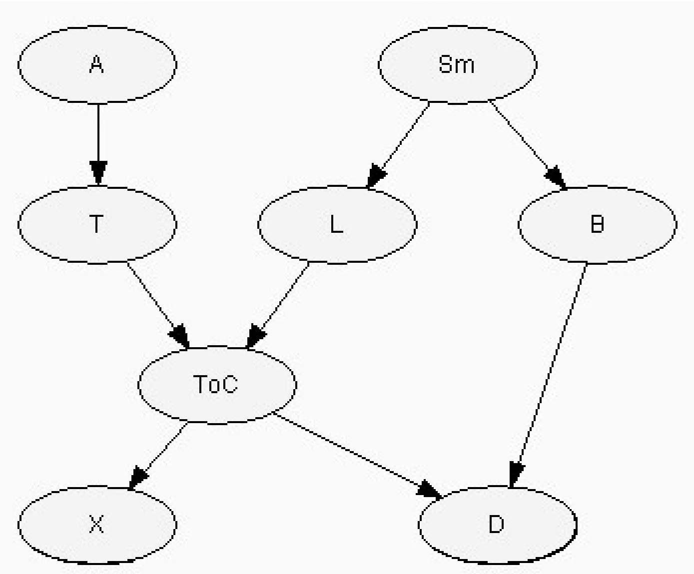


Figure 4- 4. Bayesian network Asia.

- Steps 1 and 2 are already done in Compilation Chapter

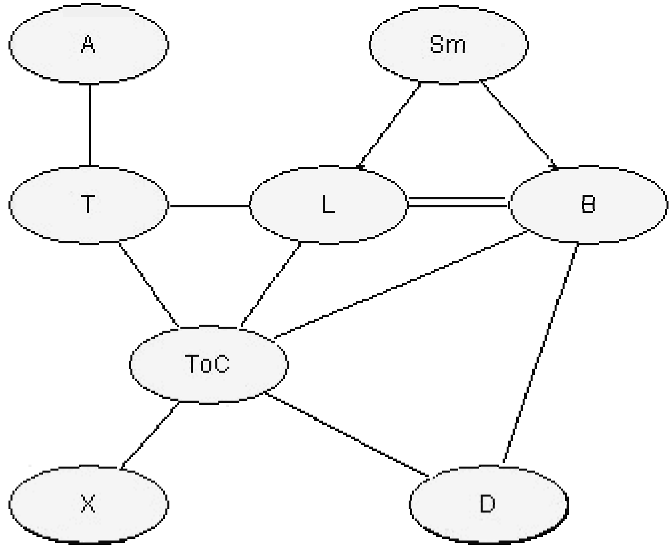


Figure 4- 5. Asia moralised and triangulated.

- Step 3 → Recursive thinning. We had the fill-in set {L,B}

Step Action
[1] -- R = {L,B}
[2] -- R' = Ø (because there is no other e_1)
[3] -- T' = Ø
[4] -- Like T is Ø we return the initial and only fill-in {L,B}.

It seems quite visible that this triangulation was already minimal, due to the small size of the graph, but anyway we wanted to show it.

To see that this algorithm would eliminate unnecessary fill-ins we could invent a non-minimal triangulation. For example we could add {A,L} and {A,Sm}. In fact, we do not have an elimination ordering to obtain these fill-ins, but the question is to see that the algorithm works. So, let see this example:

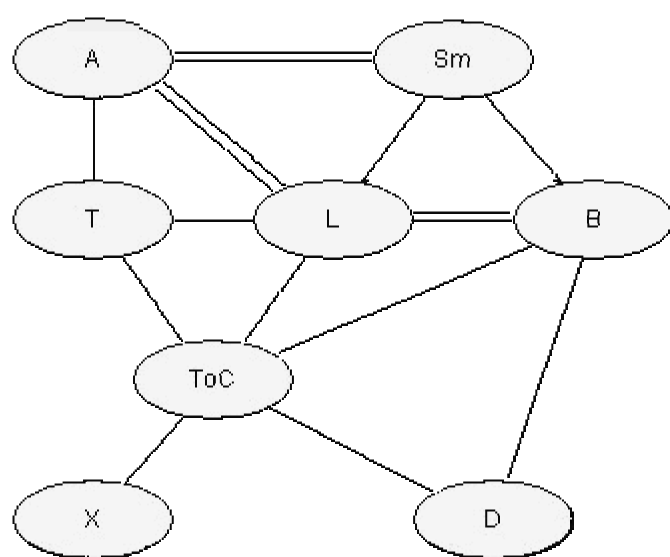


Figure 4- 6. Invented triangulation of Asia for an example of Recursive Thinning algorithm.

T = {{A,L},{A,Sm},{L,B}}

Step Action
[1] -- R = {{A,L},{A,Sm},{L,B}}

[2] -- R' = {A,L} *({A,L} ∈ T and □ {A,Sm} ∈ R s.t. {A,L}∩{A,Sm}=A ≠ Ø)*

+

{A,Sm} *({A,Sm} ∈ T and □ {A,L} ∈ R s.t. {A,Sm}∩{A,L}=A ≠ Ø)*

+

{L,B} *({L,B} ∈ T and □ {A,L} ∈ R s.t. {L,B}∩{A,L}=L ≠ Ø)*

= {{A,L},{A,Sm},{L,B}}

[3] -- T' =

- {A,L} → adj(A) ∩ adj(L) = {T,L,Sm} ∩ {T,A,Sm,ToC,B} = {T,Sm}
  Is it complete in G? No

- {A,Sm} → adj(A) ∩ adj(Sm) = {T,L,Sm} ∩ {A,L,B} = L
  Is it complete in G? Yes

- {L,B} → adj(L) ∩ adj(B) = {T,A,Sm,B,ToC} ∩ {Sm,L,ToC,D} = {Sm,ToC}
  Is it complete in G? No

=

{A,Sm}

[4] -- T' ≠ Ø ➔

(a) T = T / T' = {{A,L},{L,B}}
(b) G = (V, E ∪ T)
(c) R = T' = {{A,L},{L,B}}
(d) Goto Step 2

R = {{A,L},{L,B}}

[2] -- R' = {A,L} *({A,L} ∈ T and □ {A,Sm} ∈ R s.t. {A,L}∩{L,B}=L ≠ Ø)*

+

{L,B} *({L,B} ∈ T and □ {A,L} ∈ R s.t. {L,B}∩{A,L}=L ≠ Ø)*

= {{A,L},{L,B}}

[3] -- T' =

- {A,L} → adj(A) ∩ adj(L) = {T,L} ∩ {T,A,Sm,ToC,B} = T
  Is it complete in G? Yes

- {L,B} → adj(L) ∩ adj(B) = {T,A,Sm,B,ToC} ∩ {Sm,L,ToC,D} = {Sm,ToC}
  Is it complete in G? No

=
{A,L}

[4] -- T' ≠ Ø ➔
(e) T = T / T' = {L,B}
(f) G = (V, E ∪ T)
(g) R = T' = {L,B}
(h) Goto Step 2 ☞

---

[2] -- R' = {L,B}
[3] -- T' =
➢ {L,B} → adj(L) ∩ adj(B) = {T,Sm,B,ToC} ∩ {Sm,L,ToC,D} = {T,Sm,ToC}
Is it complete in G? No

[4] -- T'=Ø ➔ return T(={L,B}) (Which is the only necessary fill-in in this case)

- Step 4 → Already done too.

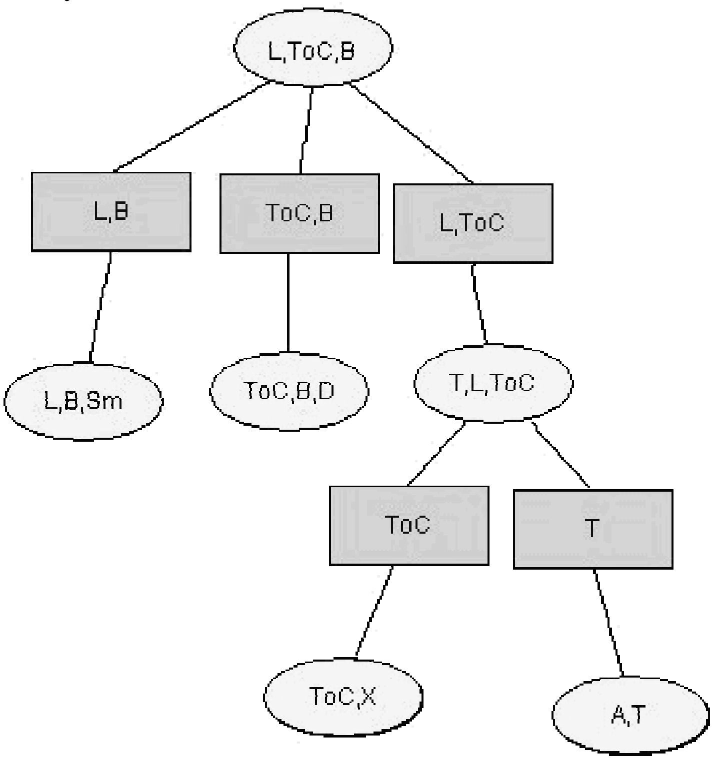


Figure 4- 7. Junction Tree for Asia.

- Step 5 → The only separator S which has a $G^M(S)$ not complete is {L,B}, since this link was added in triangulation. So the MPSD junction tree will be:

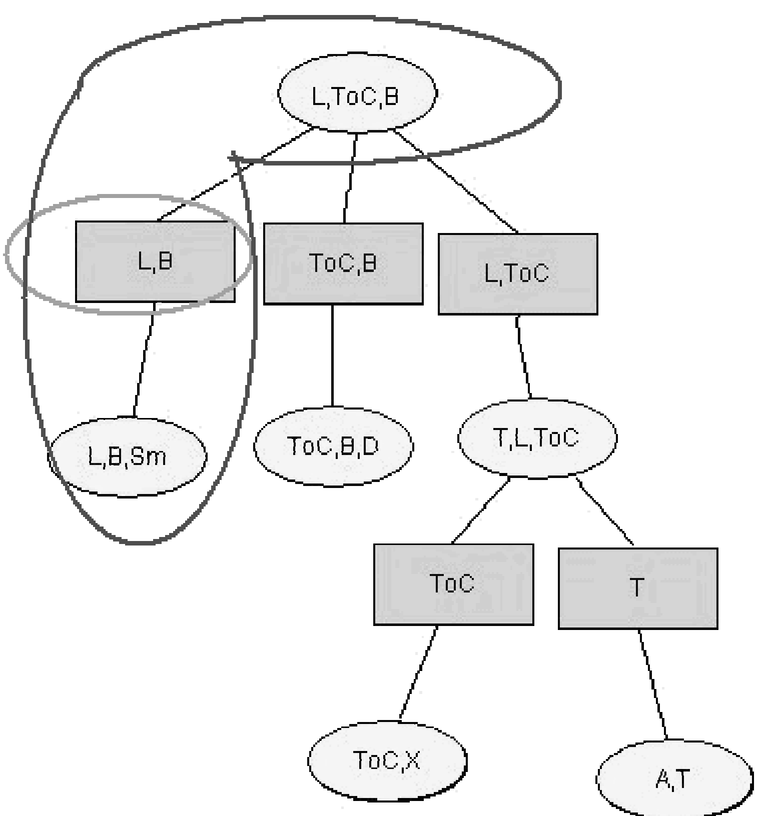


Figure 4- 8. MPSD for Asia. {L,B} is not complete in G^M.

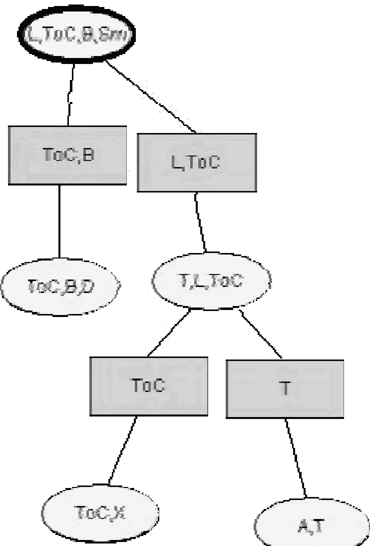


Figure 4- 9. Γ'_{MPD}. Result of combining {L,ToC,B} and {L,B,Sm}into a prime subgraph.

