Project no.:	**619209**
Project full title:	**Analysis of Massive Data STreams**
Project Acronym:	**AMIDST**
Deliverable no.:	**D5.2**
Title of the deliverable:	**Summary of software developments made covering the first 18 months and planning for the remaining part**

Contractual Date of Delivery to the CEC:	**30.06.2015**
Actual Date of Delivery to the CEC:	**30.06.2015**
Organisation name of lead author for this deliverable:	**HUGIN**
Author(s):	**Anders L Madsen, Martin Karlsen, Nicolaj Søndberg-Jeppesen, Frank Jensen, Helge Langseth, Antonio Salmeron, Thomas Nielsen**
Participants(s):	**P03, P01, P02, P04**
Work package contributing to the deliverable:	**WP5**
Nature:	**R**
Version:	**1.0**
Total number of pages:	**39**
Start date of project:	1st January 2014 **Duration:** 36 months

Project co-funded by the European Commission within the Seventh Framework Programme (2007-2013)		
Dissemination Level		
PU	Public	X
PP	Restricted to other programme participants (including the Commission Services)	
RE	Restricted to a group specified by the consortium (including the Commission Services)	
CO	Confidential, only for members of the consortium (including the Commission Services)	

Abstract:

In this document, we give a summary of the software developments performed in the four tasks of Work package 5. In particular, we describe the developed functionality on learning the structure of both restricted structure models and Bayesian networks from data in parallel. This is Task 5.1. The developments made in this task are available in the open source AMIDST toolbox through its interface to other software packages. For Task 5.2, we give a summary of the software developments made to optimize the real time performance of the HUGIN Decision Engine to meet mainly the requirements of the use-case considered in Work package 6 (the automotive use-case). The developments in Task 5.3 have focused on supporting approximate inference in dynamic Bayesian networks as required by Work package 6. Finally, limited work has been performed in Task 5.4 on adaptation of parameters in dynamic Bayesian networks as this task has just started recently. All four tasks are still running and work remains to be performed in all four tasks.

Keyword list: Software development, Parallelization, optimization, dynamic Bayesian networks.

Table of Contents

Document History

Version	Date	Author (Unit)	Description
v0.3	03/06 2015	Anders L. Madsen, Martin Karlsen, Nicolaj Søndberg-Jeppesen, Frank Jensen	First draft finished
v0.6	22/06 2015	Anders L. Madsen, Martin Karlsen, Nicolaj Søndberg-Jeppesen, Frank Jensen	Version submitted to PSRG
v0.9	25/06 2015	Anders L. Madsen, Martin Karlsen, Nicolaj Søndberg-Jeppesen, Frank Jensen	Revised Version submitted to PSRG
v1.0	30/06 2015	Anders L. Madsen, Martin Karlsen, Nicolaj Søndberg-Jeppesen, Frank Jensen, Helge Langseth, Antonio Salmeron, Thomas Nielsen	Final Version of the document

Executive summary

This document gives a summary of the software developments made in Work package 5 covering the first 18 months of the AMIDST project and outlines the planning for the remaining part of the AMIDST project period.

1 Introduction

According to the Description of Work, the objectives of Work package 5 are:

- *To develop specific elements of the AMIDST framework in the existing commercial software tool HUGIN to support the analysis of data from the use-cases. The work is focused on implementing both new and revised algorithms and methods in the HUGIN tool in order to better support scalable analysis of data and streaming data as defined by the use-cases as well as to scale up existing implementations for data analysis.*
- *The work includes both software developments, extensive testing of implementations, and methodological developments in dialog with academic researchers and use-case experts.*
- *The developments will be made based on the use-cases with the aim of being applicable in general to support the analysis of other data sets.*
- *Robustness, scalability and reactiveness are important considerations, if the solutions are to be effective.*

In short, this work package will consider the development of extensions of the HUGIN tool to support the use-case requirements for data analysis and deployment of solutions, if effective. The developments will mainly support WP6 and WP8.

Work package 5 consists of four main tasks Task 5.1 to Task 5.4. According to the Gantt diagram in the Description of Work, Task 5.1 was scheduled to start from month 1, Task 5.2 and Task 5.3 were scheduled to start from month 3 while Task 5.4 was scheduled to start from month 15. Significant progress on the software developments has, as reported in the following pages, been made for Task 5.1 to 5.3 whereas no progress on software development has been made for Task 5.4.

The software developments performed so far in the tasks of Work package 5 have been guided by the Description of Work and the use-case requirements as identified during the requirements analysis. Table A.1 in Appendix A lists the 29 use-case requirements identified in the requirements analysis in Task 1.3 to 1.5 relevant for Work package 5. The use-case requirements listed are associated with the automotive data considered in Work package 6.

The following four chapters of this deliverable provides a summary of the software developments made as part of each task. In addition, each chapter includes an outline of the planning for the remaining part of the AMIDST project period.

2 Task 5.1: Parallelization of Learning BN Structure From Data

This section summaries the software developments on learning Bayesian network structure from data made in the HUGIN tool using parallelization.

This work has produced two scientific articles describing the approach, see [16] for details on the method for learning a TAN model from data in parallel using processes and [17] for details on the method for learning Bayesian network structure in parallel using threads.

The algorithms were implemented as extensions of HUGIN AMIDST which is based on the official HUGIN software version 8.1 (current version of the software at the time of writing) [14, 15].

The parallelisation of structure learning mainly relates to the BCC Bank use-case in Work package 8.

2.1 Learning Restricted Structure Models

This section provides a short summary of the software development efforts made in Task 5.1 on learning restricted structure models (i.e., tree-augmented Naive Bayes model) from data in parallel. The software developments are based on the theoretical work described in [16]. In addition, Deliverable 5.1 [3], which was a software prototype on learning structure in parallel, includes a description of the algorithms implemented. The parallel learning of restricted structure models is based on a *Message Passing Interface* (MPI) [9] and uses balanced incomplete block designs to control the parallelization [19] and supports learning TAN models with Conditional Linear Gaussian distributions [13].

Let $\mathcal{D} = (c_1, \ldots, c_N)$ denote a data set of N complete cases over variables $\mathcal{X} = \{C\} \cup \mathcal{F}$ where C is the classification variable and $\mathcal{F}$ is a set of n features. The task of constructing a TAN model over $\mathcal{X}$ from $\mathcal{D}$ basically amounts to finding a maximal weighted spanning tree over $\mathcal{F}$, directing edges such that each vertex has at most one parent and adding C as a parent of each $F \in \mathcal{F}$. The algorithm of [10] based on [8] is basically:

1. Compute mutual information $I(F_i, F_j \,|\, C)$ for each pair, $i \neq j$.
2. Build a complete graph G over $\mathcal{F}$ with edges annotated by $I(F_i, F_j \,|\, C)$.
3. Build a maximal spanning tree T from G.
4. Select a vertex and recursively direct edges outward from it.
5. Add C as parent of each $F \in \mathcal{F}$.

The remaining subsections of this section reports on the empirical evaluation of the proposed parallel TAN learning algorithm and is based on [16]. The parallel TAN learning algorithm is based on performing Step 1 in parallel. This step dominates the time complexity of the overall algorithm.

2.1.1 Data Sets

Three different sources of data sets were considered in the empirical evaluation. Random samples were generated from two real-world Bayesian networks of different sizes, i.e., the Munin1 [5]

Table 2.1: Data sets used in the experiments.

data set	$\lvert\mathcal{X}\rvert$	N
Munin1	189	750,000
Munin2	1,003	750,000
BCC Bank	1,823	1,140,000

and Munin2 [5] networks. For each of these networks a variable was arbitrarily chosen as class variable. The third source of data was a sample generated from a real-world financial data set (the data set provided by BCC in Work package 8).

Table 2.1 describes properties of the data sets used in the experiments. Munin1 and Munin2 are data sets of 750,000 cases generated from the Munin1 and Munin2 networks, respectively, while BCC Bank is a data set with 1,140,000 cases over financial data provided by the BCC use-case considered in Work package 8 [4]. BCC Bank is an artificial data set generated from a real-world data set maintaining some of the statistical properties of the original data. Variable names and values have been anonymised. For these experiments continuous variables were discretized into five bins. All data sets used in the empirical evaluation are complete, i.e., there are no missing values in the data.

2.1.2 Hardware

The empirical evaluation was performed on three different computer systems. One Linux server and two supercomputers Fyrkat and Vilje both running Linux:

1. A linux server running Ubuntu (kernel 2.6.38-11-server) with a four-core Intel Xeon(TM) E3-1270 Processor and 32 GB RAM.

2. Fyrkat[1] is a computer cluster where each worker node used has 2 Intel Xeon (TM) X5260 Processors and 16GB RAM. It has a total of 80 such nodes. This cluster system uses SLURM (simple Linux Utility for Resource Management) for resource management.

3. Vilje[2] is a computer cluster where each worker node has dual eight-core Xeon E5-2670 Processors and 32GB. It has a total of 1404 such nodes. This cluster system uses PBS (Portable batch System) for resource management.

It is important to notice that the experiments were performed when the system was being used by other users and running other applications. This is likely to impact performance and produce a higher variance in execution times than if the experiments were performed on a dedicated system.

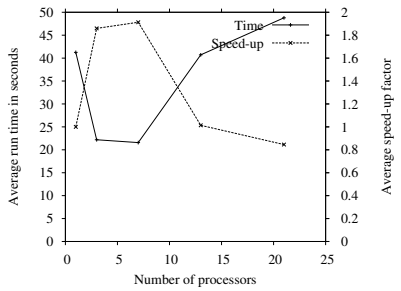
2.1.3 Results

This section reports on the results of the empirical evaluation of the proposed method for vertical parallelisation of learning the structure of a TAN. Experiments were performed using the three data sets described in Section 2.1.1 and three systems described in Section 2.1.2.

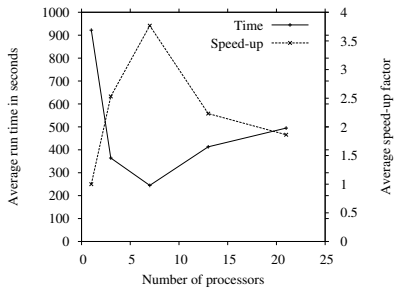
Figure 2.1 (left) shows the average run time in seconds for Munin1 on Ubuntu while Figure 2.1 (right) shows the average run time in seconds for Munin2 on Ubuntu. The left y-axis shows run time in seconds whereas the right y-axis shows the speed-up factor compared to the

[1] http://fyrkat.grid.aau.dk

[2] https://www.hpc.ntnu.no/display/hpc/Vilje



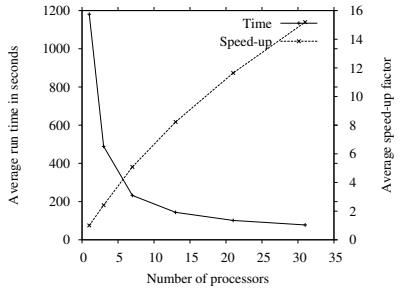
(a) Munin1



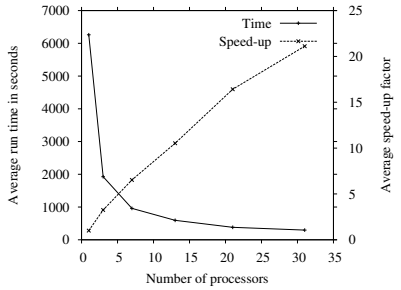
(b) Munin2

Figure 2.1: Average run times for Munin1 and Munin2 on Ubuntu.

case of using one processor. The figure shows that performance improved up to seven processes. For 13 and 21 processes performance deteriorated. This is expected as Ubuntu has only four physical cores (and eight logical cores).



