AI PRINCIPLES & TECHNIQUES

Variable Elimination Report

Ege SARI *s1034535*
Radboud University *7 January 2022*

Contents

1 Introduction

In this section we will explain about the project by giving the goals of it.

1.1 About the Project

The main goal of this project is to get an understanding of inference in Bayesian networks. This main goal consists of two sub-goals. The first one is to design and implement multi-dimensional factors with factor operations such as reduction, product, and marginalization. The second sub-goal is to design and implement variable elimination and test it with various networks. In addition to these sub-goals, implementation of different heuristics for elimination order is given as a bonus sub-goal, but it is not done in this project.

In the next sections, the design and implementation of factors, factor operations, and the variable elimination algorithm are explained in detail. Since the complexity of the variable elimination algorithm is highly dependent on the elimination order, and in this project, different heuristics for elimination order is not implemented, the complexity of the variable elimination is not discussed.

2 Design and Implementation

In this section, we will explain the design and implementation of the factors, factor operations, and variable elimination.

2.1 Design and Implementation of Factor

It can be said that a factor is a useful data structure such that it can carry data of probability tables, and some operations can be applied on it. It carries the variables, and the probability row(s), which consists of values and matching probabilities.

The optimal implementation choice to create both the factor itself and the related operations is to create a class named "Factor". By this choice, a factor object and the operations can be accessed by calling the Factor class. Even though in design a specific attribute for probability is not mentioned, for implementation it is needed. As mentioned before, the probability rows, let us say tables, saves the probability values. But after some operations such as marginalization, when a factor with only one variable is marginalized by its variable, the probability table becomes empty. But the probability value is important for marginalization, so it must be saved. So it is needed to create a double attribute called "probability" for these situations. These situations are described in detail in Section 2.2.2.

For implementing other data attributes, some structures from the template are used. For example for the probability rows, the "Table" object is used. For the variables, one of the appropriate data structures to save it is **ArrayList**. Other alternatives could also be used such as **List** or **Queue** or **Array** but since in the given template, **ArrayList** was used, and using the same data structures provides an advantage for time issues.

The design choice to initialize a factor is implementing constructors. There are three constructors used for different purposes. The first constructor is used for factor operations. It gets a list of variables and probability rows. By these variables, it sets the variable list attribute of the factor. Later, again by using variables and the probability

rows, it creates a table object to set the table attribute of the factor. To create the table, it separates the first element of the variable list and the remaining part. Then it sets the remaining part as the parent list of the first element. This design is chosen since it is the most suitable way for the given template. The pseudo-code of the algorithm used for making the table from the variable list and probability lists can be found in the Algorithm 1.

Algorithm 1 Algorithm for Building a Table

```
procedure MAKETABLE(Variable[] vars, ProbRow[] rows)
    Variable first ← vars[0]
    Variable[] parents ← ∅
    for int k = 1, k in size (vars) do
        parents.add(vars[k])
    end for
     first.setParents(parents)
    return Table(first,rows)
end procedure
```

The second constructor is used only once, for the variable elimination algorithm. It gets the table as input and transfers the variables in the table to the variables attribute of the factor. Of course, it sets the table attribute of the factor to this input table.

The third constructor is used for the blank factors, where it does not have a variable anymore (its variable may be summed out by marginalization operation). But since it has no variable, it means its table is also lost. On the other hand, for variable elimination, the probability value is needed. For this, the new factors where there is no variable in it, but probability matters, this constructor is used. This constructor creates a new factor by creating a new variable. This variable is named "Blank" and it has only one value namely "True". Then with this variable, a new table is created with only one row, with the value "True" and with the probability given from the input parameter for the constructor. Then this new variable is put in a list and is set for the variable list attribute. Also, this new table and the probability are set for the attributes. The implementation of the three constructors can be seen below :

```
1  public Factor (ArrayList<Variable> variables, ArrayList<ProbRow> table)
2      {
3          this.variables = variables;
4          Variable first = variables.get(0);
5          ArrayList<Variable> parents = new ArrayList<>();
6          for(int i =1; i< variables.size(); i++)
7          {
8              parents.add(variables.get(i));
9          }
10         first.setParents(parents);
11         this.table = new Table(first, table);
12     }
13
14     public Factor (Table table)
15     {
16         this.table = table;
17         ArrayList <Variable>variables = new ArrayList<>();
```

```
18          variables.add(table.getVariable());
19          if(table.getVariable().hasParents())
20          {
21              variables.addAll(table.getParents());
22          }
23
24          this.variables = variables;
25      }
26
27      public Factor (double probability)
28      {
29          String name = "Blank";
30          ArrayList<String> possibleValues = new ArrayList<>();
31          possibleValues.add("True");
32          Variable newVariable = new Variable(name, possibleValues);
33          ArrayList<Variable> variables = new ArrayList<>();
34          variables.add(newVariable);
35          this.variables = variables;
36          ProbRow blankRow = new ProbRow(possibleValues, probability);
37          ArrayList<ProbRow> newTable = new ArrayList<>();
38          newTable.add(blankRow);
39          this.table = new Table(newVariable, newTable);
40          this.prob = probability;
41      }
```

