iOOBN: An Object-Oriented Bayesian Network Modelling Framework with Inheritance

Md Samiullah

A dissertation submitted in fulfilment of the requirements for the degree of
Doctor of Philosophy in Information Technology

Faculty of Information Technology
Monash University, Australia

March 2020

Copyright Notice

Declaration

This thesis contains no material which has been accepted for the award of any other degree or diploma at any university or equivalent institution and that, to the best of my knowledge and belief, this thesis contains no material previously published or written by another person, except where due reference is made in the text of the thesis.

Signature:

Print Name: **Md Samiullah**

Date: **March 26, 2020**

Publications during enrolment

- Md Samiullah, Thao Xuan Hoang, David Albrecht, Ann Nicholson, and Kevin Korb. "iOOBN: A Bayesian Network Modelling Tool Using Object-Oriented Bayesian Networks with Inheritance." In 2017 IEEE 29th International Conference on Tools with Artificial Intelligence (ICTAI), pp. 1218-1225. IEEE, 2017.
- Md Samiullah, David Albrecht, and Ann Nicholson. "Supplementary materials: iOOBN framework and case study", http://bayesian-intelligence.com/publications/TR2017_1_iOOBN_Supp.pdf, Bayesian Intelligence, Technical Report TR2017/1, 2017.

This thesis is dedicated to a wonderful and lovely woman:
my better half, *Lioza Noor*

Acknowledgments

I would like to express my sincere gratitude and profound indebtedness to my supervisors Prof. Ann E. Nicholson and Dr David W. Albrecht for their constant guidance, insightful advice, helpful criticism, valuable suggestions, commendable support, and endless patience with the completion of this thesis. I am very proud to have worked with such wonderful mentors.

This research was supported by an Australian Government Research Training Program (RTP) Scholarship. A special thanks to the Australian Government, Monash University and their staff, for their support. I am grateful to Dr Gillian T. Fulcher (PhD in Sociology) for proofreading parts of this thesis and to Valerie Mobley for professional editing of my final draft.

I would like to express my gratitude to four great teachers for shaping my life: MS supervisor Professor Chowdhury Farhan Ahmed (PhD), primary school teacher Mr Foyez Ahmed, secondary school teacher Mr Mohammad Ali, and college teacher Mr MA Jabbar. Also, no words are sufficient to express my gratitude to some more eminent and kind people: Prof. Mir Hasan Ali, Mrs. Hosne Ara, Mr Mir Latafat Ali, Mr Mir Anwar Ali, Mrs. Rowshan Ara, Mr Abul Hashem, Mr Sayem Mahmood, Syed Irfanul Hasan and Syed Rezwanul Hasan. Without their support and guidance, it would not have been possible for me to start or progress my study. I would like to extend my thanks to the wonderful people around me, especially Dr SM Abdullah and Dr Shampa Shahriar, for always being there to help from the beginning of my PhD journey. Of course, I also owe my heartiest gratitude to my family, especially my brothers, my sister and my nephew (Mahin) for encouraging me.

This PhD journey has been challenging. One person who has never let me down, and who has sacrificed the most in this period, is my wife, Lioza Noor. Without her constant support, it would not have been possible for me to complete this PhD. I cannot thank her enough for encouraging and supporting me all along the way. And in times of stress, the presence of an angel, the best gift ever in my life, my daughter Safwana Noor, has given me strength.

No words can express how grateful I am to my parents. They have always supported me, despite all the troubles I have caused them throughout my life.

Finally, I am grateful to Allah (SWT) for His constant grace bestowed upon me.

Abstract

The construction of Bayesian Decision Networks (BNs) to model large-scale real-life problems is challenging. One approach to scaling up is to use Object-Oriented Bayesian Networks (OOBNs). These allow modellers to define classes and construct models with a compositional and hierarchical structure, enabling reuse and also supporting maintenance. In the OO paradigm, a key concept is inheritance, the ability to derive attributes and behaviour from pre-existing resources, which enables an even higher level of reusability and scalability. However, the concept of inheritance in OOBNs has yet to be fully defined and implemented.

In that context, this thesis presents iOOBN, a new framework that provides a fully defined inheritance for OOBNs. The framework allows modelling using an inheritance hierarchy of classes and subclasses, and provides guidance on the modelling process. The study describes a prototype implementation with an existing BN software tool (Hugin), and presents a case study where an existing large, complex, dynamic OOBN is reengineered, thus demonstrating the framework's usefulness in practice.

Reasoning with OOBNs, as with BNs, involves the computational task of inference, that is, the computing of new posterior probability distributions given a set of evidence. A widely used inference technique in ordinary BNs involves compiling the BN into a junction tree (JT) before performing the inference; the compilation step is only performed whenever the network changes. In current OOBN software, the OOBN is first transformed into the underlying BN (so-called flattening), then standard inference is performed. However, there is a method that allows an incremental compilation of BNs, rather than recompiling from scratch after each network modification. This technique can also be applied to OOBNs after flattening. None of the existing techniques can make use of inheritance during the inference process.

Hence, a new incremental compilation technique is proposed that reuses existing compiled JTs of both embedded components and superclasses, and does not require flattening. The technique can reduce compilation time.

In addition, the thesis proposes the first ever approach to learning an iOOBN class hierarchy from a set of BNs or OOBN classes. The approach includes constructing a hierarchical structure based on a supergraph, using the underlying DAGs of the BNs or OOBN classes. The effectiveness of the learning algorithm is also shown in terms of two measures – derivation cost and reusability – via a case study that shows the development of class hierarchies for a real-life project and through the experimentation conducted on some synthetic hierarchies.

Glossary

TEXTBOX 1 (GLOSSARY TABLE (ALPHABETIC))

A.K.A.	**A**lso **K**nown **A**s
ANTLR	**AN**other **T**ool for **L**anguage **R**ecognition
API	**A**pplication **P**rogramming **I**nterface
BDN	**B**ayesian **D**ecision **N**etwork
BAIRG	**BA**yesian **I**ntelligence **R**esearch **G**roup
BN	**B**ayesian **N**etwork
CaMML	**Ca**usal discovery via **M**inimum **M**essage **L**ength
CFG	**C**ontext **F**ree **G**rammar
CPD	**C**onditional **P**robability **D**istribution
CPT	**C**onditional **P**robability **T**able
DAG	**D**irected **A**cyclic **G**raph
DAPER	**D**irected **A**cyclic **P**robabilistic **E**ntity-**R**elationship
DBN	**D**ynamic **B**ayesian **N**etwork
DELWP	**D**epartment of **E**nvironment, **L**and, **W**ater, and **P**lanning
DOOBN	**D**ynamic **O**bject-**O**riented **B**ayesian **N**etwork
DPCD	**D**epartment of **P**lanning and **C**ommunity **D**evelopment
e.g.	**e**xempli **g**ratia (for example)
EPBC	**E**nvironment **P**rotection and **B**iodiversity **C**onservation
FOL	**F**irst **O**rder **L**ogic
GBN	**G**aussian **BN**
GES	**G**reedy **E**quivalence **S**earch
GMRF	**G**aussian **M**arkov **R**andom **F**ield
GUI	**G**raphical **U**ser **I**nterface
HBN	**H**ierarchical **BN**
HDE	**H**ugin **D**ecision **E**ngine
HPC	**H**ybrid **PC**
IC	**I**nductive **C**ausation

InC	**In**cremental **C**ompilation
ID	**I**nfluence **D**iagram
i.e.	**i**d **e**st (in other words)
iOOBN	**i**nheritance in **O**bject-**O**riented **B**ayesian (Decision) **N**etwork
JF	**J**unction **F**orest
JT	**J**unction **T**ree
KEBN	**K**nowledge **E**ngineering in **B**ayesian **N**etworks
LIMID	**LI**mited **M**emory **I**nfluence **D**iagram
LL Parser	**L**eft-to-right, **L**eftmost derivation (a top-down) **Parser**
MaST	**Ma**ximum-weight **S**panning **T**ree
MCMC	**M**arkov **C**hain **M**onte **C**arlo
MDL	**M**inimum **D**escription **L**ength
MEBN	**M**ulti-**E**ntity **BN**
ML	**M**achine **L**earning
MML	**M**inimum **M**essage **L**ength
MPS	**M**aximum **P**rime **S**ubgraph
MPSD	**M**aximum **P**rime **S**ub-graph **D**ecomposition
ms	**m**illi **s**econd
MSA	**M**elbourne **S**trategic **A**ssessment
MSBN	**M**ultiply **S**ectioned **BN**
MST	**M**inimum (weight) **S**panning **T**ree
NA	**N**ot **A**pplicable
NOC	**N**umber **o**f (foreign) **C**lasses that are being instantiated in a class
NON	**N**umber **o**f **N**odes in a class
NOO	**N**umber **o**f **O**bjects created for each foreign class
NOP	**N**umber **o**f maximum **P**arents per node in a class
NOS	**N**umber **o**f **S**tates per node in a class
NOPAvg	**N**umber **o**f **P**arents (**Av**era**g**e) in the nodes of a class
NP-Hard	**N**on-deterministic **P**olynomial-time **Hard**ness
NPP	**N**et **P**lus **P**lus
OO	**O**bject-**O**riented
OOBN	**O**bject-**O**riented **B**ayesian (Decision) **N**etwork
OODAG	An **OO**BN class's **D**irected **A**cyclic **G**raph
OOPRM	**O**bject-**O**riented **P**robabilistic **R**elation **M**odel

Par	**Par**ent
PC	**P**rototypical **C**onstraint-based algorithm
PEP	**P**EP is an **E**arley recursive **P**arser
PGM	**P**robabilistic **G**raphical **M**odel
PRM	**P**robabilistic **R**elation **M**odel
RBN	**R**elational **BN**
R&D	**R**esearch and **D**evelopment
SIIC	**S**hareable **I**nheritable **I**ncremental **C**ompilation
SIIC#	Special **SIIC** (that uses existing JTs)
TAN	**T**ree **A**ugmented **N**aive Bayes
UML	**U**nified **M**odelling **L**anguage
UOM	**U**nit **O**f **M**easurement
WGR	**W**estern **G**rassland **R**eserve
w.r.t.	**w**ith **r**espect **t**o

Symbols and Notations

TEXTBOX 2 (NOTATION TABLE)

$A \uplus B$	Disjoint union of A and B, i.e, $A \cap B = \varnothing$
$BN.N$	Set of vertices of $BN = < N, E, \Pi >$
$BN.E$	Set of edges of $BN = < N, E, \Pi >$
$BN.\Pi$	Set of parameters of $BN = < N, E, \Pi >$
$\mathbb{C}$	An iOOBN class
$\mathbb{C}^I$	An instance or replica of an iOOBN class $\mathbb{C}$
CG	Clique Graph
Clq	Clique
$cost$	Hierarchy construction cost
$cost_r$	Hierarchy construction cost ratio
δ	Derivation cost
δ_{ch}	Child adding cost
δ_r	Derivation cost ratio
E	Set of edges
$\mathbb{E}$	Set of edges of an iOOBN class
E_c	Set of causal edges of an iOOBN class
E_r	Set of referential edges of an iOOBN class
E_{info}	Set of information links of an iOOBN class
E_{pre}	Set of precedence links of an iOOBN class
G	DAG representing an ordinary BN
$\|G\|$	Size of a graph G
G_{com}	A common subgraph
G_C	DAG of class C
G_{mc}	A maximal common subgraph
G^r	A residual graph of a graph G
$\mathbb{G}$	A set of graphs
$\mathbb{I}$	An iOOBN interface
L_c	Maximal proper subset

$lab(n)$	The label of the node n
N	Set of vertices of an ordinary Bayesian Network $< N, E, \Pi >$
$\|N\|$	Size of N
$\mathbb{N}$	Set of Nodes of an iOOBN class
N_{In}	Set of input nodes of an iOOBN class
N_{Out}	Set of output nodes of an iOOBN class
N_{Emb}	Set of embedded nodes of an iOOBN class
N_{In}^{C}	A set of input causal nodes
N_{Out}^{C}	A set of output causal nodes
N_{Em}^{C}	A set of embedded causal nodes
N_{In}^{D}	A set of input decision nodes
N_{Em}^{U}	A set of embedded utility nodes
$n_i \rightarrow n_j$	A causal edge from the node n_i to the node n_j
$n_i \nleftrightarrow n_j$	A Referential edge between nodes n_i and n_j
$n_i \dashrightarrow n_j$	Information or precedence links between n_i and n_j
$par(n)$	A parent node of the node n
$\mathbb{O}$	Set of embedded objects of an iOOBN class
Π	Parameters of a BN or of a class, i.e. a set of CPTs/CPDs
ρ	Reusability
ρ_r	Reusability ratio
$\mathbb{S}$	A set of subsets
$\top$	Hierarchy Tree
$\mathcal{T}$	A Label Hierarchy Tree

Contents

List of Figures

List of Tables

List of Algorithms

Introduction

Our daily life is full of challenges, and the biggest challenge is the unpredictability of many of our significant life events. To deal with this unpredictability, analysing the probability of events has become very important. In particular, the theorem of English statistician *Thomas Bayes* has been revolutionary. Numerous theories and techniques have been proposed, and many tools have been developed to solve real-life problems based on the theorem, yet it is still very much an area of active research. It still attracts researchers dealing with cutting-edge technologies. One tool that has been used extensively in modelling probabilistic analysis for decades is the Probabilistic Graphical Model (PGM). This chapter briefly introduces the Bayesian network (BN) as a PGM, and describes its various aspects. The chapter outlines factors that motivated the research, the objective of this research, and finally describes the organisation of the thesis.

1.1 Probabilistic Graphical Models and BNs

PGMs are one of the classes of probabilistic models where a graph represents the structure of the conditional dependence between random variables. Bayesian decision networks (BNs), one of the widely used PGMs, have been used for reasoning under uncertainty for decades [8,9]. Reasoning is a critical process in real-life applications because of the complex nature and variety of the applications and multiple challenges of uncertainty associated with the events of the applications. PGMs are very effective in dealing with various decision-making challenges due to their ability to present causal relationships among random variables, incorporating dynamic information from a variety of areas with varying degrees of uncertainty, and to perform several kinds of reasoning (e.g., predictive, diagnostic, and intercausal). Moreover, the graphical visualization of PGMs helps specialists with diverse expertise to cooperate easily.

Among the many families of PGMs, one of the most widely used and demanding is the Bayesian network (BN) [10,11]. A BN is a probabilistic graphical model that represents causal relationships using a DAG and which supports the reasoning for decision-making under uncertainty. In real-world applications, the ability to reason under uncertainty is critical for mak-

ing decisions. BNs [12, 13] are a powerful tool for performing and supporting many forms of uncertain reasoning, including monitoring, prediction, diagnosis, risk assessment and decision support. Their usefulness as a mature modelling technology is demonstrated by the extremely wide range of areas to which they have been applied, including (with single examples only) medicine [14], education [15], agriculture [16], ecology and environmental management [17], biosecurity [18], surveillance [19], the military [20], weather forecasting [21] and software engineering [22].

A BN [1, 12] is a directed acyclic graph (DAG) whose nodes represent the random variables in the problem. A set of directed edges connects pairs of nodes, representing the direct dependencies (which are often causal connections) between variables. The set of variables pointing to a node X are called its parents and the relationship between variables is quantified by conditional probability tables (CPTs) associated with each node. The CPTs together compactly represent the full joint distribution. Users can set the values of any combination of nodes in the network that they have observed as evidence. This evidence, e, propagates through the network, producing a new posterior probability distribution $P(X|e)$ for each node X in the network. There are several efficient algorithms for performing this probabilistic updating [1], providing a powerful combination of predictive, diagnostic and explanatory reasoning.

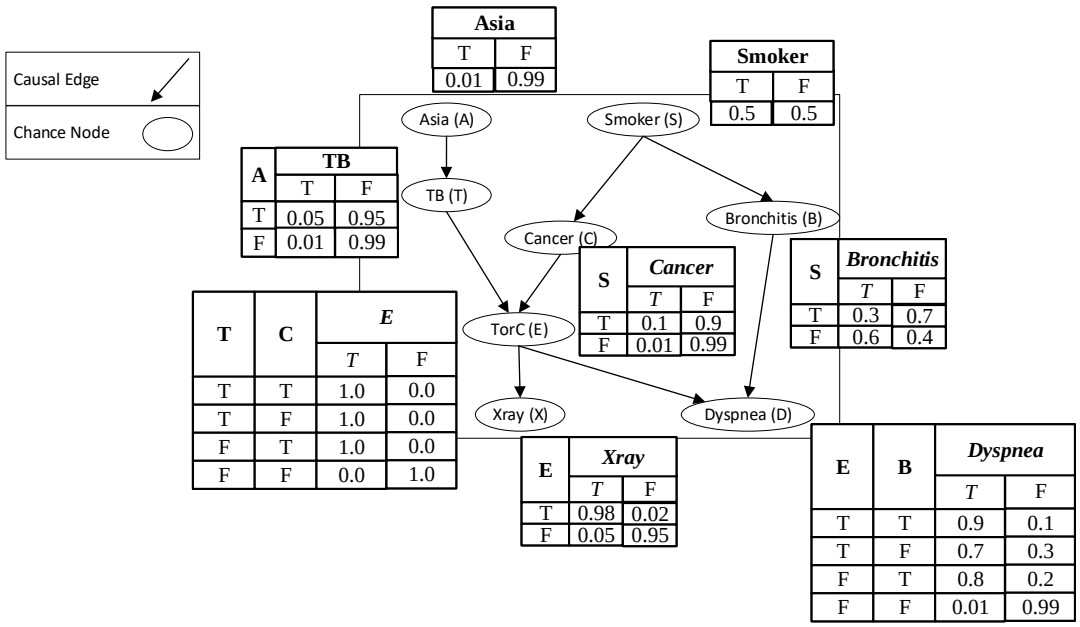


Figure 1.1: An ordinary BN : the classic "Chest Clinic" (A.K.A. "Asia BN") example

Figure 1.1 represents a well-known and popular "Chest Clinic BN" A.K.A. "Asia BN" [23] example. Note that there are eight nodes and eight directed edges. The eight nodes are: "A" to represent whether or not a patient has visited Asia, "S", "T", "B", "C", "X" and "D" to repre-

[1] Also known as compilation or probability propagation or conditioning or belief updating [1]

sent, respectively, whether or not the patient is a smoker, affected by tuberculosis, bronchitis, cancer, whether the Xray is positive, and whether the patient suffers from dyspnea (shortness of breath). This BN and inference in the BN helps in diagnosing a patient, i.e., without any evidence, probability of each disease is P(**B**=T) = 45%, P(**T**=T) = 1.04%, and P(**C**=T) = 5.5%. An evidence on the patient being a smoker (prediction) changes disease probabilities to P(**B**=T | **S**=T) = 30%, P(**T**=T | **S**=T) = 1.04%, and P(**C**=T | **S**=T) = 10% and also the probability of having dyspnea is P(**D**=T | **S**=T) = 30.31% (note that before entering the evidence on "S", the chance of having dyspnea was 39.7%). If patient then presents with shortness of breath, disease probabilities go up, namely P(**B**=T | **S**=T, **D**=T) = 80.26%, P(**T**=T | **S**=T, **D**=T) = 2.61%, and P(**C**=T | **S**=T, **D**=T) = 25.07%, but if Xray result comes back negative, then the probability of T and C go down, B goes up further, namely P(**B**=T | **S**=T, **D**=T, **X**=F) = 96.68%, P(**T**=T | **S**=T, **D**=T, **X**=F) = 0.08%, and P(**C**=T | **S**=T, **D**=T, **X**=F) = 0.72%.

There are many methods proposed for BN inference (exact and approximate), all of which are NP-Hard [24]. Among the exact inference techniques, one approach is compilation, of which 'JT-based Inference' [25] is the most widely used approach. The steps in this approach include moralisation, triangulation, clique finding, clique graph formation, and junction tree (JT) construction. However, if this traditional approach is used, given any change in the BN structure, all the operations related to 'JT-based inference' need to be revisited. These operations, especially triangulation and clique finding, are computationally expensive [26]. One approach to addressing this problem is Flores et al.'s [2,4] idea of incremental compilation.

BNs have been widely used in various application areas. Such widespread use is possible due to the development of BN software tools that have allowed technologists and modellers to build, edit, assess and deploy BNs using graphical user interfaces (GUIs) that promote ease of use. Popular tools these days include Hugin [2] [27], GeNie [3] [28], Netica [4] [29], AgenaRisk [5] and BayesiaLab [6]. While these tools remain popular, the well-known 'knowledge bottleneck' (see Section 2.8 of Chapter 2) problem has slowed and limited their further adoption. In response to this problem, machine learning techniques were invented in order to generate BNs from observational data. Some of the methods were developed and adopted for the causal discovery of special kinds of BNs, i.e., causal BNs (CBNs), so called because the arcs in the network are used for only causal interpretation, whereas BNs more generally also model non-causal associations. This subfield also enjoyed much success in the 1990s, with many of its algorithms incorporated into BN tools in the 2000s. Popular causal discovery algorithms and techniques

[2] **Hugin:** https://www.hugin.com/
[3] **GeNie:** https://www.bayesfusion.com/
[4] **Netica:** https://www.norsys.com/netica.html
[5] **AgenaRisk:** https://www.agenarisk.com/
[6] **BayesiaLab:** https://www.bayesialab.com/

include the PC algorithm [30] of the Tetrad research group (the most widely adopted algorithm for causal structure discovery, probably because it is the simplest and easiest to understand), and the BDe/BGe metrics developed by Microsoft researchers [31].

In the absence of data, BNs can be built through elicitation of domain-specific knowledge from experts, and expert elicitation and causal discovery can be combined in so-called "knowledge engineering in Bayesian networks" (KEBN). Various methodologies for KEBNs have been proposed by researchers (see more in Section 2.8 of Chapter 2). Key concepts include building BNs iteratively and incrementally [1, 32, 33]; breaking down complex models into submodels [29] or fragments [32]; and building BNs with common structures or elements called "idioms" [34].

BNs (and PGMs more generally) enable us to document a problem and its current state of knowledge, characterizing the overall model of the problem, as well as behaving like a storehouse of knowledge [35]. The process of creating a BN helps clarify assumptions and identify uncertainties within the system [33]. As information expands, new data can be added to improve the model.

However, many real-life applications are large and complex, and modelling them with BN is a difficult process. These difficulties include construction cost, which increases significantly with the BN's size; inadequate, inappropriate, and incomplete data is not feasible for some domains and applications; construction and parameterization of a BN requires significant input (time and effort) for elicitation and validation. Finally, constructing BNs incrementally is difficult because to change or add something to an existing BN requires the knowledge of the domain and the network as a whole. These are issues of compatibility – that is, the reuse of existing BNs or any segment of a BN (as well as the reuse of any inference outcome) is anything but straightforward. Section 2.3.4 describes these issues in more detail.

1.2 Object-Oriented Bayesian Network and Classes

From the late 1990s, researchers started to develop theories and techniques to overcome the challenges involved in knowledge engineering BNs. These techniques include object-oriented BNs [36, 37], generalized decision graph [38]; BN fragments [32]; and various techniques combining probabilistic relational models and objects, such as module networks [39], probabilistic relational models and plate models [40], multi-entity BNs (MEBNs) [41], Multiply Sectioned BNs (MSBNs) [42], Idioms [34], and Templates [43]. The features, merits and disadvantages of the above-mentioned approaches are outlined in Chapter 2.

In this thesis, the focus is on the approach inspired by the Object-Oriented (OO) principle [44, 45] from the field of software engineering. This has been useful for dealing with large-

scale problems, such as those seen in Object-Oriented Bayesian networks (OOBNs) [46]. The principle allows modellers to encapsulate BNs into classes and construct objects defined with attributes of similar characteristics, and build more complex models in a compositional hierarchical way, and supporting re-use of components. Due to the ever-growing complexity of constructing BNs, scalability and reusability are potentially useful advantages when constructing any large BN to use in real-life applications.

OOBNs introduce techniques to scale-up by allowing the reuse of existing components that can be connected together hierarchically and compositionally via formally defined interfaces. OOBN exhibits all of these features: **Encapsulation**, **Abstraction**, **Inheritance** and **Polymorphism**. A segment of a BN is encapsulated into a class (a blueprint of an object) with an interface of input and output nodes. A more complete and formal presentation of OOBNs, is given in Chapter 2. Figure 1.2 (a) shows how the Asia BN can be converted into an OOBN class, with "A" and S being the input nodes, and the three disease nodes ("T","C" and "B") being the output nodes. That segment can be used in other models and in larger models by making a copy (called an instance) and adding connections only to the interface node, via connections only to the input and output nodes. Such reuse helps save design time and reduce complexity. Figure 1.2 (b) shows the Asia BN being used in a larger model focusing on assessing treatment. It also supports maintenance, as any changes to a class (for example, if the CPTs are updated using new data) can be automatically applied to all the instances and its subclasses. There are many potential advantages in using OOBNs instead of ordinary BNs. A non-exhaustive list follows:

1. Reusability of tested and correctly functioning components avoids the complexity of reconstruction/design and correctness testing. That is, it avoids the tendency of "reinventing the wheel".

2. Abstracting/hiding the internal complexity of a class and providing a simpler interface to its users.

3. Avoiding redundancy (with respect to reusability) provides scalability.

4. Ensuring flexibility in modelling and strong type checking to avoid unintentional flaws in modelling.

5. Accommodating the design of large BNs with fewer flaws [7] making it easy to both maintain the existing model and manage changes [8].

[7] In the examples of Chapter 3, it is shown that how OOBN and iOOBN allow designing large scale BNs using OO principles with fewer flaws in comparison to ordinary BNs.

[8] OO principles allow designing applications in simple, reusable and unique segments or modules that reduce

6. OOBNs provide better handling of errors. (At any point of time, a modeller has to deal with a particular module of the model. This makes the error handling easier since they can check if a particular class is functioning correctly or not. If a class works well and produces correct output then the modeller need not to check for other instances of this class but need to check other classes only or their connections for possible errors.)
7. The interface structure facilitates parallel design and makes it easier to design and integrate large and complex systems with better control than the ordinary BN allows.
8. Includes modularity to accommodate dynamic domains.
9. Facilitates both spatial and temporal design.
10. Allows specification of template models in an easy and intuitive way.
11. Allows sharing and reusing of JT structures among subclasses and instances. This facilitates solving the greatest challenge of inference for BNs.

Note: while OO is very widely used in software engineering, there is little formal evidence to support some of the modelling claims coming from the software engineering field [47–49]. However, in sections 3.3.3, 3.3.4 and 3.4 of Chapter 3, how OO properties and features can add advantages in BN modelling, are described with some example scenarios.

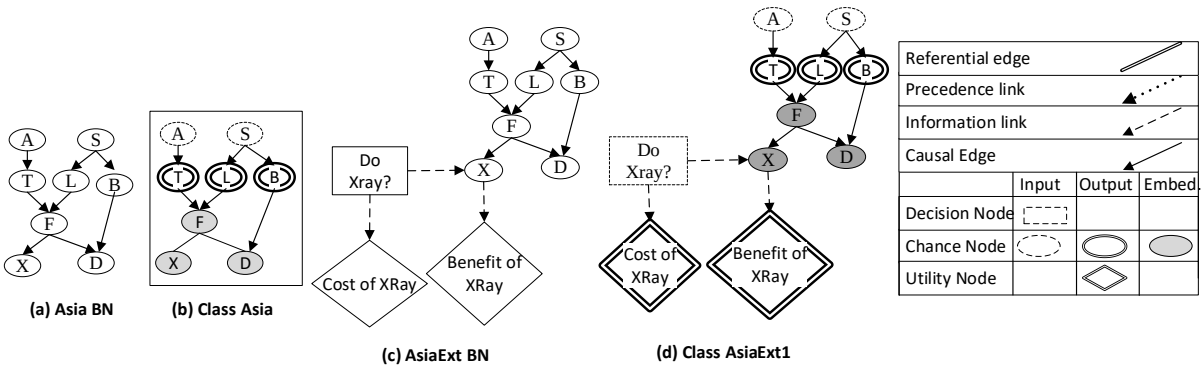


Figure 1.2: (a) Asia OOBN class and (b) Asia class object embedded in larger patient model. Note that input nodes are dashed, output nodes are double-lined, with connections limited to input and output nodes.

1.2.1 Limitations of existing OOBN frameworks

There are some noteworthy limitations in existing state-of-the-art OOBN frameworks definitions. A few of them are listed as follows:

complexity and, in turn, flaws. Moreover, reusability of classes by inheritance allows extending a working and flaw-less class to construct a new class with fewer flaws, as only potential scope of incorporating flaws is in the new elements.

- A complete OO-definition is missing; for example, inheritance is not fully defined, and polymorphism is not defined at all.
- The OOBN frameworks proposed by Koller [36] is not able to deal with "Time slice representation".
- No dedicated compilation technique is proposed. Performing inference in OOBNs is not straightforward in the existing OOBN frameworks. Thus, a complicated time-critical operation like inference suffers a lot (slow performance and high expense) due to the need for intermediate transformation into an ordinary BN and construction of a JT structure from scratch.
- Definition of the "object" and its characteristics are not explored seamlessly in any of the existing OOBNs.
- The framework defined in [36] does not account for the dynamic behaviour of an object [50] and the framework in [37, 51] is limited to only dealing with the change of interface and change of an intermediate node or link. Moreover, changing of node types, states, CPTs or attributes is not considered.
- OOBN structure learning from sample data has not been proposed to date, so this could have implications if the OOBN is widely adopted by technologists/tech-people.
- Type checking and typecasting are not defined in any of the existing formulations of OOBNs yet.

1.3 Contributions of the thesis

The main contribution of this research is the development of an OOBN theoretical framework, iOOBN, that overcomes the limitations of existing OOBN implementations by providing all the key OO principles of inheritance, encapsulation, polymorphism, and abstraction. The research provides the following attributes:

1. It develops a theoretical framework that gives more effective treatment of inheritance and other important OO features such as encapsulation, polymorphism, abstraction, type checking and typecasting by
    - demonstrating the modelling power of the framework via a set of case-studies (from the literature, and a real-world DOOBN project, i.e., Western Grassland Reserve (WGR)), reengineering existing OOBNs into the iOOBN framework; and

- implementing a prototype version of the iOOBN framework.

2. It provides a new incremental (as well as shareable–inheritable) compilation algorithm for iOOBN inference, which provides computational efficiencies compared to current methods.

3. It includes a new algorithm for learning an iOOBN inheritance hierarchy from a set of BNs and OOBNs, and proposes novel measures to compute the reusability and efficiency of different class hierarchies found in different modelling solutions.

1.3.1 The iOOBN framework

BNs are the most popular PGMs [8] because of their numerous advantages for modelling real-life problems and applications. However, their construction to model large-scale real-world problems is challenging. One suitable approach for dealing with large-scale problems is the use of Object-Oriented Bayesian networks (OOBNs).

To the best of our knowledge, the notion of the Object-Oriented principle, especially with regard to inheritance (a powerful feature that allows maximum reuse of existing resources), is neither defined fully nor implemented as yet in the context of BNs or OOBNs. Other essential features include polymorphism, type checking, typecasting, and abstraction.

Inheritance is the ability to derive attributes and behaviour from pre-existing classes. This feature allows the deriving of new classes by adding additional required attributes to the existing classes. Thus, it allows sharing of an already constructed network segment, thus avoiding the re-computation of an existing segment. Most of the new classes do not need to be built or defined from scratch. Hence, inheritance plays a substantial role in achieving the goal of scalability for real-life applications by allowing maximum reusability.

Encapsulation helps modellers protect the components of BNs (by encapsulating a segment of the BN into a class), protecting it from unauthorized [9] access. It provides access to the components through interface (input and output) nodes only. As well, the data abstraction feature allows using a segment of BN (i.e., a class and its objects) without knowing the detail of its implementation. Classes and objects help in dividing a large BN into smaller parts (i.e. classes) to be combined later by compositional modelling. The classes can be reused, in the form of an object, as often as required. This also adds another level of reusability. (These points are illustrated in Section 3.5.2 in the reengineering of WGR DOOBN.)

The comprehensive power of OO have not been delineated in existing definitions of

[9] The security is about protecting the integrity of a BN submodel from accidental damage caused by modellers. This might happen for various reasons such as lack of coordination, mismanagement of the repository, and lack of proper knowledge of a large system.

OOBNs. This was the motivation to develop a theoretical framework which has most of the OO features (encapsulation, inheritance, polymorphism, abstraction, type checking, and typecasting), and which also facilitates almost all the benefits of the OO paradigm. (See the benefits outlined in sections 1.2 and 2.5.2). Hence, in this research, the principles of OO has been defined and implemented in order to illustrate its advantages of scalability, flexibility and robustness in the construction of BNs for large and complex real-life applications.

1.3.2 Inference in iOOBN without flattening

In the existing OOBN frameworks [36, 52], dealing with inference is not straightforward. An OOBN needs to be converted implicitly into an ordinary BN in order to perform inference. This approach is not efficient enough, especially when converting to a BN is a complex and time-consuming task. A technique that can circumvent the intermediate construction of an ordinary BN significantly contributes to reducing the computational complexity.

To date, in the literature of BN compilation, Bangsø et al. [5] proposed an incremental compilation technique where flattening of an OOBN for modification of the OOBN, after building it once, is not required. However, to the best of our knowledge, the incremental compilation technique of BNs has been implemented only in [53] and has never been widely used. Besides, it is overly complicated (it is necessary to create and maintain parallel structures. That means, maintaining Maximal Prime Subgraph (MPS) decomposition and the MPS tree along with maximal weight clique tree. Performing this requires operations such as clique joining, marking and replacing the affected portion of the JT, repeatedly applying various complex (see Section 4.4 of Chapter 4) and expensive operations (those involved in JT creation). Often the overheads of incremental compilation are the same as for conventional compilation technique (see also Section 4.3.4.1 in Chapter 4). While these do apply to OOBNs, flattening is still required in the beginning. A point worth noting is that the original Incremental compilation technique [4] was proposed for the ordinary BN and its OOBN version [6] did not explain all aspects of OOBNs, such as how to deal with multiple levels of embedding [10]. Moreover, compiling OOBNs without flattening or compiling iOOBNs (OOBNs with inheritance), as proposed in Chapter 3, has not yet been proposed. To the best of our knowledge, no compilation technique has been proposed as yet that can utilize the compilation structure of a superclass or utilize the JTs of embedded objects in a class in order to build the JT of the class. Hence, the development of a compilation technique (in the iOOBN) that offers all the features mentioned above is the novel contribution of this thesis.

[10] Multiple level embedding occurs where a class is embedded in the form of an object within another class and so on. Each embedding represents a level of embedding; hence, if a class is embedded in another class and the latter class is embedded in another class, the level of embedding used is two.

1.3.3 Learning hierarchies of BNs or OOBN classes

Causal discovery, especially one which is automated, is not an easy task [54, 55]. Chickering [56] proved that finding the best causal model from the super-exponential search space of all DAGs is NP-Hard. There are many hidden factors associated with this which are complex. Nevertheless, experts and researchers proposed and developed various mechanisms to produce a solution. A few of these provided heuristic-based methods, and many provided various measures to compute, compare and choose the best model efficiently by avoiding exponential calculation. Likewise, machine learning techniques to find the best models are also prevalent in this arena. However, for OOBNs, there have been no such techniques proposed to date, perhaps because the OOBN is a relatively new extension and not yet so widely used. In this thesis, an approach to the learning hierarchy of OOBN classes from a set of BNs or a set of OOBN classes is proposed. The algorithm builds/learns the hierarchical relations between the components of the existing systems. Then it searches for structures of classes, subclasses and superclasses. The hierarchy provides guidance on how to reengineer or transform the existing systems. This algorithm can also be viewed as an intermediate step of learning OOBN causal structures from data, leading to the automation of causal structure learning of OOBN classes from data.

1.4 Thesis Organisation

This chapter introduces the thesis, the motivation for the research, and the main contribution of the project as well as providing a brief reference to the related literature. The rest of the thesis is organized as follows:

- **Chapter 2** presents, with discussion, relevant previous research to provide the necessary background to the research presented in the remainder of this thesis.
- **Chapter 3** proposes the iOOBN framework with all the necessary definitions and description of the framework, together with a set of case-studies (from the literature and including a real-world example, WGR [57]). It also contains the development architecture of a prototype version of the proposed iOOBN framework (Appendix A).
- **Chapter 4** presents and analyses the Shareable Inheritable Incremental Compilation (SIIC) technique.
- **Chapter 5** presents a new algorithm to learn an iOOBN inheritance hierarchy from a set of BNs and OOBN classes, together with analysis of its performance on artificial examples as well as the WGR real-world example.

- Finally, the thesis concludes in **Chapter 6** with a summary of the research contributions along with suggesting future directions to enhance and extend this field of research.

Background

Probability plays a very significant role in our daily life. One of the biggest challenges is the uncertainty associated with everyday events, that is, the outcomes of most events that occur are not guaranteed. Probabilistic analysis is the best way to deal with such challenges. In order to performing such analysis, there are some convenient probabilistic models (PMs) and tools. This chapter introduces these models: how they help in analysing probability, how they work, their efficiency, and also their limitations.

This chapter offers a historical perspective of probability and points to related works in the literature relevant to the thesis. It starts with a chronological account: of the necessity of probabilities, how the theory of probability evolved, and how such evolution has led to the development of models for probabilistic analysis. There have been various models proposed by researchers at different times to address different issues and with a particular goal in mind. The models differ in nature, policy, procedure and applicability. Each model has some specific advantages and also limitations. This chapter provides a classification of the models in terms of their features, policy, nature, and behaviour. It then goes into the detail of some of the most relevant models.

Next, the most essential operation in a probabilistic model (i.e. inference), is discussed. It is followed by sections that discuss how a model uses probability analysis and how it deals with the uncertainty of various real-life problems. The final section discusses KEBN in association with automated learning in BNs and OOBNs, the learning hierarchy of classes, the importance of hierarchies, how these have been serving the purpose of science and technology from ancient times, how they play an essential role in probabilistic analysis and what roles they play in various sectors of science and technology.

2.1 Probability, Decision and Utility Theory

The idea of probability has been known about for hundreds of years and used to measure the uncertainty of various facts. The application of probability is widespread in areas like

medicine, forecasting, law, gambling, farming, business, and education [58].

From the dawn of civilisation, people have had to deal with uncertainties of weather, food supply, and various other aspects of their environment. They have striven to mitigate the uncertainty and its effects [59,60]. Even the practice of gambling dates back to about the year 3500 BC, with games of chance being played with bone objects. It could be considered as a precursor of playing with dice, which was apparently prevalent in ancient Egypt and elsewhere.

It is widely accepted that the mathematical theory of probability was first set out by the French mathematician Blaise Pascal (1623–1662) and Pierre Fermat (1601–1665), when they succeeded in deriving exact probabilities for specific gambling problems involving dice. However, numerical probabilities of various dice combinations had been calculated previously by Girolamo Cardano (1501–1576) and Galileo Galilei (1564–1642) [61].

Since the seventeenth century, the theory of probability has been further developed and has been widely applied in diverse fields of study [59]. Nowadays, probability theory is an essential tool in most areas of engineering, science, and management. Researchers are actively engaged in the discovery and development of new tools and applications of probabilities in disparate fields such as medicine, meteorology, information from satellites, marketing, earthquake prediction, human behaviour analysis, the design and development of computer systems, finance, genetics, law, farming, vegetation, weather forecasting, politics, election outcome prediction, wars, data and network security, business, education, and research.

2.1.1 Bayes Theorem

Bayes Theorem [62] is a noncontroversial theorem of probability calculus [1]. This theorem was developed, in the context of gambling, in 1763 by Thomas Bayes, an English statistician and philosopher. It has found ubiquitous application in the Bayesian statistics, machine learning, data mining, bioinformatics and almost all contemporary research fields. Bayes theorem is expressed as:

$$P(hypothesis \mid evidence) = \frac{P(evidence \mid hypothesis) \times P(hypothesis)}{P(evidence)}$$

The likelihood of a "hypothesis" with respect to a piece of "evidence" is the ratio of the joint probability of both "hypothesis" and "evidence", and the prior probability of "evidence".

For example, an "Xray" test of an Asian patient, who is most likely suffering from "tuberculosis", is found to be positive. However, before advising any medication for "tuberculosis" to the patient, the doctor decides to conduct some analysis on the previous outcomes of the "Xray" test and its success in correctly indicating the presence of "tuberculosis". The doctor has come to a conclusion from their analysis that the false positive rate of "Xray" is 5% (i.e., 1

in 20 healthy, that is, not suffering from tuberculosis) people in Asia are wrongly diagnosed with "tuberculosis") and a true positive rate (test was successful in correctly indicating tuberculosis) is 98%. The analysis also showed that, in general, 40% of people in Asia suffer from "tuberculosis". The outcomes of the analysis can be summarized as follows:

$$
\begin{aligned}
&P(Xray|T) = 0.98 \\
&P(Xray|\neg T) = 0.05 \\
&P(TB) = 0.4 \\
&P(Xray) = P(T) \times P(Xray|TB) + P(1-T) \times P(Xray|\neg T) \\
&\qquad = 0.4 \times 0.98 + 0.6 \times 0.05 = 0.422
\end{aligned}
$$

On the basis of the aforementioned prior probabilities, the doctor calculated the likelihood of the patient being infected with "tuberculosis", given a positive "Xray" result, as follows:

$$
\begin{aligned}
P(T \mid Xray) &= \frac{P(Xray \mid T) \times P(T)}{P(Xray)} \\
&= \frac{0.98 \times 0.4}{0.422} \\
&= 0.93
\end{aligned}
$$

The likelihood of the patient being infected with "tuberculosis" is thus very high and suggests the doctor can be confident in advising the proper treatment for the disease.

2.1.2 Decision theory

When working with real-life applications that have associated uncertainty, probabilistic analysis on its own is not enough to make a proper and optimal decision. Along with probabilistic analysis, sound decision making is also important. The decision may involve finding the most suitable action from a set of available actions and while doing so, considering the preferences between the possible outcomes of all the available actions. In dealing with such complexity, Frank Ramsey's **Utility function** [63] offers a particular utility (or value) for each possible situation, to finally select the most appropriate action [1, 33]. A utility function can estimate the priority that reflects desirability or expectation of the available outcomes by associating or mapping them with numeric values. This mapping offers the facility of combining utility theory with probability theory and points to a decision on which action maximizes the profit (or minimises loss).

Say, for an action A, regarding all possible outcomes O_i, the utility function $U(O_i|A)$ is

available. To evaluate the action A in terms of its **Expected Utility** (or, the probability-weighted average utility), the following formula can be used.

$$EU(A) = \sum_i U(O_i|A) \times P(O_i|A)$$

As an example, consider the Asian patient example again, but now with some extra utilities and actions. Suppose, the doctor has two possible actions, namely advising the patient to "Go for Xray" or "Do not go for Xray". In this case, a utility function need to be considered, namely "Benefit of the Xray". Now, let us assume the following mapping/association of values for the utility, $U(Xray, Action)$:

$$Xray = "pos", Action = "Go\ for\ Xray", Benefit = 80$$
$$Xray = "neg", Action = "Go\ for\ Xray", Benefit = -20$$
$$Xray = "pos", Action = "Do\ not\ go\ for\ Xray", Benefit = -60$$
$$Xray = "neg", Action = "Do\ not\ go\ for\ Xray", Benefit = 0$$

Hence, according to the above equation,

$$\begin{aligned} EU("Go\ for\ Xray") &= P(Xray = "pos") \times U(Xray = pos|"Go\ for\ Xray") \\ &+ P(Xray = "neg") \times U(Xray = pos|"Go\ for\ Xray") \\ &= 0.422 \times 80 + 0.578 \times -20 \\ &= 33.76 - 11.56 = 22.2 \end{aligned}$$

$$\begin{aligned} EU("Do\ not\ go\ for\ Xray") &= P(Xray = "pos") \times U(Xray = pos|"Do\ not\ go\ for\ Xray") \\ &+ P(Xray = "neg") \times U(Xray = pos|"Do\ not\ go\ for\ Xray") \\ &= 0.422 \times -60 + 0.578 \times 0 = -25.32 \end{aligned}$$

where,

$$\begin{aligned} P(Xray = "pos") &= P(Xray = "pos"|T = "yes") \times P(T = "yes") \\ &+ P(Xray = "pos"|T = "no") \times P(T = "no") \\ &= 0.98 \times 0.4 + 0.05 \times 0.6 = 0.422 \end{aligned}$$

$$P(Xray = "neg") = 1 - P(Xray = "pos") = 0.578$$

If no other information or special condition is provided, based on the above utility values, the decision of the doctor should be to advise the patient to go for an Xray.

2.2 Probabilistic Models: An Overview

In order to perform probabilistic analysis, Probabilistic Models (PMs) are used as a prominent tool. While developing various tools to assist modellers with flexibility and ease of representation, researchers developed PMs, namely a mathematical representation of a random phenomenon defined by its sample space and the events within, and probabilities associated with each event [64]. PMs can incorporate probability distributions with random variables, where random variables represent the potential outcomes of an uncertain event. In a dice game, the probability of getting a "6" before throwing the dice can be represented using a variable. Probability distributions assign probabilities to the potential outcomes of the associated events. If the dice is fair, then all six sides have the same probability, that is, $\frac{1}{6}$. Assigning this value to all six variables representing the probability of getting a particular dice face is an example of a probability distribution.

The role of PMs in decision making is to acknowledge the associated uncertainty of the inputs and outputs. That means, in some complex applications, that even the input-taking process and the generated outputs may have uncertainty associated with them. Using a PM allows us to be able to formulate a new model to be more relevant and more appropriate for the complicated situation. The key feature of a PM is that it incorporates uncertainty explicitly in order to understand and quantify risk and to make better management decisions. There are many PMs and different people classify them differently. A snapshot of the taxonomy of PMs is shown in Figure 2.1 based on summaries in [43, 65, 66].

In order to perform probabilistic analysis, a suitable model is chosen that best represents the problem and then, in order to get the expected outcome from the model, a special operation called 'Inference' is performed. Inference in a PM calculates the final outcome for the application using the defined approach for the particular model.

A model that expresses the conditional dependence between random variables in a graphical structure is known as a probabilistic graphical model (PGM). It offers a framework that helps in visually representing causal dependencies between variables within a set of random variables in order to perform probabilistic analysis. Examples include the Bayesian network and the Markov network. PGMs use graphical representation to encode a complete distribution over multi-dimensional space. The graph represents a set of dependencies of a particular distribution in a compact form.

PGMs are mostly used in applications associated with probability theory, statistics (es-

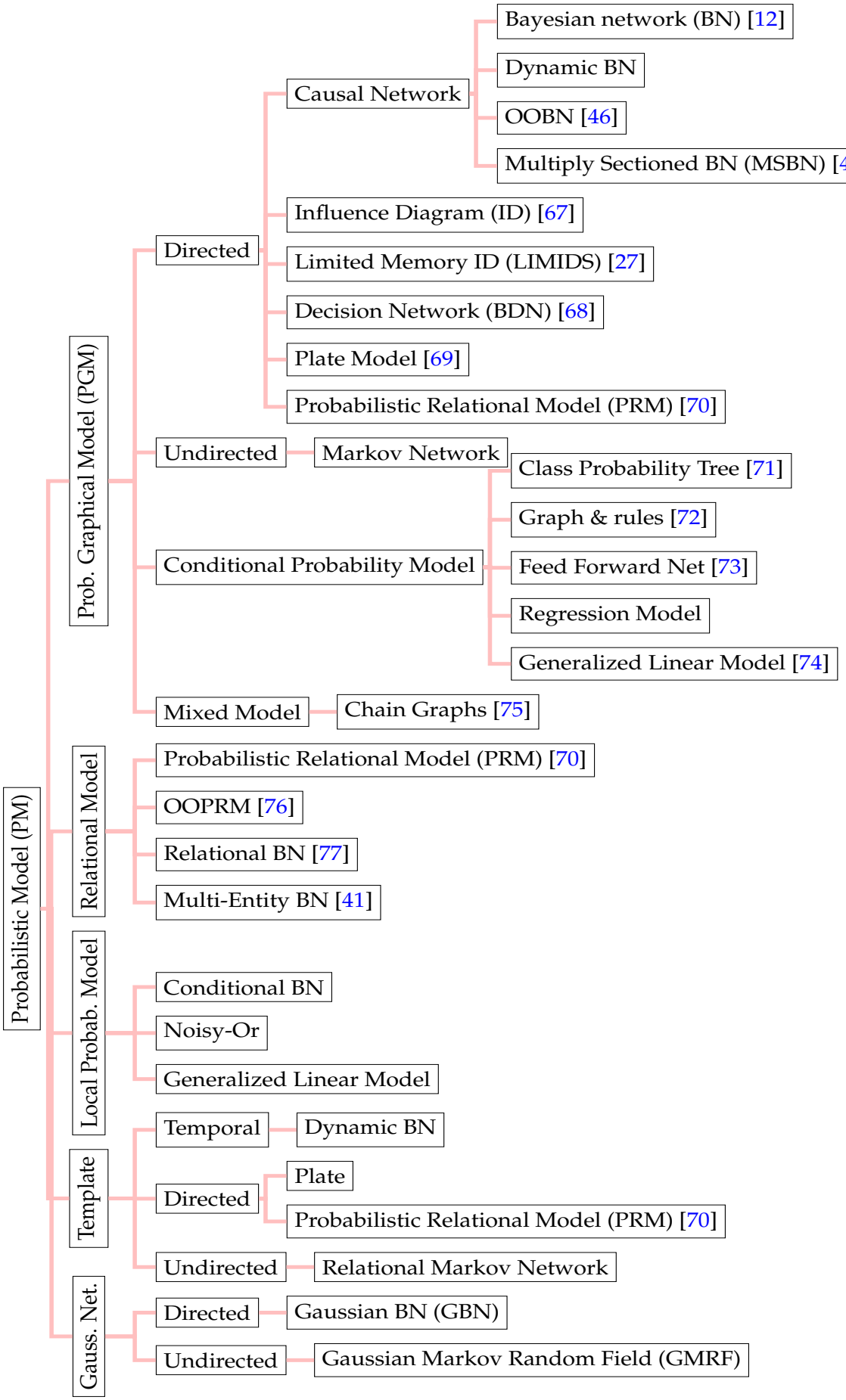


Figure 2.1: A taxonomy of Probabilistic Graphical Models (PGMs)

pecially Bayesian statistics), data mining, data analysis and machine-learning. To deal with uncertainty and probability, PGMs seem to be very effective and have become increasingly popular because of their ability to represent the conditional independence between random variables, incorporate dynamic information, and perform both predictive and diagnostic reasoning. Moreover, real-world problems usually need a combination of knowledge from a variety of areas, and this makes PGMs an ideal choice. One of the distinct characteristics of PGMs is their graphical visualization capability, a facility that, helps specialists from different fields to cooperate more efficiently.

