

AI PRINCIPLES & TECHNIQUES

VARIABLE ELEMINATION ASSIGNMENT

BURAK BALCI
Radboud University

s1073728
21/12/2023

Contents

1	Introduction	2
1.1	Variable Elimination	2
1.2	Goals	2
2	Design	2
2.1	Factor	2
2.2	Factor Operations	3
2.2.1	Reduction Operation	3
2.2.2	Product Operation	4
2.2.3	Marginalization Operation	5
2.2.4	Normalization Operation	6
2.3	Heuristics	6
2.3.1	Basic Heuristic	6
2.3.2	Least Incoming Heuristic	6
2.3.3	Fewest Factors Heuristic	6
2.4	Variable Elimination	6
3	Implementation	7
3.1	Factor	7
3.2	Factor Operations	8
3.2.1	Reduction Operation	8
3.2.2	Product Operation	9
3.2.3	Marginalization Operation	10
3.2.4	Normalization Operation	11
3.3	Heuristics	12
3.4	Variable Elimination	12
4	Testing	15
4.1	Correctness	15
4.2	Heuristics Comparison	15
5	Conclusion	16

1 Introduction

In this section variable elimination and goals of this project is shortly explained.

1.1 Variable Elimination

Variable elimination is a simple and smart algorithm in probabilistic graphical models used for inference tasks, like Bayesian networks. Its primary goal is to compute marginal probabilities of variables in a network given observed evidence. The algorithm eliminates variables from a joint probability distribution by multiplying and summing factors representing conditional probabilities. By eliminating variables while considering observed evidence, it simplifies the computations and reduces the complexity of inference tasks in graphical models.

1.2 Goals

The main goal of this assignment to get an understanding of inference in Bayesian networks.

We can divide it to two parts.

- To design and implement multi-dimensional factors with factor operations such as reduction, product, and marginalization.
- To design and implement variable elimination algorithm.

2 Design

In this section, designs of factor, factor operations, heuristic functions and variable elimination are explained.

2.1 Factor

The main goal of the `Factor` class is to create factors and apply its related operations. This class is primarily responsible for holding information about a factor's associated variables and the probability data.

Constructor of this class is used to create factors, which is explained here, and related operations are used to do necessary calculations on factors, which will be explained in the next section.

For constructor design, the class has two constructors. One to create a factor for given `Table`, and one to create a factor for given variables and related probability rows.

First constructor gets a table as input, and transfers the variables in the table to the `variables` attribute of the factor. Also, it sets the `table` attribute of the factor to input table. It is used in variable elimination when setting the factor list, and iterating through that factor list. Since given probabilities is a list of tables, this constructor is really useful for that task.

Second constructor gets a variable list and a probability row list as input, sets `variables` attributes to input variables list. It uses that variables list and the probability row list to create a table. First variable of the variables list will obviously be the main variable of this factor. Then rest of the variables in that list will be the parents of this variable. Now that the main variable and its parents are set, constructor creates the table for the main variable and its probability rows. Then sets its `table` attribute to the table created. This is the constructor that each factor operation use.

2.2 Factor Operations

In this section, design of factor operations reduction, product, and marginalization are explained with their respective helper functions. Also normalization is explained here since it's a part of `Factor` class.


2.2.1 Reduction Operation

Reduction operation is used for reducing the given variable from a factor. In variable elimination, reduction is used to get rid of observed variables.

If a value of one of the variables in a factor is known, then we can eliminate the rows that contain the other value of that known variable from that factor. An illustration given below.

$$\bullet f_3(A, B, c_1) = f_5(A, B)$$

f_3			
a_1	b_1	c_1	$0.5 \cdot 0.5 = 0.25$
a_1	b_1	c_2	$0.5 \cdot 0.7 = 0.35$
a_1	b_2	c_1	$0.8 \cdot 0.1 = 0.08$
a_1	b_2	c_2	$0.8 \cdot 0.2 = 0.16$
a_2	b_1	c_1	$0.1 \cdot 0.5 = 0.05$
a_2	b_1	c_2	$0.1 \cdot 0.7 = 0.07$
a_2	b_2	c_1	$0 \cdot 0.1 = 0$
a_2	b_2	c_2	$0 \cdot 0.2 = 0$
a_3	b_1	c_1	$0.3 \cdot 0.5 = 0.15$
a_3	b_1	c_2	$0.3 \cdot 0.7 = 0.21$
a_3	b_2	c_1	$0.9 \cdot 0.1 = 0.09$
a_3	b_2	c_2	$0.9 \cdot 0.2 = 0.18$



