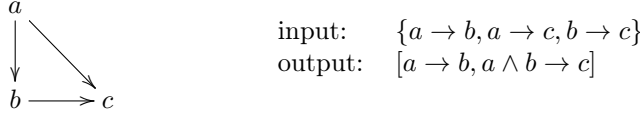


1 Network Structure

This algorithm attempts to find a meaningful way to present the *structure* of a Bayesian Network (BN). In the context of our work, we choose to see a directed edge of a BN as a causal relationship between two nodes, e.g. the edge $A \rightarrow B$ is interpreted as “ a causes b ”. Here, a is called the “*antecedent*” while b is called the “*consequent*”. Antecedents and consequents give us our first *requirement*: a consequent can be stated if and only if all of its antecedents have been stated before. In other words, nodes without antecedents can be seen as logical facts, and edges as logical implications with the extra requirement that all *antecedents* must be fulfilled. This means that the two edges $a \rightarrow c$ and $b \rightarrow c$ shall be seen as one implication $a \wedge b \Rightarrow c$, rather than as the two distinct implications $a \Rightarrow c$ and $b \Rightarrow c$.

1.1 Overview

Our algorithm takes as input a set of directed edges from a Bayesian Network, and outputs an *ordered list of rules* (see definition 1.8). The structure of the network is then explained by producing a text based on this list. For example:



Which can be read as “ a can cause b , and a and b can cause c ”. We do not address the translation to English here; Our main concern is to produce “a good enough” ordered list of rules.

1.1.1 Definitions

Definition 1.1 (*Good enough*)

Subjective notion mainly carved by what we (as a subset of the human beings) would like to avoid. In our case, a result is good enough when it fulfils a list of *requirements* that we will describe. A better *good enough* evaluation can be done through user studies.

Definition 1.2 (*Edges and nodes of a Bayesian Network*)

A Bayesian Network B is based on a directed graph. We note “edges(B)” its set of directed edges, and “nodes(B)” its set of nodes. An edge between two nodes a and b of B is noted $a \rightarrow b$, and if we have $c \rightarrow d \in \text{edges}(B)$, then we automatically have $\{c, d\} \subseteq \text{nodes}(B)$.

Definition 1.3 (*Antecedent*)

Let B be a Bayesian Network. For a node $a \in \text{nodes}(B)$, let N be the set of nodes such that $\forall n \in N, n \in \text{nodes}(B) \wedge n \rightarrow a \in \text{edges}(B)$.

We say that N is the set of “*antecedents*” of a , noted “ant(a)”.

Definition 1.4 (*Ground node*)

Let B be a Bayesian Network. A “ground node” a is a node without antecedent, i.e. we have ant(a) = \emptyset . We also say that a is “grounded”. Because Bayesian Networks are finite directed acyclic graphs, we have the guarantee that at least one node of any network is grounded.

Definition 1.5 (*Consequent*)

Let B be a Bayesian Network. For a node $a \in \text{nodes}(B)$, let N be the set of nodes such that $\forall n \in N, n \in \text{nodes}(B) \wedge a \rightarrow n \in \text{edges}(B)$.

We say that N is the set of “*consequents*” of a , noted “cons(a)”.

Definition 1.6 (*Descendants*)

Let B be a Bayesian Network. The “*descendants*” of a node $n \in \text{nodes}(B)$ is the set desc(n) made of its consequents cons(n), plus their consequents, and recursively.

Definition 1.7 (*Depth*)

Let B be a Bayesian Network, and let $n \in \text{nodes}(B)$. The “*depth*” of n is the cardinality of $\text{desc}(n)$, i.e. the total number of nodes “reachable” from n .

Definition 1.8 (*Rule*)

A rule has the general form:

$$x_1 \wedge x_2 \wedge \dots \wedge x_n \rightarrow y_1 \wedge y_2 \wedge \dots \wedge y_m$$

Following the previous definition, the x_i are called the *antecedents*, and the y_j the *consequents*. A rule is “*usable*” (or “*trigerrable*”) if and only if all of its antecedents are *available*. When a rule is used, its consequents become available, and ground nodes *can be made available* at any time.

Note the special condition over ground nodes: they are not available by default, but can be made available “on demand”. In the following examples, we usually consider that ground nodes are available. However, the distinction will be important when discussing ordering, Section 1.2.

