# Bayesian Networks - SPML assignment 3

**David Leeftink (4496612) & Mantas Makelis (1007870)**

Radboud University, Nijmegen
Search, Planning & Machine Learning: BKI212A
Dr. P.A. Kamsteeg, Dr. J.H.P. Kwisthout
January 11, 2019

## 1 Specification

This implementation of the Variable Elimination algorithm is a version of the original algorithm as developed by American computer scientist Rina Dechter.[1] The implementation is written in Java and contains additional features such as non-binary variable processing. Bayesian inference by Variable Elimination has an exponential time complexity, but this can be further reduced by choosing the right order of elimination.[2] The goal of this implementation is to create the Variable Elimination algorithm as efficient as possible, by using a proper order of elimination and efficient data structures to represent variables and factors. The research question of this paper is: "To what extend can Bayesian networks be further optimized by improving the order of variables?". This paper will first describe a specification of inference in Bayesian networks and the design of the Variable Elimination algorithm. Next will be given an overview of the implementation of the algorithm, followed by its results. In conclusion, the performance of the implementation will be evaluated in order to answer the research question.

## 2 Design

The design of this implementation is based on the original Variable Elimination algorithm as developed by Rina Dechter.[1] More specifically, it is based on the pseudo code as described by David Poole and Alan MacWorth.[3]

```
1  Procedure VE_BN(Vs,Ps,O,Q)
2      Inputs
3          Vs: set of variables
4          Ps: set of factors representing the conditional
               probabilities
5          O: set of observations of values on some of the variables
6          Q: a query variable
7      Output
8          posterior distribution on Q
9      Local
10         Fs: a set of factors
11         Fs = Ps
12     for each X in Vs - {Q} using some elimination ordering do
```

```
13          if (X is observed) then
14              for each F in Fs that involves X do
15                  set X in F to its observed value in O
16                  project F onto remaining variables
17          else
18              Rs = {F in Fs: F  involves X}
19              let T be the product of the factors in Rs
20              N = X T
21              Fs = Fs \ Rs union with {N}
22      let T be the product of the factors in Fs
23      N = Sum over Q of each T
24      return T/N
```

**Listing 1:** Pseudo code of the Variable Elimination algorithm, as described by David Poole and Alan MacWorth

The pseudo code describes how the algorithm handles Bayesian inference and makes an estimation of the probability distribution of variables. The first steps include creating the factors and eliminating the observed variables in the factors, as is done in lines 12 to 16. This is done by using a certain order of elimination, that is of big influence of the complexity of the program. Then the factors can be merged by multiplying and marginalizing them until only one factor remains, as happens in line 18 to 21. The last step is to normalize the results. This is done by dividing the outcomes of the final factor by the sum of values of the final factor, and can be seen in line 22 to 24.

There are different variations of the design possible, but the variations that are most notable are the variations in the order of elimination. Two common heuristics include minimizing the number of factors a variable appears in, or minimizing the lowest number of outgoing arcs of a variable.[2]

# 3 Implementation

## 3.1 Representation

The implementation is build on provided code for representing the variables and reading in problems by the Student Assistants. In line with this representation, separate classes are added for the algorithm and the factor. In the Factor class, each factor is represented by the variables it contains and its probability distribution. Each factor can be merged with another factor, meaning the factors are multiplied and then summed out, resulting in a new factor. Factors can also take a list of factors, which will multiply and sum all factors in the given list.

A new factor is created by combining a given variable and its probability table. This is done in line 2 to 4 in listing 2. The listing forms a part of the code, a complete version including all the original comments can be found in the appendix. After assigning the variable and its probabilities, the factor will reduce all columns that are no longer relevant based on

information from observed variables. This is implemented in two steps, namely gathering all indices and values of the items that need to be removed and the removing itself. The first one is represented between lines 6 and 25, and is the most elaborate task. It is done by looping over all the variables and collecting their values and indices if they are observed. The latter is done between lines 26 and 34 and is carried out by inserting the previously gained information.

```
1  public Factor(Variable variable, Table probTable) {
2          variables = new ArrayList<>();
3          variables.add(variable);
4          variables.addAll(variable.getParents());
5
6          ArrayList<Variable> varToRemove = new ArrayList<>();
7          ArrayList<Integer> colToRemove = new ArrayList<>();
8          for (int i = 0; i < variables.size(); i++) {
9              if (variables.get(i).isObserved()) {
10                 varToRemove.add(variables.get(i));
11                 colToRemove.add(i);
12                 String value = variables.get(i).getObservedValue()
                       ;
13                 ArrayList<ProbRow> rowToRemove = new ArrayList<>()
                       ;
14                 for (ProbRow row : probTable.getTable()) {
15                     if (!row.getValues().get(i).equals(value)) {
16                         rowToRemove.add(row);
17                     }
18                 }
19                 for (ProbRow row : rowToRemove) {
20                     probTable.getTable().remove(row);
21                 }
22             } else {
23                 probabilities = probTable.getTable();
24             }
25         }
26         for (int i = 0; i < colToRemove.size(); i++) {
27             if (variables.size() > 1 && !variable.equals(
                   varToRemove.get(i))) {
28                 variables.remove(varToRemove.get(i));
29                 for (ProbRow row : probTable.getTable()) {
30                     int index = colToRemove.get(i);
31                     row.getValues().remove(index);
32                 }
33             }
34         }
35         probabilities = probTable.getTable();
```

```
36      }
```

**Listing 2:** Implementation of the Factor class

The algorithm is represented as a class and is responsible for calling all necessary factor operations and normalizing the final factor to retrieve the queried probability. It also configures the order of elimination according to given heuristic.

## 3.2 Data Structures Used

In order to find the most optimal implementation, a variety of data structures is used from Java's library. To establish an order of elimination, a PriorityQueue is used. The priority queue is able to easily specify the comparator, which is useful in determining the order of elimination. The factors are represented in a ArrayList, as it is more optimal in getting or setting elements from the list. Since the factors often interact, this is a useful property.

## 3.3 Least Incoming Arcs First

The main heuristic used to optimize efficiency is 'least incoming arcs first', which is one of the two common used heuristics mentioned in section 2 of this paper. This heuristic is implemented by comparing for each variable the number of parents that it has. The ones with the fewest parents will then be the nodes without a parent, which are relatively easy to compute. In listing 3, this is represented in line 2 using a PriorityQueue. After this, the order of the variables is revisited based on the observed variables, as can be seen in lines 4 to 6. After having computed the order, the eliminated variable is also likely to provide information to its children variables therefore influencing the computation considerably. The order of implementation is implemented the following:

```
1 private Queue<Variable> compriseOrder(Variable query, ArrayList<
     Variable> vars, ArrayList<Factor> factors) {
2         PriorityQueue<Variable> order = new PriorityQueue<>(
             Comparator.comparing(Variable::getNrOfParents));
3
4         for (Variable var : vars) {
5             if (!var.isObserved() && !var.equals(query)) {
6                 order.add(var);
7             }
8         }
9         return order;
```

**Listing 3:** Implementation of the least incoming arcs first heuristic

## 3.4 Algorithm

The main algorithm is implemented according to the design described in section 2. First, all factors are created by gathering the variables and probabilities from the input. This is

done in line 2 of listing 4. After that, an elimination order gets established based on the given heuristic, as can be seen in line 3. In this implementation it is least incoming arcs heuristic.

The main process loops over all the variables in their respective order and merges the concerning factors into a new factor. Merging refers to the multiplication and marginalization of two (or more) factors. This is done between line 5 and 11. In lines 16 to 18 there is an exception: if the queried variable is the ancestor of the observed variable, the factor will not be merged in the same process as the other factors. This is because the observed variables are not included in the elimination order, therefore the observed variables are not always eliminated but do contain the query. Lastly, the final factor contains all the relevant information and is normalized in line 23.

```
1  public void runElimination(Variable query, ArrayList<Variable>
       vars, ArrayList<Table> probs) {
2          ArrayList<Factor> factors = factorize(vars, probs);
3          Queue<Variable> elimOrder = compriseOrder(query, vars,
               factors);
4
5          while (!elimOrder.isEmpty()) {
6              Variable eliminate = elimOrder.poll();
7              ArrayList<Factor> concerningFactors =
                   getFactorsContainingEliminate(eliminate, factors);
8
9              if (concerningFactors.size() > 1) {
10                 Factor mergedFactor = new Factor(concerningFactors
                       , eliminate);
11                 factors.add(mergedFactor);
12             }
13         }
14         Factor finalFactor;
15
16         if (factors.size() > 1) {
17             finalFactor = new Factor(factors, null);
18         }
19         else {
20             finalFactor = factors.get(0);
21         }
22
23         finalFactor = normalize(finalFactor);
24         ui.printQueryAnswer(finalFactor.toString());
```

**Listing 4:** Implementation of the main Variable Elimination algorithm

### 3.5 Processing Non-Binary Variables

As an additional feature to the algorithm, the implementation can handle non-binary values for variables. Every value is represented as a String. When multiplying and marginalizing the factors, the first value of the probability table is taken and will be used to make 'variations' of: all the other variations that need to summed with this one in order to marginalize the factor. Instead of switching the variation's value from True to False or vice versa, the algorithm loops over all the possible values a variable can have, as can be seen in line 4 and 5 of listing 5. With this implementation, it is possible to take an arbitrary amount of values for a variable and have it represented in the network. To check the position of the variation, we loop over the other entries in the table until the right one is found. Once all variations have been covered, the marginalization is complete.

```
1  // Marginalization over the variable to eliminate
2          for (ProbRow example : factor.probabilities) {
3              ArrayList<String> exampleValues = example.getValues();
4              if (!usedExamples.contains(exampleValues)){
5                  ArrayList<ArrayList<String>> variations =
                       getAllVariations(index, eliminate, example.
                       getValues());
6                  usedExamples.addAll(variations);
7                  ProbRow variation = new ProbRow(variations.get(0),
                       0);
8                  for (ProbRow row : factor.probabilities) {
9                      ArrayList<String> dummy = row.getValues();
10                     if (variations.contains(dummy)) {
11                         variation.setProb(variation.getProb() +
                               row.getProb());
12                     }
13                 }
14                 variation.getValues().remove(index);
15                 finalTable.add(variation);
16                 if (finalTable.size() == tableSize) {
17                     break;
18                 }
19             }
20         }
```