2.2 Design and Implementation of Factor Operations

There are three factor operations. These are "Product", "Marginalization" and "Reduction". The design and implementation of these operations are given in the next sections.

2.2.1 Design and Implementation of Product Operation

The product operation is applied on two factors for this project. It finds a common variable. With the same values of the common variable in each row of both factors, the operation multiplies the two probabilities coming from the probability rows and creates a new row with the values from the first factor, the value of the common variable, and the values from the second factor. An illustration can be found in Figure 1.

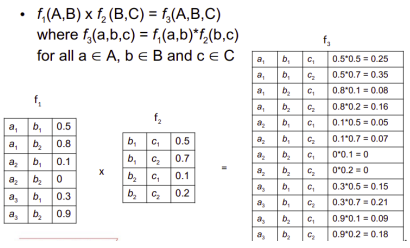

Figure 1: An example for product operation in factors.

If there is no common variable, then it makes a Cartesian product, which means for every probability row in the first factor, is merged with every probability row in the

second factor and the probabilities are multiplied. Again the result is a new factor. The occurrence of more than one common variable is out of scope for this project. For implementation, it will output an error about this situation.

If there is only one common variable, then the algorithm will make a row production based on the common variable. And then it will add these rows to a new table to make a new factor. In Algorithm 2 a pseudo-code for factor product operation can be found.

Algorithm 2 Algorithm for Product Operation

```
procedure PRODUCTION(Factor f1, Factor f2)
    Variable[] commons ← getCommons (f1,f2)
    ProbRow[] newTable ← ∅
    if size(commons) == 0 then
        for int i = 0, i in size (f1.table) do
            ProbRow pRowf1 ← f1.table[i]
            for int j = 0, j in size (f2.table) do
                ProbRow pRowf2 ← f2.table[j]
                ProbRow newRow ← makeProductRow (pRowf1, pRowf2, -1)
                newTable. add (newRow)
            end for
        end for
        Factor f3 ← Factor (getUnionVars(f1,f2), newTable)
    else if size(commons) == 1 then
        Variable common ← commons[0]
        int commonIndf1 ← indexOf (f1.variables, common)
        int commonIndf2 ← indexOf (f2.variables, common)
        for int i = 0, i in size (f1.table) do
            ProbRow pRowf1 ← f1.table[i]
            for int j = 0, j in size (f2.table) do
                ProbRow pRowf2 ← f2.table[j]
                if pRowf1.getValues()[commonIndf1]==
                    pRowf2.getValues()[commonIndf2] then
                    ProbRow newRow ←
                    makeProductRow (pRowf1, pRowf2, commonIndf2)
                    newTable.add(newRow)
                end if
            end for
        end for
        Factor f3 ← Factor (getUnionVars(f1,f2), newTable)
    elseprint ("Error : More than one common variables)
    end if
    return f3
end procedure
```

As it can be seen in the pseudo-code, there are some helper functions namely "makeProductRow", "getUnionVars" and, "getCommons". The function "getUnionVars" returns a union of variables from the input factors. The function "getCommons" returns the common variables from the input factors. The function "makeProductRow" is a bit complicated. It gets two probability rows and one integer value for the index as parameters. It creates an empty row. Then it adds every value in the given first row. And for the values in the second row, if the index of the value is not the same as the parameter, it adds that value to the row also. Finally, it multiplies the probabilities of the rows and

add them to the row and return that complete row.