f_5		
a_1	b_1	0.25
a_1	b_2	0.08
a_2	b_1	0.05
a_2	b_2	0
a_3	b_1	0.15
a_3	b_2	0.09

In the design of the function, it takes a factor and a variable to reduce from given factor. It searches every row in the table of the factor. If the row that it's searching contains the given variable, it checks if the value of that variable in the row is equal to observed value of the input variable. If it is, it removes the variable from the row and adds the row to a row list that will be used to create the new factor. After checking every row, by using the row list that contains every updated row, creates a new factor.

Pseudo code of reduction is below.

```

1 function reduction(factor, variable):
2     variableList = factor.getVariables
3     rowList = factor.getRows
4     varIndex = variableList.indexOf(variable)
5     for row in rowList:
6         if row.getValues.get(varIndex) equals variable.getValue:
7             row.remove(varIndex)
8             resultTable.add(row)
9     variableList.remove(varIndex)
10    return Factor(variableList, resultTable)

```

2.2.2 Product Operation

Product operation is used for multiplying factors on their 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 union of two factors. An illustration given below.

- $f_1(A,B) \times f_2(B,C) = f_3(A,B,C)$
 where $f_3(a,b,c) = f_1(a,b) \times f_2(b,c)$
 for all $a \in A, b \in B$ and $c \in C$

f_1				f_2				f_3			
a_1	b_1	0.5	x	b_1	c_1	0.5	=	a_1	b_1	c_1	$0.5 \times 0.5 = 0.25$
a_1	b_2	0.8		b_1	c_2	0.7		a_1	b_1	c_2	$0.5 \times 0.7 = 0.35$
a_2	b_1	0.1		b_2	c_1	0.1		a_1	b_2	c_1	$0.8 \times 0.1 = 0.08$
a_2	b_2	0		b_2	c_2	0.2		a_1	b_2	c_2	$0.8 \times 0.2 = 0.16$
a_3	b_1	0.3						a_2	b_1	c_1	$0.1 \times 0.5 = 0.05$
a_3	b_2	0.9						a_2	b_1	c_2	$0.1 \times 0.7 = 0.07$
								a_2	b_2	c_1	$0 \times 0.1 = 0$
								a_2	b_2	c_2	$0 \times 0.2 = 0$
								a_3	b_1	c_1	$0.3 \times 0.5 = 0.15$
								a_3	b_1	c_2	$0.3 \times 0.7 = 0.21$
								a_3	b_2	c_1	$0.9 \times 0.1 = 0.09$
								a_3	b_2	c_2	$0.9 \times 0.2 = 0.18$

In the design of the function, it takes two factors and a variable that's common for both factors which was found before calling the function. It creates a variable list with the union of both factors variables. Then creates a row list by multiplying rows that shares the same value for common variable from two factors and adding those created rows to the list. Uses the row list and the union of variables to create a new factor.

Pseudo code of product is below.

```

1 def product(f1, f2, commonVariable):
2     unionVariableList = union(f1, f2)
3     for rowOne in f1:
4         for rowTwo in f2:
5             if rowOne.get(commonVarIndex1) equals rowTwo.get(
6                 commonVarIndex2):
7                 resultValues = rowOne.values
8                 for val in rowTwo.values:
9                     if val.index not equals commonVarIndex2:
10                        resultValues.add(val)
11                resultTable.add(row(resultValues, (rowOne.prob * rowTwo.prob)))
12            return Factor(unionVariableList, resultTable)

```

2.2.3 Marginalization Operation

Marginalization operation is used for adding row(s) in a given factor where each value in each row is equal to each other except the variable that factor is marginalized on. Basically it's an operation to sum out a variable from a factor. An illustration given below.