**Listing 5:** Implementation of non-binary variables

## 4 Testing

To test the working and efficiency of the algorithm, the test have been designed to match different situations where bugs can occur. These tests are split into two parts: binary variables

and non-binary variables. To represent the binary variables, the tests are written for the well-known earthquake problem. To test how the algorithm processes non-binary values, a short problem is created called 'Bike Repair', represented in figure 2. It contains four states, that can take either two or three different values. Both problems are represented in the 'Belief and Decision Network' application from AISpace to verify the output.[4]

## 4.1 Test cases for the Earthquake problem

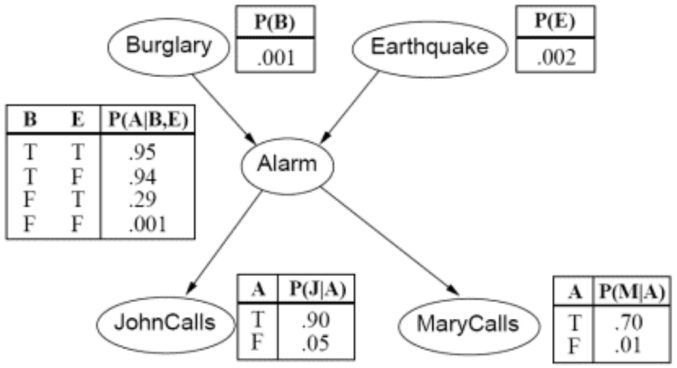


**Figure 1:** A graph representation of the 'Earthquake' sample problem.

### 4.1.1 Test 1

The first test case is relatively simple and tests the relationship between the observed parent node and queried child node work. In addition it also looks at if it gives the correct results. The test is looking at a probability of alarm going off given that the burglary is known to be happening.

**Query:** Alarm

**Observed:** Burglary = True

**Expected outcome** should be have a high probability of alarm being active and a very small probability of alarm being inactive due to the burglary being set to true.

**Actual result**

| Alarm | Probability |
|---|---|
| True | 0.9402 |
| False | 0.0598 |

As expected, the outcome for the query concludes that it is very probable for the alarm to be active and is pretty unlikely for the alarm to inactive. This makes sense given that the burglary is in progress.

#### 4.1.2 Test 2

The next test is pretty interesting. The goal of such a test is to see if the probability calculation if the query is made on a child variable of another given that another child is observed. The situation in this is that the call from Mary is received and the question follows: what is the probability of John calling after we know Mary already called?

**Query:** JohnCalls

**Observed:** MaryCalls = True

**Expected outcome** must be quite different than if the MaryCalls would not be observed. Since we know that Mary gives a call than it should increase the probability of John also giving a call.

**Actual result**

| JohnCalls | Probability |
|---|---|
| True | 0.504 |
| False | 0.496 |

Again, as expected the probability of John calling is dramatically increased. The probabilities of calling and not calling are almost at the ration 50/50. This test shows that observing one successor while querying on another parallel successor of the same variable, makes a huge difference to the outcome.

#### 4.1.3 Test 3

In test 1 the parent-child relation was tested. The next test is similar but now the observation and the query are switched around. The test looks at a situation in which the call from Mary is received. What is the probability that Mary is going to tell that the alarm went off?

**Query:** Alarm

**Observed:** MaryCalls = True

**Expected outcome** would be to see the increase in the alarm being true given that one of its successor nodes is observed.

**Actual result**

| Alarm | Probability |
|---|---|
| True | 0.53412 |
| False | 0.46588 |

The results are as expected. The observation of one of the successors increases the likelihood of the predecessor to be true.

#### 4.1.4 Test 4

The following test looks at what happens with the probabilities if all successors of the queried variable are observed. What are the chances of alarm going off given that both persons call?

**Query:** Alarm

**Observed:** JohnCalls = True, MaryCalls = True

**Expected outcome** should be such that increases the probability of alarm to be true very significantly since both calls are observed as positive. If both John and Mary calls, one should expect something serious has happened.

**Actual result**

| Alarm | Probability |
|---|---|
| True | 0.95378 |
| False | 0.04622 |

#### 4.1.5 Test 5

Next test is very similar to the previous test. It tests exactly the same relation of the queried variable and its observed child node. The difference this time is that the observed variable has another parent node which is not observed. The situation is modelled in such a way that the goal is to predict the likelihood of that the cause to alarm is the earthquake.

**Query:** Earthquake

**Observed:** Alarm = True

**Expected outcome** for the earthquake to be true given the alarm is on should not be too high since alarm has another possible cause namely the burglary.

**Actual result**

| Earthquake | Probability |
|---|---|
| True | 0.36812 |
| False | 0.63188 |

The results show a decent probability for the earthquake to be the cause of the alarm but the most likely case is that the earthquake is not to blame and there is another explanation for the alarm to go off. The results satisfy the expectation.

#### 4.1.6 Test 6

The final test of the Earthquake is our favorite. The test is quite complicated since there are three observed variables which all impact the result. The test looks into the probability of the earthquake being the cause given the burglary in progress and receiving both calls.

**Query:** Earthquake

**Observed:** Burglary = True, JohnCalls = True, MaryCalls = True

**Expected outcome** for the earthquake should be relatively low because there is already an explanation for the both observed calls.

**Actual result**

| Earthquake | Probability |
|---|---|
| True | 0.02021 |
| False | 0.97979 |

The outcome of the test is not very surprising because the cause is already known to be burglary which explains the calls received from both John and Mary.

### 4.2 Test cases for the Bike Repair problem

The following problem is made by us and the main focus of this problem is to show that the algorithm can handle variables with non-binary values. Also, it should be noted that the probabilities for the given problem are at the bottom of the appendix.

#### 4.2.1 Test 1

The test is modelled in such a way to see if it can handle the non-binary variables. The test is pretty funny and it tries to predict the probability of the weather given the quality of the bike.

**Query:** Weather

**Observed:** ConditionBike = Medium

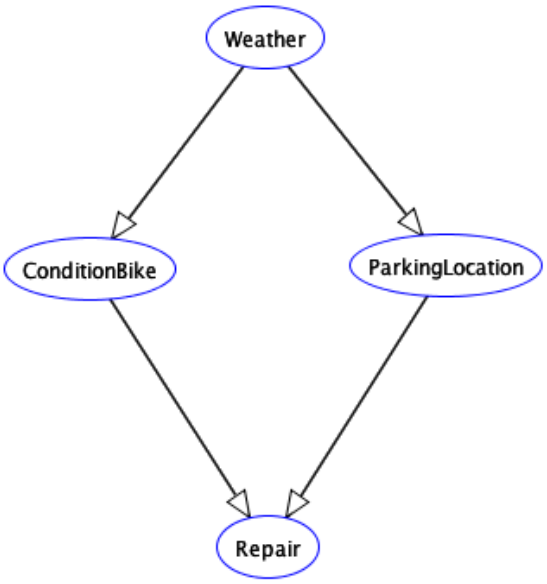


**Figure 2:** A graph representation of the 'Bike Repair' sample problem.

**Expected outcome** is quite hard to predict given the comical situation of predicting such complicated phenomenon as weather given such a weird observation as bike quality.

**Actual result**

| Weather | Probability |
|---|---|
| Rainy | 0.73684 |
| Sunny | 0.15789 |
| Snow | 0.10526 |

It looks like that according to the given problem it is most likely to be rainy if the bike is of mediocre quality.

### 4.2.2 Test 2

The test looks at the turned around situation of the previous test. Now the situation is even more comical which wants to predict the quality of the bike given the observed weather is sunny.

**Query:** ConditionBike

**Observed:** Weather = Sunny

**Expected outcome** again is hard to predict since both variables contain three possible values and the situation is quite unrealistic which leads to prediction difficulties.

**Actual result**

| ConditionBike | Probability |
|---|---|
| Good | 0.4 |
| Medium | 0.3 |
| Broken | 0.3 |

The outcome of course does not make a lot of sense but, as mentioned before, the resulting probabilities have been tested using the specialized program for such scenarios called Belief and Decision Network Tool.

# 5 Conclusion

The research question of this paper is: "To what extend can Bayesian networks be further optimized by improving the order of variables?". The implementation is based around the least incoming arcs first principle, and managed to pass all the tests (i.e. shows the same results as the Belief and Decision Network Tool from AISpace).[4] The algorithm performs well on both binary and non-binary variables according to the test. However, having the right answer is not related to the

The algorithm is written with efficiency in mind, but since the implementation features only one heuristic no comparisons can be made. Because of this, no conclusions can be drawn on the efficiency of the implementation unfortunately. This would make for an interesting future research however. Other possible future research would be to test the scaling of the running time complexity, using larger sized problems that can be found on various websites.

# References

[1] Rina derchter, wikipedia, 2018. From Wikipedia, the free encyclopedia.

[2] Variable elimination, wikipedia., 2018.

[3] David Poole and AlanMacWorth. David poole and alanmacworth, artificial intelligence: Foundations of computational agents, web version., 2018.

[4] Jacek Kisyński Shinjiro Sueda Kyle Porter, David Poole, Holger Hoos Peter Gorniak Byron Knoll, with help from Alan Mackworth, and Cristina Conati. Aispace, 2008.