Now it is more clear why -1 is used for the input index for the function "makeProductRow" when we have no common variables. Since the indices of the second row can be never -1, it will add every value in the second row. At first, for the implementation of this function, the used solution was to create two "makeProductRow" functions one gets an extra parameter for the index, but by using -1, it is not needed anymore, which makes the implementation less complex.

The implementation of product operation is done in the Factor class, as the other operations. It gets two factors as parameters and returns a new factor. Again **ArrayList** data type is used for this implementation. For the error, the "throw exception" data structure was used.

2.2.2 Design and Implementation of Marginalization Operation

Marginalization is an operation to "sum out" a variable from a factor. It can be said that, the rows with the different value of the specific variable but with the same value of the remaining variables are merged, their probability is summed together. In this way, a specific variable can be summed out from the factor. In the Figure 2, the variable B is summed out.

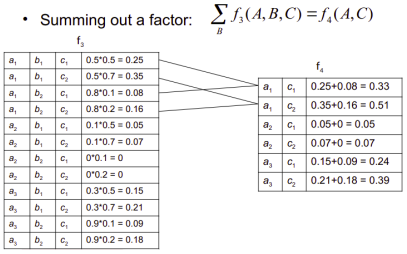

Figure 2: An example for marginalization operation in factors.

The algorithm for this operation is simple. It is based on a scanning operation. The algorithm takes the factor to be applied marginalization and a variable to be summed out as parameters. The index of the variable in the variable list of the factor is calculated. Then the first probability row is taken from the table of the factor and it is deleted. This first row is added to a row list to be used later. For each row in the table, if this row matches with our first row taken then also this matching row added to the row list. Here the row-matching means if the rows have same values for constant variables, and different values for the sum-out variable, they match. Finally summing-out operation is applied on this row list. This operation sums the rows where the sum-out value is different but other values are constant. Then it removes the sum-out value and return remaining values with total probability. So it basically creates a new row with summed probabilities and summed-out value. This new row is added to a new table.

This process is applied to every row until the initial table becomes empty. After this process, the variable as parameter is deleted from the variable list of the factor. And a final check is made. If the variable list is empty, which means the initial factor contains only one variable and the variable to be summed out is the same one, then for every row in the new table, the probabilities are summed and with that new summation value, a new factor is created based on this probability and returned. If the variable list is

not empty, then a new factor is created based on the variable list and the table and returned. The explanation of this algorithm can be understood better in Algorithm 3

Algorithm 3 Algorithm for Marginalization Operation

```
procedure MARGINALIZATION(Factor f1, Variable var)
    Variable[] allVariables ← f1.getVariables()
    ProbRow[] newTable ← f1.getTable().getTable()
    ProbRow[] newTable ← ∅
    int varIndex ← indexOf (allVariables, var)
    while table ≠ ∅ do
        ProbRow firstRow ← table[0]
        table.remove(0)
        ProbRow[] rowList ← ∅
        rowList.add(firstRow)
        for int i = 0, i in size (table) do
            if rowMatches(varIndex, firstRow, table[i]) then
                rowList.add(table[i])
            end if
        end for
        newTable.add(sumOutRow(varIndex, rowList))
    end while
    allVariables.remove(var)
    if allVariables = ∅ then
        doubleprobSum ← 0.0
        for each row in newTable do
            probSum ← probSum + row.getProb()
        end for
        Factor f2 ← Factor (probSum)
        return f2
    end if
    Factor f2 ← Factor (allVariables, newTable)
    return f2
end procedure
```

The pseudo-codes for the helper functions "rowMatches" and "sumOutRow" are given in Algorithm 4. It can give a better understanding.
For the implementation of this design, there are two remarks. The first one is about deleting the rows from the table which are matching rows. Java 11 gives a concurrency error, while making both getting and deleting operations in the same list in the same loop. To prevent this, a solution that uses a deletion list is used. After adding the matching rows to the rowList also they are added to the a list namely "rowsDeleted". After the for loop, the rows in the "rowsDeleted" are deleted from the initial table. The implementation can be found below.

```
1       if(rowMatches(varIndex, firstRow, table.get(i)))
2       {
3           rowList.add(table.get(i));
4           rowsDeleted.add(table.get(i));
5       }
6   }
7       for(ProbRow r : rowsDeleted){
```