4. When and how we can apply it in incremental compilation.

In the chapter 3 there were some cases we could not find a way to treat, and we advanced that maybe MPSD could be useful. So, in this point we are going to consider the modifications related to a Bayesian network where a link has been deleted.

Just after looking at how construction of MPSD tree works. How can we use it for our purpose. We are going to outline the method to follow:

❶ Construct the Maximal Prime Decomposition tree *[Alg. 3]*

❷ Identify the relevant Maximal Prime Subgraphs in the MPSD tree and the corresponding part of the junction tree.

❸ Redo triangulation of this(these) MPS and reconstruct the junction tree for that(those) parts.

❹ Replace the old junction tree part by the new one obtained.

❺ Connect old separators to the new JT part (check if cliques are already included in existing ones).

Examples in Asia

Example ⑦. We will now illustrate the method in a "failed" example on the previous chapter. We will go through Example 4.2.2.④. of chapter 3. So, it deals with deleting the edge from Sm to B.

1➲ *Construct MPD tree:*

It is already done, see *Figure 4-9* in this chapter.

2➲ *Identify the relevant Maximal Prime Subgraphs in the MPD tree and the corresponding part of the junction tree.*

Well, here the first complication is to determine which are the relevant MPS. After examining the problem we arrive to the conclusion that these relevant MPS are those containing both variables. It is only in these subgraphs where the deletion of the edge can affect. This deletion will provoke different cliques and prime subgraphs, but in the rest of them, it has no influence, since this edge does not participate in this union of variables.

In this case, both Sm and B. Besides, if both variables are included in the same separator then this is no more a complete set in G and Γ_{MPD} should be modified. It is important to remark that completeness of S is checked in G^M. This implies that this completeness should be checked in the modified moral graph. For that, we realise that when a link is removed it could easily (and locally) be determined what changes are implied in G^M. For example, deleting {B,D} would provoke the elimination of the moral link from B to ToC. How can we see that? Well, we know the parents of D and we can keep track of the moral links that have been introduced by this child. So, deleting {B,D} can automatically take to remove the moral links from B to the parents of D.

3➲ *Redo triangulation of this MPS and construct the junction tree for that part.*

One of the properties about Γ_{MPD} demonstrated in [Olesen and Madsen 1999] (in its section 5) is that it can identify a partial triangulation of G^M. And it is possible to triangulate the clusters of Γ_{MPD} independently. So, we are going to take advantage of this result.

The affected part that has to be retriangulated is then the original maximal prime subgraph of Γ_{MPD} {L,ToC,B,Sm}.

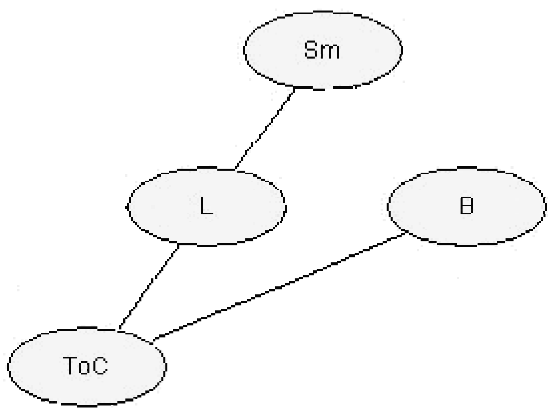


Figure 4- 10. The part of BN' (in BN it was a MPS) to retriangulate in the new network.

The retriangulation of this part does not give any fill-in, because it is already triangulated. Now, the JT for this part is quite easy to see:

Figure 4- 11. Junction tree for the affected part.

4➲ *Replace the old junction tree part by the new one obtained.*

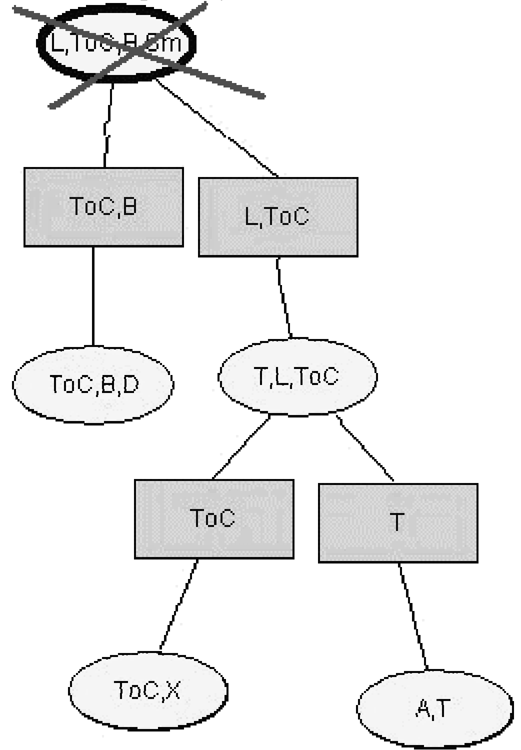


Figure 4- 12. Original JT with the affeced part removed.

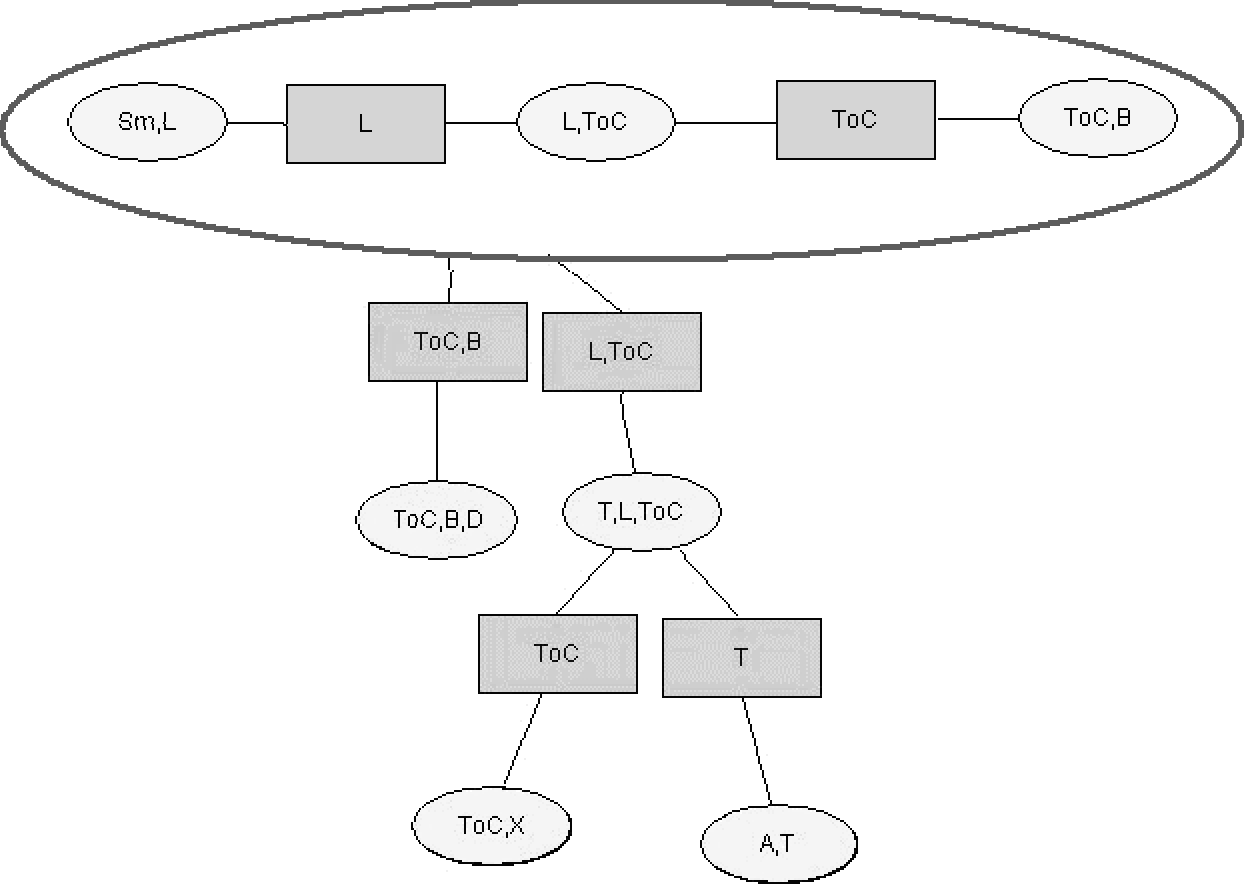


Figure 4- 13. Replacing the new JT for the affected part inside the original one.

5➲ *Connect old separators to the new JT part.*

For achieving this connection we are going to describe a systematic way to do it:

<u>for</u> each separator S connected to the deleted part in the old MPSD JT <u>do</u>

1.- Find a clique C in the new part such as $S \subseteq C$ *(It is sure to find one because S is complete in* G^M*)*

2.- <u>if</u> S = C <u>then</u>

a. Delete C and S.

b. Connect other separators of C to the clique that S was connected

So, we are going to follow this:

Going through separators:

- {ToC,B}

  1.- old separator {ToC,B} $\subseteq$ new clique {ToC,B}.

  2.- {ToC,B} = {ToC,B}

  - Delete old separator and new clique.
  - Connect other separators of C ({ToC}) to the clique that S was connected to ({ToC,B,D}).

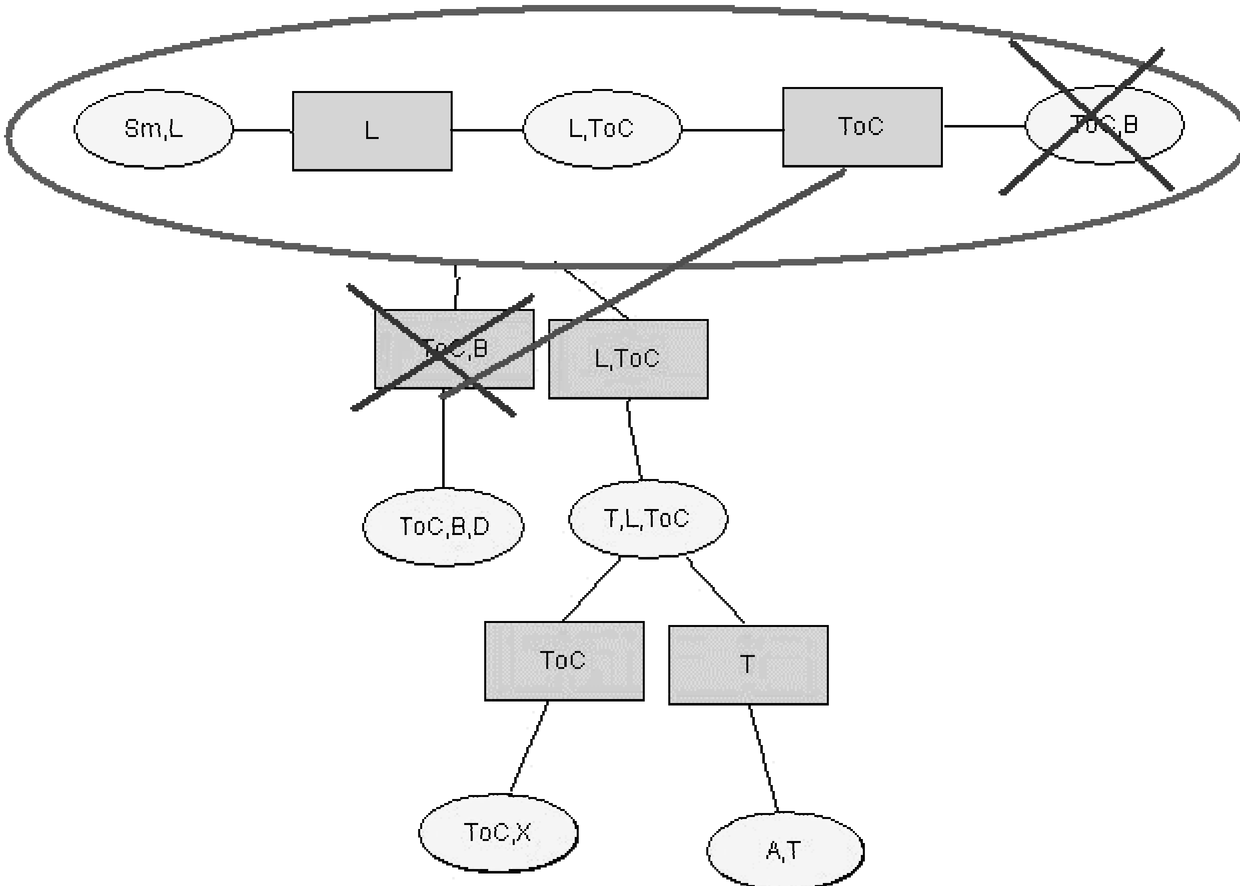


Figure 4- 14. Situation in process of connection between the old JT and the replaced part after treating separtor {ToC,B}.