# 6 Appendix

## 6.1 Log file

The following results are from the Earthquake.bif problem:
TEST 1:
The queried variable(s) is/are: Alarm
The observed variable(s) is/are:
Burglary This variable has the value: True
The reduced formula based on the network structure: P(B=True)·P(E)·P(A|B=True,E)·P(J|A)·P(M|A)
The formula of the reduced factors: f1(E)·f2(A,E)·f3(J,A)·f4(M,A)
The elimination order, based on least incoming arcs: 1. E 2. M 3. J
Merging the following factors by eliminating E variable f1(E) f2(A,E)
Formula after the merge: f1(J,A)·f2(M,A)·f3(A)
The following factor is safely ignored: f1(M,A)
Formula after the merge: f1(J,A)·f2(A)
The following factor is safely ignored: f1(J,A)
Formula after the merge: f1(A)
Final answer: Alarm [True] | 0.9402 [False] | 0.0598
TEST 2:
The queried variable(s) is/are: JohnCalls
The observed variable(s) is/are: MaryCalls This variable has the value: True
The reduced formula based on the network structure: P(B)·P(E)·P(A|B,E)·P(J|A)·P(M=True|A)
The formula of the reduced factors: f1(B)·f2(E)·f3(A,B,E)·f4(J,A)·f5(M=True,A)
The elimination order, based on least incoming arcs: 1. B 2. E 3. A
Merging the following factors by eliminating B variable f1(B) f2(A,B,E)
Formula after the merge: f1(E)·f2(J,A)·f3(M=True,A)·f4(A,E)
Merging the following factors by eliminating E variable f1(E) f2(A,E)
Formula after the merge: f1(J,A)·f2(M=True,A)·f3(A)
Merging the following factors by eliminating A variable f1(J,A) f2(M=True,A) f3(A)
Formula after the merge: f1(J)
Final answer: JohnCalls [True] | 0.504 [False] | 0.496
TEST 3:
The queried variable(s) is/are: Alarm
The observed variable(s) is/are:
MaryCalls This variable has the value: True
The reduced formula based on the network structure: P(B)·P(E)·P(A|B,E)·P(J|A)·P(M=True|A)
The formula of the reduced factors: f1(B)·f2(E)·f3(A,B,E)·f4(J,A)·f5(M=True,A)
The elimination order, based on least incoming arcs: 1. B 2. E 3. J
Merging the following factors by eliminating B variable f1(B) f2(A,B,E)
Formula after the merge: f1(E)·f2(J,A)·f3(M=True,A)·f4(A,E)
Merging the following factors by eliminating E variable f1(E) f2(A,E)
Formula after the merge: f1(J,A)·f2(M=True,A)·f3(A)
The following factor is safely ignored: f1(J,A)
Formula after the merge: f1(M=True,A)·f2(A)
Merging last remaining factors: f1(M=True,A) f2(A)
Final answer: Alarm [True] | 0.53412 [False] | 0.46588
TEST 4:
The queried variable(s) is/are: Alarm
The observed variable(s) is/are:
JohnCalls This variable has the value: True
MaryCalls This variable has the value: True
The reduced formula based on the network structure: P(B)·P(E)·P(A|B,E)·P(J=True|A)·P(M=True|A)
The formula of the reduced factors: f1(B)·f2(E)·f3(A,B,E)·f4(J=True,A)·f5(M=True,A)
The elimination order, based on least incoming arcs: 1. B 2. E
Merging the following factors by eliminating B variable f1(B) f2(A,B,E)
Formula after the merge: f1(E)·f2(J=True,A)·f3(M=True,A)·f4(A,E)
Merging the following factors by eliminating E variable f1(E) f2(A,E)
Formula after the merge: f1(J=True,A)·f2(M=True,A)·f3(A)
Merging last remaining factors: f1(J=True,A) f2(M=True,A) f3(A)
Final answer: Alarm [True] | 0.95378 [False] | 0.04622
TEST 5:
The queried variable(s) is/are: Earthquake
The observed variable(s) is/are:
Alarm This variable has the value: True

The reduced formula based on the network structure: P(B)·P(E)·P(A=True|B,E)·P(J|A=True)·P(M|A=True)
The formula of the reduced factors: f1(B)·f2(E)·f3(A=True,B,E)·f4(J)·f5(M)
The elimination order, based on least incoming arcs: 1. B 2. M 3. J
Merging the following factors by eliminating B variable f1(B) f2(A=True,B,E)
Formula after the merge: f1(E)·f2(J)·f3(M)·f4(A=True,E)
The following factor is safely ignored: f1(M)
Formula after the merge: f1(E)·f2(J)·f3(A=True,E)
The following factor is safely ignored: f1(J)
Formula after the merge: f1(E)·f2(A=True,E)
Merging last remaining factors: f1(E) f2(A=True,E)
Final answer: Earthquake [True] | 0.36812 [False] | 0.63188
TEST 6:
The queried variable(s) is/are: Earthquake
The observed variable(s) is/are:
Burglary This variable has the value: True
JohnCalls This variable has the value: True
MaryCalls This variable has the value: True
The reduced formula based on the network structure: P(B=True)·P(E)·P(A|B=True,E)·P(J=True|A)·P(M=True|A)
The formula of the reduced factors: f1(E)·f2(A,E)·f3(J=True,A)·f4(M=True,A)
The elimination order, based on least incoming arcs: 1. A
Merging the following factors by eliminating A variable f1(A,E) f2(J=True,A) f3(M=True,A)
Formula after the merge: f1(E)·f2(E)
Merging last remaining factors: f1(E) f2(E)
Final answer: Earthquake [True] | 0.02021 [False] | 0.97979
The following tests are from the problem of our own creation.
TEST 1:
The queried variable(s) is/are: Weather
The observed variable(s) is/are:
ConditionBike This variable has the value: Medium
The reduced formula based on the network structure: P(W)·P(C=Medium|W)·P(P|W)·P(R)
The formula of the reduced factors: f1(W)·f2(C=Medium,W)·f3(P,W)·f4(R)
The elimination order, based on least incoming arcs: 1. R 2. P
The following factor is safely ignored: f1(R)
Formula after the merge: f1(W)·f2(C=Medium,W)·f3(P,W)
The following factor is safely ignored: f1(P,W)
Formula after the merge: f1(W)·f2(C=Medium,W)
Merging last remaining factors: f1(W) f2(C=Medium,W)
TEST 2
Weather [Medium, Rainy] | 0.73684 [Medium, Sunny] | 0.15789 [Medium, Snow] | 0.10526
The queried variable(s) is/are: ConditionBike
The observed variable(s) is/are:
Weather This variable has the value: Sunny
The reduced formula based on the network structure: P(W=Sunny)·P(C|W=Sunny)·P(P|W=Sunny)·P(R)
The formula of the reduced factors: f1(C)·f2(P)·f3(R)
The elimination order, based on least incoming arcs: 1. R 2. P
The following factor is safely ignored: f1(R)
Formula after the merge: f1(C)·f2(P)
The following factor is safely ignored: f1(P)
Formula after the merge: f1(C)

ConditionBike [Good] | 0.4 [Medium] | 0.3 [Broken] | 0.3

## 6.2 Main class

```
1  package varelim;
2
3  import java.util.ArrayList;
4
5  /**
6   * Main class to read in a network, add queries and observed
       variables, and run variable elimination.
```

```
7   *
8   * @author Marcel de Korte, Moira Berens, Djamari Oetringer,
        Abdullahi Ali, Leonieke van den Bulk
9   * @co-author/editor Mantas Makelis, David Leeftink
10  */
11
12  public class Main {
13
14      private final static String networkName = "earthquake.bif"; //
            The network to be read in (format and other
15      // networks can be found on http://www.bnlearn.com/
           bnrepository/)
16
17      public static void main(String[] args) {
18
19          // Read in the network
20          Networkreader reader = new Networkreader(networkName);
21
22          // Get the variables and probabilities of the network
23          ArrayList<Variable> vs = reader.getVs();
24          ArrayList<Table> ps = reader.getPs();
25
26          // Make user interface
27          UserInterface ui = new UserInterface(vs, ps);
28
29          // Print variables and probabilities
30          ui.printNetwork();
31
32          // Ask user for query and heuristic
33          ui.askForQuery();
34          Variable query = ui.getQueriedVariable();
35
36          // Ask user for observed variables
37          ui.askForObservedVariables();
38          ArrayList<Variable> observed = ui.getObservedVariables();
39
40          // Ask user for heuristic
41          //ui.askForHeuristic();
42          //String heuristic = ui.getHeuristic();
43
44          // Print the query and observed variables
45          ui.printQueryAndObserved(query, observed);
46
47          //PUT YOUR CALL TO THE VARIABLE ELIMINATION ALGORITHM HERE
48          Algorithm algorithm = new Algorithm(ui);
49          algorithm.runElimination(query, vs, ps);
```

```
50      }
51  }
```