(a) Munin2

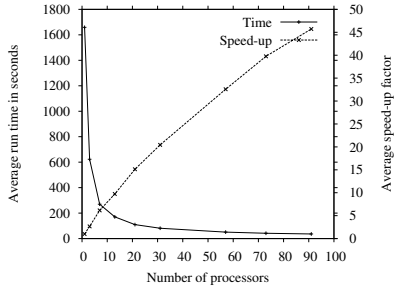


(b) BCC Bank

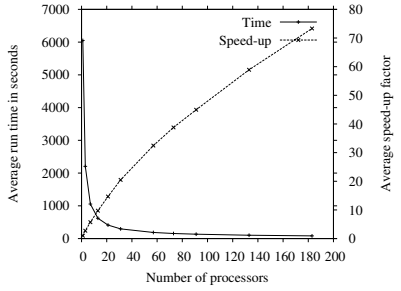
Figure 2.2: Average run times for Munin2 and BCC Bank on Fyrkat.

On Fyrkat data files were assumed mounted on the compute nodes before executing the application. Figure 2.2 (left) shows the average running time for Munin2 on Fyrkat while Figure 2.2 (right) shows the average running time for BCC Bank on Fyrkat. It is clear that the average running time improved as the number of blocks, i.e., processors used, increased. The performance should be expected to deteriorate if the number of blocks is higher than the number of processors used.

Figure 2.3 shows the average running time on Vilje for Munin2 (left) and BCC Bank (right). It is clear that the average running time improved as the number of blocks, i.e., processors used, increased.



(a) Munin2

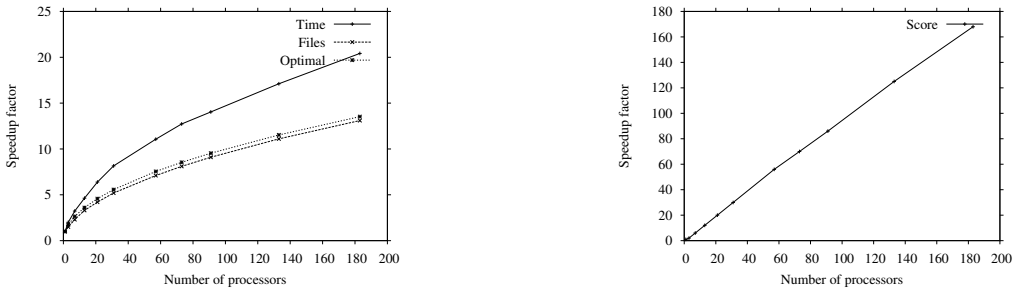


(b) BCC Bank

Figure 2.3: Average run times for Munin2 and BCC Bank on Vilje.

Figure 2.4 (left) shows the speed-up factors for reading data (*Time*) and the *theoretical* num-

ber of files read by each process (*Files*) relative to the case of one processor as well as the square root of the number of processors (*Optimal*). The *theoretical* number of files read by each process is computed as $k/v \cdot |\mathcal{X}|$ where k is the number of points in each block and v is the number of points. Figure 2.4 (right) shows the speed-up factor for scoring as a function of the number of processors used.



(a) Read - Vilje (b) Score - Vilje

Figure 2.4: Speed up as a function of the number of processors (relative to one processor).

2.2 Learning Bayesian Network Structure

This section provides a short summary of the software development efforts made in Task 5.1 on learning Bayesian network structure from data in parallel using the PC algorithm. The implementation of a parallel version of the PC algorithm is based on the theoretical work described in [17]. A performance evaluation of the main steps of the PC algorithm has identified the conditional independence testing step as (by far) the most time consuming step (results included below). Therefore, the efforts have been investing in performing this step in parallel using a thread-based approach. The parallel version of the PC algorithm also uses balanced incomplete block designs to control the parallelization [19] of the marginal independence testing and supports learning Bayesian network models with Conditional Linear Gaussian distributions.

The task of learning the structure of a Bayesian network from a data set $\mathcal{D}$ amounts to determining the DAG structure G of the Bayesian network. The PC algorithm of [18] is basically:

1. Determine pairwise (conditional) independence $I(X, Y; \mathcal{S})$.
2. Identify skeleton of G.
3. Identify v-structures in G.
4. Identify derived directions in G.
5. Complete orientation of G making it a DAG.

Step 1 is performed such that tests for marginal independence (i.e., $\mathcal{S} = \emptyset$) are performed first followed by conditional independence tests where the size of $\mathcal{S}$ iterates over $1, 2, 3$. In the process of determining the set of conditional independence statements $I(X, Y; \mathcal{S})$, it is common to use the results produced earlier to reduce the number of tests. This means, that we stop testing conditional independence of X and Y once a subset $\mathcal{S}$ has been identified such that the independence hypothesis is not rejected. When testing the conditional independence hypothesis $I(X, Y; \mathcal{S})$, the conditioning set $\mathcal{S}$ is restricted, e.g., to contain only potential neighbours of either X or Y, i.e., a variable Z is excluded from $\mathcal{S}$, if the independence test between X (or Y)

Table 2.2: Networks from which data sets used in the experiments are generated.

data set	$\lvert\mathcal{X}\rvert$	Total CPT size
ship-ship	50	130,478
Munin1	189	19,466
Diabetes	413	461,069
Munin2	1,003	83,920
sacso	2,371	44,274

and Z was previously not rejected. This is referred to as the PC* algorithm by [18], but we will refer to it as the PC algorithm.

Step 2 to Step 5 use the results of Step 1 to determine the DAG G. We will not consider Step 2 to Step 5 further in this paper as experiments demonstrate that the combined time cost of these steps is negligible compared to the time cost of Step 1. The reader is referred to, e.g., [18] for more details.

The remaining subsections of this section reports on the empirical evaluation of the proposed parallel PC structure learning algorithm and is based on [17].

2.2.1 Data Sets

Random samples of data have been generated from the five networks of different size listed in Table 2.2. Three data sets are generated at random for each network with 100.000, 250.000, and 500.000 cases. All data sets used in the empirical evaluation are complete, i.e., there are no missing values in the data.

The parallel PC algorithm is implemented employing a shared memory multicore architecture. All data is loaded into the main shared memory of the computer where the process of the program is responsible for creating a set of POSIX threads to achieve parallelisation. In the experiments, the number of threads used by the program is in the set $\{1, 2, 3, 4, 6, 8, 10, 12\}$ where the case of one thread is considered the baseline and corresponds to a sequential program.

The average computation time is calculated over five runs with the same data set. The computation time is measured as the elapsed (wall-clock) time of the different steps of the PC algorithm. We measure the computation time of the entire algorithm in addition to Step 2 and Step 3 individually as well as Step 4 and Step 5 combined.

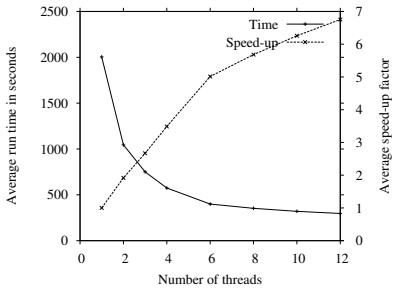
2.2.2 Hardware

The empirical evaluation is performed on a Linux computer running Ubuntu (kernel 3.10.0-229) with a six-core Intel (TM) i7-5820K 3.3GHz processor and 64 GB RAM. The computer has six physical cores and twelve logical cores.

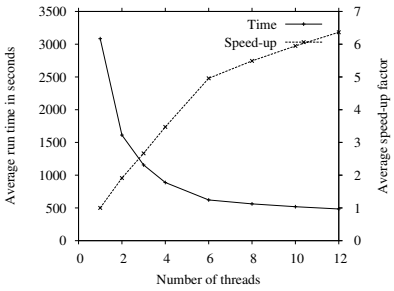
2.2.3 Results

This section reports on the results of the empirical evaluation of the proposed method for parallelisation of learning the structure of a Bayesian network using the PC algorithm. Experiments were performed using the five data sets described in Section 2.2.1 and the system described in Section 2.2.2.

Figure 2.6 (left) shows the average run time in seconds (left axis) and speed-up factor (right axis) for ship-ship using 500.000 cases. Notice that the computation time is low for the ship-

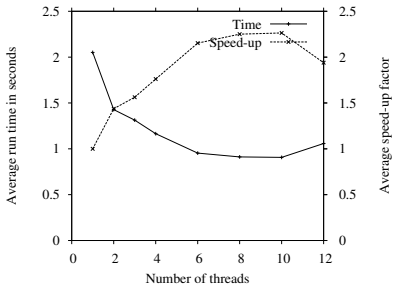


(a) Diabetes 250.000

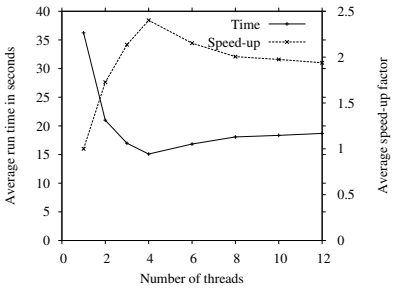


(b) Diabetes 500.000

Figure 2.5: Average run times for Diabetes with 250.000 and 500.000 cases, respectively.

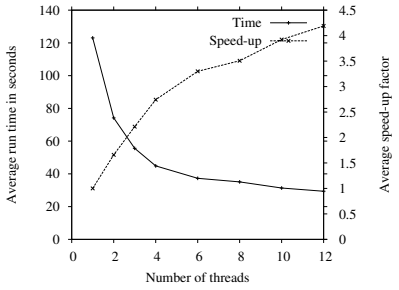


(a) ship-ship 500.000

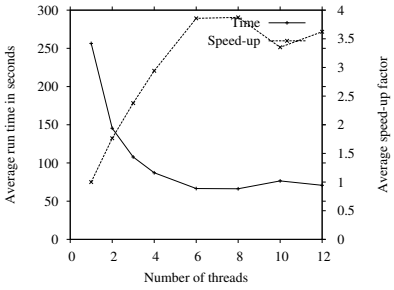


(b) Munin1 250.000

Figure 2.6: Average run times for ship-ship with 500.000 cases and Munin1 250.000 cases.



(a) Munin2 250.000



(b) Munin2 500.000

Figure 2.7: Average run times for Munin2 with 250.000 and 500.000 cases, respectively.