Probabilistic relational models [1] (PRMs) [70] arose in the early 2000s inspired by relational database theory, relational algebra and relational logic programming. It is also significant that BNs were developed for data with the traditional 2D format in mind. Due to tremendous advancement in the technology, the data is becoming more complex with expanded dimensions and a number of associated attributes [78]. Ordinary BNs are not reliable in modelling applications to deal with such data.

To provide full access to the class components in order to allow modellers to model large and complex BN applications, a PRM contains reference slots to establish relations among objects. However, these reference slots violate the encapsulation and data hiding mechanism of the OO paradigm and hence introduce challenges in decomposing large applications that were developed across a group of modellers.

2.3 Bayesian Networks and Bayesian Decision Networks

2.3.1 Bayesian networks

A Bayesian network (BN) [10, 11] is a probabilistic graphical model that (a) compactly represents the joint distribution over a set of variables in the form of conditional probability tables (CPTs), one for each variable, and (b) represents a set of conditional dependencies and independencies between the set of random variables by means of a directed acyclic graph (DAG). In a BN, nodes and edges of the DAG represent random variables and their conditional dependencies, respectively. Nodes are connected with an edge or a path consisting of a set of edges representing conditional dependencies. If there is no such edge or path between two nodes, then the nodes are known to be conditionally independent. Also, each node has a CPT attached to it. More details on the BN and its classes are in Section 2.4. BNs can be used to perform reasoning under uncertainty: more specifically, there are algorithms to compute the posterior probability distributions over the states of a subset of the variables, given a set of

[1] Note that some authors classify PRMs as directed graphical models. In this thesis, PRM has been classified separately because of the difference in the underlying principles of graphical and relational models.

evidence. A variable does not necessarily represent an event.

The BN, being a graph-based framework, represents the causal relationship among various events (cause to effect) using DAGs. This framework provides ease of understanding and enhances the expressiveness of the dependencies and conditionalization in calculating the likelihood of associated events. The calculation of the likelihood of some hypotheses (state of an event) with respect to a set of evidence (as depicted in Section 2.1.1) can be represented more effectively using a BN.

DEFINITION 2.1 : BAYESIAN NETWORK

A **B**ayesian Network (BN) (following [13]) is a Directed Acyclic Graph (DAG) given by a 3-tuple $< N, E, \Pi >$, where

(i) N = a set of **chance nodes** representing random variables,

(ii) E = a set of directed **causal edges** representing the direct dependencies between nodes, with no directed cycles, and

(iii) Π = a set of conditional probability tables (CPTs) or distributions (CPDs), one for each chance node.

A node n_i is a **p**arent of node n_j if there exists an edge from n_i to n_j, denoted by $n_i \rightarrow n_j$. For each $n \in N$, $par(n) \subset N$ is the set of parent nodes of n, and the CPD $P(n|par(n))$ is a function Φ: $par(n) \cup \{n\} \rightarrow [0:1]$.

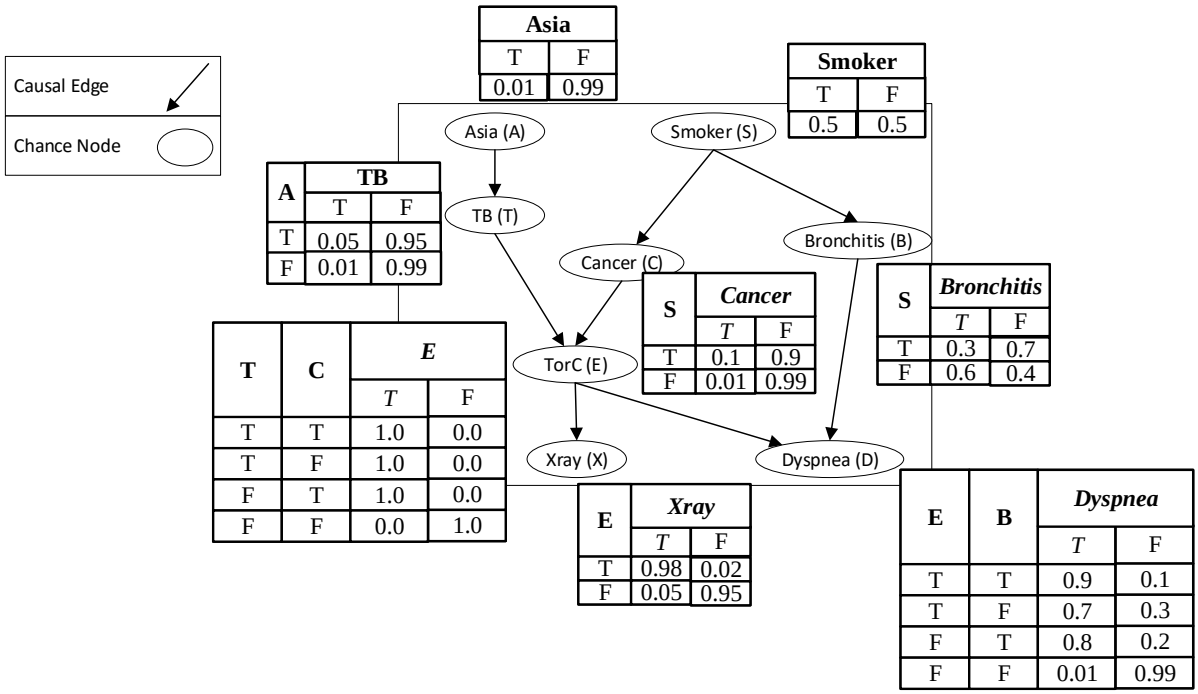


Figure 2.2: An ordinary BN: The classic "Asia" example

As an illustration, Figure 2.2 represents the well-known Asia BN [23] example (note that this BN is briefly used in Chapter 1 to introduce the BN concept). The nodes of the BN represent the factors/parameters in the form of random variables. The edges represent various dependencies and relationships among the nodes. The nodes are: "A" to represent the chance of a patient has visited Asia, "S", "T", "B", "C", "X" and "D" to represent, respectively, the likelihood of the patient being a smoker, the probability of the patient being affected by tuberculosis, bronchitis, cancer, the chance of the Xray being positive, and the possibility of the patient suffering from Dyspnea. The node "E" is used as a modelling choice to reduce the number of parents for Dyspnea and Xray. It does not have any other direct relation with the model. Each node has its own Conditional Probability Table (CPT) containing the likelihood of its different states with respect to the related nodes, i.e., parent nodes.

In a Bayesian network that is represented by a DAG, $G = \{V, E\}$, the joint probability distribution of n random variables X_1, ..., X_n is stated as:

$$P(X_1, ..., X_n) = \prod_{i=1}^{n} P(X_i|par(x_i))$$

where $par(x_i)$ is the set of parents of node X_i. As an example, for the BN in Figure 2.2, the joint distribution over all the variables can be represented compactly by CPTs in the BN in the following way:

$$P(A, S, T, C, B, E, X, D) = P(A).P(S).P(T|A).P(C|S).P(B|S).P(E|T, C).P(X|E).P(D|E, B)$$

where A = "Asia"(Visited?), S = "Smoker", C = "Cancer", T = "Tuberculosis", E = "TorC", B = "Bronchitis", X = "Xray" and D = "Dyspnoea".

The main rationale for building a BN is to calculate the posterior probability of a set of random variables using the prior probability and conditional probability of the variables and the evidence available at that time.

BNs are very effective in dealing with various decision-making processes, due to their ability to represent causal relationships among random variables, incorporating dynamic information from a variety of areas with varying degrees of uncertainty, and performing both predictive and diagnostic reasoning. Moreover, the graphical visualization power of BNs helps specialists with diverse expertise to cooperate easily [79]. BNs also allow documenting of a problem and its current state of knowledge, characterizing the overall model of the problem, as well as behaving like a storehouse of knowledge [33,35]. The process of creating a BN helps clarify assumptions and identify uncertainties within the system. As information improves, new data can be added to improve the model.

2.3.2 Reasoning with BNs

The process of thinking in a logical, sensible way to take a suitable action in a right time is known as reasoning. According to Charles Sander's Pierce [80], there are three basic irreducible and indispensable kinds of inference:

1. **Deduction**: deriving logical conclusions from known premises
2. **Induction**: deducing a universal conclusion from particular premises
3. **Abduction**: the only kind of reasoning that helps in introducing new ideas

Handling an uncertain situation and calculating the likelihood of a consequence (conclusion) with logic, statistics, and probability (premises) is known as probabilistic reasoning. An example of probabilistic reasoning is using past situations and statistics to predict the outcome of a future event. Analogously in the case of BN, the reasoning is a special action or operation to draw a conclusion using the premises or evidence provided in the network. As an example, consider the BN depicted in Figure 2.4: If the chance of "Weather" being "Wet" and evidence of the "Forecast" on the day of the match is "rainy", then the likelihood computation of the match "Result" is known as reasoning.

The process of conditioning is performed through a flow of information over the network. The flow is not bound to be directed towards the direction of the arcs. In a BN "information flow" refers to computing the posterior probability distribution for a set of query nodes, given values for some evidence (or observation) nodes. There are four types of reasoning [1]. Figure 2.3 illustrates the types of reasoning in terms of information-flow pictorially.

1. **Diagnostic reasoning:** If the information flows from symptoms to causes. The direction of information flow is opposite of network arcs in this kind of reasoning.
2. **Predictive reasoning:** When the information flow is from change in causes to the change in belief of effects. Here the direction of information flow follows the direction of network arcs.
3. **Intercausal reasoning:** The reasoning that involves the mutual causes of a common effect. If multiple causes of a common effect are initially independent, but in the presence of evidence from any of the causes, other causes are $explained\ away$ and intercausal reasoning takes place.
4. **Combined reasoning:** In a BN, any node can become a query node, and any node may contain evidence. In such a situation, none of the aforementioned reasoning types fits

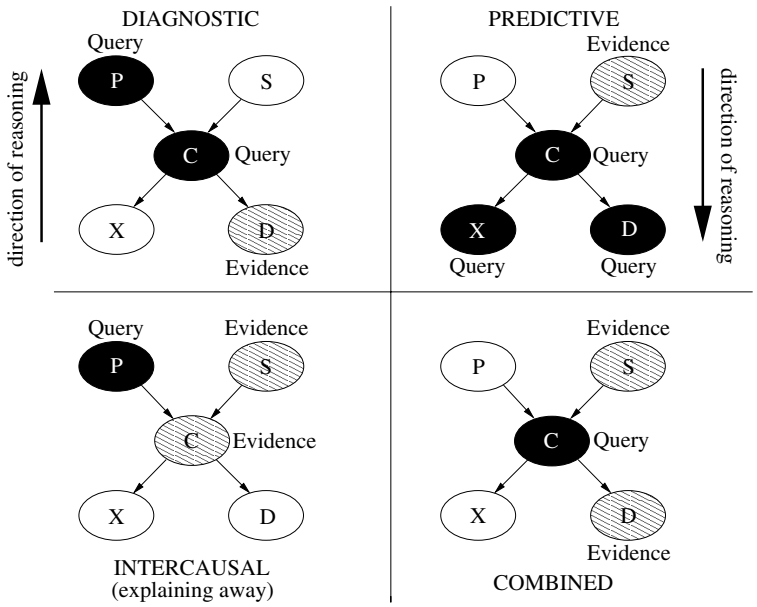


Figure 2.3: Types of 'Reasoning' in a Bayesian network [1].

well. Thus, a combined approach that fits well, can be used for reasoning. This approach is known as combined reasoning.

.

2.3.3 Bayesian decision networks

Bayesian networks can be used to make intelligent decisions under uncertainty. To perform such a crucial task, ordinary BNs are extended by adding decision and utility nodes. This extended BN is known as Bayesian decision network (Definition 2.2).

DEFINITION 2.2 : BAYESIAN DECISION NETWORK

A **B**ayesian Decision Network (BDN) is a Directed Acyclic Graph (DAG) given by a 3-tuple $< N, E, \Pi >$, where

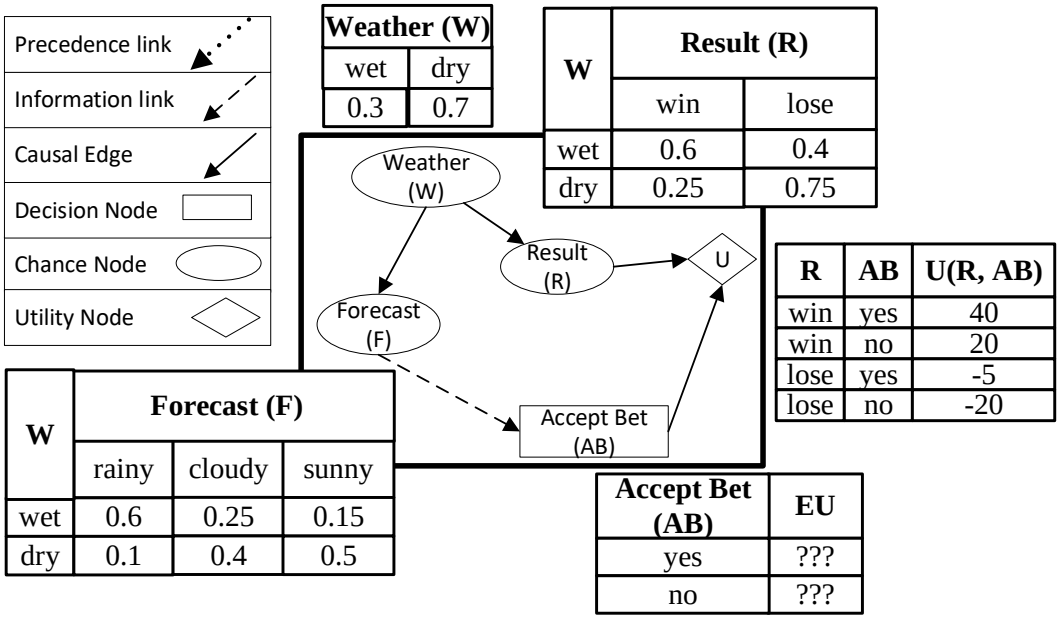
(i) N = a set of **nodes** (**chance** nodes representing random variables, **decision** nodes representing actions and **utility** nodes representing value or utility);

(ii) E = a set of **edges** (directed **causal** edges representing the direct dependencies between chance/decision nodes and chance/utility nodes, with no directed cycles, or directed dotted **information** links from chance nodes to decision nodes or **precedence** links between decision nodes);

(iii) Π = a set of conditional probability tables (CPTs) or distributions (CPDs), one for each chance node; a set of decision tables, one for each decision node; and a set of utility functions, one for each utility node.

A node n_i is a **p**arent of node n_j if there exists a causal edge from n_i to n_j, denoted by $n_i \rightarrow n_j$ or a link (information or precedence) $n_i \dashrightarrow n_j$. For each $n \in N$, $par(n) \subset N$ is the set of parent nodes of n. A node n and its parent node set $par(n)$ jointly define conditional probability or Utility function or decision table if n is a chance, utility or decision node, respectively.

Figure 2.4 shows an example of a BDN and its related symbol list. The network is built for predicting the outcome of a football match based on the weather conditions. A decision is to be taken to accept the bet or not, based on the weather forecast and the profit or loss in terms of the utility of winning or losing the bet. "Weather", "Result" and "Forecast" are the chance nodes representing, respectively, the probability of every possible state of the weather in the match day, the probability of possible results of the match, and the probability distribution of various forecasting states. "Accept Bet" is a decision node, representing a decision made by the system on the basis of posterior probabilities of the chance nodes calculated from their priors, provided evidences and the utility value defined as a utility function in the utility node "U". There are four edges "Weather" to "Forecast" and "Result", and "Result" and "Accept Bet" to utility. These edges represents the causal dependencies (child nodes are causally dependant on the parent nodes) between the random variables. The only edge from "Forecast" to "Accept



Weather (W)	
wet	dry
0.3	0.7

W	Result (R)	
	win	lose
wet	0.6	0.4
dry	0.25	0.75

R	AB	U(R, AB)
win	yes	40
win	no	20
lose	yes	-5
lose	no	-20

W	Forecast (F)		
	rainy	cloudy	sunny
wet	0.6	0.25	0.15
dry	0.1	0.4	0.5

Accept Bet (AB)	EU
yes	???
no	???

			Accept Bet		
F	**P(W=wet)**	**P(R=win)**	***EU**(yes)*	***EU**(no)*	**Decision**
rainy	0.720	0.502	**10.12**	7.55	yes
cloudy	0.211	0.324	-0.56	**3.10**	no
sunny	0.114	0.290	-2.61	**2.25**	no

Expected Utility calculation to make a decision on “Accept Bet” for the evidence on W = “wet” and R = “win”

Figure 2.4: An ordinary BDN: Football match betting [1].

Bet" is an information link. This link indicates that the "Forecast" node needs to be observed before a decision is made. The nodes, edges/links, their types and functions are explained formally in Section 3.2 of Chapter 3. A point worth noting that the decision outcome shown in the decision table for the decision node "Accept Bet" is not predefined or defined by a modeller; rather, it is calculated based on the posterior probability of the chance nodes and the utility value of the utility node. To illustrate this situation, in Figure 2.4, the decision table for the "Accept Bet" decision node, the Expected Utility (EU) column contains "???" to indicate that the value is not settable. At the bottom of the figure, a table is shown with a set of evidences and the calculated EU value for each of the states of the "Forecast" node (a node that must be observed before a decision is made). Then, based on the maximum expected utility policy, a decision is taken as shown in the last column of the table.

BDNs have been used for reasoning under uncertainty for decades. The reasoning is substantial in real-life applications because of their complex nature, large span and associated

challenges such as the uncertainty of events.

To use the facilities of BDNs, constructing BDNs or modelling in BDNs is the first step. BDNs can be built in four ways [1]:

1. By-hand, using elicitation methods to capture expert knowledge
2. From models in the literature
3. By automated learning if data is available, or
4. By combinations of the aforementioned techniques.

More detail is provided about the so-called "knowlege engineering" of Bayesian networks below in Section 2.8.

2.3.4 Limitations of BNs

Ordinary BNs suffer from several performance-related issues. The most important is that BNs are not capable of scaling-up in real-life complex domains. The limitations of BNs can be divided into the following two classes (though these are overlapping and closely related).

1. **Representational issues:** An ordinary BN follows a simple propositional logic like representation of knowledge in order to construct a knowledge base that can model under uncertainty. To overcome this issue, several approaches have been proposed, such as the dynamic BN, the relational BN, the Multi-Entity BN, and the Multiply Sectioned BN (described in the following section).

2. **Scalability issues:** The underlying techniques to build a BN are complex and elementary operations in performing inference are computationally very expensive. Note that BNs are complex and expensive to construct with cost [2] directly proportional on the size of the network. When much, or all, of the model is built by hand, as the complexity of the problem increases, BN modelling methods struggle to scale up. The resultant large complex BNs are difficult to visualise and hard for the domain experts and decision makers to understand, reducing the uptake and subsequent use of the model. An interesting fact is that almost all real-life applications are large and complex and suffer from scalability issues. To cope with such complex, large-scale and dynamic real-life problems, an ordinary BN is not the right choice.

   In order to address this issue, various scaling techniques like Idioms, Fragments, Object-Orientation, relational and template models have proposed; these are described in the following section.

[2] The construction cost includes the relevant human experts' experience and efforts, modellers endeavour, required hardware and software resources and other relevant stuffs.

2.4 Other Related Probabilistic Graphical Models

From the late 1990s, researchers started to develop theories and techniques to scale up BN modelling. Thus, there has been much effort to improve BNs in many ways over the years. The concept of BN and inference in BN have also been extended to suit particular fields to serve the purpose of probability analysis. The approaches to extending BNs also include versions of well-known techniques for handling complexity, such as dividing the problem into subparts and then combining the BN models for the subproblems, and reusing a BN model previously built and validated for another application. These techniques include Dynamic BNs [81–84], BN fragments [32], generalised decision-graphs [38], OOBNs [46], PRM [70], OOPRM [76], varieties combining PRMs and objects, such as module networks [39], PRMs and plate models [40], multi-entity BNs (MEBNs) [41], idioms [34], and template-based representations [43].

BNs do not represent temporal relationships between variables. The only way of capturing the temporal relationships between the value of the variables at different points in time (past, present and future) is to add extra variables of the same type but with different names. In real-life applications, it is important to model how the world changes with the change in time from a particular point of time. **Dynamic BNs** (DBNs) are an extension of BNs that are capable of modelling changes of probabilities, actions and evidence with respect to time. According to Korb and Nicholson [1], for a BN with n nodes that model a domain of n variables $\mathbf{X} = \{X_1, ..., X_n\}$, the DBN that models the change of values in the variables over time should contain one node for each X_i for each instance of time. For a current time instance t, the immediate past time instance and immediate future time instance are represented by $t-1$ and $t+1$, respectively. Hence, the nodes for DBN to model such a temporal system is:

1. Current (in time t): $\{X_1^t, ..., X_n^t\}$
2. Immediate past (in time $t-1$): $\{X_1^{t-1}, ..., X_n^{t-1}\}$
3. Immediate future (in time $t+1$): $\{X_1^{t+1}, ..., X_n^{t+1}\}$

Each time instance is called a "time-slice", and two new types of arcs are introduced, namely "Intra-slice arcs" and "Inter-slice or temporal arcs". As the names suggest, the arcs that represent a temporal relation between variables within a time-slice are intra-slice arcs and the arcs between the variables of successive time instances are inter-slice arcs. The latter type can be between the same variables or between different variables at successive times. These arcs deal with the effect of the change in one variable in time t on the other variable of time $t+1$.

In recent years, with the growing interest of relational pattern extraction, **Relational BNs** (RBNs) have gained much importance. These are an extension of ordinary BNs that allow

relational data representation [78]. This model was called the "Probabilistic Relational Model" (PRM) and was first proposed by Koller and Pfeffer [85]. The concept of objects (as opposed to random variables and their attributes in BNs), objects' properties, and relationship between objects are at the core of this model. It has a similar different relation to BNs as relational logic has to propositional logic. The model specifies a template for a probability distribution for a database. The template includes a relational component for describing the relational schema and a probabilistic component for describing the probabilistic dependencies [86].

Later, Nevile and Jensen [87] proposed the term "RBN" to refer to the BNs that can model relational data, i.e., the "PRMs" as proposed by Koller. They also proposed the use of the term "PRM" for the type of PGMs that mines statistical patterns from relational data rather than extended BNs that model relational databases. Note that there is another kind of RBN, proposed by Jaeger [88]. This is entirely different from the PRMs (renamed as RBNs) proposed by Koller. Jaeger's RBN is an extension of the BN using First-Order logic (FOL). For representing probabilistic relations, it is more powerful and expressive than the ordinary BN, as it uses a powerful FOL in contrast to propositions. Using FOL allows the adding of constraints on the equality of events, defining complex, nested functions and a recursive network.

Gaussian Bayesian networks (GBNs) [89] are a particular type of BN where all of the variables are continuous, and all of the CPDs for the variables are linear Gaussians. It is used to define a continuous joint distribution and provide an alternative representation for a multivariate Gaussian distribution class [70].

Multiply-sectioned BNs (MSBNs) [42] were proposed with the localisation of queries and evidence in mind. By localisation, the authors meant that at a particular point in time, queries are directed towards a part of the whole network. A point to note is that the original formulation of BNs (ordinary BNs) do not consider the structure in the domain and the whole network is treated as a homogeneous network of the variables under consideration. In such a system, probability propagation for inference is inefficient since, for localized evidence, the whole network needs to be updated. MSBN offers a localisation-preserving partition of a BN by allowing a set of separate Bayesian sub-networks. These sub-networks are transformed into a set of permanent JTs such that evidential reasoning can occur in any one of them at any time. This also ensures that the calculated marginal probabilities are the same as if they were calculated in the homogeneous network.

The **Multi-entity Bayesian network** (MEBN) [41] is a first-order language for modelling uncertainty using first-order logic language. It combines BNs with FOL to provide BNs with the power of first-order expressiveness and uses FOLs as the means of modelling probability. The MEBN specifies parameterized fragments of BNs (a.k.a. MFrags) to express probabilistic

relationships among a small collection of related hypotheses in order to form a probabilistic knowledge base. A set of instantiated, combined MFrags form an arbitrary degree of complex graphical probability models. An MFrag can be instantiated any number of times, and that enables an MEBN to express complex graphical models with repeated structures. Hence, MEBN is a compact language capable of representing knowledge at a natural level of granularity. Like BNs, the MEBN also uses directed graphs to define joint probability distributions.

Fragments, proposed by Laskey and Mahoney [32], are large-scale BN construction schemes where knowledge is specified in larger and semantically meaningful units, called "fragments". A fragment is a set of related random variables that can be constructed and reasoned about separately from the others. The OO concepts are used to represent and manipulate fragments. In fragments, input variables are used to specify interfaces and so-called "resident" variables are used to encapsulate private data.

Authors emphasise network composition rather than network construction. In fact, any vast network can be constructed from non-decomposable small units. If any method can precompute and store them, then computation for a new but slightly different (perhaps larger) network construction need not be started from scratch. The framework allows for representing asymmetric independence and canonical intercausal interaction.

Idioms: Although fragments [32] provide the ability to solve real-world, large-scale problems by providing methods for defining component-level BNs and combining them into a consistent model, knowledge engineers still need a guide to adopting past inference solutions to current problems. Inspired by design patterns, Fenton [34] described a solution to these problems based on the notion of generally applicable "building blocks" that can be combined into objects: they named these "idioms".

By combining the idea of idioms, some large-scale problems may be addressed and solved. However, there are some common problems of patterns in software engineering and Idioms in BNs. These are:

1. There is no guarantee that patterns are suitable enough to model all real-life applications.
2. It is hard to find appropriate patterns/Idioms due to overlapping segments among patterns.
3. Some complex real-life applications require more than one pattern which may lead to undesirable overheads.

Sub-networks: GeNIe [28] supports another specialised form of BNs, i.e., sub-networks. Sub-networks cannot be characterised as classes in OOBNs (see Section 2.5 for more on OOBNs and classes). It allows building a hierarchy of embedded networks and the hierarchy needs

manual maintenance. In the case of any change in one sub-network at any level of the hierarchy, all other sub-networks in the lower levels connected to this sub-network need manual changes. Moreover, if a modeller embeds multiple copies of the same sub-network, they have to make changes in each of the instances.

Object Oriented BNs (OOBNs) [46], incorporates OO features in BNs to resolve the so-called "scalability" problem by ensuring reuse of existing and previously defined components. It allows encapsulating a BN segment in a class and reuse that segment in the form of "objects". It is a great approach to utilize the OO features, such as encapsulation, abstraction, inheritance, and polymorphism, in BN arena to provide various facilities to the modellers. More on OOBNs are explained later in Section 2.5 in detail.

In **Template**-based representation, a PGM specifies a joint distribution over a fixed set of random variables. This fixed set and the distribution can be used in many different situations. For example, a student performance observation network can be applied to multiple students. Basically, all the students share the same structure – the components in this structure can be viewed as attributes. Only the attributes' variables differ between students. Koller and Friedman [43] called this model "variable-based" because of the focus on the presentation on random variables.

Koller and Friedman [43] offered a general framework for defining templates for fragments of the probabilistic model. These templates can be reused both within a single model and across multiple models with different structures. The two template-based representation languages that can be applied to the theory of OOBNs are Plate models and PRMs [40].

Object Oriented Probabilistic Relational Model (OOPRM): Probabilistic relational models (PRMs) [70] (introduced briefly in Section 2.2) are an alternative to OOBNs that were proposed in the early 2000s, inspired by relational database theory, relational algebra and relational logic programming.

While the original PRM did not include inheritance, the OOPRM [76] extends the PRM framework by introducing OO concepts such as interfaces, inheritance and polymorphism. Compared with the PRM, it has another special feature, i.e., the inverse reference slot, which helps in more efficiently accessing the attributes of classes. However, an interesting fact is that the OOPRM has the same issues as the PRM with reference slots, while inverse reference slots also violate encapsulation, and the OOPRM is thus less flexible to extension, modification, decoupling, and decomposition. Moreover, in an OOPRM, the reference slot chains and inverse reference slot chains make modification and reuse far more complicated, as the modeller needs to consider how a class is embedded in the whole system when making even a simple modification to a single class. In addition, while the PRM and OOPRM frameworks allow

the compact representation of relationships between classes that are instantiated with multiple objects, they do not provide the utility and decision nodes that allow BNs (and OOBNs) to be used for decision-making and utility computation. Finally, while the OOPRM has been implemented in a research software tool, AGrUM [90], it is not available in any commercial modelling tool, and there seem to be very few real-world OOPRM models described in the literature.

Table 2.1 demonstrates a comparative study of the probabilistic models, described in Section 2.4 and Section 2.5, in terms of their significant features, limitations and a suggestive model to overcome the limitations.

Table 2.1: Comparing Probabilistic Models.

Models	Features	Limitations	Overcoming Techniques
Ordinary BN	Represents knowledge using propositions and probability Distributions	Scalability	OOBN, Templates, Fragments, Idioms, PRM, OOPRM
		Time-slice representation	Dynamic BN
		Repeated structure	OOBN, Template, Idioms, Fragments
		Dealing with localisation of query	MSBN
		Expressiveness	MEBN, Relational BN
Dynamic BN	Modelling change in uncertainty with respect to time	Requires extra nodes to deal explicitly with time steps	OOBN, OOPRM
		Scalability	OOBN, OOPRM
Relational BN	Extends BN by adding First Order Logic, facilitates nested and complex function definitions	Same as BNs except expressiveness	OOBN, Templates, Fragments Idioms, PRM, OOPRM
MSBN	(1) Facilitates localized query (2) Avoids probability propagation throughout whole BN	(1) Does not provide maximum reusability such as OO definitions (2) Same as BN except scalability	OOBN, OOPRM
MEBN	(1) Provides a First-order language for modelling application under uncertainty (2) Defines MFrag to allow repeated structure	(1) Does not provide maximum reusability like OO definitions (2) Same as BN except for expressiveness (3) Decision and Utilities are not supported	OOBN, OOPRM
Fragments	(1) Offers fragment (non-decomposable unit) as a set of variables (2) Preliminary Idea of OOBNs	Not a complete OO-system and past inference results cannot be used	OOBN, Idioms
Idioms	(1) Offers compositional probability modelling (2) A large system can be represented by a set of Idioms to solve it by combining the solutions of the idioms	(1) All problems may not be decomposable by Idioms (2) may lead to undesired overheads	OOBN
Templates	(1) Offers a special set of random variables, called template. (2) Template is a general form of an analogous segments in a model and helps in defining a common solution	All the features of OO-paradigm such as inheritance, encapsulation, polymorphism are not defined	OOBN
OOBN	Introduces OO features (inheritance, encapsulation, polymorphism) to BNs	Recursive definition not allowed	OOPRM
PRM	(1) Extends BNs by introducing classes and defining the relations of attributes by reference slots (2) Models relational data by relational logic	Reference slots violate encapsulation	Pure OO-notion
		Dependency added by reference slots makes extension, maintenance and decomposition difficult	
		Only random variables are supported	OOBN
OOPRM	(1) Introduces OO features (inheritance, encapsulation, polymorphism) to PRMs (2) Inverse reference slot and interface	Complex representation	OOBN
		Only random variables are supported	OOBN
		Reference and inverse slots violates encapsulation	Pure OO-notion
		Maintenance, decomposition and extension is difficult	Pure Encapsulation and abstraction

2.5 Object-Oriented Bayesian Networks (OOBNs)

This section introduces OOBNs, a variant of ordinary BNs which include object-oriented features from software engineering, and is the focus of this thesis. The core features and principles of Object-Orientation are discussed briefly before several variants of OOBNs are explained and compared. Note that a fuller and more formal presentation of OOBNs is given in Chapter 3, as part of the new iOOBN framework.

2.5.1 Object-Orientation

If the whole universe can be seen as a programmed system, then every component in the universe is an object of a particular class. This notion has been adopted in the programming paradigm to introduce object-oriented programming. Object-Oriented programming (OOP) is a paradigm that represents real-life entities as "objects" (an entity with particular attributes/-data and behaviour of the attributes in the form of methods or functions) [91]. By defining the interaction between these objects, computer programs and applications are created [92]. A feature of objects is that an object's procedures can access and often modify the data fields of the object with which they are associated with (note that objects have a notion of "this" or "self"). In OO programming, computer programs are designed by building them out of objects that interact with one another. The Object-Oriented concept revolutionised software technology by allowing maximum reuse of the existing resources and by providing scalability.

In general, objects are instances of classes which typically also determine their type. The classes are considered as blueprints (how and what makes an object, how all components are related), and objects are their instances. The four most significant features of OOP are,

1. Encapsulation: Hiding data from unauthorized access (see Footnote 9 of Chapter 1). Data and its behaviour defining methods are packetized into objects.

2. Abstraction: Providing a generalized view to allow distributed, modularized, reusable and expandable development.

3. Polymorphism: Having multiple forms with a single name. This allows substituting the appropriate version of an object/instance for a particular application.

4. Inheritance: The same properties between classes are written once in a class (in the parent class or superclass) and are inherited in other classes (in the child class or subclass). This avoids the need to create a class from scratch. Inheritance helps avoid "reinventing the wheel" by ensuring reusability of existing resources and minimizing redundancy, and thus it supports scalability.

2.5.2 Motivation for OOBNs

A well-known drawback of the BN is that it becomes complex when used to design large real-life applications. Extending the system to add functionalities, modifying an existing application to fit that in modelling a similar application, and maintaining a large complex BN system are hard. Even very little change in the system requires full domain knowledge. One approach that has been adopted from object-oriented principles of software engineering that has been useful in dealing with large-scale problems is Object-Oriented Bayesian networks (OOBNs) [36, 37]. OO principles allow BNs to encapsulate networks into classes and objects defined with attributes of similar characteristics and with a single goal of computation which provides scalability (in terms of reusability) to BNs. Due to the ever-growing complexity of constructing BNs, reusability is a potential technique for constructing a large BN to use in real-life applications. Introducing OO features in the BN paradigm solves a lot of well-known limitations of BNs. Reusability, anti-redundancy, and modularity (where a particular class is responsible for one particular type of operation) decouples a large system and makes it easy-to-maintain and extend. Encapsulation enables data hiding and privacy. Abstraction provides an easy interface in working with some segments of BNs without knowing the details of the segment. These attributes make the OO concept attractive for use in BNs to be used in large and complex applications.

There are many advantages claimed for using OOBNs instead of ordinary BNs. A non-exhaustive list follows:

1. Provision of a modular structure which allows greater flexibility and robustness [6]. It adopts encapsulation and makes probabilistic objects reusable

2. Reusability of tested and correctly functioning components avoids the complexity of reconstruction/design and correctness testing. That is, it avoids the tendency of "Reinventing the wheel"

3. Abstracting/hiding the internal complexity of a class and providing a simpler interface to its users

4. Redundancy (with respect to reusability) avoidance provides scalability

5. Along with its flexibility, strong type checking can ensure avoidance of unintentional flaws in modelling

6. Enables design of large BNs with fewer flaws with easier maintenance and management of changes

7. Superior error handling
8. Interface structure facilitates parallel design and thus supports a team approach in designing and integrating large and complex systems
9. Provides modularity to accommodate dynamic domains
10. Facilitates both spatial and temporal design [93]
11. Allows specification of template models in an easy and intuitive way
12. Allows sharing and reusing of JT structures among subclasses and instances that in turn solves the greatest challenge of inference for BNs

OOBNs are inspired by object-oriented principles from software engineering (e.g. [94]), where subparts of the overall model are represented by classes that contain both nodes and objects (instances of other classes), giving a composite and hierarchical structure. These key concepts were first introduced by Koller and Pfeffer [36], and are described in some detail below. Connections between objects are strictly limited to define input and output interface nodes, supporting the OO concepts of encapsulation, abstraction and information hiding. Thus a modeller may embed an object in the larger model without knowing its details, and the decision-maker can view all or parts of the model at different levels of abstraction, aiding understanding and acceptance. The use of classes also supports the maintenance of the models, as any change (e.g., updating the model parameters when new data becomes available) needs only be made once, in the class, and the change is automatically propagated to all the objects of that class, in any number of OOBNs. Other advantages of OOBNs include supporting the building of large OOBNs in parallel by multiple modellers, who only need to agree on the class interfaces; providing modularity which limits the scope of changes and reduces the chance of a model change introducing errors; and facilitating the design of both temporal [3] and spatial models.

In the OO paradigm, inheritance is the ability to derive attributes and behaviour from pre-existing classes, which enables a greater level of reusability and scalability. In current OOBN modelling, every new class is a separate entity, regardless of its similarities to existing classes. Incorporating inheritance into an OOBN framework means allowing classes to be "extended", with new classes – called subclasses – to be defined in terms of "inheriting" certain elements from the original class, along with the differences. This should allow the reuse of already constructed network segments, as well as supporting maintenance.

[3] Hugin implements dynamic Bayesian networks, used for explicit reasoning over time, using OOBNs.

To solve the limitations of BN, especially scalability issues, OOBN theories, and techniques have been developed to cope with large-scale modelling problems by adopting OO concepts from software engineering. This approach integrates abstraction and composition concepts into BNs' structure, giving them the ability to model compositionally and hierarchically.

Until now, OOBNs have been applied in a variety of fields, such as medicine [95], environmental management [96], agriculture [97] and mechatronics [98]. A survey of the applications of OOBNs can be found in [1]. The application area of BNs is vast. Some of the applications are listed in [1, 99–102]. Moreover, from their inception, BNs have been improved in many ways and applied in uncountable applications. Listing all of them is beyond the scope of the thesis; however it is clear that while BNs are now very widely used, OOBNs have not been anywhere near as widely adopted, despite their potential to help what has been called a "bottleneck" in BN knowledge engineering [1].

The claimed advantages for modelling OOBNs (listed above), over ordinary BNs, are in the most part based on claimed advantages of an OO approach in software engineering [47–49]. However many of them have not been demonstrated in practice in OOBN, in part because of the limited support for OO features, including inheritance, and the limited number of OOBN applications (at least compared to other BN applications).

2.5.3 The OOBN proposed by Koller and Pfeffer

Koller and Pfeffer first proposed Object-Oriented BNs in [36], where they extended Bayesian networks formulation with encapsulation and inheritance capabilities, to deal with scalability issues, and to achieve compactness and modularization.

Koller and Pfeffer proposed OOBNs in terms of classes and objects with a particular interface for objects. The interface helps objects to deal/interact with the outer world. This definition allows OOBNs to be interpreted as a stochastic function. In addition, their framework provides an algorithm for belief propagation in OOBNs.

This framework, however, has several significant drawbacks. Though the encapsulation is well defined and inheritance is introduced, the latter concept (inheritance) was not fully defined and hence was never implemented. More specifically, they described inheritance in terms of some graphical structure sharing but offered no clear guidelines for parameter sharing among OOBN classes. For example, if a class is given (DAG), it can be used to define new classes, where the new class DAG is a super-DAG of the given DAG. Moreover, other key aspects of object-orientation in software engineering, for example, type checking and typecasting facilities, were neither described nor defined at all. Further, polymorphism, a vital feature of object-orientation arising from inheritance, is not explained. Moreover, how to deal with many

practical issues such as (a) interfaces having unequal number of nodes, (b) different number of embedded nodes, and (c) differences in the types of the nodes in a subclass, were not defined in Koller and Pfeffer's formulation of OOBNs. Other notable OO-features such as polymorphism, abstraction and type checking are also missing. The framework does not contain any definition or explanation on how to change an object, embedded in a class, or how to construct an object. Hence, the real power of the OO paradigm is not exploited. Most importantly, these authors did not implement their proposed OOBN framework.

2.5.4 The OOBN proposed by Bangsø et al.

Bangsø et al. [5, 37] proposed another OOBN framework, the "Plug and Play OOBN", where several limitations of the existing OOBN (Koller-Pfeffer's) definitions have been overcome and some additional features, such as the dynamic changing of object structures, have been introduced. This framework introduces incremental compilation and performing probability propagation in an OOBN very efficiently.

The authors proposed a mechanism for changing the dynamic object via minimal operation and also an efficient technique for belief propagation by a JT modification based approach. This approach means that there is no need to construct a JT from scratch for each and every change in the BN or OOBN. Instead, it is only necessary to modify the minimal subgraph(s) that have been affected by the change, and only the affected portion in the JT. This approach has proved to be significantly effective one for compiling a BN.

However, there are still some gaps in the definition. The authors did not define inheritance in details in the framework as this formulation still lacks: A definition of multiple-level embedding (as described in Footnote 10 above), any policy to deal with different types of nodes in the interface or in the embedded level of a class, and facility to restrict the sharing of a particular segment of a superclass. Furthermore, their framework did not include type checking or polymorphism. Most importantly, even the features they allow for have never been implemented in any of the existing frameworks or software applications to date.

The concepts of object formation, instantiation, dynamic maintenance, inheritance and other important properties of the OO paradigm, which enable the highest level of reusability, have not been fully adopted or developed as yet. These concepts have only been proposed and described as a promising extension of OOBNs in several publications such as [36, 37], [97], and [1].

2.5.5 The OOBN proposed by Huang et al.

Recently, Huang et al. proposed another formulation of OOBNs [50] with the limitation of the previous two formulations (i.e., Koller's and Bangsø's OOBNs) in mind. It is claimed that this framework contains a better and clearer explanation of inheritance in OOBNs. Indeed, the explanation of encapsulation in this framework is clearer and the guidelines provided regarding inheritance is superior than the other formulations/frameworks. However, the inheritance explanation is still not adequate. These authors did not explain any specific inference technique dedicated to the framework that best uses the inheritance facility. Moreover, some very significant features of OO-paradigm, such as polymorphism, type checking and typecasting, are not defined in the framework.

The framework is partially implemented (not all inheritance features are implemented) by the authors using the API of UnBBayes in Java programming language. There are some Java and UnBBayes version related inconsistencies in the implementation. Hence, the code base (that they shared on request) for the implementation is not helpful for adding new features or extending the existing features.

2.5.6 Comparison of existing OOBN frameworks

Table 2.2 illustrated a comparison of the features supported by the OOBN formulations proposed by Koller and Pfeffer, Bangsø and Huang et al.

The first implementation of OOBNs was in the 2003 version of Hugin [103], a widely used and (with Netica) the longest-established commercial BN software tool, but one that still does not include any implementation of inheritance. The OOBN framework presented by Bangsø et al. [37] did provide a limited form of inheritance (although only a subclass is able to change the interface and the hidden structure).

Table 2.2: Comparison of Koller–Pfeffer's, Bangsø's and Huang's OOBN frameworks.

Features	**OOBN (Koller)**	**OOBN (Bangsø)**	**OOBN (Huang)**
Encapsulation	✓✓	✓✓	✓✓
Inheritance	✓	✓	✓✓
Abstraction	✓	✓	✓✓
Inference	✓	✓✓	✓
Time-slice representation	–	✓✓	✓✓
Real object notion	–	–	–
Typecasting and Type-checking	–	–	–
Polymorphism	–	–	–

✓✓ : Clearly defined	✓ : Vaguely defined	– : Not defined at all

2.6 Software for Existing Probabilistic Models

There are numerous implementations of various Probabilistic Graphical Models. Each has some speciality and a list of features. Some are free, some are open source, and some are commercial. Some support BNs (directed models), while some support relational or undirected models. They also differ in underlying methodologies, mechanisms to perform inference, learning, and making decisions. Some of the tools support particular programming languages and are compatible with various operating systems. Each serves a particular purpose and has limitations for some features as opposed to others. Some of the tools support GUI+API, some offer only API, or only programming languages, and some offer query languages.

Table 2.3 compares the critical features of some popular BN tools, while Table 2.4 does the same for some popular relational modelling tools.

Table 2.3: Comparing key features of popular BN tools

Bayesian network									
Tool name	**Commercial / Open source**	**GUI / Programming**	**OO-features Supported**	**Inference Feature**	**Learning Feature**	**Continuous Data Support**	**Languages supported for API**	**Well Documented in English**	**Other Features**
Hugin [103]	Commercial	GUI	Yes	Exact and Approximate	Learning BNs with missing data	Yes	C, C++, Java, .NET, ActiveX-server	Yes	Sensitivity Analysis
Netica [29, 104]	Commercial	GUI	No	Various Inference	Learning with missing data	Discretized Continuous Data	C, C++, C#, Visual Basic, MatLab, CLisp	Yes	Sensitivity Analysis
BayesiaLab [105]	Commercial	GUI	No	Exact and approximate	Parameter learning by maximum likelihood. Structure learning with missing data	Discretized Continuous Data	Java	Yes	Supervised and unsupervised learning
ProbaYes/ ProBT [106]	Commercial	Structured Programming Language	No	Exact and approximate	Parameter and Structure with missing value	Yes	C++, C#, Java, Python, Excel plugin	Yes	DBN, HMM,
Genie [28]	Commercial	GUI	Yes [a]	Exact and approximate	Parameter and Structure	Yes	C++, Python, Java, .NET, MS Excel	Yes	Web-browser and Mobile device
UnBBayes [107]	Open source	GUI	Yes	Exact Inference	Learning: K2, B, CBL-A, CBL-B, and Incremental	No	Java	Mostly in Portuguese	supports MSBN and HBN
BayesNet for Matlab [108]	Open source	API only	No	Exact and approximate	Parameter and structure learning with missing data MCMC, IC, PC are there	API only	Matlab, C	Yes	Both Directed and Undirected PGM
BNLearn in R [109]	Open source	No	No	Exact and approximate	Parameter and Structure Max Min Hill Climbing Hybric HPC	Yes	R	Yes	Random data generation, TAN
OpenMarkov [110]	Open source	GUI	No	Exact Inference	PC and Hill climbing Parameter by Laplace-like correction	No evidence	Java	Yes	Sensitivity Analysis
Elvira [53]	Open source	Both	No	Exact Inference	Structure Learning	No	Java	In Spanish	Decision Making
BN Tool in Java [111]	Open source	API	No		Structure Learning	Continuous State	Java	Yes	ID and DBN

API = **A**pplication **P**rogramming **I**nterface, GUI = **G**raphical **U**ser **I**nterface, PC = **P**rototypical **C**onstraint-based algorithm, HPC = **H**ybrid **PC**, IC = **I**nductive **C**ausation, MCMC = **M**arkov **C**hain **M**onte **C**arlo, HBN = **H**ierarchical **BN**, TAN = **T**ree **A**ugmented **N**aive Bayes, ID = **I**nfluence **D**iagram, LIMID = **LI**mited **M**emory **ID**.

[a] GeNIe support sub-networks (though not as classes - you can build a hierarchy of embedded networks, though not connected via an interface, and if you embedded multiple copies of the same subnetwork you have to make changes in each of the instances.

Table 2.4: Comparing key features of popular relational modelling tools

Relational Modelling tools									
Tool name	**Commercial / Open source**	**GUI / Programming**	**OO-features Supported**	**Inference Feature**	**Learning Feature**	**Continuous Data Support**	**Languages supported for API**	**Well Documented in English**	**Other Features**
ProbReM [112]	Free and Open source	Language to describe the relationship	No	Inference for DAPER models by MCMC	Parameter learning by ML (max. likelihood)	No	Python XML based data representation	Yes	Directed Graphical Model
Alchemy [113]	Open source	Programming	No	Logic inference in Markov logic net	statistical relational learning, structure learning	Yes	C++	Yes	Lifted Belief Propagation Sampling
Primula [114, 115]	Open source	GUI	No	Exact and Approximate Inference for RBN	parameter learning for RBNs, Complex and nested models	No evidence	Java	Yes	Supports Inheritance and Nesting and DBN
BLOG [116]	Open source	Probabilistic Modelling Language	No	Default Inference	No evidence	Yes (Limited to static continuous)	Java .NET	Yes	Provides a Query language, Dynamic
UnBBayes [107]	Open source	GUI	No	Inference	No evidence	No	Java	Mostly in Portuguese	Supports MEBN
Proximity [117]	Open source	Programming and QGraph	No	Inference	Learn from relational data but not for RBNs	Yes	QGraph visual query language	No evidence	Supports highly expressive domains
AGrUM [90, 118]	Open source	Programming API	Yes	Lifted prob. inference	Learning graphical models	Yes	C++	Yes	Decision Trees
IBAL [119]	Open source	Programming	No	Approximate and Exact Inference	Parameter Learning in the form of Bayesian parameter estimation	No	Objective CAML	Yes	Strongly typed built-in extensibility

DAPER = **D**irected **A**cyclic **P**robabilistic **E**ntity-**R**elationship, MCMC = **M**arkov **C**hain **M**onte **C**arlo, ML = **M**achine **L**earning.