1.1.2 Requirements

The definition 1.8 is a really general definition; In our case, rules must comply with several requirements that we are going to define now. We will justify them according to the definition 1.1. Let’s start with our introduction’s first requirement.

Warning: in the following, do not mix up rules and edges: Some requirements seems trivial but are necessary constraints as rules are not edges!

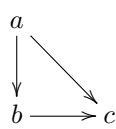
Requirement 1.1

A node from the Bayesian Network can be stated if and only if all of its antecedents have been stated before (i.e. are available). Given a set of initial grounded nodes, our output must be a list of “trigerrable” rules. When a rule is triggered, it makes its consequents available for the following rules.

Requirement 1.2

A node from the Bayesian Network can only appears once as a rule’s consequent in the output.

For example:

	$[b \rightarrow c, a \rightarrow c, a \rightarrow b]$	✗ Req. 1.1	b is not grounded
	$[a \rightarrow b, a \rightarrow c, b \rightarrow c]$	✗ Req. 1.2	c appears twice
	$[a \wedge b \rightarrow c, a \rightarrow b]$	✗ Req. 1.1	b is not grounded
	$[a \rightarrow b, a \wedge b \rightarrow c]$	✓	a is grounded

Requirements 1.1 and 1.2 together constrain the rules themselves: because all antecedents must be stated before a consequent, and because a consequent can only appears once, a rule always contains all the antecedents of its consequents. However, this does not prevent us from having extra antecedents in our rules.

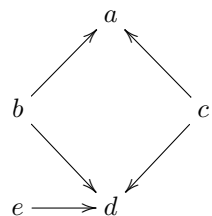
Requirement 1.3

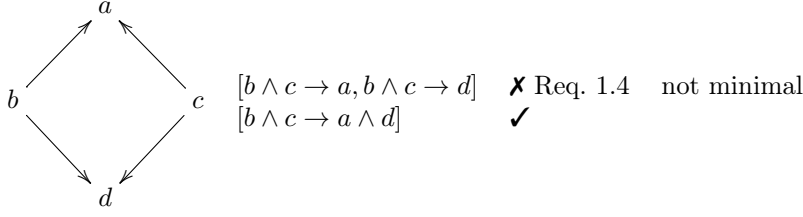
Given a node from a Bayesian Network, its set of antecedents in a rule are exactly the set of antecedents from the Bayesian Network, and nothing more.

Requirement 1.4

Our output list should be as small as possible.

Use of requirements 1.3 and 1.4 are illustrated below:

	$[b \wedge c \wedge e \rightarrow a \wedge d]$	✗ Req. 1.3	$e \notin \text{ant}(a)$
	$[b \wedge c \rightarrow a, b \wedge c \wedge e \rightarrow d]$	✓	

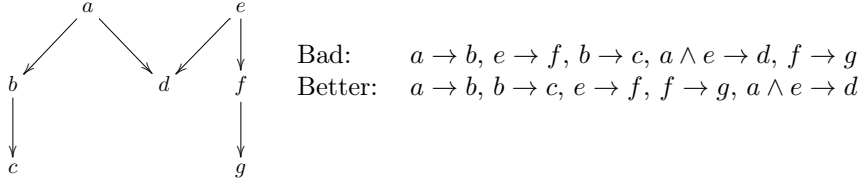


Those 4 first requirements are pretty natural and mainly address the rules creation and their basic ordering. However, they are far from enough as the basic ordering constraints allow to produce arguably poor explanations. As a first intuition, notice that the two lists:

$$[b \wedge c \rightarrow a, b \wedge c \wedge e \rightarrow d] \quad \text{and} \quad [b \wedge c \wedge e \rightarrow d, b \wedge c \rightarrow a]$$

are both acceptable according to the current requirements.

One of the main problem with the basic requirements is that they allow rules to be ordered in any way as long as they form a triggerable sequence: while the “overall ordering” of the rules is correct, there is no concept of “current subject”, i.e. of *grouping together related rules*. The explanation may “jump” from one part of the network to another, and then get back to the first part. This make the explanation hard to follow, hence not desirable. For example:



The nodes a and e are grounded which satisfy the dependencies of both b and f . If we state b , the dependencies for c and f are satisfied, i.e. we can now state f . However, this feels unnatural as we switch into an other branch of the network; it is better to state c immediately after b .

Requirement 1.5

The rules should be ordered “by branches”, i.e. by giving priority to the rules using the nodes made the most recently available.