- {L,ToC}

  1.- old separator {L,ToC} $\subseteq$ new clique {L,ToC}.

2.- {L,ToC} = {L,ToC}

- Delete old separator and new clique.
- Connect other separators of C ({L}) to the clique that S was connected to ({T,L,ToC}).

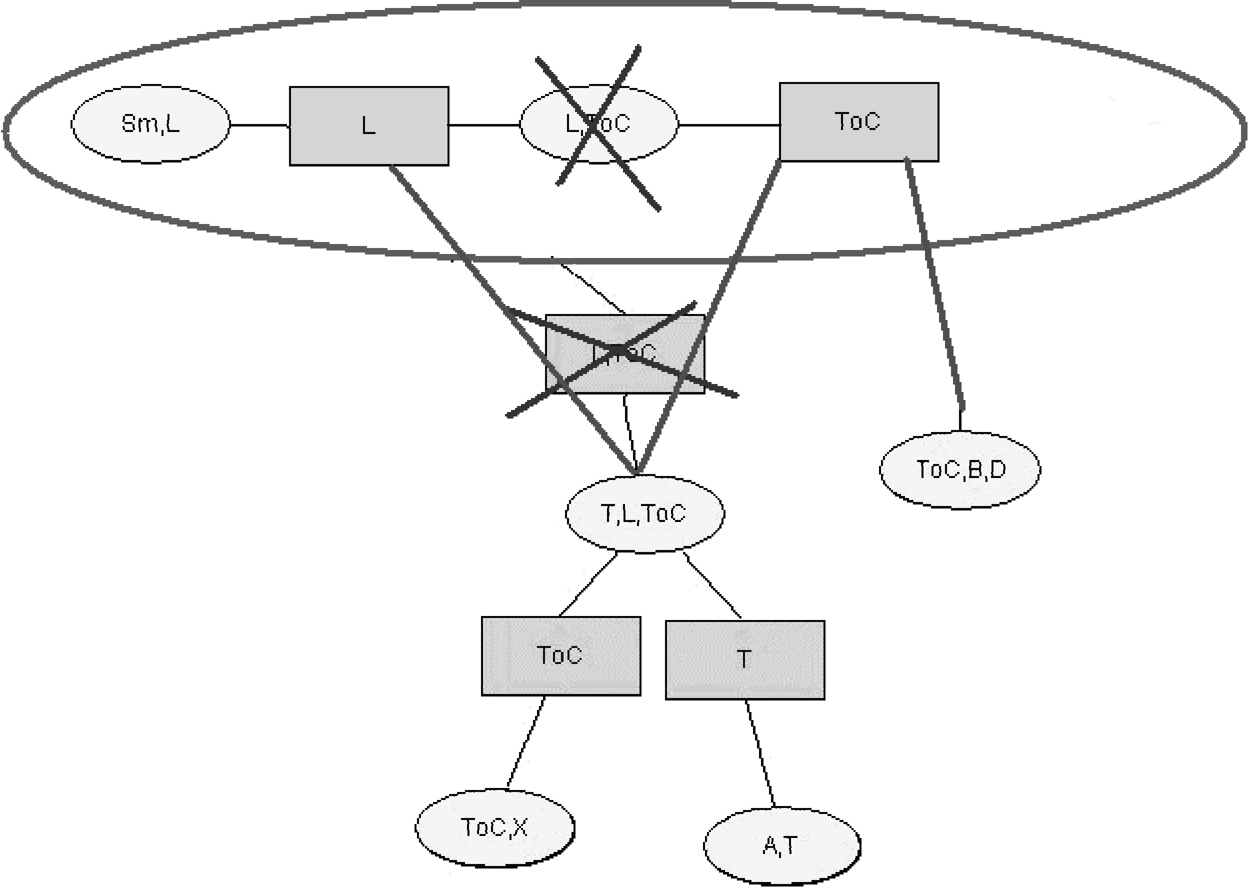


Figure 4- 15. Situation in process of connection between the old JT and the replaced part after treating separtor {ToC,B}.

So, the final junction tree turns to be the one shown in *Figure 4-16*.

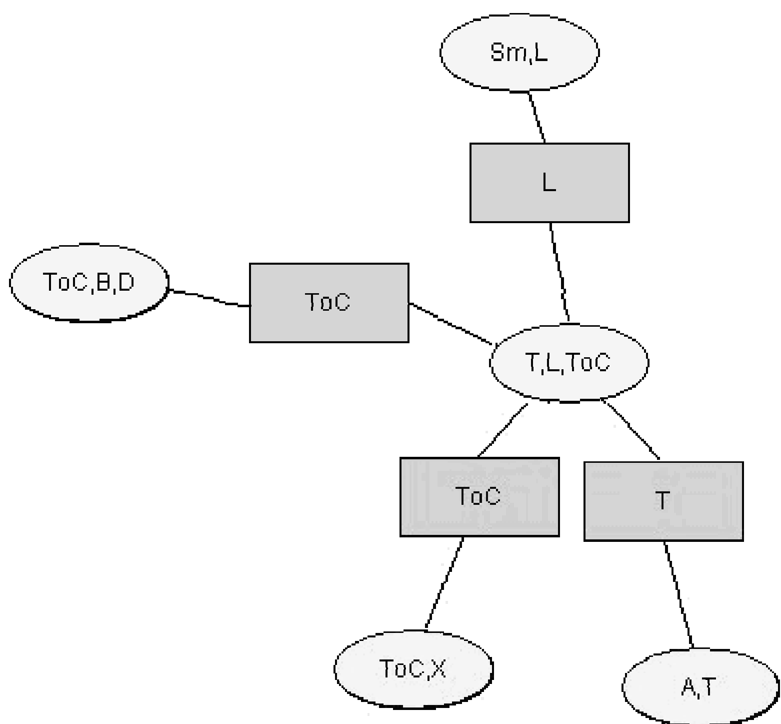


Figure 4- 16. New junction tree obtained after step 5.

We can observe it coincides to the one we reached in chapter 3. (*Figure 3-46*). Then, we have reached the same tree from two different ways. The first one implied a total recompilation of the network, and this one avoids most of the steps we used in that recompilation. Using this intermediate structure, MPSD, the work to do is maybe not easier but more effective.

In addition, with point 2 of step 5 we have already checked if there were some cliques included in existing ones.

And, then, the process has finished.

But, in this example, there is only one MPSD node containing both extremes of the deleted edge. So, we can wonder: What happens if there are more? In this case the two implicated variables will also be present in a separator between them. But this separator is no more complete, since the edge joining them has disappeared.

Example ②. So, to illustrate this case, we are going to follow the process in one example commented previously as well, due to the fact that this also implies deletion of a moral link. The example, deleting edge from B to D corresponds to 4.2.2.③ in chapter 3.

1➲ *Construct MPD tree:*

Once more MPD tree for Asia is already seen (*Figure 4-9*).

2➲ *Identify the relevant Maximal Prime Subgraphs in the MPD tree and the corresponding part of the junction tree.*

Here, as we mentioned we find one peculiarity. If B and D are no more connected, then the moral link from ToC to B has no sense. For that, we have to be able to detect these situations. So, before going on the process we need to modify our basis MPD tree. So, first of all, we take separator {ToC,B} from original MPSD tree and delete it, since they are no more complete. Afterwards, we merge the tree and we obtain the following one:

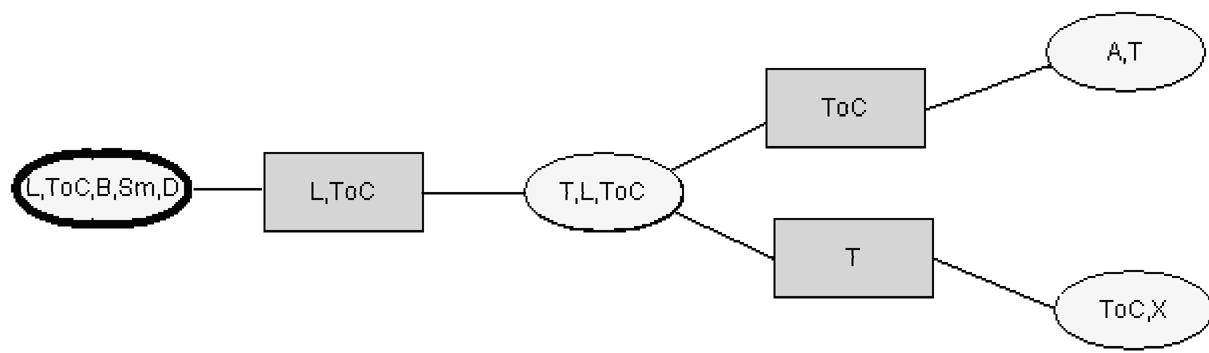


Figure 4- 17. MPSD tree for Asia after deleting edge from b to D (moral link {ToC,B} disappears as well).

Now, we are able to see what are the relevant MPS and this is {L,ToC,B,Sm,D}, the only one which contains B and D (and ToC and B, the moral link consequence of the first one).

3➲ *Redo triangulation of this MPS and construct the junction tree for that part.*

We must remember that now our G^M is:

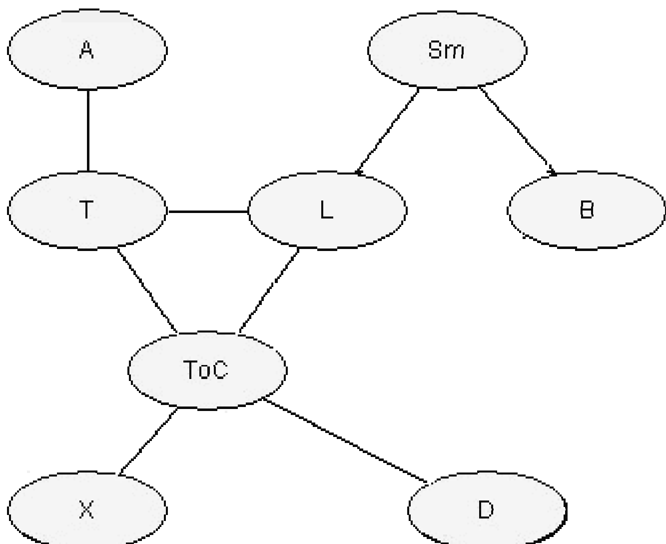


Figure 4- 18. G^M for Asia in the MPSD process after deleting edge from B to D.

