AI PRINCIPLES & TECHNIQUES

Variable Elimination Algorithm

TIM JANSSEN *s1125701*
BRITT STOFFELS *s1054953*
Radboud University *21-12-2023*

Contents

1 Introduction

This report will give an overview of implementing the Variable Elimination (VE) algorithm for exact inference in probabilistic graphical models, particularly Bayesian networks. The main purpose of the VE algorithm is to compute marginal probabilities efficiently. It will produce a thorough understanding of Variable Elimination and its applications in probabilistic graphical modeling. In pursuit of these objectives, we implemented the VE algorithm and applied it to analyze two distinct Bayesian networks.

2 Method

The following chapter explains the methods that this project used to achieve efficient calculation of marginal probabilities in Bayesian networks. Given a Bayesian network, which consists of nodes representing random variables and edges representing probabilistic dependencies, and a set of conditional probability distributions associated with each node, the problem is to efficiently calculate marginal probabilities. Variable elimination involves iterative elimination operations, specifically the product operation (combining factors) and marginalization operation (summing out variables). VE also involves identifying:
1. The variable for which you want to compute the marginal probability
2. Any observed or known values (evidence) for other variables in the network.
- If there is evidence for some variables: reduce the factors by setting evidence values. remove rows inconsistent with the evidence.

These strategies may incorporate heuristics to streamline navigation through the solution space. In essence, Variable Elimination (VE) algorithms present a systematic approach to efficiently explore and discover solutions that fulfill a sophisticated array of constraints.

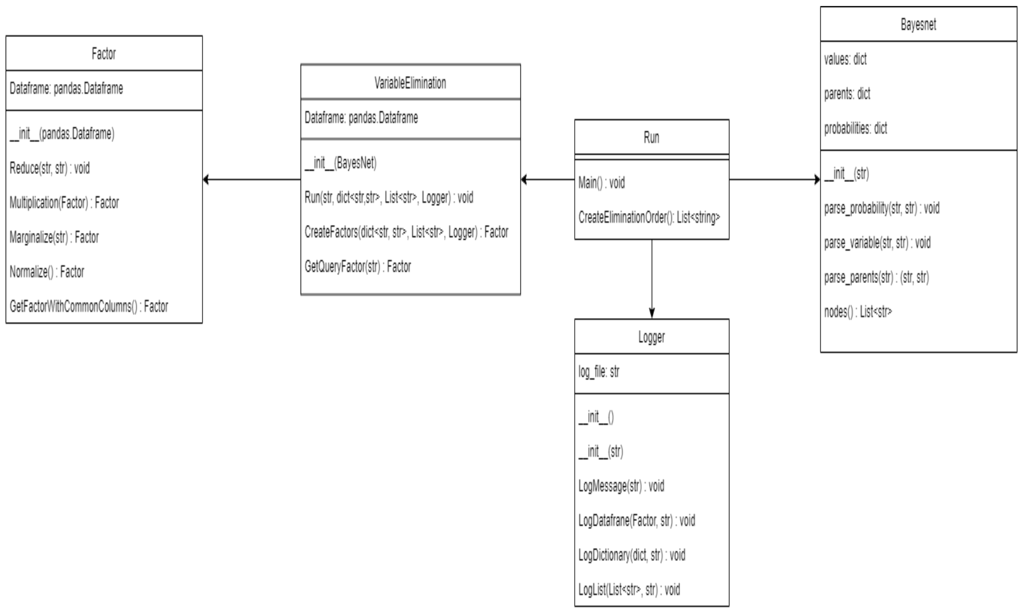
2.1 Specification

The given assignment consists of 4 general aims:
1. Get a thorough understanding of inference in Bayesian networks, by using Variable Elimination, and applying it on inference queries in different Bayesian networks.
2. Implement multi-dimensional factors and do the necessary calculations on them (reduction, product, marginalization), building on this to implement VE.
3. Experimenting with the processing of non-binary variables.
4. Experimenting with different heuristics for the elimination order.

