CS5011 - Practical 3

[190026921]

Mar 30, 2022

Word Count: 2146

Contents

1 Introduction and Overview

The objective of this practical was to implement a representation of BNs and model a concrete system that represents the risk of security attacks in a company.

1.1 Implementation status

All requirements were implemented, including an extension.

- **P1:** Attempted & fully working (A/FW)
- **P2:** A/FW
- **P3:** A/FW
- **P4:** A/FW (Greedy Min Edges, Max Cardinality)
- **P5:** A/FW; browser based expert system

1.2 Compiling and running instructions

1.2.1 Basic usage

As per the specification, the program can be compiled and run from the `src` directory as follows (for parts 1 to 4):

```
javac *.java
java A3Main <Pn> <NID> [<order-algo>]
```

where

Parameter	Meaning	Options	Available for
<Pn>	Part	P1 | P2 | P3 | P4	
<NID>	Network ID	BNA | BNB | BNC | CNX	P1 - P4
<order-algo>	Ordering algorithm	GME | MC	P4

1.2.2 Instructions for extension, testing, and evaluation

Running the extension starts a simple HTTP server (`ExpertSystemServer.java`) that serves the expert system website on `http://localhost:9876`. Since the server uses a dependency (GSON), I had to put it into a separate directory (`expertsystem`) to allow stacscheck to run as required.

To run the server, run:

```
# from the project root (the directory containing ‘src‘)
make compile
make serve
```

This ensures that the dependencies are included in the classpath.

The same applies when running the JUnit tests or the evaluation.

All Make commands must be run from the project root and `make compile` must be run first to compile the project.

The following commands are available:

```
compile      Compile all java files
clean        Remove java class files
test         Run JUnit tests
stacscheck   Run stacscheck checks (only on school server)
evaluation   Run the evaluation (run experiments, create plots)
serve        Run the server for the expert system
help         Print available commands
```

2 Design and Implementation

This section details the implementation details of the practical.

2.1 Part 1: Bayesian Network

2.1.1 Assumptions about the CNX network

For the alert system, I have interpreted the specification as follows. Values that I have chosen in addition to the specification are marked with an asterisk (*).

Firewall and Maintenance

There is a 10% (*) chance of the system being in maintenance at any given point in time. In terms of days, this would mean that maintenance occurs on average every 10 days.

Information about the current maintenance can be outdated, which happens with a 2% chance.

If the information is outdated, the firewall will be switched off in 50% of cases (this is my interpretation of “vulnerabilities in the CNX”).

If the information is up to date, maintenance will lead to the firewall being switched off in 3% of cases. Up-to-date information and no maintenance leads to the firewall being inactive in 1% (*) of cases due to other errors occurring in the system. I chose a relatively low value of 1% since most commercial firewalls offer reliability in that range (or actually even better).

Websites

It is assumed that 5% (*) of websites accessed by the clients are risky. The website blocker will block 85% of unsafe websites and 5% (*) of safe websites (since some false positives will occur, which is the case with most cybersecurity systems).

Denial of Service

As per the specification, 45 of 360 days are holidays, i.e., there is a 12.5% chance of a day being a holiday.

The risk on holidays is higher, i.e. there is a 15% (*) chance of a DDoS attack. On work days, the risk is low with 5% (*) chance of an attack. This value was chosen arbitrarily but seems reasonable for large firms that will come under attack every few weeks.

Logging system

The logger tracks client activity, 95% (*) of which is assumed to be legitimate. If the client activity is risky, the logger notes the incident 70% of the time. If the client activity is not risky, the logger falsely notes the incident 30% of the time.

Alert system

The alert system receives input from the website blocker, the DDoS checker, the firewall, and the logger. Each input is weighted equally (25% (*) each).

When receiving input from the subsystems, the incoming truth-values are weighted and multiplied by 95%. For example, if all four systems suggest a threat, an alert will be triggered in 95% of cases. If two of four systems suggest a threat, the alert will be triggered in (2 * 25% * 95%) = 47.5% of cases.