2.7 Inference / Conditioning / Belief Updating / Probability Propagation

Inference (see Section 2.3.2) is essential to perform reasoning in a BN [120]. It is a step towards reaching a conclusion on the basis of evidence and reasoning. Korb and Nicholson [1] describe belief updating or probabilistic inference in an inference system as the rudimentary operation in computing a posterior distribution for a set of query nodes with given values for some evidence nodes.

DEFINITION 2.3 : INFERENCE

Inference, following [1], in a BN, represented by a DAG $G = <V, E>$, is the process of calculating posterior probabilities of a set of variables X (where each variable is represented as a node $v \in V$) with respect to a set of evidence E and can be denoted as $P(x_i|E)$ $x_i \in X$. This process is also known as compilation or probability propagation or conditioning or belief updating.

Cooper [24] has shown that performing inference in an ordinary BN is an NP-Hard problem. Various approaches have been proposed to reduce the computational cost by avoiding combinatorial explosion [42]. The approaches are broadly classified into two types.

1. **Approximate inference:** Refers to which value of x maximizes $P(x|e)$ is calculated provided that x is the query event and e is the evidence in the network. Instead of calculating exact posterior probability, it eliminates very small numbers to reduce computation cost (such as in [121]) to save computation The approaches that explore approximations by stochastic simulation are outlined in [122].

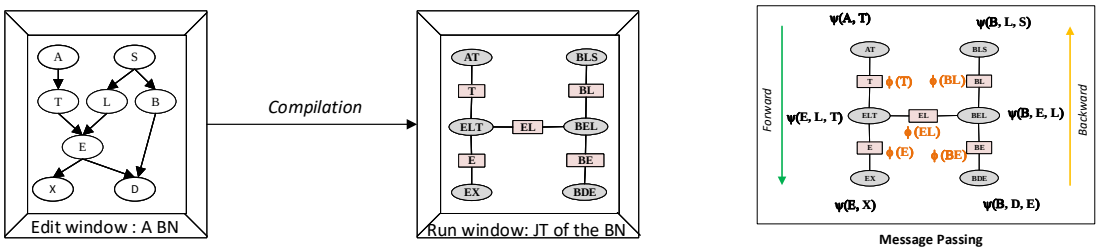
2. **Exact inference:** The approaches that compute exact probabilities $P(x|e)$ (where x is the query event and e being the evidence provided) efficiently by exploiting the structure of the problem [42].

To calculate the whole probability distribution over the variable D in the Asia BN (shown earlier in Figure 2.2), the following equation can be used:

$$P(D) = \sum_{A,T,S,E,C,B,X} P(A,T,S,E,C,B,X,D)$$

$$= \sum_{D} P(D|E,B) \sum_{E} P(E|T,C) \sum_{C,B} P(C,B|S) \sum_{T} P(T|A)P(A)P(S)$$

Approximate inference is beyond the scope of this thesis and so only exact inference is considered here.



(a) (b)

Figure 2.5: Inference in a BN: (a) Compilation (originally depicted in [2]), (b) Message passing in a Junction Tree

One of the widely used existing exact inference approaches was proposed by Madsen et al. and is known as the JT-based approach [25, 123]. In this approach, the inference has two phases as follows:

1. **Compilation:** To performing belief updating in a BN, an intermediate conversion from BN to JT is performed. This conversion is known as compilation (refer to Figure 2.5a).

2. **Probability propagation:** After constructing the JT, it is subjected to a procedure that follows "Message Passing Protocol". This calculates the final posterior probability by propagating the potentials of the cliques throughout the whole JT (See Figure 2.5b).

2.7.1 Junction tree construction

To construct the Junction Tree for a BN, the DAG of the BN is transformed into an undirected graph, and moral edges are added to the graph. Moral edge refers to the edge added between any two of the parent nodes of a node in the DAG of a BN. After adding all the moral edges to the undirected version of the DAG, the graph is triangulated by adding fill-in edges.

After finding the triangulated graph, a set of maximal cliques are detected. The set of cliques form a set of nodes. A clique graph is formed using the set of nodes where the edges for the graph are formed by connecting clique nodes with at least one common variable. The number of common items (variables) between any two clique nodes is referred to as the weight of the edge.

From the clique graph, a maximum (weight) spanning tree is formed based on the weight of the edges. To form this maximum spanning tree, a simple modification of the minimum spanning tree algorithm is sufficient, i.e., instead of minimizing the sum of the weights of the edges, maximise the sum of the weights. Note that the number of edges remains the same for the spanning tree, i.e., for n nodes, $n-1$ edges are added in the tree to connect the nodes. The resultant Maximum Spanning Tree (MST) is then the JT. Figure 2.6 demonstrates the aforementioned JT construction procedure step by step for the Asia BN. For some graphs, more than one

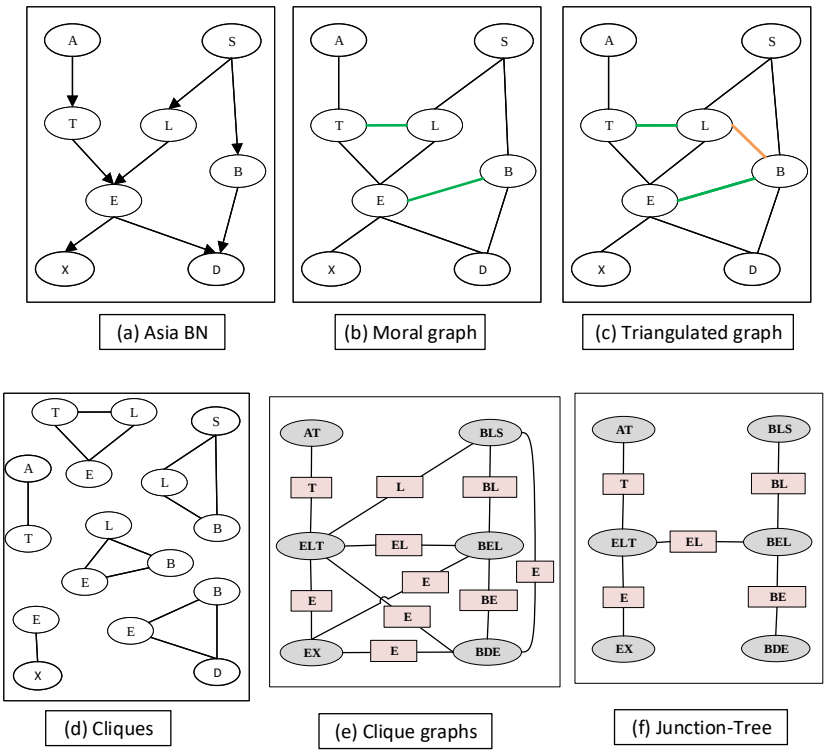


Figure 2.6: (a) The Asia BN, (b) Moral graph for the Asia BN, (c) Triangulated graph for the Asia BN, (d) A set of cliques found from the triangulated graph, (e) The clique graph formed by the cliques, (f) A JT for the Asia BN

MST is possible. Similarly, for some clique graphs, multiple JTs can be formed.

Note that triangulation, especially minimal triangulation, and hence JT construction, are expensive operations regarded as NP-Hard. Hence, for finding minimal triangulation, various heuristics [124] and algorithms [26, 125–128] have been proposed. Some approaches address the issue of finding minimal triangulation and constructing efficient JTs in different ways. These include the maximum cardinality search for computing minimal triangulations [129]; generating the minimal separators and the maximal cliques of a chordal graph [130]; optimal JT construction [131]; vertex elimination order finding [132, 133]; triangulation by retriangulation [134]; applying genetic algorithm [127]; and finding perfect orders [135].

2.7.2 Message passing protocol

To calculate the conditional posterior probability with respect to evidence on a set of random variables, the JT of the BN and a decomposable form of the probability is passed to the message-passing procedure. Each node in the JT contains a cluster of random variables and knows only the local potential of the cluster and its neighbours. Then each node sends a message, comprised of potentials, to all of its neighbours. Each node combines its local potential with the received messages from its neighbours and computes the marginal distribution of its

variables. The message-passing protocol operates in two phases:

1. **Collect message:** The leaf nodes of a JT send a message (i.e., their local potential) to their parents, and the parents pass the message after augmenting it by multiplying their own local potential with the potential in the received message. This upward message passing continues up to the root node and the first phase of message passing, that is, "Collect message" finishes. Figure 2.7 represents the initial stage of the potential distribution for the JT (of Asia BN) nodes before message passing begins. Figure 2.8 demonstrates the "Collect message" phase for the JT of the Asia BN.

2. **Distribute message:** In the "Distribute message" phase, collected messages from neighbours are augmented with their own potential and are then propagated from the root node down to the leaf nodes. Figure 2.9 shows the "Distribute message" phase in a JT of the Asia BN.

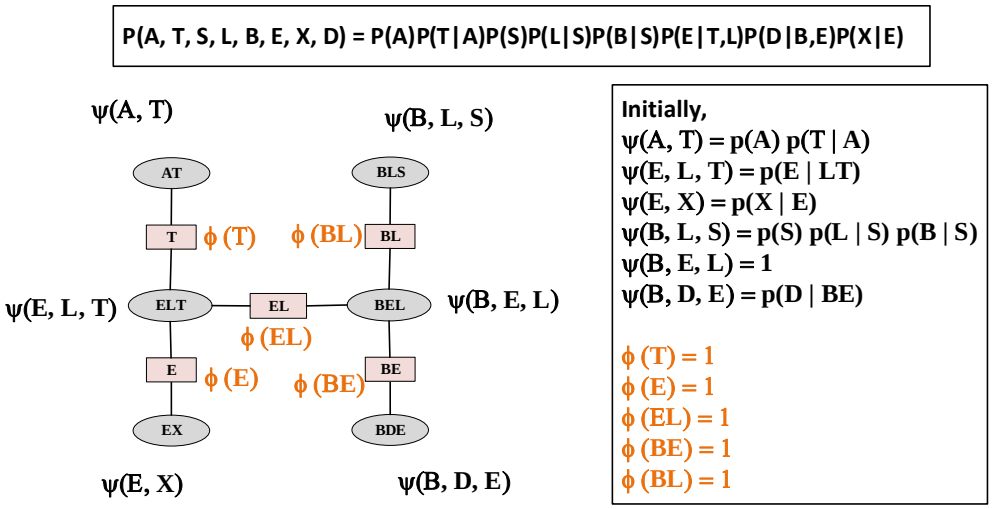


Figure 2.7: Message passing on the Asia BN: (A) Initial potential distribution and assignment on cliques and separators of JT.

Figure 2.10 demonstrates consistency preservation in the local potential distribution after message passing in the JT finishes.

Note that, in the last phase of message passing, a consistency check is performed where the potentials of the cliques on both sides of each separator are same and equal to the potential of the separator. This situation indicates that message passing is complete and no more potential calculation is required. In the meantime, soon after both phases of message passing are complete, the posterior for each of the node of the BN has been calculated. To get the posterior of each node, some simple calculation, such as ratio computation of the potentials of the separators and cliques, is sufficient. Depiction of each posterior calculation based on some particular

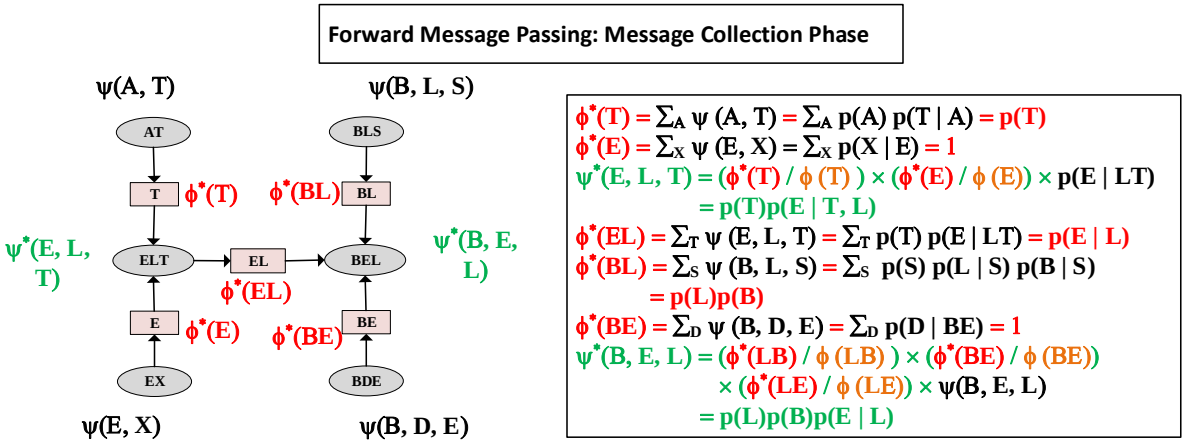


Figure 2.8: Message passing on the Asia BN: (B) Forward message passing phase

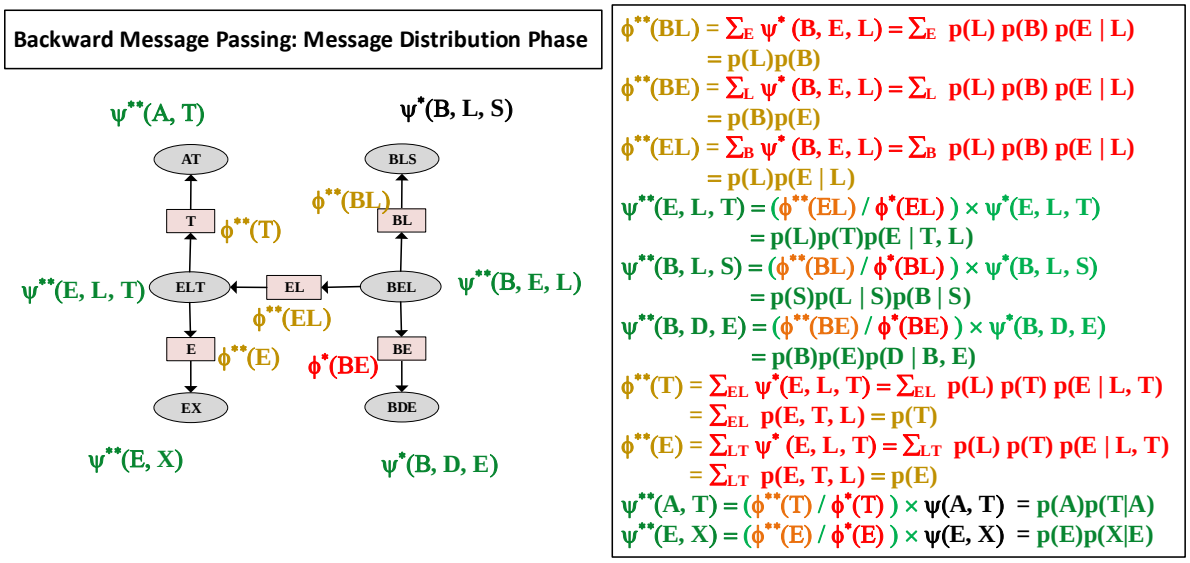


Figure 2.9: Message passing on the Asia BN: (C) Backward message passing phase

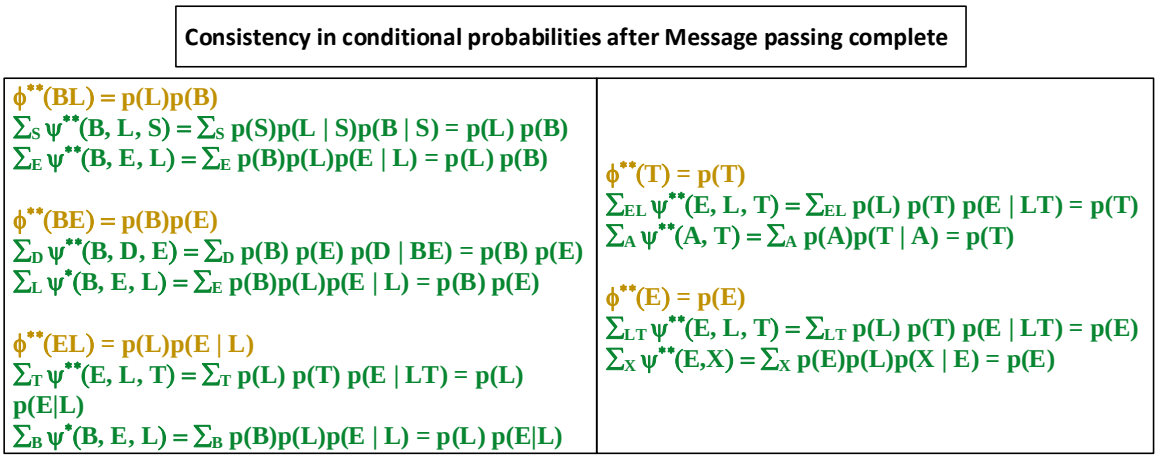


Figure 2.10: Message passing on the Asia BN: (D) Consistency checking of the JT before and after probability propagation.

numeric prior probabilities is beyond the scope of the thesis. Neapolitan demonstrated such computation in detail in [135].

2.7.3 Incremental compilations

To perform belief updating in a BN, it must be converted into a JT. This process is known as compilation. Each modification of a BN requires repeating the compilation steps from the beginning: that means revisiting the steps of JT-based inference which include various computationally expensive operations such as triangulation and clique finding [26].

Motivated by the above-mentioned issue of revisiting expensive operations repeatedly, Flores et al. [4] proposed a variant of JT-based compilation known as incremental compilation (InC). InC does not require rebuilding a new JT every time there is any change in the BN. Flores et al. proposed a Maximal Prime subgraph (MPS) decomposition-based compilation technique [136] where to get an updated JT (following any modification to the BN) does not require performing the steps of "JT-based inference" for the whole BN. It constructs an MPS tree in parallel to JT construction during ordinary BN compilation. Then it keeps track of the changes performed in the last BN structure and marks the parts of the MPS tree that are affected by the changes. Then the marked portion is re-triangulated, and an intermediate JT for the affected portion is constructed which finally replaces the marked portion of the original JT. This new JT corresponds to the changed BN. This compilation technique is particularly useful when the BN is only partially modified.

In brief, MPS [137–139] are the minimal subgraphs that can be triangulated independently. In a model where frequent changes are made, an incremental compilation approach is the most appropriate because it requires no JT construction from scratch, and it saves a considerable amount of time. Changes required are usually limited to alteration (adding or deleting) of nodes and edges. The approach produces a minimal JT for a particular BN, a process which might take a long time if the minimal JT had to be constructed from scratch.

Notation	Ful form
G	DAG representing a BN
G^M	Moralized G
G^T	Triangulated G^M
JT	Junction-Tree of G^T
MPD_JT	MPSD version of G^M
g^M, g^T	Moral and Triangulated graph of affected portion of the original graph due to any change in the BN
jt	Junction-tree of g^T

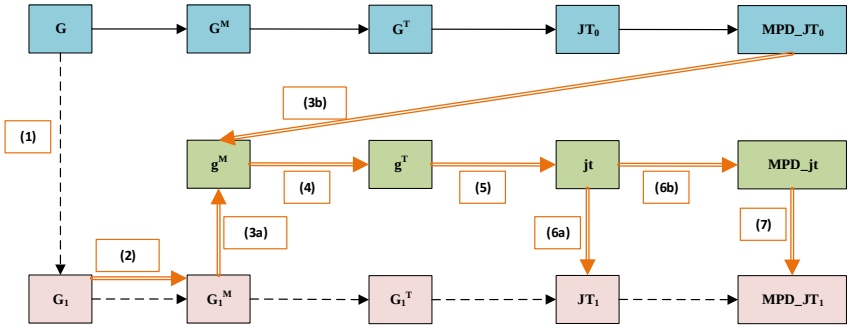


Figure 2.11: Workflow diagram of the Incremental Compilation algorithm

Figure 2.11 shows a high-level view of the "Incremental Compilation" algorithm (redrawn

from [5]). This figure illustrates the steps and their order of execution to provide incremental compilation to a BN. In the algorithm, the authors considered the following four basic operations and defined the steps required to perform them.

1. Adding a node
2. Deleting a node
3. Adding an edge
4. Deleting an edge

Any subsequent operations such as adding or deleting a set of nodes and edges can be transformed into a series of these basic operations. However, any subsequent operations could lead to a set of additional operations from the above list. For example, deleting a node may also lead to deleting a set of edges.

Figure 2.12 illustrates the formation of a JT and MPS decomposition as an intermediate step of incremental compilation for the Asia BN. The DAG of the Asia BN is moralised and triangulated. The triangulated graph is used to find the JT and the MPS tree. In the figure, the MPS tree is followed by MPS decomposition.

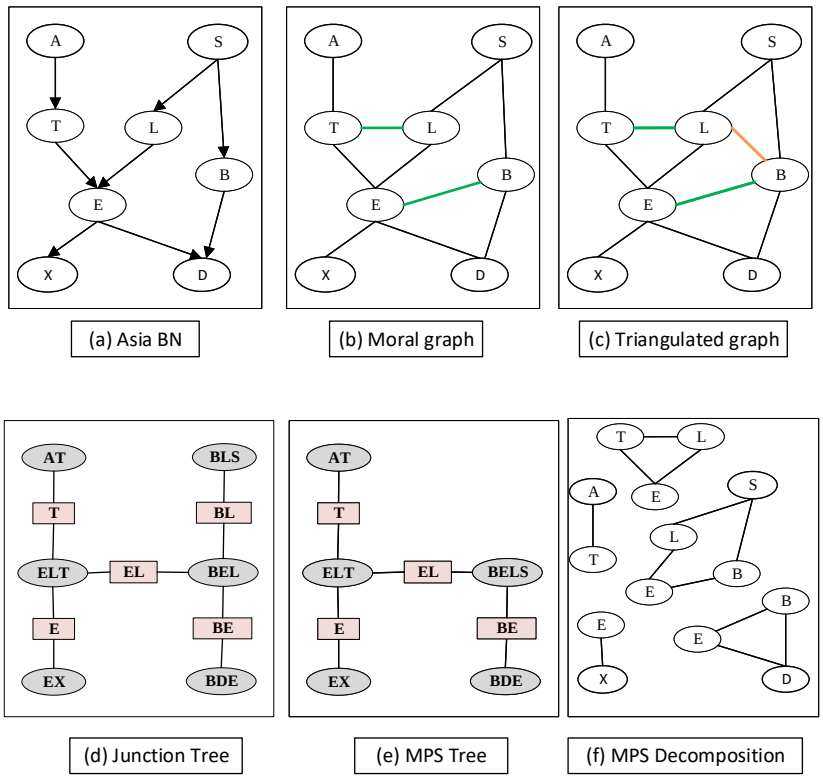


Figure 2.12: (a) The Asia BN, (b) Moral graph for the Asia BN, (c) Triangulated graph for the Asia BN, (d) A JT for the Asia BN, (e) MPS tree for the Asia BN, (f) MPS decomposition for the Asia BN

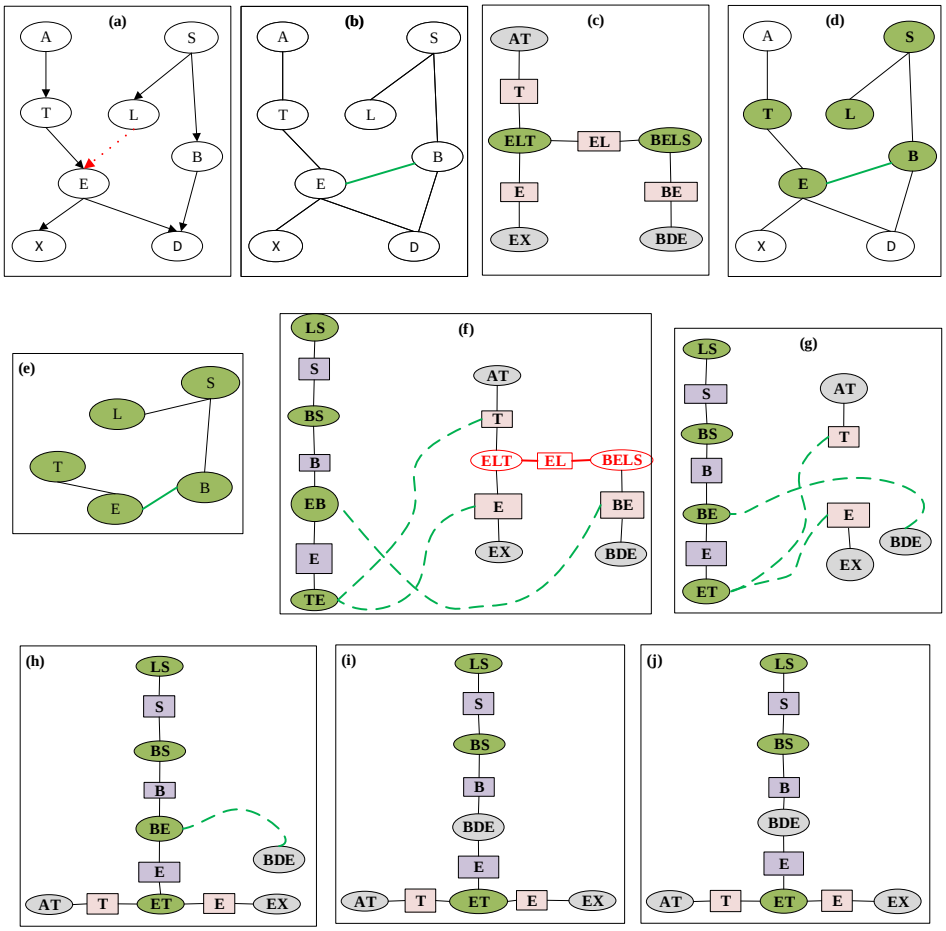


Figure 2.13: InC example of deleting an edge: (a) The Asia BN (Edge "$L \to E$" to be deleted); (b) Moral graph for the modified DAG; (c) Affected portion of the MPS Tree for the Asia BN is marked; (d)–(e) Affected graph portion extracted, moralised, and triangulated; (f) JT construction for affected graph and connecting with MPS tree of the Asia BN; (g)–(i) Constructing the updated JT, (j) Constructing the updated MPS tree.

Figure 2.13 demonstrates how the incremental compilation algorithm constructs a new JT when the edge "$L \to E$" is deleted from the Asia BN (as shown in part (a)) and how the algorithm takes the initial JT and the MPS tree and builds the updated JT from that information. The steps are ordered as per the execution of the algorithm and the corresponding operations performed on the different underlying structures. Part (b) shows the moral graph of the BN after deleting the edge "$L \to E$". In part (c–d) of the figure, the affected portions of the MPS tree and the moral graph, respectively, due to edge deletion in the BN are marked. Part (e) represents the triangulation of the affected graph segment. Part (f) shows a significant operation of the connecting process of the affected JT, formed from the affected part of the MPS tree and moral graph, to the original JT. Parts (g–j) show the process of replacing the affected portion of the JT by the newly formed JT constructed from the affected graph segment and building the final updated JT.

Figure 2.14 represents the steps required in the incremental compilation algorithm to update a JT when a node "Z" and two edges "$A \rightarrow Z$" and "$Z \rightarrow X$" are added to the original Asia BN. In part (a) of the figure, the new BN is depicted. Figure 2.14 (b–j) illustrates the execution of the incremental compilation algorithm, starting with the initial JT and the MPS tree of the Asia BN and ending in building the updated JT. Part (b) shows the updated JT of the Asia BN after adding the node "Z". The new moral graph for the extended Asia BN (after adding the node and the two edges) is shown in part (c) of the figure. In parts (d–e), the affected portions of the MPS tree and the extended BN, respectively, due to adding the node and edges in the BN, are marked. Part (f) represents the triangulation of the affected graph segment. Part (g) shows the connecting process of the JT formed from the affected part to the original JT. Parts (h-j) illustrate the process of replacing the affected portion of the JT by the newly formed JT constructed from the affected graph segment and building the final updated JT.

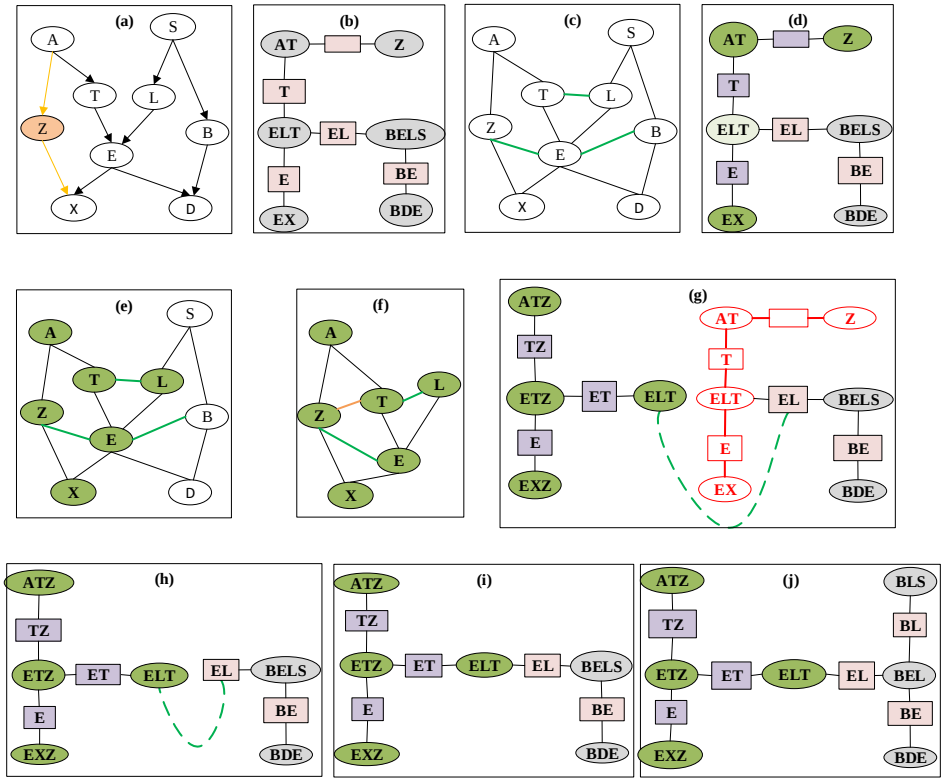


Figure 2.14: InC example of adding a node and two edges: (a) The Asia BN (with node "Z", edges "$A \rightarrow Z$" and "$Z \rightarrow X$" to be added); (b) The JT of the Asia BN after adding node "Z"; (c) Moral graph for the modified DAG; (d) Affected portion of the MPS Tree for the Asia BN is marked; (e)–(f) Affected graph portion extracted; (g) JT construction for affected graph and connecting with the MPS tree of the Asia BN; (h)–(i) Constructing updated JT; (j) Constructing updated MPS tree.

The illustration of the other two basic operations like "adding a node" and "deleting a node" (provided that it requires no edge deletion) are not considered here as those are quite straightforward. Interested readers are referred to the original work proposed in [5].

This compilation technique is particularly useful when the BN is partially modified. If a traditional approach is used instead of InC, expensive operations need to be revisited repeatedly.

2.7.4 Inference in OOBNs

To the best of our knowledge, there is no inference algorithm that works on the OOBN structure itself. Currently, to perform inference in an OOBN, "Flattening" to an ordinary BN is done first, then any of the existing inference approaches (notably, the JT-based approach [25]) can be applied. Even in the most widely used tool, Hugin [103], an OOBN is first flattened into an ordinary BN, and then inference technique is applied. Any change to the OOBN requires starting with flattening the new OOBN. Moreover, if the OOBN supports inheritance, as in [140], where any OOBN class can be derived from another class, then any change in a class in the hierarchy would result in a series of changes to the related OOBN classes, starting by flattening for all of the OOBN classes that were changed.

Although "Incremental Compilation" (InC) [4] could be a potential approach to perform efficient inference in OOBNs, it has not been explored practically yet. Both of these techniques (Hugin's OOBN compilation and InC), however, have some significant issues because of the complexities of these approaches. (Note that InC is still required to use a flattened BN, instead of directly working on an OOBN). Moreover, this approach needs retriangulation of certain portions of the existing structure as well as extra computational and storage burden for the MPS tree structure, besides JT construction. It may encounter some scenarios, as in Figure 2.15, where the whole network structure, or a large portion of the structure, needs to be retriangulated, thus leading to complexity similar to that seen with a traditional flattening approach. Suppose InC is being used to add an edge $X \rightarrow N$ to the network, as shown in Figure 2.15 (left window). The right window of the figure also contains the JT for the initial network with the affected and marked JT segment due to the adding of the edge. The incremental compilation approach affects eight cliques out of ten of the JT. This implies that almost the whole network needs to be re-triangulated, and the marked portion needs to be replaced with the modified portion to arrive at the resultant structure.

In [140] (and outlined in Chapter 3), an extended OOBN framework, iOOBN, has been proposed, based on the findings of this research, where features of the OO paradigm have been exhaustively explored, especially inheritance and polymorphism. In Chapter 4 an algorithm is proposed for inference in OOBNs that does not require flattening. The approach, known as "Shareable and Inheritable Incremental Compilation" (SIIC), allows inference to be performed in an iOOBN while building it in an incremental fashion. It works directly on the OOBN

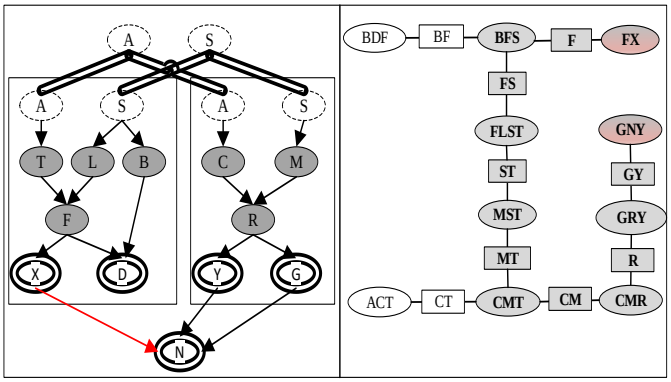


Figure 2.15: A limitation of the IC algorithm.

structure to perform inference of iOOBN classes without flattening. The JT of a class can be shared within subclasses. In concrete classes, partial compilation can be borrowed from its parent class, as well as, from the corresponding classes of the embedded objects. This is a potential way of saving a large amount of computation effort.

2.8 Knowledge Engineering in BNs (KEBN)

This section presents Knowledge Engineering with Bayesian Networks (KEBN) which is a process that consists of the elements of BN technology, automatic discovery of causal structures, learning parameters from data, examples and ideas about how to utilize these techniques in developing expert systems.

The KEBN process is a collection of some methodologies that combines the aforementioned diverse techniques into a single process, i.e. "knowledge engineering", to facilitate the construction of BN models under a variety of circumstances. This means, binding together various techniques and algorithms for building BNs and supplementing them with additional methods adapted from the software engineering field to propose a generalized methodology for the development and deployment of BNs known as KEBN. [1, Chapter 10]

Knowledge Engineering is the acquisition, structuring and refinement of knowledge [141]. The goal of knowledge engineering is to make information accessible to people or computer systems [142].

2.8.1 KEBN methodologies

Deploying BNs to solve Artificial Intelligence (AI) and reasoning-related problems has up to now been limited due to their computational complexity. Hence, the initial focus of the BN community was on discovering efficient inference algorithms to make BN technology compu-

tationally viable.

BN modellers struggle for methods to build and parameterise their systems [143]. Knowledge engineering (i.e., finding human domain experts, extracting their knowledge and putting it into production systems) is a potential avenue for a solution. However, knowledge engineering, especially finding domain experts and encoding their knowledge into BNs, is challenging and this creates a knowledge acquisition bottleneck. To deal with this impediment within expert systems, researchers have transferred their attention to automated learning of BNs (structure and parameters). But very few studies have been published in the field and these only briefly refer to the area of knowledge engineering [10, 144]. Moreover, to date, there have been very few works, such as [1, Chapter 10] and [33] that aim to, develop a methodology for the creation of large-scale BNs to solve real-life problems and provide expert system based knowledge. An approach similar to software development that embraces the principles of Knowledge Engineering is greatly needed. Knowledge Engineering with BNs (KEBN), is one such approach.

KEBN can be performed via three main approaches, namely, (1) expert elicitation and manual construction by hand, (2) automated learning from data, and (3) a combined approach.

Knowledge elicitation is the method of KEBN that involves encoding human knowledge and decision making rules. The encoding is targeted at construction of the causal structure and associating probabilities (CPTs) using the experience and reasoning power of human experts. Both structure building and parameterisation present different kinds and levels of difficulties. For example, CPTs can grow exponentially [145, 146] and graphical structures need a significant amount of time and effort and are not free from human errors. Structure learning involves two primitive jobs to be done, namely, domain variable identification and adding causal relationships to the variables [147]. The more complicated the system/structure is, the better reasoning it can be provided with [143]. Parameterising is the process of eliciting probabilities [143]. It is hard to access human expertise/knowledge. Human knowledge may contain inconsistencies that are unrecognised or unacknowledged [148]. Some techniques have been developed to better parameterise BNs [149]. An early approach involves the use of "bets". Another process of probability elicitation is an analytical hierarchy process, a ranking solicitation process adapted to probability solicitation [148]. The quest for more accurate probability solicitation is ongoing.

Automated BN learning from data is a KEBN process that has attracted the attention of many researchers. In this process, from a set of data, a BN structure is built using various machine learning or pattern-mining techniques. Among various approaches, Causal discovery with MML (CaMML) [150] that follows the Minimum Message Length (MML) principle with

Gibbs sampling [151] is a pioneering work. To deal with the exponential search space in finding the best causal structure, it uses a machine learning approach, namely the "MCMC" (Markov Chain Monte Carlo) greedy algorithm.

The combined approach of KEBN uses automated learning of BNs in association with expert elicitation. Of course, the approach is not free from the risk of combining two different approaches or their outcomes [147], which may lead to the worst consequences. This might be likely to happen when different approaches have different assumptions. In particular, experts' opinions are given in the form of constraints or heuristics (as guidance) to the automated learning algorithms. In [152], such an idea is implemented by O'Donnell.

Although there are some noteworthy (though not comprehensive) works on KEBN, there has been no work on OOBN knowledge engineering methodologies, i.e., "KEOOBN". This thesis proposes an algorithm for automated learning of an OOBN class hierarchy and it can thus be considered as a pioneering work in the KEOOBN domain.

2.8.2 Learning BNs

BN structure and parameter estimations are broadly referred to as "Bayesian Network Learning". Because of the widespread acceptance and popularity of BNs, the associated challenges of modelling large, complex real-life applications by hand, and the need for scarce dependency on human experts to build the networks, researchers have sought to develop methods to automate the learning of BNs.

Constructing BNs by hand requires expert elicitation with domain knowledge. This process is time-consuming, hard to manage without proper domain knowledge and expertise, expensive and the resultant models are often not feasible to extend. Moreover, in domains where no expert knowledge or very little expert knowledge is available, automated learning becomes the only solution for modelling.

Automated learning of BNs is referred to as learning BNs from a set of stored information, data, and knowledge using machine-learning techniques. Automation of BN construction is vitally important to overcome the drawbacks of manual construction. However, automated learning techniques also have limitations. For instance, they require large, clean and complete datasets, and the algorithms devised so far are not as capable of capturing sensible domain knowledge as a human expert. To overcome these limitations, researchers have proposed various techniques, such as "Learning Continuous Time" BNs [153], Learning linear causal models by MML sampling [154], "Learning Bayesian Belief Networks" based on the MDL principle [155], Learning BNs with discrete variables from data [156], variations on the PC algorithm [157], a Bayesian approach to Causal Discovery [158], Learning BNs by unifying

discrete and Gaussian domains [159], Learning causal structure from mixed data with missing values using Gaussian copula models [160], Learning causal models from relational data [161], Causal discovery with weak links and small samples [162], Ordering-based search for learning BNs [163], Learning BNs with restricted causal interactions [164], Learning BN model structures from data [165] and learning the PRMs of link structures [166].

2.8.2.1 Learning structures

In the two main areas of BN learning (structure learning and parameter learning), the more challenging is the learning of structures. Chickering proved that the automated learning of a BN structure is an NP-Hard problem [56]. Parameter estimation can easily be accomplished using a statistical or Bayesian approach [38], once the structure is available or learned. Hence, most emphasis has been placed on learning BN structures. BN structure learning methods fall into three main classes [78].

1. **Constraint Satisfaction Problem (CSP):** this approach looks for conditional (causal) dependencies and uses statistical tests. Then it finds the best fitting graphical structure to represent the available information. In the literature of automated BN learning, lots of algorithms have been proposed to date that follow this strategy. Some very interesting methods include the Inductive Causation (IC) algorithm [167], the Spirtes-Glymour-Scheines (SGS) algorithm [168], and the Prototypical Constraint-based (PC) algorithm [30]. These approaches work well with sparse networks but are prone to fail in statistical independence tests. Hence these techniques are neither scalable nor reliable for large and complex system learning.

2. The **Score-based Optimisation problem (SOP):** this approach tests for a fit of a structure in terms of a score function. Then a structure that maximizes the score is chosen. There are numerous algorithms proposed for learning BNs that follow this approach, such as K2 [169], the Greedy Equivalence Search (GES) [170], the Maximum Weight Spanning Tree (MWST) [171], and the Causal MML (CaMML) [150].

3. **Hybrid approach (SOP and CSP):** this approach, pioneered by Singh and Valrota [172], is a combination of the two approaches as mentioned above. It is scalable and superior to the previous two approaches. Some popular algorithms that follow this method are the feature selection based algorithm [173], and the Benedict algorithm [174].

2.8.2.2 Learning parameters

Inference on unobserved variables, parameter learning, and causal structure learning are the three most critical operations in a Bayesian network. In the previous section, BN structure learning techniques are briefly described. However, without parameterization, the structure of the BN is useless. With the intention of completing a BN model (i.e., to fully define the joint probability distribution), it is necessary to specify the probability distribution of each node in terms of its parents [31,38].

The distribution can be a Gaussian distribution or a constraint-based distribution determined by the principle of maximum entropy. Often the constraint can be missing parameters or incomplete data. Based on the degree of completeness of the data, parameter learning techniques can be divided into two classes. According to [175], the classes are:

1. **Algorithms that work on complete data:** (a) Maximum likelihood estimate [176] and (b) Bayesian method [177].

2. **Algorithms that work on incomplete data:** (a) Expectation-Maximisation [178] (deterministic approach) (b) Gibbs sampling [179] (a stochastic approach) (c) The Monte Carlo method [180] and (d) Gaussian approximation method [181].

As a detailed discussion on the methods for learning parameters is beyond the scope of the thesis, readers are referred to the works cited above for more information.

2.8.3 Learning OOBNs

The trend of using OOBNs is steadily increasing. People are turning to modelling applications with OOBNs that need probability analysis and have uncertainty associated with them. This interest is due to the facilities of the OO paradigm provided in an OOBN. Therefore, it is imperative to have some methodologies to learn OOBNs automatically from a set of data, for the same reasons that automated learning of BNs is desirable (see the preface to Section 2.8). However, to the best of our knowledge, there has been almost no work done on automated learning of OOBN structures from data. However, Langseth et al. proposed a parameter learning approach for OOBNs from data [182] and Bangsø et al. proposed a method for structure learning in OOBNs [183]. The structure learning algorithm requires expert elicitation/opinion and can only be partially automated. Moreover, these approaches are not widely used in practice.

This deficiency is not very surprising, as the first OOBN was only proposed in 1997, and the first commercially developed OOBN was released in Hugin [103] in 2003. Hence, OOBNs have only recently started to emerge into the mainstream. An extensive literature survey has been undertaken for this research and it confirms the urgency of developing a method to learn

OOBNs from data. Before taking "hierarchy learning" into consideration, the initial interest for the present research was focused on the structure learning of OOBNs. In that context the following Heuristic 1 was developed:

Heuristic 1. *If BN structure (a DAG) could be constructed from data, then frequent induced subgraphs of the graph (DAG) can be mined using any of the well-known graph mining algorithms such as [184–186]. The mined subgraphs can be fed to a machine-learning algorithm (yet to be devised) to suggest some classes, one for each such frequent subgraph where instances of the classes could replace all of the occurrences of the subgraphs.*

This heuristic needs more work, such as developing a framework to prove its admissibility, accuracy and effectiveness. Moreover, the approach also needs an extensive investigation to check whether the OOBN classes built by this approach are robust enough not to fail in an "independence test" and other property tests such as local Markov property test, and the faithfulness test.

These matters are left for future research work.

2.8.3.1 Learning hierarchy

OOBNs are new and not widely used as BNs. There are some reasons for this, such as not all features of OOBNs (especially "inheritance") being available in any of the existing software, and that some OO features are hard to understand by users, especially if they do not belong to Information Technology or Computer Science, have a little or no background in computer programming. Moreover, people who are interested in working on OOBNs after realising their merits find it challenging to convert their existing BN projects and BN repositories into OOBN projects and repositories.

Therefore, devising a mechanism to convert ordinary BN repositories into OOBN repositories seems more important than OOBN structure learning. An exhaustive search in the literature resulted in the conclusion that there has been no work done on automated learning of OOBN hierarchies or the conversion of ordinary BN repositories into OOBN repositories.

Some related studies in other domains (i.e., software engineering and Object-Oriented programming) have been undertaken to date. The proposed methods are for maximising reuse and factoring the codes written in OO programming language to build software. Manually designing an inheritance hierarchy that has maximised factoring is very difficult, particularly if the system is large, complex and has been constructed by a group of people working in a distributed manner. Moreover, sometimes, a well-designed system may not be optimal in terms of factoring, or it may lose factorisation due to maintenance, extension or upgrading [187]. To address these issues and to resolve them, researchers have developed various techniques such as

the evolution of inheritance hierarchy [187], the automatic restructuring of hierarchy [188], the automatic inferring of inheritance [188–195], to mention a few of those in the current software development arena.

The aforementioned studies have added a new dimension to automated hierarchy learning in OOBNs. If there is already a hierarchy of classes for an OOBN project, automated hierarchy learning can assist in checking for its effectiveness and efficiency in terms of factoring and reusability and also to check if it can be improved or whether its hierarchy can be better designed. Moreover, reshaping of a previously built hierarchy that has been updated and extended in a non-organised way can be done with such a method.

Heuristic 2 below might support automatic learning of the hierarchy of OOBN classes.

Heuristic 2. *If a BN repository can be treated as a set of DAGs and an order between the DAGs can be formed where a graph comes after the subgraphs of the graph itself in the order, then this order suggests a hierarchy*

In Chapter 5 an algorithm to find such an order by constructing a supergraph from the DAGs of the OOBN classes is proposed. There have been numerous works done on graph to find efficient algorithms such as constructing supergraphs [196], finding subgraphs by checking subgraph isomorphism [197] efficiently, and counting the frequency of subgraphs in a single large graph [185], or from a set of graphs [186, 198, 199]. The proposed algorithms and related works in the aforementioned works have helped in devising the approach of constructing/learning a hierarchy in Chapter 5.

2.9 Summary

This chapter presented the historical background of probabilistic analysis, various probabilistic models, their families, pedigree, properties, features, limitations and advantages. The main focus of the discussion was on the original formulation of the BN, its purpose, limitations and ways of extending the original formulation to overcome the limitations. The limitations of the BN are divided into two main classes, and corresponding remedial models are suggested and discussed. A summary table represents the core of the discussion in a nutshell. This is followed by a list of existing software for modelling applications that have uncertainty associated with the related events. The discussion also highlights the importance of OO features in BNs and how these overcome the issues of BN modelling efficiently. The chapter also emphasises the shortcomings of the existing frameworks and why those should be addressed in new frameworks or software and then outlines how BNs are used in order to reason under uncertainty. The discussion is followed by a discussion of the types of reasoning and inference, and

how the types of reasoning vary and the usefulness of the variations in probabilistic analysis. Then types of inference, the steps and actions required to perform inference in a BN, and how to efficiently perform the operations are discussed. An approach to overcome the issues of traditional inference, incremental compilation, is suggested and its mechanisms and potential problems are outlined. How the compilation process, required for inference, can be done more efficiently, is discussed next. Finally, the importance of automated learning of BNs (parameter and structure) are presented with a pair of heuristics suggesting approaches to automatic learning of OOBN structures and hierarchy. This chapter concludes with the importance of learning (and reshaping) an inheritance hierarchy and a discussion on how the learning of hierarchy can be performed.

iOOBN Framework

The construction of Bayesian decision networks (BNs) [1] to model large-scale real-life problems is challenging. Given that most or all of the model is built by hand, BN modelling methods do not scale up well; the resultant large and complex BNs are difficult to visualise and hard for the domain experts and decision makers to understand, reducing the acceptance and subsequent use of the model. One useful approach to scaling up is the Object-Oriented Bayesian decision networks (OOBNs). This innovation provides modellers with the ability to define classes and develop a compositional and hierarchical structure, enabling reuse of the model and supporting maintenance. However, as discussed in chapters 1 and 2, there are limitations in existing OOBN frameworks. In the OO programming paradigm, a key concept is inheritance, the ability to derive attributes and behaviour from preexisting classes, which enables an even higher level of reusability and scalability (as shown in Section 3.5.2 by demonstrating the reengineering of WGR DOOBN). The problem is that inheritance in OOBNs has yet to be fully defined and implemented. This thesis therefore presents iOOBN, a framework that provides a fully defined inheritance for OOBNs, including concepts such as polymorphism, and abstraction, as well as encapsulation. This chapter provides guidance on modelling in the iOOBN and demonstrates its applicability by referring to several problems in the literature illustrating how OOBN (without inheritance) has been applied. It also discusses a case study of reengineering an existing, large complex OOBN, namely the WGR [3, 57] DOOBN system. A prototype version of the framework has been implemented using Java programming language and the HUGIN-Expert API; see Appendix A).