Algorithm 4 Algorithms for Helper Functions

```
procedure rowMatches(int varIndex, ProbRow row1, ProbRow row2)
    String[] values1 ← row1.getValues()
    String[] values2 ← row2.getValues()
    for int i = 0, i in size (values1) do
        if (i ≠ varIndex && values1[i] ≠ values2[i]) |||| (i = varIndex && values1[i] =
values2[i]) then
            return false
        end if
    end for
    return true
end procedure
procedure sumOutRow(int varIndex, ProbRow[] rowList)
    String[] values ← ∅
    values ← rowList[0] values.remove(varIndex)
    double prob ← 0.0
    for int i = 0, i in size (rowList) do
        prob ← prob + rowList[i].getProb()
    end for
    return new ProbRow(values, prob)
end procedure
```

```
8           table.remove(r);
9       }
```

The second remark is about using "equals" function when comparing values. Since the values in the probability rows have **String** data type, using equals function is the most optimal one. Otherwise, for instance using = operator creates errors. An example implementation can be seen below.

```
1   if((i!= varIndex && !values1.get(i).equals(values2.get(i))) ||
2                   (i==varIndex && values1.get(i).equals( values2.get(i)) ))
3               {
4                   return false;
5               }
```

2.2.3 Design and Implementation of Reduction Operation

The reduction operation is used to reduce a variable from a factor. It is used especially to get rid of the observed values. An illustration can be found in Figure 3.

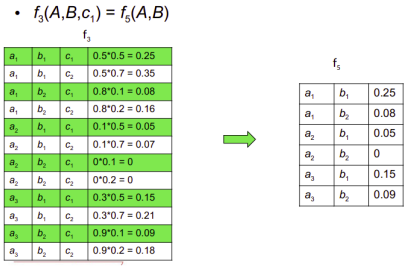

Figure 3: An example for marginalization operation in factors.

For design it takes a factor, a variable to be reduced and a value of the variable to be reduced. Its design principle is very simple. It searches for every row in the table of the factor. If the row contains the value, then its variable to be reduced is removed and the row is added to a new row list. At the end again a check is done about the size of the variable list of the table. As in the marginalization if the variable list contains no element then a probability calculation is done with the same logic and algorithm. If there is at least one element then a new factor is created with the remaining variables and the new table. The pseudo-code for this algorithm can be found in Algorithm 5.

Algorithm 5 Algorithm for Reduction Operation

```
procedure REDUCTION(Factor f1, Variable var, String value)
    Variable[] allVariables ← f1.getVariables()
    ProbRow[] newTable ← f1.getTable().getTable()
    ProbRow[] newTable ← ∅
    int varIndex ← indexOf (allVariables, var)
    for each row in table do
        if row.getValues().get(varIndex) = value then
            row.getValues().remove(varIndex)
            newTable.add(row)
        end if
    end for
    variables.remove(varIndex)
    if allVariables = ∅ then
        doubleprobSum ← 0.0
        for each row in newTable do
            probSum ← probSum + row.getProb()
        end for
        Factor f2 ← Factor (probSum)
        return f2
    end if
    Factor f2 ← Factor (allVariables, newTable)
    return f2
end procedure
```

For the implementation, there is no need for the third parameter a value. In given template there is a getter method for the observed value for variables, namely "getObservedValue()". Rather than getting the value from a parameter using this method is better and easier. The implementation is given below.

```
if(row.getValues().get(varIndex).equals( var.getObservedValue()))
            {
                row.getValues().remove(varIndex);
                newTable.add(row);

            }
```

2.3 Design and Implementation of Variable Elimination

Variable Elimination is a very useful method used for inference in Bayesian Networks. It makes the inference much more easy in the context of computation and complexity.

The algorithm for variable elimination is long but simple. The algorithm takes the network, query variable and the observed variables. The first step is essential. At this step, the product formula is prepared based on the network, query variable and the observed variables. Then the factors are created based on the product formula and added to a factor list. The next step is also an essential part. It is to fix an elimination order, but for this project it is out of scope. So for this project the elimination order is the list of the variables excluding the query variable. For each variable in this elimination order, the factors contain that variable are removed from the factor list and they are multiplied. Then that variable is summed-out by marginalization operation from the product. Finally this new factor is added to the list of the factors. This operation is done until there is no variable left in the elimination order. At the final step, the outcome factor is normalized. Here normalization means, the outcome factor is divided to its marginalized version. The result gives the query variable.