• Summing out a factor: $\sum_B f_3(A, B, C) = f_4(A, C)$

f_3					f_4		
a_1	b_1	c_1	$0.5 \cdot 0.5 = 0.25$		a_1	c_1	$0.25 + 0.08 = 0.33$
a_1	b_1	c_2	$0.5 \cdot 0.7 = 0.35$		a_1	c_2	$0.35 + 0.16 = 0.51$
a_1	b_2	c_1	$0.8 \cdot 0.1 = 0.08$		a_2	c_1	$0.05 + 0 = 0.05$
a_1	b_2	c_2	$0.8 \cdot 0.2 = 0.16$		a_2	c_2	$0.07 + 0 = 0.07$
a_2	b_1	c_1	$0.1 \cdot 0.5 = 0.05$		a_3	c_1	$0.15 + 0.09 = 0.24$
a_2	b_1	c_2	$0.1 \cdot 0.7 = 0.07$		a_3	c_2	$0.21 + 0.18 = 0.39$
a_2	b_2	c_1	$0 \cdot 0.1 = 0$				
a_2	b_2	c_2	$0 \cdot 0.2 = 0$				
a_3	b_1	c_1	$0.3 \cdot 0.5 = 0.15$				
a_3	b_1	c_2	$0.3 \cdot 0.7 = 0.21$				
a_3	b_2	c_1	$0.9 \cdot 0.1 = 0.09$				
a_3	b_2	c_2	$0.9 \cdot 0.2 = 0.18$				

In the design of the function, it takes a factor and a variable to marginalize on. It takes a row from factor and removes it from the factor list, iterates through other rows and checks if the row that function is on is summable with the first row that's taken. To check if they are summable, it checks two rows each variable except the input variable is equal to each other. If it is, function add the summable row to a new list that will be marginalized later. Next, function sums each row in the list of rows to be marginalized with the first row it took and removes each row that's summed up with the first row from factors row list. Then it adds the marginalized row to a result rows list. If it can't find any rows to add, it simply removes the given variable from the first row and adds it to the result row list. After the main row list of the factor is completely empty, it uses the result row list and variables of the input factor by removing the input variable, creates a new factor.

Pseudo code of marginalization is below.

```

1 function marginalization(factor, variable):
2     while (!factor.rows.isEmpty):
3         firstRow = factor.rows.remove(0)
4         for row in factor.rows:
5             if summable(firstRow, row):
6                 toMarginalize.add(row)
7         if (!toMarginalize.isEmpty):
8             for row in toMarginalize:
9                 firstRow.sum(row)
10            resultRows.add(firstRow)
11        else:
12            firstRow.remove(variable)
13            resultRows.add(firstRow)
14    factor.variables.remove(variable)
15    return Factor(factor.variables, resultRows)

```

2.2.4 Normalization Operation

Normalization operation is used for normalizing the input factor. Its main purpose is to after variable elimination, normalize the query variable. It takes a factor, and finds its total probability. Then for each row in factor, it updates the probability to $\text{prob}/\text{totalProb}$. That way when we add up the probability of each row after normalization, we get 1.

Pseudo code of normalization is below.

```
1 function normalization(factor):  
2     totalProb = factor.sumAllProbs  
3     for row in factor:  
4         row.prob = prob / totalProb  
5     return factor
```

2.3 Heuristics

Heuristic class has four attributes. These attributes are a heuristic string to identify which heuristic will be used, a variable list to apply heuristic on, a query variable to remove query after ordering, and a factor list that's only used for fewest factors heuristic. It has a constructor, a function to apply the heuristic and a function to get variables which will be used after ordering.

2.3.1 Basic Heuristic

Basic heuristic just simply shuffles the given variable list randomly, and removes the query from the list.

2.3.2 Least Incoming Heuristic

Least incoming heuristic priorities variables with least incoming edges, so it sorts the given variable list by their incoming edges count, and removes the query.

2.3.3 Fewest Factors Heuristic

Fewest factors heuristic priorities variables that contained in fewer factors, so it sorts the given variable list by their number of factors that they are contained in, and removes the query.

2.4 Variable Elimination

Variable elimination is the main algorithm that's used in this project. It has five attributes. A query variable, a variable list for all variables, a table list of tables of all variables, a variable list for observed variables, and a string for heuristics. Constructor of this class is used for creating a variable elimination object with all attributes mentioned, and then applying variable elimination on this object.

There are some steps before the algorithm to set up some of the lists, heuristics, also a logger but that will be explained in the implementation section.

First thing to do is apply the heuristic function and reduce the variable and table list. Next, reduces the variable list to only necessary variables. Basically, any variable that is not related to query, or any of the observed variable is eliminated from the variables list. After that, it sorts the updated variable list with the given heuristic function. It also updates the table list so that only the tables that belongs to query and remaining variables left. After setting up everything, algorithm starts.