This requirement tightens a bit more the field of possibilities. However, we still do not have a unique ordering. All the followings are possible:

$$\begin{aligned}
 &[a \rightarrow b, b \rightarrow c, e \rightarrow f, f \rightarrow g, a \wedge e \rightarrow d] \\
 &[a \rightarrow b, b \rightarrow c, a \wedge e \rightarrow d, e \rightarrow f, f \rightarrow g] \\
 &[e \rightarrow f, f \rightarrow g, a \rightarrow b, b \rightarrow c, a \wedge e \rightarrow d] \\
 &[e \rightarrow f, f \rightarrow g, a \wedge e \rightarrow d, a \rightarrow b, b \rightarrow c] \\
 &[a \wedge e \rightarrow d, a \rightarrow b, b \rightarrow c, e \rightarrow f, f \rightarrow g] \\
 &[a \wedge e \rightarrow d, e \rightarrow f, f \rightarrow g, a \rightarrow b, b \rightarrow c]
 \end{aligned}$$

At this stage, all 6 possibilities are acceptable. But again, some seem more “natural” than the other, e.g. alphabetical ordering (starting with the a node before the e node).

1.2 Order

Because our output is an ordered list of rules, the ordering is a crucial part of our algorithm. Of course, this is all heuristics, and other orderings would probably be equally good or better. We have two ways to control the order of the rule: First, we can choose which nodes are available, limiting triggerable rules. Second, the limited set of triggerable rules itself can be sorted. The final ordering is an interaction between all the ordering we are using in our algorithm.

Ground Nodes Ordering By controlling in which order the ground nodes are *made available* (cf definition 1.8), we can control how the explanation starts, and then partially control the flow of the explanation. The primary criterion is the depth of a node, in *decreasing* order (hence nodes with greatest depth come first). On tie, the alphabetical order is used. We use the depth as a proxy to the importance of a node (at least, visual importance in the graph); In this respect, starting with greatest depth allows to first focus on “big parts” of the network.

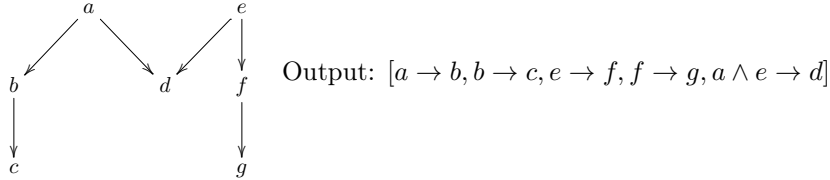
Consequents Ordering Consequents are sorted by *increasing* depth (so, in the opposite order of the ground nodes), and then alphabetically. After selecting a “big part” to talk about with the ground node, starting within that big part with small depth potentially allows to first state “visually obvious” things before digging further. When a rule is triggered, its consequents are made available one by one according to this ordering.

Rules Ordering Even when nodes are introduced in a controlled manner, several rules may be triggerable at the same time. To select them, we compute a score s . Let M be the cardinality of the largest consequent set among all rules:

$$s(x_1 \wedge \dots \wedge x_n \rightarrow y_1 \wedge \dots \wedge y_m) = n \times M + m$$

The rule with the lowest score is triggered; on tie, an **arbitrary rule is chosen**¹. The effect of this score is to compute a number from a 2 dimensions value ($|ant|, |cons|$), with the number of antecedents being the main criterion, hence the multiplication by the maximal cardinality of any consequent set. Choosing the lowest score follows the same heuristic as for the *Consequents Ordering*: we try to start with the “visually obvious” edges.

In particular, our ordering helps us to group nodes (requirement 1.5) by controlling the sequence in which nodes are made available. Let’s see how this work on our example.



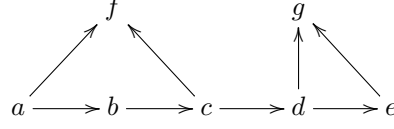
Explanation:

1. Both a and e have the depth (of 3). We make a available (alphabetical order).
2. $\{a\}$ is available, the only rule usable now is $a \rightarrow b$, making b available.
3. $\{a, b\}$ are available, and only $b \rightarrow c$ can be triggered, making c available.
4. $\{a, b, c\}$ are available, but no new rules can be triggered. We *backtrack* to our list of ground nodes.
5. $\{a, b, c, e\}$ are available. Both $e \rightarrow f$ and $a \wedge e \rightarrow d$ are triggerable. Comparing their scores leads to choose $e \rightarrow f$ ($1 * 1 + 1 < 2 * 1 + 1$).
6. $\{a, b, c, e, f\}$ are available, $f \rightarrow g$ is triggered because of its score.
7. $\{a, b, c, e, f, g\}$ are available, $a \wedge e \rightarrow d$ is triggered.