**Listing 6:** Main.java

## 6.3 Class representing algorithm

```
1   package varelim;
2
3   import java.util.*;
4
5   /**
6    * Represents the variable elimination algorithm.
7    *
8    * @author Mantas Makelis, David Leeftink
9    */
10  public class Algorithm {
11
12      private UserInterface ui;
13
14      public Algorithm(UserInterface ui) {
15          this.ui = ui;
16      }
17
18      /**
19       * Runs variable elimination algorithm which makes factors and
             merges them in defined order until only one factor is
             left and the
20       * probability of the query can be determined.
21       * The order of the algorithm:
22       * 1. Collect all factors
23       * 2. Fix order
24       * 3. Multiply factors in order
25       * 4. Marginalize factors
26       * 5. Normalize results
27       *
28       * @param query the variable for which the probability needs
             to be determined
29       * @param vars a list of all the variables in the Bayesian
             network
30       * @param probs a list of the probability tables for each
             variable
31       */
32      public void runElimination(Variable query, ArrayList<Variable>
             vars, ArrayList<Table> probs) {
33          ui.printProductFormula(vars);
34          // Initialise factors according to all variables
```

```
35          ArrayList<Factor> factors = factorize(vars, probs);
36
37          // Create the order of elimination according to the number
                of parents
38          Queue<Variable> elimOrder = compriseOrder(query, vars,
               factors);
39          ui.printEliminationOrder(new PriorityQueue<>(elimOrder));
40
41          while (!elimOrder.isEmpty()) {
42              Variable eliminate = elimOrder.poll();
43
44              // Retrieve factors which contain the popped variable
45              ArrayList<Factor> concerningFactors =
                   getFactorsContainingEliminate(eliminate, factors);
46              ui.printMergingFactors(concerningFactors, eliminate);
47
48              // Eliminate the variable which is contained in at
                   least 2 factors
49              if (concerningFactors.size() > 1) {
50                  Factor mergedFactor = new Factor(concerningFactors
                       , eliminate);
51                  factors.add(mergedFactor);
52              }
53              // Any factor concerning 1 variable results in (1,1)
                   probability.
54              ui.printFactorFormula(factors, false);
55          }
56          Factor finalFactor;
57
58          // In case the query is the ancestor of observed variables
                .
59          if (factors.size() > 1) {
60              ui.printLastFactorMerge(factors);
61              finalFactor = new Factor(factors, null);
62          }
63          else {
64              finalFactor = factors.get(0);
65          }
66
67          // Gather results
68          finalFactor = normalize(finalFactor);
69          ui.printQueryAnswer(finalFactor.toString());
70
71      }
72
73      /**
```

```
74      * Normalize the final factor by the formula : factor / (sum
            of probabilities of the factor)
75      *
76      * @param finalFactor
77      * @return
78      */
79     private Factor normalize(Factor finalFactor) {
80         // Sum up all the probabilities
81         double sumProb = 0;
82         for (ProbRow row : finalFactor.getProbabilities()) {
83             sumProb += row.getProb();
84         }
85         for (ProbRow probability : finalFactor.getProbabilities())
               {
86             // Normalize results, round to 5 digits.
87             probability.setProb(Math.round((probability.getProb()
                   / sumProb) * 100000.0) / 100000.0);
88         }
89         return finalFactor;
90     }
91
92
93     /**
94      * Converts all the variables to a list of the factors.
95      *
96      * @param vars a list of all the variables in the Bayesian
            network
97      * @param probs a list of the probability tables for each
            variable
98      * @return a list containing all the factors
99      */
100    private ArrayList<Factor> factorize(ArrayList<Variable> vars,
           ArrayList<Table> probs) {
101        ArrayList<Factor> factors = new ArrayList<>();
102        // Create factors out of all variables which are NOT
               observed
103        for (Variable var : vars) {
104            Factor factor = new Factor(var, getProb(var, probs));
105            factors.add(factor);
106        }
107        ArrayList<Factor> fullyObserved = getFullyObserved(factors
               );
108        ui.printFactorFormula(fullyObserved, true);
109        return fullyObserved;
110    }
111
```

```
112     /**
113      * Gets the fully observed factors removed.
114      * Remove factors with only observed.
115      *
116      * @param factors factor list
117      * @return a list of factors
118      */
119     private ArrayList<Factor> getFullyObserved(ArrayList<Factor>
            factors) {
120         ArrayList<Factor> updatedFactors = new ArrayList<>();
121         for (Factor factor : factors) {
122             boolean remove = true;
123             for (Variable variable : factor.getVariables()) {
124                 if (!variable.isObserved()) {
125                     remove = false;
126                 }
127             }
128             if (!remove) {
129                 updatedFactors.add(factor);
130             }
131         }
132         return updatedFactors;
133     }
134 
135     /**
136      * Comprises the order in which to eliminate the variables.
137      * Consists of least-arcs incoming, fewest factors and random.
138      *
139      * @param query the variable for which the probability needs
            to be determined
140      * @param vars a list of all the variables in the Bayesian
            network.
141      * @return the order of elimination
142      */
143     private Queue<Variable> compriseOrder(Variable query,
            ArrayList<Variable> vars, ArrayList<Factor> factors) {
144         // Initialise the priority queue which compares members by
                the number of their parents
145         PriorityQueue<Variable> order = new PriorityQueue<>(
               Comparator.comparing(Variable::getNrOfParents));
146 
147         // Add variables which are NOT observed and is not a query
148         for (Variable var : vars) {
149             if (!var.isObserved() && !var.equals(query)) {
150                 order.add(var);
151             }
```

```
152         }
153         return order;
154     }
155 
156     /**
157      * Retrieves the corresponding probability table for a
            variable.
158      *
159      * @param var the variable for which to retrieve the table
160      * @param probs a list of the probability tables for each
            variable
161      * @return the corresponding probability table
162      */
163     public Table getProb(Variable var, ArrayList<Table> probs) {
164         for (Table prob : probs) {
165             if (prob.getVariable().equals(var)) {
166                 return prob;
167             }
168         }
169         return null;
170     }
171 
172     /**
173      * Gets factors which contain the given variable.
174      *
175      * @param var the variable for which to look in factors
176      * @param factors a list of all the factors
177      * @return a list containing only the factors which contain
            the given variable
178      */
179     public ArrayList<Factor> getFactorsContainingEliminate(
           Variable var, ArrayList<Factor> factors) {
180         ArrayList<Factor> containing = new ArrayList<>();
181         // Add factor to the containing if the factor contains
               variable to eliminate
182         for (Factor factor : factors) {
183             if (factor.containsVariable(var)) {
184                 containing.add(factor);
185             }
186         }
187         // Remove all factors from the factor list that contained
               the variable to eliminate
188         for (Factor factor : containing) {
189             factors.remove(factor);
190         }
191         return containing;
```

```
192     }
193 }
```