3.1 iOOBN: Informal Overview

This chapter describes the iOOBN components and outlines a proper treatment of all the features of the OO paradigm, especially inheritance. The definition of OOBNs used in [200]

[1] The acronym BN covers the Bayesian network and Bayesian decision network interchangeably. Similarly, the acronym OOBN stands for the object-oriented Bayesian network or object-oriented Bayesian decision network throughout the whole thesis unless stated otherwise.

and [1], and implemented in Hugin are extended, adopting the terminology and notation used by Hugin. In this section, an informal summary of the terminology and definitions of iOOBN, as originally presented in [140] is provided. More formal treatment is given in the following sections.

The ordinary BN definitions of chance nodes (representing random variables), decision nodes and utility nodes, directed arcs, and conditional probability tables (CPTs), as presented in Chapter 2, are assumed.

The main components of an iOOBN, the proposed Object-Oriented BN framework, are "class" (the blueprint of objects) and "object" (an instance of a class). An iOOBN model is itself an object of a top-level class consisting of nodes, *objects* (which are instances of other classes), and edges. The aim of building the model is that this top-level class can be compiled and run to perform reasoning.

As with standard BNs (see Section 2.3.1 of Chapter 2), all the nodes in an iOOBN class are named and are either chance nodes, decision nodes or utility nodes. *Chance nodes* represent random variables. They have a set of states, a CPT, and are represented by an oval shape. *Decision nodes* represent the possible actions from which a decision-maker must choose. They have a set of actions and are represented by a rectangular shape. Finally, there is the *utility node*, which has an associated utility table (representing a utility function over its parent chance and decision nodes) and is represented by a diamond shape.

In an iOOBN class (as in ordinary OOBNs) there are four types of edges or links: causal edges, information links, precedence links, and referential edges. *Causal edges* are the standard edges in a BN; they can be from a chance node to a chance node, from a chance node to a utility node, decision to chance, or from a decision node to a utility node. An *information link* is represented by a dotted line arrow that run from a chance node to a decision node; it indicates when a chance node must be observed before a decision is made (see Section 2.3.3). A *precedence link* is also represented by a dotted line arrow and is from a decision node to a decision node; it represents the sequence order between decisions. Finally, *referential edges* are used to connect nodes to nodes within objects and are represented by double dotted lines. For two nodes to be linked by a referential link, they must either be both chance nodes with the same states or both decision nodes with the same actions. When an iOOBN is converted into an ordinary BN, so-called "flattening" (see Section 3.2.1), nodes that are joined by referential links are represented by a single node. The edges within an iOOBN must be such that it flattens out to a valid BN, that is, a DAG.

Nodes in classes (and hence objects) in iOOBN (as in ordinary OOBNs) are categorized as input, output and embedded nodes. The *input* and *output nodes* of a class are also called its

interface, while the nodes other than interface nodes are *embedded nodes*. A class can also contain objects, or instances of other classes. An object is connected to a node in the class in which it sits (called its *encapsulating* class) if there is a referential link between the node and an input node in the object's interface. A node in the interface can only have one referential link to a node outside the object. In addition, there may be standard links from an object's output nodes to embedded or output nodes outside the object. If an input node of an embedded object (of a concrete class) is not connected via a referential edge, it must have a default CPT for chance nodes (as per [13]), default decision table for decision nodes, and default utility table for utility nodes; if connected via a referential link, the CPT/decision table/utility table of the connected node overrides the default one.

There are two types of classes: abstract and concrete. Concrete classes correspond to classes as defined in previous formulations of the OOBN. In contrast, *abstract classes* are not fully parameterized, i.e., cannot be used for reasoning. Furthermore, iOOBN has a special data structure called an *interface* that contains the interface nodes of a class without any parameter.

An iOOBN allows the creation of new classes (subclass) from existing classes and interfaces and new interfaces (sub-interface) from existing interfaces. This process is known as "*inheritance*" in the OO-paradigm. Here the new iOOBN component, i.e., class or interface, contains all the graphical structure and parameters (if any) of the existing graphical structures with additional required components (nodes or edges) and parameters if class inheritance is used (see Section 3.3 for type of inheritance). Moreover, if interface inheritance is used, then any embedded node can be shared, any embedded node can be skipped from sharing and all the interface nodes must be shared in subclasses. The iOOBN also allows substituting an instance of a subclass by an instance of its superclass. In the OO paradigm, this is known as polymorphism.

3.1.1 Livestock farming example

The livestock farming example of the "Old McDonald" (OMD) OOBN [201] is used here as a running illustrative example, with extensions where required to capture components of the iOOBN framework. In OMD, the farm has several relevant classes ("cow", "milk", "drafting", "calving"), augmented with decision and utility nodes to model farming decisions based on profit-making.

Figure 3.1 presents the "Profit" iOOBN class including all the required elements of iOOBN (and the notation) that have been presented informally above. A table shown upper-left contains a notation table of the components where dotted, double-lined and shaded shapes are for input nodes, output nodes and embedded nodes, respectively. Ovals, rectangles and dia-

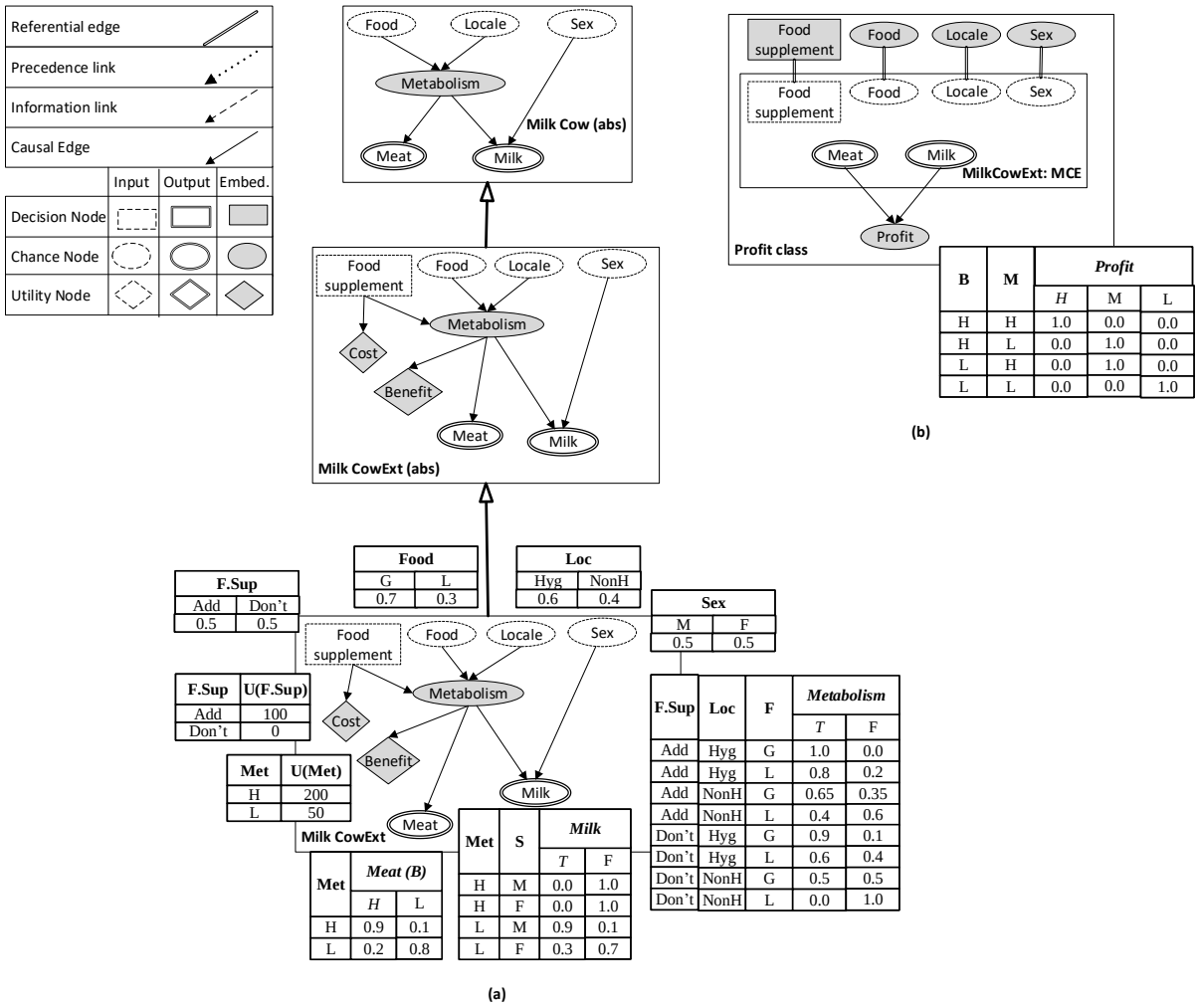


Figure 3.1: An example of the iOOBN class "Profit". (a) An abstract class "Milk Cow" is extended to another abstract class "Milk CowExt". A concrete class is derived by adding required CPTs, decision tables and utility tables, (b) An object of "Milk CowExt" concrete class is used to make "Profit" (concrete) class.

In the tables of the figure, the CPTs are represented with **C: Cow**, **M: Milk**, **F: Food**, **Loc: Locale**, **Met: Metabolism**, and **P: Profit**; the decision table is represented by **F.Sup: Add Food Supplement?**; and the utility tables are represented by **Cost** and **Benefit**. Furthermore, the states are represented with *H: High*, *L: Low*, *Med: Medium*, *Male: Male*, *Fem: Female*, *Hyg: Hygienic*, *NonH: Non-hygienic*, *Don't: Don't add*, and *G: Good Quality*.

monds are used for, respectively, chance, decision and utility nodes; double-lined undirected edges, solid directed lines, dashed and dotted lines are for referential edges, causal edges, information links and precedence links, respectively. Part (a) of Figure 3.1 contains a hierarchy of classes. The root of the hierarchy is the abstract class "Milk Cow (abs)" from the extended OMD farm. This class has three input chance nodes to represent the factors that have impact on milk production, namely $Food$ provided to the cows, $Locale$ of the farm and Sex of the cattle; an embedded node $Metabolism$ that represents the general health of the cattle; and two output chance nodes to represent the production of each animal, namely $Meat$ and $Milk$.

The "milk" class is extended in "Milk CowExt (abs)" abstract class by adding an input decision node "*Food supplement*" to help in deciding whether adding food supplements in the cow's food would improve the animal's metabolism, and hence increase milk or meat production. There are two utility nodes, namely "$Cost$" for the cost of the food supplement and "$Benefit$" for the expected benefit of good metabolism. In order to use the class, say, by instantiating an object, a fully parameterized class needs to be built. Hence, the abstract class "MilkCowExt (abs)" is parameterized in the concrete version of the class, i.e., "MilkCowExt". An instance, "MCE" of the class "MilkCowExt" is used to model a class "Profit" (as shown in part (b) of the figure) to predict the expected level of profit.

3.2 iOOBN: Classes, nodes, edges, parameters and objects

The formal iOOBN definitions and terminologies are now given, starting with the components that are also present in ordinary OOBNs. This section should be considered in conjunction with Section 3.1 where an informal description is given that maintains a sequential order of the components and their dependency, and Section 3.3 where examples are given showing how to build a class and use various components and features. In addition, we would expect people to be using the software with the GUI interface that gives more support (and examples) of how to create an iOOBN model, rather than reading formal definitions.

First, a partial definition of a class is given, then each of its components is defined in turn. Note that as per the informal formulation in Section 3.1, each iOOBN class and each node in an iOOBN class has a unique and valid name that is used as an unique identifier in the iOOBN model. Moreover, an iOOBN class may also contain the name of it's parent class and a list of names of it's parent interfaces. However, the names are more implementation-related and not important for theoretically defining the framework. Hence, the definitions in this section do not contain any name information.

DEFINITION 3.1 : CLASS

An iOOBN **Class** $\mathbb{C}$ is a Directed Acyclic Graph (DAG) given by a 4-tuple $< \mathbb{N}, \mathbb{E}, \Pi, \mathbb{O} >$, where

(i) $\mathbb{N}$ = a set of nodes (defined in Definition 3.2)

(ii) $\mathbb{E}$ = a set of edges (defined in Definition 3.8)

(iii) Π = a set of parameters (defined in Definition 3.13)

(iv) $\mathbb{O}$ = a set of **objects** (defined in Definition 3.16).
Here, each $o \in \mathbb{O}$ is **encapsulated** within $\mathbb{C}$, and $\mathbb{C}$ is called the **encapsulating** class of the object o.

As the iOOBN is a type-based framework, the type of BN/OOBN nodes and OOBN classes play a significant role in the overall framework.

DEFINITION 3.2 : IOOBN NODES

The set of **iOOBN Nodes** of class $\mathbb{C}$ is $\mathbb{N}$ [a] = $N_{In} \uplus N_{Em} \uplus N_{Out}$ where N_{In}, N_{Em} and N_{Out} are mutually disjoint [b], and

(i) $N_{In} = N^C_{In} \uplus N^D_{In}$ = a set of **input** nodes, where
(a) N^C_{In} = A set of input chance nodes
(b) N^D_{In} = A set of input decision nodes

(ii) $N_{Out} = N^C_{Out}$ = a set of **output** chance nodes,

(iii) $N_{Em} = N^C_{Em} \uplus N^U_{Em}$ = a set of **embedded** [c] nodes, where
(a) N^C_{Em} = A set of embedded chance nodes
(b) N^U_{Em} = A set of embedded utility nodes

Any node $n \in (N_{In} \uplus N_{Out})$ [b] is called an **Interface** node.

[a] The attributes of chance, decision and utility nodes are well-defined in the literature and in Section 2.1.1 and 2.1.2

[b] n sets $A_1, A_2, \ldots A_n$ are mutually disjoint if $A_i \bigcup A_j = \varnothing$ for $i \neq j$ and denoted as $A_1 \uplus A_2 \uplus \cdots \uplus A_n$. Note that if the set of input and output nodes are not mutually disjoint then a node can be marked as both input and output node and that does not add anything in the modelling. Because, an input node n_{In} connected via a referential edge is same as the external node n on the other side of the referential edge. Hence, any purpose of additionally marking n_{In} as output node (say, the node becomes n_{In_Out}) can be served without doing so. As an instance, any edge $n_{In\_Out} \rightarrow m$ going out from that node (marked as both input and output) to an external node m can be replaced by a direct edge $n \rightarrow m$.

[c] Embedded nodes are elsewhere called Hidden attributes [46] and Protected nodes [5].

DEFINITION 3.3 : TYPE OF A NODE

The **Type** of a node (following [27]) is a 4-tuple < $Sub\text{-}type$, $StateSpace$, $Category$, $Kind$ > where

- $Sub\text{-}type$ is one of the following four attributes: "Chance", "Decision", "Utility"
- $StateSpace$ is the number of states of the node
- $Category$ is either "Discrete" or "Continuous" if the node is a "Chance" node [a]
- $Kind$ is one of the following four attributes ("Boolean", "Categorical", "Ordered" and "Number") if the node is a "Chance" node

[a] "Discrete" represents that the random variable has a discrete state-space, where "continuous" is for the random variables that are continuous and has acquired discrete state space.

Type of node is required to check for compatibility of interface nodes of an embedded object (an OOBN class) and the external embedded nodes connected to the interface nodes. Note, there is another "type" definition in any iOOBN, i.e., type of an OOBN class. This is required for polymorphism and dynamic typecasting/changing. Knowing the "type" of a class is essential for testing the compatibility of classes/objects to replace an object of one class by an object of a type-compatible class.

DEFINITION 3.4 : TYPE OF AN OOBN CLASS

The **Type** of an OOBN class $\mathbb{C}$ is a 3-tuple < $ParentClass$, $ParentInterfaces$, $InterfaceNodes$ > where

- $ParentClass$ is the class from which the class $\mathbb{C}$ is extended/derived,
- $ParentInterfaces$ is the set of interfaces the class $\mathbb{C}$ implements, and
- $InterfaceNodes$ is the set of input and output nodes of the class $\mathbb{C}$

Similar to the ordinary BNs, all the nodes in an iOOBN class have a name and a type. The three categories of the iOOBN nodes, namely, input, output, and embedded, are defined as follows.

DEFINITION 3.5 : INPUT NODES

In iOOBN, **Input Nodes** N_{In} of class $\mathbb{C}$ is a set (i.e., $|N_{In}| > 1$) of nodes that have no parent node in $\mathbb{C}$ and are only accessible by nodes from the external nodes through referential edges.

DEFINITION 3.6 : OUTPUT NODES

In iOOBN, **Output Nodes** N_{Out} of class $\mathbb{C}$ is a set (i.e., $|N_{Out}| > 1$) of nodes that are the only nodes of $\mathbb{C}$ that can be the possible parents of the nodes in any encapsulating class of an object representing $\mathbb{C}$.

DEFINITION 3.7 : EMBEDDED NODES

In iOOBN, **Embedded Nodes** N_{Em} of class $\mathbb{C}$ is a set of nodes that are neither accessible by the external nodes nor visible from outside of class $\mathbb{C}$.

The four types of iOOBN edges: causal edges, referential edges, information links, and precedence links are defined below. Note that two of the edge types are named as "edges" and two as "links". The reason behind using the two different terminologies are that edges refer to the connections in a mathematical graph to represent a relationship between the terminal nodes whereas links are the connections that represent an ordering between terminal nodes.

DEFINITION 3.8 : IOOBN EDGES

The set of **iOOBN edges** is $\mathbb{E} = E_c \cup E_r \cup E_i \cup E_p$ in a class $\mathbb{C}$ where

- E_c is a set of **causal** edges
- E_r is a set of **referential** undirected edges
- E_i is a set of **information** links
- E_p is a set of **precedence** links

The iOOBN edges, except referential edges which are only relevant to OOBNs/iOOBNs, are also defined in Definition 2.2 of Chapter 2. That chapter describes the attributes, functions and purposes of BNs with an example network.

For each node $n \in \mathbb{N}$, in $\mathbb{C} =< \mathbb{N}, \mathbb{E}, \Pi, \mathbb{O} >$, **parents** $par(n)$ is a subset of $\{\mathbb{N} \cup \bigcup_{o \in \mathbb{O}} N_{out}\}$, where N_{out} is the set of output nodes of the embedded objects $o \in \mathbb{O}$.

DEFINITION 3.9 : CAUSAL EDGES

In iOOBN, the set of **causal edges** is $E_c = \{e : n_i \rightarrow n_j\}$ (i.e., a set of directed edges) in class $\mathbb{C} =< \mathbb{N}, \mathbb{E}, \Pi, \mathbb{O} >$, where

- n_i is an output chance or decision node of an object in $\mathbb{C}$ or $n_i \in \{N^C_{In} \cup N^C_{Out} \cup N^C_{Em} \cup N^D_{In}\}$ (any chance node or input decision node), and
- $n_j \in \{N^C_{Out} \cup N^C_{Em} \cup N^U_{Em}\}$;

DEFINITION 3.10 : REFERENTIAL EDGES

In an iOOBN, the set of **referential edges** is $E_r = \{e : n_i \nleftrightarrow n_j\}$ (i.e., a set of undirected edges) in a class $\mathbb{C} =< \mathbb{N}, \mathbb{E}, \Pi, \mathbb{O} >$ where n_i and n_j have the same state-space and type, and $n_i \nleftrightarrow n_j$ is a one-to-one connection,

(a) $n_i \in N_{Em} \cup N_{In}$ and n_j is an input node of an object in $\mathbb{C}$; or

(b) n_i is an output node of an object in $\mathbb{C}$ and n_j is an input node of another object in $\mathbb{C}$ [a]. Further, referential edges to N_{In}^C are optional, however referential edges to all N_{In}^D are required.

[a] Note that this connection is not allowed between the output nodes of one encapsulated object and the input nodes of another, because Hugin makes this constraint. However, it is an artificial constraint and can be easily overcome by going via an additional chance node in C, as done in [202], at the expense of modelling complexity.

Note that when n_i and n_j are joined by a referential edge, this implies that they represent the same variable in the flattened version (same chance node, same decision node or same utility node), which must be taken into account in any inference algorithm. This implication is essential in the compilation algorithm proposed in Chapter 4. *Referential edges* are used to connect nodes to nodes within objects and are represented by double dotted lines. For two nodes to be linked by a referential edge, they must be both chance nodes with the same states and types, or both decision nodes with the same actions, or both utility nodes of the same type. When an iOOBN is flattened into a BN, nodes that are joined by referential edges are represented by a single node. The edges within an iOOBN must be such that it flattens out (see Algorithm 3.1) to a valid BN, that is, a DAG.

DEFINITION 3.11 : INFORMATION LINKS

In an iOOBN, the set of **information links** is $E_i = \{e : n_a \dashrightarrow n_b\}$ (i.e., a set of directed edges) in a class $\mathbb{C} =< \mathbb{N}, \mathbb{E}, \Pi, \mathbb{O} >$, where

(a) n_a is an output chance node of an object in $\mathbb{C}$ or $n_a \in \{N_{In}^C \cup N_{Out}^C \cup N_{Em}^C\}$ (any chance node), and

(b) $n_b \in \{N_{In}^D\}$;

In other words, information links are directed edges from a chance node to a decision node. An information link $n_a \dashrightarrow n_b$ indicates that chance node n_a must be observed before making any decision for node n_b. Note that the term "information link" in a decision network is standard terminology. A decision node without an information link is also possible and functions accordingly.

DEFINITION 3.12 : PRECEDENCE LINK

In an iOOBN, the set of **precedence links** is $E_p = \{e : n_a \dashrightarrow n_b, (n_a, n_b) \in N_{In}^D \text{ and } n_a \neq n_b\}$ i.e., a set of directed edges representing an order between the nodes in N_{In}^D in a class $\mathbb{C} =< \mathbb{N}, \mathbb{E}, \Pi, \mathbb{O} >$, where $n_a \dashrightarrow n_b$ represents an order $n_a \prec n_b$.

A precedence link $n_a \dashrightarrow n_b$ indicates that decision node n_a precedes decision node n_b in the whole decision-making process (see Section 2.1.2) in the network. This precedence plays a significant role in calculating the expected utility of a decision node in the network.

Note that there is a potential threat of conflict between the orders of decision nodes defined within an object (the order defined by a modeller during building the corresponding class) and the external decision nodes that are connected via referential edges. In an iOOBN the default order defined in the class is overridden by the order of external decision nodes connected via referential edges. This was the motivation behind requiring all N_{In}^D nodes in embedded object classes to have referential links, and it ensures that all the decisions are pushed up to the outermost external class, where precedence ordering on all the decision nodes at that level must be given. This also has the additional benefit of enforcing information hiding in the embedded objects. See Figure 3.2 as an illustration of order overriding.

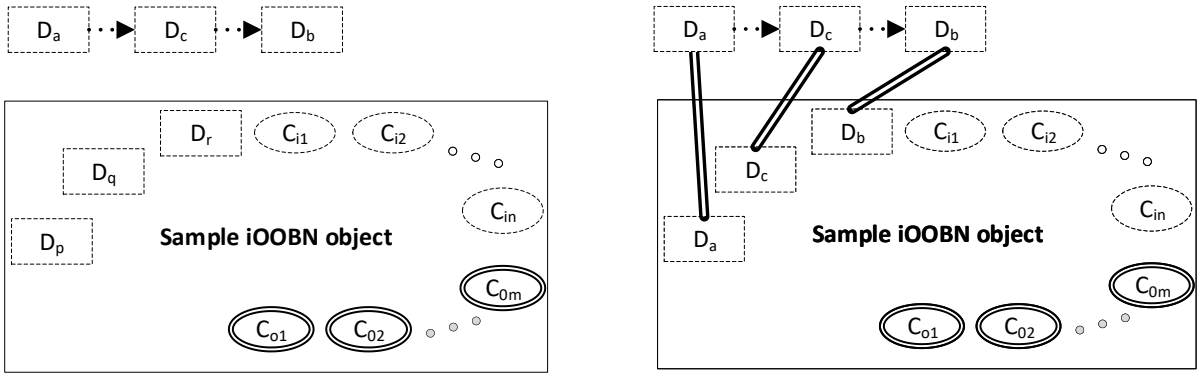


Figure 3.2: Illustration of overriding the order of decision nodes

In Figure 3.2a, there are three external decision nodes in the order $D_a \prec D_c \prec D_b$ and three input decision nodes in the sample object. When the input decision nodes are connected with the external decision nodes via referential edges, the order of the object's input decision nodes are overridden (see in Figure 3.2b) [2].

Table 3.1 summarises the notation and style that is followed to denote various nodes, edges and links in an iOOBN unless otherwise stated.

[2] If there are two different orders in decision nodes connected via referential edges, one order in the external decision nodes and another in the input decision nodes of an object, then there is an inconsistency in the overall ordering of the decisions. This inconsistency would lead to complications in computing expected utility from the network.

DEFINITION 3.13 : PARAMETERS

In an iOOBN class $\mathbb{C} = < \mathbb{N}, \mathbb{E}, \Pi, \mathbb{O} >$, the parameter $\pi \in \Pi$ associated with a node $n \in \mathbb{N}$ w.r.t. the parent chance nodes and parent decision nodes (if any) is either of the following.

- A conditional probability distribution (CPD) if n is a chance node. Note that the CPD $P(n|par(n))$ is a function Φ: $par(n) \cup \{n\} \rightarrow [0:1]$.
- A decision table if n is a decision node. The decision table states a function Ψ: $par(n) \cup \{n\} \rightarrow o$, where $o \in O$ is an outcome from a set of plausible outcomes.
- A utility table representing a utility function if n is a utility node (see Section 2.1.2) .

Table 3.1: iOOBN node and edge representation

Locality	**Element Class**	**Notation (for a set)**	**Shape**	**Format**	**Textual**
Input	Chance	N^C_{In}	Circle	Single Dotted line	Italic
	Decision	N^D_{In}	Rectangle		
Output	Chance	N^C_{Out}	Circle	Double-lined	Bold face
Embedded	Chance	N^C_{Em}	Circle	Shaded	Underline
	Utility	N^U_{Em}	Diamond		
	Instance	$\mathbb{O}$	3-D Box		
Causal edge		E_c	Arrow	Single line	$\rightarrow$
Referential edge		E_r	line	Double dotted	$\leftrightsquigarrow$
Information link		E_i	Arrow	dotted line	$\dashrightarrow$
Precedence link		E_p	Arrow	dotted line	$\dashrightarrow$

To recap, an iOOBN *class* consists of nodes, *objects* which are instances of classes, and edges; these components are now formally defined, so now the definition of "Class" can be extended, given partially in Definition 3.14, for two kinds of classes: concrete and abstract.

DEFINITION 3.14 : CONCRETE CLASS

An iOOBN **Concrete Class** $\mathbb{C}$ is a Directed Acyclic Graph (DAG) given by a 4-tuple $< \mathbb{N}, \mathbb{E}, \Pi, \mathbb{O} >$, where

(i) $\mathbb{N}$ = a set of nodes = $N_{In} \cup N_{Em} \cup N_{Out}$, where

- N_{In} = a non-empty set of input nodes,
- N_{Em} = a set of embedded nodes,
- N_{Out} = a non-empty set of output nodes,

(ii) $\mathbb{E}$ = a set of edges = $E_c \cup E_r \cup E_p \cup E_i$, where

- E_c is a set of **causal** edges
- E_r is a set of **referential** undirected edges
- E_i is a set of **information** links
- E_p is a set of **precedence** links

(iii) Π = a set of CPDs, one for each chance node; a set of decision tables one for each decision node; and a set of utility functions, one for each utility node.

(iv) $\mathbb{O}$ = a set of **objects**, representing instances of iOOBN concrete classes that have been specified previously,

Here, each $o \in \mathbb{O}$ is **encapsulated** within $\mathbb{C}$, and $\mathbb{C}$ is called the **encapsulating** class of the object o.

DEFINITION 3.15 : ABSTRACT CLASS

An iOOBN **Abstract Class** $\mathbb{C}'$ of a concrete class $\mathbb{C} = < \mathbb{N}, \mathbb{E}, \Pi, \mathbb{O} >$ is a Directed Acyclic Graph (DAG) given by a 4-tuple $< \mathbb{N}, \mathbb{E}, \pi, \mathbb{O} >$, where $\mathbb{N}$ = a non-empty set of nodes (standard chance, decision or utility) , $\mathbb{E}$ = a set of edges, $\mathbb{O}$ = a set of objects (replicas [a] of other classes) and π = a set of CPDs, decision tables, and utility tables with $\pi \subset \Pi$.

[a] Generally, a "Replica" is a copy of something. In OOBN/iOOBN a replica is an exact copy of a class that represents an instance of that class. In a replica, the graphical structure and the parameters of an OOBN/iOOBN class are preserved. However, as a graphical distinction and ease of use, Hugin shows only the interface nodes of a class to represent an instance of it.

Concrete classes are fully parameterized – with fully specified tables for all three kinds of decision nodes – and thus can be flattened (see Algorithm 3.1) into an ordinary BN, compiled and used to compute posterior probabilities and expected utilities (as for all BNs, as described in

Section 2.3.1). These also correspond to classes as defined in previous formulations of OOBN. In contract, *abstract classes* are not fully parameterized, i.e., do not have fully specified CPTs, decision or utility tables, and therefore cannot be compiled, i.e., cannot be used for reasoning. To our knowledge, the iOOBN is the first OOBN framework to include abstract classes, as distinct from fully specified (concrete) classes.

An abstract class is useful when modellers have partial data or no data available during the modelling phase to specify tables (CPD, decision or utility). The abstract class allows modelling to progress when there is a complete graphical structure and partial or missing tables. As shown in the following Section 3.3.4, it also forms part of the class inheritance hierarchy. Associated with every abstract and concrete class, there is also a "type", which refers to types of the interface nodes of that class.

DEFINITION 3.16 : INSTANCE

An **instance** or an **object** (following [36]) $\mathbb{C}^I$ is a replica or copy of a concrete class $\mathbb{C}$. [a]

The type of an instance is the same as the type of its encapsulating class.

[a] In the OOP an object is created based on the definition of a class. Some of the data in the object are initialized statically during compilation and some of them are set dynamically in run-time through the parameters to a setter method. The same could be done for OOBN objects, i.e., the structure of the objects can be obtained by copying the class structure statically during compilation time. Later, the parameters to the nodes of the object structures can be assigned dynamically in the run time. However, Hugin does not do the instantiation of classes in this way. This is because, the model keeps checking if any cycle has been added during modelling so that the modellers become aware of an erroneous model instantly. Hence, allowing the modellers to model an application based on the interface of a class and adding the detailed structure later at compile time is not possible.

An object is connected to a node in the class in which it sits (called its *encapsulating* class) via a referential edge between the node and an input node in the object's interface. If an input node of an embedded object (of a concrete class) is not connected via any referential edge, it must have a default CPT/decision table/utility table. However, on the other hand, if an input node is connected via a referential edge, the CPT/decision table/utility table of the connected node overrides the default one.

In addition to concrete and abstract classes, the iOOBN also includes an *interface* as a separate data structure, containing only input and output nodes, without any parameters (i.e., these nodes have empty CPTs). This structure is new for iOOBN and is not defined in previous formulations of OOBNs.

DEFINITION 3.17 : INTERFACE

An iOOBN **Interface** $\mathbb{I}$ is a non-empty set of nodes $N = < N_{In}, N_{Out} >$, representing the signature of an iOOBN class $\mathbb{C}$, where

- N_{In} = a non-empty set of random variables representing input nodes of $\mathbb{C}$,
- N_{Out} = a non-empty set of random variables representing output nodes of $\mathbb{C}$.

The type of an interface is the list of type information of the nodes in the interface.

For every abstract or concrete class, there is always exactly one associated interface (though the modeller might not choose to use it explicitly). An interface acts as a placeholder in the encapsulating class during the model design process, allowing the model builder to use submodels built previously, or being built simultaneously by other modellers, without having to understand the inner details of that submodel. Hence, when building a large, complex iOOBN, the modellers need only agree on the mutual interfaces of classes beforehand and combine their works later. The aim of the model building process is to produce a fully specified iOOBN, that is an object (instance) of the top-level class that can be compiled and run. At some stages during that process, all interfaces and abstract classes must be replaced by an object, an instance of a concrete class, having the same interface nodes.

The "type" of an interface is comprised of the types of the nodes in the interface. As an example, the type of the interface "Milk cow" in Figure 3.8 (a) is {Type($Food$), Type($Locale$), Type(Sex), Type($Milk$), Type($Meat$)}, where, according to Definition 3.3, Type($Food$) = < "Chance", {"Good", "Bad"}, "Discrete", "Boolean">. Similarly, the type of other nodes can be found.

An interface can be viewed as a special type of incompletely specified class, where edges, embedded nodes, CPDs/decision/utility tables associated with any node and objects are absent. The interface allows modellers to design a very high-level skeleton model. Thus in the iOOBN, the hierarchy of structures is maintained: interface, the least specified, at the top; then the abstract class, with only the parameters not fully specified; and, finally, the concrete class, fully specified, at the bottom of the hierarchy.

The specification of abstract classes, and interfaces, which provide the OO features of abstraction, encapsulation and generalisation, are key components of the iOOBN that make it different from previous OOBN formulations. Limiting the connections between classes to the interface nodes provides encapsulation, a way to deny unauthorized access to the information and structure of the encapsulated submodels. However, the information hiding within an iOOBN is not "pure". Evidence can be entered for any node in the iOOBN, no matter how

deeply it is embedded, and the posteriors (for chance nodes), and expected utilities for decisions are computed by the inference algorithm and are available as the iOOBN model "outputs". [3]

3.2.1 iOOBN equivalence to ordinary BNs

Next, the flattening of an iOOBN class is considered. This flattening must be carried out in the iOOBN and motivated from the OOBN to BN flattening (as discussed in Section 2.7.4 of Chapter 2) so that it is able to use any of the existing compilation and inference algorithms.

Flattening (Algorithm 3.1) of an iOOBN class $\mathbb{C}$ is the process of forming an ordinary BN from the components of $\mathbb{C}$. The formation includes the following operations:

- recursively replacing all object nodes by the components of the corresponding class
- collapsing each node pair connected via a referential link into a single node

The edges within an iOOBN must be such that it flattens out [4] to a valid BN, that is, a directed acyclic graph. Figure 3.3 shows a flattened version of the iOOBN class "Profit" (see Figure 3.1).

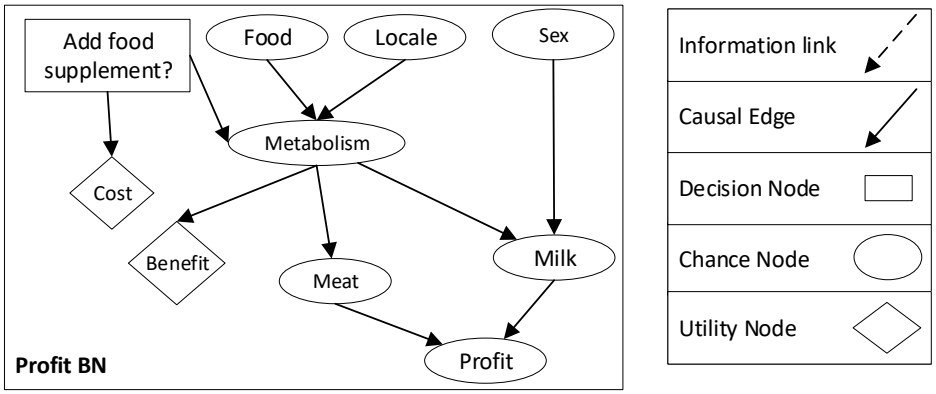


Figure 3.3: iOOBN class "Profit" flattened to a BN "Profit"

[3] The process of entering evidence and obtaining the posterior of embedded nodes are analogous to I/O in OO programming, which can occur in any object in the overall software.

[4] This is the process whereby Hugin converts the OOBN into the underlying BN, as per its API function "Creating Runtime Domain" [27].

ALGORITHM 3.1 (FLATTENING AN IOOBN CLASS)

Call : $Flattening\_iOOBN(C) \rightarrow BN$

Input: $\mathbb{C} =< \mathbb{N}, \mathbb{E}, \Pi, \mathbb{O} >$: ($\mathbb{N} = N_{In} \uplus N_{Em} \uplus N_{Out}$, $\mathbb{E} = E_c \uplus E_r \uplus E_p \uplus E_i$)

Output: $BN =< N, E, \pi >$: An ordinary BN

```
1  begin
2      N ← ℕ
3      E ← E_c ⊎ E_p ⊎ E_i
4      π ← Π
5      foreach instance o ∈ 𝕆 do
6          c ← o.class()
7          // Assuming that instance "o" has an attribute "class" holding the associated
             class name
8          bn ← Flattening(c) // The recursive call to the flattening process
9          foreach Referential edge e_r =< n_i, n_j > ∈ E_r do
10             foreach edge e =< n_j, n_k > ∈ bn.E do
11                 bn.E ← bn.E ⋃ e' =< n_i, n_k >
12                 bn.E ← bn.E ∖ e
13             bn.N ← bn.N ∖ n_j
14             N ← N ⋃ bn.N
15         E ← E ⋃ bn.E
16         π ← π ⋃ bn.Π
17     return BN
```

3.3 iOOBN and Inheritance

3.3.1 Motivation

The BN modelling process, in practice, is usually iterative and incremental [1, 32], particularly when building a model partially or entirely by hand, rather than learning from data. So the modelling process can be seen as a sequence of changes to the BN: adding or deleting nodes or edges, changing the name or states of a node, changing some or all of a CPT, and so on. Also, some of these changes necessarily generate others: for example, adding a new state to a node means changing the CPT of both the node itself and also the CPTs of any child nodes.

When building an OOBN, as well as the ordinary BN modelling changes, there are other possibilities for modelling steps, such as embedding an object within a class, creating a new class, changing the referential edges to an embedded object, and so on. When modelling with ordinary OOBNs, as implemented in Hugin, if the modeller wants to create a new class that

has common elements with an existing class – for example, if s/he had created a "Cow" class, and wanted to make a new "Milk Cow" class, this would involve copying the "Cow" class, giving it a new name, then making changes in the new "Milk Cow" class. When modelling complex problems, the number of classes can grow quite large (as in the case study in Section 3.5, making it difficult for the modeller to keep track of the differences and similarities of the various new classes and increasing the chance of inadvertent errors and inconsistencies. In addition, if at any point the modeller wants to change some part of the model that is common to more than one class, the modeller often needs to make the same change in more than one class. For example, in ordinary OOBN modelling, changing the state-space of the Locale node in the "Cow" class node would suggest the same change should be made in the "Milk Cow" class (and any others) [5].

Inheritance provides an approach for solving these issues. By using inheritance, the connection can be recorded between the variations of a class, and this allows a change in one class to flow through to other relevant classes. Thus, using the iOOBN framework, the modelling steps are: create a new subclass (e.g. "Cow"), give it a new name (e.g. "Milk Cow"), then make a sequence of changes to the new subclass itself. The iOOBN system must then first check that any given change is valid, and then keep track of those changes. Thus the subclass is internally represented by what it inherits from its parent, i.e., where it is identical, and then by the changes that override elements of the parent class.

3.3.2 Sub-interface, subclass, inheritance hierarchy and polymorphism

Inspired by the OO principle of inheritance, in iOOBN all classes and interfaces can inherit structure and parameters (analogous to inheriting attributes and/or behaviours in OO programming) from another entity. The class or interface that inherits is then known denoted as "sub-types". A sub-type entity is a variation, usually thought of as a specialisation, that includes changes to structure and/or parameters.

In OO programming there are two types of inheritance, namely "subtyping" or "interface inheritance" and "implementation inheritance". In subtyping, the interface of the superstructure (class or interface) is maintained in the substructures, but the underlying implementation may vary. In implementation inheritance, the implementation of the superstructure, along with its interface, is copied and extended in the substructure. Both of these types facilitates maximal resource reuse and enhance simultaneous workability. This advantage motivated the introduction of subclasses and sub-interfaces that give the modellers "subtyping" capability

[5] Of course, ordinary OOBNs still provide efficiency in that a change to a class is pushed out to any object created from that class. In contrast, when using subnetworks in the GeNIe software [28], the modeller can make copies of the same network fragment, but to make a change in that fragment, s/he has to change it in all the subnetwork copies, a tedious process.

while extended class and extended interface have now been defined to allow"implementation inheritance".

DEFINITION 3.18 : SUB-INTERFACE

In iOOBN, let $\mathbb{I}' =< N'_{In}, N'_{Out} >$ is a **Sub-interface** of an interface $\mathbb{I} = < N_{In}, N_{Out} >$ if $N_{In} \subseteq N'_{In}$ and $N_{Out} \subseteq N'_{Out}$. In other words, $\mathbb{I}$ is a superinterface of $\mathbb{I}'$.

DEFINITION 3.19 : SUBCLASS

In iOOBN, $\mathbb{C}' =< \mathbb{N}', \mathbb{E}', \Pi', \mathbb{O}' >$ (where $\mathbb{N}' = N'_{In} \cup N'_{Em} \cup N'_{Out}$) is a **Subclass** of class $\mathbb{C} = < \mathbb{N}, \mathbb{E}, \Pi, \mathbb{O} >$ (where $\mathbb{N} = N_{In} \cup N_{Em} \cup N_{Out}$) if $N_{In} \subseteq N'_{In}$ and $N_{Out} \subseteq N'_{Out}$. In other words, $\mathbb{C}$ is a **Super**-class of $\mathbb{C}'$ [a].

[a] There are certain cases, particularly when making a concrete class from an abstract class before instantiating an object of a class, where a subclass needs to be defined with reference to probability tables. There is no general mapping between Π and Π', i.e., the CPDs of the superclass and the subclass, respectively, because as part of the hierarchy, all the CPTs (in an extreme case) could be replaced. However, in practice, many will be the same, or when arcs are deleted, there may be marginalisation.

A *sub-interface* is defined as one which inherits all the input and output nodes from its superinterface, and one that may have any number of additional input or output nodes. Similarly, a *subclass* inherits all the input and output nodes from its *superclass*, and may have any number of additional input or output nodes. The subclass inherits all of the internal structure of the superclass (embedded nodes and objects), and may also contain additional nodes and edges. The interface nodes of the subclass then become a superset of the superclass. Because all input nodes in concrete classes must have default parameters specified, an object of a superclass may be replaced with a subclass. The inheritance relationship is denoted with a solid line with an open arrow directed into the superclass, following standard UML inheritance notation.

DEFINITION 3.20 : EXTENDING A CLASS

In iOOBN, a class $\mathbb{C}' =< \mathbb{N}', \mathbb{E}', \Pi', \mathbb{O}' >$ (where $\mathbb{N}' = N'_{In} \cup N'_{Em} \cup N'_{Out}$) is an **Extended Class** of

- an interface $\mathbb{I} = < N_{In}, N_{Out} >$ if $N_{In} \subseteq N'_{In}$ and $N_{Out} \subseteq N'_{Out}$, or
- a class $\mathbb{C} = < \mathbb{N}, \mathbb{E}, \Pi, \mathbb{O} >$ if $N_{In} \subseteq N'_{In}$, $N_{Out} \subseteq N'_{Out}$, $N_{Em} \subseteq N'_{Em}$, $\mathbb{O} \subseteq \mathbb{O}'$ and $\Pi \subseteq \Pi'$

In other words, if $\mathbb{C}'$ is constructed by adding necessary additional structures and parameters to a copy of $\mathbb{I}$ or $\mathbb{C}$, then $\mathbb{C}'$ is called an extended class of $\mathbb{I}$ or $\mathbb{C}$.

Next, an important structure **Inheritance hierarchy**, is introduced, which is constructed through the relationships between subclasses and the superclass, concrete classes and abstract classes, and interfaces and classes.

DEFINITION 3.21 : INHERITANCE RELATIONSHIP AND HIERARCHY

An **Inheritance hierarchy** in iOOBN is a tree $T =< V, E >$, where each vertex $v \in V$ is an iOOBN class or interface and each edge $e = v_i \rightarrow v_j \in E$ represents a relationship called **inheritance relationship** between the corresponding classes or interfaces, with the following restrictions:

- If v_j is an interface, then v_i can only be an interface, i.e., no interface can be extended from an abstract or a concrete class.
- If v_j is an abstract class, then v_i can only be an interface or an abstract class, i.e., no abstract class can be extended from a concrete class.
- If v_j is a concrete class, then v_i can be an interface, an abstract class or a concrete class.

The edge/relationship also denotes that v_j is extended from v_i.

A crucial aspect of OO inheritance, which must apply here in iOOBN as well, is that the classes remain backward compatible; this means that an encapsulated object can be replaced by an object of any sub-type (direct, or via the hierarchy) and the resultant iOOBN is still a valid iOOBN – nothing "breaks", and when made fully concrete, it can be compiled and run (i.e., used for reasoning). The inclusion of backward compatibility gives the iOOBN another prominent feature of the OO paradigm, namely "polymorphism", which allows an object to take many forms if and when required. Unlike OO programming, this replacement of an object by another that is a sub-type cannot be done at runtime; instead, it must be done during the modelling process. This particular case can be simply be modelled as a set of classes that differ only in their embedded objects, as a hierarchy of classes.

DEFINITION 3.22 : POLYMORPHISM

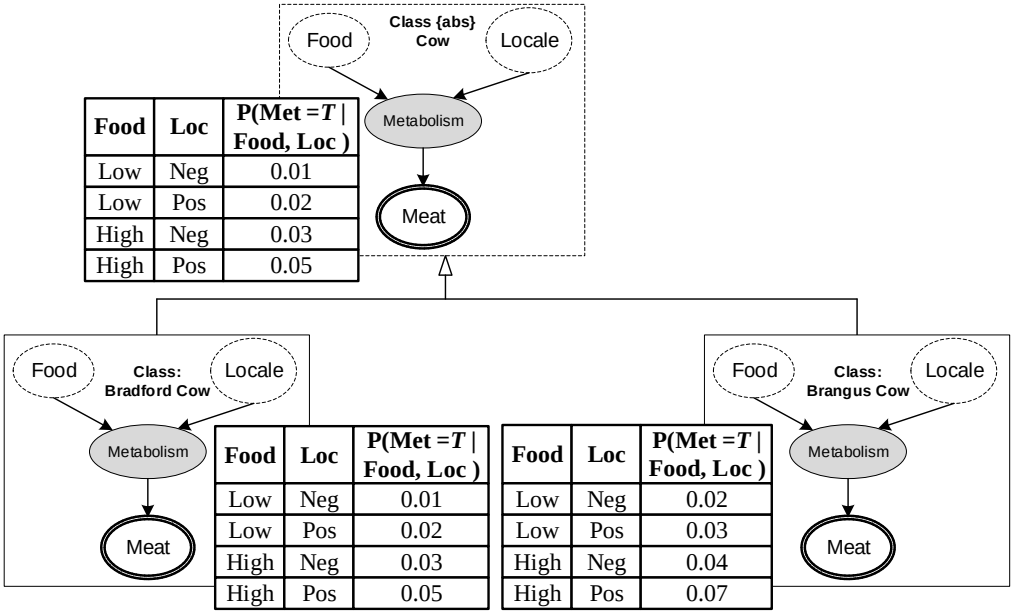
Let $T =< V, E >$ be an inheritance hierarchy and a pair $< \mathbb{A}, \mathbb{D} >$ of iOOBN interface or class where $\mathbb{A}, \mathbb{D} \in V$. $\mathbb{D}$ is known to be **Polymorphic** of $\mathbb{A}$ if $\mathbb{A}$ is an ancestor of $\mathbb{D}$ in T.

Polymorphism in an iOOBN (as in any OO paradigm) is a compelling feature. It allows modellers to model a system with a simpler structure (a partially defined class, a superclass or an interface) or any structure that is available at the time of modelling and later to extend the model by replacing with a relatively more complex or more suitable structure (the most appropriate concrete class).

Further analogies can be made with OO programming where an iOOBN subclass inherits certain elements from its superclass, such that any changes to those elements, (e.g., changes

in the domain or types of node, constitute a form of **overriding**. An iOOBN is a strongly typed framework where each entity has an associated type, which allows type checking to be performed to ensure that the types of interface nodes are preserved in the subclass. Moreover, when an object of any subclass can replace a superclass object, then this is referred to as a form of **typecasting**.