2.1.2 Network graph

I implemented the network using the `Bayes` tool from `aispaces.org` to compare the results of my program to those of the tool [AIspaces.org, 2016]. Figure 1 depicts the network. The CPT of each node can be inspected by loading the XML file (`doc/networks/CNX.xml`) into the tool.

2.1.3 Implementation

Figure 2 depicts the core data structures of the BN implementation and their classes.

A BN contains a list of nodes, with each node containing references to its parent and children nodes. This allows the recursive exploration of all its ancestors and descendants (done in the `Node.getAllAncestors` method using DFS).

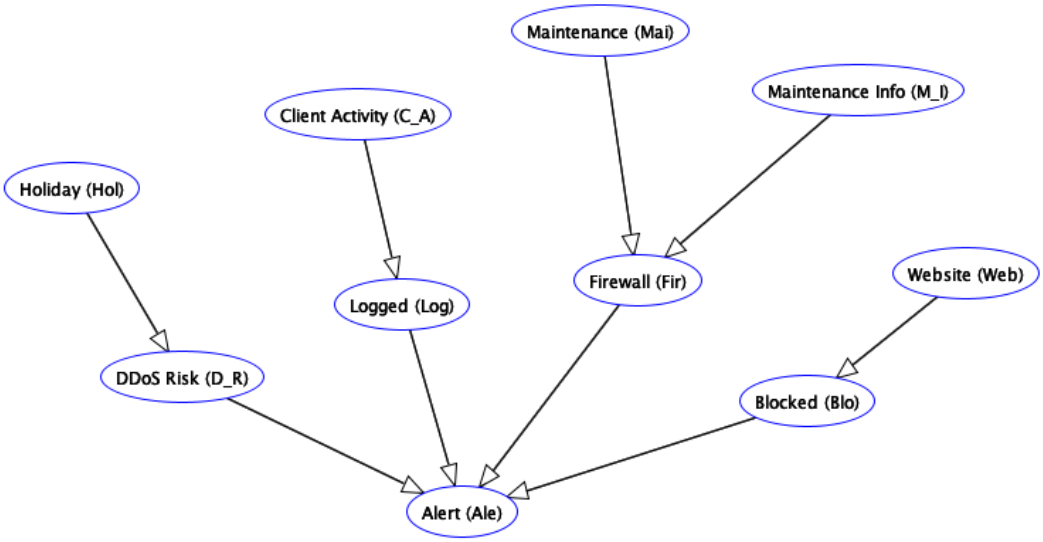

Figure 1: Bayesian network for the CNX system.

The nodes further have an instance of `Factor` attached, which serves as the conditional probability table. Each column (i.e. each node in the factor) is represented as a `FactorColumn` instance.

When an edge is added from node x to node y, the factor of y automatically updates by including a factor column representing x. This way, for each domain values of x, a new combination with the existing rows of the CPT is created.

The factor further contains an auxiliary field `rowIndex`, which maps from a `FactorRowKey` to the associated row. The factor row key uniquely identifies each row in the CPT by the domain values of the variables in the row. This auxiliary data structure allows for quicker computations during some of the other algorithms of the project as it allows O(1) access to each row (of course, at the cost of an extra field to keep updated).

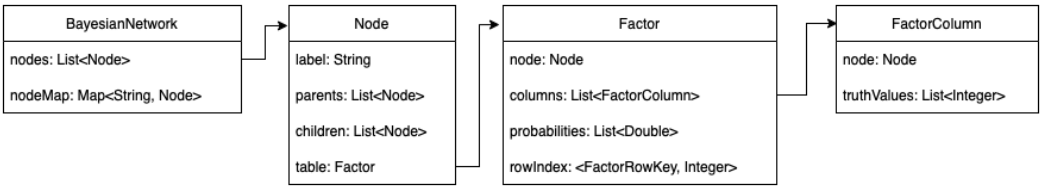

Figure 2: Simplified class diagram of the Bayesian Network and its dependencies.

2.2 Part 2: Querying

To load the CNX network programmatically, an abstract factory `BayesianNetworkFactory` has the networks BNA, BNB, BNC, and CNX as presets.