The output of the code should be a log file, containing the steps the algorithm does on a particular network with a particular elimination ordering. This will contain: query/evidence variables, the factors, the elimination order, which factors are processed, amount of multiplications and amount of marginalizations. In essence it will contain every step the program does to get to the resulting factor.

2.2 Design

The program is written in an object-oriented manner. This style ensures enhanced readability and simplifies maintenance. This method also guarantees code clarity and facilitates seamless collaboration by separating concerns within the project.



Factor
Dataframe: pandas.Dataframe
__init__(pandas.Dataframe)
Reduce(str, str) : void
Multiplication(Factor) : Factor
Marginalize(str) : Factor
Normalize() : Factor
GetFactorWithCommonColumns() : Factor
VariableElimination
Dataframe: pandas.Dataframe
__init__(BayesNet)
Run(str, dict<str,str>, List<str>, Logger) : void
CreateFactors(dict<str, str>, List<str>, Logger) : Factor
GetQueryFactor(str) : Factor
Run
Main() : void
CreateEliminationOrder(): List<string>
Bayesnet
values: dict
parents: dict
probabilities: dict
__init__(str)
parse_probability(str, str) : void
parse_variable(str, str) : void
parse_parents(str) : (str, str)
nodes() : List<str>
Logger
log_file: str
__init__()
__init__(str)
LogMessage(str) : void
LogDatafrane(Factor, str) : void
LogDictionary(dict, str) : void
LogList(List<str>, str) : void

Figure 1: Class diagram of Variable Elimination

Figure 1 represents the class diagram of the VE project that vividly captures the intricate relationships and interactions among various classes. The Python template has undergone enhancements, particularly in the "Factor" class, where comprehensive logic for variable elimination has been consolidated. Additionally, the "Logger" class encapsulates the essential logic for data logging to a file. The "Variable Elimination" class, serving as the core component, has undergone modifications and expansions to incorporate the Variable Elimination algorithm seamlessly.

The following figure demonstrates the pseudocode for the implemented VE algorithm. The provided pseudocode is sourced from "Exact Inference in Bayes Nets – Pseudocode Page 2" by MIT CSAIL (n.d.).

Pseudocode for the VE algorithm

```
function ELIMINATION−ASK(X, e, bn) returns a distribution over X
    inputs: X, the query variable
         e, observed values for some set of variables E
         bn, a Bayes net
    factors <− [for each variable v in bn.VARS, the CPT for v given
        e]
    for each var in bn.vars if var is not in e and var is not X do
        relevant−factors <− [all factors that contain var]
        factors.remove(relevant−factors)
        factors.append(SUM−OUT(var, POINTWISE−PRODUCT(relevant−
            factors)))
 return NORMALIZE(POINTWISE−PRODUCT(factors))
```

Note. Adapted from *Exact Inference in Bayes Nets – Pseudocode* Page 2, by MIT CSAIL, (n.d.), link to page.

Figure 2: Pseudocode of the implemented VE algorithm

The implementation of the VE algorithm adheres closely to the provided steps from the pseudocode in figure 2. The clarity of the pseudocode is important to achieving a successful VE algorithm. The utilization of factors in the implementation is an insight taken from the pseudocode. The Variable Elimination algorithm in Bayesian networks is a sound and complete method, ensuring exact solutions for inference problems. However, it is essential to note that the efficiency of VE in finding the solution can vary based on factors such as the structure and size of the Bayesian network, the chosen elimination order, and the complexity of the probabilistic dependencies.

2.3 Implementation

As mentioned before, the code closely resembles the pseudocode in figure 2. The main algorithm loop iterates through the specified order of node elimination. Within each iteration, a while loop processes the factors associated with the current node, multiplying two factors at a time and appending the resulting factor back to the list. The process continues until the length of the list is reduced to one. After the while loop, a check determines whether the length of the current node's list is one. If so, the last remaining factor is marginalized and added to the next node's list. If the current node is the last in the order, the marginalized factor is added to its own list.