3.3.3 Changes allowed in iOOBN inheritance



Food	Loc	P(Met =*T* | Food, Loc)
Low	Neg	0.01
Low	Pos	0.02
High	Neg	0.03
High	Pos	0.05

Food	Loc	P(Met =*T* | Food, Loc)
Low	Neg	0.01
Low	Pos	0.02
High	Neg	0.03
High	Pos	0.05

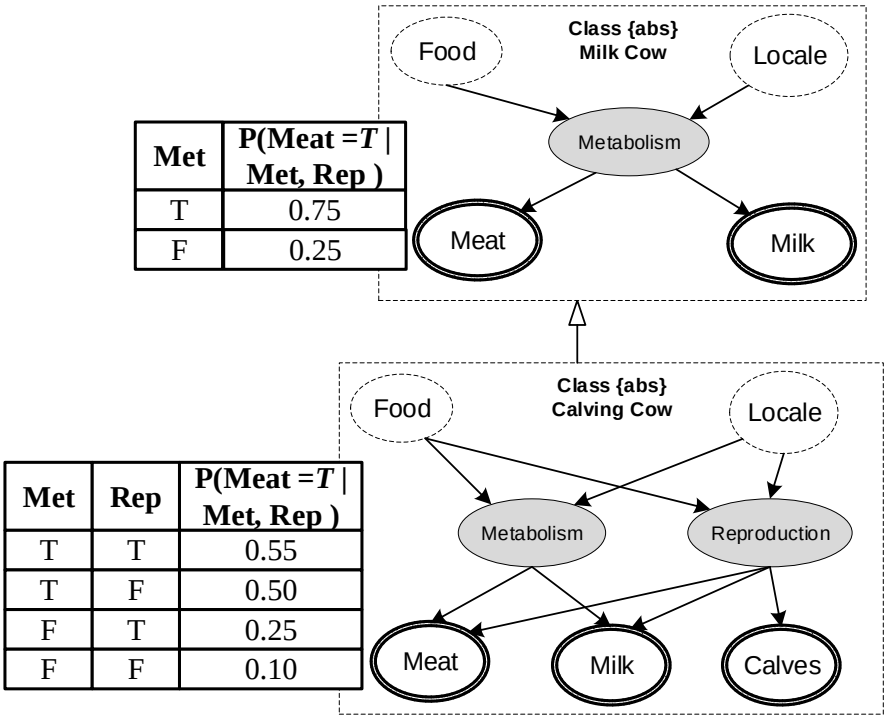
Food	Loc	P(Met =*T* | Food, Loc)
Low	Neg	0.02
Low	Pos	0.03
High	Neg	0.04
High	Pos	0.07

Figure 3.4: OMD example: Changing CPTs of inherited nodes in the subclasses

The simplest form of subclass is one where only the parameters of the subclass are changed, and all the structural elements remain the same. Figure 3.4 gives an example of this, with two subclasses of the "Cow" class created to represent two different species of cows, where the CPT of the Metabolism node has been changed from the parent "Cow" class, to represent the different metabolic rates of two different breeds of cows (**Bradford** and **Brangus**). Note that the figure shows the resultant new CPT, not the internal representation of the inherited CPT plus changed elements.

Any node – input, output or embedded – can be added to a subclass. Node adding would typically involve the addition of edges; otherwise, the change results in the degenerate case having a disconnected node. Embedded nodes can be deleted, but the definition of inheritance – where the interface of a class and subclass must be identical – forbids the deletion of input or output nodes. However, if an input node is not relevant in the subclass, it is possible to delete the edges from an input node to its children, disconnecting it from the rest of the structure.

Figure 3.5 illustrates an example where the subclass has involved the creation of new nodes



Met	P(Meat =*T* | Met, Rep)
T	0.75
F	0.25

Met	Rep	P(Meat =*T* | Met, Rep)
T	T	0.55
T	F	0.50
F	T	0.25
F	F	0.10

Figure 3.5: OMD example: Adding edges and nodes within subclasses

as well as edges, in order to model the reproduction process in the OMD example.

Another modelling step the iOOBN system must handle is a change in the states of a node. The first example for this is a change to the states for an embedded node in a class; this is straightforward and handled in exactly the same way as a change in the CPT of a node.

A second and more interesting case is when the desired change is either to the states of an input node or the states of an output node of a class, $\mathbb{C}$. By definition, this is not allowed, as the interface of a class and its subclass must be the same. However, the desired change can be made by modifying the non-interface structure of the class, adding a duplicate node with the different type and connecting it to the interface node, leaving the interface unchanged. Thus, the algorithm for the modelling steps to, in effect, change the type of an input node **I**, without changing the actual interface, is shown in Algorithm 3.2, while the steps to change the type of an output node, **O**, in class $\mathbb{C}$ are shown in Algorithm 3.3.

Figure 3.6 depicts an example of this state change, when the states {*pos*, *Neg*} for the node *Locale*, are changed to {*Fav*, *Neut*, *Harsh*}, and the states {*Good*, *Bad*} for the node **Meat** are changed to {*High*, *Avg*, *Low*}.

Figure 3.7 gives a schematic representation of the changes effected by Algorithms 1 and 2. Since in the input node case, its children can be either embedded or output nodes, the notations are combined (shading plus bolded outline) for these child nodes, while for the output node case, its parents can be either embedded or input nodes, and the shaded and dotted outline

ALGORITHM 3.2 (MODELLING STEPS WHEN CREATING A NEW SUBCLASS BY ADDING STATES TO AN INPUT NODE)

Call: $AddingStateToInput(\mathbb{C}, I) \rightarrow \mathbb{C}'$

Input: Class $\mathbb{C}$

Input Node I

Output: Subclass $\mathbb{C}'$

1 **Create** the subclass $\mathbb{C}'$ /* Replica of C */
2 **Add** a new node I' that has the new set of states
3 **Add** an edge from I to I'
4 **foreach** *edge* e *from* I *to its children* $(child)$ *in* $\mathbb{C}$ **do**
5 **Remove** e
6 **Add** a new edge from I' to $child$
7 **Update** CPT of $child$
8 **Construct** a CPT for I'
9 **return** *subclass* $\mathbb{C}'$

ALGORITHM 3.3 (MODELLING STEPS WHEN ADDING STATES TO AN OUTPUT NODE)

Call: $AddingStateToOutput(\mathbb{C}, O) \rightarrow \mathbb{C}'$

Input: Class $\mathbb{C}$

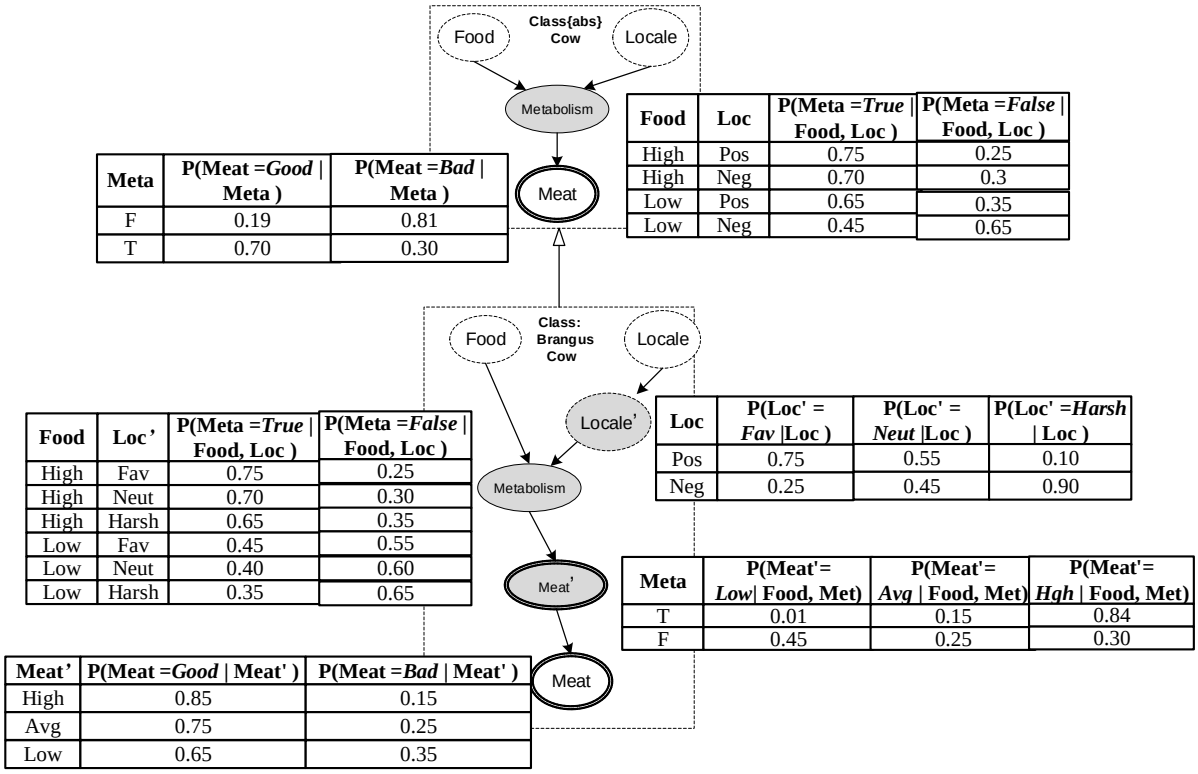
Output Node O

Output: Subclass $\mathbb{C}'$

1 **Create** the subclass $\mathbb{C}'$ /* Replica of C */
2 **Add** a new node O' that has the new set of states
3 **Add** an edge from O' to O
4 **foreach** *edge* e *to* O *from its parents* $(parent)$ *in* $\mathbb{C}$ **do**
5 **Remove** e
6 **Add** a new edge from $parent$ to O'
7 **Create** the CPT for O'
8 **Construct** a CPT for O
9 **return** *subclass* $\mathbb{C}'$;

notations are combined for these parent nodes.

As these examples show, there are many possible changes that can be made when creating a new subclass, and it is both domain-dependent and at the discretion of the modeller, as to



Meta	P(Meat =*Good* | Meta)	P(Meat =*Bad* | Meta)
F	0.19	0.81
T	0.70	0.30

Food	Loc	P(Meta =*True* | Food, Loc)	P(Meta =*False* | Food, Loc)
High	Pos	0.75	0.25
High	Neg	0.70	0.3
Low	Pos	0.65	0.35
Low	Neg	0.45	0.65

Food	Loc'	P(Meta =*True* | Food, Loc)	P(Meta =*False* | Food, Loc)
High	Fav	0.75	0.25
High	Neut	0.70	0.30
High	Harsh	0.65	0.35
Low	Fav	0.45	0.55
Low	Neut	0.40	0.60
Low	Harsh	0.35	0.65

Loc	P(Loc' = *Fav* |Loc)	P(Loc' = *Neut* |Loc)	P(Loc' =*Harsh* | Loc)
Pos	0.75	0.55	0.10
Neg	0.25	0.45	0.90

Meta	P(Meat'= *Low*| Food, Met)	P(Meat'= *Avg* | Food, Met)	P(Meat'= *Hgh* | Food, Met)
T	0.01	0.15	0.84
F	0.45	0.25	0.30

Meat'	P(Meat =*Good* | Meat')	P(Meat =*Bad* | Meat')
High	0.85	0.15
Avg	0.75	0.25
Low	0.65	0.35

Figure 3.6: Typecasting: Type Changing of inherited Input-Output nodes in subclasses

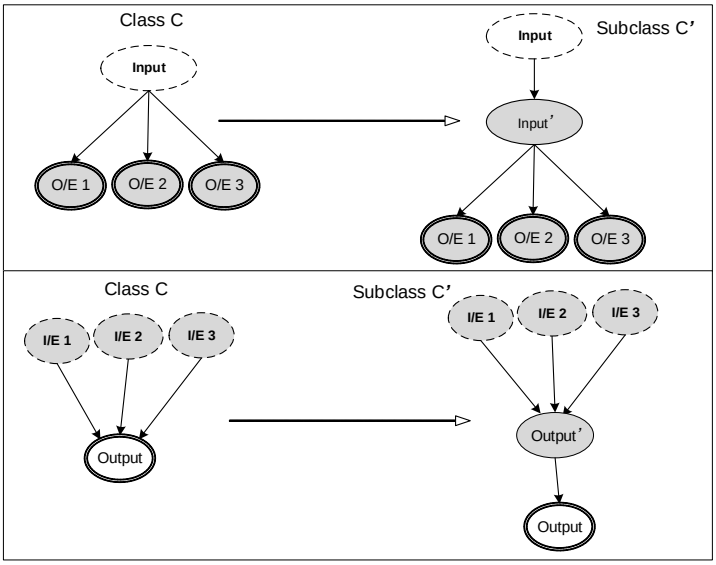


Figure 3.7: Schematic diagram showing the model change required to effect a change in the states of an input (above) or output (below) nodes.

how many changes to make in a single inheritance step. There can be decisions about whether to create multiple levels of subclasses or to create a single level with multiple subclasses. Note that, in some cases, an iOOBN class hierarchy created incrementally (as the algorithm proposed in Chapter 5) might well be suboptimal and that regular reengineering might be beneficial; however, that is undoubtedly an area for future investigation, after further experience in

modelling within the iOOBN framework.

Table 3.2 shows a summary of the allowed operations. It shows affected components due to the operations, and refers to an appropriate example.

3.3.4 Inheritance in the livestock farming example

Figure 3.8 (part (a) and part (b)) shows a UML-type representation of the OMD hierarchy in the iOOBN. Here, the fundamental components are described in this iOOBN example; the following section describes the class and interface hierarchy elements of the example.

Table 3.2: iOOBN inheritance types and affected components of the iOOBN class.

***Inheritance Type* (*Changes allowed in a subclass*)**	**Affected Components**									Example Figure No.
	Interface nodes			Embedded nodes			Objects	Edges	CPTs	
	Name	Type	States	Name	Type	States				
Replacing an abstract class with a concrete class	-	-	-	-	-	-	-	-	✓	3.8 (part-c)
Replacing an interface with a class	-	-	-	✓	✓	✓	✓	✓	✓	3.8 (part-c)
Update CPTs values	-	-	-	-	-	-	-	-	✓	3.4
Add edges	-	-	-	-	-	-	-	✓	✓	3.5
*Remove edges**	-	-	-	-	-	-	-	✓	✓	
Add interface nodes	✓	✓	✓	✓	✓	-	-	✓	✓	3.8(part-b)
Remove interface nodes	Not allowed									
Modify interface nodes	✓	-	-	✓	✓	✓	-	✓	✓	3.6
Add-alter embedded nodes	-	-	-	✓	✓	✓	-	✓	✓	3.5
*Remove embedded nodes**	-	-	-	✓	✓	✓	-	✓	✓	
Add-alter-replace objects	-	-	-	-	-	-	✓	✓	✓	3.8(part-b)
*Remove objects**	-	-	-	-	-	-	✓	✓	✓	
Replacing a superclass instance with a subclass instance	-	-	-	-	-	-	✓	✓	✓	3.8(part-c)

* : Removing components is subject to the policy of the modellers.
✓ : The component in the column is affected by the action in the row.
\- : The component is not affected by the action.

3.3.4.1 Using encapsulation and abstraction

Consider the concrete "Calving Cow" class in Figure 3.8 (part-b, bottom): it has five input nodes ($Food$, $Locale$, Sex, $Health$ and $Breed$), two embedded nodes (Metabolism and Reproduction), an embedded object *Productivity*, which is an instance of the *Productivity* class, and three output nodes (**Meat**, **Milk** and **Calves**). There are referential edges (undirected double dotted lines) from three of those input nodes ($Health$, $Locale$, and $Breed$) to the input nodes of the Productivity object, and a single causal edge (solid line) from the embedded object (the output **Productivity** node, not shown) to the embedded Reproduction node. Because it is a concrete class, all the CPTs must be fully specified (although this is not shown in the figure for reasons of space). The abstract Calving Cow class has the same input and output nodes as the concrete Calving Cow class; it has a different internal structure – this is explained

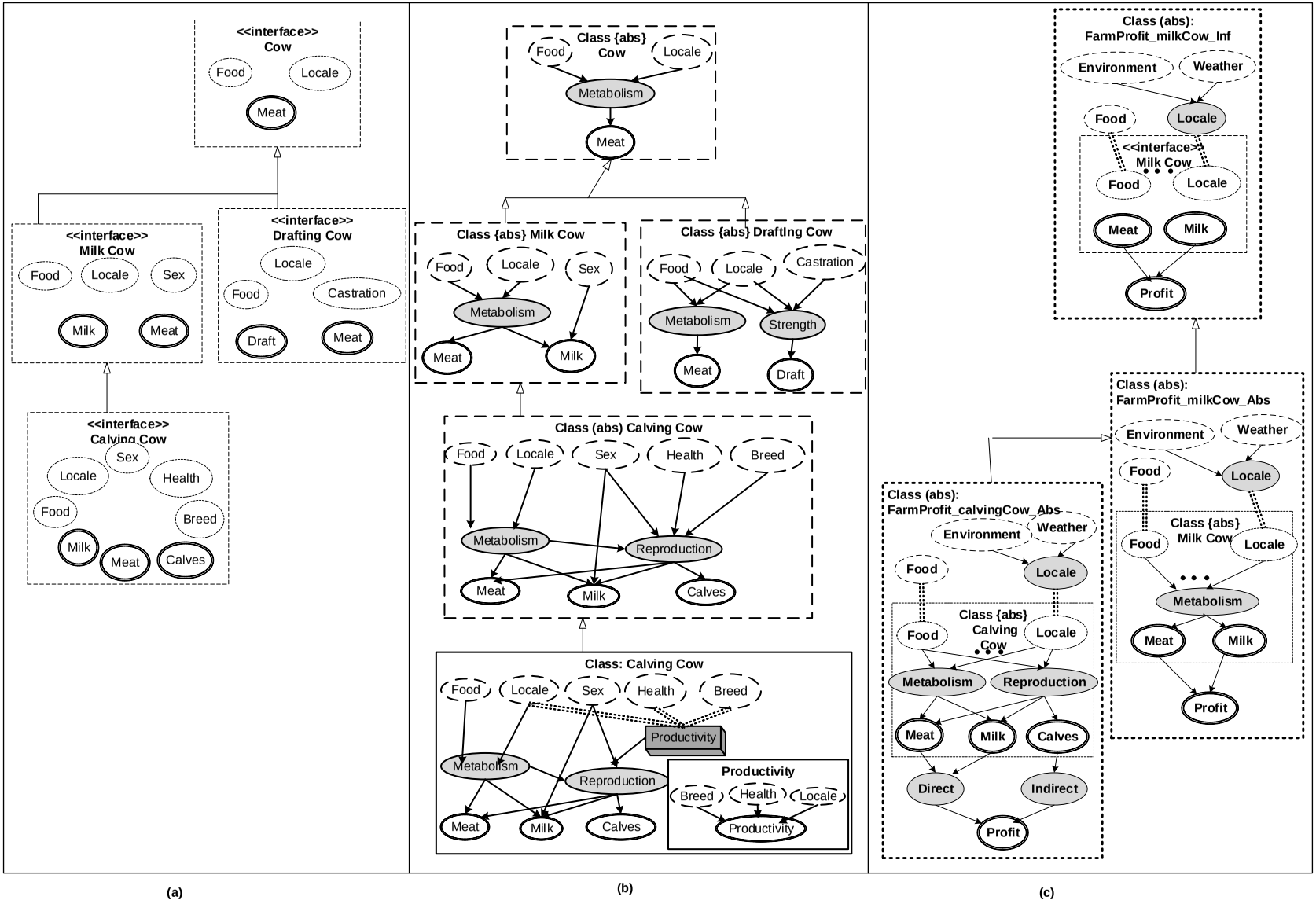


Figure 3.8: Example of (a) interface hierarchy and (b) class hierarchy for the OMD livestock example used to illustrate the components of the iOOBN framework. (c) How different objects can be interchangeably embedded within a class: an interface and two different abstract classes, one (Calving Cow) a subclass of the other (Milk Cow); this demonstrates the OO features of polymorphism.

below. There is also an interface Calving Cow class (part-a), connected via a dotted line with open arrowhead going from the abstract class to its corresponding interface class (following standard OO UML notation).

Consider the OMD example in Figure 3.8 again. The highest level class is the abstract Cow class (part-b, top) because all cows in this farm example can be used for meat production. The Cow class, in turn, has two (abstract) subclasses (Milk Cow and Drafting Cow). Both these subclasses inherit the Cow superclasses input nodes (*Food* and *Locale*) and the output node (**Meat**) but have additional interface nodes (*Sex* and **Milk** for Milk Cow class, *Castration* and **Draft** for Drafting Cow). Drafting Cow has an additional interface node, *Strength*, an essential attribute for a drafting cow, as well as an additional output node (**Draft**). Similarly, the previously described Calving Cow abstract class is a subclass of Milk Cow and has a more complex structure: two new input nodes (*Health* and *Breed*) and the additional output node (**Calves**) and different internal structure around the new embedded node *Reproduction*. The concrete Calving Cow is itself a subclass of the abstract Calving Cow class, with the same interface nodes, but with a different internal and more complex structure, specifically around an additional embedded object *Productivity*. Similarly, there is an inheritance hierarchy between the interfaces Cow, Milk Cow and Calving Cow, with the same solid line with an open arrow notation; as to be expected, these correspond to the different interfaces found in the class hierarchy.

3.3.4.2 Using inheritance

As an example depicted in Figure 3.8 (part-c), where the **FarmProfit_milkCow_Inf** superclass contains an embedded interface object ("Milk Cow" interface), its subclass, **FarmProfit_milkCow_Abs**, contains an embedded object of the corresponding "Milk Cow" abstract class, and in turn the **FarmProfit_calvingCow_Abs** subclass (derived from the class, **FarmProfit_milkCow_Abs**), contains an embedded object of the abstract subclass "Calving Cow" [6].

3.3.4.3 Using polymorphism and typecasting

The iOOBN is a strongly typed framework. It allows changing the type as a special feature with a systematic approach. The type changing mechanism follows the definition of "Type" for BNs and OOBNs.

In checking compatibility for polymorphism (dynamic changing of objects) of class C with another class C', the "Type" of both classes is compared. If the "ParentClass", "ParentInterface"

[6]Note that adding an additional element to iOOBN has been considered, namely to have "variants" of a class which differ only in the embedded object and all the rest of the class remains the same (including the referential edges to the encapsulated object). This feature would be utilised in applications where a script is used to generate different versions of the model for different purposes (as in the WGR case study described in Section 3.5), allowing a form of run time binding. However, this remains a possible future extension to the iOOBN.

and "InterfaceNodes" of $\mathbb{C}$ are subsets of the "ParentClass", "ParentInterface" and "InterfaceNodes" of $\mathbb{C}'$, respectively.

Figure 3.8, part-c illustrates replacing an interface with its implementing abstract class instance, and later this instance is replaced by another instance of the abstract subclass extended from this abstract class. This replacing process is an example of polymorphism.

3.4 Applying iOOBN to Previous OOBN Problems

This section revisits some well-known real-life examples of BN/OOBN modelling such as the car accident [36], computer diagnosis problem [203], farmland modelling [37], and power surge problem [76]. The aforementioned problems have been re-visited (reengineered or reengineered and extended) and, using the proposed iOOBN system, some ways have been demonstrated of extending the systems in order to enable them to cope-up with changes. The limitations of the existing frameworks are compared to the capability of the proposed iOOBN and the limitations are considered in modelling the extended/reengineered systems. Furthermore, once the reengineering is accomplished, a "validation" (see Section 3.5.4) must be conducted to ensure the correctness of the reengineered model and its viability.

3.4.1 The Asia BN

As an example, the classic "**Asia BN**" case is revisited in order to test for the feasibility of the iOOBN framework. A modeller, using an ordinary BN such as the Asia BN (shown in Figure 3.9 (a)), needs to know the complete network configuration in order to work with the BN (For example, using it somewhere else, extending or modifying it). Moreover, let us say the modeller wants to extend the model by adding some extra nodes (e.g., decision and utility) in order to make it a decision-making system, as shown in Figure 3.9 (c) (i.e., into an **AsiaExt** BN). The **AsiaExt** BN can now help a user to decide whether performing an Xray is beneficial or not with respect to the provided evidence. This additional facility is added by using the decision node *Do Xray?* and utility nodes *Cost of Xray* and *Benefit of Xray*. In order to build such a BN from the Asia BN in part (a), the modeller needs to know full details of the Asia BN.

However, if the modeller uses OOBN and makes a class, **Asia**, (as shown in Figure 3.9 (b)) in the Asia BN, where A and S are marked as input nodes and T, L and B as output nodes, then the modeller does not need to know the details of the network other than the interface nodes (A, S, T, L, and B).

In order to use the BN, encapsulating it in a class, instantiating (i.e., making an instance of) the class, i.e., **Asia**, and adding the required additional nodes and edges is sufficient to extend

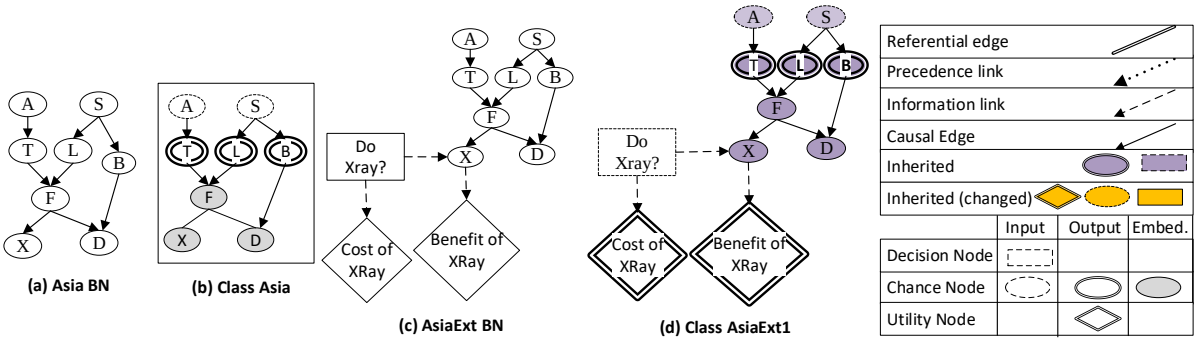


Figure 3.9: Example: encapsulating a BN in a class

the model. Another approach that is well suited in this situation is to extend the class **Asia**, using the inheritance feature of iOOBN and form the class, **AsiaExt1**, (as shown in Figure 3.9 (d)).

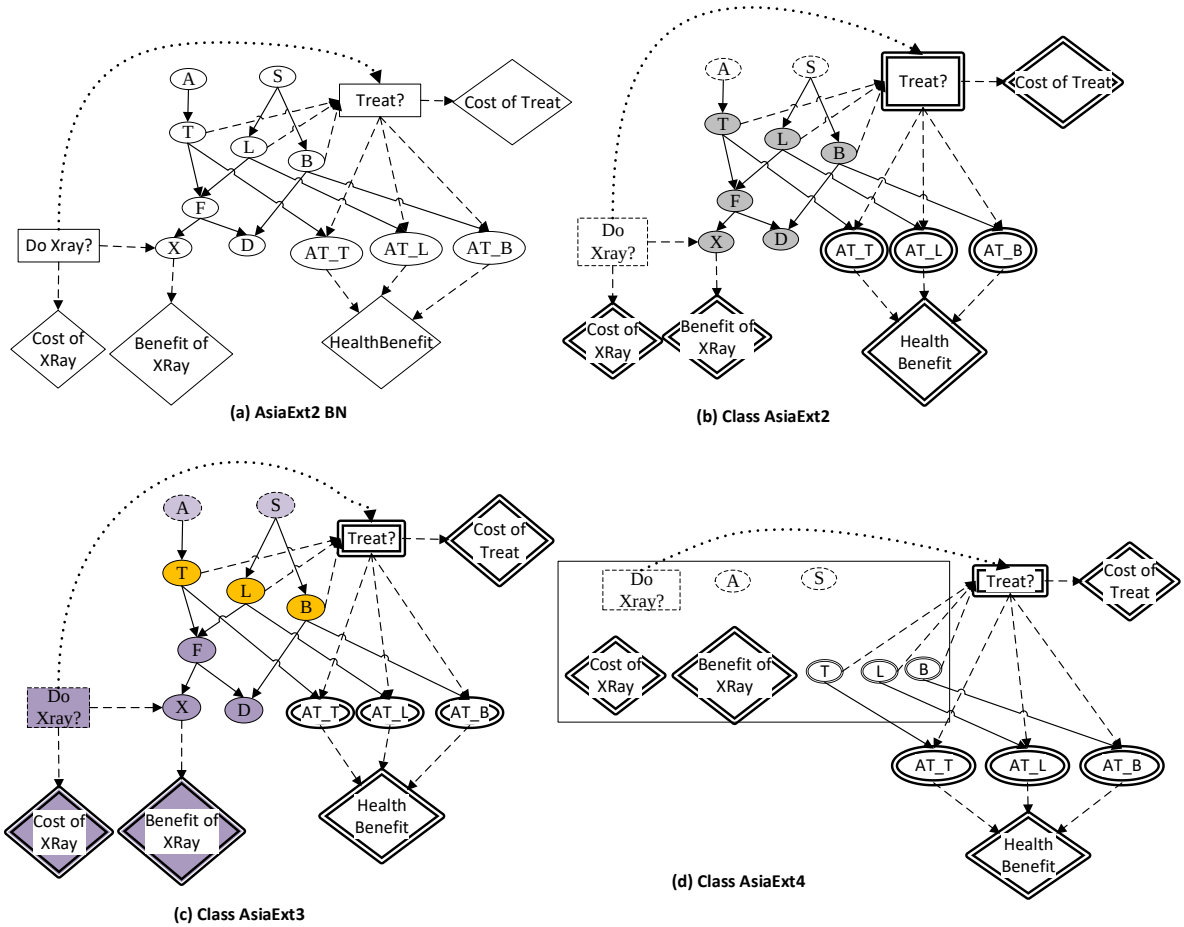


Figure 3.10: Example: forming a class by reusing by means of inheritance and instantiation

Let's say the modeller is interested in making a decision making tool in order to help users know if taking treatment for a particular disease would be beneficial or not in a given situation

where associated cost and profit are provided. In the case of an ordinary BN, the**AisaExt2** BN (as shown in Figure 3.10 (a)) serves the purpose [7]. The **AsiaExt2** BN is obtained by adding three extra chance nodes: *AT_T* (after treatment the chance of TB), *AT_C* (after treatment the chance of cancer) and *AT_B* (after treatment the chance of bronchitis); one decision node: *Treat?* (should take treatment?); and two utility nodes: *Cost of treatment* and *Health Benefit* (the benefit of taking the treatment). In order to utilize the reusability feature of iOOBN, there are several ways of accomplishing the design and obtaining the class *AsiaExt2*, as shown in Figure 3.10 (b).

One of the approaches can be to use inheritance and extend the class **AsiaExt1**, of Figure 3.9 (d). The extension of the class is shown in Figure 3.10 (c) with the inherited segments marked in colours. Another approach is to use the alternative options provided by the polymorphism and abstraction features of the iOOBN framework. Since the modellers only need to know the interface nodes of a class in order to use it, the aforementioned iOOBN features allow modellers to work more flexibly. They can design a system in the abstract form beforehand (as shown in Figure 3.10 (d)). This facility is especially helpful if they have no predefined submodel or class available to fit in, or have no data at all, or have incomplete data available during the construction of the model. This technique can also be referred to as building a prototype model where detailed class implementations are postponed by adding a placeholder/interface for the class that might be completed later or added by another group of people. In order to complete the modelling, the placeholder/interface can be replaced by instance of a detailed implementation class, or it can be replaced either by an instance of class, as an example AsiaExt1 (Figure 3.9 (d)), or an instance of a class that has the same (or superset of) interface nodes.

3.4.2 The car accident

Figures 3.11a and 3.12a present the car accident model proposed in [36] as an OOBN model. The model consists of a class containing instances of three different subclasses (1. $Driver$, 2. Car, 3. $Road$) and some embedded standard nodes. Among those instances, the class for the instance Car is shown in detail. It consists of standard embedded nodes and some objects from 1. *Owner*; 2. *Engine*; 3. *Steering*; 4. *Tyre*; 5. *Brake* classes.

The original car accident model was designed in OOBN. As iOOBN is a backward-compatible framework, reproducing the car-model in it is quite straight forward. In figures 3.11b and 3.12b, the reengineered model is shown. The iOOBN offers more facilities, such as extending the already built model without changing the existing system, and allows maximal reuse of the existing components.

[7] The **AsiaExt** BN and **AsiaExt2** BN, and their equivalent OOBN classes shown in Figure 3.9 and 3.10 are for illustration purpose only and were not developed in consultation with medical experts.

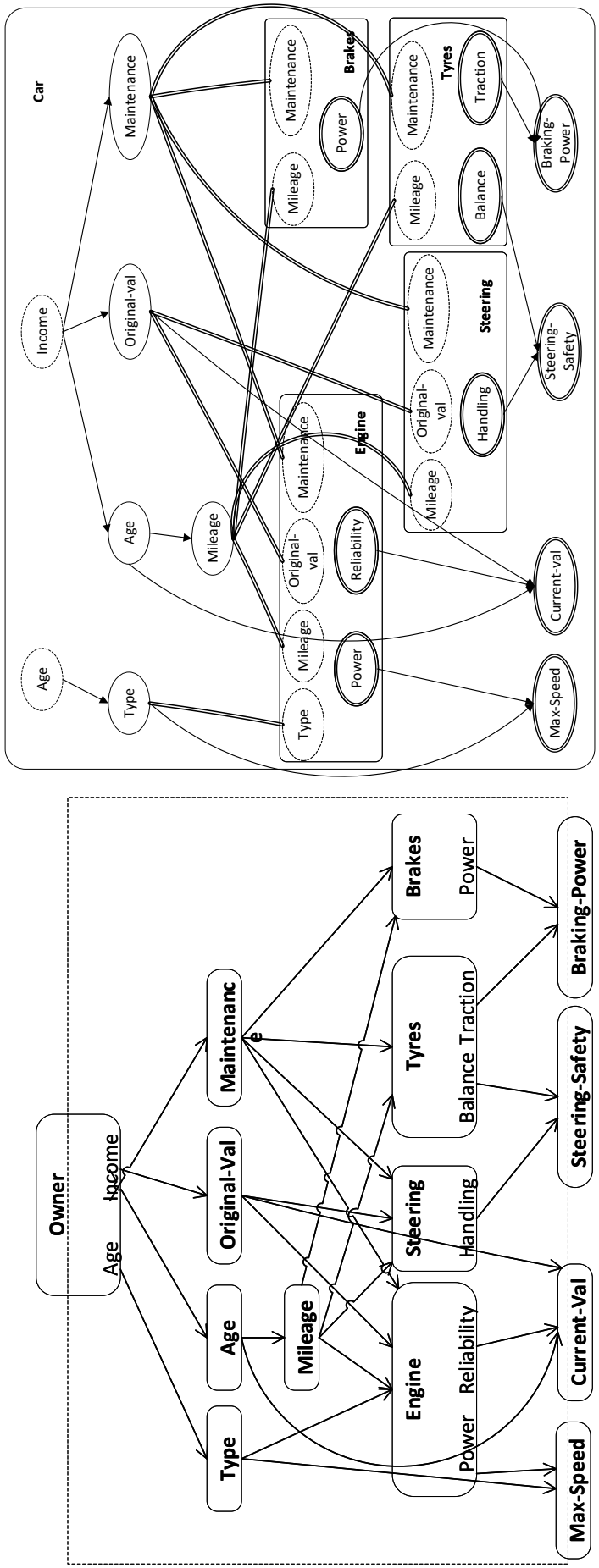
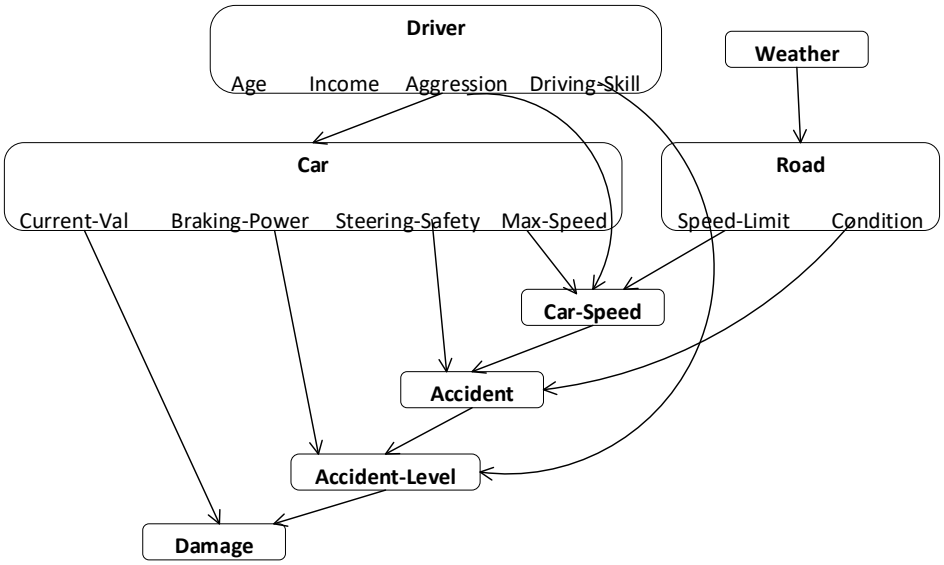
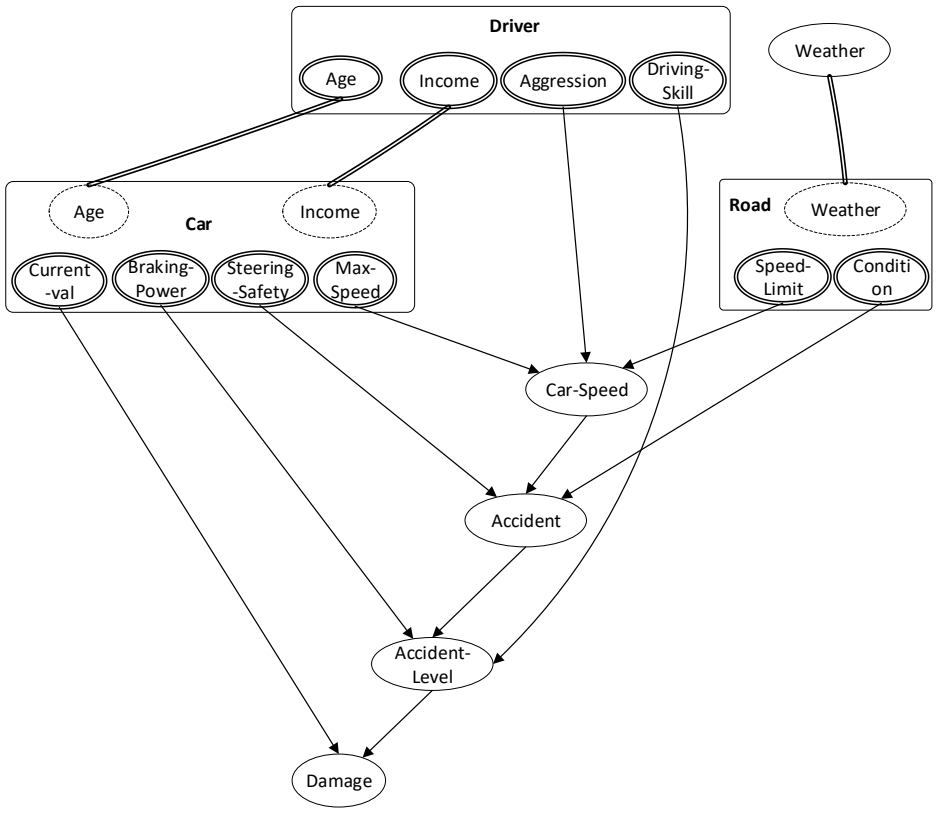


Figure 3.11: (a) Car Accident OOBN Model (Koller): the car OOBN class, (b) Car Accident iOOBN Model: the car iOOBN class



(a)



(b)

Figure 3.12: (a) Car Accident OOBN Model (Koller): the "Main" class. (b) Car Accident iOOBN Model: the "Main" class

A number of plausible ways to extend the model are suggested as follows.

1. The first thing to be considered is how to extend the model to a generalized vehicle accident model. The general vehicle can be categorized into *Bus*, *Ship*, *Speedboat*, and *Plane* if only motor vehicles are considered. The car accident model then falls into a

subcategory of the motor vehicle class. Coping-up with such a significant paradigm shift is possible using an iOOBN modelling framework.

2. The components in the vehicles can also be categorized based on the *Medium* they run on (*Road*, *Water way* or *Air way*), the part that is used to control the direction (*Steering* or *Helm*), the person who controls/drives it (*Driver*, *Sailor* or *Pilot*), and the part that helps in moving the vehicle (*Tyres* or *Propeller*).

3. Subdivisions can be plugged in using iOOBN without starting the modelling afresh so the accident model can incorporate the alternatives crash, damage, faulty/breakdown, burnt and sunk.

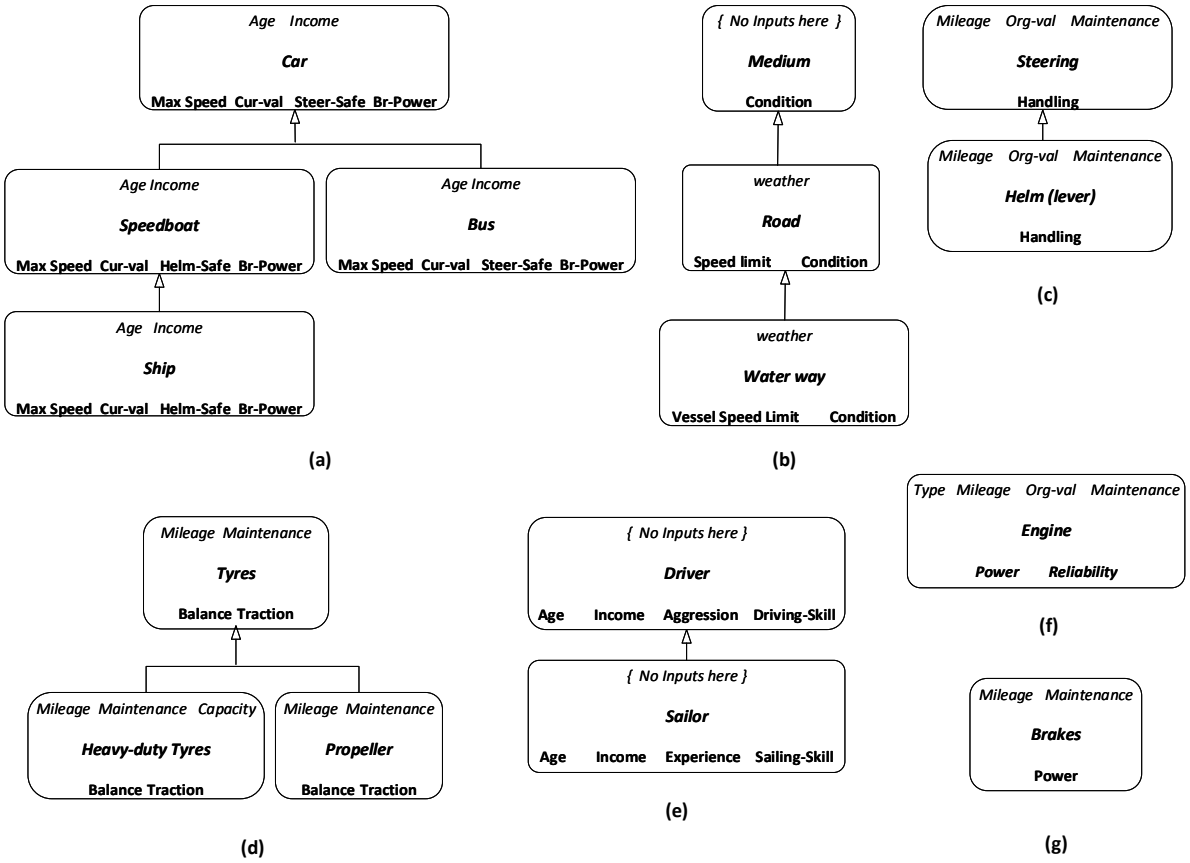


Figure 3.13: Car accident reengineered model (class hierarchies): (a) Car, Speedboat, Bus and Ship classes, (b) Road and Waterway classes, (c) Steering and Helm classes, (d) Tyres, Heavy-duty Tyres and Propeller classes, (e) Driver and Sailor classes, (f) Engine class, and (g) Brakes class.

For reasons of space, the hierarchy is shown in a semi-graphical format where interface nodes are represented textually rather than using ovals. The classes are represented by three lines of text within a solid-lined rectangle, (1^{st} Line) List of input node names, shown in *Italic* fonts, (2^{nd} Line) The class name, shown in ***Bold-Italic*** fonts, and (3^{rd} Line) List of output node names, shown in **Bold** fonts.

Figure 3.13 shows a potential class hierarchy tree to further extend the car accident problem. This model is reengineered and extended using the proposed iOOBN framework. As dis-

cussed earlier, the framework allows the reuse of components and facilitates the extension of a model. Hence, the hierarchy indicates that a car accident scenario can be extended to model a ship accident, bus accident, or speedboat accident with minimal effort. By using iOOBN, the **Bus** class and **Speedboat** classes can be derived from the**Car** class by changing parameters and types of nodes, adding or removing nodes, edges and objects. In the same way, the **Speedboat** class can be extended to a **Ship** class that can be used to model ship accidents. Furthermore, a **Medium** class is created that is used as a parent class of the **Road** class used in the original model (Koller). The **Road** class is further used to derive a new class, **Waterway**, by changing the type of **Speed Limit** to the **Vessel Speed Limit**. The **Medium** class helps to maximise reusability in case of defining a class which contains any of the attributes of the **Road** and **Waterway** classes. The **Driver** class can be easily extended to **Sailor**; thus, that is not shown here. Similarly, the **Helm** and **Heavy-duty Tyre** classes are derived from the **Steering** and **Tyres** classes, respectively, with necessary modifications. All these derivations facilitate modelling a more comprehensive, extensible and dynamic system. Table 3.3 shows how the extended classes for the extended model are found and what the alternative modelling choices are [8]. A **Ship Accident** model is shown in Figure 3.14 that uses the hierarchy of Figure 3.13, built using iOOBN. The advantages of the iOOBN in extending the original model are discussed in Section 3.4.5.

[8] The modelling choice is a random selection from a candidate list of potential ways of making a hierarchy.

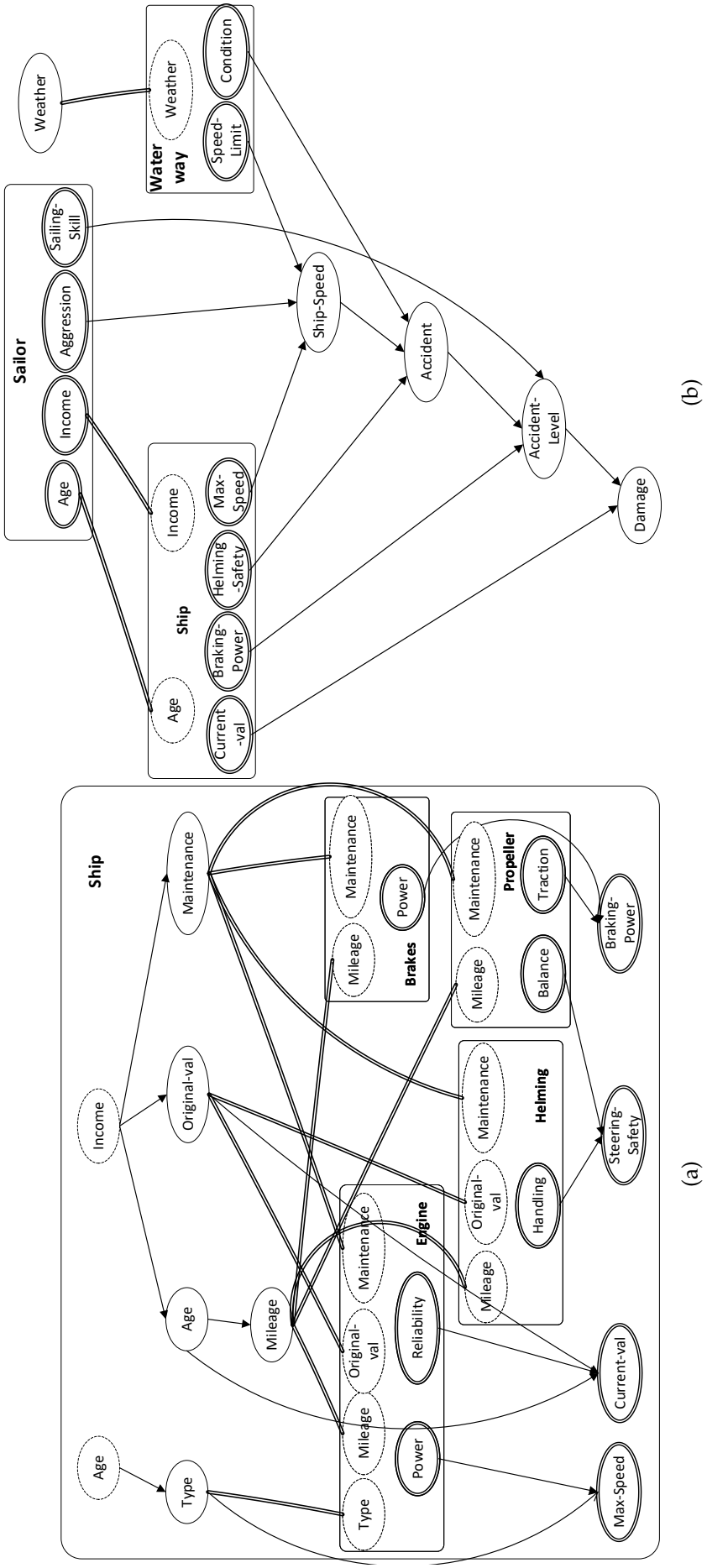


Figure 3.14: Ship accident iOOBN Model

Table 3.3: The modelling choice from the modelling options in constructing the Accident Hierarchy.