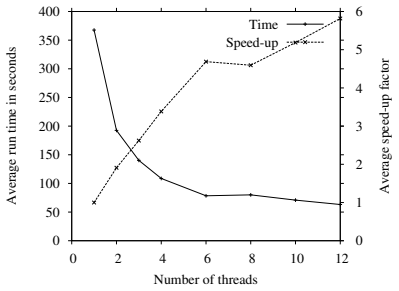
Table 2.3: Average run times in seconds for Step 2 to Step 5.

Data set	Skeleton	v-structures	Orientation
ship-ship	0	0	0
Munin1	0.005	0	0.001
Diabetes	0.001	0.004	0.002
Munin2	0.006	0.002	0.034
sacso	0.051	5.692	0.502

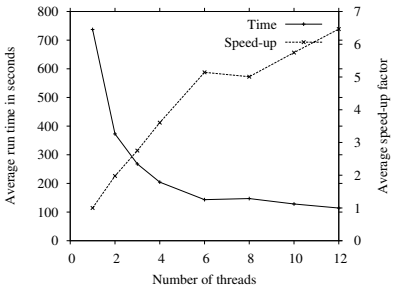
ship network even with one thread meaning that the potential improvement from parallelization is limited as the evaluation shows. Figure 2.6 (right) shows the average run time and speed-up factor for Munin1 using 250.000 cases where the speed up deteriorates for six or more threads.

Figure 2.5 (left) and Figure 2.5 (right) show the average run time and speed-up factor for Diabetes using 250.000 and 500.000 cases, respectively. The speed up factor increases smoothly for both 250.000 and 500.000 cases.

Figure 2.7 (left) and Figure 2.7 (right) show the average run time and speed-up factor for Munin2 using 250.000 and 500.000 cases, respectively. For 250.000 cases there is a smooth improvement in speed-up whereas for 500.000 cases the speed up factor drops slightly using ten or twelve threads.



(a) sacso 250.000



(b) sacso 500.000

Figure 2.8: Average run times for sacso with 250.000 and 500.000 cases, respectively.

Figure 2.8 (left) and Figure 2.8 (right) show the average run time and speed-up factor for sacso using 250.000 and 500.000 cases, respectively. The experiment on sacso using 500.000 cases is the most complex task considered in the evaluation and the task producing the most significant average speed-up of a factor 6.46 with average run time dropping from 737 to 114 seconds.

Table 2.3 shows the average time cost of identifying the skeleton, identifying the v-structures, identifying derived directions and completing the orientation to obtain a DAG.

It is clear from Table 2.3 that the costs of Step 2 to Step 5 are negligible compared to the cost of performing the independence tests. Therefore, it makes sense to focus on the cost of independence testing.

2.3 Planned work

Planned work for the remaining period includes horizontal parallelization (i.e., split the data into subsets of cases over all variables as opposed to vertical parallelization as considered above) and

bug fixes (if errors are reported) as well as adjustments to the implementation with the aim of improving performance.

3 Task 5.2: Optimization of Real Time Performance

The focus of this task is to support WP6 in the deployment of Bayesian networks and dynamic Bayesian networks on the target platform of the car. The requirements of the Daimler use-case are specified in deliverable D1.2 [2] with more details on the properties of the target platform described in deliverable D6.1 [1].

The high lights of this tasks so far has been on reducing the ROM and RAM requirements of the HUGIN C library to meet the constraints of the target platform in Work package 6 and adjusting the functionality of the HUGIN Code Wizard as well as improving time performance by parallelisation of belief update using the Divide and conquer approach.

3.1 Write Log

There is no file system on the target platform. Therefore, it is necessary to remove functionality that requires a file system from the HUGIN library. This amounts to, for instance, functionality to load (or save) a model specification, logging information in a file, load data from file. Notice that not being able to load a network specification from file requires that the network is constructed using HUGIN API calls. The Code Wizard in the HUGIN GUI supports automatic generation of code to construct a network for different programming languages.

This part of Task 5.2 has focused on removing code related to writing HUGIN Decision Engine information to a log file. This has the additional advantage of reducing the ROM requirements of the library as its size is reduced (marginally though).

3.2 Triangulation Code Refactored

The target platform on the vehicle in the Daimler use-case puts significant constraints on the size of an application. This includes severe constraints on the RAM and ROM requirements of the application.

Elements of the triangulation code of the HUGIN Decision Engine has been refactored to optimize the memory requirements of the HUGIN Decision Engine C library enabling real time performance on the target platform. The triangulation code has been refactored such that it is possible to provide a triangulation order without linking the triangulation code into the application. This reduces the binary size of the HUGIN C API library. The Code Wizard now includes a triangulation order in the generated code, see Figure 3.1.

At the time of writing, the size of the object file (*triangulate.o*) for the triangulation code, which is not linked into the library is 47Kb. This translates directly to a corresponding reduction in the ROM requirements of the library.

3.3 State Index Operator in Expressions

The Daimler use-case heavily relies on the (efficient) use of *expressions* in the Table Generator to define the content of the conditional probability distributions in the model using mathematical expressions. For instance, a (discretized) variable X may have a Normal distribution and its distribution can simply be specified as $Normal(\mu, \sigma^2)$ where μ is the mean and σ^2 is the variance. The HUGIN Decision Engine will generate the distribution of X from this expression. This is much more efficient than generating the distribution using another software and entering the numbers by hand.

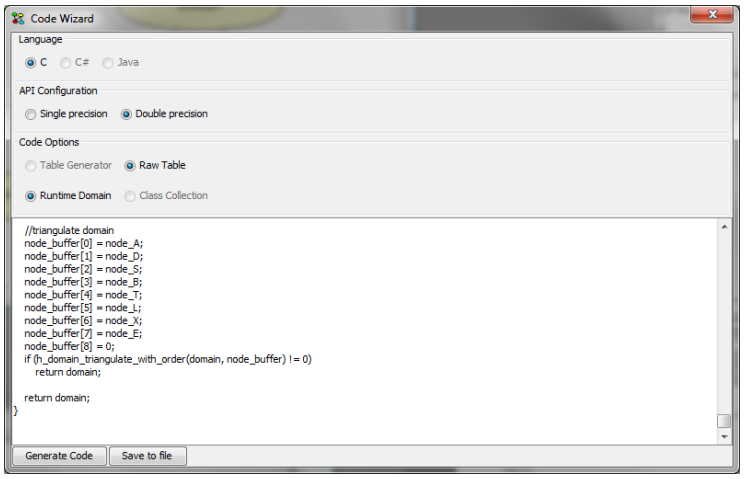


Figure 3.1: The code for specifying a triangulation order generated by the Code Wizard.

In the Daimler models (Work package 6) there is often a need to refer to the state index of a variable in an expression. In order to support this the Table Generator has been extended with the #-operator that allows the user to refer to a specific state of a parent variable in an expression. This is particularly useful when the parent is a variable of type IntervalDCNode where each state represents an interval of numbers.

The documentation of the operator is:

```
h_operator_state_index

This operator denotes the index (a nonnegative integer) of the
state of a discrete node (that is, the argument to this operator
must be an expression constructed using h_node_make_expression
applied to a discrete node). This is a numeric expression. In
the syntax of expressions, this operator is represented by
the # symbol.
```

This functionality has been released as part of HUGIN 8.2 on 29 May 2015.

3.4 Divide And Conquer Library

The Bayesian networks developed in the Daimler use-case (Work package 6) generally have large total conditional probability table (CPT) sizes and as a result produce huge clique sizes resulting in significant memory requirements. Large junction tree tables are not compatible with the use-case requirements on both space and time performance. For the dynamic Bayesian network models the number of temporal nodes in the networks makes it impossible to represent the belief state exactly. Fortunately, the maneuver networks developed in Work package 6 have three important characteristics that make it possible to reduce memory requirements and improve time performance by parallelization in a target application:

1. The network has a tree-like structure where information is propagated inwards from from leaf fragments to a single node.

2. All the leaf fragments are instances of a small set of identical sub-network structures.

3. Each leaf fragment is totally independent of surrounding fragments.

This kind of network can be divided into a set of distinct sub-network structures which by themselves occupy far less memory than the entire network and which in some cases can be processed in parallel. In this part of Task 5.2, HUGIN has developed a C code library to compute beliefs for such networks using only the distinct sub-network structures and a network (or set of networks) to combine the results produced by the sub-networks.

This section describes how the divide and conquer library works and how to implement and integrate a maneuver network in a target application. For clarity and brevity error handling code is omitted from the C code examples given.

3.4.1 Dividing The Maneuver Network

A generic maneuver network structure is depicted in Figure 3.2.

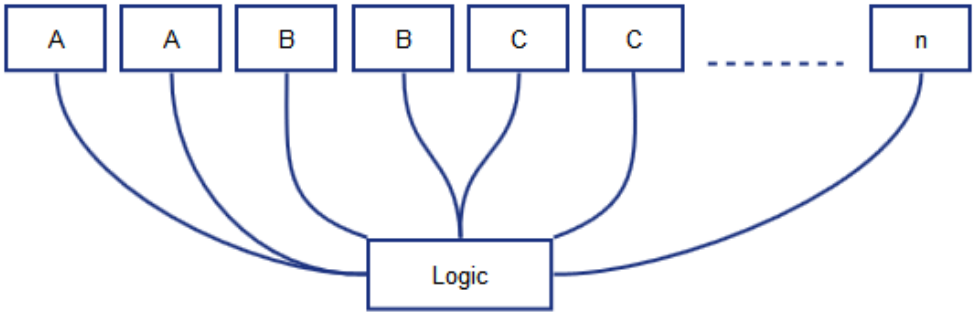


Figure 3.2: Generic structure of maneuver network. Ideal for the divide and conquer approach as independent leaf fragments repeat identical sub-network structures.

Each leaf component is an instance of a specific sub-network, e.g., the network contains a number of instances of sub-network 'A', a number of instances of 'B' and so on. Instead of having a single runtime data structure for the entire maneuver network with the same sub-networks duplicated in a number of places, we can instead get the same results by having just a single instance of each sub-network and do the computations multiple times. Due to the fact that each leaf fragment is independent it is possible to iterate a single sub-network to compute beliefs for all leaf fragments implementing that particular network. After all leaf fragments are computed, results are collected and combined in the *logic* network (this network can, in principle, be any type of Bayesian network, but in the Daimler use-case this part is implementing logical relations and therefore referred to as the *logic* network) and finally the maneuver advice can be computed. This divide and conquer approach requires a bit more bookkeeping and decision engine instrumentation but reduces memory requirements to that of the set of small distinct network structures - which is much better compared to the big maneuver network with many duplicated network structures. Furthermore, it is possible to process the sub-networks in parallel to potentially improve time performance.