**Listing 7:** Algorithm.java

## 6.4 Class representing Factor

```
1  package varelim;
2
3  import java.util.ArrayList;
4  import java.util.Comparator;
5  import java.util.LinkedList;
6  import java.util.PriorityQueue;
7
8  /**
9   * Represents the factor in variable elimination algorithm and is
        the part of it.
10  * The class contains few constructors. One constructor
        initialises factors, others merge a list of factors into one.
11  *
12  * @author Mantas Makelis, David Leeftink
13  */
14 public class Factor {
15
16     private ArrayList<Variable> variables; // Variables that are
           contained in the factor
17     private ArrayList<ProbRow> probabilities; // The probability
           table of this factor
18
19     /**
20      * This constructor is used to initialise the factor out of a
           variable.
21      * NOTE: It also eliminates the unnecessary probabilities of
           the observed variables.
22      *
23      * @param variable the variable to which the factor belongs
24      * @param probTable the probability table of the variable
25      */
26     public Factor(Variable variable, Table probTable) {
27         // Add all variables that the factor contains
28         variables = new ArrayList<>();
29         variables.add(variable);
30         variables.addAll(variable.getParents());
31
32         // collecting items of the observed values in the factors
33         ArrayList<Variable> varToRemove = new ArrayList<>();
34         ArrayList<Integer> colToRemove = new ArrayList<>();
```

```
35          for (int i = 0; i < variables.size(); i++) {
36              if (variables.get(i).isObserved()) {
37                  varToRemove.add(variables.get(i));
38                  colToRemove.add(i);
39                  String value = variables.get(i).getObservedValue()
                        ;
40                  ArrayList<ProbRow> rowToRemove = new ArrayList<>()
                        ;
41                  for (ProbRow row : probTable.getTable()) {
42                      if (!row.getValues().get(i).equals(value)) {
43                          rowToRemove.add(row);
44                      }
45                  }
46                  for (ProbRow row : rowToRemove) {
47                      probTable.getTable().remove(row);
48                  }
49              } else {
50                  probabilities = probTable.getTable();
51              }
52          }
53
54          // removing the observed values from the factors
55          for (int i = 0; i < colToRemove.size(); i++) {
56              if (variables.size() > 1 && !variable.equals(
                    varToRemove.get(i))) {
57                  variables.remove(varToRemove.get(i));
58                  for (ProbRow row : probTable.getTable()) {
59                      int index = colToRemove.get(i);
60                      row.getValues().remove(index);
61                  }
62              }
63          }
64
65          // Add probability table to the factor
66          probabilities = probTable.getTable();
67      }
68
69      /**
70       * Constructor of the factor from other factors with the
             variable which must be eliminated in progress
71       * This constructor is only used to merge a number of factors
             into one.
72       *
73       * @param factors a list of factors which contain the
             eliminate variable which must be merged
74       * @param eliminate a variable which will be eliminated in the
```

```
         process
     */
    public Factor(ArrayList<Factor> factors, Variable eliminate) {
        Factor factor = new Factor();

        // the general case
        if (eliminate != null) {
            factor = normalMerge(factors,eliminate);
        }

        // in case where query is the ancestor of the observed
        else {
            factor = specialMerge(factors);
        }

        // set final factor values
        variables = factor.getVariables();
        probabilities = factor.getProbabilities();
    }



    /**
     * The general merging case.
     * Takes two factors as a parameter, then multiplies them and
        marginalizes them.
     *
     * @param factors
     * @param eliminate
     * @return Merged factor (multiplied, then marginalized).
     */
    private Factor normalMerge(ArrayList<Factor> factors, Variable
        eliminate) {

        // Initialise null factor for future use
        Factor factor = new Factor();

        PriorityQueue<Factor> factorQueue = new PriorityQueue<>(
           Comparator.comparing(Factor::getNoOfVariables));
        factorQueue.addAll(factors);

        // Loop over factors
        while (!factorQueue.isEmpty()) {
            if (factor.isNull()) {
                // First time of the loop, pop two factors out and
                    merge them with another constructor
```

```
116                 factor = mergeFactors(factorQueue.poll(),
                        factorQueue.poll(), eliminate);
117             } else {
118                 // All the other times of the loop, pop one factor
                        and merge with the previously merged factor
119                 factor = mergeFactors(factor, factorQueue.poll(),
                        eliminate);
120             }
121         }
122 
123         // Calculate the right probability table size
124         int tableSize = 1;
125         for (Variable var : factor.getVariables()) {
126             if (!var.equals(eliminate) && !var.isObserved()) {
127                 tableSize *= var.getNumberOfValues();
128             }
129         }
130         ArrayList<ProbRow> finalTable = new ArrayList<>();
131         ArrayList<ArrayList<String>> usedExamples = new ArrayList
                <>();
132 
133         int index = factor.variables.indexOf(eliminate);
134 
135         // Marginalization over the variable to eliminate
136         for (ProbRow example : factor.probabilities) {
137             ArrayList<String> exampleValues = example.getValues();
138             if (!usedExamples.contains(exampleValues)){
139                 ArrayList<ArrayList<String>> variations =
                        getAllVariations(index, eliminate, example.
                        getValues());
140                 usedExamples.addAll(variations);
141                 ProbRow variation = new ProbRow(variations.get(0),
                        0);
142                 for (ProbRow row : factor.probabilities) {
143                     ArrayList<String> dummy = row.getValues();
144                     if (variations.contains(dummy)) {
145                         variation.setProb(variation.getProb() +
                                row.getProb());
146                     }
147                 }
148                 variation.getValues().remove(index);
149                 finalTable.add(variation);
150                 if (finalTable.size() == tableSize) {
151                     break;
152                 }
153             }
```

```
154         }
155 
156         // Retrieve and remove variables from the merged factor
157         ArrayList<Variable> vars = factor.getVariables();
158         vars.remove(eliminate);
159         return new Factor(vars,finalTable);
160     }
161 
162     /**
163      * Merges the factors, in case the query variable is the
             ancestor of the observed variable(s).
164      * @param factors
165      * @return merged factor (multiplied)
166      */
167     private Factor specialMerge(ArrayList<Factor> factors) {
168 
169         // Initiate new factor for future use
170         Factor factor = new Factor();
171         Variable query = new Variable();
172 
173         // determine query variable and its factor
174         for (Factor mainFactor : factors) {
175             if (mainFactor.getVariables().size() == 1) {
176                 query = mainFactor.getVariables().get(0);
177                 factor = mainFactor;
178                 break;
179             }
180         }
181         factors.remove(factor);
182         LinkedList<Factor> eliminateList = new LinkedList<>(
               factors);
183         ArrayList<Variable> varsToRemove = new ArrayList<>();
184 
185         // Loop over all factors to eliminate, and then merge.
186         while (!eliminateList.isEmpty()) {
187             Factor factorToMerge = eliminateList.poll();
188             for (Variable var : factorToMerge.getVariables()) {
189                 if (!var.equals(query)) {
190                     varsToRemove.add(var);
191                     break;
192                 }
193             }
194             factor = mergeFactors(factor, factorToMerge, query);
195         }
196 
197         // remove unneccesary the variables
```

```
198         for (Variable var : varsToRemove) {
199             factor.variables.remove(var);
200         }
201         return factor;
202     }
203 
204     /**
205      * Creates possible variations that need to be add up in the
            marginalization.
206      * @param index
207      * @param eliminate
208      * @param example
209      * @return List of variations (Which is an ArrayList of
            Strings)
210      */
211     private ArrayList<ArrayList<String>> getAllVariations(int
        index, Variable eliminate, ArrayList<String> example) {
212         ArrayList<ArrayList<String>> values = new ArrayList<>();
213         for (String value : eliminate.getValues()) {
214             ArrayList<String> newExample = (ArrayList<String>)
                example.clone();
215             newExample.set(index, value);
216             values.add(newExample);
217         }
218         return values;
219     }
220 
221     /**
222      * The merger factor constructor which is only used in the
            constructor of multiple factors.
223      * This constructor creates one factor out of two.
224      *
225      * @param factor1 first factor
226      * @param factor2 second factor
227      * @param eliminate the variable to eliminate
228      */
229     private Factor mergeFactors(Factor factor1, Factor factor2,
        Variable eliminate) {
230         // Initialise the array of combined probabilities
231         ArrayList<ProbRow> combinedProbs = new ArrayList<>();
232 
233         // index used for identifying the right variable
234         int index1 = factor1.variables.indexOf(eliminate);
235         int index2 = factor2.variables.indexOf(eliminate);
236         ArrayList<ProbRow> probs1 = factor1.probabilities;
237         ArrayList<ProbRow> probs2 = factor2.probabilities;
```

```
238
239         // Loop over both factors
240         for (ProbRow row1 : probs1) {
241             for (ProbRow row2 : probs2) {
242
243                 // determine right format for the new factor
244                 if (row1.getValues().get(index1).equals(row2.
                        getValues().get(index2))) {
245                     double prob = row1.getProb() * row2.getProb();
246                     ProbRow row;
247                     if (probs1.size() > probs2.size()) {
248                         row = new ProbRow(row1.getValues(), prob);
249                     } else {
250                         row = new ProbRow(row2.getValues(), prob);
251                     }
252                     combinedProbs.add(row);
253                 }
254             }
255         }
256
257         // Create the merged factor
258         ArrayList<Variable> finalVars = factor1.variables.size() <
                factor2.variables.size() ? factor2.variables : factor1
               .variables;
259         Factor merged = new Factor(finalVars, combinedProbs);
260         return merged;
261     }
262
263     /**
264      * Empty constructed, used to initialize variables.
265      */
266     public Factor() {
267     }
268
269     /**
270      * A constructor which sets the variables and probabilities.
271      *
272      * @param variables variables of the factor
273      * @param probabilities probabilities of the factor
274      */
275     private Factor(ArrayList<Variable> variables, ArrayList<
           ProbRow> probabilities) {
276         this.variables = variables;
277         this.probabilities = probabilities;
278     }
279
```

```
280     /**
281      * A check if the factor is empty.
282      *
283      * @return true if not empty, otherwise, false
284      */
285     private boolean isNull() {
286         return probabilities == null && variables == null;
287     }
288 
289     /**
290      * Getter for the variables of the factor.
291      *
292      * @return variables
293      */
294     public ArrayList<Variable> getVariables() {
295         return variables;
296     }
297 
298     /**
299      * Getter for probabilities of the factor.
300      *
301      * @return probabilities
302      */
303     public ArrayList<ProbRow> getProbabilities() {
304         return probabilities;
305     }
306 
307     /**
308      * Gets variable count in the factor.
309      *
310      * @return count of variables
311      */
312     public int getNoOfVariables() {
313         return variables.size();
314     }
315 
316     /**
317      * A check to see if the supplied variable is in the factor.
318      *
319      * @param var the variable for which to look
320      * @return true if the factor contains the variable, otherwise
         , false
321      */
322     public boolean containsVariable(Variable var) {
323         return variables.contains(var);
324     }
```

```
325
326     /**
327      * Converts the factor probabilities table to a string.
328      *
329      * @return a string of the table
330      */
331     @Override
332     public String toString() {
333         StringBuilder sb = new StringBuilder();
334         sb.append("\n").append(variables.get(0).getName()).append(
                "\n");
335         for (ProbRow pr : probabilities) {
336             sb.append(pr.getValues()).append(" | ").append(pr.
                    getProb()).append("\n");
337         }
338         return sb.toString();
339     }
340 }
```