Extended Classes	Modelling Options	Modelling Choice
Bus	1) Changing parameters of Car class 2) Start from scratch	Changing parameters of Car class
Speedboat	1) Changing parameters of Car class 2) Changing the type of nodes of Car class 3) Start from scratch	Changing the type of nodes of Car class
Ship	1) Changing the type of nodes of Car class 2) Changing parameters of Speedboat class 3) Start from scratch	2) Changing parameters of Speedboat class
Road	1) Extending Medium class 2) Start from scratch	Extending Medium class
Waterway	1) Extending Medium class 2) Change type of Road class nodes 3) Change parameters of Road class nodes 4) Start from scratch	Change type of Road class nodes
Helm	1) Changing internal structures and parameters of Steering class 2) Start from scratch	Changing internal structures and parameters of Steering class
Heavy-duty Tyres	1) Adding nodes to Tyres class and changing parameters as required 2) Start from scratch	Adding nodes to Tyres class and changing parameters as required
Propeller	1) Changing internal structures and parameters of Tyres class 2) Start from scratch	Changing internal structures and parameters of Tyres class
Sailor	1) Changing parameters of Driver class 2) Changing the type of nodes of Driver class 3) Start from scratch	Changing the type of nodes of Driver class

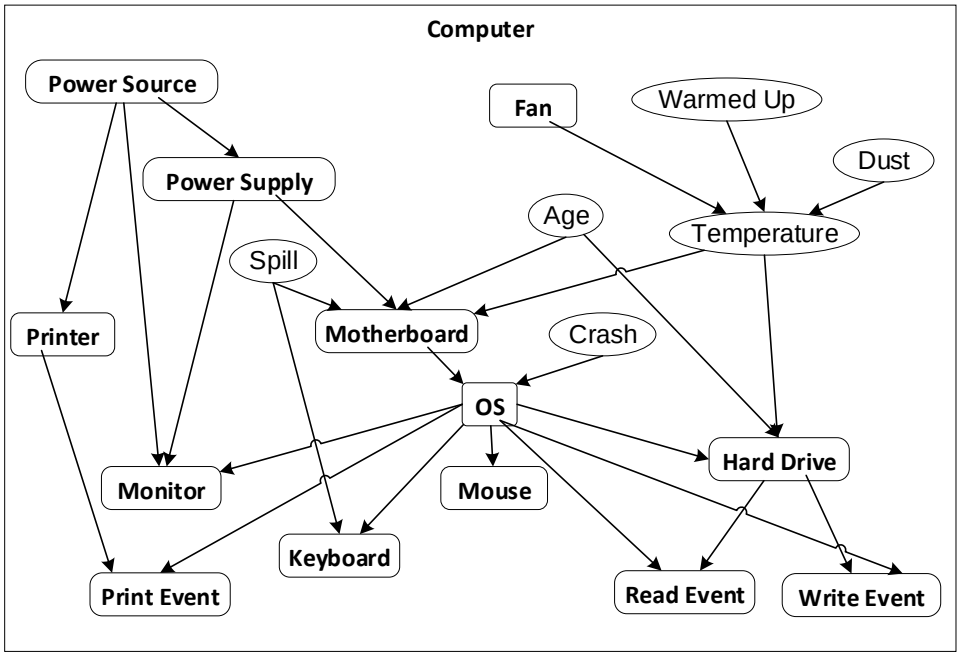
3.4.3 Computer problem diagnosis

In the original model for $Computer\ Diagnosis$, modelled in Pfeffer's thesis [203], (and as shown in figures 3.15a and 3.16a), there is a main class, namely $Computer$ that uses instances of the following subclasses: 1. $HardDrive$; 2. $PowerSupply$; 3. $PowerSource$; 4. $Printer$; 5. $Monitor$; 6. $Keyboard$; 7. $Mouse$; 8. $MotherBoard$; 9. $ReadEvent$; 10. $WriteEvent$; 11. $PrintEvent$; and 12. Some embedded standard nodes. The class $HardDrive$ contains instances of the following subclasses: 1. $DriveMechanism$; (which consists of instances of (a) $Motor$ (b) $Head$) 2. FAT; 3. $Controller$; 4. $Surface$; 5. $Cable$; and 6. Some embedded standard nodes.

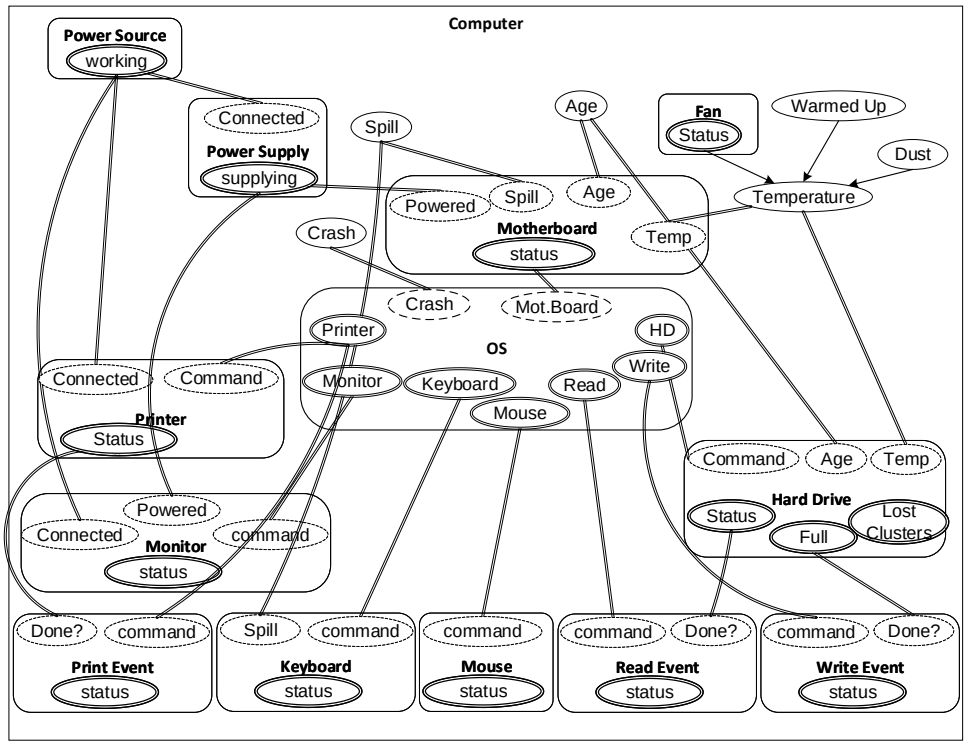
The depicted diagnosis system is modelled on the existing OOBN framework. As the iOOBN is a backward compatible system, reengineering the diagnostic system in iOOBN needs no changes. The **Computer Diagnosis** model is reengineered using iOOBN, as shown in figures 3.15b and 3.16b.

In order to exhibit more features of the OO paradigm, the system can be extended into an iOOBN. Since iOOBN allows inheritance, polymorphism, typecasting and abstraction, redesigning the whole system is possible to make it more flexible, extensible and more reusable.

In the original model (Pfeffer), all the possible computer system components are not included. If we seek to extend the system by adding more components (e.g.) for a more com-



(a)



(b)

Figure 3.15: The **Computer** class of the "Computer Diagnosis" model: (a) OOBN model (Pfeffer) (b) iOOBN model

plete diagnosis or if we want a system that is incremental, able to add new features on the go with the use of various technologies, then the iOOBN is a perfect fit. An original model can be extended to support the addition of more features with less effort (maximum reusability) using iOOBN. The hierarchy of classes and interfaces, constructed to extend the original model,

is shown in Figure 3.17. The hierarchy allows, for example, replacing **Hard Drive** with **SSD Drive** or adding support for **SSD**. The **storage** interface contains all the common IO nodes of **Hard Drive** and **SSD Drive**. This situation allows the reuse of common attributes by defining them only once. Similarly, the **Keyboard** class is derived from the **Mouse** class where **Mouse** is extended from the **Peripheral** interface. The **Printer** class is derived from the **Peripheral** interface and is used to derive the **Monitor** class. The hierarchy is easy to extend and ensures maximum reuse. In Section 3.4.5, more advantages of the hierarchy are suggested.

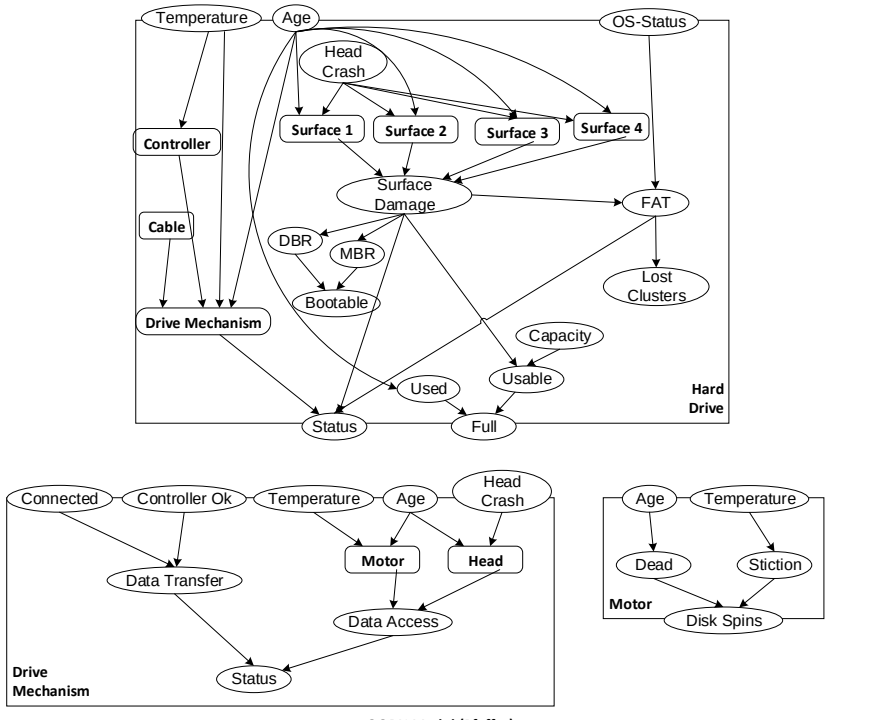
Figure 3.17 has three more hierarchies that start with the interfaces **Cable**, **Event** and **Drive-Mechanism**. The **Cable** interface is extended by **SATA** and **PATA** classes to support both old and new technology of data transfer. The **Drive Mechanism** interface is used to derive **HD Mech.** and **SSD Mech.** classes to control the **HardDrive** and **SS Drive** operations. The **Event** interface is extended in building the **PrintEvent**, **ReadEvent** and **WriteEvent** classes. These classes model the event manager of printers and storage devices.

Table 3.4: The modelling choice from the modelling options in constructing the computer diagnosis hierarchy

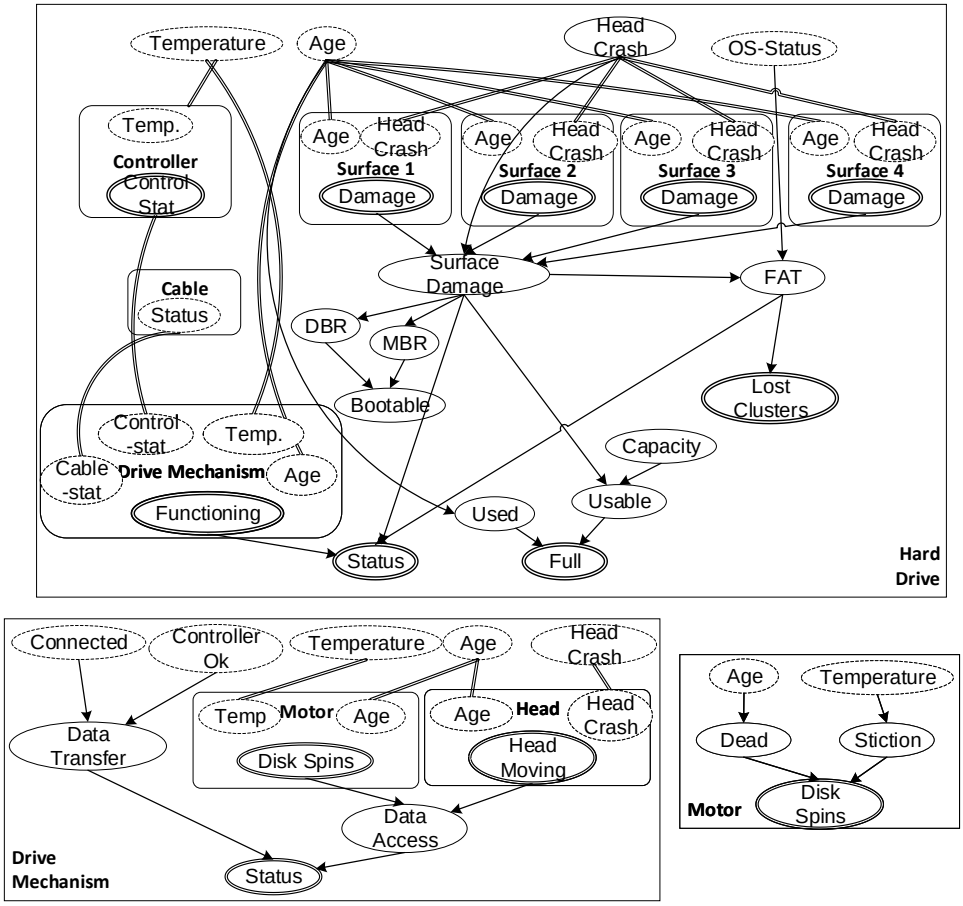
Extended Classes	Modelling Options	Modelling Choice
Mouse	1) Extending the Peripheral Interface 2) Start from scratch	Extending the Peripheral Interface
Keyboard	1) Extending the Mouse class 2) Extending the Peripheral Interface 3) Start from scratch	Extending the Mouse class
printer	1) Extending the Peripheral Interface 2) Start from scratch	Extending the Peripheral Interface
Monitor	1) Extending the Printer class 2) Extending the Peripheral Interface 3) Start from scratch	Extending the Printer class
SATA	1) Extending the Cable Interface 2) Changing the parameter of PATA class 3) Start from scratch	Extending the Cable Interface
PATA	1) Extending the Cable Interface 2) Changing the parameter of SATA class 3) Start from scratch	Extending the Cable Interface
Hard-Drive	1) Extending the Storage Interface 2) Changing the parameter and type of SS-Drive class 3) Start from scratch	Extending the Storage Interface
SS-Drive	1) Extending the Storage Interface 2) Changing parameter and type of Hard-Drive class 3) Start from scratch	Extending the Storage Interface
HD-Mechanism	1) Extending the DriveMechanism Interface 2) Changing the type and parameter of SSD-Mechanism class 3) Start from scratch	Extending the DriveMechanism Interface
SSD-Mechanism	1) Extending the DriveMechanism Interface 2) Changing the type and parameters of HD-Mechanism class 3) Start from scratch	Extending the DriveMechanism Interface
Print-event	1) Extending the Event Interface 2) Changing parameters of Read-event or Write-event class 3) Start from scratch	Extending the Event Interface
Read-event	1) Extending the Event Interface 2) Changing parameters of Print-event or Write-event class 3) Start from scratch	Extending the Event Interface
Write-event	1) Extending the Event Interface 2) Changing parameters of Print-event or Read-event class 3) Start from scratch	Extending the Event Interface

Each of the hierarchies, shown in Figure 3.17, can be built in several alternative ways and different forms, and the hierarchies displayed are representative of all possible hierarchies. A list of the possible ways is shown in Table 3.4 with the possible modelling options [8] and the chosen option to build the hierarchies is shown in Figure 3.17.

To demonstrate the extended model and how it can be used in making an extensible, growing and incremental model that supports a vast range of devices connected to the computer, where each type of device can represent a different technology or manufacturer, figures 3.18a and 3.18b represent two computers (Ann's and Sam's). Ann's computer has a **HardDrive** and related mechanism, whereas Sam's computer has an **SS Drive** and related mechanism. Both of the computer models are built using the classes in the hierarchies to support a generalized computer diagnosis.



(a)



(b)

Figure 3.16: Hard Drive, **Drive Mechanism** and **Motor** class of the Computer Diagnosis model : (a) OOBN model (Pfeffer) (b) iOOBN model

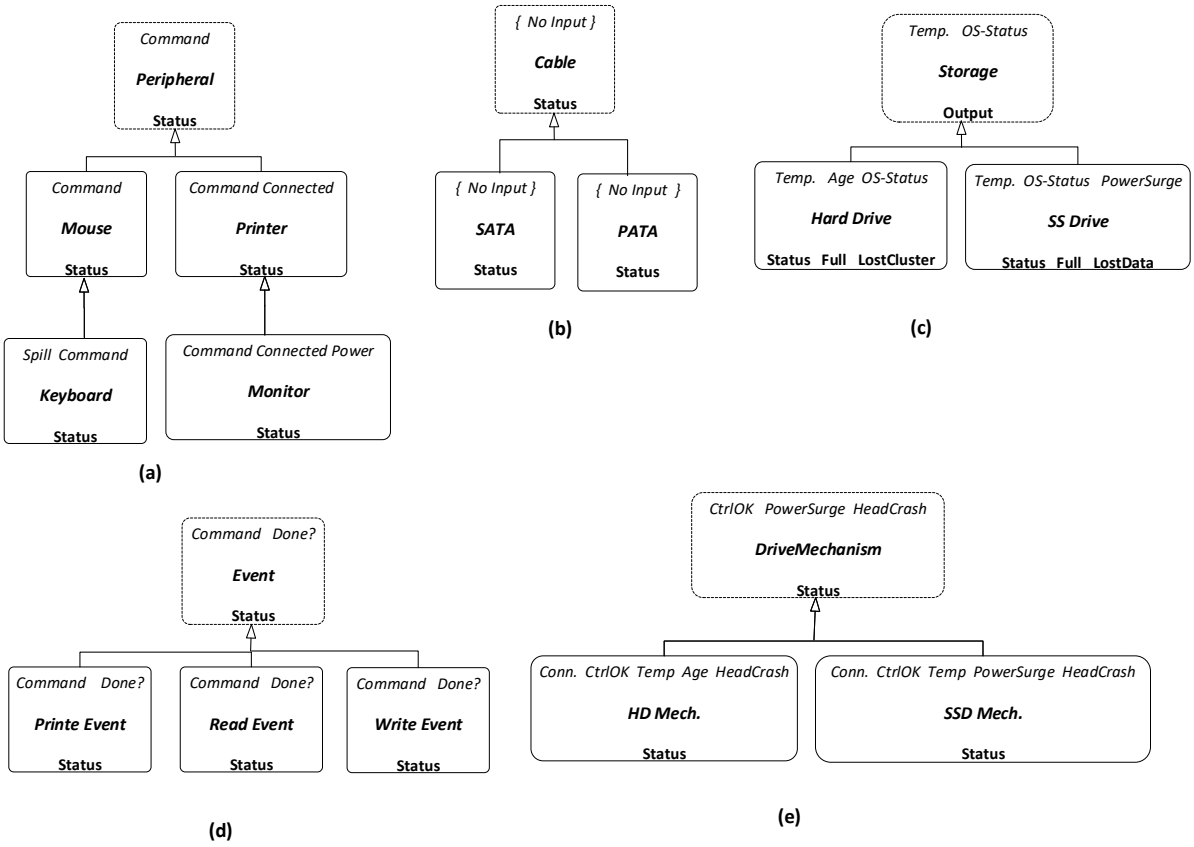
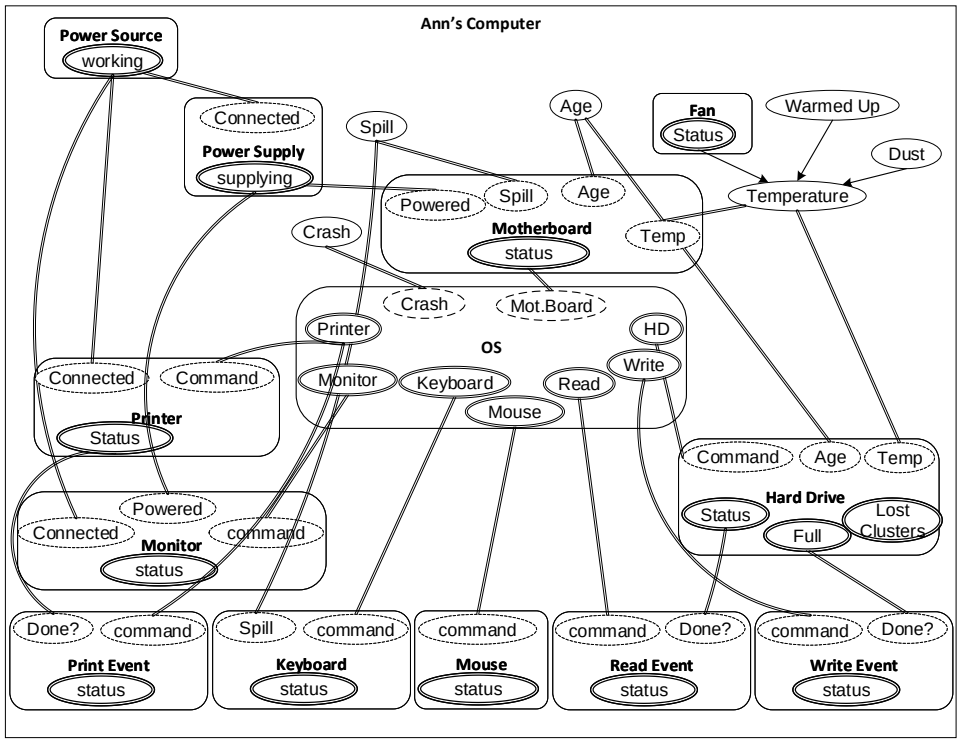
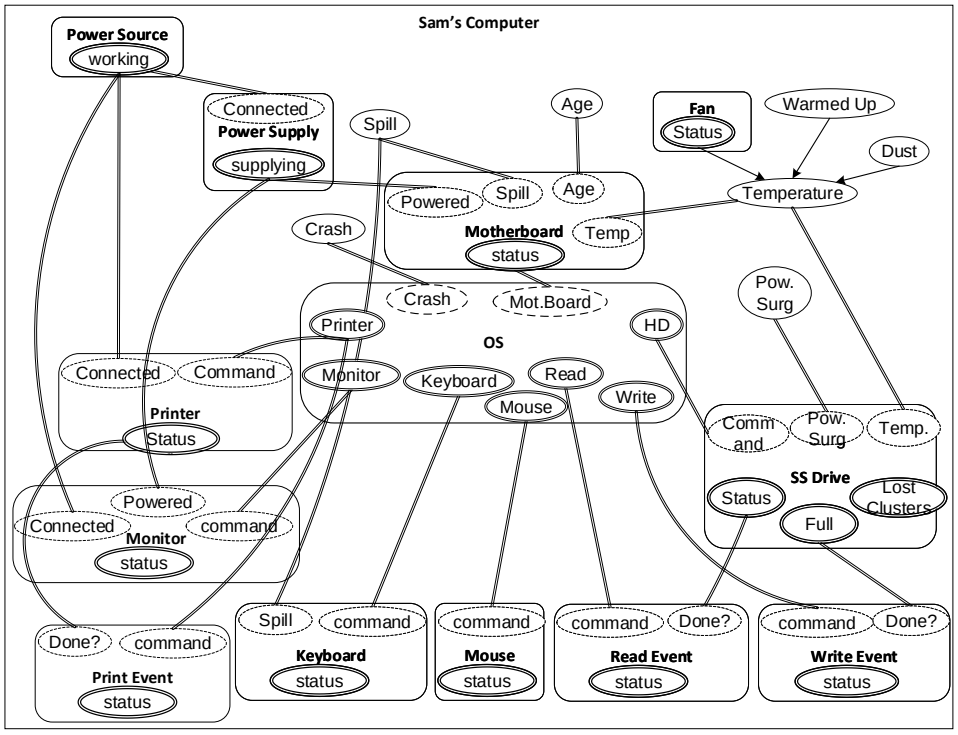


Figure 3.17: Computer Diagnosis iOOBN Class Hierarchy: (a) Mouse, Keyboard, Printer and Monitor classes, and Peripheral interface, (b) SATA and PATA classes, and Cable interfaces, (c) Hard Drive and SS Drive classes, and Storage Interface, (d) Print Event, Read Event and Write Event classes, and Event interface and (e) HD Mech. and SSD Mech. classes and DriverMechanism interface

The hierarchy is shown in a semi-graphical format where interface nodes are represented textually rather than using ovals. The classes/interfaces are represented by three lines of text within solid-lined rectangles and interfaces are shown with dotted line rectangles, (1^{st} Line) list of input node names, shown in *Italic* font, (2^{nd} Line) class name shown in ***Bold-Italic*** font, and the (3^{rd} Line) list of output node names is shown in **Bold** font.



(a)



(b)

Figure 3.18: Computer Diagnosis iOOBN model: the classes derived from the "Computer" class (shown in Figure 3.15b) (a) Ann's computer, (b) Sam's computer.

3.4.4 Power surge problem

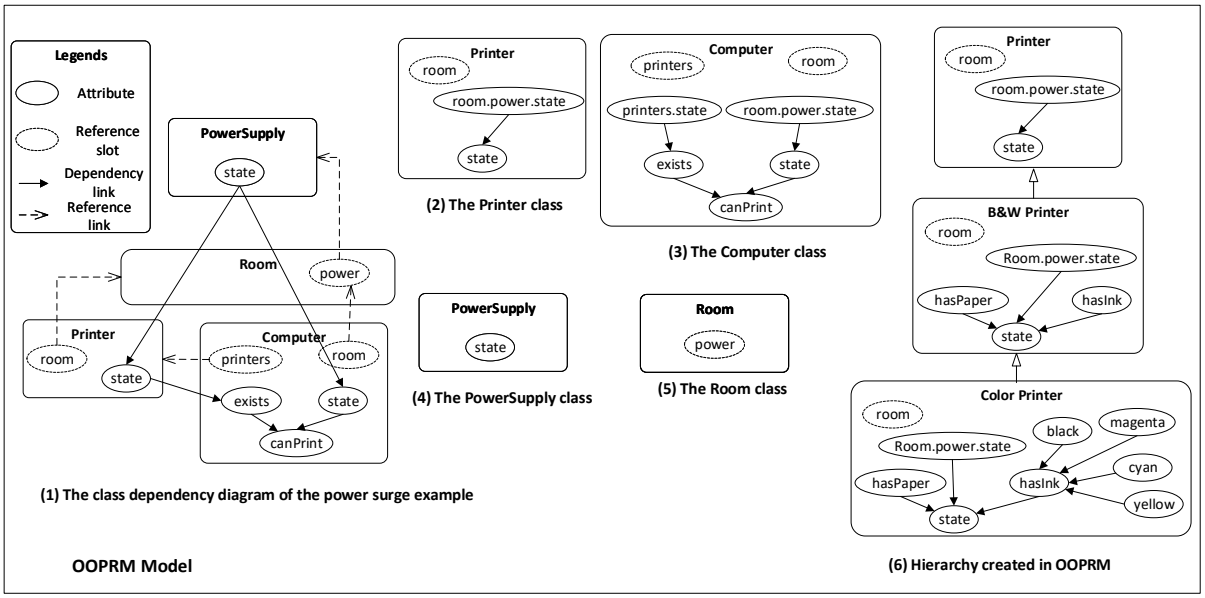
The original power surge (voltage spike) problem was proposed and modelled in [76] by Torti et al. They modelled a system using OOPRM (see OOPRM at the end of Section 2.4) to deal with such a problem. The dependency diagram in Figure 3.19a (1) depicts the relation between the classes and connections among their components. The classes in the model are: 1. $Printer$, 2. $Room$, 3. $PowerSupply$, and 4. $Computer$, as shown in the diagrams of Figure 3.19a (2–5).

The dependency diagram has two types of links, namely Dependency and Reference links. The former links are equivalent to the causal edges in the BN, OOBN and iOOBN. The latter links help to complete the semantics of the dependency links. To be precise, the reference links actualize the dependencies between the attributes (equivalent to chance nodes in BN) by pointing to the actual terminal nodes of a dependency link. As an example, in Figure 3.19a, there is a dependency between the *State* attributes of the **PowerSupply** and **Printer** classes. This link represents that in a BN version of the **Printer** class, there are two nodes, namely the *state* and *room.power.state*. The latter node is obtained by the reference slot *room* in the **Printer** class. This slot is connected with the **Room** class, and the **Room** class has a reference slot, namely *power*, via which a **Room** is connected to the *PowerSupply*. This class has an attribute, namely *state*; hence, the other node of the **Printer** class is *room.power.state*. The reference slots in this model are truly analogous to pointers in programming languages.

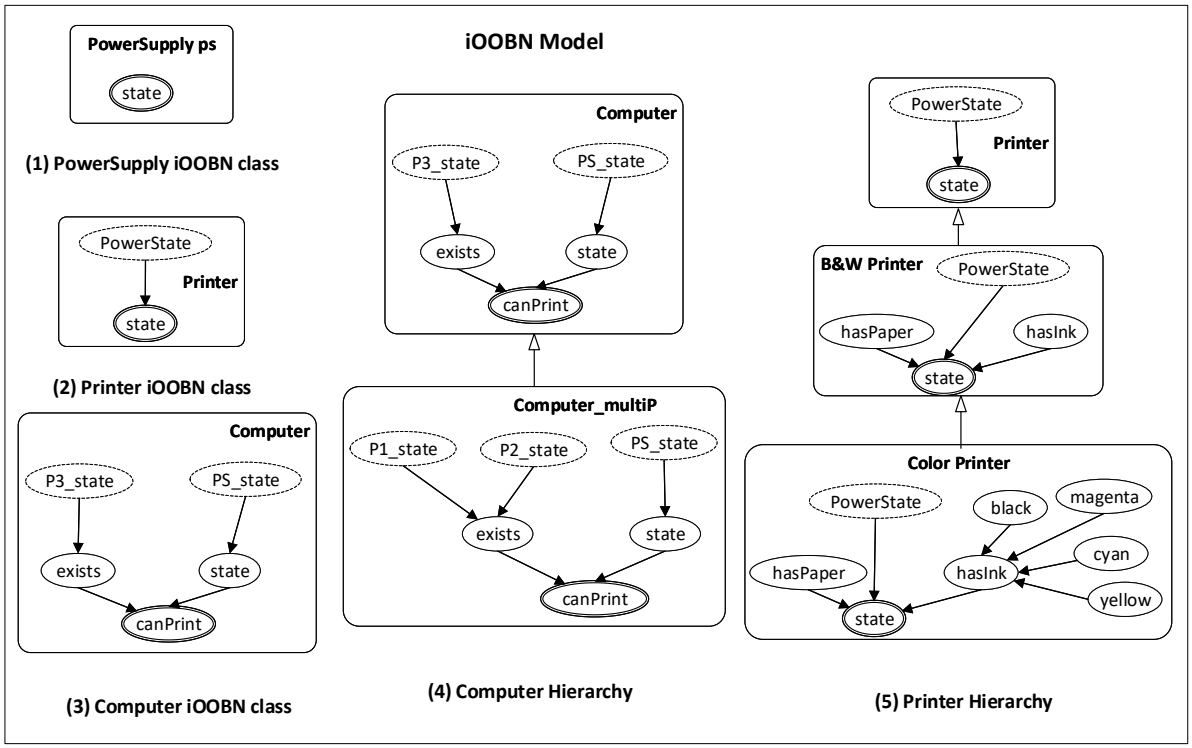
The model designed in OOPRM (Figure 3.20a) describes how sets of computers and printers are located in different rooms and powered by the electricity supplies of the corresponding rooms. Generally, power sources can fail due to voltage spikes, caused by power outages, tripped circuit breakers, lightning strikes, and various other reasons. Electronic devices like computers and printers may break down, depending on the intensity of the power surge and the age of the devices.

Each equipment has an attribute $state$ = {OK,NOK}. Computers have an additional attribute $canPrint$ = {can, cannot}. The power supplies are described by one of three power states, $power$ = {on, off, surge}. Finally, a computer is usually plugged in to one or more printers, and it can print if at least one of its printers is functional. The model also includes inheritance, as depicted in Figure 3.19, where a $Printer$ class is extended to a $B\&W$ or $Colour$ printer.

Reengineering of the model with an iOOBN is straightforward as OOPRM is a probabilistic analysis model that is based on relational algebra. An iOOBN, as a modern-day PGM, has the necessary capacity to deal with all the parameters that could previously be handled by OOPRMs and "relational algebra" and to perform robust probabilistic analysis. There are several ways to model such a system in an iOOBN. One way is to assign a different class for the



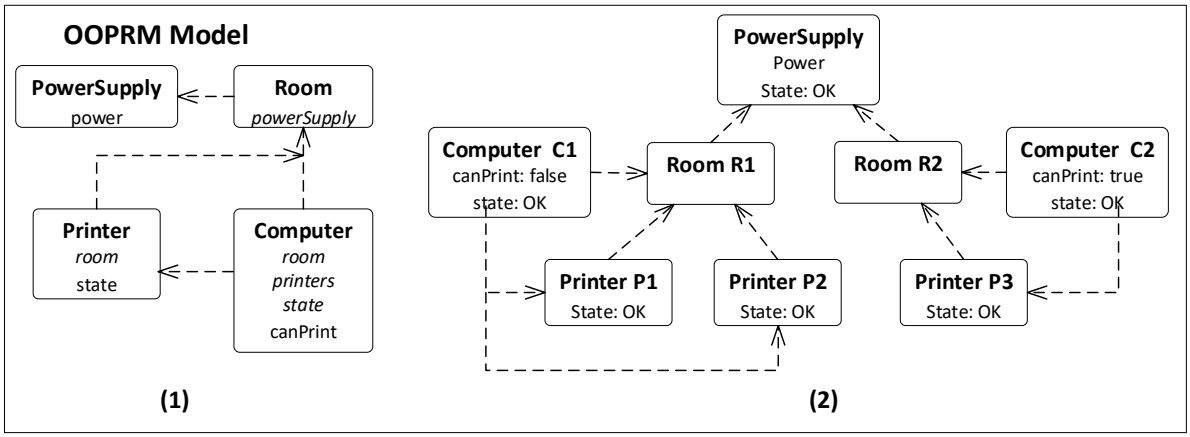
(a)



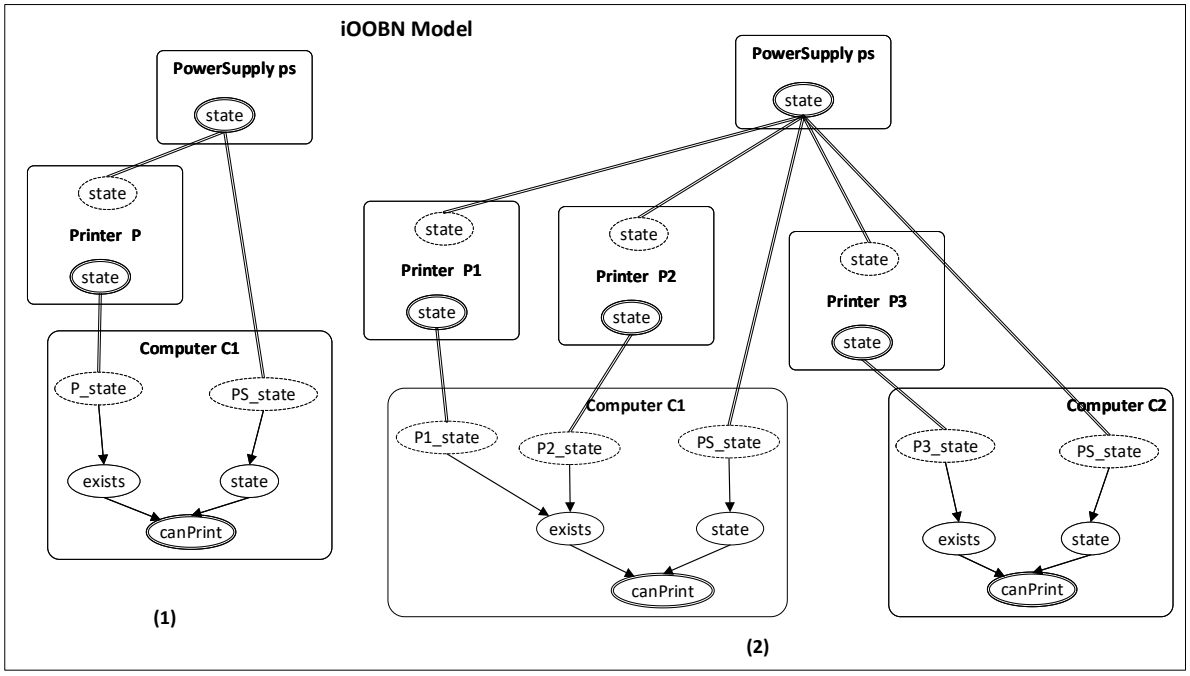
(b)

Figure 3.19: (a) Power Surge OOPRM Model (Torti) : (1) Class dependency diagram, (2–5) The Printer, the Computer, the PowerSupply and the Room class represented graphically. (6) inheritance of OOPRM to extend the Printer class. (b) Reengineered Power Surge model using iOOBN hierarchies: (1–3) PowerSupply, Printer and Computer classes, (4) the hierarchy with two classes, namely computer class and multi-printer computer class, (5) the hierarchy that is rooted at Printer class, then B&W Printer and Colour Printer.

specific number of printers connected to each computer and a different class for all the computers in a room. These classes specify the connections between the components or attributes.



(a)



(b)

Figure 3.20: The Power Surge model (a): OOPRM Model, (b): iOOBN Model with: (1) Single computer in a room, connected with a single printer, (2) two computers in a room, connected with one and two printers, respectively.

Another way of accomplishing such modelling in the iOOBN is to use the inheritance facility. If there is a class, **n_Computer**, having n number of printers connected (actually n instances of Printer class is embedded), then for another computer that is connected with more than n printers, the **n_Computer** class can be extended by adding only the additional printers. This strategy can be adopted where rooms might have different numbers of computers. The modelling of such a system would save modelling time and effort. As well, it saves computation time if incremental compilation or compilation (like $SIIC$ as presented in Chapter 4) is applied.

Figures 3.19b and 3.20b show a reengineered model, built using iOOBN, with **Computer**, **Printer** and **Power Supply** classes and their hierarchies (depicting the extensibility via inheritance). The classes in the root of the hierarchies are extended by adding required additional attributes (i.e., nodes) and edges for constructing new classes in order to make the model flexible to change and extensible.

A snapshot of the reengineered iOOBN model is shown in Figure 3.20b. In part (1) of the figure, the required number of instances of the three classes **Printer**, **Computer** and **PowerSupply** are shown. The state-space of the attributes are the same as the attributes of the OOPRM Model. A catch in the over all model is that a room might have one printer connected to a computer, while another room can have multiple printers connected to a single computer. In part (2) of the figure, a modelling technique to deal with such a critical scenario is shown. This kind of modelling is also accomplished easily in iOOBN and shown in the figure.

3.4.5 Advantages of iOOBN framework in reengineering and extending systems

The advantages of the aforementioned reengineering using iOOBN are discussed below:

A group of people can work on different parts of the system in parallel as per their expertise. So, in the beginning, an initial design can be made based on interfaces or abstract classes. This initial design helps to test and use the minimal system, as well as combining each part once they are complete.

The reengineered system is easy to extend and pluggable to similar domain problems, such as vehicle accidents those run on water (for car accident problem) and to a scanner or other peripheral device (for the power surge problem).

Every class (as shown in the hierarchy of figures 3.13, 3.17 and 3.19b) extends to subclasses, making it possible to add new ones to the model, such as one for the driver's physical or mental condition. Because the proposed framework makes the system incremental, this helps reduce the computation time and maximise the reuse of resources.

If inheritance is used to extend/derive one class from another class, then the proposed compilation technique (as outlined in Chapter 4) also helps to reduce the processing time of the network. In addition, the use of inheritance makes it easy to increment or extend the system. Simply adding additional components to the existing class results in a new class.

The proposed iOOBN system creates a model adaptable to change. If the target is enabling the portability of a system to cope-up with changes from the traditional to cutting-edge technologies, the iOOBN system can meet this challenge. New features can be added easily and changing some existing ones to new ones using the interface feature and dynamic binding is straightforward. To explain, dynamic binding is replacing an object of a class with an object of

its superclass or subclass without making any additional change in the instantiating class.

The modelled system becomes modular, thus helping to diagnose system issues easily and reducing the overall complexity of design and other system elements.

The iOOBN allows designing of a shadow-system when complete data is not to hand. It also facilitates different groups of people working with different technologies to combine their work thus meeting the the requirement to facilitate compositional modelling.

One of the biggest competitors of the OOBN is the OOPRM. This system has a significant issue in its formulation. It uses "reference slots" to define relations. The reference slots allow accessing components of a class remotely, i.e., without using an instance of the class, which actually violates the **encapsulation** property. The iOOBN, however, supports encapsulation and all the other principles of an OO system.

Another advantage of the iOOBN over the OOPRM relates to visualization. Although the OOPRM is a structured way of representing relations, it is a complicated model in which it is difficult to visualize; compared with the iOOBN, detecting faults and troubleshooting is hard. Problems of visualization, fault detection and troubleshooting in the OOPRM are largely due to the complex reference slots in that system, whereas the iOOBN system, using graphical representation, avoids these problems.

3.5 Case Study: Western Grassland Reserve Project

As a proof-of-concept case study, reengineering a real-life modelling project "WGR" (Western Grassland Reserve) [3] that was undertaken in Melbourne, Australia, was considered. The source project of this modelling project is the "Western Grassland Reserve" project run by the State of Victoria, Department of Environment, Land, Water and Planning (DELWP). (A detailed introduction to WGR is available in [3]). Originally, Bayesian Intelligence Pte Ltd (BAIPL) [9], a consulting firm, collaborated with DELWP in the project to develop an environmental decision support tool, modelling grassland species composition and forecasting how this might change under different management regimes. The tool was modelled using a Dynamic Object-Oriented Bayesian network (DOOBN) and implemented in Hugin software.

For reengineering purposes, this collaborative research activity was carried out with experts from DELWP and BAIPL, the bodies involved in the WGR project. Long discussion sessions, workshops and frequent meetings took place with the experts to gain insight into the whole underlying system.

The whole DOOBN model of WGR was analysed. That enabled extraction of a hierarchical structure from the hidden relations between the factors and components of the original

[9] www.bayesian-intelligence.com

DOOBN system. Then the whole system was redesigned using iOOBN software (see Appendix A for more about the software). The feasibility of the reengineered model was discussed with the experts.

3.5.1 Introducing WGR DOOBN components

The original DOOBN model for the grassland project included several input variables that represent the grassland at a particular time marked as "Start" time. The model predicts the values of these variables at a time marked as "end" time. The "end" predictions are then used as new "start" variables and make another prediction, and so it continues in time steps. Each timestep is considered as one season. The model was designed to make necessary management predictions over the long term (5 to 20 years).

The model predicts the state and likelihoods for the following variables: 1. Foliage covers, 2. Basal areas, and 3. Plant densities of species groups that inhabit the relevant grasslands (15 groups)). The grass/herb species are: 1. Kangaroo Grass; 2. Red-Leg Grass; 3. Windmill Grass and Panics; 4. Spear Grasses; 5. Wallaby Grasses; 6. Serrated Tussock; 7. Needle Grasses; 8. Exotic Annual Grasses; 9. Grain Crop; 10. Sens. Native Herbs; 11. Hardy Native Herbs; 12. Native Ruderals; 13. Blanket Weed; 14. Broadleaf Weeds; 15. Onion Grass. Experts defined these groups and aggregated the species based on similarity in growth histories and management responses. The relative covers, basal areas and growing densities of these species jointly describe the target grassland and allow assessment of its quality [3, 57].

The original DOOBN model is modular in construction, where each separate object represents each species group, allowing for viewing, editing, running and testing the species separately. There are certain factors and properties of each of the species that need consideration for the restoration of the grassland. These factors influence the predictions made by the model. These can be categorised into two groups: management interventions and the impact of nature (environmental factors). Certain management interventions were taken for plant species' preservation as follows. [10]

1. **Burn:** This removes all cover and basal area (at different rates for different species). This action is only taken if there is sufficient biomass.

2. **Grazing by cattle:** In order to remove the basal area and cover from certain species groups, cattle can be introduced. The introduction of cattle depends on the species' vulnerability to grazing. Grazing is performed at different combinations of time duration and rates of stocking.

[10] The management interventions with their brief discussion are mostly taken from the WGR report [3] that is available in the official website of DELWP.

3. **Herbicide:** Herbicides can be applied to remove basal area and cover. A specific herbicide removes basal area and cover from species groups that are vulnerable to the herbicide. Multiple herbicides and spraying areas (a single spot or across the whole site) need to be accommodated in the model.

4. **Remove rocks:** To convert an intact grassland into cultivatable land, physically removing rocks is performed, and this item models the action in DOOBN.

5. **Scarify:** Scarifying the soil makes the site more favourable for seedling growth and survival.

6. **Sow Themeda:** Sowing Kangaroo Grass (Scientific name "Themeda triandra", hereafter "Themeda") introduces new Themeda plants, which respond favourably to climate, competition with rival plants and local soil conditions to mature and thus increase basal area and basal cover.

7. **Remove Topsoil:** In order to re-set an area with low nutrients by removing all basal area and cover, it is necessary to remove seed banks. Topsoil removal is one way to do this.

8. **Carbon Boost (add Carbon):** Another action that temporarily reduces nutrient is Carbon boosting. That is a tool used to increase the competition among the species groups.

9. **Add fertiliser:** this action makes some of the species groups relatively stronger in the competition to survive by improving the soil composition of the grassland.

The environmental (natural) factors that need to be considered in the model are:

1. **Competition:** In different seasons the cover and basal area are added or subtracted to different species group. This mimics the effect of competition between species. Given a particular season, the amount of cover added or subtracted to a species group depends on its competitive ability. This ability varies according to climate, nutrient status, the abundance of other species groups, and soil scarification.

2. **Climate:** The DOOBN model is capable of cycling through summer, autumn, winter and spring in turn by assigning different growth conditions using meteorological data. This cycling action allows simulation of the scheduling of management interventions in different seasons.

Finally, "Cost" and "Benefit" are the two other critical factors considered in the modelling of WGR. Each management intervention has a cost associated with the implementation of the

action. The costs incurred by implementing the management actions are reported as a cumulative cost. Some interventions are expensive if applied on a large area (e.g. spot spraying), others are not necessarily expensive, even if applied on a larger patch (e.g. burning). Thus, the inputs specify the size of the site in order to adjust the cost accordingly. In order to assess the benefits of outcomes, the 'Key Performance Indicators' (KPIs) as defined by DELWP are used. Outcomes may be both positive and negative depending on the KPIs. Another metric is developed by DELWP to balance (i.e. trading off) between the KPIs. It can assess the overall change in grassland quality.