Then we are going to retriangulate the part of {L,ToC,B,Sm,D}:

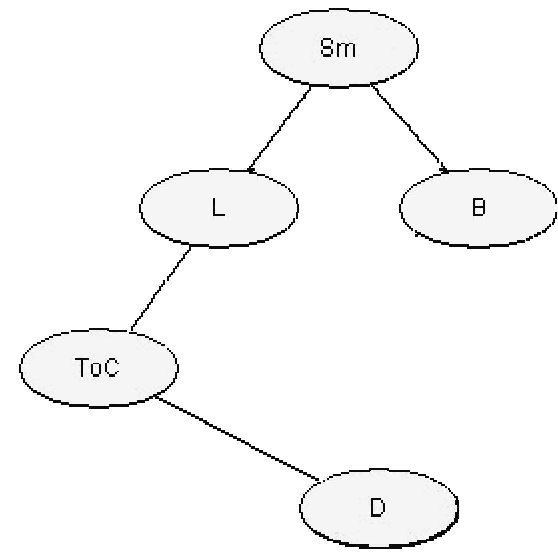


Figure 4- 19. Part to retriangulate ({L,ToC,B,SmD}.

But this is already triangulated. So, the corresponding JT is immediate:

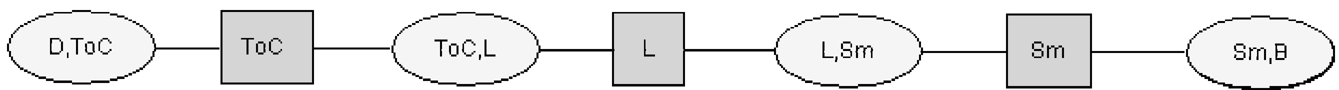


Figure 4- 20. JT from the new part after deleting edge from B to D.

4➲ *Replace the old junction tree part by the new one obtained.*

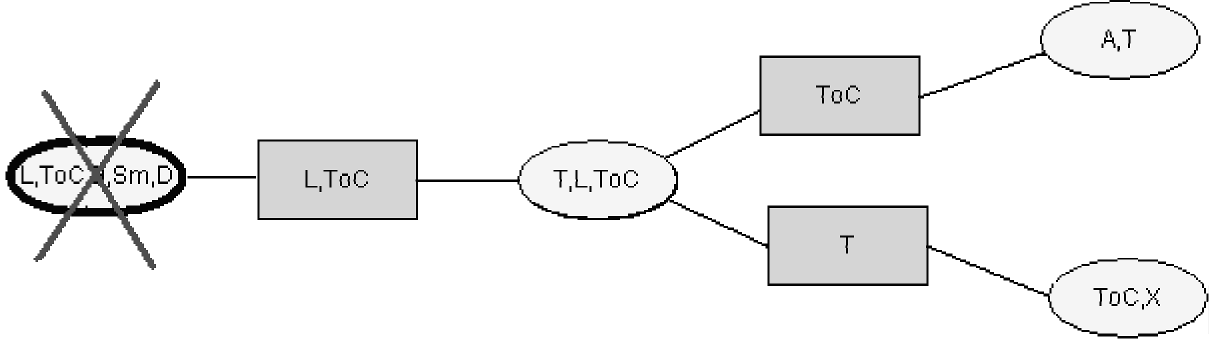


Figure 4- 21. Deleting affected part (L,ToC,B,Sm,D} in the MPSD.

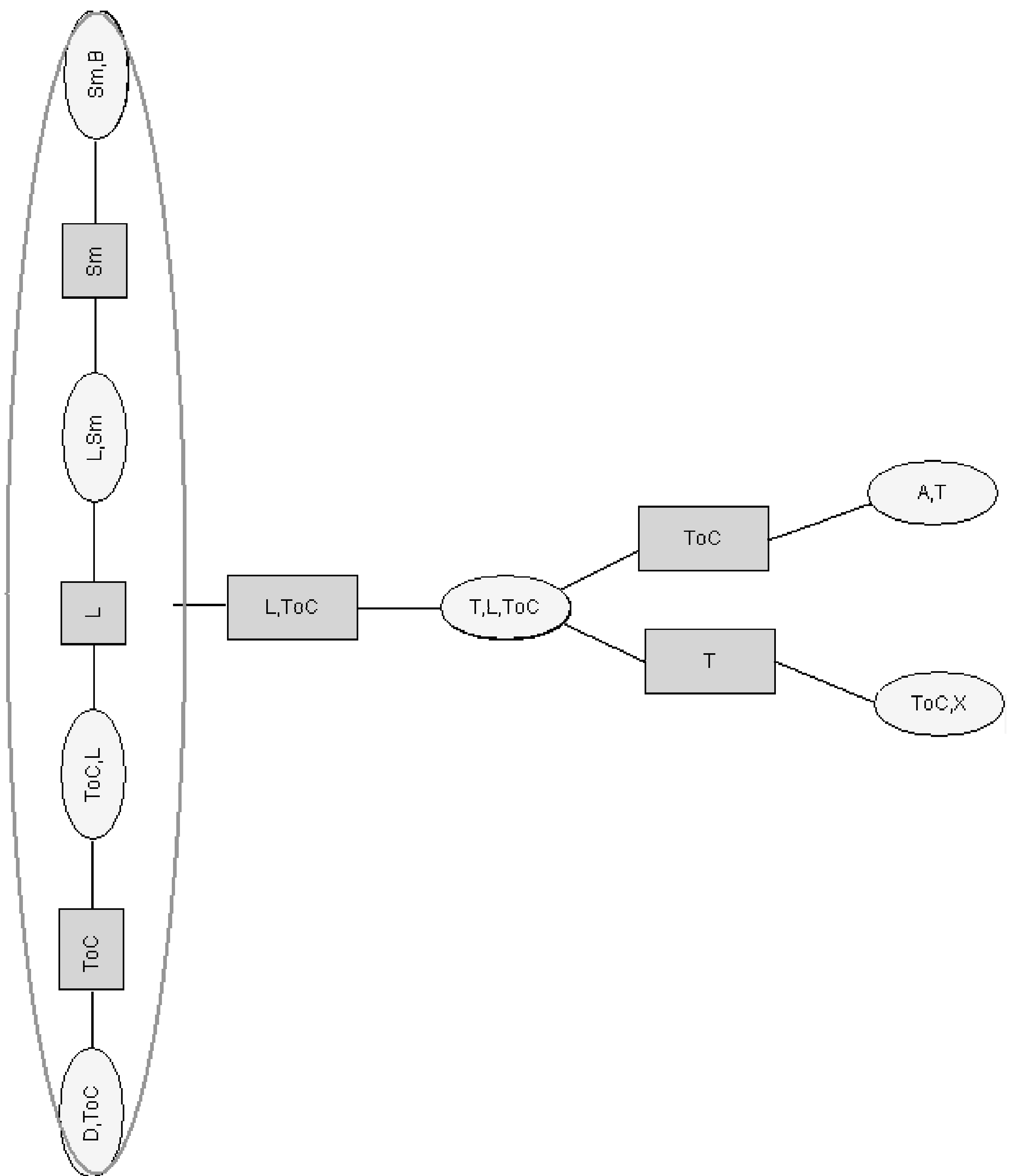


Figure 4- 22. Replacing new part to the previous MPSD.

5➲ *Connect old separators to the new JT part.*

Going through separators: Now there is only one, {L,ToC}. It is equal to one clique in the new JT part. So, both of them disappear (step 2a). And later we have to join the separators connected to {ToC,L} to the clique the old separator was connected before (step 2b):

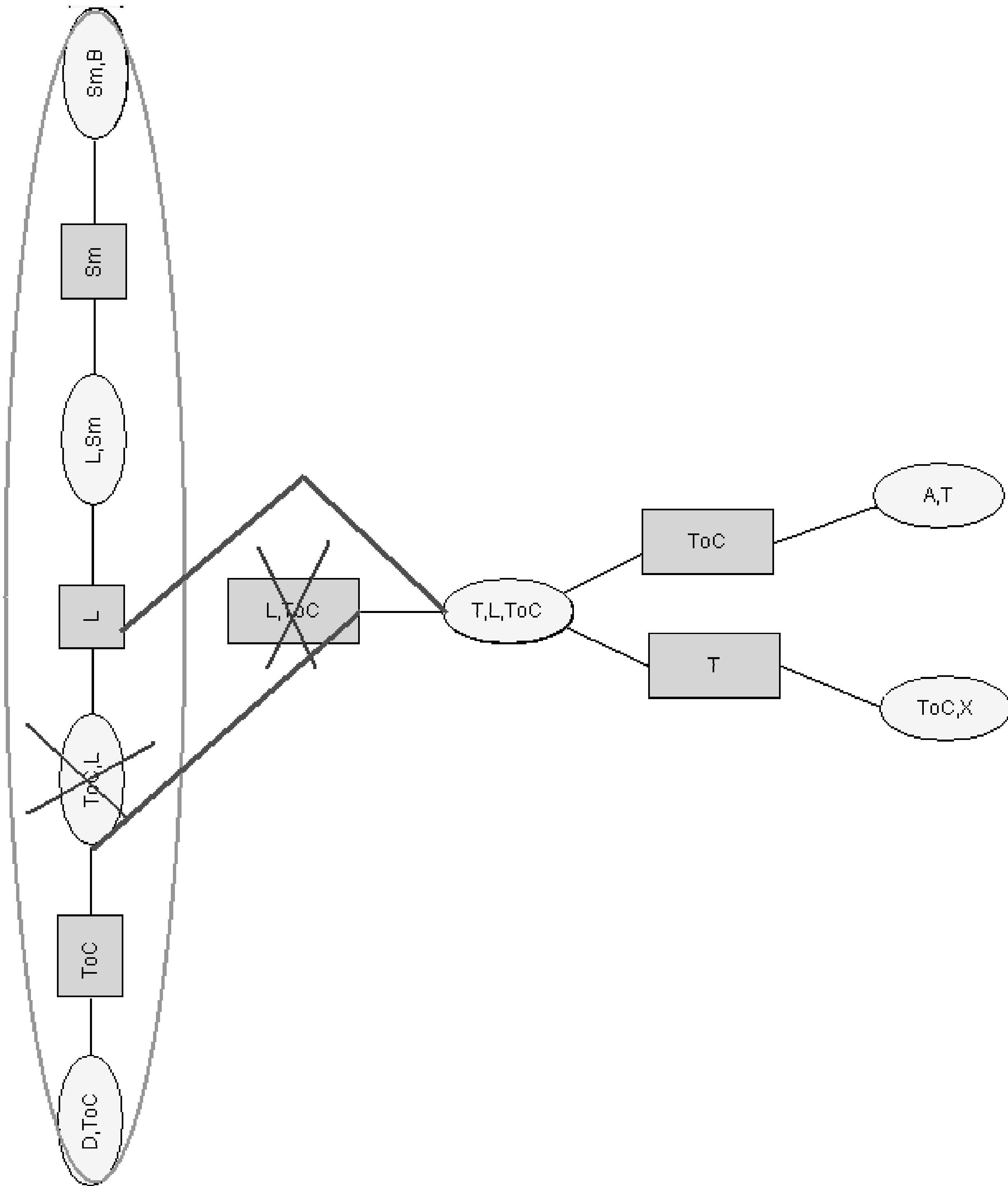


Figure 4- 23. Disappearance of both clique and separator {L,ToC} an we connect other separators of C (Toc and L) to the clique it was connected to ({T,L,ToC}) .

To see it properly, the final result is then:

Sm,B
Sm
Sm,L
L
T,L,ToC
ToC
D,ToC
T
A,T
ToC,X
ToC,X

Figure 4- 24. Resulting junction tree after the process.

This tree is the same we obtained with the conventional way, that is, redoing all the compilation process. (See *Figure3-43*).

5. Discussion.

To conclude this chapter, we should point out the results obtained from this new method using MPSD. As we have shown we have been able to reduce the task of recompilation for problems we could not solve in the previous chapter.

Taking one of the cases, deletion of an edge, we think we have presented a quite interesting algorithm to solve it. A justification of each of the necessary steps has been done, and the application of this method in two examples over the network used in all the project helps us to show the positive results. We are convinced that this idea (the MPSD) could be used in many more cases and its importance will grow with the size of the network to recompile. The larger the network the more time we will save.

