Mini-Project: Variable Elimination

Ana Oliveira 201503082
Bruno Casteleiro 201505347
Fábio Pereira 201506051

April 25, 2019

1 Introduction

As part of the course's practical work, we were asked to implement an algorithm related to Bayesian Networks (BN), more specifically, a variable elimination (VE) algorithm to perform inference in any BN. Inference in BNs is important as it is analogue to the "prediction" in other ML methods, i.e., with inference in BNs we are able to determine the probability of some event happening / some hypothesis being true in a given BN.

Note that we do not train any BN since that is not the goal of this project. We only produce a method that is able to make inferences on pre-trained BNs.

In this report, we first outline the code structure and how the user should proceed in the execution. We follow with an explanation of the VE algorithm and what was needed for its full implementation. Next, we compare some running times and memory usage of the procedure. Finally, we give our concluding remarks for this project and propose some improvements to this work.

2 Code Structure and Execution

For this project we decided to use R as the programming language. Our choice was based on some built-in functionality the language provides plus the availability of many useful packages such as bnlearn for Bayesian networks.

All the provided code was documented and it should suffice to answer any question regarding its use and behavior. Therefore, besides explaining any detail that we see as relevant, we won't provide any code snippet throughout the report.

2.1 Structure

The project is structured in five different R script files:

- **main.R**: The main script, it can be seen as the executable of our project. This code is intended to: 1. verify required packages are installed and loaded, 2. load required dependencies (other script code) and 3. handle user input and run the main program loop while it is still needed.
- **algorithm.R**: Where all the core variable-elimination code is implemented.

- **selection.R**: Where all the the variable elimination order selection code is implemented.
- **pruning.R**: Where an extra network-pruning procedure is implemented. More on this script later.
- **utils.R**: Useful functions needed in different parts of the project code.

Providing such organization has the advantage of allowing one to execute the code in both the required manner (as a program) but also from within a R session by manually sourcing the required scripts and calling the functions as needed (package-like behavior).

2.2 Execution

As stated in the last section, there are two possible ways to use our code, however, here we will only specify how to use it as a program (as required by the professor). It should be fairly simple to use it the other way for anyone who has basic knowledge of R.

2.2.1 Prerequisites

The user is required to have installed R alongside the packages bnlearn, dplyr, purrr and collections. If you are unsure to have any of the packages, you can still move on to the next section and execute the program, you will get a specific error for missing packages.

It is also required to have the desired Bayesian network file (*.bif* format) in the root directory of the project (where *main.R* is located).

2.2.2 Run

Open a terminal session in the directory where the project is located and follow these steps:

1. *$ cd VariableElimination/*
2. *$ Rscript main.R [--debug] [--prune]*

It should then be easy to follow the execution and answer any required input, we made sure to make an intuitive flow. The *--debug* flag is optional and, when set, outputs detailed information from every relevant function. We advise a first run with the flag unset, our code will still output the major results from each function. The *--prune* flag is also optional and, when set, removes irrelevant nodes from the network (with respect to the query and evidence inputs).

3 Procedure

To perform VE, there must first be an order in which to eliminate the variables. After the ordering and VE algorithms done, we stumped in a memory error for larger networks when executing the procedure, and so we saw the opportunity to implement a pruning algorithm to improve reliability. This algorithm, in theory, should also reduce the procedure execution time. In this section, we explain the 3 algorithms implemented by order of execution.

Note that we will only give a summary of what is implemented, the reader should still refer to the code for further insights. Again, documentation was provided for greater legibility and understanding.

3.1 Pruning

This algorithm will prune all the nodes from the network that wouldn't have an impact on the VE algorithm output. For that, we will not infer what nodes we can prune, but rather the nodes we can't prune – relevant nodes, i.e., the only ones that influence the VE output [1]. Given a query Q and evidence E:

- Q is relevant;
- if any node Z is relevant, its parents are relevant;
- if $A \in E$ is a descendent of a relevant node, then A is relevant.