Finally, a loop traverses the dictionary to inspect the size of each node's list. The algorithm concludes when a non-empty list of factors is encountered. The algorithm retrieves the factor corresponding to the query variable and multiplies it with the resulting factor of the algorithm. Following the multiplication, the resulting factor undergoes normalization. The counters 'multiplicationCounter' and 'marginalizationCounter' track the number of multiplications and marginalizations performed throughout the algorithm.

In the code snippets, the comments and logging have been removed for readability.

VE algorithm

```
factors = self.CreateFactors(observed, elim_order, logger)
multiplicationCounter = 0
marginalizationCounter = 0
for elim in elim_order:
    while factors[elim].__len__() > 1:
        fac1 = factors[elim].pop()
        fac2 = fac1.GetFactorWithCommonColumns(factors[elim])
        factors[elim].remove(fac2)
        multiplied_factor = fac1.Multiplication(fac2)
        multiplicationCounter+=1
        factors[elim].append(multiplied_factor)

    if factors[elim].__len__() == 1:
        marginalized_factor = factors[elim].pop().Marginalize(elim)
        marginalizationCounter+=1
        if elim in factors.keys():
            if  elim_order.index(elim)+1 != elim_order.__len__():
                next_variable = elim_order[elim_order.index(elim) +
                    1]
                factors[next_variable].append(marginalized_factor)
            else:
                factors[elim_order[-1]].append(marginalized_factor)
        else:
            factors[elim_order[-1]].append(marginalized_factor)
for key, value in factors.items():
    if  value.__len__() == 0:
        continue
    else:
        query_factor = self.GetQueryFactor(query)
        result_factor = value[0].Multiplication(query_factor)
        multiplicationCounter+=1
        result_factor_normalized = result_factor.Normalize()
        return result_factor_normalized
```

In the 'CreateFactors' function, a dictionary named 'factorsdictionary' is initialized to store factors associated with specific nodes. Each factor is linked to its respective node through the dictionary keys.

Additionally, a list named 'tempProbabilities' is created within the function to prevent the generation of duplicate factors. This list is a copy of the original network probabilities and is employed to ensure that each factor is only created once.

CreateFactors function

```
def CreateFactors(self, observed, elim_order, logger):
    factorsdictionary = dict()
    tempProbabilities = self.network.probabilities.copy()
    for node in elim_order:
        probabilities = []
        for value, prob in self.network.probabilities.items():
            if tempProbabilities.keys().__contains__(value):
                if  prob.columns.__contains__(node):
                    new_factor = Factor(prob)
                    keys = observed.keys()
                    for key in keys:
                        if  new_factor.Dataframe.columns.
                            __contains__(key):
                            new_factor.Reduce(key, observed[key])
                    probabilities.append(new_factor)
                    del tempProbabilities[value]
        factorsdictionary[node] = probabilities
    return factorsdictionary
```

In the "Multiplication" function, the current factor is multiplied by another factor using pandas DataFrame functions. The function identifies common columns between the two factors, performs the multiplication operation, and returns the resulting factor.

Multiplication function from factor class

```
def Multiplication(self, other : 'Factor'):
    common_columns = list(set(element for element in self.Dataframe.
        columns if element != 'prob')
                        .intersection(element for element in other
                            .Dataframe.columns if element != 'prob
                            '))

    if(common_columns.__contains__("prob")):
        common_columns.remove("prob")

    if not common_columns:
        print("No common columns found.")
        return self

    merged = self.Dataframe.merge(other.Dataframe, on=common_columns
        , suffixes=('_self', '_other'))
    merged['prob'] = merged['prob_self'] * merged['prob_other']
    merged = merged.drop(columns=['prob_self', 'prob_other'])

    return Factor(merged)
```

In the "Marginalize" function, the current factor is marginalized based on the variable provided as a parameter. The function drops the columns corresponding to the specified variable and returns the resulting factor after summing the probabilities.