The algorithm can be understood better with an illustration. Pseudo-code for this algorithm can be found in Algorithm 6.
As it can be seen in the algorithm, there are two helper functions namely "multiply" and "normalize". The first one is used to multiply a list of factors. It takes first two factors from the list and then multiplies them with "product" operation and add this result to the list. Then it calls itself in a recursive way until there is only one factor is left, which is the result. The second function is used to normalize the outcome. It divides the input factor to its marginalized version. By this way, it normalizes the input factor.

For implementation there are some remarks. As the implementation of marginalization operation, here it is needed to remove elements from the list after searching is finished. Below you can see the illustration.

```
factors.removeAll(containsVarList);
```

For the implementation, there are four parameters needed which is different from the designed pseudo-code. The reason for this is in the given template, the variables and the probabilities are already extracted from the network. So the implementation about the parameters looks like this :

Algorithm 6 Algorithm for Variable Elimination

```
procedure VARIABLEELIMINATION(Network network, Variable query, Variable[] observed)
    Factor[] factors ← ∅
    for each table in network.probabilities do
        Factor newFactor ← new Factor(table)
        for each variable in observed do
            if newFactor.getVariables().contains(variable) then
                newFactor ← reduction(newFactor, variable)
            end if
        end for
        factors.add (newFactor)
    end for
    Variable[] varSumList ← empty
    network.variables.remove(query)
    varSumList ← network.variables
     for(Variable var :varSumList)
     ArrayList¡Factor¿ containsVarList = new ArrayList¡¿(); //find the factors contain Variable var and add to the list for(Factor f : factors) if(f.getVariables().contains(var)) containsVarList.add(f); factors.removeAll(containsVarList);
     Factor newFactor = Factor.marginalization(multiply(containsVarList), var); if(newFactor.getProbability()!=1.0 ) factors.add(newFactor);
    for each variable in varSumList do
        Factor[] containsVarList ← ∅
        for each factor in factors do
            if factor.getVariables().contains(var) then
                containsVarList.add(factor)
                factors.remove(factor)
            end if
        end for
        Factor newFactor = marginalization(multiply(containsVarList), variable)
        factors.add(newFactor)
    end for
    return normalize(factors[0])
end procedure
```

```
1 public static void variableElimination (Variable query, ArrayList<Variable> observed,
2 ArrayList<Variable> variables, ArrayList<Table> probabilities)
```

The most important remark about the implementation is about the normalize function. Unfortunately the function does not work as intended even though it is correctly implemented according to the design in a logical way. The assumption about the source of the problem is about pointing a reference rather than a value. The function deletes the probability rows of the given factor after marginalize it. In marginalize function it is normal, but normally it should change the reference for it. As a solution, even cloning is tried, but again it was not effective. So unfortunately, the program only prints the unnormalized factor. Although the normalization is an important step, and it is believed that the implementation is correct, it does not work in the intended way.

3 Testing

I have tested my design and implementation in a hybrid way and in multi-steps. Unfortunately there was no automated option for testing, i.e. there was no Bayesian network inference calculator. At first I thought I can do all the calculations by hand. But all of these reductions and marginalizations and productions would take too much time. But I thought I can use my own implementations of operations if I can prove them they work correctly by testing.

At first I have tested marginalization on couple samples. These samples were, summing out Alarm from the Factor f(Alarm, Burglary, Earthquake), summing out Burglary from the Factor f(Alarm, Burglary, Earthquake), summing out MaryCalls from the Factor f(MarryCalls, Alarm). They have been proven to work correctly. Then I have tested reduction and production in the same way.

After it is proven that they are working correctly, I have tested some cases with pen and paper, but for marginalization and other operations, I have used my program. I have tested 4 cases which are : Q: Alarm, E: NON, Q:JohnCalls E: NON, Q:JohnCalls, E: MarryCalls,True, Q:Alarm, E:MarryCalls.

The results seems matching what I have calculated with pen and paper. Even though there are lots of test cases can be made, my program seems to work correctly.

4 Discussion and Results

In this project, a practical way for inference in Bayesian Networks, the variable elimination is explained, designed and implemented. Based on the test, it can be said that the design and the implementation is correct.

Beyond this, also it can be said that variable elimination is an efficient way to calculate inference. In this project the hard way, computing by hand is also experienced and it can be easily understood that variable elimination algorithm is an essential algorithm.

5 Appendix

The whole implementation of this project can be found in the link :
gitlab/ege.sari/variable-elimination