We implement such definition in the following manner:

1. Mark all query nodes as relevant;
2. Mark all antecedents of query nodes as relevant;
3. For any evidence node that is descendent of a relevant node, mark it as relevant;
4. Mark all antecedents of relevant evidence nodes as relevant.

With the set of relevant nodes defined, we finally prune the network so that the only nodes left on the network are the relevant ones.

3.2 Ordering

The order by which VE does elimination is very important since, depending on the network one is working with, different orders may strongly alter the algorithm running time. It is known, however, that finding the best order for elimination is a NP-hard problem. What is commonly done to suppress such difficult problem is to resort to some heuristic cost functions and use a simple greedy procedure to retrieve an elimination order:

- Compute and store each node's heuristic value.
- Sort values by ascending order.
- Return node (e.g index) order according to sorted heuristic values.

This is precisely what we also do in our project. Technically speaking, our implementation follows the description above, using only a priority queue to simplify the code and time complexity.

Concerning the cost functions, we provide implementation for five different ones:

1. **min-table**: Cost equals the size of the node's probability table. For nodes with no dependencies, this will be the same as the number of different values the node (variable) can assume. If not specified, the algorithm assumes this heuristic by default.
2. **min-children**: Cost equals the node's number of children (descendants) in the graph.
3. **min-parents**: Cost equals the node's number of parents in the graph.
4. **min-neighbors**: Cost equals the node's total number of connections (children + parents) in the graph.
5. **min-weight**: Cost equals the product of weights (probability table sizes) of the node's neighbors.

These are not the only heuristics we could have used, so, with that in mind, we provided an implementation that is easily extendable: 1. Head to *selection.R*, 2. add *min-yourHeuristic* to the array of order options and 3. implement *minyourHeuristic* function receiving the network and node index as arguments and returing the cost value for the given node. You don't need to change anything in the selection algorithm itself, it is all dynamic.

3.3 Variable Elimination

We can finally, given a network (pruned or not), a query, evidence and order of elimination, proceed to do VE.

The implementation follows:

1. Create a list of nodes corresponding to each node of the network;
2. Condition the nodes' data given the evidence;
3. For each variable *var* in the elimination order:
   (a) Join tables corresponding to the nodes in which *var* appears;
   (b) Multiply frequency columns;
   (c) Sum frequencies by grouping the table by all columns except *var* and frequency, i.e., sum frequencies for which all variables except *var* are the same.
   (d) Append a new node with the freshly calculated table to the rest of the nodes and delete those in which *var* appeared.
4. Join all remaining tables (query and evidence);
5. Divide final non-normalized frequencies by their sum in order to normalize them, finally reaching the probabilities of each of the query variable's values.

4 Results

In this section, we will compare execution times and maximum memory necessary in 2 different network sizes – small and medium (11 and 37 nodes, respectively). We will also provide some notes on the obtained results.

Take into account that the execution times are in respect to the whole procedure (from pruning to VE), while the maximum memory necessary only represents the maximum allocated object inside the VE algorithm.

4.1 Small Network – SACHS

Three different queries were tested:

1. PIP2
2. Erk
3. Mek|Akt,Jnk,P38

4.1.1 Execution time comparison

The following table shows the execution times (in seconds) for each of the stated queries, considering each one of the heuristics.

Query (#nodes)	H1	H2	H3	H4	H5
Q1 (11)	0.05661	0.04781	0.05601	0.04721	0.04891
Q2 (11)	0.05701	0.04891	0.05731	0.04751	0.04921
Q3 (11)	0.04221	0.04390	0.03671	0.04471	0.04559
Pruned Q1 (3)	0.01050	0.01342	0.01060	0.01428	0.01380
Pruned Q2 (5)	0.02060	0.02401	0.02060	0.02351	0.02252
Pruned Q3 (8)	0.02991	0.03541	0.03001	0.03441	0.03471

We can see from the execution times that the pruning of the networks has a good impact on the time it takes to output the final result. We can also confirm that the more nodes pruned, the faster the procedure ran, as was expected. These same conclusions are seen throughout all the comparisons.

Furthermore, there doesn't seem to be a significant difference in the heuristics, probably due to the small network size.