The `Agent` class contains the logic to answer queries without evidence and the implementation follows lecture slides. An interesting detail is the `Agent.join` method, which calculates the point-wise product of the factors that are summed out. To calculate the point-wise product, the algorithm first creates an empty new factor for the result, and adds the nodes to be contained in the result. For all resulting rows, it then generates the product of the probabilities associated with the corresponding rows in the left factor and the right factor. For this, the factor row keys described in Part 1 are used to efficiently access the rows of the left and right factor in constant time. Since the my algorithm differs slightly from that described in the lectures, it is provided in Algorithm 1.

Algorithm 1 Procedure of the `Agent.pointWiseProduct` method

```
procedure AGENT.POINTWISEPRODUCT(FACTOR f1, FACTOR f2)

    leftSideNodes = f1.getNodeSet()
    rightSideNodes = f2.getNodeSet()
    newFactor = new Factor()
    newFactor.addNodes(leftSideNodes, rightSideNodes)
    newProbabilities = new List()                    ▷ list to hold new probabilities

    for all i in 1...newFactor.numRows() do
        row = newFactor.getRow(i)
        leftKey = getKey(f1, row)                    ▷ key to identify corresponding row in f1
        rightKey = getKey(f2, row)                   ▷ key to identify corresponding row in f2

        p = f1.getProbability(leftKey) * f1.getProbability(rightKey)
        newProbabilities.add(p)
    end for

    newFactor.setProbabilities(newProbabilities)

    return newFactor
end procedure
```

During the marginalisation step, I first generate the truth tables for the resulting factor. This can be thought of as a group-by operation, during which each unique row (excluding the variable to be summed out) of the input factor is assigned a new probability. The probability is given by an aggregation function that sums up the probabilities of each group.

These new probabilities are then assigned to a new factor. The group-by step of the algorithm (`Agent.groupFactorByVariable`) is described in Algorithm 2.

Algorithm 2 Procedure of the `Agent.groupFactorByVariable` method

```
procedure AGENT.GROUPFACTORBYVARIABLE(FACTOR f, STRING y)

    newProbabilities = new Map()          ▷ Map from factor row key to new probability
    explored = new Set()                  ▷ Set of explored rows to prevent duplicated search

    for all i in 1...f.numRows() do
        factorKey = f.getKeyForRow(i) without node y
        probabilitySum = 0

        for all j in 1...f.numRows() if not j in explored do
            if row j matches factorKey then
                probabilitySum += f.getProbabilityForRow(j)      ▷ aggregate probabilities
                explored.add(j)                                  ▷ no need to explore this row again
            end if
        end for

        newProbabilities.put(factorKey, probabilitySum)

    end for

    return newProbabilities
end procedure
```

2.3 Part 3: Adding Evidence

In part 3, evidence may be added to the queries. This is implemented in the `AgentWithEvidence` class, which extends `Agent`. In particular, it overrides the `getResult` method to add the steps necessary to consider evidence.

The additional steps are `projectEvidence` and `joinNormalize`. The `projectEvidence` method is straightforward. For each provided evidence, each factor that contains the node of the evidence has its probabilities set to 0 in all rows that contradict the evidence.

The `joinNormalize` method is more evolved. The algorithm takes as input the summed out factors with the projected evidence. The first of these factors is copied and is the starting point of the procedure. A set of factor row keys is derived for all rows of the starting factor. Using these keys, all remaining factors are then joined: the probability of the left factor (in the row of the factor row key) is multiplied by that of the corresponding row in the right factor. This is done for all factors until a single factor remains. Lastly, this factor is normalized by summing all its probabilities and then dividing each value by this sum. Algorithm 3 gives an overview of this.