First, algorithm identifies factors and reduces the observed variables from those factors. By iterating through the tables list, it creates a factor on each iteration for the table, and reduces each variable from that factor if factor contains the observed variable. After reduction, adds the factor to the factor list. Pseudo code for identifying factors and reducing observed variables is below.

```

1 function identifyReduce():
2     for table in tables:
3         factor = Factor(table)
4         for var in observedVariables:
5             if factor.contains(var):
6                 factor.reduce(var)
7         factors.add(factor)

```

Then, for each variable Z in elimination order, which is the variables list that updated when setting up the function by reducing variables list and sorting it by using heuristic, it adds each factor that contains the variable Z to a new factors list. If there are more than one factors that contain variable Z, product method is applied on those factors. If there is only one factor that contains variable Z, product operation is not needed. After removing every factor that contains variable Z from the main factor list, it applies the marginalization method either on production of factors, or the only factor that contains Z. Lastly it adds the result factor to the main factor list. By iterating through every observed variable and applying the explained methods, it successfully updates the factors step by step.

Pseudo code for the algorithm is below.

```

1 function algorithm():
2     for Z in variables:
3         for f in factors:
4             if f.contains(Z):
5                 factorsWithZ.add(f)
6         if factorsWithZ.size() > 1:
7             factors.removeAll(factorsWithZ)
8             factorProduct = productAll(factorsWithZ, Z)
9             factorMarg = marginalization(factorProduct, Z)
10        else:
11            factors.remove(factorsWithZ)
12            factorMarg = marginalization(factorsWithZ, Z)
13        factors.add(factorMarg)

```

To finish it up, it normalizes the final factor, which is explained in the implementation of normalization, and variable elimination is done.

3 Implementation

In this section, implementation of factors, factor operations, heuristic functions and variable elimination are explained.

3.1 Factor

Implementation of attributes and constructors of Factor is below.

```

1 public class Factor {
2     private final Table table;
3     private final ArrayList<Variable> variables;
4
5     public Factor(Table table) {
6         this.table = table;
7         ArrayList<Variable> tempVars = new ArrayList<>();

```

```

8      tempVars.add(table.getVariable());
9      if (table.getVariable().hasParents()) {
10         tempVars.addAll(table.getParents());
11     }
12     this.variables = tempVars;
13 }
14
15 public Factor(ArrayList<Variable> variables , ArrayList<ProbRow> rows)
16 {
17     this.variables = variables;
18     Variable firstVar = variables.get(0);
19     ArrayList<Variable> parents = new ArrayList<>();
20     for (int i = 1; i < variables.size(); i++) {
21         parents.add(variables.get(i));
22     }
23     firstVar.setParents(parents);
24     this.table = new Table(firstVar , rows);
25 }
26
27 public Table getTable() {
28     return table;
29 }
30
31 public ArrayList<Variable> getVariables() {
32     return variables;
33 }

```

3.2 Factor Operations

3.2.1 Reduction Operation

In the implementation of reduction, unlike the design, the variables are not removed from rows when creating new factor. That's because if input factor has only one variable, removing that variable will cause errors. For other cases, marginalization operation will remove that variable anyways so it doesn't cause any problems to not remove a variable from the factor in this operation.

Possible solution for that would've been checking the factor size and its variables to see if removing variable would become problematic at the beginning of the function, just like in marginalization, but I decided with the implementation I explained above.

Implementation of reduction is below.

```

1 public static Factor reduction(Factor factor , Variable var) {
2     ArrayList<Variable> vars = factor.getVariables();
3     ArrayList<ProbRow> table = factor.getTable().getTable();
4     ArrayList<ProbRow> resultTable = new ArrayList<>();
5     Factor resultFactor;
6     int varIndex = vars.indexOf(var);
7     for (ProbRow row : table) {
8         if (row.getValues().get(varIndex).equals(var.getObservedValue()))
9         {
10             resultTable.add(row);
11         }
12     }
13     resultFactor = new Factor(vars , resultTable);
14     return resultFactor;
15 }

```


3.2.2 Product Operation

Implementation of product is pretty much the same. The difference from the design is I used a helper function to calculate the product row list of two factors, but the logic behind it is the same, explained in the design section.