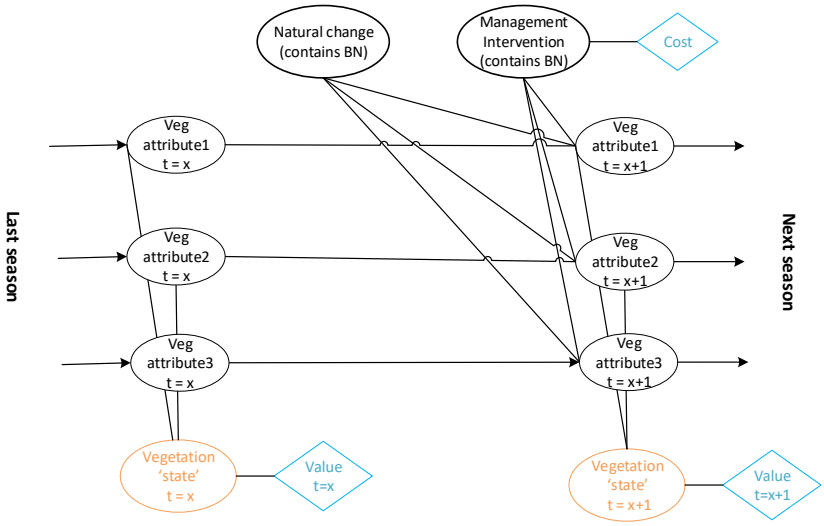


Figure 3.21: The template of the grassland DOOBN model classes. (copied and redrawn from WGR report [3])

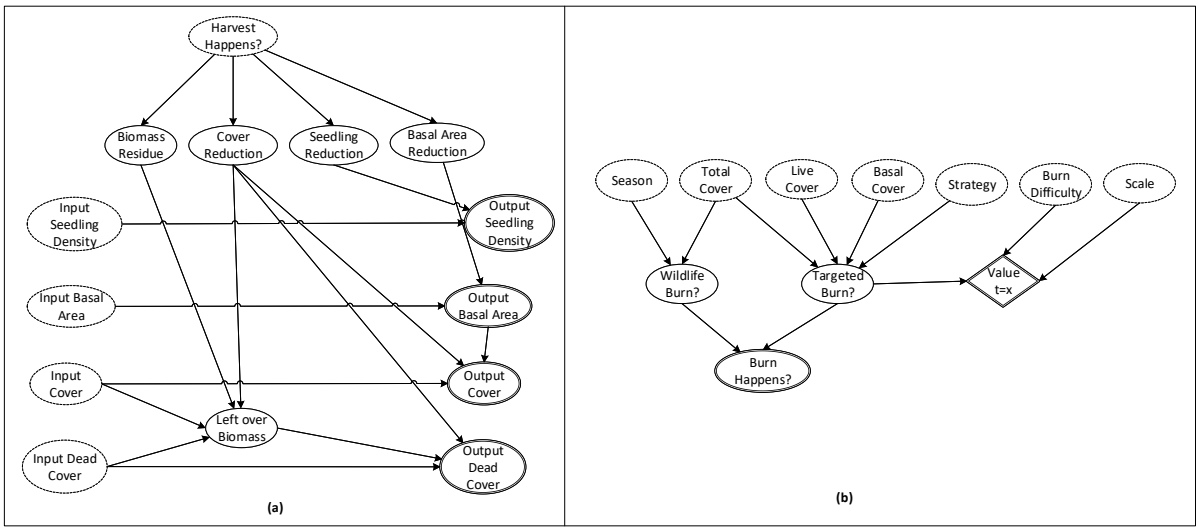


Figure 3.22: Two of the 129 classes of WGR (a list of the 129 classes are given in Figure B.3 of Appendix B): (a) Themeda Harvest, (b) Burn Intervention (copied and redrawn from WGR class repository [3])

In Figure 3.21, a template DOOBN network structure, used in WGR modelling, is shown. The *veg* attributes on the left are inputs to the model for the current state in time "x" that gets potentials from the output nodes of the previous state, i.e., in time "x-1". The attributes on the right are the output of the current state and provide the potential to the next state, i.e. in time "x+1". The nodes on the top are to model natural and management interventions with associated utility "cost". The utilities can also be carried forward to the next states, as presented at the bottom. In the original WGR model, there are 129 classes (as listed in Figure B.3 of Appendix B) and Figure 3.22, part (a) and (b), shows two of them as an example to illustrate the use of the template.

3.5.2 Reverse engineering the WGR DOOBN using iOOBN

In order to check the suitability and efficiency of the proposed framework in large-scale complex real-life applications, the WGR [3] was chosen for reengineering. In a grassland reserve, there are many factors to consider and many of them are uncertain. The factors include various species of grasses and a variety of behaviours, the effect of weather and environment, and the need for numerous classes in the DOOBN modelling. Hence, the reengineered system is modelled using the proposed framework in order to provide a simplified view of the problem and a scalable expandable solution with inputs from domain experts involved in developing the original WGR DOOBN. Domain experts in this case are the group of people with knowledge and experience in agriculture and species of plants prevalent in western grasslands. Local experts provided with information of similarities and dissimilarities of the characteristics, behaviours, biological attributes of the plants and plant species; their reactions to foreign substances like chemicals, fertilisers, and insecticides; and how they respond to managerial actions like burning off and soil scarifying.

The limitations of the original implementation with the DOOBN model are also considered in reengineering the model. There is already a form of OOBNs proposed [5, 46, 50] (see sections 2.5.3, 2.5.4, and 2.5.5 of Chapter 2) and available implemented with Hugin BN software. It is not only a well-designed and fully implemented working model, but also contains sufficient complexity to demonstrate what is expected to be revealed as the advantages of using an iOOBN. For example, it can accommodate multiple class structures that have similar elements, e.g., Themeda and other native grasses. By reengineering (and re-implementing) using iOOBN, the accuracy and efficiency of the iOOBN version can be compared with the original DOOBN version. In Section 5.5.4 of Chapter 5 and Appendix D, there is another kind of comparison with the original WGR and the automatically learned model, learned by the proposed algorithm (Chapter 5).

The ecological management response model presented in [3] is complex and has several backward features due to the Hugin software used for implementation, which does not support inheritance. These relate to OO principles, including absence of reusability and inheritance. The weak points of the model, other than lack of inheritance are:

1. Much specific information is required (e.g., the life history and germination requirements of many grassland species).

2. Many interventions are needed.

3. Multiple scales of management.

4. The incorporation of many expert opinions.

The analysis of components started by considering the semantics, as well as the commonalities of structures, of the 129 classes in the original WGR DOOBN. Broadly, two clusters of classes, with no intercluster similarity were identified; hence the reengineering into iOOBN was done separately for each of the two clusters as noted below.

1. Group of 96 classes: containing only chance nodes and the same set of attributes distributively (representing the growth cycle of different kinds of native and exotic grasses). Figure B.1 shows the reengineered model in the form of a UML class diagram for this cluster.

2. Group of 33 classes: containing another set of attributes. They contain chance, decision and utility nodes containing other sets of attributes (representing the management strategies and their effect on the growth cycle) Figure B.2 depicts this part of the reengineered model in the form of a UML class diagram.

The mapping between the new classes (in the reengineered model) and the original WGR classes is shown in Figure B.3. Note, the mapping is between the leaf nodes of the hierarchy tree of the reengineered model and the original 129 classes of the WGR DOOBN. The leaf nodes of the reengineered models are not the actual concrete classes that can replace the original 129 classes. Actually, the mapping represents that the classes in the leaf nodes can be extended to derive a particular set of original WGR classes.

In the first phase, the chance-node-only classes were classified into 25 groups based on their similarities and dissimilarities with respect to ecological and biological characteristics. The classification and clustering resulted in eight interface nodes, three abstract classes and 25 concrete classes; these were in four distinct hierarchies with 23 derived classes (classes that have been constructed by inheriting properties of other classes or interfaces rather than from

scratch). For a concrete class representing different species and original WGR classes with their only difference being that of parameters, the concrete class is extended, and the parameters are changed accordingly. Objects of those classes are then created to use in the final model to act and serve like the original WGR model. As well, among the 25 classes, most have been constructed by inheriting the properties of existing classes or interfaces rather than from scratch. This fact also plays an important role in the proposed framework to provide efficiency. If a new class (reengineered iOOBN class) in Figure B.1 and B.2 has mapping with more than one of the old classes (from the original WGR DOOBN), it means that those old classes have same structure but different parameters. Hence, the reengineered classes then can be extended into different concrete classes with required parameterization to use them in the main class of the WGR model to make it fully functional.

The new system was modelled using the developed prototype version of the iOOBN tool. It ensures an optimized and scalable system design. From the first cluster, the gain obtained is 63% less computation due to reusing of components (since for 96 original classes the reengineered iOOBN model has eight interfaces, three abstract and 25 concrete classes, that is, 36 total structures, we have $\frac{96-36}{96} = 63\%$). Such scalability is an indicator of the efficiency of the proposed framework. Note, the 63% efficiency gain is just an indication of how much effort can be saved in constructing the graphical structures of the classes if the proposed framework with inheritance is used. However, in terms of parameterisation and number of concrete classes required to construct, this gain is negligible. In reality, it is difficult to actually quantify the gain. As an example, assume that "Kangaroo grass", "Wallaby grass", and "Onion" are three classes representing three different plants. Also assume that "Wallaby grass" has a superstructure of "Kangaroo grass" and "Onion" also has a superstructure of "Wallaby grass". Now, if a modeller wishes to make "Wallaby grass" class from "Kangaroo grass" class and then "Onion" class from "Wallaby grass" with no inheritance, they need to manually copy and paste the classes and make the required changes. This manual class creation has the risk of human error and requires human effort. Let us suppose a set of changes is required to be performed in "Kangaroo grass" and similar changes need to be conducted in "Wallaby grass" and also in "Onion" class. This is also known as propagation of changes in an inheritance hierarchy that needs considerable effort and has a high potential of risk if performed manually. If the modeller uses systems/tools that support inheritance, such as iOOBN, these issues will be managed with no human effort and hence no risk of human errors.

There is another way of measuring the efficiency of the reengineered model, presented in Chapter 5 and applied to the reengineered WGR model in Section 5.5.4. In section 5.5.4, the computation is detailed for the reengineered WGR model.

In the second phase, the 33 more complex classes were considered. They contained 13 classes that were distinct, dissimilar and large in size; no reengineering into a class hierarchy was possible for these. Of the remaining 20 classes, the reengineering resulted in 16 concrete classes of three hierarchies with 13 derived classes. A UML class diagram for the reengineered WGR model, along with a mapping of the original WGR classes into the iOOBN classes, is provided Figure B.1 and B.2, respectively.

3.5.3 Summary of the reengineering

Knowledge extraction from the original DOOBN system is laborious and requires extensive computation of the same repeated redundant structures. Inference, decision-making, or generating reports from such a large system is subject to efficiency and scalability issues. Moreover, extending such a complex and vast system is quite challenging.

In contrast, the proposed iOOBN framework is extensible, reusable, scalable and efficient. Overall, for the original 129 classes with no hierarchy and inheritance (reuse of existing components/classes), iOOBN modelling offers an inheritance hierarchy with eight interfaces, three abstract classes, 55 concrete classes (25 in cluster-1 + 13 in cluster-2 (not reengineered) + 16 in cluster 2 (reengineered) + 1 main class) and 36 derived classes. These classes can be used to derive the original 129 classes with no extra structures to be added. Therefore, in terms of human effort, required to construct the 129 classes, there is an evident 49% reduction (129 classes down to 66 components). This reduction demonstrates that in large complex models, a class hierarchy can represent common elements in more compact ways (rather than working with a large number of unrelated classes that actually have many common elements, as in the original WGR OOBN).

The reengineering case study suggests that the iOOBN framework can be useful in producing more efficient, reusable, extensible and scalable complex BN models. More importantly, the reengineering demonstrates the potential to capture in the iOOBN framework commonalities across different parts of a model, captured in interface, abstract and concrete classes that are not leaves of the class hierarchy. These also show the potential for reuse. The reengineered iOOBN version retained the encapsulation of the original OOBN model (via embedded object), and demonstrated that the implemented iOOBN type checking/typecasting worked as expected.

The reengineered model of WGR, implemented in iOOBN had the following advantages:

1. The reengineered model is flexible enough to cope with group-based development. Interfaces and abstract classes provide abstraction and facilitate even incomplete and partial implementation of a particular segment. Later these segments could be replaced by

completed, more efficient or more related segments.

2. More specialisation can be assured by strong type checking where only explicit typecasting is possible. Along with encapsulation and inheritance, typecasting and overriding provide scalability and ensure maximisation of resource reuse.

3. More compact representation such as this facilitates understanding of the model and obviously reduces the complexity of the associated documentation.

4. Extending such a complex and large real-life model is quite challenging, especially if the original BN modellers and/or the domain experts who build the original version are no longer available.

3.5.4 Validating reengineered models

A reengineered model needs to be validated to check for its feasibility, and, ideally, should be equivalent to the original model. A reengineered model is worthless unless its viability and validity is proved. There are three ways to check for equivalence of the systems.

1. **Flattening both systems into BNs and comparing the resultant BNs:** In Section 3.2.1, a special operation, "flattening" is described. This process is used to transform an iOOBN and OOBN class into a BN. This BN is used for inference in the absence of incremental compilation or advanced compilation techniques such as SIIC (proposed in Chapter 4). Therefore, to check for the equivalence of the reengineered iOOBN system with the original WGR DOOBN, both systems were flattened into BNs and their structural similarity check was performed using a simple graph matching technique. Then node-by-node parameter testing was performed. This ensured the equivalence of both systems. According to WGR DOOBN experts, the original project is huge and not runnable directly in Hugin. Hence, they had to develop a stochastic system to retrieve the inference outcome from the system. The reengineered iOOBN system is able to produce an inference outcome if a SIIC compilation algorithm is used. Hence, it would not be straightforward to check for equivalence of the systems by the flattening method.

2. **Expert opinion:** A second validation method, used for the WGR case study, was to have domain experts (in this case, some of the same people used for expert elicitation to build the original DOOBN model) review and validate the reengineered iOOBN WGR classes. They accepted the semantics of the revised hierarchy and confirmed the similarity in in-

ference outcomes of partial and small segments of both the original and the reengineered systems.

3. **Comparing the reasoning/decision outcomes of both systems:** The final method for validating the reengineered models is by comparing the posterior probabilities and/or expected utility computations for decisions from the reengineered models to the originals. This was done for all the toy example models presented in Section 3.4 above; however it was not possible to perform this for the complete WGR model, as only the OOBN classes were provided, not the software to run the 20-year model predictions (using stochastic simulation, rather than exact compilation-based methods).

3.6 Summary

Various forms of OOBNs have been proposed in the literature to help make BN technology cope with large-scale problems and to support reuse and ease of maintenance. These frameworks have resulted in OOBN systems that contain hierarchical composition along with encapsulation. However, the key OO feature of inheritance, which brings a higher level of reusability and scalability, although proposed long ago, has not been formally or fully specified for OOBNs, nor implemented in any OOBN software that we are aware of. This chapter presented the iOOBN theoretical framework, which defines inheritance, polymorphism, encapsulation and abstraction for OOBNs. A prototype version of the framework has been implemented (see Appendix A) using the API of an existing BN software package. The framework has been illustrated via a simple running example, and its potential efficacy has been demonstrated by the reengineering of a number of small OOBN examples from the literature, as well as by reengineering a large real-world dynamic OOBN, into the iOOBN framework. The reengineered WGR iOOBN class hierarchy demonstrates the framework's potential to capture similarities in different parts of the model, i.e. generalise elements of the model.

The iOOBN framework has been designed to support team-based development of complex BN models. Its interfaces and abstract classes provide abstraction and should support the incomplete and partial implementation of parts of a larger system, where later in the model development process, these segments can be replaced by more detailed or fully concrete classes. The strong type checking feature within iOOBNs should assist modellers to avoid confusion or modelling flaws which may lead to significant problems in the resultant model. Along with encapsulation and inheritance, typecasting and overriding may provide scalability and increase the potential for component reuse. While the WGR reengineering case study gives some indication that iOOBN may lead to such benefits, until there is extensive real-world development

of iOOBN models, there can be no definite evidence for or against these claims.

It is apparent that compiling an iOOBN into a flattened BN (as done in the Hugin software) introduces a complexity issue for either exact or approximate inference (as occurred with the WGR DOOBN model). The next chapter presents an exact inference algorithm for classes in iOOBNs which can be used to avoid flattening.

Chapter 4

Incremental Compilation in iOOBN

Object-Oriented Bayesian Decision networks (OOBNs) allow modellers to construct compositional and hierarchical models, using an inheritance hierarchy of classes and subclasses, enabling reuse and supporting maintenance. Reasoning with both ordinary BNs and OOBNs requires the important computational task of inference, the computing of new posterior probability distributions given a set of evidence. A widely used inference technique in ordinary BNs involves compiling the BN into a junction tree (JT) before performing inference; the compilation step is only performed when the network changes. In current OOBN software, the OOBN is transformed into the underlying BN (so-called "flattening") and; then any standard inference can be performed. Researchers have proposed methods for the incremental compilation of BNs, rather than recompiling from scratch for each network modification; these can also apply to OOBNs after flattening. This chapter presents a new incremental compilation technique that reuses existing compiled JTs of both embedded components and superclasses, and that does not require flattening. The description of the proposed technique follows detailed and illustrative related terms. It is shown that this can reduce compilation time. Afterwards, the performance of the proposed algorithm is analysed asymptotically and empirically and compared with the compilation technique used by a widely used commercial software package (Hugin) [27].

4.1 Inference in OOBNs

Bayesian Decision networks (BNs) [13, 204] are a powerful and widely used tool for reasoning under uncertainty. They can be built by automated learning if data is available or by using elicitation methods to capture expert knowledge when it is not. One of the key objectives of building a BN is to perform inference in order to perform reasoning under uncertainty. There have been many attempts to perform such a complicated and expensive operation efficiently. Inference in a BN is divided into two main steps, namely, compilation and message-passing. Compilation is the most complicated and expensive step which involve some underlying oper-

ations that are NP-Hard in complexity if performed optimally. Some heuristic-based methods have been adopted to suboptimally performing the task. However, there is no direct compilation (or inference) technique proposed to date. Currently, to perform inference in an OOBN, the network is first transformed into the underlying "flattened" ordinary BN (as shown in Algorithm 3.1) [1], then any of the existing inference approaches, including the widely used junction tree (JT)-based approach [205], can be applied. Flores et al. [4] proposed "Incremental Compilation" (InC), to make the ordinary BN inference more efficient by only re-compiling part of the network, with Bangsø et al. [5] proposing a similar incremental compilation for OOBNs, after flattening the network first.

In this chapter, a new incremental inference algorithm is presented for the iOOBN that reuses JTs compiled for either embedded classes, or when the new class is inherited from a previously compiled class. Section 4.2, presents the background, definitions and terminology of the relevant BN and iOOBN representations, including JT inference.

4.2 Inference, Clique Graph, Junction Tree, and Junction Forest

This section adopts, the ordinary BN and OOBN/OOBN definitions and terminologies, as used in [200] and [1] and as implemented in Hugin. Terminologies of iOOBN defined in Chapter 3 are used to formulate the proposed compilation algorithm. Note that for a BN with decision and utility nodes [13], incremental compilation applies only to the chance nodes representing random variables, so the algorithm does not consider decision and utility nodes. The algorithm also treats only discrete BNs, where all the nodes have discrete state spaces. The extension of the algorithm to cover classes with the chance (discrete and continuous), decision and utility nodes are a subject for further research and investigation. Hence, it is left as a future research direction.

DEFINITION 4.1 : INFERENCE

Inference in a BN (Definition 2.2) is the process of calculating posterior probabilities of a set of variables X (where each variable $v \in X$ is represented as a node v) given a set of evidence E and is denoted by $P(v|E)$ $v \in X$. This process is also known as compilation or probability propagation or conditioning or belief updating [1].

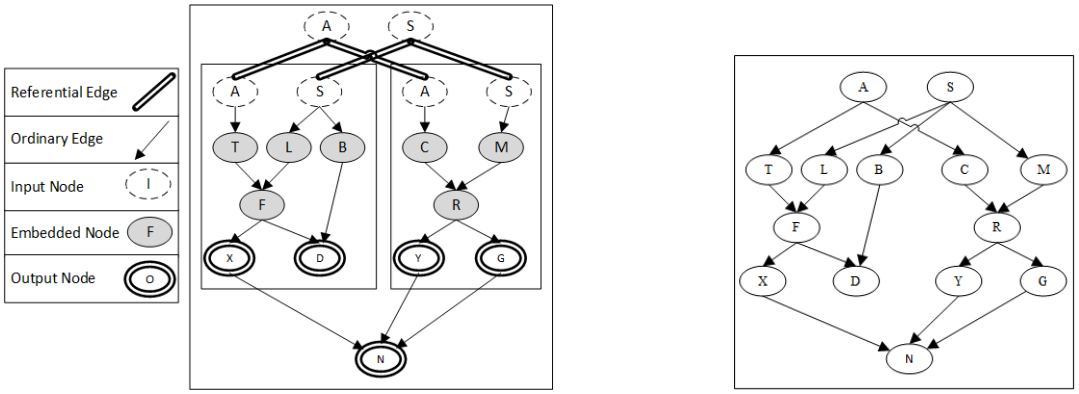
Note that the iOOBN framework defined in Chapter 3 and in [140] has two types of classes: abstract and concrete. In an abstract class, some of the parameters Π are not fully defined. So only instances of concrete classes can be compiled, and hence are the only classes used in the compilation algorithm. For the remainder of this chapter, all classes are assumed to be concrete

[1] An assumption is made that all OOBNs considered in this chapter can be flattened to a valid connected BN.

classes and are referred to as simply classes. Moreover, Chapter 3 proposes two types of inheritance, namely interface inheritance and class inheritance. In class inheritance, subclasses are derived by only adding new components to an existing class, whereas in interface inheritance subclasses are derived by both adding and removing components other than interface nodes of an existing class. In this chapter, only class inheritance is considered in order to keep things simple. Incremental compilation of OOBNs where interface inheritance is allowed is left for future research work.

Also, note that once two nodes v_i and v_j have been joined by a referential edge, implying that they represent the same random variable. This association must be taken into account in any inference algorithm, including the one proposed.

Moreover, the edges within an iOOBN must be such that it "flattens out" [2] to a valid BN, that is, a DAG.



(a) Example iOOBN Class

(b) Example of flattened iOOBN Class

Figure 4.1: Example of flattening an OOBN class

Figure 4.1a shows an example of an iOOBN class having two input nodes (dashed ovals), namely A and S, two embedded objects (rectangles) and an output node N (double-lined oval). The embedded object on the left is an instance of the well-known Asia BN [23] where A and S are also the names of its input nodes, X and D are its output nodes, and T, L, B, and F are its embedded nodes. The other embedded object is an instance of another class with A and S also being its input nodes, Y and G being its output nodes, and C, M, and R being its embedded nodes. This example iOOBN class flattens into the BN given in Figure 4.1b.

[2] This is the process where Hugin converts the OOBN into the underlying BN, as per its API function "Creating run-time Domain" [27].

DEFINITION 4.2 : CLIQUE GRAPH

A **Clique Graph** CG = {V, E } of an iOOBN class C is a weighted undirected graph where V is a set of clique nodes, V = {Clq_1, Clq_2, ..., Clq_n}, with each clique containing nodes of C, and E is a set of edges, where each edge is a connection between a pair of clique nodes Clq_i and Clq_j and the weight of the connection is $|Clq_i \bigcap Clq_j| \geq 1$. The set $Clq_i \bigcap Clq_j$ is called the **separator set**. Also associated with each clique node is a probability potential, which is a function of the variables in the clique and the product of all the potentials is the joint probability of all the variables in the clique graph.

DEFINITION 4.3 : JUNCTION TREE

A graph JT is a **junction tree** if it is a clique graph that (i) is a tree and (ii) it has the running intersection property: for any pair of cliques, Clq_i, $Clq_j \in JT$, all the cliques in the path between Clq_i and Clq_j in the tree must contain $Clq_i \bigcap Clq_j$

DEFINITION 4.4 : JUNCTION FOREST

A Junction Forest, JF, is a finite set of disjoint junction trees.

4.2.1 Inference techniques

Inference (see Section 2.7 of Chapter 2), the most important purpose behind BN construction, has been extensively studied and explored in numerous pieces of research such as [24, 206–211]. One of the very widely used techniques of BN inference is "JT-based inference" [205]. The main steps of this method, as depicted in Figure 4.2 (A), are (i) moralization (making the DAG undirected and marrying/connecting the parents of each node), (ii) triangulation (adding fill-in edges to form triangulation and cliques), (iii) clique graph formation (making a graph of the cliques found from the previous step where nodes are cliques and any two nodes are connected by an edge if there are common items between them with weight equal to the number of common items), (iv) formation of JT/ Junction Forest (finding the Maximum spanning tree of the clique graph, where cost function works on the weight of the edge) and (v) message passing (collect and distribute) to propagate joint probabilities and evidence.

Figure 4.2 (B) depicts the steps of Incremental Compilation (InC) proposed by Flores et al. [4]. This approach was motivated by the fact that all the operations in JT-based inference, especially triangulation and clique finding, are computationally expensive [26]. InC is an MPS (Maximal Prime Subgraph) decomposition-based compilation technique [136] where any modification to the BN does not require performing the steps mentioned in Figure 4.2 nor constructing the JT from scratch. Instead, it constructs an MPS tree in parallel with JT con-

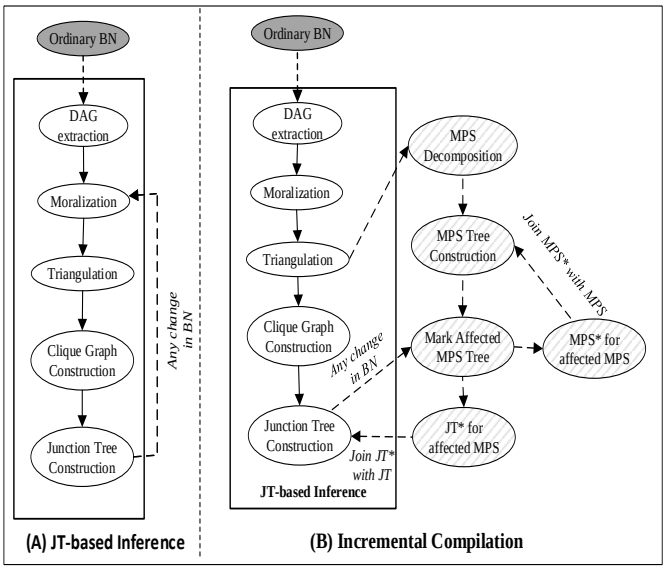


Figure 4.2: Process-flow diagram of (A) JT-based and (B) Incremental BN Compilation [4]

struction during ordinary BN compilation. It keeps track of the changes made in the last BN structure and marks the affected parts of the MPS tree and the JT. Then the marked portion is re-triangulated, and an intermediate JT for only the affected portion is constructed, which finally replaces the marked portion of the original JT. The resultant new JT presents the modified BN. The InC method is particularly useful when the modification to the BN is minor and local, with expensive operations avoided for parts of the networks that are unchanged.

A potential solution to avoid the cost of repeated JT construction that utilises the InC was proposed by Bangsø [5], with an implemented version described by Merten [6]. As depicted in Figure 4.3 (A), changes in the OOBN classes are transformed into a series of equivalent changes to the corresponding flattened BN, and then InC is applied. There are also some situations where, using this method, a large portion of the BN needs to be re-triangulated (because InC re-triangulates affected portions of the MPS tree); an example of this scenario is given in more detail in Section 4.3.4.

For OOBNs, to the best of our knowledge, no inference algorithm has been developed that works on the OOBN structure itself. In Hugin [27], an OOBN is flattened into an ordinary BN, and any traditional exact or approximate inference technique may be applied (see Figure 4.3 (A)). Any change, no matter how minor it is, to the OOBN structure generates full re-compilation, starting from flattening the new OOBN.

If an OOBN framework supports inheritance, such as iOOBN [140] (also described in Chapter 3), where any OOBN class can be derived from another class, then any change in the hierarchy generates a series of changes to all the subclasses below it in the inheritance hierarchy, so incremental compilation becomes even more important.

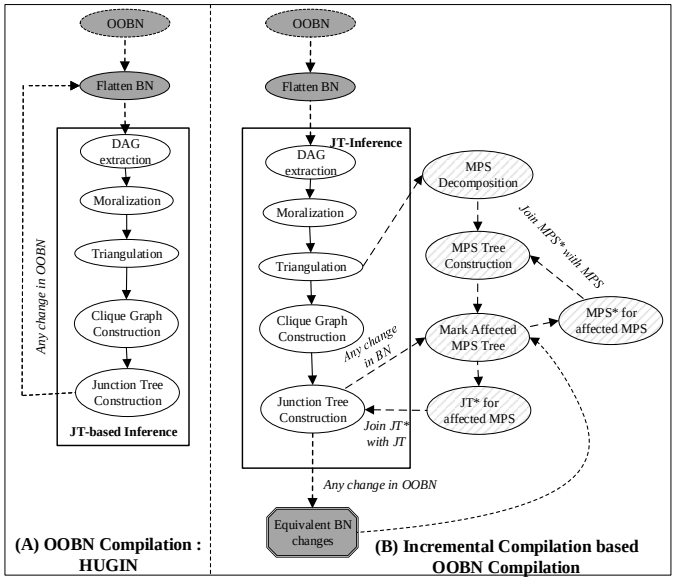


Figure 4.3: Process-flow diagram of (A) HUGIN OOBN Compilation and (B) Incremental Compilation-based OOBN compilation [5, 6]

4.3 Shareable Inheritable Incremental Compilation (SIIC) Algorithm

4.3.1 The proposed algorithm: SIIC

In this section, the proposed "Shareable Inheritable Incremental Compilation" (SII compilation) algorithm is presented; it constructs the junction tree (JT) for an iOOBN, by reusing previously constructed JTs (of embedded objects and superclasses) without flattening it into an ordinary BN. The algorithm can significantly reduce the amount of recompilation required when a class in an iOOBN class hierarchy is modified.

At a high-level, the SII compilation algorithm (see Algorithm 4.1 and Figure 4.4) takes an iOOBN class C as input, performs some preprocessing, retrieves any previously constructed JTs for its superclass (if there is one) and for any of its embedded objects, constructs any new JTs required, produces a Junction Forest (JF), then connects the junction trees in the JF to form a single resultant JT for C. Below, the algorithm is described in more detail.

The algorithm has four main stages:

1. Preprocessing (Line 2 Algorithm 4.1)

2. Creating a JF, including previously compiled JTs (Line 3, Algorithm 4.1) and the JTs constructed by any JT construction algorithm using the remaining class components after removing inherited (superclass) components and embedded objects; then joining the trees in the forest using referential edges (line 4).

ALGORITHM 4.1 (SII COMPILATION ALGORITHM)

Call : $SIIC(C) \rightarrow JT_{new}$
Input: C: an iOOBN class
(a set of interfaces, embedded standard and instance nodes)
Output: JT_{new}: A junction tree

```
1  begin
2      C ← Preprocessing(C)     /* Adding pseudo Ref. edges */
3      < JF, RE > ← CreateJunctionForest(C)
4      Edges ← ConnectJunctionTrees(JF, RE)
5      JT_new ← φ
6      foreach edge E ∈ Edges do
7          {Clq_i, Clq_j} ← getTerminalCliques(E)
8          JT_i ← JunctionTree(Clq_i)
9          JT_j ← JunctionTree(Clq_j)
10         if JT_i == JT_j then
11             /* connection between two cliques of same JT */
12             JT_new ← AddByMaintainingJTProperty(JT_i, Clq_i, Clq_j)
13         else
14             JT_new ← JoinJTPair(JT_i, JT_j, Clq_i, Clq_j)
15         JT_new ← PostPruning(JT_new)
16     // Assuming minimum clique size = 3
       JT_new ← Thinning(JT_new, 3)
17     JT_new ← PostPruning(JT_new)
18     return JT_new
```

3. Construction of the JT (Lines 5–15, Algorithm 4.1), and

4. Post-processing (Lines 16–17, Algorithm 4.1).

In **Preprocessing stage**, for each standard edge from an output node in an embedded object to an embedded node in the encapsulating class, a copy of the output node is introduced together with a referential edge between the original output node and its copy (see Figure 4.5c). These referential edges are referred to as *pseudo-referential edges*, as in practice these edges do not ever have to exist, and this step is only added to simplify the explanation of the algorithm.

Another vital operation in this stage is **renaming**. There are two types of renaming.

1. **Embedded node renaming:** the embedded nodes of different objects, those having the same name, have to be renamed. In fact this situation is handled easily by adding the

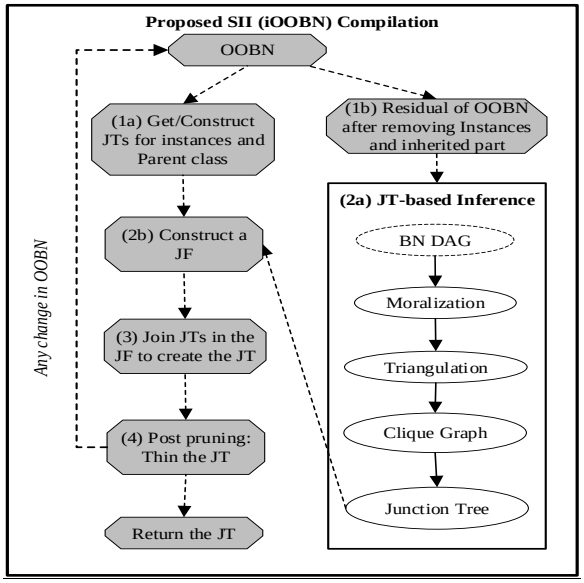


Figure 4.4: Process-flow diagram of iOOBN compilation using proposed SII compilation

object name as a prefix to the node name. In this chapter, a different name is used for simplicity and illustration purpose. The renaming operation is straightforward and just needs $O(n)$ times adding a prefix to the variable name where n is the number of nodes in a class.

2. **Renaming of terminal nodes of a referential edge:** In OOBN and in iOOBN, two nodes connected via a referential (and pseudo referential) edge are considered as the same node. Hence, giving them a single name (if they are named differently within different embedded objects) is important for the JT construction algorithm to be correct.

The **JF creation stage** requires the recursive generation of the JF (via Algorithm 4.2). First, the JF is retrieved (if previously compiled) or recursively created from the superclass. Then the algorithm (Algorithm 4.2, Line 10) identifies the components (nodes and edges) of the class being compiled that were not present in the superclass. In the case of no superclass existing, the components will be all of them. The referential edges and pseudo-referential edges are then removed (Lines 11–12), as well as any embedded object. Then the remaining class components are used to create new JT(s) (Line 13). The creation of the new JT can be done using any JT-based inference algorithm since there are no embedded objects left in the class. The JT construction of the remaining portion of the class that has no embedded object is the base case of the recursive JT/JF creation process.

In the **Joining of JTs stage**, the junction trees are linked together into a single junction tree. The joining operation is performed for each of the referential edges and pseudo referential edges. At first, all referential and pseudo referential edges are converted to JT connections.

ALGORITHM 4.2 (CREATE JUNCTION FOREST)

Call : $CreateJunctionForest(C) \rightarrow <JF, RE>$
Input: C: an iOOBN class
Output: JF: A set of junction trees
RE: A set of referential edges

```
1  begin
2      C_sup ← superClass(C)
3      if C_sup exists then
4          if C_sup is Compiled then
5              JF ← getJunctionForest(C_sup)
6          else
7              JF ← CreateJunctionForest(C_sup)
8      /* if any class C is extended from C_sup with additional components in C then only the
         Junction Forest for C – C_sup needs to be formed */
9      /* Remove inherited components from C */
10     C ← removeComponents(C, C_sup)
11     RE ← popOutReferentialEdges(C)
12     Obj ← popOutInstanceNodes(C)
13     /*Traditional JTConstruction( ) returns a set of JTs using traditional JT-Based
         inference approach*/
       JF ← JF ⋃ TraditionalJTConstruction(C)
14     /*Due to the deletion of instance nodes and Referential edges, remaining C returns a
         Junction Forest with at least one Junction Tree*/
15     foreach instance O ∈ Obj do
16         /* Get/create the JF for the corresponding classes of the instances */
17         class_O ← getClass(O)
18         if class_O is Compiled then
19             /* a compiled class has its JT constructed */
20             JF ← JF ⋃ getJunctionForest(class_O)
21         else
22             JF ← JF ⋃ CreateJunctionForest(class_O)
23     return JF, RE
```

The connections are made between those cliques that contain the terminal nodes of the referential edges. Since a terminal node connected via a referential edge may reside in more than

one clique, the SIIC algorithm ensures that for each referential edge, a single JT connection is created. This conversion from edge to connection can be done more effectively to build a more compact JT. However, in this version of SIIC, the clique pair that have more items in common got the preference. In case of a tie, the cliques that have fewer such existing connections get preference.

While joining the JTs into a JF, one of the following two operations are performed for each of the JT connections.

- **Joining a pair of JTs** straightaway. If a JT connection exists between the two cliques of two different JTs, then a single JT is formed by joining both of the JTs. The joining is performed by adding a separator connection between the two cliques those were connected by the JT connection.

- Just **maintaining JT property** is enough. If a JT connection exists between the two cliques of the same JT, there is nothing to add here other than just adding some common items between the two cliques to all of the separators and cliques that reside in the path between the two cliques.

During this process, the newly formed clique graph (JT_{new}) is regularly pruned (via Algorithm 4.3), which involves removing unnecessary cliques and separators.

Finally, in the **Postprocessing stage**, some thinning is done (via Algorithm 4.4), which involves splitting large cliques up into a path of smaller connected cliques, using the information of the cliques that constructed the large cliques. This thinning step is a simple linear checking and removal of redundant fill-in edges similar to the recursive thinning proposed in [200]. Then a final pruning step, as taken in stage three of the algorithm (via Algorithm 4.3), is taken. The pruning step removes unnecessary cliques and separators from a JT. Note that a clique is referred to as an "unnecessary clique" if it is connected with the JT to a clique through a separator having the same item/node set as the clique (the former one) itself has. As an example, in Figure 4.7a, the clique "A" is connected with $JT1,4$ to clique "AT" via the separator "A". Hence, the clique "A" is unnecessary and can be absorbed in clique "AT". This absorption is performed in the postpruning step of the algorithm. In the Algorithm 4.3, a linear scanning is performed to check if a clique and any of its separators are the same in terms of the node group that makes the clique or separator. If they are same, then both of them are removed from the JT in a way so as not to violate JT properties. A point worth noting here is that the pruning takes place on the edge of the tree and the pruning operation is "loss-less". Generally, the "running intersection property" is affected only if a clique or a separator in the middle of a branch of a tree is changed. Therefore, the JT property holds after the pruning if it was held before pruning.

ALGORITHM 4.3 (POSTPRUNING)

```
1  /*It prunes unnecessary cliques from a clique Tree to ensure its JT properties*/
   Call : PostPruning(JT) → JF
   Input: JT: a Junction Tree
   Output: JT: a pruned Junction Tree
2  begin
3      foreach Clique Clq in JT do
4          NC ← NeighbourClique(Clq)
5          foreach Clique N in NC do
6              if Clq == S then
7                  S ← N ∩ Clq
8                  /* S is the separator between N and Clq */
9                  JT.Clq ← JT.Clq − Clq
10                 JT.Sep ← JT.Sep − S
11                 RNC ← NC − N
12                 foreach Clique K in RNC do
13                     Sep ← K ∩ N
14                     connect(K, Sep, N)
15     return JT
```

Theorem 1. *Proposed SII compilation (Algorithm 4.1) generates junction tree.*

Proof:

When creating the JF (Algorithm 4.2), assuming any precompiled JFs retrieved are valid, and any new JT (line 15–22) is constructed using a standard valid JT construction method, **CreateJunctionForest()** returns a valid JF. Also assuming, there is no variable names in common between any two JTs in a JF. There are two cases to consider when creating the new JT from the JF as discussed earlier in the **Joining of JTs** stage of the SIIC algorithm.

1. Let Clq_1 be a clique in JT_1 containing V_1 and Clq_2 be a clique in JT_2 containing V_2. Replace V_2 in JT_2 by V_1, and denote this tree by JT_2^*. Then connect Clq_1 to Clq_2 with an edge to form the clique graph G. Since G is a clique graph constructed by joining two trees by a single edge, it follows that it is a clique tree. Now to prove that G has a Running Intersection Property, take any cliques Clq_1^* and Clq_2^* in G and let X $\in Clq_1^* \cap Clq_2^*$. If Clq_1^* and Clq_2^* belong to the subgraph JT_1, then since JT_1 is a junction tree, it follows that X belongs to all cliques in the path between Clq_1^* and Clq_2^*. If Clq_1^* and Clq_2^* belong to subgraph JT_2^*, then since JT_2^* is a junction tree, it follows that X belongs to all

ALGORITHM 4.4 (THINNING)

```
1  /*It performs a specialised operation, thinning, on the JT*/
   Call : Thinning(JT, τ) → JT
   Input: JT: a Junction Tree,
          τ: Clique size threshold
   Output: JT: a thinned Junction Tree
2  begin
3      CS ← stack of cliques in JT of size ≥ τ
4      while CS is not empty do
5          C ← CS.pop()
6          NC ← NeighbourCliques(C)
7          AOC ← AssociatedOriginalCliques(C)
8          JT_is_thinner ← True
9          foreach Partition P of AOC into |NC| parts do
10             NewCliques ← φ
11             foreach Part of Cliques of P do
12                 NP ← MergeCliques(Cliques)
13                 if |NP| == |C| then
14                     JT_is_thinner ← False
15                     break
16                 NewCliques ← NewCliques ⋃ NP
17             if JT_is_thinner then
18                 JT ← Replace(JT, C, NewCliques)
19                 push any clique in NewCliques of size ≥ τ onto CS
20                 break
21     return JT
```

cliques in the path between Clq_1^* and Clq_2^*. The last possibility to consider is Clq_1^* is in JT_1 and Clq_2^* is in JT_2^*. Since JT_1 and JT_2 have no variables in common (actually, the algorithm ensures that through the renaming of embedded nodes), $Clq_1^* \cap Clq_2^* = \varnothing$ or $\{V_1\}$. In case, $Clq_1^* \cap Clq_2^* = \varnothing$, G follows the running intersection property. In the case of X = V_1, the path between Clq_1^* and Clq_2^* must contain V_1, since JT_1 and JT_2^* are junction trees, and Clq_1 and Clq_2 contain V_1. Therefore G has the running intersection property and hence is a junction tree.

2. Let JT be a junction tree, and Clq_1 be a clique in JT containing V_1 and Clq_2 be a clique in JT containing V_2. Also, let P be the path between Clq_1 and Clq_2 in JT. Replace V_2 in JT by

V_1. Now add V_1 to each clique in the path P to form the graph G. It is assumed that any clique contains at most one V_1 in G, so that any clique containing multiple V_1's has them replaced by a single V_1.

Since JT is a tree, it follows that G is a also clique tree. Also, since JT is a junction tree, it has the running intersection property. Moreover, by adding a variable to all the cliques in a path in JT, it still has the running intersection property. Hence G has the running intersection property and hence is a junction tree.

□

4.3.2 Constructing JTs using the SIIC algorithm: an example

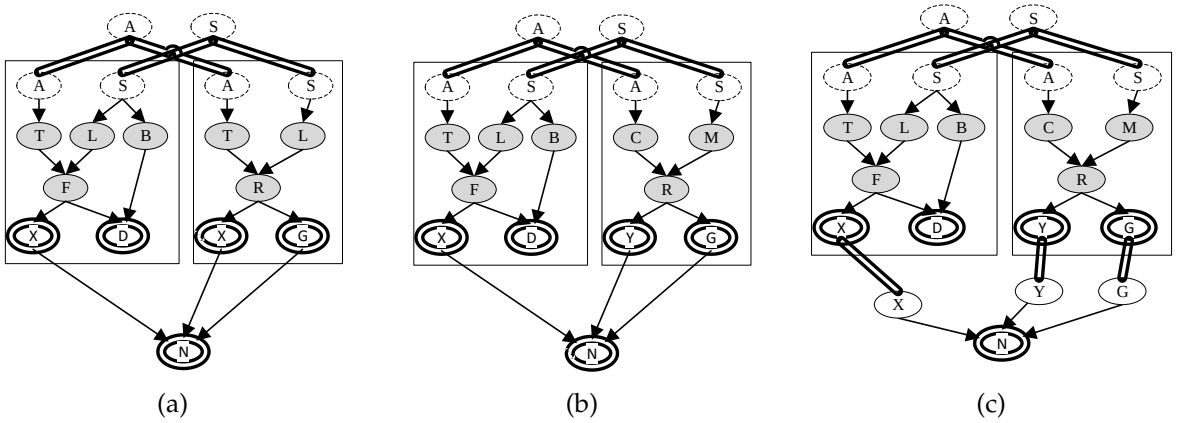
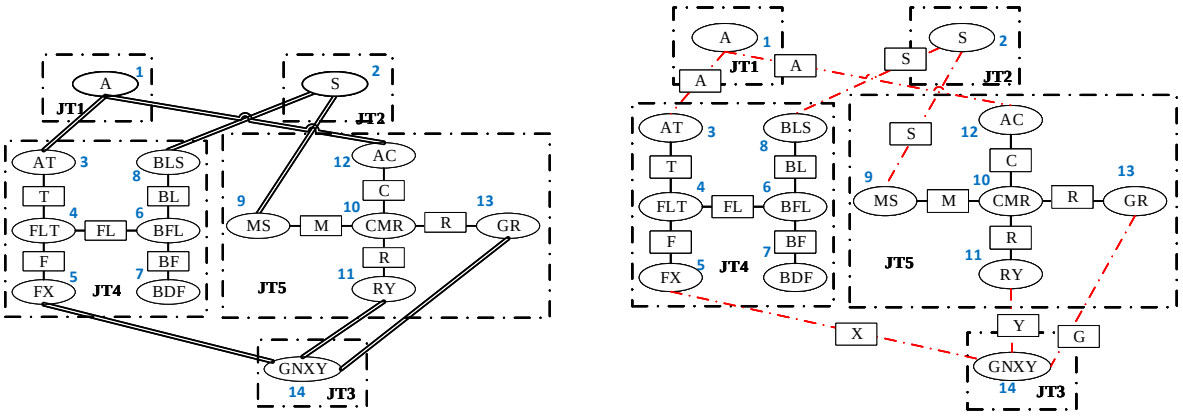


Figure 4.5: (a) Sample OOBN Class: C, Preprocessing C [Line-2]: (b) Duplicate node names resolved, and (c) Pseudo Ref. Edge adding

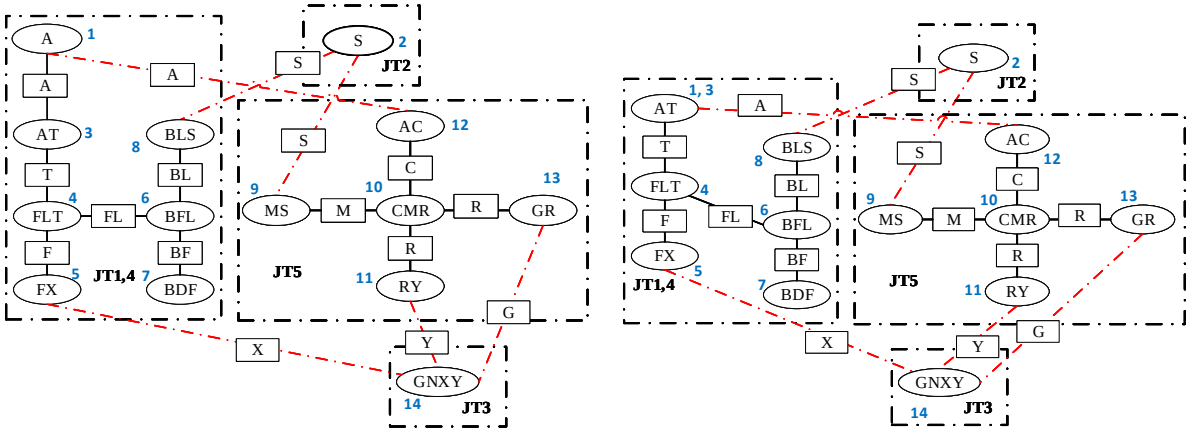
Figures 4.5 to Figure 4.12 show how Algorithm 4.1 constructs a JT for the same example OOBN from Figure 4.1a (repeated in Figure 4.5b for easy reference). Note that this example only shows reuse of the JT from the embedded object JTs, but does not show reuse of a JT from a superclass. Figure 4.5c shows the result after preprocessing where duplicate names have been changed following the approach discussed in Section 4.3 under the preprocessing step, and pseudo-referential links have been added to the copies of the embedded output nodes X, Y and G, which are parents of N. In Figure 4.6a, the JF has been formed using previously compiled JTs, JT_4 and JT_5, as well as the derived JTs, namely JT_1, JT_2 and JT_3, and connected into a JT via referential edges (shown with double lines): for example, JT_4 and JT_3 are connected via a referential edge between clique nodes FX and GNXY. Note that each clique is given a unique index, shown in blue. Figure 4.6b shows the networks after these referential edges have been converted to edges in the clique graph (red dashed lines), indicating connections between the JTs, with the separators also shown; for example, X is the separator on the red edge connecting JT_4 and JT_3. Figure 4.7a shows the result after joining JT_1 and JT_4. Next,

postpruning removes clique node 1 containing only variable A, shown in Figure 4.7b; note that clique AT is now labelled "1, 3", indicating it was created by the merger of cliques 1 (A) and 3 (AT). Figure 4.8a shows the result after joining $JT_{1,4}$ and JT_5; this step was straightforward, with the edge between AT and AC (with separator A) remaining and no other changes or post-processing required.