3.4.2 MNVRLIB Overview

The C code library is called MNVRLIB (short for maneuver library) and goes through a number of initialization stages before it can compute maneuver advice. During these initialization stages the mnvrlib specific data structures are instantiated and configured. In short the following is performed:

1. Allocate main divide and conquer data structure
2. Inject sub-network domain structures

3. Create tasks
4. Wire in the evidence source
5. Create batches (i.e. put tasks together in groups)

Once initialization is completed the mnvrlib is ready to compute maneuver advice. A single case is processed in these distinct steps:

1. Enter case evidence the into divide and conquer data structure (or instruct mnvrlib to skip evidence)
2. Invoke *perform_single_dac* function to compute beliefs
3. Get results using *h_node_get_belief* on maneuver advice node

The library consists of a number of helper functions for setting up the infrastructure needed to carry out the divide and conquer propagation. For a specific maneuver network some service code must be implemented. To implement different maneuver networks one only needs to change the implementation of the specific service code in a single header/c-file pair, yielding a small generic interface to make switching between different variants of a maneuver network in the target application easier.

A main data structure *hugin_dac* must be defined and configured for the specific maneuver network. The data structure declares all the needed *h_domain_t* sub-network domains, a data structure for gathering evidence and the tasks for performing propagation in the distinct network fragments.

This is done in a C header file *hugin_dac.h*:

```
struct hugin_dac {
  //domains
  h_domain_t LOGIC;
  h_domain_t ....

  //data
  struct source_data* online_data;
  struct source_discrete_evidence_driver* online_driver;

  //all the tasks
  struct task_description* task_LOGIC;
  struct task_description* task_...

  //grouping tasks into batches
  struct task_runner* batch_...
  struct task_runner* batch_LOGIC;
};
```

A data structure for gathering evidence is also defined, specific for the evidence nodes of the maneuver network. Each member of the data structure is a place holder for evidence needed for computing a specific network fragment. Good naming convention could be letting the first part of member name denote the specific fragment and the last part of the name identify the node.

```
struct source_data {
  h_number_t FRAGMENT_A_NODE_MEASUREMENT_X;
  h_number_t FRAGMENT_A_NODE_MEASUREMENT_Y;
  h_number_t ...
}
```

Finally the helper functions are also declared in *hugin_dac.h*:

```
create_hugin_dac
setup_tasks
create_online_data
setup_online_evidence
setup_task_batches
set_skip_data
perform_single_dac
```

The implementation of these functions are specific to the maneuver network in question and go in the *hugin_dac.c* file.

3.4.3 Implementing a Maneuver Network Using MNVRLIB

The main challenge to integrate a maneuver network is to correctly sew together all the network fragments, i.e., which nodes in one fragment produce a probability distribution to insert into another fragment, which nodes receive evidence and so on. Also, if work is to be carried out in parallel using threads then the *h_domain_t* domain data structures must be properly synchronized which is done at a later point by grouping tasks into batches and scheduling correct execution order.

The sewing together of all the network fragments is done in the *setup_tasks* function. A task is created for each specific network fragment and is configured using maneuver library service functions for transfer of likelihood to and from other tasks and to exercise temporal nodes:

```
//creation of some task
hdac->task_SOMETASK = create_task(0);

//setup some nodes in SOMETASK to be receivers
//of likelihood transfer from other tasks
configure_likelihood_nodes(hdac->task_SOMETASK,
  //node names in domain
  (const char *[]){ "somenode", "node1", "node2", ... },
  //number of nodes
  10);

//setup transfering of likelihood from
//node in OTHERTASK to node in SOMETASK
configure_belief_copy(hdac->OTHERTASK, "othernode",
  hdac->task_SOMETASK, "somenode");

//setup moving of time window for temporal nodes in SOMETASK
allocate_temporal_transfer(SOMETASK,
  //clone node names in domain
  (const char*[]){ "T0.nodeA", "T0.nodeB" },
  //number of temporal clones
```

```
  2);

configure_temporal_transfer(SOMETASK,
  //target clone name in domain
  "T0.nodeA",
  //joint marginal master nodes in domain
  (const char*[]){ "T1.nodeA" },
  //number of nodes in joint marginal
  1);

configure_temporal_transfer(SOMETASK,
  //target clone name in domain
  "T0.nodeB",
  //joint marginal master nodes in domain
  (const char*[]){ "T1.nodeB" },
  //number of nodes in joint marginal
  1);
```

Notice that temporal nodes makes it possible to manage dynamic Bayesian networks where the belief state over interface nodes are transferred between time slices. In the implementation it is possible to transfer single marginals meaning that a joint distribution over a set of interface nodes is approximated using a product of marginals (one for each node in the interface). This can been seen as a simplified form of the Boyen-Koller schme [6, 7].

After tasks are created, we sew together members of the evidence data structure and the target nodes in all tasks, this is done in the *setup_online_evidence* function like so:

```
configure_source_evidence(hdac->task_SOMETASK, hdac->online_driver,
  (const char*[]){ "nodeA", nodeB"},
  //mapping to source data pointers
  (h_number_t*[]){ &hdac->online_data->SOMETASK_A,
      &hdac->online_data->SOMETASK_B },
  //number of nodes
  2))
```

Finally the grouping of tasks into batches is done in the *setup_task_batches* function. These batches can be either of blocking or asynchronous fashion, meaning that main thread of execution can block until all tasks in a batch has been executed, or the batch can be dispatched and executed in parallel in a worker thread without blocking the main thread. For parallel divide and conquer the blocking/asynchronous batches and a function *wait_for_tasks* for waiting for completion of all tasks in a batch is the only synchronization method available.

Using these simple features we can schedule work to be carried out in a manner where each thread operates on its own distinct resources. This is done by having a thread sequentially compute all network fragments that use the 'A' domain, meanwhile another thread sequentially computes all network fragments that use the 'B' domain and so on and finally wait for all threads to complete before collecting and computing result in the 'LOGIC' domain.

This is illustrated by the example in Figure 3.3, depicting the path of execution in the target application at the point where divide and conquer takes place. Application logic executes in the main thread of execution and at some point initiates divide and conquer computation. At this point worker threads are spawned and carry out their tasks, e.g. one thread computes two fragments using a single 'A' domain, other thread computes two fragments using a single 'B' domain while third thread computes four fragments using single 'C' domain. Only place where

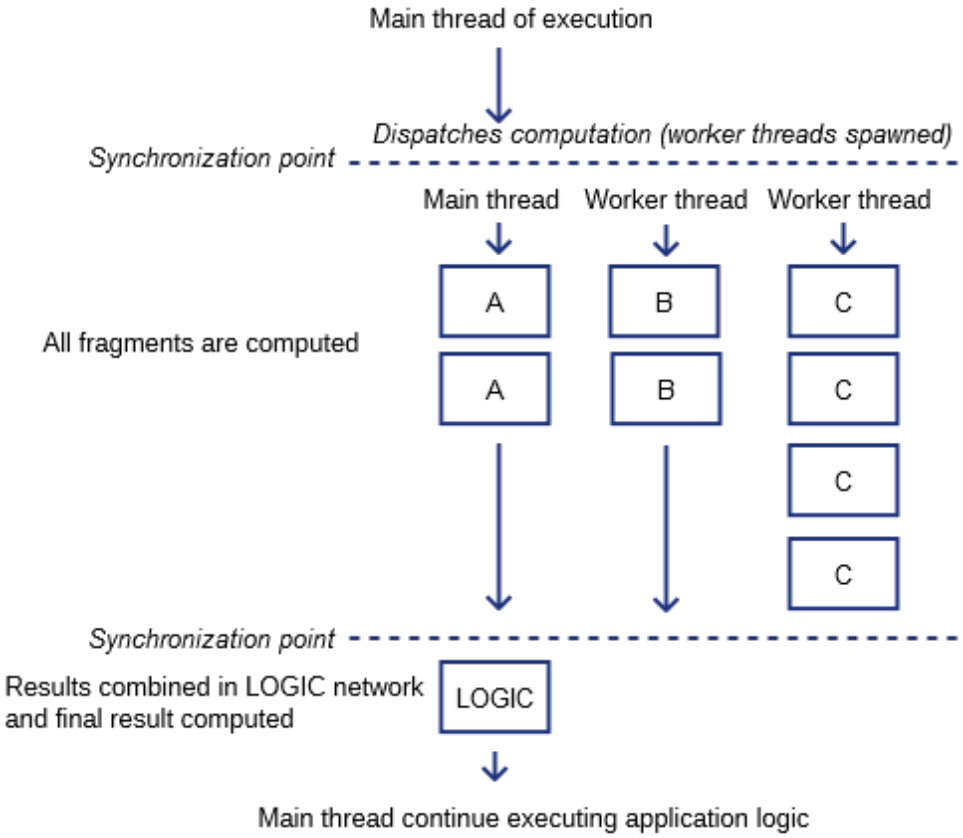


Figure 3.3: Parallel computation.

information is exchanged between data structures are at the synchronization points. Finally the results from all the fragment computations are combined in the LOGIC domain and the final maneuver advice result is computed.

The batches were created using the following code:

```
hdac->batch_A = create_blocking_task_runner({ hdac->task_LEFT_A,
  hdac->task_RIGHT_A },
  //number of tasks
  2);

hdac->batch_LOGIC = create_blocking_task_runner({ hdac->task_LOGIC},
  //number of tasks
  1);

hdac->batch_B = create_async_task_runner({ hdac->task_LEFT_B,
  hdac->task_RIGHT_B },
  //number of tasks
  2);

hdac->batch_C = create_async_task_runner({ hdac->task_LEFT_FIRST_C,
  hdac->task_LEFT_SECOND_C,
  hdac->task_RIGHT_FIRST_C,
  hdac->task_RIGHT_SECOND_C },
  //number of tasks
  4);
```

And the following code dispatches the computation illustrated in Figure 3.3. As a blocking batch executes in the main thread it is important that this batch is dispatched last before any synchronization point, else dispatching of the other batches will take place after the blocking batch has executed introducing an unneccesary delay.

```
//first synchronization point.
//evidence has been inserted and ready to compute.
dispatch_tasks(hdac->batch_C);
dispatch_tasks(hdac->batch_B);
dispatch_tasks(hdac->batch_A);

//second syncrhonizatino point.
//wait for all tasks to complete before continuing.
wait_for_tasks(hdac->batch_C);
wait_for_tasks(hdac->batch_B);
wait_for_tasks(hdac->batch_A);

//now compute final result using LOGIC network.
dispatch_tasks(hdac->batch_LOGIC);
wait_for_tasks(hdac->batch_LOGIC);
```

3.4.4 Integrating The Maneuver Network

Once the implementation of a maneuver network is complete we only need a few lines of code to integrate the network in the target application. This code mainly takes care of calling all the initialization functions implemented in the previous section and for compiling and injecting the *h_domain_t* domain data structures. The configuration phase amounts to the following steps and should be placed in the initialization section of the target application:

1. Allocate the main divide and conquer datastrucure to keep track of all our objects

   ```
   struct hugin_dac* hdac;
   hdac = create_hugin_dac();
   ```

2. Inject *h_domin_t* domain sub-networks

   ```
   hdac->LOGIC = logic_domain;
   hdac->... = ...;
   ```

3. Triangulate and compile domains

   ```
   h_domain_triangulate(hdac->LOGIC, h_tm_best_greedy);
   h_domain_compile(hdac->LOGIC);
   ...
   ```