Additionally, I added a function that calculates the product of multiple functions. For given factor list, it uses product operation for the first two factors of that list, removes them from the list and adds the product factor to the list. It does that until factor list has the size 1, hence finds the product of multiple factors. That's needed since when finding factors that share a variable, if the list of factors has more than two factors, productAll function will be used. Implementation is below.

```
1 public static Factor productAll(ArrayList<Factor> factors , Variable var) {
2     Factor resultFactor;
3     while (factors.size() != 1) {
4         Factor factorOne = factors.get(0);
5         Factor factorTwo = factors.get(1);
6         Factor tempFactor = product(factorOne , factorTwo , var);
7         factors.remove(factorOne);
8         factors.remove(factorTwo);
9         factors.add(tempFactor);
10    }
11    resultFactor = factors.get(0);
12    return resultFactor;
13 }
```

Implementation of product operation is below.

```
1 public static Factor product(Factor factorOne , Factor factorTwo , Variable
2     var) {
3     Factor resultFactor;
4     ArrayList<Variable> tempVars = getTotalVariables(factorOne , factorTwo)
5     ;
6     ArrayList<ProbRow> resultTable ;
7     resultTable = productOneCommonVariable(factorOne , factorTwo , var);
8     resultFactor = new Factor(tempVars , resultTable);
9     return resultFactor;
10 }
```

Implementation for product on common variable is below.

```
1 public static ArrayList<ProbRow> productOneCommonVariable(Factor factorOne
2     , Factor factorTwo , Variable commonVar) {
3     ArrayList<ProbRow> resultTable = new ArrayList<>();
4     int commonIndexFactorOne = factorOne.getVariables().indexOf(commonVar)
5     ;
6     int commonIndexFactorTwo = factorTwo.getVariables().indexOf(commonVar)
7     ;
8     for (int i = 0; i < factorOne.getTable().size(); i++) {
9         ProbRow rowOne = factorOne.getTable().get(i);
10        for (int j = 0; j < factorTwo.getTable().size(); j++) {
11            ProbRow rowTwo = factorTwo.getTable().get(j);
12            if (rowOne.getValues().get(commonIndexFactorOne).equals(rowTwo
13                .getValues().get(commonIndexFactorTwo))) {
14                ProbRow rowProduct;
15                ArrayList<String> resultValues = new ArrayList<>(rowOne.
16                    getValues());
17                for (int k = 0; k < rowTwo.getValues().size(); k++) {
18                    if (k != commonIndexFactorTwo) {
19                        resultValues.add(rowTwo.getValues().get(k));
20                    }
21                }
22                rowProduct = new ProbRow(resultValues , (rowOne.getProb() *
23                    rowTwo.getProb()));
24            }
25        }
26    }
```

```

18         resultTable.add(rowProduct);
19     }
20 }
21 }
22 return resultTable;
23 }

```

Implementation for checking if two factors are summable is below.

```

1 public static Factor productAll(ArrayList<Factor> factors, Variable var) {
2     Factor resultFactor;
3     while (factors.size() != 1) {
4         Factor factorOne = factors.get(0);
5         Factor factorTwo = factors.get(1);
6         Factor tempFactor = product(factorOne, factorTwo, var);
7         factors.remove(factorOne);
8         factors.remove(factorTwo);
9         factors.add(tempFactor);
10    }
11    resultFactor = factors.get(0);
12    return resultFactor;
13 }

```

3.2.3 Marginalization Operation

In the implementation of marginalization, there is an important difference from the design.

It checks if given factor is marginalizable at the beginning of the function. In cases where input of this operation is a factor with one variable, not checking if operation is possible would cause problems. Also, although the logic behind is the same, implementation of this operation is a little bit different. There are a lot of removing and adding to variable and row lists, so I implemented a lot of temporary lists.

Also, there is a helper function for checking if two rows are summable. It checks if two input rows are equal to each other on each value except the given index. Implementation of that function is below.

```

1 public static Boolean summable(ProbRow rowOne, ProbRow rowTwo, int index)
2 {
3     ArrayList<String> valuesOne = rowOne.getValues();
4     ArrayList<String> valuesTwo = rowTwo.getValues();
5     for (int i = 0; i < valuesOne.size(); i++) {
6         if (!valuesOne.get(i).equals(valuesTwo.get(i)) && i != index) {
7             return false;
8         }
9     }
10    return true;
11 }