¹This may be improved in the future, mainly based on the antecedents alphabetical order — but first, see “Antecedents Ordering”.

1.3 Context

Now let's see an other example:



In that scenario, according to the ordering on rules, we will start with

$$[a \rightarrow b, b \rightarrow c, c \rightarrow d, d \rightarrow e]$$

Because we just finished with d and e , we would like to mention $d \wedge e \rightarrow g$ *before* $a \wedge c \rightarrow f$. However, the actual ordering does not guarantee that. Several solutions are possible. One could be to modify the score function to take into account when the antecedents have been introduced, penalising the rules with “old” antecedents (**not tested, just an idea**). An other solution, adopted in our case, is to use “*local contexts*”.

Context and Local Context Each rule r is triggered inside a “*context*” of available nodes.

When a rule r is triggered, its consequents form a new “*local context*”, which becomes the “current context”. When no rule can be triggered in the current context, it is “folded back” into the previous context, creating a new current context.

To denote the context, we use the following notation:

$$\{\text{current context}\} / \{\text{previous context}_n\}, \dots, \{\text{previous context}_0\}$$

Back to our example:

1. In $\emptyset /$, the only ground node a is selected
2. In $\{a\} / \emptyset$, the only rule is $a \rightarrow b$
3. In $\{b\} / \{a\}, \emptyset$, the only rule is $b \rightarrow c$
4. In $\{c\} / \{b\}, \{a\}, \emptyset$, the only rule is $c \rightarrow d$
5. In $\{d\} / \{c\}, \{b\}, \{a\}, \emptyset$, the only rule is $d \rightarrow e$
6. In $\{e\} / \{d\}, \{c\}, \{b\}, \{a\}, \emptyset$, no rule, folding back
7. In $\{d, e\} / \{c\}, \{b\}, \{a\}, \emptyset$, the only rule is $d \wedge e \rightarrow g$
8. In $\{g\} / \{d, e\}, \{c\}, \{b\}, \{a\}, \emptyset$, no rule, folding back
9. In $\{d, e, g\} / \{c\}, \{b\}, \{a\}, \emptyset$, no rule, folding back
10. In $\{c, d, e, g\} / \{b\}, \{a\}, \emptyset$, no rule, folding back
11. In $\{b, c, d, e, g\} / \{a\}, \emptyset$, no rule, folding back
12. In $\{a, b, c, d, e, g\} / \emptyset$, the only rule is $a \wedge c \rightarrow f$
13. In $\{f\} / \{a, b, c, d, e, g\}, \emptyset$, no rule, folding back
14. In $\{a, b, c, d, e, g, f\} / \emptyset$, no rule, folding back
15. In $\{a, b, c, d, e, g, f\} /$, End of process

1.4 Antecedents Ordering

The right-hand side of rules are the only unordered bit left! A nice (according to MH) property is to have the antecedents always presented in the same order, and in the order in which they were introduced.

We number the nodes in the order in which they are made available. Then, we sort the antecedents in the rule before presenting them according to this order.

1.5 PseudoCode

1.5.1 Creating the rules from the set of directed edges

From a of directed edges, it is easy to recover the set of all nodes nodes, and to construct the $\text{ant}(n)$ for every node $n \in \text{nodes}$. Note that ant for ground nodes is the empty set.

We first create the relation R_0 , which map the antecedents to the consequents. The ant relation gives us the conjunction of antecedents for a given node; R_0 allows to give us the conjunction of consequents sharing the same antecedents.

Algorithm 1: Creating the R_0 relation

Input: The set of nodes nodes

Input: The $\text{ant}(n)$ relation defined $\forall n \in \text{nodes}$

Result: A mapping R_0 between a sets of nodes ($\text{ant} \mapsto \text{cons}$)

By convention, $R_0[n] = \emptyset$ if the mapping is undefined.

```

1 for each node  $n \in \text{nodes}$  do
2    $R_0 \leftarrow R_0 + (\text{ant}(n) \mapsto R_0[n] \cup \{n\})$ 

```

From R_0 we create the set of rule R . Note that the consequents in the rule are sorted, hence they should be implemented on a ordered collection such as a list; the antecedents are not sorted yet so a set is fine.