4. Call function to setup all the tasks. This is the part where all the network fragments are configured.

   ```
   setup_tasks(hdac);
   ```

5. Create evidence data structure

```
hdac->online_data = create_online_data();
```

6. Create and configure evidence driver

```
//e.g. with a capacity of 10 nodes
hdac->online_driver = create_source_discrete_evidence_driver(10);
```

7. Sew together evidence and tasks

```
setup_online_evidence(hdac);
```

8. Call function to group tasks into batches. Dealing with threads and concurrency for parallel execution is handled here.

```
setup_task_batches(hdac);
```

9. Pointer to the node where results are read from is also needed later on

```
h_node_t ADVICE;
ADVICE = h_domain_get_node_by_name(hdac->LOGIC, "ADVICE");
```

The mnvrlib framework and the maneuver network have now been initialized and is ready for computing maneuver advice. Code for exercising the maneuver network should be placed in the run time part of the target application. A single computation is done in the following steps:

1. Set evidence on the *hdac->online_data->** members through simple value assignment. This way we can easily integrate evidence from any source available within the target application.

```
hdac->online_data->FRAGMENT_A_NODE_MEASUREMENT_X = 0.35;
hdac->online_data->NODE_Y = 1;
//..
```

2. Now instruct mnvrlib to load the evidence found in *hdac->online_data*

```
source_load_case(hdac->online_driver);
```

3. Instruct mnvrlib to perform its computations by either using or skipping loaded evidence. This function is useful if the target application could not produce data within the required deadline for the next computation and we need to proceed.

```
//instruct tasks to either skip or use evidence:
//0 enter evidence and propagate, 1 propagate with no evidence
set_skip_data(hdac, 0);
```

4. Call function to perform divide and conquer propagation

```
perform_single_dac(hdac);
```

5. And finally read out results from target node using regular *h_node_get_belief*

```
h_node_get_belief(ADVICE, 0)
h_node_get_belief(ADVICE, 1)
..
```

The target application now contains an implementation of a specific maneuver network and run time part of application has a code segment to enter evidence and compute maneuver advice using divide and conquer propagation.

3.5 Planned Work

Planned work for the remaining period includes refinements of the implementation to meet the requirements of the use-case in Work package 6 as well as the implementation of instantiated junction trees to take advantage of knowledge of variables that are always observed. The objective of the latter is to improve both time and space performance compared to an approach not taking advantage of the evidence during belief update in a Bayesian network.

4 Task 5.3: Approximate Inference in Dynamic BN Models

This task is devoted to extending the support for dynamic Bayesian network models including both exact and approximate inference in such models. The focus is on support for dynamic Bayesian network models through time-slicing (i.e., a sliding time-window representing a finite number of steps). The exact inference algorithm is based on [11, 12] whereas the approximate inference algorithm is to be based on [6, 7]. Notice that the Divide and Conquer library introduced in Section 3.4 implements a simple version of the Boyen-Koller approach.

The work on improving the support for the dynamic Bayesian networks is driven by the requirements of the Daimler use-case in Work package 6. The developed functionality is relevant for both Task 5.3 and Task 5.4, which considers adaptation of parameters in dynamic Bayesian networks. The developments so far have focused on improving the support for dynamic Bayesian networks in the Data Frame of the HUGIN GUI in order to support the development of dynamic models in Work package 6. This includes functionality to process streaming data as well as to evaluate both static and dynamics models processing streaming data.

4.1 Code Wizard

The code wizard feature available in the HUGIN AMIDST Graphical User Interface (GUI) has been extended to support generating C, C# and Java code for class collections in turn making it possible to generate code for constructing a Dynamic Bayesian network. This code can be used to integrate the specification of a Dynamic Bayesian network in a software program (as opposed to loading the specification from file). This is relevant for the Daimler use-case in Work package where the target platform does not have a file system to store the model specification.

4.2 Dynamic Bayesian Network Models and Data Frame

Efficient use of Dynamic Bayesian networks requires functionality to test and evaluate the performance of a dynamic Bayesian network in the HUGIN AMIDST GUI. In the HUGIN AMIDST GUI this is done using the Data Frame. The Data Frame (or Data Tool) shown in Figure 4.1 allows the user to enter a batch of data cases specified in a data file into a model in HUGIN. In Task 5.3, this tool has been extended with the ability to perform belief update using DBN models, i.e., to perform in smoothing, filtering and prediction. This functionality involves preprocessing of time series data, selection of beliefs to compute and propagate evidence.

4.2.1 Assumptions and notation

Let $\mathcal{M}_n^{\mathcal{V}}$ be a dynamic Bayesian network model with n time slices where each time slice is a Bayesian network where the qualitative part is a DAG $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ and $\mathcal{V} = \{A, B, C, \ldots\}$ is a finite set of nodes, $\mathcal{E}$ is a set of edges $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$ of ordered pairs of distinct nodes. When $(v, u) \in \mathcal{E}$ then there is a directed link from v to u in $\mathcal{G}$. We shall use subscript notation to refer to the DAG representing the ith time slice i.e. $\mathcal{V}_i$ is the set of nodes of the ith time slice. A temporal interface in $\mathcal{M}_n^{\mathcal{V}}$ is a set of edges connecting time slice $i-1$ with time slice i, i.e. $\mathcal{I}_i \subseteq \mathcal{V}_{i-1} \times \mathcal{V}_i$ is the temporal interface of time slice i which consists of a set of directed edges. Hence, when $(v_{i-1}, u_i) \in \mathcal{I}_i$ where $v_{i-1} \in \mathcal{V}_{i-1}$ and $u_i \in \mathcal{V}_i$, then there is a temporal link from node from v_{i-1}

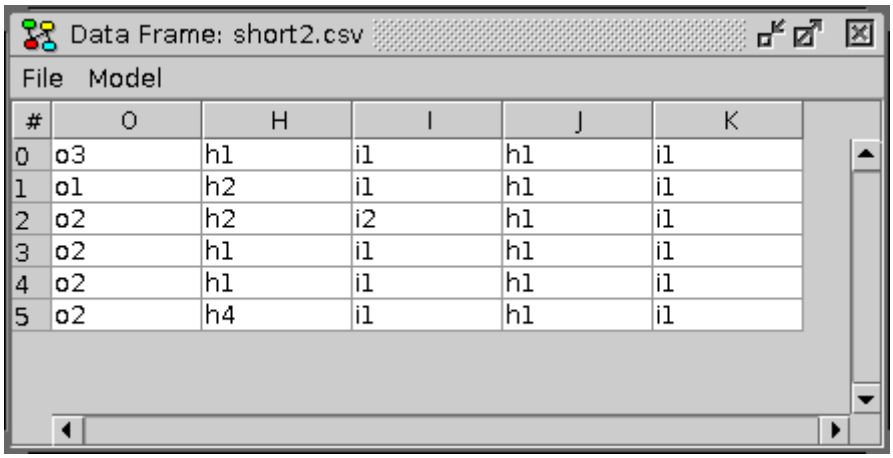


Figure 4.1: The Data tool for inspecting and manipulating input data.

to u_i for all $1 \leq i \leq n$. Furthermore, we shall refer to $\mathcal{V}_n$ as the complete set of nodes of $\mathcal{M}_n^{\mathcal{V}}$, i.e. $\mathcal{V}_n = \{A_1, B_1, \ldots, A_n, B_n \ldots\}$. Finally, we shall also refer to n as time horizon of $\mathcal{M}_n^{\mathcal{V}}$.

We assume that some system has generated a time series data set D containing observations of the variables in $\mathcal{V}$ at discrete time steps $t = 1, t = 2, \ldots, t = k$. For instance, D could have been generated by monitoring some process with a set of sensors that are reporting values a certain frequency. Thus, the jth row in D, denoted d_j, is a vector, representing an instantiation of the variables in $\mathcal{V}$ at discrete time step j ($j > 0$). We shall allow D to contain missing values (MAR or MCAR).

4.2.2 Preprocessing Time Series Data in the Data Frame

A tool has been added to the HUGIN AMIDST GUI that is able to process D and for a particular model, $\mathcal{M}_n^{\mathcal{V}}$, say, and report on the belief states of certain variables in each time step. To obtain this a new procedure has been implemented in HUGIN which transforms a time series data set D to a new data set D^{Tn} in which each row (d_j^{Tn}) corresponds to an instantiation of the nodes in $\mathcal{M}_n^{\mathcal{V}}$ at time step j. The data in each case is transformed from a time series to a sequence of time windows with horizon n. Formally, $d_j^{Tn} = \{d_j, d_{j+1}, d_{j+2}, \ldots, d_n\}$ for all $1 \leq j \leq 1 + k - n$.

Assume D containing the data in Table 4.1 which is a time series over the variables O and H where O has states o_1 and o_2 while H has states h_1 and h_2 and $\mathcal{M}_3^{\{O,H\}}$ is a 3-time-slice-DBN over O and H. Then, the procedure generates the new data shown in Table 4.2.

Table 4.1: A time series dataset over the discrete variables H and O.

#	H	O
1	h1	o1
2	h2	o1
3	h1	o2
4	h1	o1
5	h2	o2
6	h1	o2

Table 4.2: The D^{Tn} transformation of the data in Table 4.1.

#	H_1	O_1	H_2	O_2	H_3	O_3
1	h1	o1	h2	o1	h1	o2
2	h2	o1	h1	o2	h1	o1
3	h1	o2	h1	o1	h2	o2
4	h1	o1	h2	o2	h1	o2

4.2.3 Tool for Belief Update in DBNs

$\mathcal{M}_n^{\mathcal{V}}$ can be used for Bayesian smoothing, filtering as well as prediction and functionality for evaluating $\mathcal{M}_n^{\mathcal{V}}$'s performance for some variable has been added to the Data Frame of HUGIN AMIDST.

Assume a user wishes to evaluate $\mathcal{M}_n^{\mathcal{V}}$'s for computing the posterior belief a set of nodes $\mathcal{F} \subseteq \mathcal{V}_n$ in the model.

Let $P(X_i|D_{(1...i)})$ denote the probability distribution calculated by the model on node X_i given that the model in the past has seen the sequence of observations $d_1^{Tn}, d_2^{Tn}, \ldots, d_i^{Tn}$, corresponding to the first i rows of D^{Tn}. The DBN belief update functionality implemented in HUGIN AMIDST propagates through the transformed data set D^{Tn} and for each row i and for each node $f \in \mathcal{F}$, it calculates $P(f|D_{(1...i)})$ and appends it to d_i. When $j = i$ the calculation $P(Tj.V_k|D_{(1...i)})$ is Bayesian filtering, when $j > i$ it is prediction and when $j < i$ it is smoothing.

An example, where a belief update has been performed on the variable H at temporal time step 4 is shown in Table 4.3.

Table 4.3: The data from Table 4.2 with predictions performed for the variable H one time step ahead and two time steps ahead respectively.