Algorithm 3 Procedure of the `AgentWithEvidence.joinNormalize` method

```
procedure AgentWithEvidence.joinNormalize(List[Factor] factors, Node y)

    newFactor = factors.get(0)
    keys = List[FactorRowKey]

    for all row in newFactor do
        keys.add(new FactorRowKey(row))              ▷ Unique identifier for the row
    end for

    for all rightFactor in factors without first factor do          ▷ Join other factors
        for all key in keys do
            p = newFactor.getRow(key) * rightFactor.getRow(key)
            newFactor.setProbabilityForRow(key, p)                  ▷ Set new probability
        end for
    end for

    probSum = sum(newFactor.getProbabilities())          ▷ Denominator of normalization

    for all row in newFactor do                                          ▷ Normalize
        p = newFactor.getProbability(row)
        newFactor.setProbability(row, p / probSum)
    end for

    return newFactor
end procedure
```

2.4 Part 4: Order Algorithms

For Part 4, the program calculates an elimination order automatically. I implemented the Greedy Min Edges (GME) and Maximum Cardinality (MC) algorithms. Both use an induced graph, which I implemented as an adjacency list in the `InducedGraph` class.

Since the implementation follows the lecture slides closely, the pseudo-code is omitted. The difference between the order algorithms is further discussed in Section 4.

2.5 Part 5: Expert System

As an extension, I implemented an expert system that allows the user to query the CNX network using a simple website. The code for the extension is included in the `expertsystem` directory. Refer to Section 1 for running instructions.

The website is hosted by a simple HTTP server, serving `expertsystem/www/index.html`. The logic of the website itself is implemented in Vue.js and any imputs from the user are sent to the server, which runs the query and returns a JSON response. The response contains the result of the query and the runtime.

The website supports both diagnostic and predictive queries of the CNX network. A screenshot of the system if provided in Figure 3. The user may also specify a custom order by dragging the nodes into the desired order (see Figure 4).

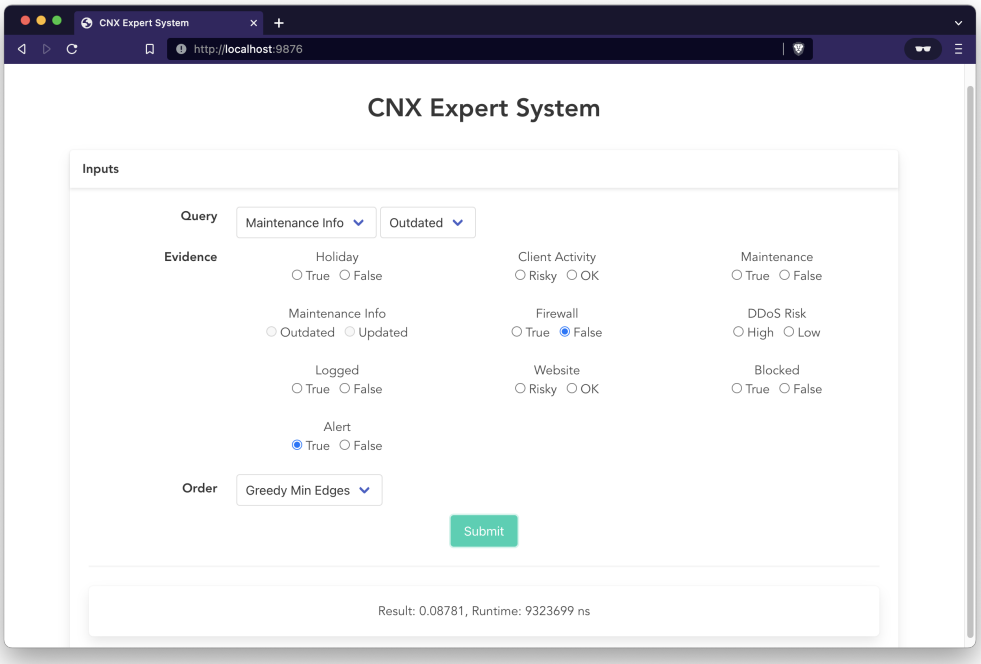

Figure 3: Screenshot of running a query on the expert system.