Marginalize function from factor class

```
def Marginalize(self, variable : str):
    if not self.Dataframe.columns.__contains__(variable):
        return self

    dtf = self.Dataframe.drop(variable, axis=1)
    vars = [col for col in dtf.columns.tolist() if col != "prob"]
    dtf = dtf.groupby(vars, as_index=False)["prob"].sum()

    if isinstance(dtf, pandas.Series):
        dtf = pandas.DataFrame(dtf)

    return Factor(dtf)
```

The "Normalize" function serves to normalize the values within the current factor. This is achieved by dividing each probability in the factor by the total sum of the probability column.

Normalize function from factor class

```
def Normalize(self):
    total_prob = self.Dataframe['prob'].sum()

    if total_prob != 0:
        self.Dataframe['prob'] = self.Dataframe['prob'] / total_prob

    return self
```

3 Testing

The time complexity of the Variable Elimination (VE) algorithm depends on various elements, including the structure of the Bayesian network, the chosen elimination order, and the specific implementation details. To calculate the runtime of an algorithm or runtime disparity between different heuristics, one can also employ the Big O notation. The Big O notation is a mathematical notation used to describe the performance or (time) complexity of an algorithm concerning the size of the input. It does so by providing an upper bound on the growth rate of an algorithm's runtime concerning the size of the input. It can help with comparing the performance of different algorithms or adding heuristics within an algorithm.

The VE algorithm, in its basic form without the inclusion of heuristics or non-binary processing, exhibits a worst-case time complexity of $\mathrm{O}(nd^2)$, where n represents the number of variables, and d signifies the maximum domain size. This scenario can be caused by the choice of elimination order, like a random elimination order, where intermediate factors could become larger, leading to a higher computational cost. Nevertheless, a random elimination order may still end up leading to the best-case time complexity of VE. Its best-case complexity is outlined as $\mathrm{O}(nd)$, which can be achieved by choosing the elimination order such that the size of the intermediate factors is consistently minimized. This could be accomplished by employing heuristics to guide the selection of an elimination order.

However, it is important to note that finding the truly optimal elimination order is an NP-hard problem. This means that, in the general case, there is no known polynomial-time algorithm to find the most optimal elimination order. Heuristics are used as approximation methods to find good, but not necessarily optimal elimination orders in a more reasonable amount of time.

Furthermore, when testing the algorithm with the AISpace that was supplemented by the university, the outputs did not match. While we did get the same result for the tampering network, this was not the case for the earthquake network. This dissimilarity seems attributable to a mistake with the rounding of the numbers. However, this is not certain.

4 Results

The data in Table 1 and Table 2 is derived from the log text file that is computed in the program.

Table 1: Results of the tampering.bif

Tampering	Prob
True	0.028436
False	0.971564
Amount of multiplications: 52	Amount of marginalizations: 3

Table 1 illustrates the normalized factor outcomes for the Tampering Bayesian network, revealing a 97.16% probability of 'Tampering' being 'False' and 2.84% probability of it being 'True.' The table also includes the computational details, indicating 52 multiplications and 3 marginalizations performed during the Variable Elimination algorithm execution.
In Big O notation, you might express the time complexity as O(M + N), where M is the number of multiplications and N is the number of marginalizations performed across all queries. So, separately for Tampering, the time complexity is O(55).

Table 2: Results of the earthquake.bif

Alarm	Prob
True	0.050374
False	0.949626
Amount of multiplications: 32	Amount of marginalizations: 3

Table 2 presents the normalized factor outcomes for the Earthquake Bayesian network, showcasing a 94.96% probability of 'Alarm' being 'True' and a 5.04% probability of it being 'False.' The table also includes computational details, specifying 32 multiplications and 3 marginalizations executed during the Variable Elimination algorithm. For Earthquake, the overall time complexity expressed as O(M + N) is O(35)