Chapter 5. IMPLEMENTATION

1. r-Hugin tool.

r-Hugin (research Hugin) is a programming environment based on Hugin code [r-Hugin 1998]. Its main reason is to provide a set of classes and methods about Bayesian networks. All this code makes use of Hugin one, but it has one advantage: the simplicity. r-Hugin presents a code much clearer to follow than the Hugin one. The latter searches optimal efficiency that makes it quite confused to understand easily.

Therefore, r-Hugin has been conceived for research purposes, and this work has an investigation side. So, we thought it could be interesting to use this existing tool.

2. Programming with Visual C++.

r-Hugin was available for Unix systems. In the department network (*cs.auc.dk*) it can be found from /user/raistlin/RHugin directory. Inside it there are some documentation and figures, we can see one Specs folder with includes some net-files, they have the same format as specification files for Hugin.

In the first phase of the development of this work, I tried to divide my time. On one hand it was necessary to study and attack directly Bayesian networks and especially their compilation process. And on the other hand, I had to make the existing r-Hugin code for Unix work in the Windows environment. This task was not so obvious, because of the known differences between both Operating Systems. The programming environment in Windows system had to be Visual C++ due to University available software. Although C++ is a standard language, this kind of environments usually present their particular features. This fact made my task a little more difficult. But finally, after working on it and having made some small changes, the whole set of header and source files was correctly combined with the hugin header (*hugin.h*) and its api (*hugin.api*). Fortunately, we had a program (*project* or *workspace*) that could be finally compiled and linked.

The technical problem was then solved. Later, for being able to work on this chapter, we needed to know about the hierarchy model *(*see Appendix A*)*, classes, methods and some knowledge about their implementation. Once I have assumed these points, it was time to implement some examples in order to illustrate some of the ideas introduced in this work.

So, in the following section we will try to show programs based on some of the examples in chapter 3.

3. Implemented examples.

Once r-Hugin seemed to work in Windows environment and after having outlined the analysis part, it was time to start the real programming. Obviously, this task has had to be done at the end of the development of this work.

Implementing the proposed solutions turned to be a complicated work. The most difficult part is to understand the whole mechanism between classes and methods, and how they match with the theoretical ones. For example, how a tree is represented, how to introduce potentials, the interactions between elements (variables, cliques, separators, probabilities)…

We started programming from the easiest case and with the intention of arriving as far as possible. But, finally, we could not do too much. In those cases where we have to modify variables, or delete certain structures like cliques, we have found many problems, and unexpected errors. Most of them are due to the interactions we talked about. Most of the elements are linked in lists and they are referred and also make references to elements of other nature. For example, variables refer cliques, cliques refer separators, nodes refer variables, etc. We have to bear in mind many factors every time we modify, insert or remove a structure. For example, modifying a variable needs to modify every reference to it, and all these references are not clear to find. Probably with some more time this task could be accomplished, but at least we are going to show a couple of examples.

- **Compilation program**

The first program we have done is one that makes the compilation of a network. This program takes a Bayesian network from a Hugin net specification file and compiles it, the main program is *compilation.cpp* (see Appendix A.3.1).

- **Changing potentials in a variable of the Bayesian network**

For this first example we thought about showing two different programs:

1.- The conventional form, with recompilation. It means that first we will take the Bayesian network BN and we will compile it. Afterwards, we will do the corresponding modification to transform it to BN'. Finally BN' will be compiled giving the resulting JT'.

2.- Our proposed solution. We compile BN and obtain JT. From this JT we are going to reach JT', without recompiling BN'. We have to remember that the proposed solution was multiply by the new potential value and divide by the old one in the corresponding clique potential.

These two programs have been done for a concrete case, the one showed in chapter 3, example 3.①., that is, changing A probability from (0.01, 0.99) to (0.20, 0.80). For the first case we can see *ChangingPotRecompilate.cpp* (see Appendix A.3.2.a) and for the second one *ChangingPotProposedSol.cpp* (see Appendix A.3.2.b)

For the following examples we decided not to take both possibilities because the first one is not so interesting, that is what we actually do. If we change a network for example in Hugin, to compile it we will need to do a complete compilation again. But, for further studies in larger networks it will be interested to have both programs and compare times.

So, for the rest we will write programs that take the original Bayesian network BN and compiles it giving the resulting JT. Later we will make the modifications commented on chapter 3, depending on the case, to obtain JT' from JT. Using *compilation.cpp* or Hugin tool over the modified Bayesian network (in a net specification file) we will be able to see if the results are the good ones.

- **Deleting a variable child of only other one**

As we saw (example 4.1.2.① in chapter 3), this was the easiest case in deleting variables. It meant that we had to remove the corresponding branch in the junction tree and no potentials (apart from the deleted variable that has already disappeared) have to be touched.

So we have implemented the mentioned example 4.1.2.①, deleting variable X from Asia network. The program *deletingX.cpp* can be seen in Appendix A.3.3.

There are many examples we have not implemented, we have already presented the reason. These two cases are maybe the easiest ones to implement because they do not have many dependencies when we do changes. For example, changing the number of states of a variable (example 4.1.①. and 4.1.②.) can seem simple, but after trying it we can conclude it is not, since this change means changing almost all the structure.

4. Discussion.

With this chapter we had two immediate objectives. Firstly, using the r-Hugin tool, a programming environment for working with Bayesian networks. This tool wants to imitate most of the Hugin functionalities. r-Hugin presents a less optimal code, but this is also the point which makes it easier to understand and work with. This feature is especially important for research purposes.

And secondly, we wanted to prove that the proposed solutions given in chapter 3, and also in chapter 4 were possible. In fact, we have already shown it, with the different examples done "by hand", but using the implementation we can see that these modifications are practicable.

At last, we have fulfilled the first goal. This is the first time that r-Hugin tool has been used for someone different from its programmers. And we have also been able to port the code to a Windows machine.

Unfortunately, all examples of chapter 3 are not implemented, for time limitations. And the same for the method based on MPSD used in chapter 4. Anyhow, we are satisfied because we have been able to show a couple of them and how the global structure is. Besides we find it quite interesting if in a future the rest can be done.

Chapter 6. CONCLUSION

From the introduction of this project the goal to achieve was very clear: how to do a partial recompilation of a Bayesian network. The reason for undertaking this subject was basically the great computation time that the compilation process of a Bayesian network can take. This drawback becomes especially relevant in large models. So, it seems quite attractive looking for alternative ways to a full recompilation.

This idea rises from the fact that once a Bayesian network is already compiled, modifications can be made on it. But, it is probable that these modifications will not be so serious, since the network is still almost the same.

For this study we have chosen an example Bayesian network to inspect. This network, Asia, is one of the most referenced in the literature. In spite of its small number of nodes and edges, it is quite descriptive because we can find several cases in its structure.

Firstly, we have described in an informal way, but also a detailed one, how compilation is accomplished. And it is in this chapter where we start applying the process of compilation on the Asia network in order to see what is really done.

Secondly, we go on analysing directly the problem. For that, the method has been searching the possible modifications that can be made in a Bayesian network. As always, we have used the Asia network as an example, and we have tried to show the different possible modifications in a network. We have explained how these modifications will imply problems of distinct nature in obtaining the final junction tree, and we proposed a set of solutions for most of the presented cases. In addition, we leave an open door to future research on this subject, since the number of possibilities is large.

After this, we have tried to take advantage of an intermediate structure between a Bayesian network and its associated junction tree. The process of compilation takes us from one Bayesian network to its associated tree and this intermediate point could somehow take us to a faster way than the full recompilation. This structure is called a Maximal Prime Subgraph Decomposition (MPSD) tree and it was described in [Olesen and Madsen 1999]. One of its properties, the possibility of triangulate each maximal prime subgraph independently from the rest, has allowed us to only execute a partial recompilation. Even more, we have written an algorithm that uses this idea about $\Gamma_{MPD,}$ and we have applied it on Asia obtaining the same results as we obtained by completely recompiling the network.

The last part tries to give a real view of all these analysis and studies. Taking as a basis the C++ code of RHugin, we have implemented a couple of the mentioned cases in order to show in more detailed how our solutions can be used.

To finish, I would only like to say that I have found quite interesting the development of this project, and since there has been limitations of time and work, I trust this report could at least show an investigation line with possible solutions with reference to the viability of an incremental compilation of a Bayesian network.

Bibliography

📖 [Jensen 1996] Finn V. Jensen. *An Introduction to Bayesian networks.* UCL Press, London 1996.

📖 [Kjærluff 1993] Uffe Kjærulff. *Aspects of efficiency improvement in Bayesian networks*. PhD Thesis. Department of Computer Science. Aalborg University. Aalborg 1993.

📖 [Olesen and Madsen 1999] Kristian G. Olesen and Anders L. Madsen. *Maximal Prime Subgraph Decomposition of Bayesian Networks*. Department of Computer Science. Aalborg University. Aalborg 1999.

📖 [r-Hugin 1998] *r-Hugin Functionalities*. Unpublished document.

Appendix A. r-Hugin details

1. Introduction to r-Hugin for Visual C++.

In this work we do not want to give an exhaustive explanation about this subject, since it is not the main point. But for following the code a little bit, it would be convenient to present the main ideas.

As we told in chapter 5 about *Implementation*, the r-Hugin code has been taken from a Unix environment to a Windows one. In the latter Operating System we have used the Microsft Visual C++ 6.0 tool. To do our programs we have taken the source and header files from r-Hugin in Unix. Some small modifications were necessary, because the compiler of Visual C++ was stricter than the Unix one. These files are:

bn.hpp and *bn.cpp*	hugin_stuff.hpp and *.cpp*	*variable.hpp* and *.cpp*
bn2jt_hugin.hpp and *.cpp*	*jt.hpp* and *.cpp*	*types.hpp* and *.cpp*
graph.hpp and *.cpp*	*potential.hpp* and *.cpp*	*yapc.hpp*
hugin2rhugin.hpp and *.cpp*	*status.cpp*	

And the support files *linkedlist.hpp* and *array.hpp* used to organise structures.The header file *hugin.h* (from Hugin) is also quite important, since it allows the interaction with Hugin and we also need to link *hugin.api* for making it work.

We do not include the code all of them because we consider it is not necessary, the most important is to follow the main ideas. For further detail this code could be provided.

So, we have programmed with Visual C++ workspaces, like projects, where all these files were contained and also a main .cpp one, which we have written for each case. It is this main program that we will include in A.3 section.

2. Class hierarchy.

Before attacking directly the programs, a global view of r-Hugin structure can help. The class structure will be shown and commented in order to understand it better. *Figure A-1* shows the hierarchy class of r-Hugin. We must say it is not complete. First, the root class should be Model that is divided into Bayesian network and Influence Diagram. Furthermore, we cannot see the Utility and Decision nodes, variables and potentials, they these classes are present. There is something missed about edges. In r-Hugin there are different classes for directed edges (utility, informational or causal) and the same happens with the undirected ones (fill-in and moral). And

finally, a class configuration in relation to variable. But these missing elements are not relevant for our programming task, since they do not influence on it.