Algorithm 2: Creating the set of rule R

Input: The relation R_0

Result: A set R of rules (see definition 1.8)

```

1 for each set of antecedents  $\text{ant} \in R_0$  do
2    $\text{cons} \leftarrow R_0[\text{ant}]$  ;
3    $\text{cons}' \leftarrow \text{Consequents\_Ordering}(\text{cons})$  (see Section 1.2) ;
4    $R \leftarrow R \cup \{\text{ant} \rightarrow \text{cons}'\}$ 

```

1.5.2 Getting the ground nodes

The ground nodes are in the relation R above, associated to the empty set. In other word, the set of ground nodes is $R[\emptyset]$. The result of $R[\emptyset]$ is actually an ordered collection following the Consequents Ordering. We have to order them according to the Ground Node Ordering.

Algorithm 3: Creating the ordered collection of ground nodes G

Input: The set of rules R

Result: An ordered collection of ground nodes

```

1  $G \leftarrow \text{Ground\_Nodes\_Ordering}(R[\emptyset])$ 

```

1.5.3 The main algorithm

The main algorithm is divided into two functions. Because the context idea is intrinsically recursive, we will define a corresponding recursive function. The other function is used to kick-start the process.

The general recursive function takes 3 “In-Out” parameters and one “In” parameter.

Algorithm 4: General recursive function sorting the rules

```

// Global
InOut: The ordered list of rules  $L$ 
InOut: The set of rule  $R$ 
InOut: The order in which node are introduced  $O$ 
// Local
Input: An ordered list of nodes to be added  $N$ 
Output: The local context of available node  $C$ 
1 Function getRulesRec( $L, R, O, N$ ):
2    $C \leftarrow \emptyset$ ;
3   for Next node in  $n \in N$  do
4     // Update the local context with next available node
      $C \leftarrow C \cup \{n\}$ ;
     // Get all rules matching the local context and sort them with the
       Rules Ordering
5      $MR \leftarrow \text{Rules\_Ordering}(\text{rule } r \in R \mid \text{ant}(r) \subseteq C)$ ;
6     for Next rule  $r \in MR$  do
7       // Remove the rule from the global set
        $R \leftarrow R - r$ ;
       // Order ant alphabetically; Add in  $O$  if absent
8       foreach  $a \in \text{Alpha\_Ordering}(\text{ant}(r))$  do
9         if  $a \notin O$  then  $O \leftarrow O + (a \mapsto O.\text{Size})$ ;
       // Sort the antecedents according to  $O$ , extends  $L$ 
10       $\text{ant}(r) \leftarrow O\_Ordering(\text{ant}(r))$ ;
11       $L.\text{Append}(r)$ ;
       // Recursive call, updating  $L, R$  and  $O$ 
12       $C \leftarrow C \cup \text{getRulesRec}(L, R, O, \text{cons}(r))$ ;
       // Update  $MR$  in the new context
13       $MR \leftarrow \text{Rules\_Ordering}(\text{rule } r \in R \mid \text{ant}(r) \subseteq C)$ ;

```

Explanation:

- L, R and O are global, i.e. updated across every call. L contains the final result, and grows as R shrinks: a rule taken out from R and put at the end of L , producing an ordered list. In the process, O allow to sort the antecedents of the rules according to the order in which they were introduced.
- After the recursive call, the local context C is updated with the local context of the recursive call, following Section 1.3. The set of Matching Rule MR must be updated as new rules may be triggerable.

1.5.4 The entry points

The entry points only allow to launch the main algorithm. It using the ordered list of ground nodes G instead of a consequents of some rule (see line 12 above).

Algorithm 5: Entry point for sorting the rules

Output: The ordered list of rules

```
1 Function getRules( $L, R, O, N$ ):  
2    $L \leftarrow$  new empty list ;  
3    $R \leftarrow$  result of Algorithm 2 ;  
4    $O \leftarrow$  Empty map from node to integer ;  
5    $G \leftarrow$  result of Algorithm 3 ;  
6   getRulesRec( $L, R, O, G$ ) ;  
   Result:  $L$ 
```

2 Network Segmentation

A segment is a Markov Blanket minus some nodes. A Markov Blanket is made of a “*central node*” (sometimes called the “*target node*” of the MB), and the nodes directly connected to it, plus the other causes of the effects nodes. Because it requires a central node, Markov Blankets (hence, segments) are built per node in the network. The challenge is to order those Markov Blanket correctly.