4.1.2 Maximum memory needed comparison

The following table shows the maximum memory necessary (in bytes).

Query (#nodes)	H1	H2	H3	H4	H5
Q1 (11)	119744	33888	119744	33888	33888
Q2 (11)	119744	33888	119744	33888	33888
Q3 (11)	19632	33024	19632	33024	33024
Pruned Q1 (3)	4112	4208	4112	4208	4208
Pruned Q2 (5)	7808	13312	7808	11232	17376
Pruned Q3 (8)	12136	22960	12136	22960	20920

4.2 Medium Network – ALARM

Three different queries were tested:

1. VENTALV
2. VENTLUNG|CATECHOL
3. BP|PVSAT,MINVOL

4.2.1 Execution time comparison

The following table shows the execution times (in seconds).

Query (#nodes)	H1	H2	H3	H4	H5
Q1 (37)	8.90720	0.20805	0.39359	0.20845	0.19934
Q2 (37)	4.32547	0.22895	0.32097	0.19664	0.19634
Q3 (37)	0.76567	0.21544	0.33778	0.25234	0.20645
Pruned Q1 (8)	0.03871	0.04231	0.03861	0.03811	0.04081
Pruned Q2 (18)	0.10252	0.09872	0.09952	0.09652	0.10392
Pruned Q3 (25)	0.16493	0.15183	0.15744	0.14983	0.13966

From this, we can affirm that, for this particular case, the first heuristic is, by far, the worst when not pruning the network. It also happens that without pruning the network, the third heuristic is the second slowest one by a factor of two (when compared to the other 3 heuristics). This is probably due to the increased size of the network (number of nodes), and if so, that means that the bigger the network in which we are performing VE, the more difference in the heuristics' timings will be seen.

4.2.2 Maximum memory needed comparison

The following table shows the maximum memory necessary (in bytes).

Query (#nodes)	H1	H2	H3	H4	H5
Q1 (37)	39822048	104424	1334240	120024	104424
Q2 (37)	19915488	103072	1334240	103072	103072
Q3 (37)	3326688	103024	587400	411704	103024
Pruned Q1 (8)	19040	20032	19040	20032	23000
Pruned Q2 (18)	51264	51264	45888	45888	46144
Pruned Q3 (25)	187152	66000	127176	64392	64392

The same that could be seen in the time execution for this network, can also be seen memory-wise in the table above.

4.3 Notes

Recall that the pruning was implemented because, for somewhat big networks, our procedure would fail raising a memory error. We should make the reader aware that this happens inside the VE algorithm. This means that even with the pruning implemented, if not enough nodes are pruned, then the VE algorithm will still raise the memory error (unless there is enough memory available in the R session).

The reader should also know that for the results we averaged 10 execution times to be more confident of the values, since they were never constant.

Something curious that we also noted was that for any given query, the first execution time would always be slower (by a factor of 3) when compared to consecutive execution times. We think that this is probably due to some data being already cached in memory. Because of that, we never consider the first execution time for any test.

5 Conclusion

In this work, we implemented a procedure to infer probabilities in BN. We started by doing the VE algorithm, and then proceeded to build layers upon it. First, the ordering algorithm, that provides the order of the nodes to eliminate. Then, realizing that the VE by itself would fail for medium+ size networks, we implemented a pruning algorithm so that the VE would execute on possibly smaller networks and therefore fail less often.

We then assessed the performance of the VE algorithm by itself versus with the help of pruning, and verified that the pruning indeed helps the procedure, diminishing the timings throughout the tests. We also compared the heuristics and verified that for the medium size network, the difference in the heuristics timings were starting to be noticed.

In further work, it should be tested the hypothesis that the bigger the network, the more differences would result in the heuristics timing / memory and even determine the order of quality for the heuristics. That can only be done with enough memory or better memory management in the VE algorithm.

References

[1] Pedro Pacheco
CSC384: Lecture 11,
http://www.dgp.toronto.edu/~ppacheco/course/384/Lectures/Lecture11Notes.pdf