```

Implementation of marginalization is below.

```

1 public static Factor marginalization(Factor factor, Variable var) {
2     if (factor.getVariables().size() == 1 && factor.getVariables().get(0) == var) {
3         return factor;
4     }
5     Factor resultFactor;
6     ArrayList<ProbRow> resultTable = new ArrayList<>();
7     ArrayList<Variable> vars = new ArrayList<>(factor.getVariables());
8     ArrayList<ProbRow> tempTable = factor.getTable().getTable();
9     int varIndex = factor.getVariables().indexOf(var);
10    while (!tempTable.isEmpty()) {
11        ProbRow tempRowOne = tempTable.remove(0);

```

```

12         ArrayList<ProbRow> toMargRows = new ArrayList<>();
13         for (ProbRow row : tempTable) {
14             if (summable(tempRowOne, row, varIndex)) {
15                 toMargRows.add(row);
16             }
17         }
18         if (!toMargRows.isEmpty()) {
19             double sumProb = tempRowOne.getProb();
20             for (ProbRow row : toMargRows) {
21                 sumProb += row.getProb();
22                 tempTable.remove(row);
23             }
24             ArrayList<String> marginalizedValues = new ArrayList<>(
25                 tempRowOne.getValues());
26             marginalizedValues.remove(varIndex);
27             ProbRow marginalizedRow = new ProbRow(marginalizedValues,
28                 sumProb);
29             resultTable.add(marginalizedRow);
30         }
31         else {
32             ArrayList<String> tempValues = new ArrayList<>(tempRowOne.
33                 getValues());
34             tempValues.remove(varIndex);
35             ProbRow tempRow = new ProbRow(tempValues, tempRowOne.
36                 getProb());
37             resultTable.add(tempRow);
38         }
39     }
40     vars.remove(var);
41     resultFactor = new Factor(vars, resultTable);
42     return resultFactor;
43 }

```

3.2.4 Normalization Operation

Implementation of the normalization operation is the same as the design. For finding the factor to normalize, variable elimination algorithm finds the last factor that's equal to query from factors list. Then calls `normalize`. Implementation for finding variable to normalize, followed by normalization operation is given below.

```

1  for (Factor f : factors) {
2      if (f.getVariables().contains(query)) {
3          finalFactor = f;
4      }
5  }
6  finalFactor = Factor.normalize(finalFactor);

```

```

1  public static Factor normalize(Factor factor) {
2      Factor resultFactor;
3      ArrayList<ProbRow> finalTable = new ArrayList<>();
4      double finalProb = 0.0;
5      for (ProbRow row : factor.getTable().getTable()) {
6          finalProb += row.getProb();
7      }
8      for (ProbRow row : factor.getTable().getTable()) {
9          ProbRow tempRow = new ProbRow(row.getValues(), (row.getProb() /
10              finalProb));
11          finalTable.add(tempRow);
12      }
13      resultFactor = new Factor(factor.getVariables(), finalTable);
14      return resultFactor;
15 }

```

3.3 Heuristics

Implementation of the Heuristics class is given below.

```
1 public class Heuristic {
2     private final String heuristic;
3     private ArrayList<Variable> variables;
4     private final Variable query;
5     private final ArrayList<Factor> factors;
6
7     public Heuristic(String heuristic, ArrayList<Variable> variables,
8         Variable query, ArrayList<Factor> factors) {
9         this.heuristic = heuristic;
10        this.variables = variables;
11        this.query = query;
12        this.factors = factors;
13    }
14
15    public void applyHeuristic() {
16        ArrayList<Variable> orderedVars;
17        if (heuristic.equals("least-incoming")) {
18            orderedVars = new ArrayList<>(variables);
19            orderedVars.sort(Comparator.comparingInt(Variable::
20                getNrOfParents));
21        }
22        else if (heuristic.equals("fewest-factors")) {
23            orderedVars = new ArrayList<>(variables);
24            orderedVars.sort(Comparator.comparingInt(v -> {
25                int factorCount = 0;
26                for (Factor factor : factors) {
27                    if (factor.getVariables().contains(v)) {
28                        factorCount++;
29                    }
30                }
31                return factorCount;
32            })));
33        }
34        else {
35            orderedVars = new ArrayList<>(variables);
36            shuffle(orderedVars);
37        }
38        orderedVars.remove(query);
39        variables = orderedVars;
40    }
41
42    public ArrayList<Variable> getVariables() {
43        return variables;
44    }
45 }
```

3.4 Variable Elimination

The implementation of variable elimination was not complicated at all since I followed the design explained in the design section of variable elimination.