#	H_1	O_1	H_2	O_2	H_3	O_3	$P(H_4 = h1\|d_1 \ldots i)$	$P(H_5 = h1\|d_1 \ldots i)$
1	h1	o1	h2	o1	h1	o2	0.82	0.84
2	h2	o1	h1	o2	h1	o1	0.87	0.71
3	h1	o2	h1	o1	h2	o2	0.77	0.76
4	h1	o1	h2	o2	h1	o2	0.81	0.84

4.3 Implementation in the Data Tool of HUGIN AMIDST

4.3.1 Transformation from D to D^{Tn}

The transformation from D to D^{Tn} has been implemented in the Data Tool. The dialog is available in the menu that appears on a right-click on one of the column headers. The menu item "Transform to time slices (DBN only)" is active when the associated run time model is a DBN and when at least one column header matches a variable name in the model (without the Tn prefix), otherwise it is greyed out. The dialog is shown in Figure 4.2.

4.3.2 Selection of Variables to Monitor

A dialog for selecting the nodes, time slices and states to monitor in the belief update process has been implemented. An outline of this dialog is shown in Figure 4.3 where the probability

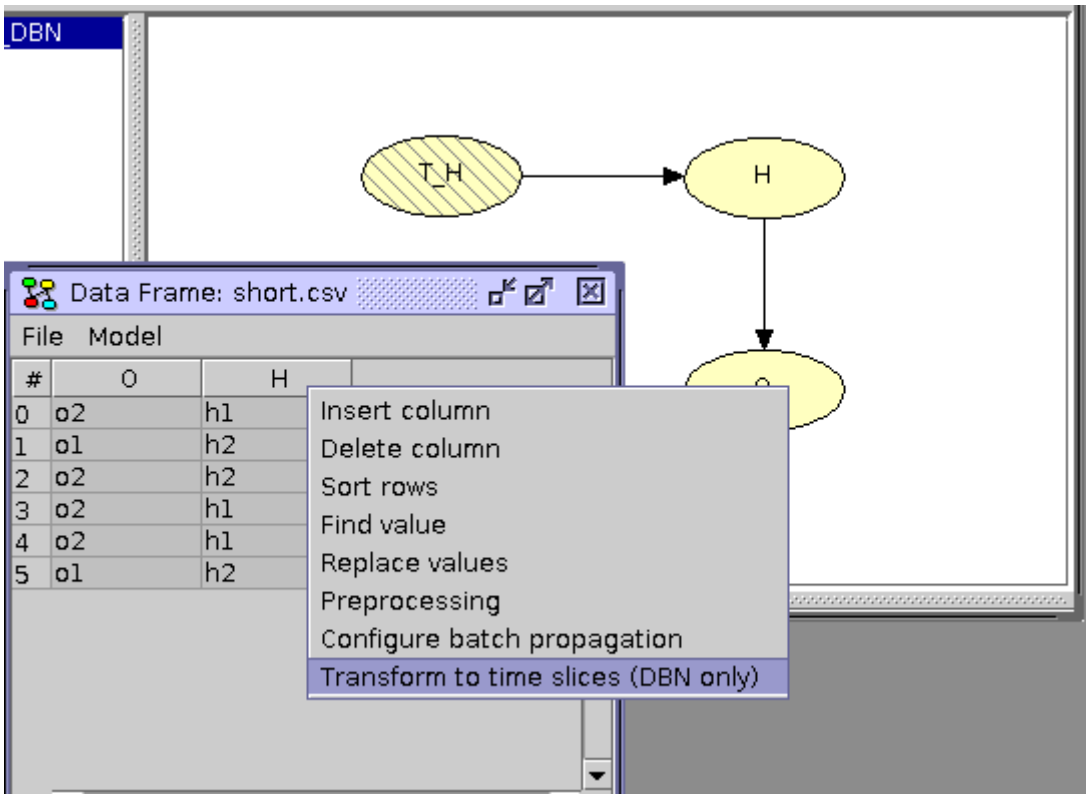


Figure 4.2: The Data tool menu item selected for performing the transformation from time series D to D^{Tn}.

$P(H|D^{Tn})$ has been selected for the time slices 4, 5 and 6.

The implementation determines whether the user wishes to perform smoothing, filtering or prediction simply by comparing the selected time slice with the model's number of time slices. If the selected time slice is beyond the last time slice of the DBN, then it will be regarded as a prediction otherwise it will be assumed to be filtering or smoothing. When Bayesian filtering or smoothing is selected, the filtered or smoothed variable sometimes is included in the data D^{Tn} and in such cases the user is queried whether he/she wishes to ignore this column as input to the belief update process. Otherwise the resulting calculations will be deterministic.

Update of the Batch Configuration Dialog

In order to obtain similar dialog windows, the Belief menu item has been updated to have a similar appearance as the configuration menu for belief update in the DBN. The menu has the same appearance except from the option of selecting a time slice. See figure 4.4.

Propagation of Selections

To be able to propagate the evidence when selections have been entered, the procedure that propagates rows has been updated to use DBN prediction. The dialog for selecting to propagate all rows is shown in Figure 4.5 and the resulting table is shown in Figure 4.6.

Supported Node Types

The implementation of support for belief update in DBNs supports the following types of nodes:

- **Discrete Chance Nodes and Discrete Function Nodes**

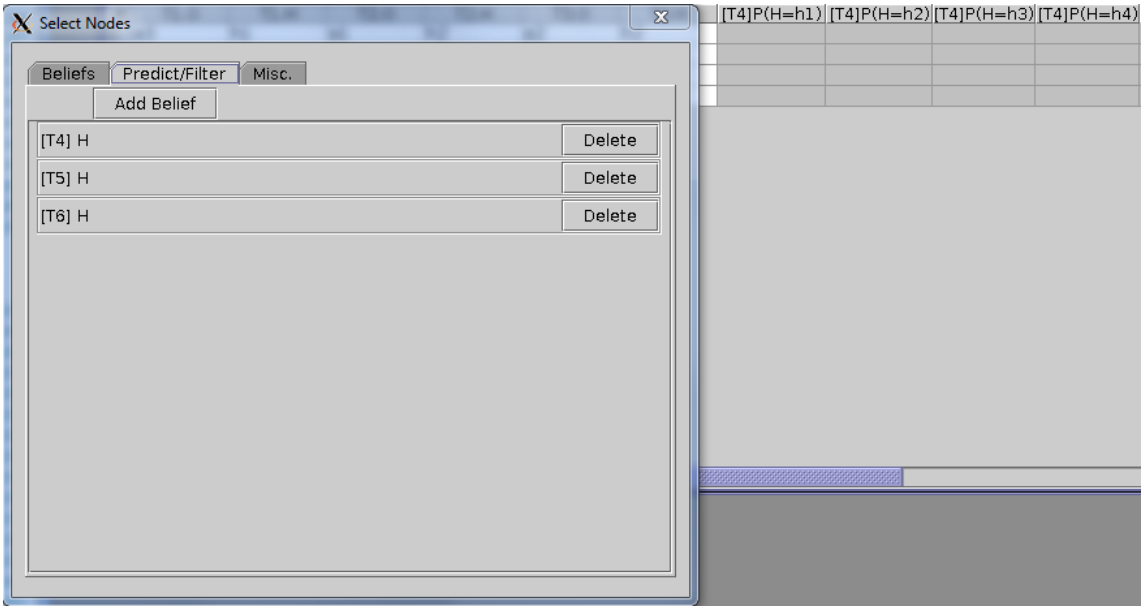


Figure 4.3: The Data Tool dialog for configuration of belief update in a DBN. In the figure the tool has been configured to perform predictions on the variable H in time slices 4,5 and 6. The loaded model has 3 time slices, (T1, T2 and T3) and hence, it will calculate predictions for H in these 3 time slices.

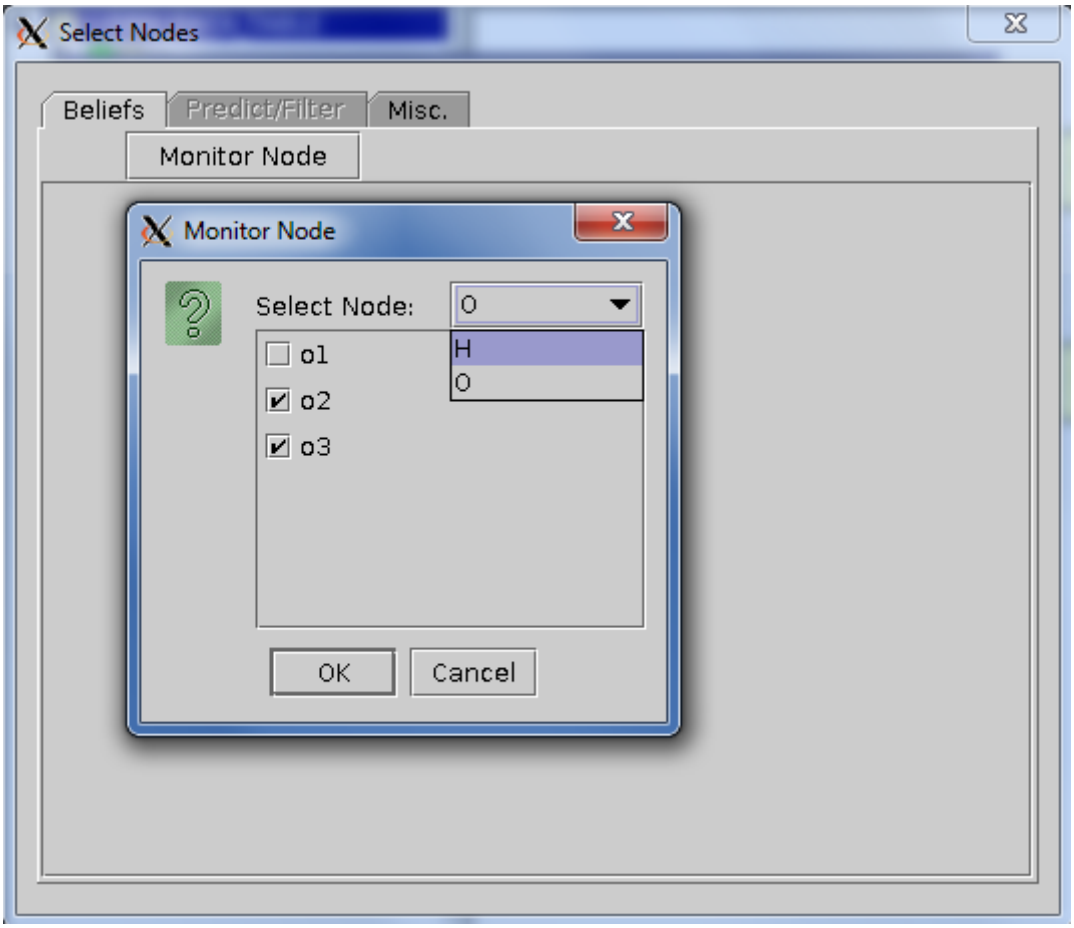


Figure 4.4: The Data tool's menu for selection of which nodes to monitor was updated to resemble the Smooth/Predict/Filter interface.