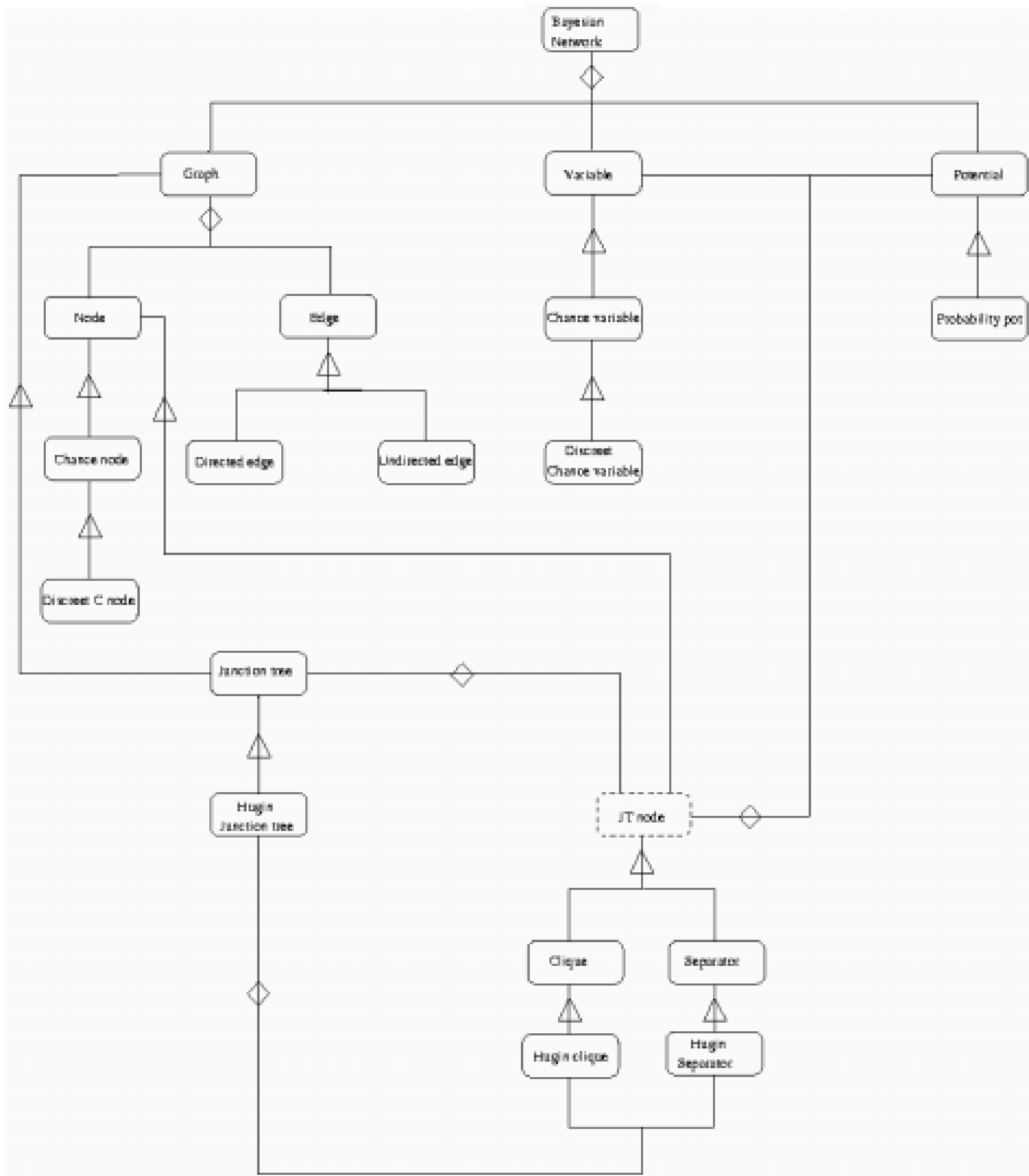


Figure A- 1. Class Hierarchy in r-Hugin.

Looking at *Figure A-1* we see the relations considered in r-Hugin. A *Bayesian Network* is divided into a *Graph*, *Potentials*, (remember BN = {G, *P*}) and *Variables*. Here we start seeing one of the dependencies we talked about: nodes (in graph) and variables are quite related. *Node* presents methods both to obtain and to set the associated variable.

Afterwards a graph is constituted of *Nodes* and *Edges* (remember G=(V,E)), but also connected to a Junction tree structure. This conexion illustrate even more the interaction between structures. Later the *Junction tree* has a specialised class *Hugin Junction tree* which, at the same time is made up of *Cliques* and *Separators*. The other specialisation of classes (*Edges*, *Variables* and *Potentials*) are easy to understand from the Bayesian network theory.

3. Program examples

The way of presenting the programs will be the following one: main program (with grey background), obtained output from this program (this is quite large, but we will mark the most important results in bold and also the less important parts will be omitted), and the corresponding output in Hugin, for both programs in section 3.2 this Hugin output is the same. For the last example, we cannot give the variable marginal values, again due to references problems, but the resulting junction tree is correct because we have contrasted it by the one *compilation.cpp* would give.

3.1. *Compilation.cpp*

💻 Program

```
//  -*- C++ -*-
//
// compilation.cpp - to prove compilation of a Bayesian network in order to use
// it in the project "Incremental compilation of Bayesian networks"
//
// Author          : Julia Flores
//we will need to include the following headers from r-Hugin
#include "types.hpp"
#include "bn.hpp"
#include "jt.hpp"
#include "graph.hpp"
#include "potential.hpp"
#include "variable.hpp"
#include "bn2jt_hugin.hpp"
#include "hugin2rhugin.hpp"
```

```
main(int argc, char **argv)
{  char *pname = argv[0];
  //this program will need an argument, the name of a Hugin specification network
  //(these networks can be obtained from Hugin, for example, taking the option
  //File -> Save Net File
  if (argc!=2)
    {
      cerr << "usage : " << pname << " <hugin specfile>" << endl;
      exit(1);
    }
  cout << "starting...\n";
  //first of all we take the Bayesian network
  //This is in Hugin specification format
  BayesianNetwork *bn = (BayesianNetwork*)hugin2rhugin(argv[1]);

  //function hugin2rhugin is extern (files hugin2rhugin.hpp and hugin2rhugin.hpp)
  //now bn is a Bayesian ntework of RHugin hierarchy

  //We print it to see all her components
  cout << *bn << endl;

  //  Afterwards we compile it obtaining the corresponding junction tree.
  //compile is a function defined on bn2jt_hugin.h and implemented on
  //bn2jt_hugin.cpp:
  //     HuginJunctionTree* compile(Model* r_bn)
  //  This uses Hugin functionality. The followed strategy to do it is
  //    Start up Hugin
  //    Convert rBN into hBN
  //    Compile hBN into hJT
  //    Convert hJT into rJT
  //    Close down Hugin
  //In this fucntion we could indicate the triangulation heuristic we want to use.
  //Looking at chapter 2 (Compilation) it makes all the work described in step 1:
  //moralisation, triangualtion and tree construction.

  HuginJunctionTree *jt = compile(bn);

  //Once we have compiled the bayesian network. this junction tree must be inicialised.
```

```
//For this, the class HuginJunctionTree his own function: initialize(). That would do
//what we explained in step 2.1 of chapter 2.
jt->initialize();

//We print it as well, in order to show its components
cout << *(HuginJunctionTree*)jt << endl;

//And finally we propagate to make it consistent.
//Propagation was the last point (step 2.2 and 2.3 in chapter 2). If we see
//the code of this function it makes exactly that: first it calls
//function collectEvidence(...) and tlater distributeEvidence(...)
jt->propagate();

//To finish we show the resulting tree after propagation. For example
//for asia.net we have obtained its junction tree as the one of Figure 2-4.
//The numerical results are also correct.
cout << "After propagation the tree is" << endl;
cout << *(HuginJunctionTree*)jt << endl;

//To finish with, we will show every variable
for (bn->variables()->to_first(); !bn->variables()->at_end();
     bn->variables()->next())
  {
    if (bn->variables()->get()->type()==rh_chance_variable)
      {
        cout << *bn->variables()->get()->marginal(rh_equilibrium_sum)
           << endl;
        if (bn->variables()->get()->discrete())
          cout << *(DiscreteChanceVariable*)bn->variables()->get() << endl;
      }
    else
      if (bn->variables()->get()->type()==rh_decision_variable)
        {
          cout << *((DiscreteDecisionVariable*)bn->variables()->get())->marginal()
               << endl;
          cout << *(DiscreteDecisionVariable*)bn->variables()->get() << endl;
        }
  }
// Like that we can see that the marginal belief of each variable coincides with
```

```
  //the result we expected. (This has been checked using Hugin output).
cout << "finishing...\n";
}
```

Program Output

Output of compilation.cpp with asia.net (hugin net specification for Asia network):

```
starting...
Bayesian Network
Variable
-name       : E
-identifier : 8
Variable
-name       : B
-identifier : 2
Variable
-name       : L
-identifier : 6
Variable
-name       : S
-identifier : 5
        .........
Graph
-nodes :
Node
-index    : 0
-variable : Variable
-name       : D
-identifier : 3
Node
-index    : 0
-variable : Variable
-name       : A
-identifier : 4
        .........
-edges :
Edge
-head : Node
-index    : 0
-variable : Variable
-name       : X
-identifier : 1
-tail : Node
-index    : 0
-variable : Variable
-name       : E
-identifier : 8

Edge
-head : Node
-index    : 0
-variable : Variable
-name       : B
-identifier : 2
-tail : Node
-index    : 0
-variable : Variable
-name       : S
-identifier : 5

        .........
Hugin Clique
-identifier : -1
-potential :
Probability Potential
-identifier  : 0
-size        : 8
-heads       : Variable
-name       : B
-identifier : 2
Variable
-name       : E
-identifier : 8
Variable
-name       : D
-identifier : 3
-tails       :
-numbers  :  0.9  0.1  0.8  0.2  0.7 0.3 0.1 0.9
-potential :
UtilityPotential
-identifier : -1
-size       : 8
-domain     : Variable
-name       : B
-identifier : 2
Variable
-name       : E
-identifier : 8
Variable
-name       : D
-identifier : 3
-numbers    : 0 0 0 0 0 0 0 0

-variables  :
Discrete Chance Variable
-identifier : 2
-name       : B
-states     : 2 0 yes 1 no

Discrete Chance Variable
-identifier : 8
-name       : E
-states     : 2 0 yes 1 no

Discrete Chance Variable
-identifier : 3
```

-name : D
-states : 2 0 yes 1 no

-separators :
Separator
-variables : Variable
-name : E
-identifier : 8
Variable
-name : B
-identifier : 2
………
Hugin Clique
-identifier : -1
-potential :
Probability Potential
-identifier : 0
-size : 4
-heads : Variable
-name : T
-identifier : 7
Variable
-name : A
-identifier : 4
-tails :
-numbers : 0.0005 0.0099 0.0095 0.9801
………

Probability Potential
-identifier : 0
-size : 2
-heads : Variable
-name : **E**
-identifier : 8
-tails :
-numbers : **0.064828 0.935172**
………
Probability Potential
-identifier : 0
-size : 2
-heads : Variable
-name : **B**
-identifier : 2
-tails :
-numbers : **0.45 0.55**
………
Probability Potential
-identifier : 0
-size : 2
-heads : Variable
-name : **L**
-identifier : 6
-tails :
-numbers : **0.055 0.945**
………

Probability Potential
-identifier : 0
-size : 2
-heads : Variable
-name : **S**
-identifier : 5
-tails :
-numbers : **0.5 0.5**
………
Probability Potential
-identifier : 0
-size : 2
-heads : Variable
-name : **X**
-identifier : 1
-tails :
-numbers : **0.11029 0.88971**
………
Probability Potential
-identifier : 0
-size : 2
-heads : Variable
-name : **T**
-identifier : 7
-tails :
-numbers : **0.0104 0.9896**

………
Probability Potential
-identifier : 0
-size : 2
-heads : Variable
-name : **A**
-identifier : 4
-tails :
-numbers : **0.01 0.99**

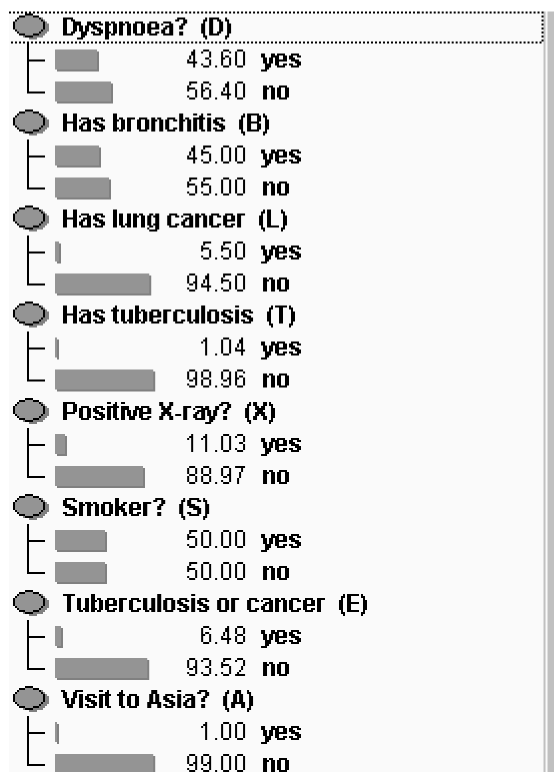
………
Probability Potential
-identifier : 0
-size : 2
-heads : Variable
-name : **D**
-identifier : 3
-tails :
-numbers : **0.435971 0.564029**

Discrete Chance Variable
-identifier : 3
-name : D
-states : 2 0 yes 1 no

finishing...