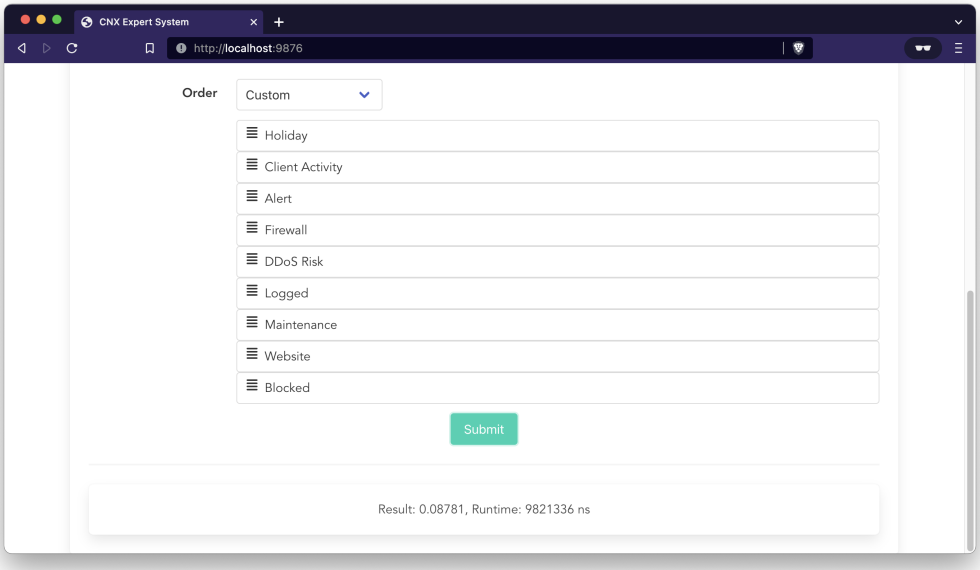

Figure 4: Specifying a custom order on the expert system.

2.6 Discussion and example queries

This section provides a discussion of some example queries.

Part 2

The following query asks for the probability of the firewall being active. The system returns 0.88612 as a result. This suggests that (given no other information) there is an 88.612% probability of the firewall being active at the current point in time.

```
# Example query 1
java A3main P2 CNX
Query:
Fir:T
Order:
Hol,D_R,C_A,Log,Mai,M_I,Web,Blo,Ale
0.88612
```

The second query asks for the probability of an alert occurring in the current time period. The resulting probability of 0.32267 indicates that an alert occurs with a 32.267% chance.

```
# Example query 2
```

```
java A3main P2 CNX
Query:
Ale:T
Order:
Hol,D_R,C_A,Log,Mai,M_I,Web,Blo,Fir
0.32267
```

Part 3

For part 3, evidence may be provided. The following predictive query predicts the chance of an alert occurring given that it is a holiday (Hol:T), the system is under maintenance (Mai:T), and a website has been blocked (Blo:T). Given this evidence, the program returns a probability of 35.848%, which is higher than the baseline without evidence (seen in query 2).

```
# Example query 3
java A3main P3 CNX
Query:
Ale:T
Order:
Hol,D_R,C_A,Log,Mai,M_I,Web,Blo,Fir
Evidence:
Hol:T Mai:T Blo:T
0.35848
```

The following diagnostic query calculates the probability that given an alert occurrs, the maintenance information is outdated. The output suggests that in the case of an alert, there is a 1.432% chance of the maintenance information being outdated.

```
# Example query 4
java A3main P3 CNX
Query:
M_I:T
Order:
Hol,D_R,C_A,Log,Mai,Web,Blo,Fir,Ale
Evidence:
Ale:T
0.01432
```

Part 4

The following two orders run the same query as in example query 3, but with an order algorithm.

```
# Example query 5 with Greedy Min Edges order
```

```
java A3main P4 CNX GME
Query:
Ale:T
Evidence:
Hol:T Mai:T Blo:T
Order: Hol, C_A, Web, Mai, M_I, D_R, Log, Blo, Fir
0.35848

# Example query 6 with Max Cardinality order
java A3main P4 CNX MC
Query:
Ale:T
Evidence:
Hol:T Mai:T Blo:T
Order: Web, M_I, Mai, C_A, Hol, Blo, Fir, Log, D_R
0.35848
```

Unsurprisingly, the order does not affect the predicted probability. However, providing a different order leads to factors being combined differently. The larger the factors, the higher the space requirements. A detailed evaluation of this is provided in Section 4.