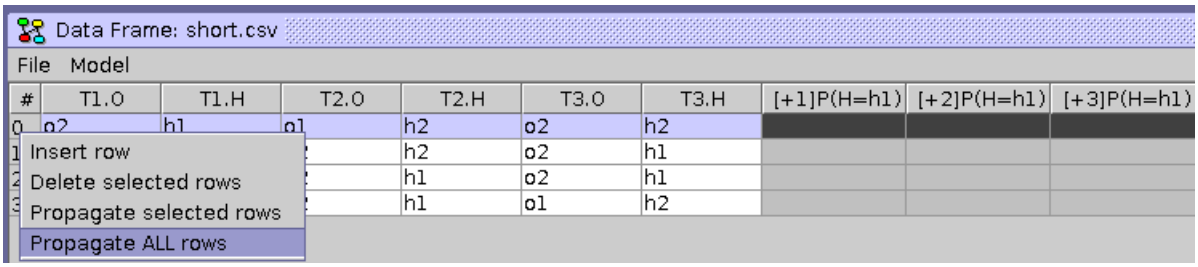
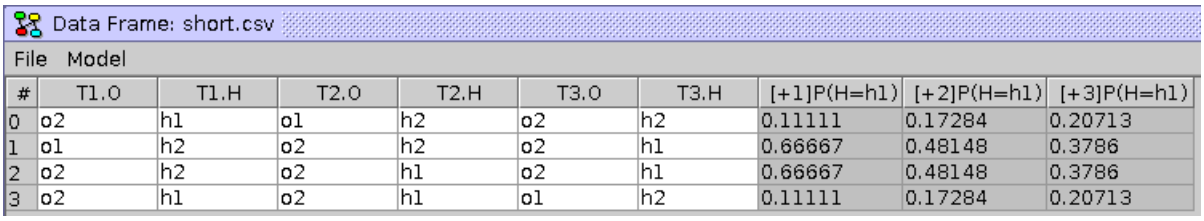


Figure 4.5: The Data tool for inspecting and manipulating input data.



#	T1.0	T1.H	T2.0	T2.H	T3.0	T3.H	[+1]P(H=h1)	[+2]P(H=h1)	[+3]P(H=h1)
0	o2	h1	o1	h2	o2	h2	0.11111	0.17284	0.20713
1	o1	h2	o2	h2	o2	h1	0.66667	0.48148	0.3786
2	o2	h2	o2	h1	o2	h1	0.66667	0.48148	0.3786
3	o2	h1	o2	h1	o1	h2	0.11111	0.17284	0.20713

Figure 4.6: The Data tool for inspecting and manipulating input data.

  - Each State on the node can be selected individually.

- **Continuous Chance Node**
  - Mean (μ) and
  - Variance (σ^2)

- **(Continuous) Function Node**
  - The calculated value according to propagated input.

Support for Function Nodes as Input

In the Data Tool's data propagation procedure Function nodes obtain special treatment. Usually, function nodes have an associated expression which defines a real-valued function over the configuration of the parent nodes. But function nodes may also be parents of other nodes in valid HUGIN models. Evidence on function nodes is unable to influence parent nodes, as evidence on Function nodes is restricted to follow the functional trail. Thus, if it is possible to specify the value of a function node it will influence its descendants. Functionality has been added to the Data Tool that replaces the expression on a function node with the associated value from a data file. When the propagation of a row is done the expression is restored on the node.

Leave One Out Parameter Learning and Propagation

A typical use of Data Propagation in the Data Tool is evaluation of a learned model. The evaluation may however, be misleading if the evaluation is performed on the same data which was used for learning the model. It is always possible to separate learning data from evaluation data but it is a time consuming task. Therefore a tool that simplifies this separation is necessary.

A new propagation method has been implemented that, for each case learns the probability tables on the model using EM Learning on the case based on all the data – except from the current case.

More specifically, assume that we use model $\mathcal{M}$ to propagate the data set D and that the current case $d \in D$ is about to be propagated. We then learn a new model $\mathcal{M}'$ based on the data in $D \backslash \{d\}$ and use $\mathcal{M}'$ to propagate d.

We call this propagation method *Leave-One-Out propagation* and it is accessible in the Data Tool. Before running this propagation the user can configure a set of variables as targets for analysis (e.g. the class variable in a classifier). These variables will be included while learning the model $\mathcal{M}'$ but they will be excluded when $\mathcal{M}'$ is used to propagate row d.

Evaluation of Belief Update in DBN

The Data Processor (Previously called the Data Preprocessor) is a tool that allows a user to define a set of operations that he/she wishes to perform on a data set. The tasks can be saved in a file and stored for later use or shared with other users with similar data.

In order to allow users to evaluate the results of belief update in a DBN, it was necessary to be able to align the predicted value in the data set with the actual future value. In a data set the needed operation is a translation of a column containing a prediction in n say, time steps, is to shift all cells of that column n rows down. This functionality has been added to the Data Processor.

The Data Processor also obtained possibility to clone a column and provide it with a name in order to avoid loosing the original data. The Data Processor executes tasks in the order the tasks appear in a list, so a new useful feature is that the tasks can be rearranged in the task list. Furthermore, a new feature is that the result of applying the selected tasks can be previewed in a table.

Evaluation of Classifiers - Multistate Confusion Matrix

The Data Tool has been extended to support calculation of a confusion matrix for class variables with more than two states. A multistate confusion matrix has now been implemented. It works by simply assuming that the classifier simply returns the most likely state. Figure 4.7 shows the user interface for the multistate confusion matrix implemented in HUGIN AMIDST.

Data Analysis - Data Plot Improvements

The Data Tool was previously able to plot a pair of data sets, one for the X-Axis and one for the Y-Axis. This tool has been extended with the ability to plot multiple data sets simultaneously and to select to plot the data against case number such that they are shown as sequences. For convenience the tool has the ability to configure the color for each curve and to include a legend i the plot. Another small but convenient improvement is that it is possible to resize the plot which enables the user to get the graph in the exact size that he/she wants.

This functionality is used as part of the model evaluation in Work package 6. It is the basis of future developments of functionality to evaluate dynamic Bayesian networks using time AUC as defined by the user requirements.

4.4 Planned Work

Planned work for the remaining period includes the implementation of the Boyen-Koller algorithm for approximate inference in dynamic Bayesian networks as well as refinements of the HUGIN Graphical User Interface to support the efficient use and evaluation of inference in dynamic Bayesian networks. This includes functionality to evaluate the performance of a dynamic Bayesian network on time series data as required by Work package 6.

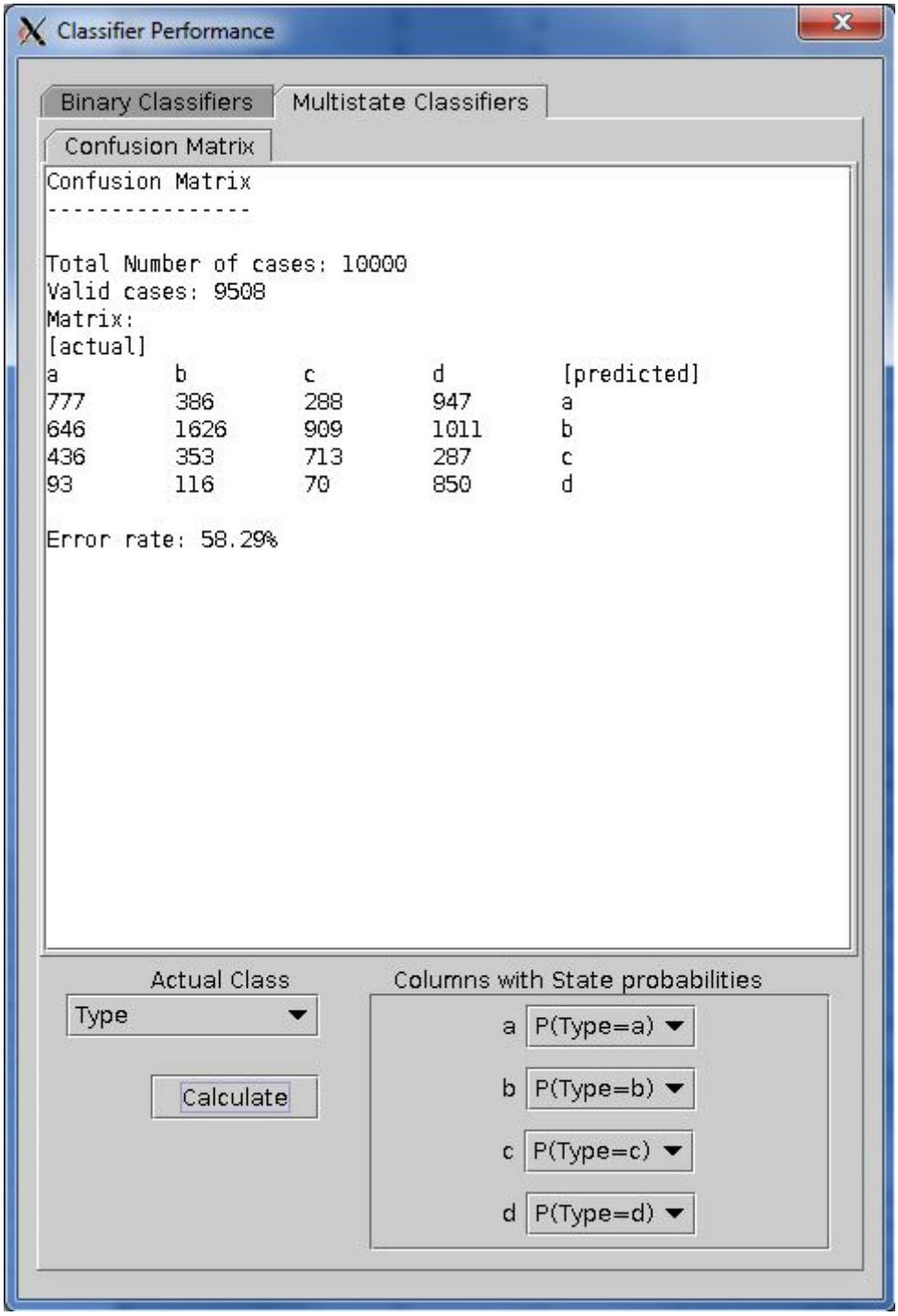


Figure 4.7: The user interface for the multistate-confusion matrix.