5 Discussion

Comparing the computational complexities of the two Bayesian networks, the Tampering network exhibits a higher demand with a complexity of 55 (52 multiplications and 3 marginalizations), while the Earthquake network has a complexity of 35 (32 multiplications and 3 marginalizations). This disparity suggests that the Tampering network involves a more intricate interplay of variables and dependencies, making it computationally more demanding for the Variable Elimination algorithm.
In terms of probabilities, the Variable Elimination algorithm for Tampering reveals a substantial 97.16% probability of 'Tampering' being 'False.' This high probability underscores the significant impact of the network structure and contextual characteristics, suggesting a prioritization of scenarios where tampering events are infrequent. Similarly, for the Earthquake network, the algorithm indicates a high 94.96% probability of 'Alarm' being 'True,' emphasizing the substantial impact of the network structure and context. These results highlight the significant influence of the inherent characteristics of each Bayesian network on the outcomes uncovered by the Variable Elimination algorithm. The consistent computational effort across both networks underscores the efficiency of the algorithm in navigating diverse solution spaces shaped by these distinctive network features. The observed differences in computational complexity and probability distributions collectively suggest that the Tampering network, with its higher complexity and dominance of the 'False' state, presents a more intricate and nuanced representation of the underlying real-world scenarios compared to the Earthquake network.

6 Conclusion and Reflection

In summary, our exploration of the Variable Elimination (VE) algorithm in Bayesian networks provided valuable insights into probabilistic inference. Despite successful achievements in understanding multi-dimensional factors and navigating binary variables, time constraints prevented the exploration of non-binary variable processing and experimentation with elimination order heuristics. These unexplored aspects highlight opportunities for future research and development, acknowledging the dynamic nature of probabilistic reasoning.
Within this exploration, the Tampering network revealed higher complexity, indicated by a 97.16% probability of 'Tampering' being 'False,' emphasizing infrequent tampering events. In contrast, the Earthquake network, with a 94.96% probability of 'Alarm' being 'True,' demonstrated efficient uncovering of probabilistic relationships. The varying complexities of these networks, reflected in distinct probabilities, underscore the algorithm's adaptability to different structures. While certain aspects remained unexplored, this experience highlights the complexities of algorithmic implementation in Bayesian network inference, providing foundational insights for future research.

Writing the program revealed two interesting observations:
1. **Amount of marginalizations & products** $\rightarrow$ The amount of marginalizations and products needed differed between the two different Bayesian networks. More marginalizations and products generally imply a higher computational complexity, suggesting a more complex Bayesian network. In the case of this project, the tampering Bayesian network seems more complex, with a more intricate network structure.
2. **The effect of nodes** $\rightarrow$ Expanding on the earlier observation, a network with more nodes leads to an increased number of factors, consequently requiring more multiplications. The number of marginalizations corresponds to the nodes slated for elimination from the factor dictionary. Hence, the count of marginalizations can be minimized by leveraging the observed variables.

Navigating through the complexities of the VE algorithm proved to be a shared challenge for both partners in our collaborative project. Despite having a solid grasp of the theoretical background of the VE algorithm, we encountered significant challenges when transitioning from theory to implementation. Understanding how to translate intricate theoretical concepts into a functional program proved to be a complex task. Throughout this journey, we have worked to enhance our Python proficiency, but it's important to note that, as learners, we are not yet experts. Therefore, while we strive for accuracy and precision in our implementation, some mistakes may occur due to the ongoing learning curve in both algorithmic concepts and programming expertise."

Work division: Our collaboration continues to thrive on complementary skills. Comparable to our last two partnerships, Tim took the lead in programming, as did Britt in writing the report. Building upon the successes of our previous assignments, we have both experienced growth in our skills in academic writing and programming. We have also gained more confidence in our abilities and remain committed to continuously improving our skills throughout other courses.

7 References

MIT CSAIL. (n.d.). Exact Inference in Bayes Nets – Pseudocode. MIT Computer Science and Artificial Intelligence Laboratory.
Retrieved from https://courses.csail.mit.edu/6.034s/handouts/spring12/bayesnets-pseudocode.pdf