3 Test Summary

I performed extensive unit testing using JUnit to ensure the correctness of my code. In total, I wrote 81 unit tests that focus on testing the critical parts of my code. Some of the tests actually would be better described as integration test, as they run the entire program on a provided setting. These integration tests include the settings from stacscheck, which I converted into JUnit test. In addition, I added several edge case scenarios (e.g. queries that are neither diagnostic nor predictive) to uncover problems in the code that do not occur in the stacscheck scenarios.

Overall, my unit tests yield a tests coverage of 69.2%, however, this includes files such as `A3Main.java`, which were not relevant for testing. In classes with critical code, I aimed for a test coverage of $> 90\%$. A HTML coverage report is included in the `doc/coverage` directory.

Further, I set up an automated test pipeline using GitHub Actions, which automatically runs all my unit test when I push changes to the remote repository, hosted privately on GitHub. This helped ensure that any changes I made did not interfere with code I had written previously and helped uncover issues with the code quickly.

Beyond automated testing, I ran a large number of manual tests, where I compared the output of my program to that generated by the `Bayes` tool from `aispaces.org`. Using the 'verbose' querying in the `Bayes` tool allowed me to track and compare each step of the algorithms.

4 Evaluation and Conclusions

4.1 Evaluation

To gain further insight into the algorithms used, this section evaluates both predictive and diagnostic queries in terms of their runtime and memory usage. The two ordering algorithms GME and MC are evaluated against each other.

For the evaluation, I generated 200 random queries for each network (100 predictive, 100 diagnostic). Figures 6 and 5 depict the results for the runtime for diagnostic and predictive queries respectively.

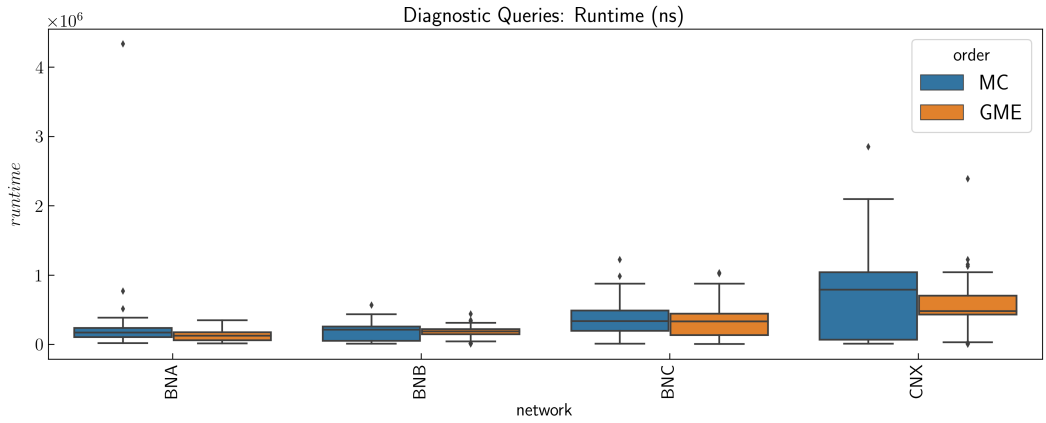

Figure 5: Runtime of diagnostic queries.