Since, we understand the difficulty to follow this output for the next examples we will only show the final part.

Hugin Output for Asia



3.2. *Changing potentials in A*

3.2.a.ChangingPotRecompilate.cpp

Program

```
//  -*- C++ -*-
//
// changingPotRecompilate.cpp -
// After compilating we change potentials and recompilate all the network again
//
// Author          : Julia Flores

#include "types.hpp"
#include "bn.hpp"
#include "jt.hpp"
```

```
#include "graph.hpp"
#include "potential.hpp"
#include "variable.hpp"
#include "bn2jt_hugin.hpp"
#include "hugin2rhugin.hpp"

//This program tries to see the Example 3.1 of chapter 3
//For this first example, we are going to see both pograms:
// -the one with recompilation (this one)
// -and the one with our proposed solution
//
// That is for showing the differemces between them. But later, we will use
//another method to compare results. We will take the net specification file
//to the modified Bayesian network (BN') and compile with compilation.cpp
//from which we started.
//
// And these results will be taken as a basis to check the good results for
//the proposed solutions.

// The next two functions do the same.
// They are more or less like the compilation.cpp program.
// And the action is to carry out the whole compilation process, from the
//graph and its moralisation and triangulation, for later constructing
//the jt structure. And finally initialise and propagate.
// The only difference between both of them is their arguments.
//  - process_compilation takes a string that corresponds to the name of a
//  net specification file, and compiles it.
//
//  - process_compilation_jt takes a Bayesian network (and object of the r-Hugin
//class BayesianNetwork and returns a junction tree in the form of an object of
//r-Hugin class HuginJunctionTree)
void process_compilation(char *net_name);
//given the name of a spec hugin file, compiles it
HuginJunctionTree* process_compilation_jt(BayesianNetwork *bn);
//given a Bayesian network return its jt associated
main(int argc, char **argv)
{
  char *pname = argv[0];
  //this program will need an argument, the name of a Hugin specification network
```

```
if (argc!=2)
  {
    cerr << "usage : " << pname << " <hugin specfile>" << endl;
    exit(1);
  }
// First, we call the compilation process with the name of the network
//in our case asia.net. That would do the same as if we execute compilation.exe
process_compilation(argv[1]);
// Now bn is correctly compilated. We are going to make the modifications on it.

// In this program we are going to present the implementation of example 3.1
//of chapter 3: Possible modifications in Bayesian networks. If we read it,
//the change is the following one:
// in original Bayesian network A probability is yes:0.99 no:0.01
// After the modification its probabilty will be yes:0.80 no:0.20
// So, it's that what we are going to do now, modify the Bayesian network:
//We take the original network and change A potentials to the new values.
BayesianNetwork *bn = (BayesianNetwork*)hugin2rhugin(argv[1]);
//We need to have the Bayesian network in its structure, so we catch the
//original one from the specification net file
//For changing A probability we have thought of going through all the
//variables of the network and look for it. We have not found a more direct
//way of obtaining it.
for (bn->variables()->to_first(); !bn->variables()->at_end();
     bn->variables()->next())
         { if (bn->variables()->get()->type()==rh_chance_variable)
           cout << "Name : "<< *bn->variables()->get()->name()<<endl;
       if (*bn->variables()->get()->name()=='A')
              {   //Now we have A
                      cout << "This is A" << endl;

       //According to the structure of classes to create a new PropbabilityPotential
       //we have to indicate the variables head and tail, that is, for one variable
       //the conditional probability is P(head|tail), where head and tail is a list
       //of variables. In this case head will be A and tail an empty set, since A has no
       //parents.
       Set<Variable*> *potAhead = new Set<Variable*>();
       potAhead->insert(bn->variables()->get());
       Set<Variable*> *potAtail = new Set<Variable*>();
```

```
        ProbabilityPotential *NewProbPot = new ProbabilityPotential(potAhead, potAtail);
        //Now we introduce the new values for the probability potential
        (*NewProbPot)[0] = 0.2;
        (*NewProbPot)[1] = 0.8;
        //And insert them in the corresponding variable A
        bn->variables()->get()->potential(NewProbPot);
        //Now A has the probability values we decided to change
        }
   }
   //So, it's time to recompilate, doing a second compilation for this new network.
   //The new network has been obtained from modifications of the initial one.
   //To compilate we use the second fucntion process_compilation_jt. This function gives
   //the tree already initialised and with evidence propagated.
    cout << "JUNCTION TREE IS " << *(HuginJunctionTree*)process_compilation_jt(bn) << endl;
   cout << "End of the program" <<endl;
    //The result is exactly the same as if we realise these modifications
   //in the network of Hugin and compile again.
 }

void process_compilation(char *net_name)
{ cout << "starting process_compilation... for " << net_name << "\n";
  //first of all we take the Bayesian network
  //This is in Hugin specification format
  BayesianNetwork *bn = (BayesianNetwork*)hugin2rhugin(net_name);
  //now bn is a Bayesian network of RHugin hierarchy
  //we print it to see all her components
  cout << " PRINTING BAYESIAN NETWORK" << endl;
  cout << *bn << endl;
  cout << "------------------- END OF BAYESIAN NETWORK-------------------" << endl;
  //afterwards we compile it obtaining the corresponding junction tree
  HuginJunctionTree *jt = compile(bn);
  //this junction tree must be inicialised
  jt->initialize();
  //We print it as well, in order to show its components
  cout << " PRINTING JUNCTION TREE" << endl;
  cout << *(HuginJunctionTree*)jt << endl;
  cout << "------------------- END OF JUNCTION TREE----------------------" << endl;
  //And finally we propagate to make it consistent
  jt->propagate();
```

```
  //To finish with, we will show every variable
  //We are only interested in Discrete Chance ones
  cout << "PRINTING VARIABLES" << endl;
  for (bn->variables()->to_first(); !bn->variables()->at_end();
       bn->variables()->next())
      if (bn->variables()->get()->type()==rh_chance_variable)
        {
      cout << "PRINTING 'rh_chance_variable'" << endl;
          cout << "Name : "<< *bn->variables()->get()->name()
             << endl;
          cout << *bn->variables()->get()->marginal(rh_equilibrium_sum)
             << endl;
          if (bn->variables()->get()->discrete())
          { cout << *(DiscreteChanceVariable*)bn->variables()->get() << endl;
      }
        }
cout << "finishing process compilation for " << net_name <<"\n";
}
HuginJunctionTree* process_compilation_jt(BayesianNetwork *bn)
{
  //now we have the structure of BN for rHugin, so
  //we can work directly over it
  //we print it to see all her components to see the network
  cout << "PRINTING BAYESIAN NETWORK" << endl;
  cout << *bn << endl;
  cout << "------------------- END OF BAYESIAN NETWORK-------------------" << endl;
  //afterwards we compile it obtaining the corresponding junction tree
  HuginJunctionTree *jt = compile(bn);
  //this junction tree must be inicialised
  jt->initialize();
  //We print it as well, in order to show its components
  cout << "PRINTING JUNCTION TREE" << endl;
  cout << *(HuginJunctionTree*)jt << endl;
  cout << "------------------- END OF JUNCTION TREE----------------------" << endl;
  //And finally we propagate to make it consistent
  jt->propagate();
  //To finish with, we will show every variable again
  cout << "PRINTING VARIABLES" << endl;
  for (bn->variables()->to_first(); !bn->variables()->at_end();
```

```
       bn->variables()->next())
      if (bn->variables()->get()->type()==rh_chance_variable)
        {
      cout << " PRINTING 'rh_chance_variable'?" << endl;
         cout << "Name : "<< *bn->variables()->get()->name()
            << endl;
         cout << *bn->variables()->get()->marginal(rh_equilibrium_sum)
            << endl;
         if (bn->variables()->get()->discrete())
         { cout << *(DiscreteChanceVariable*)bn->variables()->get() << endl;
      }
        }
  //Finally, we return the resulting tree
 return jt;
}
```

Program Output

starting process_compilation... for ..\..\asia.net

…………

Probability Potential

-identifier : 0

-size : 2

-heads : Variable

-name : **E**

-identifier : 8

-tails :

-numbers: **0.064828 0.935172**

…………

-name : **B**

-numbers : **0.45 0.55**

…………

-name : **L**

-numbers : **0.055 0.945**

…………

-name : **S**

-numbers : **0.5 0.5**

…………

-name : **X**

-numbers : **0.11029 0.88971**

…………

-name : **T**

-numbers : **0.0104 0.9896**

…………

-name : **A**

-numbers : **0.01 0.99**

…………

-name : **D**

-numbers : **0.435971 0.564029**

finishing process compilation for ..\..\asia.net

…………

(recompilation)

Probability Potential

-name : **E**

-numbers : **0.07201 0.92799**

…………

-name : **B**

-numbers : **0.45 0.55**

…………

-name : **L**

-numbers : **0.055 0.945**

…………

-name : **S**

-numbers : **0.5 0.5**

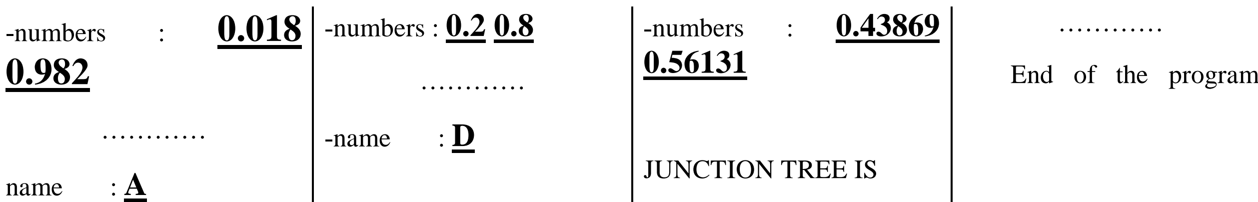
…………

-name : **X**

-numbers : **0.116969 0.883031**

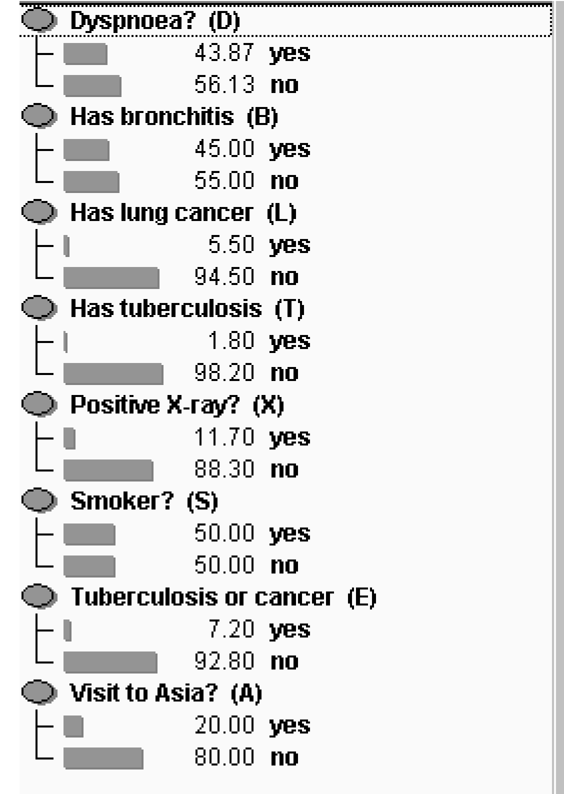
…………

-name : **T**



-numbers : **<u>0.018</u>** **<u>0.982</u>** ………… name : **<u>A</u>**	-numbers : **<u>0.2</u>** **<u>0.8</u>** ………… -name : **<u>D</u>**	-numbers : **<u>0.43869</u>** **<u>0.56131</u>** JUNCTION TREE IS	………… End of the program