**Listing 8:** Factor.java

## 6.5 Class representing user interface

```
1  package varelim;
2
3  import java.util.ArrayList;
4  import java.util.PriorityQueue;
5  import java.util.Scanner;
6
7  /**
8   * Class that handles the communication with the user.
9   *
10  * @author Marcel de Korte, Moira Berens, Djamari Oetringer,
        Abdullahi Ali, Leonieke van den Bulk
11  * @co-author/editor Mantas Makelis, David Leeftink
12  */
13
14 public class UserInterface {
15
16     private ArrayList<Variable> vs;
17     private ArrayList<Table> ps;
18     private Variable query = null;
19     private ArrayList<Variable> obs = new ArrayList<>();
20     private String line;
21     private String heuristic;
22     private Scanner scan;
23
```

```
24      /**
25       * Constructor of the user interface.
26       *
27       * @param vs, the list of variables.
28       * @param ps, the list of probability tables.
29       */
30      public UserInterface(ArrayList<Variable> vs, ArrayList<Table>
            ps) {
31          this.vs = vs;
32          this.ps = ps;
33      }
34
35      /**
36       * Asks for a query from the user.
37       */
38      public void askForQuery() {
39          System.out.println("\nWhich variable(s) do you want to
                query? Please enter in the number of the variable.");
40          for (int i = 0; i < vs.size(); i++) {
41              System.out.println("Variable " + i + ": " + vs.get(i).
                    getName());
42          }
43          scan = new Scanner(System.in);
44          line = scan.nextLine();
45          if (line.isEmpty()) {
46              System.out.println("You have not chosen a query value.
                    Please choose a query value.");
47              askForQuery();
48          }
49          try {
50              int queriedVar = Integer.parseInt(line);
51              if (queriedVar >= 0 && queriedVar < vs.size()) {
52                  query = vs.get(queriedVar);
53              } else {
54                  System.out.println("This is not a correct index.
                        Please choose an index between " + 0 + " and "
55                          + (vs.size() - 1) + ".");
56                  askForQuery();
57              }
58          } catch (NumberFormatException ex) {
59              System.out.println("This is not a correct index.
                    Please choose an index between " + 0 + " and "
60                      + (vs.size() - 1) + ".");
61              askForQuery();
62          }
63      }
```

```
64
65      /**
66       * Ask the user for observed variables in the network.
67       */
68      public void askForObservedVariables() {
69
70          obs.clear();
71          System.out.println("Which variable(s) do you want to
                observe? Please enter in the number of the variable, \n
                "
72              + "followed by a comma and the value of the observed
                    variable. Do not use spaces. \n"
73              + "If you want to query multiple variables, delimit
                    them with a ';' and no spaces.\n"
74              + "Example: '2,True;3,False'");
75          for (int i = 0; i < vs.size(); i++) {
76              StringBuilder values = new StringBuilder();
77              for (int j = 0; j < vs.get(i).getNumberOfValues() - 1;
                    j++) {
78                  values.append(vs.get(i).getValues().get(j)).append
                        (", ");
79              }
80              values.append(vs.get(i).getValues().get(vs.get(i).
                    getNumberOfValues() - 1));
81
82              System.out.println("Variable " + i + ": " + vs.get(i).
                    getName() + " - " + values);
83          }
84          scan = new Scanner(System.in);
85          line = scan.nextLine();
86          if (!line.isEmpty()) {
87              if (!line.contains(",")) {
88                  System.out.println("You did not enter a comma
                        between values. Please try again");
89                  askForObservedVariables();
90              } else {
91                  while (line.contains(";")) { // Multiple observed
                        variables
92                      try {
93                          int queriedVar = Integer.parseInt(line.
                                substring(0, line.indexOf(",")));
94                          String bool = line.substring(line.indexOf(
                                ",") + 1, line.indexOf(";"));
95                          changeVariableToObserved(queriedVar, bool)
                                ;
96                          line = line.substring(line.indexOf(";") +
```

```
                            1); // Continue
97                      // with
98                      // next
99                      // observed
100                     // variable.
101                 } catch (NumberFormatException ex) {
102                     System.out.println("This is not a correct
                            input. Please choose an index between "
                             + 0 + " and "
103                          + (vs.size() - 1) + ".");
104                     askForObservedVariables();
105                     return;
106                 }
107             }
108             if (!line.contains(";")) { // Only one observed
                    variable
109                 try {
110
111                     int queriedVar = Integer.parseInt(line.
                            substring(0, line.indexOf(",")));
112                     String bool = line.substring(line.indexOf(
                            ",") + 1);
113                     changeVariableToObserved(queriedVar, bool)
                            ;
114                 } catch (NumberFormatException ex) {
115                     System.out.println("This is not a correct
                            input. Please choose an index between "
                             + 0 + " and "
116                          + (vs.size() - 1) + ".");
117                     askForObservedVariables();
118                 }
119             }
120         }
121     }
122 }
123
124 /**
125  * Checks whether a number and value represent a valid
        observed value or not and if so, adds it to the
126  * observed list. If not, asks again for new input.
127  */
128 public void changeVariableToObserved(int queriedVar, String
        value) {
129     Variable ob;
130     if (queriedVar >= 0 && queriedVar < vs.size()) {
131         ob = vs.get(queriedVar);
```

```
132             if (ob.isValueOf(value)) {
133                 ob.setObservedValue(value);
134                 ob.setObserved(true);
135             } else {
136                 System.out.println("Apparently you did not fill in
                        the value correctly. You typed: \"" + value
137                     + "\"Please try again");
138                 askForObservedVariables();
139                 return;
140             }
141             obs.add(ob); // Adding observed variable and it's
                    value to list.
142         } else {
143             System.out.println("You have chosen an incorrect index
                    . Please choose an index between " + 0 + " and "
144                     + (vs.size() - 1));
145             askForObservedVariables();
146         }
147     }
148 
149     /**
150      * Print the network that was read-in (by printing the
            variables, parents and probabilities).
151      */
152     public void printNetwork() {
153         System.out.println("The variables:");
154         for (int i = 0; i < vs.size(); i++) {
155             StringBuilder values = new StringBuilder();
156             for (int j = 0; j < vs.get(i).getNumberOfValues() - 1;
                        j++) {
157                 values.append(vs.get(i).getValues().get(j)).append
                            (", ");
158             }
159             values.append(vs.get(i).getValues().get(vs.get(i).
                    getNumberOfValues() - 1));
160             System.out.println((i + 1) + ") " + vs.get(i).getName
                    () + " - " + values); // Printing
161             // the
162             // variables.
163         }
164         System.out.println("\nThe probabilities:");
165         for (int i = 0; i < ps.size(); i++) {
166             if (vs.get(i).getNrOfParents() == 1) {
167                 System.out.println(ps.get(i).getVariable().getName
                            () + " has parent "
168                     + vs.get(i).getParents().get(0).getName());
```

```
169             } else if (vs.get(i).getNrOfParents() > 1) {
170                 StringBuilder parentsList = new StringBuilder();
171                 for (int j = 0; j < vs.get(i).getParents().size();
                        j++) {
172                     parentsList.append(vs.get(i).getParents().get(
                            j).getName());
173                     if (!(j == vs.get(i).getParents().size() - 1))
                            {
174                         parentsList.append(" and ");
175                     }
176                 }
177                 System.out.println(ps.get(i).getVariable().getName
                        () + " has parents "
178                         + parentsList);
179             } else {
180                 System.out.println(ps.get(i).getVariable().getName
                        () + " has no parents.");
181             }
182 
183             Table probs = ps.get(i);
184             for (int l = 0; l < probs.size(); l++) {
185                 System.out.println(probs.get(l));      // Printing
186             }                           // the
187             System.out.println();               // probabilities.
188         }
189     }
190 
191     /**
192      * Prints the query and observed variables given in by the
            user.
193      */
194     public void printQueryAndObserved(Variable query, ArrayList<
            Variable> Obs) {
195         System.out.println("\nThe queried variable(s) is/are: ");
                // Printing
196         // the
197         // queried
198         // variables.
199         System.out.println(query.getName());
200         if (!Obs.isEmpty()) {
201             System.out.println("The observed variable(s) is/are: "
                    ); // Printing
202             // the
203             // observed
204             // variables.
205             for (Variable Ob : Obs) {
```

```
206                 System.out.println(Ob.getName());
207                 System.out.println("This variable has the value: "
                        + Ob.getObservedValue());
208             }
209         }
210     }
211 
212     /**
213      * Asks for a heuristic.
214      */
215     public void askForHeuristic() {
216         System.out.println("Supply a heuristic. Input 1 for least-
                incoming, 2 for fewest-factors and enter for random");
217         scan = new Scanner(System.in);
218         line = scan.nextLine();
219         if (line.isEmpty()) {
220             heuristic = "empty";
221             System.out.println("You have chosen for random");
222         } else if (line.equals("1")) {
223             heuristic = "least-incoming";
224             System.out.println("You have chosen for least-incoming
                    ");
225         } else if (line.equals("2")) {
226             heuristic = "fewest-factors";
227             System.out.println("You have chosen for fewest-factors
                    ");
228         } else {
229             System.out.println(line + " is not an option. Please
                    try again");
230             askForHeuristic();
231         }
232         scan.close();
233     }
234 
235     /**
236      * Getter of the observed variables.
237      *
238      * @return a list of observed variables given by the user.
239      */
240     public ArrayList<Variable> getObservedVariables() {
241         return obs;
242     }
243 
244     /**
245      * Getter of the queried variables.
246      *
```

```
247         * @return the variable the user wants to query.
248         */
249        public Variable getQueriedVariable() {
250            return query;
251        }
252
253        /**
254         * Getter of the heuristic.
255         *
256         * @return the name of the heuristic.
257         */
258        public String getHeuristic() {
259            return heuristic;
260        }
261
262        /**
263         * Prints the answer of the user query.
264         *
265         * @param finalTable a string of the table of the last factor
               after elimination
266         */
267        public void printQueryAnswer(String finalTable) {
268            System.out.println(finalTable);
269        }
270
271        /**
272         * Prints the reduced product formula based on the network
               structure
273         *
274         * @param vars a list of all variables
275         */
276        public void printProductFormula(ArrayList<Variable> vars) {
277            StringBuilder sb = new StringBuilder();
278            sb.append("\nThe reduced formula based on the network
                   structure:\n");
279            for (Variable var : vars) {
280                sb.append("P(").append(var.getName(), 0, 1);
281                if (var.isObserved()) {
282                    String value = var.getObservedValue();
283                    sb.append("=").append(value);
284                }
285                if (var.getNrOfParents() != 0) {
286                    sb.append("|");
287                    for (Variable parent : var.getParents()) {
288                        sb.append(parent.getName(), 0, 1);
289                        if (parent.isObserved()) {
```

```
290                         String value = parent.getObservedValue();
291                         sb.append("=").append(value);
292                     }
293                     if (var.getParents().indexOf(parent) != var.
                           getParents().size() - 1) {
294                         sb.append(",");
295                     }
296                 }
297             }
298             sb.append(")");
299             if (vars.indexOf(var) != vars.size() - 1) {
300                 sb.append("  ");
301             }
302         }
303         sb.append("\n");
304         System.out.println(sb.toString());
305     }
306 
307     /**
308      * Prints the formula of the reduced factors.
309      *
310      * @param fullyObserved reduced factor list
311      */
312     public void printFactorFormula(ArrayList<Factor> fullyObserved
           , boolean firstTime) {
313         StringBuilder sb = new StringBuilder();
314         if (firstTime) {
315             sb.append("The formula of the reduced factors:\n");
316         } else {
317             sb.append("Formula after the merge:\n");
318         }
319         addFactorsToStringBuilder(fullyObserved, sb, false);
320         System.out.println(sb.toString());
321     }
322 
323     /**
324      * Print the elimination order.
325      *
326      * @param elimOrder a list of variables in the elimination
            order
327      */
328     public void printEliminationOrder(PriorityQueue<Variable>
           elimOrder) {
329         StringBuilder sb = new StringBuilder();
330         sb.append("The elimination order, based on least incoming
              arcs:\n");
```

```
331         int order = 1;
332         while (!elimOrder.isEmpty()) {
333             Variable var = elimOrder.poll();
334             sb.append(order).append(". ").append(var.getName(), 0,
                    1).append("\n");
335             order++;
336         }
337         System.out.println(sb.toString());
338     }
339 
340     /**
341      * Prints the factors that are going to be merged.
342      *
343      * @param concerningFactors the list of factors to merge
344      * @param eliminate the variable to eliminate
345      */
346     public void printMergingFactors(ArrayList<Factor>
           concerningFactors, Variable eliminate) {
347         StringBuilder sb = new StringBuilder();
348         if (concerningFactors.size() > 1) {
349             sb.append("Merging the following factors by
                   eliminating ").append(eliminate.getName(), 0, 1).
                   append(" variable\n");
350             addFactorsToStringBuilder(concerningFactors, sb, true)
                   ;
351         } else {
352             sb.append("The following factor is safely ignored:\n")
                   ;
353             addFactorsToStringBuilder(concerningFactors, sb, true)
                   ;
354         }
355         System.out.println(sb.toString());
356     }
357 
358     /**
359      * Prints the last factors that are merged. Only used if the
           query variable is the ancestor of the observed variable(s)
           .
360      *
361      * @param factors a list of factors to be merged
362      */
363     public void printLastFactorMerge(ArrayList<Factor> factors) {
364         StringBuilder sb = new StringBuilder();
365         sb.append("Merging last remaining factors: \n");
366         addFactorsToStringBuilder(factors, sb, true);
367         System.out.println(sb.toString());
```

```
368     }
369
370     /**
371      * An auxiliary function to add factors list to the string
            builder.
372      *
373      * @param factors a list of factors
374      * @param sb string builder
375      * @param newLine a check if a new line is needed
376      */
377     private void addFactorsToStringBuilder(ArrayList<Factor>
            factors, StringBuilder sb, boolean newLine) {
378         for (int i = 0; i < factors.size(); i++) {
379             sb.append("f").append(i + 1).append("(");
380             for (Variable var : factors.get(i).getVariables()) {
381                 if (var.isObserved()) {
382                     sb.append(var.getName(), 0, 1).append("=").
                            append(var.getObservedValue());
383                 } else {
384                     sb.append(var.getName(), 0, 1);
385                 }
386
387                 if (factors.get(i).getVariables().indexOf(var) !=
                        factors.get(i).getVariables().size() - 1) {
388                     sb.append(",");
389                 }
390             }
391             sb.append(")");
392             if (i != factors.size() - 1 && !newLine) {
393                 sb.append("   ");
394             } else {
395                 sb.append("\n");
396             }
397         }
398     }
399 }
```