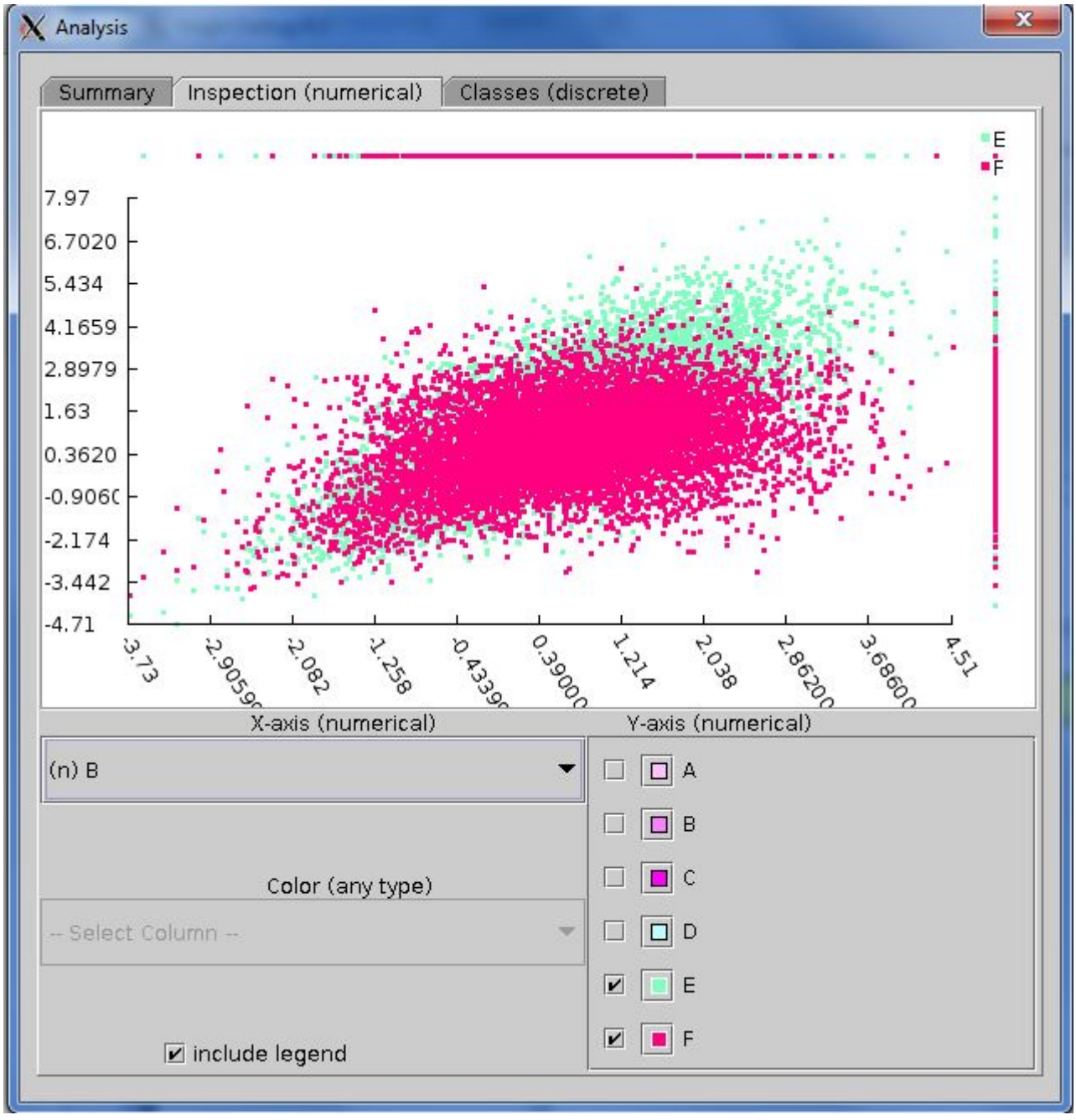


Figure 4.8: Updated interface for plotting data. It is now possible to plot multiple data sets and ordered sequences. It is also possible to customize colors and to include a legend.

5 Task 5.4: Adaptation of Parameters From Data in Dynamic BNs

According to the Gantt diagram of the Description of Work, Task 5.4 is scheduled to start in month 15 and complete by month 30. The work in this task should be coordinated with the work in Task 4.4 on model adaptation. Since Task 5.4 has just started recently there are no software developments to report on. Therefore, the planned work remains unchanged compared to the original plan as outlined in the Description of Work.

6 Summary

This deliverable has provided a summary of the software developments made in Work package 5 covering the first 18 months of the project period as well as outlined a plan for each task for the remaining duration of the project.

The functionality developed in Work package 5 includes parallelization of TAN learning and the PC algorithm (Task 5.1), development of functionality to support the efforts in Work package 6 on improving time and space performance of the solution developed (Task 5.2) and development of functionality mainly in the HUGIN AMIDST GUI to support approximate inference in dynamic Bayesian networks (Task 5.3). Task 5.4 has only recently been initiated and there is no progress to report.

This deliverable does not mark the completion of Work package 5. Further developments are required in all tasks of the Work package. In Task 5.1 the developments include horizontal parallelization. In Task 5.2 future optimizations are dependent on the work performed in Work package 6 on the automotive use-case and detailed planning is therefore not possible. In Task 5.3 the work on implementing the Boyen-Koller algorithm for approximate inference in dynamic Bayesian networks is ongoing and a prototype implementation is expected to be available by the end of 2015. Notice that the Divide and Conquer implementation in Task 5.2 supports the use of the Boyen-Koller principle where the belief state over the interface is approximated as a set of single marginals through the *belief copy* functionality. Finally, the work in Task 5.4 on adaptation of parameters from data in dynamic Bayesian networks has not started yet. This work is to be coordinated with Task 4.4 on Model adaptation.

In conclusion, the work in Work package 5 is on track and the planned results are expected to be achieved in accordance with the plan.

Appendices

A Use-case Requirements

ID	Relevant Subphase	Must/Should/Could	Points	Task
DAI.U2.D2	Framework instantiation	Must	50	
DAI.U2.O1	Production testing	Must	100	5.2-5.4
DAI.U3.D1	Framework instantiation	Must	70	5.3, 5.4
DAI.U3.D2	Framework instantiation	Must	30	5.2-5.4
DAI.U3.O1	Production testing	Must	100	5.2-5.4
DAI.U4.D1	Framework development and instantiation	Should	20	5.3
DAI.U4.D2	Framework development and instantiation	Should	20	5.3
DAI.U4.D3	Framework development	Should	15	5.3
DAI.U4.D4	Framework development	Should	15	5.3
DAI.U4.D5	Framework instantiation	Should	20	5.3
DAI.U4.O1	Production testing	Should	100	5.3
DAI.U6.D4	Framework development	Must	5	
DAI.U6.D5	Framework development	Must	10	
DAI.U6.D6	Framework development	Must	20	5.4
DAI.U6.D10	Framework instantiation	Should	10	5.2-5.4
DAI.U6.D13	Framework development	Must	90	5.2, 5.3
DAI.U6.O1	Interface to existent systems	Must	50	5.2-5.4
DAI.U7.D1	Framework instantiation	Must		5.2
DAI.U7.D2		2) Must	45	5.2, 5.3
DAI.U7.D3		3) Should	10	5.2
DAI.U7.O2	Production testing	Must	40	5.2, 5.3
DAI.U7.O3	Production testing	Must	20	5.2, 5.3
DAI.U7.O4	Production testing	Must	10	5.2
DAI.U8.D1	Framework development	Must	70	5.4
DAI.U8.D2	Testing	Must	30	5.4
DAI.U8.D3	Framework development	Could	60	5.2, 5.4
DAI.U8.D4	Framework development	Could	40	5.2, 5.4
DAI.U8.O1	Production testing	Should	10	5.4
DAI.U8.O2	Production testing	Could	90	5.4

Table A.1: Use-case requirements specific to Work package 5.

A Bibliography

[1] AMIDST Consortium. First report on the use-case based on task 1.3, 2014. Deliverable 6.1 of the AMIDST project, amidst.eu.

[2] AMIDST Consortium. Requirements for the automotive, oil and financial data domains, 2014. Deliverable 1.2 of the AMIDST project, amidst.eu.

[3] AMIDST Consortium. Software package prototype with functionality to support structure learning in parallel, 2014. Deliverable 5.1 of the AMIDST project, amidst.eu.

[4] AMIDST Consortium. First report on the use-case based on task 1.5, 2015. Deliverable 8.1 of the AMIDST project, amidst.eu.

[5] S. Andreassen, F. V. Jensen, S. K. Andersen, B. Falck, U. Kjærulff, M. Woldbye, A. R. Sørensen, A. Rosenfalck, and F. Jensen. MUNIN — an expert EMG assistant. In *Computer-Aided Electromyography and Expert Systems*, chapter 21. Elsevier Science, 1989.

[6] X. Boyen and D. Koller. Tractable inference for complex stochastic processes. In *Proceedings of the Fourteenth Annual Conference on Uncertainty in Artificial Intelligence (UAI-98)*, pages 33–42, 1998.

[7] X. Boyen and D. Koller. Exploiting the architecture of dynamic systems. In *Proceedings of the Sixteenth National Conference on Artificial Intelligence (AAAI-99)*, pages 313–320, 1999.

[8] C. Chow and C. Liu. Approximating discrete probability distributions with dependence trees. *IEEE Transactions on Information Theory*, IT-14(3):462–467, 1968.

[9] T. M. Forum. MPI: A Message Passing Interface, 1993.

[10] N. Friedman, D. Geiger, and M. Goldszmidt. Bayesian network classifiers. *Machine Learning*, pages 1–37, 1997.

[11] U. Kjærulff. A computational scheme for reasoning in dynamic probabilistic networks. In *Proceedings of the Eighth Conference on Uncertainty in Artificial Intelligence*, pages 121–129, San Francisco, California, 1992. Morgan Kaufmann Publishers.

[12] U. Kjærulff. dHugin: a computational system for dynamic time-sliced Bayesian networks. *International Journal of Forecasting*, 11:89–111, 1995.

[13] S. L. Lauritzen and F. Jensen. Stable local computation with conditional Gaussian distributions. *Statistics and Computing*, 11(2):191–203, 2001.

[14] A. Madsen, F. Jensen, U. Kjærulff, and M. Lang. HUGIN - The Tool for Bayesian Networks and Influence Diagrams. *International Journal on Artificial Intelligence Tools 14*, 3:507–543, 2005.

[15] A. Madsen, M. Lang, U. Kjærulff, and F. Jensen. The Hugin Tool for Learning Bayesian Networks. In *Proceedings of the Seventh European Conference on Symbolic and Quantitative Approaches to Reasoning and Uncertainty*, pages 549–605, 2003.

[16] A. L. Madsen, F. Jensen, A. Salmerón, M. Karlsen, H. Langseth, and T. Nielsen. A new method for vertical parallelisation of tan learning based on balanced incomplete block designs. In L. van der Gaag and A. Feelders, editors, *PGM 2014*, volume 8754 of *Lecture Notes in Artificial Intelligence*, pages 206–221. Springer, 2014.

[17] A. L. Madsen, F. Jensen, A. Salmerón, H. Langseth, and T. Nielsen. Parallelization of the PC Algorithm, 2015. Under review.

[18] P. Spirtes, C. Glymour, and R. Scheines. *Causation, Prediction, and Search*. Adaptive Computation and Machine Learning. MIT Press, second edition, 2000.

[19] D. Stinson. *Combinatorial Designs — Constructions and Analysis*. Springer, 2003.