Hugin Output for Asia changing A potentials from (0.01, 0.99) to (0.20,0.80)



3.2.b.ChangingPotProposedSolution.cpp

Program

We will not show the body of the compilation functions, since they are always the same.

```
//  -*- C++ -*-
//
// changingPotProposedSol.cpp -
// After compilating we change potentials and we want to avoid
// recompilation. In this case, the solution will be multiply
// by the new potential and dicide by the old one (Example 3.1.
// in chapter "Possible Modifications in a Bayesian network")

// Author          : Julia Flores
```

```
//Necessary header files
#include "types.hpp"
#include "bn.hpp"
#include "jt.hpp"
#include "graph.hpp"
#include "potential.hpp"
#include "variable.hpp"
#include "bn2jt_hugin.hpp"
#include "hugin2rhugin.hpp"

//given a Bayesian network return its  associated junction tree.
//This was alredy used in changingPotRecompilate.cpp. So, we omit
//further comments.
HuginJunctionTree* process_compilation_jt(BayesianNetwork *bn);

main(int argc, char **argv)
{
  char *pname = argv[0];

  //this program will need an argument, the name of a Hugin specification network
  if (argc!=2)
    {
      cerr << "usage : " << pname << " <hugin specfile>" << endl;
      exit(1);
    }
  //First, we call the initial compilation process.
  //For that we need to "read" the Bayesian network from the file.
  BayesianNetwork *bn = (BayesianNetwork*)hugin2rhugin(argv[1]);
  //And later to obatin the resulting junction tree, we compile it
  HuginJunctionTree *jt = (HuginJunctionTree*)process_compilation_jt(bn);
  //Showing tree on the screen.
  cout <<" JT for the original BN is "
          << *jt
          << endl;
  //  Now we are going to modify the potential table of A,t directly.
  //AS we told in the analysis chapter our proposed solution is to
  //take the table and in each entry we have to multiply by the new value
  //and divide by the old one.
```

```
//
//  As we mentioned if the old one is 0, this solution is not possible.
//We will have to take it into account if we design a general program.
//However for this program which reflects a specific case, we don't have
//to check it. We actually introduce the exact values.
// So, we declare a new variable which is a Discrete Chance one.
DiscreteChanceVariable *MyVbleA;
// In the network we lookmfor A, and in this auxiliar variable
//we will store it so that we can use its values.
for (bn->variables()->to_first(); !bn->variables()->at_end();
     bn->variables()->next())
        { if (bn->variables()->get()->type()==rh_chance_variable)
          cout << "Name : "<< *bn->variables()->get()->name()<<endl;
       if (*bn->variables()->get()->name()=='A')
             {   cout << "Encontrada A" << endl;
             MyVbleA=(DiscreteChanceVariable*)bn->variables()->get();
             }
        }
//  Now we can take the old probabilty values of A and we have them
//in OldProbPot
ProbabilityPotential *OldProbPot=(ProbabilityPotential *)MyVbleA->potential()->copy();

//  Afterwards we introdue the new vaules we want to have. yes:0.80 no:0.20
//  For this purpose we will use the object NerwProbPot (ProbabilityPotential)
     Set<Variable*> *potAhead = new Set<Variable*>();
     potAhead->insert(MyVbleA);
     Set<Variable*> *potAtail = new Set<Variable*>();
     ProbabilityPotential *NewProbPot = new ProbabilityPotential(potAhead, potAtail);
     (*NewProbPot)[0] = 0.2;
     (*NewProbPot)[1] = 0.8;


//Now we don't want to recompilate, we know that A potentials have change
// It was (0.01, 0.99)
// and now it is (0.2,0.8)
// So, we have the data we wanted. It is only necessary to find the corresponding cliques
//in the junction tree for the original BN. This is jt, the result of compilation. And
//we will do the proposed changes on it to reach JT', the junction tree for the modified
network BN'.
 // We need to go through all cliques and to see which ones are affected by A
```

```
  //To start experimenting we only display them
  // We need to go through all cliques and to see which ones are affected by A
  //To start experimenting we only display them
  for (jt->cliques()->to_first(); !jt->cliques()->at_end();
      jt->cliques()->next())
    {
        cout << "----------------------------------------------------- " << endl;
        cout << " Clique is  " << *(HuginClique*)(jt->cliques()->get()) << endl;
        if (jt->cliques()->get()->variables()->contains(MyVbleA))
                 {
        //clique to be modified
       cout << "This clique contains A" << endl;
         //And we do these two operations
         // 1.- value' (potential of the clique)  = value * "new potential"
       jt->cliques()->get()->probabilityPotential()->multiply_potential((Potential*)NewProbPot);
         // 2.- value'' (definitive potential of the clique)  = value'' / "old potential"
        jt->cliques()->get()->probabilityPotential()->divide_potential((Potential*)OldProbPot);

        //The r-Hugin structure does the operations in the right way, so that each entry
        //would be multiplied by the corresponding value of A, depending on the state.
                   }
      }
 cout << "=================================================================" << endl;
 //Later we have to propagate evidence to make the tree consistent
 jt->propagate();
 cout << "After changes the junction tree is " << *jt << endl;
 //And now we print variables to be able to compare the result with the
 //correct ones.
 cout << "PRINTING VARIABLES" << endl;
 for (bn->variables()->to_first(); !bn->variables()->at_end();
      bn->variables()->next())
    if (bn->variables()->get()->type()==rh_chance_variable)
      {
     cout << "PRINTING 'rh_chance_variable'" << endl;
        cout << "Name : "<< *bn->variables()->get()->name()
           << endl;
        cout << *bn->variables()->get()->marginal(rh_equilibrium_sum)
           << endl;
        if (bn->variables()->get()->discrete())
```

```
          { cout << *(DiscreteChanceVariable*)bn->variables()->get() << endl;
         }
       }
  //This output coincides with the one given by Hugin tool.
  cout << "End of program ChangingPotProposedSolution" <<endl;
}
```

Program Output

………..

Probability Potential

-identifier : 0

-size : 2

-heads : Variable

-name : **E**

-identifier : 8

-tails :

-numbers : **0.07201 0.92799**

………..

-name : **B**

-numbers : **0.45 0.55**

………..

-name : **L**

-numbers : **0.055 0.945**

………..

-name : **S**

-numbers : **0.5 0.5**

………..

-name : **X**

-numbers : **0.116969 0.883031**

………..

-name : **T**

-numbers : **0.018 0.982**

………..

-name : **A**

-numbers : **0.2 0.8**

………..

-name : **D**

-numbers : **0.43869 0.56131**

End of program ChangingPotProposedSolution

Hugin Output

The same as 3.2.a.

3.3. *Deleting X*

Program

```
//  -*- C++ -*-
//
// DeletingX.cpp
// Here we try to reproduce the example 4.1.2.1 in chapter 3
// This was the simplest case, since deletion of X does not
// affect in the potentials of other variable (it has no children)
// Author          : Julia Flores

// include lines
#include "types.hpp"
#include "bn.hpp"
#include "jt.hpp"
```

```
#include "graph.hpp"
#include "potential.hpp"
#include "variable.hpp"
#include "bn2jt_hugin.hpp"
#include "hugin2rhugin.hpp"

//The next two function are the same we used in the other programs
//given the name of a spec hugin file, compiles it
void process_compilation(char *net_name);
//given a Bayesian network return its jt associated
HuginJunctionTree* process_compilation_jt(BayesianNetwork *bn);
main(int argc, char **argv)
{
  char *pname = argv[0];
  //this program will need an argument, the name of a Hugin specification network
  if (argc!=2)
    {
      cerr << "usage : " << pname << " <hugin specfile>" << endl;
      exit(1);
    }
  BayesianNetwork *bn = (BayesianNetwork*)hugin2rhugin(argv[1]);
  //First, we call the compilation process
  HuginJunctionTree *jt = process_compilation_jt(bn);
  //And now we are going to delete the corresponding clique (X,ToC) as we
  //saw in chapter 3.
  for (jt->cliques()->to_first(); !jt->cliques()->at_end();
       jt->cliques()->next())
           {
                  //we print the cliques
                  cout << "----------------------------------------------------- " << endl;
                  cout << " Clique is  " << *(HuginClique*)(jt->cliques()->get()) << endl;
                  jt->cliques()->get()->variables()->to_first();
           if(*jt->cliques()->get()->variables()->get()->name()=='E')
                  {   //if clique X-ToC
                         cout << "This is clique X,ToC" << endl;
                         //We remove it
                         jt->cliques()->remove();
            }//endif
           }//end for
```

```
// We will probably remove the separator, but that gave some problems. This has been
//commented in chapter 5, where we have presented the encountered problems at programmig
//time.
  //We peopagate the new "evidence". It's the same but without X whose values did not
  //affect on any child.
  jt->propagate();
  cout << "=================================================================" << endl;
  cout << "After changes the junction tree is " << *jt << endl;
   cout << "End of program DeletingX" <<endl;
}
```

Program Output

After changes the junction tree is

Hugin Junction Tree

Hugin Clique

-identifier : -1

-potential :

Probability Potential

-identifier : 0

-size : 8

-heads : Variable

-name : **B**

-identifier : 2

Variable

-name : **E**

-identifier : 8

Variable

-name : **D**

-identifier : 3

-tails :

-numbers : **0.0322672 0.00358524 0.331318 0.0828295 0.0202829 0.00869268 0.0521024 0.468922**

………..

Hugin Clique

-identifier : -1

-potential :

Probability Potential

-identifier : 0

-size : 4

-heads : Variable

-name : **T**

-identifier : 7

Variable

-name : **A**

-identifier : 4

-tails :

-numbers : **0.0005 0.0099 0.0095 0.9801**

Hugin Clique

-identifier : -1

-potential :

Probability Potential

-identifier : 0

-size : 8

-heads : Variable

-name : **L**

-identifier : 6

Variable

-name : **E**

-identifier : 8

Variable

-name : **T**

-identifier : 7

-tails :

-numbers : **0.000572 0.054428 0 0 0.009828 0 0 0.935172**

Hugin Clique

-identifier : -1

-potential :

Probability Potential

-identifier : 0

-size : 8

-heads : Variable

-name : L

-identifier : 6

Variable

-name : B

-identifier : 2

Variable

-name : E

-identifier : 8

-tails :

-numbers : **0.0315 0 0.0235 0 0.0043524 0.414148 0.0054756 0.521024**

Hugin Clique

-identifier : -1

-potential :

Probability Potential

-identifier : 0

-size : 8

-heads : Variable

-name : **S**

-identifier : 5

Variable

-name : **L**

-identifier : 6

Variable

-name : **B**

-identifier : 2

-tails :

-numbers : **0.03 0.02 0.27 0.18 0.0015 0.0035 0.1485 0.3465**

………..

End of program DeletingX

Output using *compilation.cpp* to show the tree.

To do it easier we have compacted it.:

BDE

-numbers : 0.0322672 0.00358524 0.331318 0.0828295 0.0202829 0.00869268 0.0521024 0.468922

AT

-numbers : 0.0005 0.0099 0.0095 0.9801

LET

-numbers : 0.000572 0.054428 0 0 0.009828 0 0 0.935172

LBE

-numbers : 0.0315 0 0.0235 0 0.0043524 0.414148 0.0054756 0.521024

SLB

-numbers : 0.03 0.02 0.27 0.18 0.0015 0.0035 0.1485 0.3465

As we can see the values are the same.