**Listing 9:** UserInterface.java

## 6.6 Class representing variable

```
1 package varelim;
2
3 import java.util.ArrayList;
4 import java.util.Objects;
5
6 /**
```

```
7  * Class to represent a variable.
8  *
9  * @author Marcel de Korte, Moira Berens, Djamari Oetringer,
       Abdullahi Ali, Leonieke van den Bulk
10 * @co-author/editor Mantas Makelis, David Leeftink
11 */
12 public class Variable {
13
14     private String name;
15     private ArrayList<String> possibleValues;
16     private ArrayList<Variable> parents; // Note that parents is
           not set in the constructor, but manually set with
17   // setParents() because of the .bif file layout
18     private String observedValue;
19     private boolean observed = false;
20     private int nrFactors;
21     /**
22      * Constructor of the class.
23      *
24      * @param name, name of the variable.
25      * @param possibleValues, the possible values of the variable.
26      */
27     public Variable(String name, ArrayList<String> possibleValues)
           {
28         this.name = name;
29         this.possibleValues = possibleValues;
30     }
31
32     public Variable() {}
33
34     /**
35      * Transform variable and its values to string.
36      */
37     public String toString() {
38         StringBuilder valuesString = new StringBuilder();
39         for (int i = 0; i < possibleValues.size() - 1; i++) {
40             valuesString.append(possibleValues.get(i)).append(", "
                   );
41         }
42         valuesString.append(possibleValues.get(possibleValues.size
               () - 1));
43         return name + " - " + valuesString;
44     }
45
46     /**
47      * Getter of the values.
```

```
48          *
49          * @return the values of the variable as a ArrayList of
                Strings.
50          */
51         public ArrayList<String> getValues() {
52             return possibleValues;
53         }
54 
55         /**
56          * Check if string v is a value of the variable.
57          *
58          * @return a boolean denoting if possibleValues contains
                string v.
59          */
60         public boolean isValueOf(String v) {
61             return possibleValues.contains(v);
62         }
63 
64         /**
65          * Getter of the amount of possible values of the variable.
66          *
67          * @return the amount of values as an int.
68          */
69         public int getNumberOfValues() {
70             return possibleValues.size();
71         }
72 
73         /**
74          * Getter of the name of the variable.
75          *
76          * @return the name as a String.
77          */
78         public String getName() {
79             return name;
80         }
81 
82         /**
83          * Getter of the parents of the variable.
84          *
85          * @return the list of parents as an ArrayList of Variables.
86          */
87         public ArrayList<Variable> getParents() {
88             return Objects.requireNonNullElseGet(parents, ArrayList::
                    new);
89         }
90 
```

```
91      /**
92       * Setter of the parents of the variable.
93       *
94       * @param parents the list of parents as an ArrayList of
             Variables.
95       */
96      public void setParents(ArrayList<Variable> parents) {
97          this.parents = parents;
98      }
99
100     /**
101      * Check if a variable has parents.
102      *
103      * @return a boolean denoting if the variable has parents.
104      */
105     public boolean hasParents() {
106         return parents != null;
107     }
108
109     /**
110      * Getter for the number of parents a variable has.
111      *
112      * @return the amount of parents as an int.
113      */
114     public int getNrOfParents() {
115         if (parents != null) {
116             return parents.size();
117         }
118         return 0;
119     }
120
121     /**
122      * Setter of the observed value of the variable.
123      *
124      * @param observedValue as a String to which observed value
             the current value of the variable should be set.
125      */
126     public void setObservedValue(String observedValue) {
127         this.observedValue = observedValue;
128     }
129
130     /**
131      * Getter of the observed value of the variable.
132      *
133      * @return the value of the variable as a String.
134      */
```

```
135     public String getObservedValue() {
136         return observedValue;
137     }
138 
139     /**
140      * Setter for if a variable is observed.
141      *
142      * @param observed a boolean denoting if the variable is
             observed or not.
143      */
144     public void setObserved(boolean observed) {
145         this.observed = observed;
146     }
147 
148     /**
149      * Getter for if a variable is observed.
150      *
151      * @return a boolean denoting if the variable is observed or
             not.
152      */
153     public boolean isObserved() {
154         return observed;
155     }
156 
157     @Override
158     public boolean equals(Object var) {
159         Variable variable = (Variable) var;
160         return name.equals(variable.getName());
161     }
162 
163     /**
164      * Get total number of appearances in factors
165      * @return number of appearances
166      */
167     public int getNrFactors(){
168         return nrFactors;
169     }
170 
171     /**
172      * Setter for total number of appearances in factors
173      * @param i new count
174      */
175     public void setNrFactors(int i){
176         nrFactors = i;
177     }
```

```
}
```

**Listing 10:** Variable.java

## 6.7 Class representing probability row

```
package varelim;

import java.util.ArrayList;

/**
 * Class to represent a row of a probability table by its values
    and a probability.
 *
 * @author Marcel de Korte, Moira Berens, Djamari Oetringer,
    Abdullahi Ali, Leonieke van den Bulk
 * @co-author/editor Mantas Makelis, David Leeftink
 */
public class ProbRow {

    private ArrayList<String> values;
    private double prob;

    /**
     * Constructor of the class.
     *
     * @param values, values of the variables (main variable+
        parents) in the row, of which the value of the main
     * variable itself is always first.
     * @param prob, probability belonging to this row of values.
     */
    public ProbRow(ArrayList<String> values, double prob) {
        this.prob = prob;
        this.values = values;
    }

    /**
     * Transform probabilities to string.
     */
    public String toString() {
        StringBuilder valuesString = new StringBuilder();
        for (int i = 0; i < values.size() - 1; i++) {
            valuesString.append(values.get(i)).append(", ");
        }
        valuesString.append(values.get(values.size() - 1));
        return valuesString + " | " + Double.toString(prob);
    }
```

```
39
40      /**
41       * Getter of the values of this probability row.
42       *
43       * @return ArrayList<String> of values
44       */
45      public ArrayList<String> getValues() {
46          return values;
47      }
48
49      /**
50       * Getter of the probability of this probability row
51       *
52       * @return the probability as a double.
53       */
54      public double getProb() {
55          return prob;
56      }
57
58      /**
59       * Setter of the probability
60       *
61       * @param prob the probability
62       */
63      public void setProb(double prob) {
64          this.prob = prob;
65      }
66  }
```

**Listing 11:** ProbRow.java

## 6.8 Class representing probability table

```
1   package varelim;
2
3   import java.util.ArrayList;
4
5   /**
6    * Class to represent a probability table consisting of
       probability rows.
7    *
8    * @author Marcel de Korte, Moira Berens, Djamari Oetringer,
       Abdullahi Ali, Leonieke van den Bulk
9    * @co-author/editor Mantas Makelis, David Leeftink
10   */
11
12  public class Table {
```

```
13
14      private Variable variable;
15      private ArrayList<ProbRow> table;
16
17      /**
18       * Constructor of the class.
19       *
20       * @param variable, variable belonging to the current
             probability table.
21       * @param table, table made out of probability rows (ProbRows)
             .
22       */
23      public Table(Variable variable, ArrayList<ProbRow> table) {
24          this.variable = variable;
25          this.table = table;
26      }
27
28      /**
29       * Returns the size of the Table (amount of probability rows).
30       *
31       * @return amount of rows in the table as an int.
32       */
33      public int size() {
34          return table.size();
35      }
36
37      /**
38       * Transform table to string.
39       */
40      public String toString() {
41          StringBuilder tableString = new StringBuilder(variable.
                getName() + " | ");
42          for (int i = 0; i < variable.getNrOfParents(); i++) {
43              tableString.append(variable.getParents().get(i).
                    getName());
44              if (!(i == variable.getNrOfParents() - 1)) {
45                  tableString.append(", ");
46              }
47          }
48          for (ProbRow row : table) {
49              tableString.append("\n").append(row.toString());
50          }
51          return tableString.toString();
52      }
53
54      /**
```

```
55          * Gets the i'th element from the ArrayList of ProbRows.
56          *
57          * @param i index as an int.
58          * @return i'th ProbRow in Table.
59          */
60         public ProbRow get(int i) {
61             return table.get(i);
62         }
63 
64         /**
65          * Getter of the table made out of ProbRows
66          *
67          * @return table as an ArrayList of ProbRows.
68          */
69         public ArrayList<ProbRow> getTable() {
70             return table;
71         }
72 
73         /**
74          * Getter of the variable that belongs to the probability
               table.
75          *
76          * @return the variable.
77          */
78         public Variable getVariable() {
79             return variable;
80         }
81 
82         /**
83          * Getter of the parents that belong to the node of the
               probability table.
84          *
85          * @return the parents as an ArrayList of Variables.
86          */
87         public ArrayList<Variable> getParents() {
88             return variable.getParents();
89         }
90 }
```