In our algorithm, we follow a “evidences to target” approach, where “target” refers to the “query” node. Some easy rules follow:

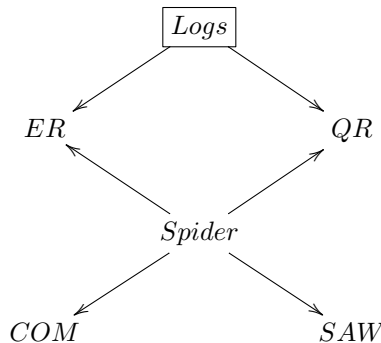
- Evidences are never used as a Markov Blanket’s central node as they will be included in the Markov Blanket of other nodes. This is also true for evidences that also are “common effect nodes”.
- If there is a “loop”, it must be broken somewhere. The breaking location may be arbitrary as we did not discuss specific breaking rules.

Let’s have an overview of the algorithm:

1. Compute the free paths from the evidences to the target.
2. Create a *tree* whose root is the target node. “Hang” paths on the tree, starting at the root/target node.
 - Paths with same prefix will share a common sub-branch.
 - However, common suffixes will be duplicated.
3. A special case of loop induced by common effect is handled here
4. Create an ordered list of nodes through a prefix run over the tree
5. Filter out evidence and duplicated
 - The filtering of duplicated node naturally breaks the loops
 - First nodes seen in the list are kept, other are removed
 - This may cause “ugly” breaking, and is not a controlled breaking: it indeed depends in the order in which paths were “hung” on the tree, hence of the tree itself, as it the base of the prefix run.
6. Go through the ordered list, creating the segments

2.1 Building the tree

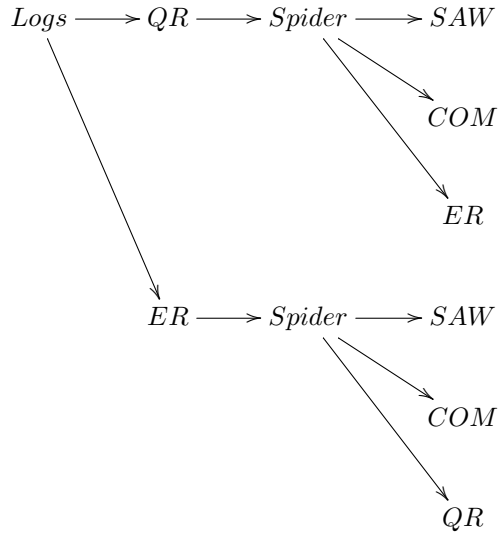
Let B be a Bayesian Network, $T \in \text{nodes}(B)$ the target node, and $\text{ev}(B) \subset \text{nodes}(B)$ the set of evidences. Let P be the set of free paths between the target T and every evidences $E \in \text{ev}(BN)$. Let’s take the spider network as an example, with $T = \text{Logs}$ and $\text{ev} = \{\text{COM}, \text{SAW}, \text{ER}, \text{QR}\}$



We have the following 8 free paths:

Logs – QR
 Logs – QR – Spider – SAW
 Logs – QR – Spider – COM
 Logs – QR – Spider – ER
 Logs – ER
 Logs – ER – Spider – SAW
 Logs – ER – Spider – COM
 Logs – ER – Spider – QR

Which give us the following tree:



Also, by construction, we have the guarantee that all path in P start with T .

Algorithm 6: Hanging a path in the tree

InOut: A tree or subtree t

Input: A path p

1 **Function** hang(t, p):

 // Precondition: $p.first = t$

2 $p \leftarrow p.removeFirst$;

3 **if** p has more than one node left **then**

4 $n \leftarrow p.first$;

5 **if** $t.hasChild(n)$ **then**

 // Get the child for n , then recurse

6 hang($t.getChild(n), p$);

7 **else**

 // Create a new child for n , then recurse

8 $c \leftarrow t.addChild(n)$;

9 hang(c, p);

10 **else**

 // p has only one node left: create the child if needed

11 $n \leftarrow p.first$ **if** $\text{Not } t.hasChild(n)$ **then**

12 $t.addChild(n)$

Result: Updated t

2.2 Special handling of a certain kind of loop

Warning: This has been made specifically for the Spider case, as it was the first requirement. It will work in all similar situation, but is not general at all. If anything, this part should be put in parenthesis until someone come up with something better (even if it is currently in the deployed version of the code).