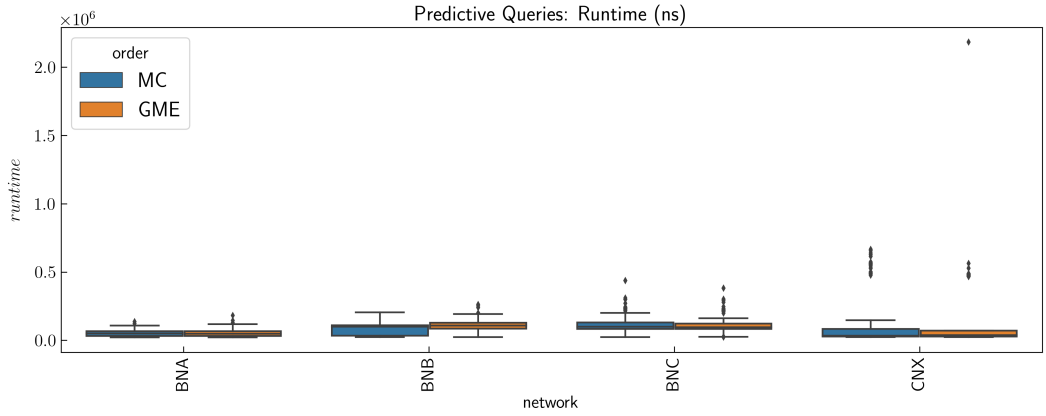

Figure 6: Runtime of predictive queries.

From the experiments, it appears that GME and MC ordering tend to produce very similar runtimes. For diagnostic queries, the figures suggest that the variability of the runtime is larger for MC than for GME (see for example Figure 5, CNX network).

This could mean that either the algorithms produce similar factor sizes and thus do not differ in terms of memory usage, which makes them similar in terms of runtime.

On the other hand, consider Figure 7, which shows the results of running the Java Async profiler with the experiments and tracking the memory usage per method in bytes.

Diagnostic, MC

Method	Allocation size...	Own Allocation siz...
ExperimentReaderWriter.main(String[])	60,285,920	0
ExperimentReaderWriter.runExperiments(ArrayList)	50,890,704	0
JsonConf.runConfiguration(JsonConf, BayesianNetwork)	42,439,288	0
AgentWithEvidence.getResult(Node, Order, int, ArrayList)	40,892,232	0
Agent.combineFactors(Order, List)	34,537,944	0

Diagnostic, GME

Method	Allocation size...	Own Allocation siz...
ExperimentReaderWriter.main(String[])	71,305,216	0
ExperimentReaderWriter.runExperiments(ArrayList)	61,901,592	0
JsonConf.runConfiguration(JsonConf, BayesianNetwork)	53,043,488	0
AgentWithEvidence.getResult(Node, Order, int, ArrayList)	49,941,704	0
Agent.joinMarginalize(List, String)	41,068,576	0

Predictive, MC

Method	Allocation size...	Own Allocation siz...
ExperimentReaderWriter.main(String[])	48,758,784	0
ExperimentReaderWriter.runExperiments(ArrayList)	39,864,784	0
JsonConf.runConfiguration(JsonConf, BayesianNetwork)	28,261,104	0
AgentWithEvidence.getResult(Node, Order, int, ArrayList)	24,574,224	0
Agent.combineFactors(Order, List)	18,819,728	0

Predictive, GME

Method	Allocation size...	Own Allocation siz...
ExperimentReaderWriter.main(String[])	51,902,464	0
ExperimentReaderWriter.runExperiments(ArrayList)	42,507,168	0
JsonConf.runConfiguration(JsonConf, BayesianNetwork)	32,889,792	0
AgentWithEvidence.getResult(Node, Order, int, ArrayList)	29,746,128	0

Figure 7: Output of the Java Async profiler for memory allocation per method in bytes. The relevant method `getResult` is highlighted in pink.

The output suggests that for diagnostic queries, the GME algorithm uses approximately (22%) more memory than the MC algorithm (for the `getResult` method). For predictive queries, the difference is similar with 21%.

This implies that in our case, MC appears to provide a lower memory footprint than GME at the same runtime. Note that this is dependent on the networks at hand. The difference in memory footprint suggests that GME tends to produce larger factors (for our networks), causeing it to use more memory.

4.2 Concluding remarks

The implemented program allows the user to run any query on the provided networks. Evidence may be provided to run both diagnostic and predictive queries. Using the ordering algorithms, one can potentially improve the performance of the agent, however, the evaluation showed that while this manifests in memory usage, it does not affect runtime.

References

AIspaces.org. Belief and decision networks, version 5.1.10, 2016. URL http://aispace.org/bayes/index.shtml.