First, I implemented some helper functions to set up the variable and table lists like mentioned in design. Implementation of necessaryVariables and necessaryTables are below.

```
1 private ArrayList<Variable> necessaryVariables(ArrayList<Variable>
2     variables, ArrayList<Variable> observed) {
3     ArrayList<Variable> relevant = new ArrayList<>();
4     ArrayList<Variable> result = new ArrayList<>();
5     ArrayList<Variable> queue = new ArrayList<>();
```

```

5     queue.add(query);
6     queue.addAll(observed);
7     while (!queue.isEmpty()) {
8         Variable tempVar = queue.remove(0);
9         relevant.add(tempVar);
10        if (tempVar.hasParents()) {
11            ArrayList<Variable> parents = tempVar.getParents();
12            for (Variable var : parents) {
13                if (!relevant.contains(var)) {
14                    queue.add(var);
15                }
16            }
17        }
18    }
19    for (Variable var : variables) {
20        if (relevant.contains(var)) {
21            result.add(var);
22        }
23    }
24    return result;
25 }
26
27 private ArrayList<Table> necessaryTables(ArrayList<Variable> variables ,
28     ArrayList<Table> probabilities) {
29     ArrayList<Table> result = new ArrayList<>();
30     for (Table table : probabilities) {
31         if (table.getVariable() == query || variables.contains(table.
32             getVariable())) {
33             if (!result.contains(table)) {
34                 result.add(table);
35             }
36         }
37     }
38     return result;
39 }

```

Next thing I added was the Logger to keep track of each step the algorithm takes. After implementing it, calling `logger.info()` with the input that is needed for each step was enough to check if each step is working like it should. Implementation of Logger is below.

```

1  Logger logger = Logger.getLogger("LogVE");
2  try {
3      FileHandler fileHandler = new FileHandler("LogVE.log", false);
4      fileHandler.close();
5      fileHandler = new FileHandler("LogVE.log", true);
6      fileHandler.setFormatter(new SimpleFormatter());
7      logger.addHandler(fileHandler);
8      logger.info("Starting Variable Elimination...\n");
9  } catch (Exception e) {
10     logger.info("Error occurred.");
11     e.printStackTrace();
12 }

```

I also added a timer at the beginning and the end of the algorithm to experiment with different cases and heuristics. I used `System.nanoTime()` for this.

Next I completed the algorithm by using the functions that I implemented for setting up everything, and following my design.

Full implementation of variable elimination is below. Note that log or time calls, and comments are not included here. However, they can be found in the actual code.

```

1 public void variableElimination() {
2     ArrayList<Factor> factorsForHeuristic = new ArrayList<>();
3     for (Table table : probabilities) {
4         factorsForHeuristic.add(new Factor(table));
5     }
6     variables.remove(query);
7     ArrayList<Variable> reducedVariables = necessaryVariables(variables,
8         observed);
9     Heuristic h = new Heuristic(heuristic, reducedVariables, query,
10        factorsForHeuristic);
11     h.applyHeuristic();
12     ArrayList<Variable> eliminationOrder = h.getVariables();
13     Factor finalFactor = null;
14     ArrayList<Table> reducedProbabilities = necessaryTables(
15        eliminationOrder, probabilities);
16     ArrayList<Factor> factors = new ArrayList<>();
17     for (Table table : reducedProbabilities) {
18         Factor tempFactor = new Factor(table);
19         for (Variable obs : observed) {
20             if (tempFactor.getVariables().contains(obs)) {
21                 tempFactor = Factor.reduction(tempFactor, obs);
22             }
23         }
24         factors.add(tempFactor);
25     }
26     for (Variable Z : eliminationOrder) {
27         ArrayList<Factor> factorsContainZ = new ArrayList<>();
28         for (Factor f : factors) {
29             if (f.getVariables().contains(Z)) {
30                 factorsContainZ.add(f);
31             }
32         }
33         Factor factorZ;
34         if (factorsContainZ.size() > 1) {
35             for (Factor f : factorsContainZ) {
36                 factors.remove(f);
37             }
38             factorZ = Factor.productAll(factorsContainZ, Z);
39             factorZ = Factor.marginalization(factorZ, Z);
40         }
41         else {
42             Factor f = factorsContainZ.get(0);
43             factors.remove(f);
44             factorZ = Factor.marginalization(f, Z);
45         }
46         for (Factor f : factors) {
47             if (f.getVariables() == factorZ.getVariables()) {
48                 factors.remove(f);
49             }
50         }
51         factors.add(factorZ);
52     }
53     for (Factor f : factors) {
54         if (f.getVariables().contains(query)) {
55             finalFactor = f;
56         }
57     }
58     assert finalFactor != null;
59     finalFactor = Factor.normalize(finalFactor);
60 }

```

4 Testing

This section is divided to two parts. Correctness of the variable elimination algorithm, where we test if the algorithm is doing what we want, and comparison between heuristics.