The spider network exhibits a very special kind of “loop” between Logs, ER, QR and Spider. As a result, we can see in the tree that there is a certain amount of repeated nodes (all but logs), and that QR, Spider and ER play a special role in it. This because QR and ER are common effect nodes. In our algorithm, we implemented a special case to handle loops induced by a two common effects nodes.

NO OTHER KIND OF LOOP IS HANDLED SPECIFICALLY

To detect this special situation, we look if a set of sibling branches are sharing a two identical common effect nodes². Here, the node Logs have only two branches, if of them containing both *ER* and *QR*. Once we have the two nodes, we search a path (the direction does no matters) between them; Let’s call this set r . In our example that would be $ER - Spider - QR$ or $QR - Spider - ER$.

Algorithm 7: Find special loops

Input: A tree t obtained following the previous section

Output: A set r of path between “loop inducing” common effect nodes

1 **Function** findLoop(t):

2 $r \leftarrow \emptyset$;

3 **foreach** node $n \in t$ **do**

4 // For all possible couple of children of t

5 **foreach** $(c_1, c_2) \in \binom{t.children}{2}$ **do**

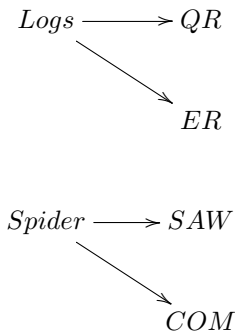
6 **if** c_1 and c_2 share a subset S of 2 common effect nodes **then**

$r \leftarrow r \cup \{t.findPath(S[0], S[1])\}$

Result: r

Let $p \in r$ be a path forming a special loop. Because the loops are created from two common effect nodes, p necessarily contains a node x with only outgoing edges. In our case, this is the spider node, e.g. $ER \leftarrow Spider \rightarrow QR$ ³. Select that node, and remove all of its occurrences from the tree t . Also copy one of those occurrence: we now have a second tree t' . Remove from t' all the nodes from the path p but x .

We now have a forest where the new trees come *after* t . This is important as this will influence the order of the nodes when doing a prefix run.



Note that Spider is absent from the first tree. However, Spider will be present in the Markov Blanket of Log, linking everything together.

²Again, this is not general at all, and if a loop with more than two common effect nodes show up, it will not be handled at all!

³We are only looking at the edges on the selected path; Spider could have ingoing arrows from elsewhere.

2.3 Prefix run and segments generation

Given a forest resulting from the previous section, we do a prefix run on the trees. This process outputs an ordered list of nodes. In general, the trees will contain some remains of “loops” in the form of duplicated nodes. Their position will be quite arbitrary as this depends on the order in which the paths have been “hung”.

In our example, the prefix run will give us the following list:

$[Logs, QR, ER, Spider, SAW, COM]$

Now, remove the evidences:

$[Logs, Spider]$

Finally, if there is duplicated nodes, only keep their first occurrence. The result is a list L of “central nodes”, i.e. nodes for which we are going to produce Markov Blanket and the corresponding segment. We go through the list L , and for each node we create its Markov Blanket. We remove from the Markov Blanket the previous *central node* we already processed, producing a segment. This may produce a segment with only one node, i.e. the central node. If this is the case, we ignore this segment, else we push the result on a stack.

Algorithm 8: Generating the segment from a list of central node

Input: A list L of central nodes

Output: A stack S of segments

```

1 Function getSegments( $L$ ):
2    $S \leftarrow \emptyset$ ;
3   foreach central node  $c \in L$  do
4      $m \leftarrow c.getMarkovBlanket$ ;
5      $s \leftarrow m - \text{previous central nodes}$ ;
6     if  $s.size > 1$  then
7        $S \leftarrow S.push(s)$ ;
  Result:  $S$ 

```

Traversing the stack from top to bottom gives us our ordered list of segments:

Spider, SAW, COM, QR ER

Logs, QR, ER, Spider