AI PRINCIPLES & TECHNIQUES

Assignment 3: Variable Elimination

Nick van Oers *s1009378*
Jord Cluitmans *s1052807*
Radboud University *December 2022*

Contents

1 Introduction

When computing probabilities in Bayesian networks, the order of operations can make a large difference in the runtime and memory complexity of the computation. This effect can be really large and makes the complexity worst exponential. An algorithm that can help with this is the Variable Elimination algorithm. It aids to find an efficient ordering of computations and then finally to actually calculate the probability by using this order. In this assignment we will implement this algorithm and see what it does on a particular elimination ordering.

2 Implementation

We implemented the Variable Elimination algorithm by defining two classes: `variable_elim` and `factor`. To execute the variable elimination algorithm, you need to create an object of `variable_elim`. The `factor` class is only called in the `variable_elim` class so it does not need an object when executing the variable elimination algorithm. The variable elimination algorithm can be executed by using the `run` method defined in the `variable_elim` class. This method takes three arguments: Query, Observed and EliminationOrdering. Query contains the query for which the probability needs to be computed. Observed takes a list of variables that already have been observed. EliminationOrdering takes a list that contains an elimination ordering. This method returns a `factor` class object with the CPT of the queried variable.
A `factor` object takes a CPT and network as arguments. It has the three main functions `product`, `marginalize`, `reduce`. These functions correspond to the operations that are needed for the variable elimination algorithm. All of these functions return a new `factor` object. The original objects are not altered.
The `reduce` function removes all variable values from the DataFrame except for the value given as parameter. This has as a result that the `product` function first goes through the variable values that are left in the DataFrame and only uses those in the multiplication. Since we are using DataFrames and our class structure, our implementation of the

algorithm can also take non-binary values as input. Below is the `product` function that computes the product of two factors.

```
def product(self, factor):
    """
    Computes the product of this factor and the factor parameter

    return: A new factor
    """
    variables = []
    for col in self.table.columns.values[:-1]:
        variables = variables + [col]
    for col in factor.table.columns.values[:-1]:
        if col not in variables:
            variables = variables + [col]

    df = pd.DataFrame(columns=variables + ['prob'])

    self.fillDataframe(df, variables, [], factor)

    return Factor(df, self.net)
```

You can see that the function only computes the columns that are actually required. From there it fills an empty DataFrame using the `fillDataframe` method. This methods computes a table that holds the new values of the factors of which we have computed the product. Below is the `fillDataFrame` method.

```
def fillDataframe(self, dataframe: pd.DataFrame, variables, values, factor
    ):
    """
    Computes the product of a vector by recursively filling the dataframe
        parameter
    """
    if len(values) == len(variables):
        dict = {}
        for i in range(0, len(variables)):
            dict[variables[i]] = values[i]

        prob1 = self.getProbability(dict)
        prob2 = factor.getProbability(dict)
        dataframe.loc[len(dataframe)] = values + [prob1 * prob2]
    else:
        current_variable = variables[len(values)]
        all_values = self.getValues(current_variable)
        if all_values is None:
            all_values = factor.getValues(current_variable)
        for value in all_values:
            self.fillDataframe(dataframe, variables, values + [value],
                factor)
```

The variables parameter contains all of the variables that need to be included inside the DataFrame. The values parameter contains an empty list. The empty list will eventually be filled up with values of the variables. This is done using recursive calls. A possible recursive call of the function could look like the following:

$$\texttt{variables} = [Burglary, Earthquake, Alarm], \texttt{values} = [True, False, True]$$

This call means that variable *Burglary* has a value of $True$, *Earthquake* has a value of $False$ and *Alarm* has a value of $True$. The position of the values in the list corresponds to each value in the list of variables. This means that if the list of values has the same length as the list of variables, all variables are assigned a value. After that, the function will assign probabilities to all variables based on the products that were computed.

Now that we know how the `factor` classworks, we can look into the implementation of the Variable Elimination algorithm. The code for this is made in the `run(self, query, observed, eliminationOrder)` and looks as follows

```
def run(self, query, observed, eliminationOrder):
    """
    Use the variable elimination algorithm to find out the probability
    distribution of the query variable given the observed variables

    Input:
        query:       The query variable
        observed:    A dictionary of the observed variables {variable:
            value}
        eliminationOrder: A list specifying the elimination ordering

    Output: A factor holding the probability distribution
            for the query variable

    """

    self.file.write("Starting with query=" + query + ", observed=" + str(
        observed) + " and elimination ordering=" + str(eliminationOrder) +
        ".\n")
    notReduced = [Factor(x, self.network) for x in self.network.
        probabilities.values()]
    self.file.write("Required factors: \n" + str(notReduced) + "\n")
    factorsCopy = self.reduceFactors(notReduced, observed)
    self.file.write("Reduced factors: \n" + str(factorsCopy) + "\n")

    for variable in eliminationOrder + [query]:
        self.file.write("Variable to be eliminated: " + variable + "\n")
        toMultiply = []
        for factor in factorsCopy:
            contains = False
            for column in factor.table.columns.values[:-1]:
                if column == variable:
                    contains = True
            if contains:
                self.file.write("Factor " + str(factor) + " added to
                    multiplication list.\n")
                toMultiply = toMultiply + [factor]

        for factor in toMultiply:
            factorsCopy.remove(factor)
        if len(toMultiply) >= 1:
            newFactor = toMultiply.pop()
            while len(toMultiply) > 0:
                newFactor = newFactor.product(toMultiply.pop())
            self.file.write("After multiplication: " + str(newFactor) + "\
                n")
            if variable is not query:
                newFactor = newFactor.marginalize(variable)
                self.file.write("After marginalization: " + str(newFactor)
                    + "\n")
            factorsCopy = factorsCopy + [newFactor]
            self.file.write("Resulting factor: " + str(newFactor) + "\n")
        self.file.write("Factors after elimination of " + variable + ": "
            + str(factorsCopy) + "\n")

    returnValue = factorsCopy[0]
    returnValue = self.normalizeFactor(returnValue)
    self.file.write("Done!")
    return returnValue
```

We first create a list of all factors named `notReduced`. From this list, we only need the parent factors of the query variable and parent factors of the evidence variables, so we delete all the unnecessary factors and save them in the list `factorsCopy`. Next we go into a loop that goes over the elimination ordering with the query variable to be eliminated last. During every iteration of the loop, we collect all factors with a column of the variable that is to be eliminated in them. These will be saved in the list `toMultiply`. We then remove these factors from out `factorsCopy` list, since they will be replaced by a single factor after the iteration. We then compute the product of all factors in the `toMultiply` list marginalize the factor that it computes. After this loop we then only need to normalize our final remaining factor before returning it.

3 Discussion

Our algorithm seems to work completely fine, but we did not properly check this. By designing unit tests for our program, we could have verified that our program did exactly what we would expect. Next to this we also did not keep efficiency too much in mind when designing our code. This means that the complexity of our implementation can most likely be improved.

4 Conclusion

To conclude we find that the Variable Elimination algorithm is a useful algorithm to efficiently calculate probabilities in Bayesian networks. It helps with finding an efficient order of computations and then calculates the correct (conditional) probabilities.