4.1 Correctness

Unfortunately, I didn't have time to calculate any case by hand and compare the results for the algorithm. But, I saved every step to a log file to keep track of the steps that taken by the algorithm. Each calculation and step was correct.

Algorithm creates a new `Variable` list that consists only necessary variables for the query. Then, applies the given heuristic for setting the elimination order. Then reduces factors of those variables and query if it can, and adds them to a `Factor` list. By using the elimination order, it eliminates each variable by first using `product` method if it can, and marginalizes. Finally, it normalizes the query.

We can check the log file to see each step mentioned with calculation of probabilities.

4.2 Heuristics Comparison

Design and implementation of the heuristic functions used for this assignment are explained in **Section 2.3**. Shortly, basic heuristic just shuffles the elimination list. Least incoming heuristic priorities variables with least incoming edges first, and fewest factors heuristic priorities variables contained in fewest factors first.

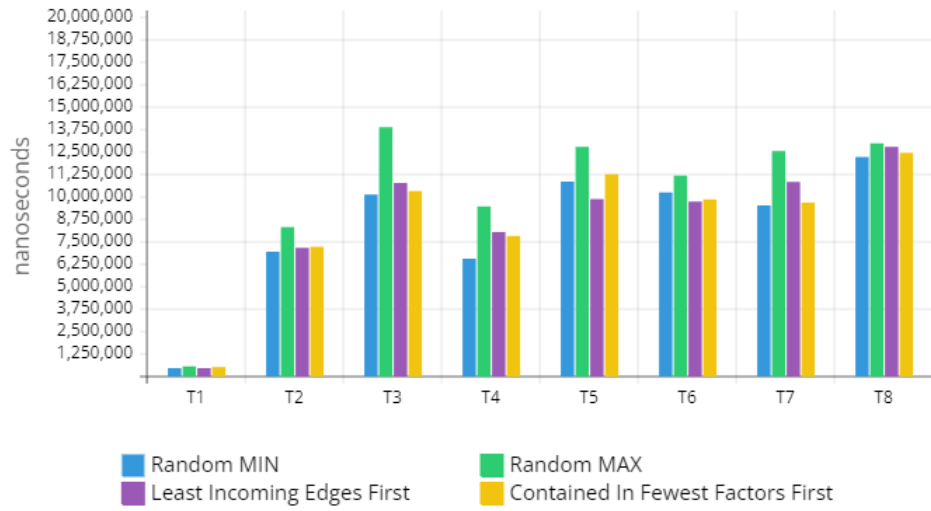
To compare them, `nanoTime` is used. We get the time when we start the variable elimination, and also when we are done. Difference between them gives us the time it took for variable elimination in nanoseconds.

Table below shows the tests ran for this comparison.

Test Numbers	Tests
T1	$P(\text{Burglary})$
T2	$P(\text{Alarm})$
T3	$P(\text{JohnCalls})$
T4	$P(\text{Alarm} \text{Burglary} = \text{True})$
T5	$P(\text{Alarm} \text{MaryCalls} = \text{False})$
T6	$P(\text{MaryCalls} \text{Alarm} = \text{False})$
T7	$P(\text{MaryCalls} \text{Burglary} = \text{True})$
T8	$P(\text{JohnCalls} \text{Burglary} = \text{True}, \text{Earthquake} = \text{False}, \text{MarryCalls} = \text{True})$

To compare the heuristics, I ran each case 5 times with each heuristic. I took average of the nanoseconds for both least incoming heuristic and fewest factors heuristic. For random ordering, I took minimum time it took and maximum time it took in my tests. The reason I did that is since it's completely random, averaging the results didn't make much sense for me.

Graph below shows the comparison between heuristics.



5 Conclusion

In this project, design, implementation and logic of a practical way for inference in Bayesian Networks called variable elimination along with other necessary operations to apply variable elimination is explained.