**Listing 12:** Table.java

## 6.9 Class representing network reader

```
1  package varelim;
2 
3  import java.io.BufferedReader;
4  import java.io.FileNotFoundException;
```

```
5  import java.io.FileReader;
6  import java.io.IOException;
7  import java.util.ArrayList;
8  import java.util.Arrays;
9  import java.util.Collections;
10
11 /**
12  * Class that reads in a network from a .bif file and puts the
        variables and probabilities at the right places.
13  *
14  * @author Marcel de Korte, Moira Berens, Djamari Oetringer,
        Abdullahi Ali, Leonieke van den Bulk
15  * @co-author/editor Mantas Makelis, David Leeftink
16  */
17
18 public class Networkreader {
19
20     private ArrayList<Variable> vs = new ArrayList<>();
21     private ArrayList<Table> ps = new ArrayList<>();
22     private ArrayList<ProbRow> probRows;
23     private String probName;
24     private ArrayList<Variable> parents = new ArrayList<>();
25     private int nrOfRows;
26
27     /**
28      * Constructor reads in the data file and adds the variables
           and its
29      * probabilities to the designated arrayLists.
30      *
31      * @param file, the name of the .bif file that contains the
           network.
32      */
33     public Networkreader(String file) {
34         BufferedReader br;
35         try {
36             String cur; // Keeping track of current line observed
                by BufferedReader
37             br = new BufferedReader(new FileReader(file));
38             try {
39                 while ((cur = br.readLine()) != null) {
40                     if (cur.contains("variable")) {
41                         //Add variable to the list
42                         String varName = cur.substring(9, cur.
                            length() - 2);
43                         cur = br.readLine();
44                         ArrayList<String> possibleValues =
```

```
                        searchForValues(cur);
                    vs.add(new Variable(varName,
                        possibleValues));
                }
                if (cur.contains("{")) {
                    parents = new ArrayList<>();
                }
                if (cur.contains("probability")) {
                    // Conditional to check for parents of
                        selected variable
                    searchForParents(cur);
                }
                if (cur.contains("table")) {
                    //Conditional to find probabilities of 1
                        row and add Probabilities to
                    // probability list
                    ArrayList<ProbRow> currentProbRows =
                        searchForProbs(cur);
                    probRows.addAll(currentProbRows);
                    Table table = new Table(getByName(probName
                        ), probRows);
                    ps.add(table);
                }
                if (cur.contains(")") && cur.contains("(") &&
                    !cur.contains("prob")) {
                    // Conditional to find probabilities of
                        more than 1 row;
                    // add probabilities to probability list
                    ArrayList<ProbRow> currentProbRows =
                        searchForProbs(cur);
                    probRows.addAll(currentProbRows);
                    if (probRows.size() == nrOfRows) {
                        Table table = new Table(getByName(
                            probName), probRows);
                        ps.add(table);
                    }
                }
            }
        } catch (IOException ignored) {}
    } catch (FileNotFoundException e) {
        System.out.println("This file does not exist.");
        System.exit(0);
    }
}

/**
```

```
81      * Searches for a row of probabilities in a string
82      *
83      * @param s a string s
84      * @return a ProbRow
85      */
86     public ArrayList<ProbRow> searchForProbs(String s) {
87         ArrayList<ProbRow> currentProbRows = new ArrayList<>();
88         int beginIndex = s.indexOf(')') + 2;
89         if (s.contains("table")) {
90             beginIndex = s.indexOf('e') + 2;
91         }
92 
93         int endIndex = s.length() - 1;
94         String subString = s.substring(beginIndex, endIndex);
95         String[] probsString = subString.split(", ");
96         double[] probs = new double[probsString.length];
97         for (int i = 0; i < probsString.length; i++) {
98             probs[i] = Double.parseDouble(probsString[i]);
99         }
100 
101         if (!s.contains("table")) {
102             ArrayList<String> parentsValues = new ArrayList<>();
103             ArrayList<String> nodeValues = new ArrayList<>();
104             beginIndex = s.indexOf('(') + 1;
105             endIndex = s.indexOf(')');
106             subString = s.substring(beginIndex, endIndex);
107             String[] stringValues = subString.split(", ");
108             Collections.addAll(parentsValues, stringValues);
109             for (Variable v : vs) {
110                 if (probName.equals(v.getName())) {
111                     nodeValues = v.getValues();
112                 }
113             }
114             for (int i = 0; i < probs.length; i++) {
115                 parentsValues.add(0, nodeValues.get(i));
116                 ArrayList<String> currentVal = new ArrayList<>(
                        parentsValues);
117                 currentProbRows.add(new ProbRow(currentVal, probs[
                        i]));
118                 parentsValues.remove(0);
119             }
120         } else {
121             ArrayList<String> values = new ArrayList<>();
122             ArrayList<String> nodeValues = new ArrayList<>();
123             for (Variable v : vs) {
124                 if (probName.equals(v.getName())) {
```

```
125                     values = v.getValues();
126                 }
127             }
128             for (int i = 0; i < probs.length; i++) {
129                 nodeValues.add(values.get(i));
130                 ArrayList<String> currentVal = new ArrayList<>(
                        nodeValues);
131                 ProbRow prob = new ProbRow(currentVal, probs[i]);
132                 currentProbRows.add(prob);
133                 nodeValues.clear();
134             }
135         }
136         return currentProbRows;
137     }
138 
139     /**
140      * Searches for values in a string
141      *
142      * @param s a string s
143      * @return a list of values
144      */
145     public ArrayList<String> searchForValues(String s) {
146         int beginIndex = s.indexOf('{') + 2;
147         int endIndex = s.length() - 3;
148         String subString = s.substring(beginIndex, endIndex);
149         String[] valueArray = subString.split(", ");
150         return new ArrayList<>(Arrays.asList(valueArray));
151     }
152 
153     /**
154      * Method to check parents of chosen variable.
155      *
156      * @param cur, which gives the current line.
157      */
158     public void searchForParents(String cur) {
159         if (cur.contains("|")) { // Variable has parents
160             extractParents(cur);
161         } else { // Variable has no parents
162             probName = cur.substring(14, cur.length() - 4);
163             for (Variable v : vs) {
164                 if (probName.equals(v.getName())) {
165                     nrOfRows = v.getNumberOfValues();
166                 }
167             }
168         }
169         probRows = new ArrayList<>();
```

```
170     }
171
172     /**
173      * Gets a variable from variable Vs when the name is given
174      *
175      * @return variable with name as name
176      */
177     private Variable getByName(String name) {
178         Variable var = null;
179         for (Variable v : vs) {
180             if (v.getName().equals(name)) {
181                 var = v;
182             }
183         }
184         return var;
185     }
186
187     /**
188      * Extracts parents and puts them in a list of parents of that
             node.
189      *
190      * @param cur, a string to extract from
191      */
192     public void extractParents(String cur) {
193         probName = cur.substring(14, cur.indexOf("|") - 1);
194         Variable var = getByName(probName);
195         String sub = cur.substring(cur.indexOf('|') + 2, cur.
               indexOf(')') - 1);
196         while (sub.contains(",")) { // Variable has more parents
197             String current = sub.substring(0, sub.indexOf(','));
198             sub = sub.substring(sub.indexOf(',') + 2);
199             for (Variable v : vs) {
200                 if (v.getName().equals(current)) {
201                     parents.add(v); // Add parent to list
202                 }
203             }
204         }
205         if (!sub.contains(",")) { // Variable has no more parents
206             for (Variable v : vs) {
207                 if (v.getName().equals(sub)) {
208                     parents.add(v); //
209                 }
210             }
211         }
212
213         var.setParents(parents);
```

```
214             nrOfRows = computeNrOfRows(probName);
215         }
216 
217         /**
218          * Computes the number of rows needed given the current
                parents
219          *
220          * @return the number of rows
221          */
222         private int computeNrOfRows(String probName) {
223             int fac = 1;
224             for (Variable parent : parents) {
225                 fac = fac * parent.getNumberOfValues();
226             }
227             for (Variable v : vs) {
228                 if (probName.equals(v.getName())) {
229                     fac = fac * v.getNumberOfValues();
230                 }
231             }
232             return fac;
233         }
234 
235         /**
236          * Getter of the variables in the network.
237          *
238          * @return the list of variables in the network.
239          */
240         public ArrayList<Variable> getVs() {
241             return vs;
242         }
243 
244         /**
245          * Getter of the probabilities in the network.
246          *
247          * @return the list of probabilities in the network.
248          */
249         public ArrayList<Table> getPs() {
250             return ps;
251         }
252 
253 }
```

**Listing 13:** Networkreader.java

## 6.10 Probabilities of the Bike Repair problem

probability ( Weather ) table 0.5, 0.25, 0.25; probability ( ConditionBike | Weather ) (Rainy) 0.15, 0.7, 0.15; (Sunny) 0.40, 0.30, 0.30; (Snow) 0.4, 0.2, 0.4; probability ( ParkingLo-

cation | Weather ) (Rainy) 0.90, 0.1; (Sunny) 0.5, 0.5; (Snow) 0.6, 0.4; probability ( Repair | ParkingLocation , ConditionBike) (Inside, Good) 0.02, 0.98; (Inside, Medium) 0.5, 0.5; (Inside, Broken) 0.99, 0.01; (Outside, Good) 0.8, 0.2; (Outside, Medium) 0.4, 0.6; (Outside, Broken) 0.2, 0.8;