(a) (b)

Figure 4.6: (a) Constructing Junction Forest [Line-3], (b) Connecting JTs [Line-4]



(a) (b)

Figure 4.7: (a) Joining JT1 and JT4 [Line -14], (b) Postpruning JT1,4 [Line-16]

Similarly, Figure 4.8b shows the formation of $JT_{1,2,4,5}$. After this formation step, postpruning by merging clique 2 (S) with clique 8 (BLS) is performed as shown in Figure 4.9a. Next, Figure 4.9b shows the removal of the connection between BLS and MS by adding S (shown in blue) where required to cliques along another path between BLS and MS, in order to preserve the running intersection property (Figure 4.10a). Figure 4.10b shows the structure after joining all five original JTs in the JF, $JT_{1,2,3,4,5}$, while Figure 4.10c shows the result after removing the connection from RY to GNXY (postpruning step), which then required adding Y to two other cliques; at this point there is only one unresolved connection from the original referential

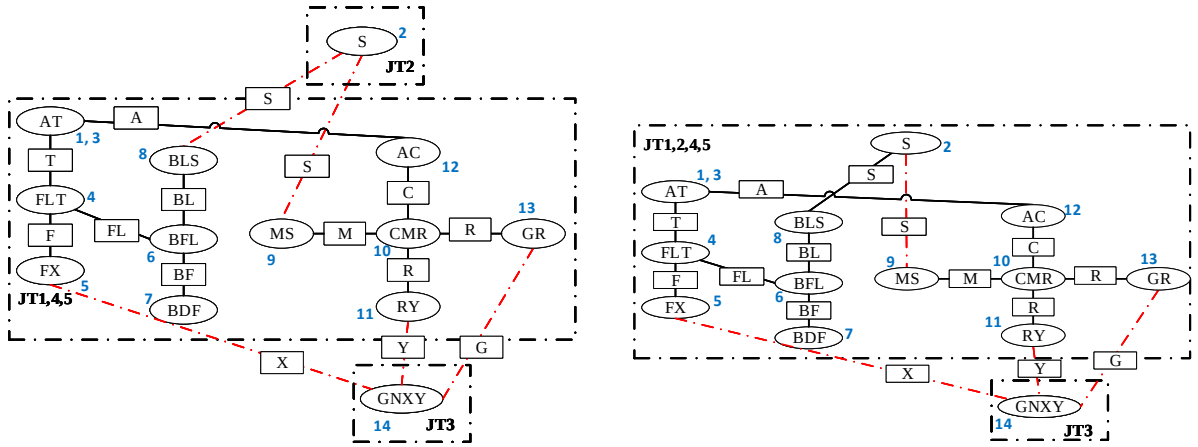


Figure 4.8: (a) Joining JT1,4 and JT5 [Line-14], (b) Joining JT1,4,5 and JT2 [Line-14]

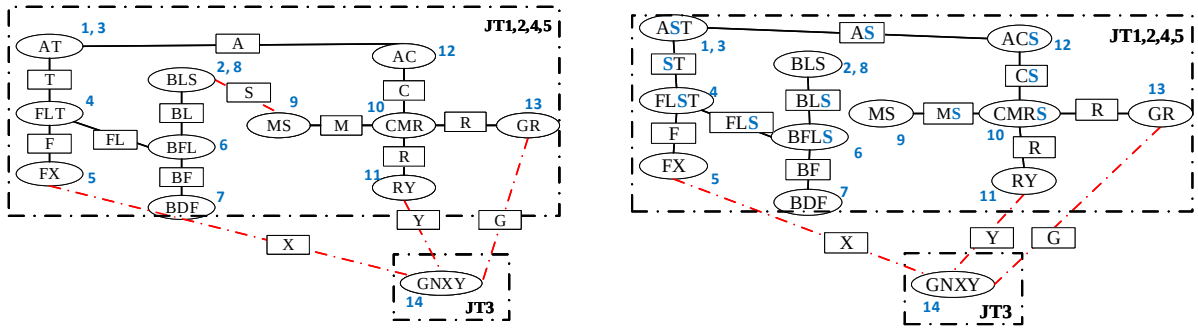


Figure 4.9: (a) Postpruning JT1,2,4,5 [Line-16], (b) Processing the edge having separator S between cliques of same JT1,2,4,5 [Line-12]

edges remaining (coloured red), that from FX to GNXY. Figure 4.11a shows the JT after merging cliques 4 and 5, and resolving that final edge, with the associated addition of X to cliques 13, 9–10–11, 1–3 and the new 4–5, and another postpruning.

The last remaining step is the thinning, which uses information about the original cliques that was stored throughout the JT combination steps (via the blue numbers associated with each clique). The thinning takes large cliques (using a clique size threshold) and iteratively splits them. For example, the CMRSXY clique is removed (Figure 4.11b), first replaced by CMSX and CMRXY, then by CMSX, CMRX, RXY and finally by CMSC and CMRX (after the final postpruning of the unnecessary RXY). Figure 4.11b shows the final resultant JT.

In order to compare the JT produced by SII compilation to Hugin's method (flattening and recompiling from scratch), the flattened ordinary BN for the example OOBN class C is shown in Figure 4.12b and the Hugin-generated JT in Figure 4.12c. It can be seen that, not surprisingly, there are some similarities, including the two cliques (GRXY and GNXY) at the right end of the SIIC generated JT (Figure 4.11b), with the clique at the left end (BDF) also being a leaf in the

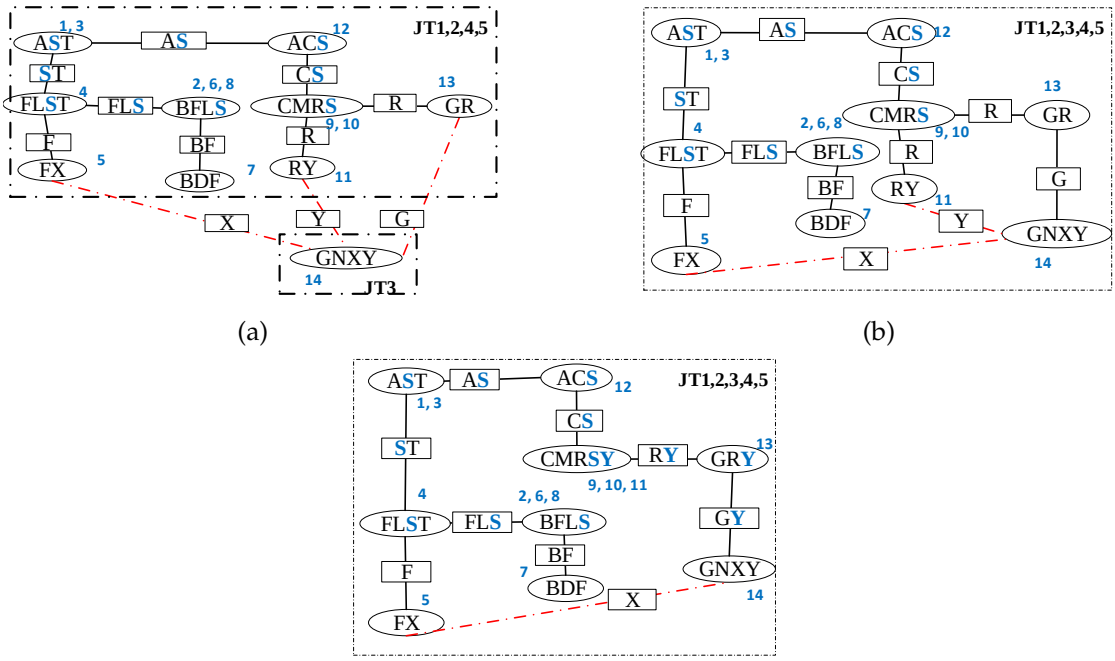


Figure 4.10: (a) Postpruning JT1,2,4,5 [Line-16], (b) Joining JT1,2,4,5 and JT3 [Line-14], (c) Processing the edge having separator Y between cliques of same JT1,2,3,4,5 [Line-12], Postpruning [Line-16]

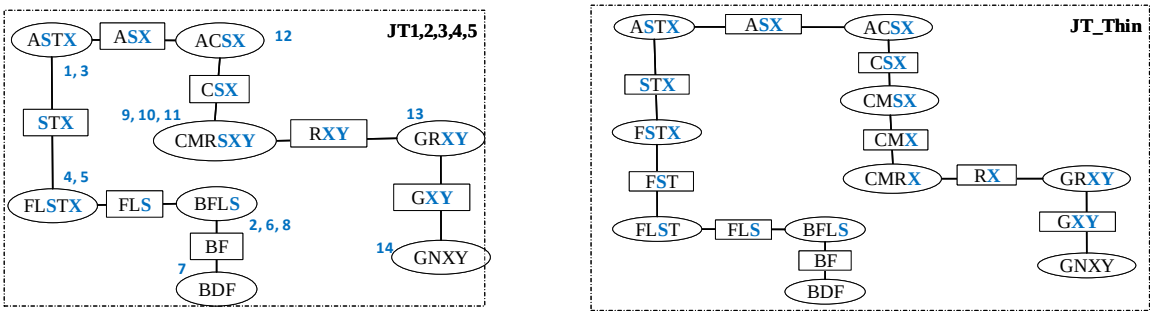


Figure 4.11: (a) Final JT of class C using SII compilation, after processing the last edge with separator X, (b) Thinning the JT (Removing redundant fill-in edges) [Line-17]

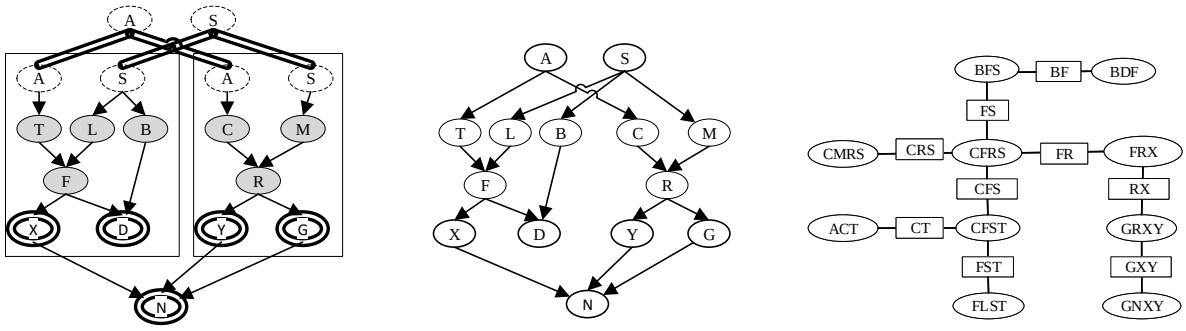


Figure 4.12: (a) Sample OOBN Class: C (no duplicate node names), (b) Flattened BN of OOBN class C (c) HUGIN junction tree of flattened BN.

Hugin JT. Overall, the JTs are very different, with the SII compilation JT tending to have larger cliques; this feature is further discussed in the following subsection.

4.3.3 Advantages of the proposed compilation approach

In brief, the prime advantages of the compilation technique are as follows:

1. No flattening is required for an iOOBN/OOBN class, no matter how deep the level of embedding.
2. It can reuse the compilation outcome, i.e., JTs, produced previously for a same (unedited) BN or OOBN/iOOBN class.
3. The technique allows sharing of partial compilation outcome of an OOBN/iOOBN class through reuse of the JTs of embedded objects.
4. The compilation outcome of a parent iOOBN class can be used/reused/shared to create JTs of its child classes.
5. Any modification in a compiled class, i.e., a class having a JT created, does not require recreating the JT from scratch. Only newly added nodes and edges can be added in a form of a small JT added to the previous JT. Note that the current formulation of the algorithm has no direct way of dealing with incrementally adjusting the JT in case of deletion of nodes and edges.
6. No complex and time-consuming operations required, no matter whether a fresh JT is created or an existing JT is adjusted.

4.3.4 Efficiency

The efficiency of the proposed compilation technique can be evaluated from two perspectives: it reduces JT compilation computation time, and makes the resultant JT more efficient for subsequent inference.

4.3.4.1 JT construction complexity

SII compilation allows reusing the JT of its superclass if it exists, and class inheritance is used where deriving a subclass from a another class (e.g., a superclass) does not involve the removal of any of the superclass nodes or edges. It also allows the reuse of, if there exist, previously compiled JTs from embedded objects. In the base case of the recursion of the algorithm, new

JTs are compiled on an ordinary BN using any JT compilation method. However, it is not possible to give performance guarantees as to the reduction of compilation computation because that depends on the specific structure of the class being compiled and the structures of any embedded objects. Naturally, in an extreme case, if there are no superclasses or embedded classes (i.e., if it is not an iOOBN), SII compilation incurs computation overheads without any reduction in compilation/computation time. An empirical investigation (see Section 4.5) is performed of the computational savings that can be achieved in practice, over a range of real and synthetic OOBNs.

The widely used traditional flattening approach always required starting from scratch and does not allow reuse of existing outcomes. Its primary computation involves flattening of the OOBN as well as the standard triangulation and JT construction costs. Though flattening is a linear-time operation, with hierarchies of embedded classes, it may introduce significant complexities to the computation [26]. The JT construction cost is polynomial, according to the number of cliques in the clique graph, incurring the same cost as Prim's Minimum Spanning Tree construction. Minimal triangulation is an NP-Hard problem because such heuristic-based suboptimal triangulation requires polynomial-time depending on the number of variables in the BN.

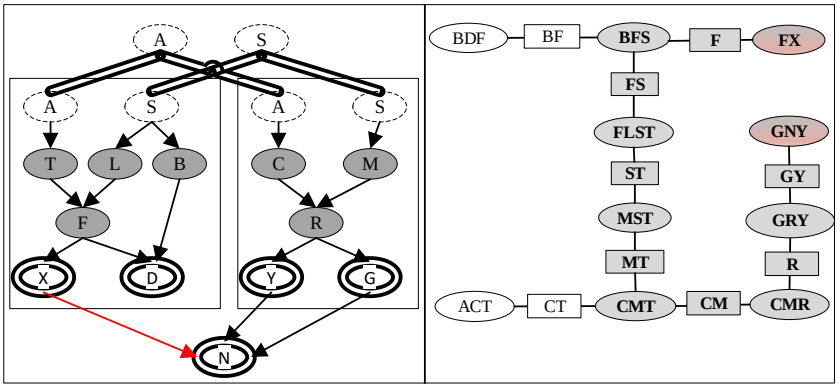


Figure 4.13: Example of the limitation of Incremental Compilation (InC).

As described in Section 4.2.1, the incremental compilation (InC) approach allows reusing existing compiled structures. However, it still retriangulates a portion of the existing structure as well as incurring extra computational and storage burdens for the MPSD structure maintained in parallel with the JT. It has also been observed that it may encounter scenarios where the whole network structure, or a large portion of the structure, needs to be re-triangulated and that this leads to similar redundant computational steps as the traditional flattening based approach. For example, suppose InC is being used to add an edge $X \rightarrow N$ to the network shown in Figure 4.13. Figure 4.13 also contains the JT for the initial network with the affected and

marked JT segment due to the addition of the edge. The incremental compilation approach affects eight of the ten cliques of the JT, which means that almost the whole network needs to be re-triangulated and the modified portion needs to be joined, in order to obtain the resultant structure.

4.3.4.2 The resultant JT

The structure of the resultant JT has implications for the subsequent inference computation time. A metric to roughly measure this time based on a message passing inference algorithm is the so-called JT cost [212] [1, p 74]:

$$JT_{cost} = \sum_{C_i \in \{C_1,\ldots,C_n\}} \left(K_i \prod_{X \in C_i} |\Omega_X| \right)$$

where $C_1, \ldots, C_n$ represent the cliques in the JT, K_i denotes the sum of the number of parent and child cliques of C_i in JT (reflecting the Arity of the JT), and Ω_X is the state-space of node X in clique C_i.

The JT cost, therefore, depends on the structure of the JT, as well as the size of the state spaces of the nodes of the OOBN class. For the example OOBN C, if all the nodes are binary, the cost of the JT produced by Hugin's traditional flattening JT-based approach (shown in Figure 4.12c) is 8096. In comparison, the cost of the JT produced by SII compilation prethinning (Figure 4.11a) is 13312, which after thinning (Figure 4.11b) reduces to 5376 (significantly smaller than the Hugin-generated JT).

One reason behind this observed efficiency, at least in this example, is that the JT produced by SII compilation in this case, is a binary JT (a JT with no clique node with more than three connections). In [213, 214] Shenoy et al. proved that a binary JT is more efficient than a non-binary JT. However, any theoretical result about the arity of the resultant JT has not yet been established. This remains an area for further research.

4.4 Complexity Analysis

To compile an OOBN/iOOBN class $C = < \mathbb{N}, \mathbb{E}, \Pi, \mathbb{O} >$, Hugin flattens the class into an ordinary BN.

4.4.1 Hugin JT construction Complexity analysis

The JT construction includes moralization, triangulation, and then construction of JT from clique graphs.

Moralization is a polynomial-time operation which depends on the number of nodes (variables); constructing JT is also polynomial according to the number of cliques. An underlying cost of identifying cliques to construct the clique graph is also polynomial to the number of variables. The most significant and crucial operation is the triangulation which has a suboptimal solution of polynomial-time. The optimum solution is known to be NP-complete.

The cost involved in the JT construction algorithm used by Hugin is as follows:

1. **Moralization**: $O(|V| + |E|)$
2. **Triangulation**: $O(|V|^2)$
3. **Identifying cliques**: $O(3^{|V|/3})$
4. **Clique graph construction**: $O(q^2)$
5. **JT construction**: $O(q^2 log q)$

where q is the number of cliques in JT.

If class C is compiled using the traditional approach assuming each object in C has L levels of recursive embedded objects in it connected through r number of referential edges, the flattening cost is $O(p \times (r \times e)^L)$. Here $p = |O|$ is the number of objects in C, and $e = |E| - r$ is the number of causal edges going out from the input nodes those connected with external nodes by the referential edges.

4.4.2 SIIC JT construction complexity analysis

The proposed SIIC considers that the nodes in class C are clustered/grouped into external embedded nodes (i.e., $\mathbb{N}$), and the nodes embedded in the other classes are named as "foreign" classes in the proposed approach: i.e., those that instantiate various embedded objects in O. For $p = |O|$ number of embedded objects in C, there are $p + 1$ number of clusters (assuming the nodes in $\mathbb{N}$ create a single JT instead of a JF, for simplicity).

JTs are constructed for each of the clusters that correspond to each of the embedded objects. If the instantiating classes of the embedded objects have been previously compiled, the already constructed JTs are used for those classes. Then a JT is constructed using the embedded nodes $\mathbb{N}$. Finally, the JTs are joined to produce the JT of class C.

Assuming each cluster of nodes p_i has a JT with q_i number of cliques, then the SIIC JT construction cost involves the following components:

1. **Preprocessing**: (Adding pseudo-referential links) $O(\sum_{i=1}^{p} |N_{out}|)$, where N_{out} is the set of output nodes of O_i.

2. **Partition-based JT construction**: $O(\sum_{i=1}^{p+1} q_i^2 log q_i)$
3. **JT joining**: To join any two JTs with q_1 and q_2 number of cliques, Joining cost is $O(q_1^2 + q_1 q_2)$. Therefore, the joining cost of a set of clusters $\{q_1, ..., q_{p+1}\}$, is $O(\sum_{i=1, j=2}^{j \leq p+1} q_i^2 + q_i q_j)$
4. **Postprocessing**: In order to remove unwanted redundant cliques, the required cost is $O(\sum_{i=1}^{p+1} q_i)$

4.4.3 JT construction (asymptotic) costs of Hugin compared with SIIC

In order to compare the complexity of both algorithms, a neutral platform is needed because both have their own scope and limitations.

Assuming class C has only one embedded object which has no embedded object in it, the DAG associated with the flattened BN of C is $G = < V, E >$. SIIC assumes $V = V_1 \uplus [3] V_2$ and $E = E_1 \uplus [3] E_2$, where V_1 is the set of embedded chance nodes in C and V_2 is the set of nodes embedded in the only object of C. This means that there are two clusters from where q_1 and q_2 number of cliques can be constructed. The following table (Table 4.1) depicts the underlying step-wise comparison of SIIC and Hugin JT construction algorithms.

Table 4.1: Asymptotic analysis of the algorithms

Steps involved	**Hugin JT Complexity**	**SIIC Complexity**
Flattening	$O(p \times (r \times e)^L) = O(E_2)$ where, $L = 1$, $\|O\| = 1$	×
Preprocessing	×	$O(\sum_{i=1}^{p} \|N_{out}\|) = O(\|V_2\|)$
Moralization	$O(V + E)$	$O(V_1 + E_1) + O(V_2 + E_2) = O(V + E)$
Triangulation	$O(V^2)$	$O(V_1^2 + V_2^2)$
Identifying cliques	$O(3^{V/3})$	$O(3^{V_1/3})$
Clique Graph	$O(q^2)$	$O(q_1^2)$
JT Construction	$O(q^2 log q)$	$O(q_1^2 log q_1 + q_2^2 log q_2 + q_1^2 + q_1 q_2 + q_1 + q_2)$ $= O(q^2 log q)$, [where $q_1 = q_2 = q$]

For a class with only a single and simple (no recursive embedding) object, SIIC is superior to Hugin. In real-life modelling, classes that contain multiple objects and multiple levels of embedding are generally more complex. The embedding makes flattening, triangulation, clique identifying, clique graph-building and JT construction much more expensive.

[3] n sets $A_1, A_2, \ldots A_n$ are mutually disjoint if $A_i \cap A_j = \varnothing$ for all i, j where $1 \leq i, j \leq n$ and $i \neq j$ and denoted as $A_1 \uplus A_2 \uplus \cdots \uplus A_n$

4.5 The Experimental Settings

To analyse and compare the performance of the proposed incremental algorithm, Hugin (a widely used commercial software for OOBN), has been used. The proposed approach has two versions, SIIC and SIIC# (where SIIC reuses the existing JTs of embedded objects and superclasses if there are any, rather than constructing them from scratch). The compilation time of professionally developed Hugin and the proposed SIIC are computed with a PC (Intel(R) Core(TM) i5-8259U CPU, 8GB RAM). The analysis was performed on a synthetic repository of OOBN classes. The approach for generating OOBNs is outlined in Algorithm 4.5 below.

ALGORITHM 4.5 (GENERATING A RANDOM OOBN)

Call : generateRandomOOBN(NOC, NON, NOP, NOS) $\rightarrow \mathbb{C}_{main}$

Input: NOC: Number of instantiating additional classes,
NON: Number of nodes per class,
NOO: Number of objects per class instantiating class,
NOP: Number of maximum parents per node,
NOS: Number of States per node

Output: $\mathbb{C}_{main}$: An OOBN

```
1  begin
2      LIST_obj ← ∅
3      for i : 1 to NOC do
4          bn ← GenerateRandomBN(NON, NOP, NOS) // using the algorithm of [215]
5          C' ← ConvertBNToOOBN(bn) // Using the algorithm 5.3 of Chapter 5
6          for j : 1 to NOO do
7              LIST_obj ← LIST_obj ∪ C'
8      BN_main ← GenerateRandomBN(NON, NOP, NOS) //using the algorithm of [215]
9      C ← ConvertBNToOOBN(BN_main) // Using the algorithm 5.3 of Chapter 5
10     ℂ_main ← GenerateMainOOBN(C, LIST_obj)
11     return ℂ_main
```

4.5.1 Complexity measures

In [216], Nicholson et al. proposed an equation of computing the complexity of BNs as follows.

$$comp(BN) = \sum_{n \in N} (|\Omega_n| - 1) \left(\prod_{pa \in par(n)} |\Omega_{pa}| \right)$$

where,

- N is the set of nodes in BN
- Ω_n is the set of parameters for a node n
- par(n) is the set of parent nodes of the node n

The above-mentioned "Nicholson" measure is a noncontroversial measure of estimating the complexity required to calculate the posterior in a BN. This measure represents the number of parameters or the size of the CPTs of a BN model to perform inference in it. The above complexity measure is generalized for OOBNs. For an OOBN class C, the complexity-deriving equation can be defined as follows.

$$comp(C) = \sum_{O_i \in \mathbb{O}}^{O_i = C_j^I} comp(C_j) + \sum_{n \in \mathbb{N}} (|\Omega_n| - 1) \left(\prod_{pa \in par(n)} |\Omega_{pa}| \right)$$

where,

- **N** is the set of nodes in C
- **O** is the set of objects in C
- $O_i = C_j^I$ means O_i is an object instantiated by class C_j; in the other words, O_i is a replica of the class C_j
- Ω_n is the set of parameters for a node n
- par(n) is the set of parent nodes of the node n

4.5.2 Parameters

The performance of the proposed algorithm was compared over a range of different OOBNs that were generated using the parameter values given in Table 4.2.

The parameters are varied to some extent so as to make the OOBNs realistic, by choosing parameter values of small, medium and large. In order to justify this point, the larger OOBNs were chosen big enough that Hugin would fail to compile in some cases, leading to the Java run-time error "Memory limit exceeded". As an example, a main class comprised 50 nodes, where each node had five states, an average of five parents, and into which three additional classes were instantiated three times (that means nine instances in total). Each of these classes had the same configuration as the main class.

Using the parameter and their values (as shown in Table 4.2, columns 1 and 2), 1456 ($7 \times 4 \times 4 + 7 \times 4 \times 4 \times 3 \times 4$) different parameter configurations are generated and for each configuration five OOBN classes were generated to perform a 5-fold analysis, thus constituting in total 1456

Table 4.2: Parameters and terms used in the experimentation

Parameters	Terms	Full form	Ranges (Values)
Num. **o**f **N**odes	NON	Number of Nodes per class (main and foreign)	5, 10, 15, 20, 25, 30, 50
Num. **o**f **S**tates	NOS	Number of States per Node in a class	2, 3, 4, 5
Num. **o**f **P**arents	NOP	Max. Number of Parents per Node in a class	2, 3, 4, 5
Num. **o**f Foreign **C**lasses	NOC	Number of (foreign) classes used for instantiating objects	0, 1, 2, 3
Num. **o**f **Objects**	NOO	Number of Objects instantiated from each foreign class	1, 2, 3, 4
Num. **o**f **P**arents (**A**verage)	NOPAvg	Average Number of Parents per node in a class	
JT **Exists?**		If the JTs of the classes preexist in the storage for reuse instead of building the JTs	Yes (SIIC#), No (SIIC)
Num. of **Folds** per Class		How many OOBN classes are built for each configuration	5
Run **Repeats** per Class		How many times each algorithm is run on an OOBN class	4

× 5 = 7280 OOBNs. During the experimentation, an average of four runs for each algorithm was performed so as to reduce the fluctuation of the run time differences. Along with the run-time of the algorithms, the complexity of the flattened BN is computed (equivalent to the OOBN complexity presented in the equation of Section 4.5.1), Cost of JTs (as proposed by Kanazawa [212]) produced by the Hugin and SIIC (SIIC#) algorithm.

4.5.3 Generating synthetic (random) OOBNs

In order to run the experiments, random OOBNs were generated. Algorithm 4.5 represents the adopted approach for generating the OOBNs. The approach is also illustrated in Figure 4.14 as a flowchart that uses two modules (depicted in figures 4.16 and 4.15). It takes the following parameters, namely (i) configuration of an OOBN and (ii) number of folds (number of randomly generated OOBNs with the same configuration), as input. Configuration refers to the number of nodes per class, the maximum number of parents per node, the number of states per node, the number of instantiated objects per foreign class (classes to instantiate embedded objects in the main OOBN class referred to as "Foreign class" in this chapter) and the number of foreign OOBN classes. Using the algorithm proposed in [215], an ordinary BN is generated. The BN is converted to an OOBN using Algorithm 5.3 proposed in Chapter 5. This OOBN will be used as the basic skeleton of the intended OOBN named as "main.oobn". Then for each of the additional classes, one ordinary BN is generated using the same algorithm given in [215]. The BNs are converted to OOBNs using Algorithm 5.3. The OOBNs are copied "number of

objects per instantiating class" times to create objects as per the specification.

To keep things simple, in the current implementation of the generation of synthetic (random) OOBNs (Algorithm 4.5), the level of embedding was kept to 1; that means, there is no multi-level embedding of objects in an OOBN. Regardless, the next step is to connect the "main.oobn" class nodes to the OOBNs representing objects to be encapsulated in the main OOBN class (shown in Algorithm 4.6). In order to make the connections, the nodes in the main class are partitioned into "input partition" (potential nodes to be connected with input nodes of the objects) and "output partition" (potential nodes to be added to an incoming causal edge from the output nodes of the objects). The partitioning procedure is outlined in Algorithm 4.7. The partition is done carefully to avoid the potential threat of adding cycles in the OOBNs. The nodes of the main class are sorted in topological order, and a specific section of the sorted nodes is identified as the input partition and the remaining section as the output partition. The ratio of the number of input nodes of an object to the total number of nodes in the main class is taken into consideration to make the partitions even.

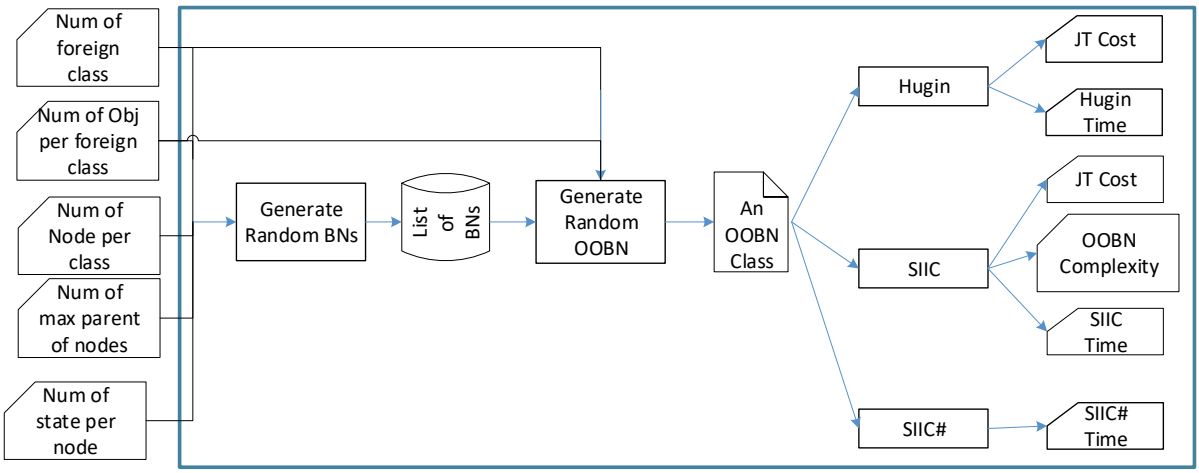


Figure 4.14: Schematic diagram of the experimental design

A node is selected from the input partition randomly for each of the input nodes of the objects, connecting them via a referential edge. Then a node from the output partition is randomly selected that follows the configuration of the OOBN given in the specification, so that the maximum number of parents holds. Then a causal edge is added from one of the output nodes of the objects to the selected output partition node. Adding such edges is done for each of the output nodes of the object. In the worst case scenario, if no such output partition node is left, in order to keep the creation of the OOBN simple, the constraint of the maximum number of parents per node could be relaxed, and the output node could be added as a node having a minimum number of parents. The output node could be left with no connections, and

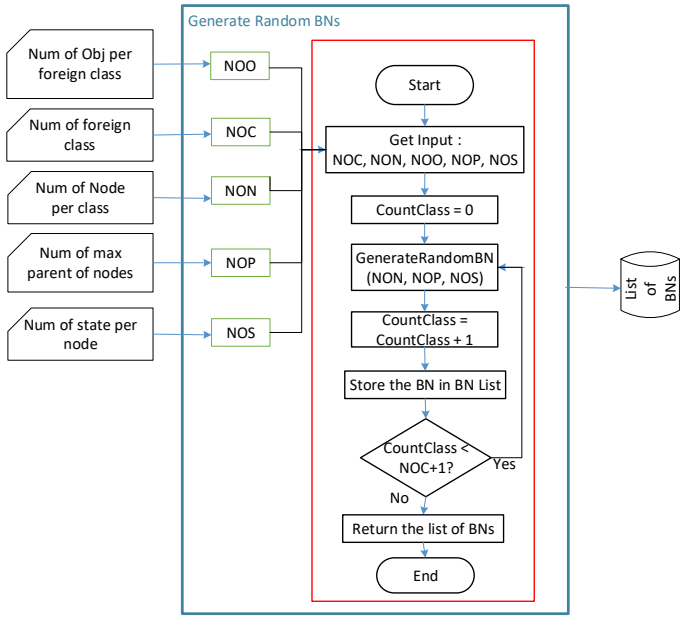


Figure 4.15: Flowchart to generate a synthetic (random) BN

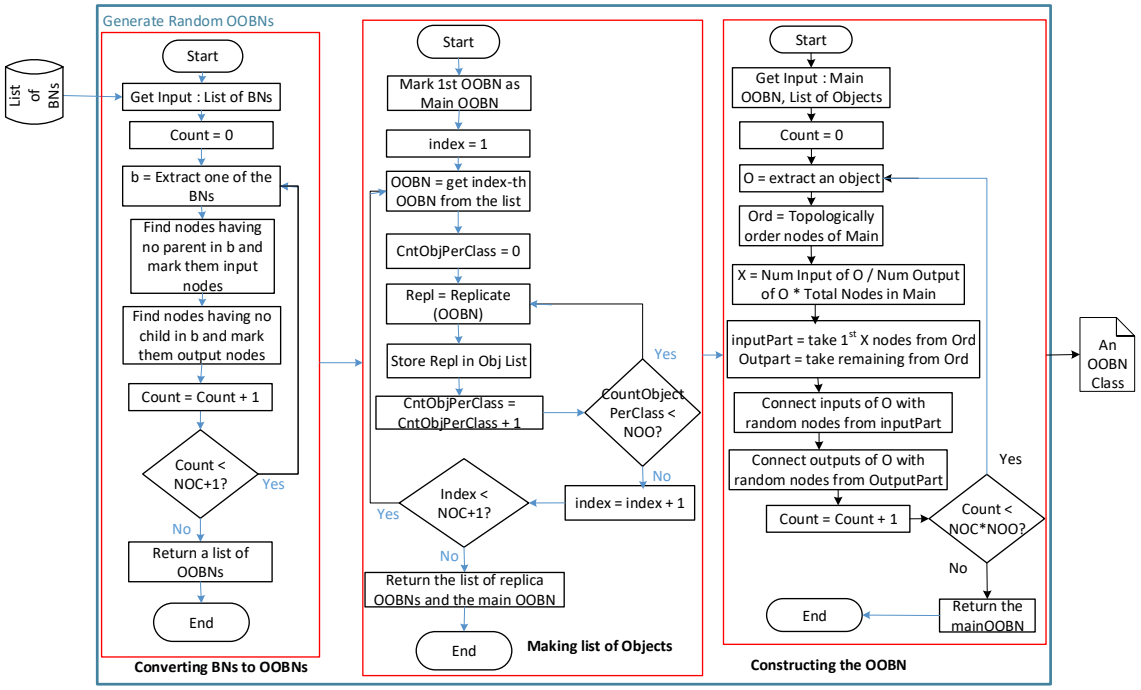
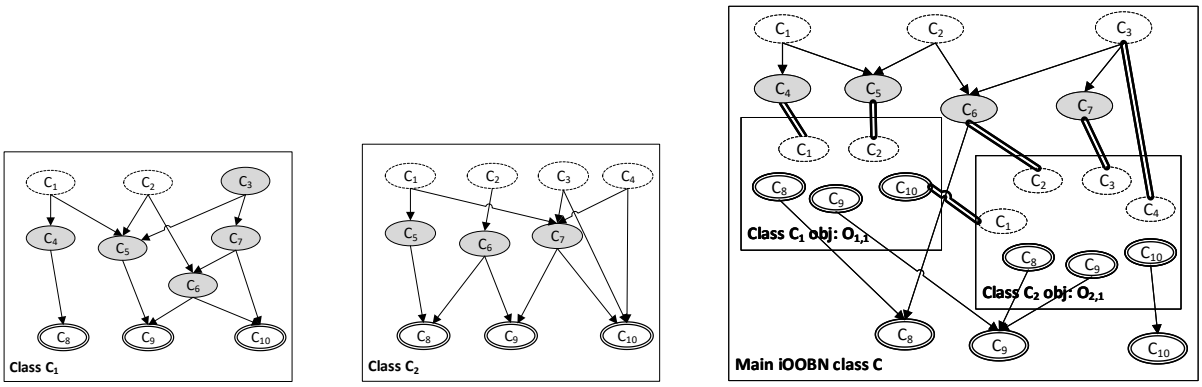


Figure 4.16: Flowchart to generate a synthetic (random) OOBN class

that would still produce a valid OOBN. Alternatively, the MCMC approach could be used to ease the constraints by repartitioning the nodes. This possibility is left for future work on the effective generation of OOBNs.

A randomly generated synthetic OOBN following the procedure outlined above is shown in Figure 4.17.



(a) (b) (c)

Figure 4.17: A sample OOBN with NON = 10, NOC = 2, NOO = 1, NOP = 3: (a–b) two foreign classes C_1 and C_2 (no embedded objects), and (c) the "main.oobn" class C having embedded objects

ALGORITHM 4.6 (GENERATE MAIN OOBN CLASS)

Call : GenerateMainOOBN(C, $LIST_{obj}$) → $\mathbb{C}_{main}$
Input: C: An OOBN class,
$LIST_{obj}$: A list of objects
Output: $\mathbb{C}_{main}$: An OOBN class

begin
  foreach $O \in LIST_{obj}$ **do**
    $< \mathbb{N}'_I, \mathbb{N}'_O >$ ← partitionOOBNNodes(C, O)
    foreach $input : N_{in} \in O$ **do**
      n_i ← RandomlySelectANode($\mathbb{N}'_I$)
      AddRefEdge(C, n_i, $input$)
    foreach $output : N_{out} \in O$ **do**
      n_o ← RandomlySelectANodeWithConstraint($\mathbb{N}'_O$)
      AddACausalEdge(C, $output$, n_o) // $output \rightarrow n_o$
  return C

ALGORITHM 4.7 (PARTITIONING OOBN NODES)

Call : partitionOOBNNodes(C, O) → $< \mathbb{N}'_I, \mathbb{N}'_O >$
Input: C: An OOBN class,
O: The object to be connected
Output: $< \mathbb{N}'_I, \mathbb{N}'_O >$: a pair of node partitions for potential causal and referential edges

begin
  N ← topologicalOrdering($\mathbb{N} \in C$)
  $\mathbf{N}_{obj}$ ← GetAllNodes(O)
  N_{in} ← GetAllInputNodes(O)
  $x \leftarrow \frac{|N_{in}|}{|N_{obj}|} \times |N|$
  $\mathbb{N}'_I$ ← selectFirstParition(N, x) //select 1st x items from N
  $\mathbb{N}'_O \leftarrow \mathbb{N}_{obj} \setminus \mathbb{N}'_I$
  return $< \mathbb{N}'_I, \mathbb{N}'_O >$

4.5.4 Producing outcomes (JT, construction time and cost) for analysis

Algorithm 4.8 outlines a high-level view of the approach followed to achieve the experimental outcomes of the compilation algorithms. In short, an OOBN class is randomly chosen, Hugin compilation is then applied and provided Hugin can successfully compile the class, it stores three values: compilation time, produced JT-cost, and the complexity of the flattened BN of the class.

ALGORITHM 4.8 (PRODUCING OUTPUTS FROM THE ALGORITHMS)

Call : produceCompilationOutcomes($Algos$, $LIST_{OOBN}$) →
$< Cost_{JT_H}, Cost_{JT_S}, Complexity, RT_{Hug}, RT_{SIIC}, RT_{SIIC\#} >$
Input: $Algos$: A list of compilation algorithms,
$LIST_{OOBN}$: A list of OOBNs
Output: $Cost_{JT_H}$,$Cost_{JT_S}$: Cost of JTs
$Complexity$: Complexities of the OOBNs
$< RT_{Hug}, RT_{SIIC}, RT_{SIIC\#} >$: Running time list of 3-tuple (Hugin, SIIC, SIIC#)

1 **begin**
2 $< RT_{Hug}, Cost_{JT_H} > \leftarrow \varnothing$
3 $< RT_{SIIC}, Cost_{JT_S}, Complexity > \leftarrow \varnothing$
4 $RT_{SIIC\#} \leftarrow \varnothing$
5 **while** $LIST_{OOBN} \neq \varnothing$ **do**
6 $C \leftarrow extractAnOOBNRandomly(OOBN)$
7 $< RT_{Hug}, Cost_{JT_H} > \leftarrow < RT_{Hug}, Cost_{JT_H} > \cup HuginCompilation(C)$
8 $< RT_{SIIC}, Cost_{JT_S}, Complexity > \leftarrow < RT_{SIIC}, Cost_{JT_S}, Complexity > \cup SIICompilation(C)$
9 $RT_{SIIC\#} \leftarrow RT_{SIIC\#} \cup SIICompilationJTExist(C)$
10 **return** $< Cost_{JT_H}, Cost_{JT_S}, Complexity_{BN}, RT_{Hug}, RT_{SIIC}, RT_{SIIC\#} >$

Irrespective of the success of the Hugin compilation, SIIC and SIIC# are run on the same OOBN class. SIIC returns the compilation time and the produced JT-cost. SIIC# looks for existing JTs for each of the encapsulated objects, hence producing the same JT as SIIC. Hence, only the compilation time of SIIC# is recorded.

As the OOBNs are generated randomly with very high and low values of the parameters, the complexity, running time and JT costs of the OOBNs reside within a extremely wide range between the very high and very low values. In the experimentation, recorded in the following sections, logarithmic scale is used to capture the values that fall within a reasonable range.

Table 4.3 shows, a summary of the overall number of cases considered in the experimentation. In order to build a fairer performance comparison platform, the table depicts how the experimentation outcome was cleaned.

Table 4.3: Summary of the cases considered in the experimentation.

	#Cases
Input	29120
Cases where all algorithms run	11976
Successful Folds:	
Count	**#Cases**
1	930
2	966
3	986
4	1539
Total (1x930 + 2 x 966 + 3 x 986 + 4 x 1539)	11976
After removing cases where Folds = 1	11046
After removing Hugin outlier (i.e., Hugin log(running time) >11)	11043
After removing SIIC# outlier (i.e., SIIC# log(running time) >8)	11043

Table 4.4: List of Units of Measurement (UOMs) for various measures used in the experimentation to evaluate the performance of the algorithms.

Evaluation Measures	**Unit of Measurement (UOM)**	**Explanation (if UOM = NA)**
Running/Compile time	Milli second (ms)	
Running time difference	Milli second (ms)	
JT cost	NA	A number found by the Equation in Section 4.3.4.2
JT Construction cost	NA	Asymptotic notation of Complexity analysis of Section 4.4
BN Complexity	NA	A number found by applying the first Equation in Section 4.5.1
OOBN Complexity	NA	A number as per the second Equation in Section 4.5.1

To report and demonstrate the experimental analysis, mostly box-plots and bar-charts (for distribution) were used. Box plots are ideal to compare distributions because the centre of the distribution (median), spread (quarters) and overall range of the distribution are immediately apparent. This is a way of summarising a set of interval-scaled data and is often used in explanatory data analysis [217].

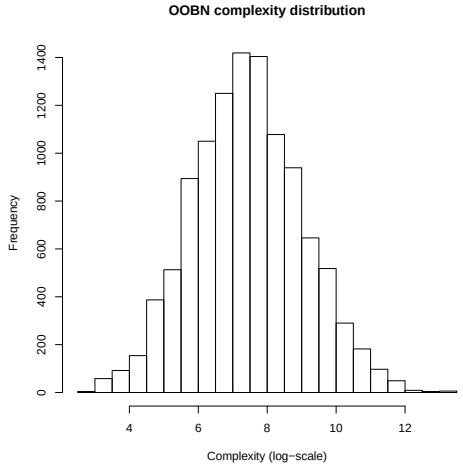
The data points (dependant variable values) found in the empirical analyses were scattered over a wide range of values. The reason for this scattering (variation) relates to is various parameters considered in generating the OOBNs, the mutual effects of the parameters, their overlapping behaviour, and the wide range of the values of independent variables. To deal with such unusual behaviour of the values, some of the outliers were removed by filtering a particular range of values. Then, plots were made from the filtered data. Furthermore, to neutralise the effects of any remaining scattered values, median values were considered (with the use of box-plots) rather than mean values.

4.5.5 Complexity of generated OOBNs

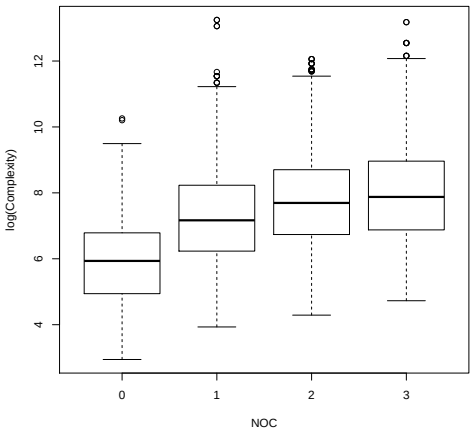
Figure 4.18 shows the distribution of the complexity (the generalized complexity of OOBNs described in Section 4.5.1) of the synthetic OOBNs using the approach described in Section 4.5.3.

In Figure 4.18a, the overall distribution of complexities of all the generated OOBNs are shown. Then figures 4.18b to 4.18f show the distributions of complexities in terms of varying NOC, NON, NOO, NOP, and NOS, respectively. It can be inferred that OOBN complexity increases if any value of NOC, NON, NOP, or NOS increases. The only exception is NON, where the complexity increases for increasing values of NON only up to 15. After 15 the complexity median exhibits a fall. The reason is that for higher NON, all the algorithms, namely Hugin, SIIC and SIIC#, fail to produce output and the current implementation of the algorithms fails to calculate the complexity of OOBNs if the algorithm fail to generate any JT.

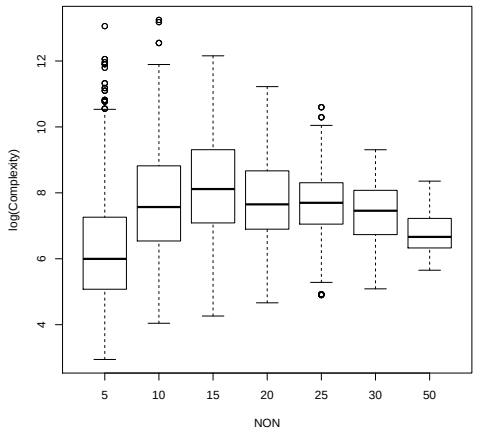
The table in Figure 4.18g summarizes the analyses performed using histograms in Figure 4.18a.



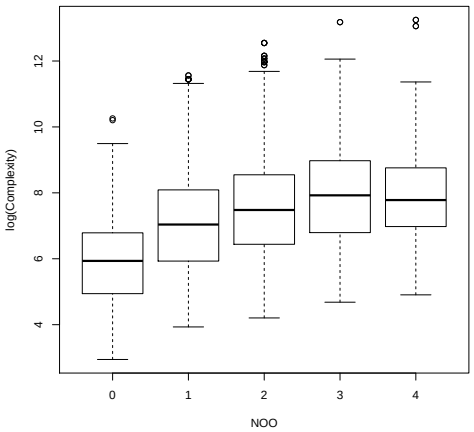
(a)



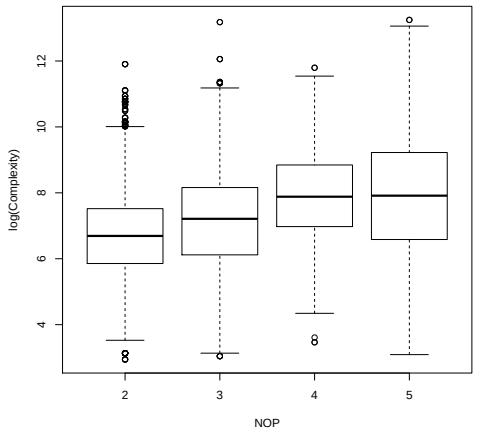
(b)



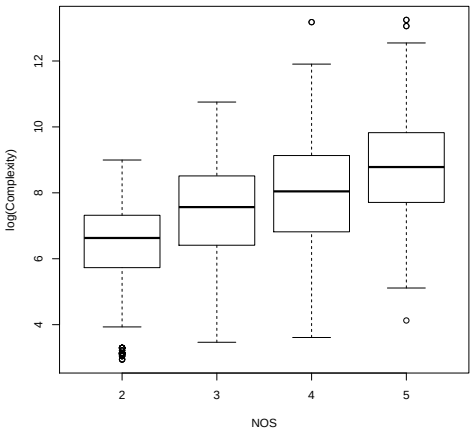
(c)



(d)



(e)



(f)

Summary	Min	1st Qu	Median	Mean	3rd Qu	Max
Complexity	2.944	6.308	7.371	7.425	8.484	13.247

(g) Summary of the overall complexity distribution of the generated OOBNs

Figure 4.18: Complexity distribution of the synthetic OOBNs: (a) overall, and w.r.t. (b) NOC, (c) NON, (d) NOO, (e) NOP, and (f) NOS. Complexity increases with the increase in NOC, NOO, NOP and NOS (except for NON).

4.6 Performance of Hugin, SIIC and SIIC# Algorithms

This section outlines the experimental comparison of the existing (Hugin JT construction) and proposed compilation (SIIC and SIIC#) algorithms. Note that SIIC# is a special SIIC that retrieves JTs rather than constructing JTs for objects embedded in an OOBN class and JTs for a superclass if there are any. The comparison of the performance of the algorithms is performed based on two main factors, namely time required to create JTs (i.e., running time) and the cost [212] of the produced JTs.

4.6.1 Time required to compile OOBNs

Figures 4.19a, 4.19b and 4.19c show the distributions of the running time of Hugin, SIIC and SIIC# algorithms. Running times (logarithmic scale) range between 3.5 and 10 for Hugin with the maximum frequency between 4 and 5. For SIIC, the frequency range is from 3 to 10, with higher frequencies observed between 4 and 6. For SIIC#, the running time frequency is from 0 to 6, and high frequency is between 3 and 4.

In figures 4.19d, 4.19e and 4.19f, the running times of Hugin, SIIC and SIIC#, respectively, are plotted. The median of running times of Hugin and SIIC are approximately 5.0, where SIIC#'s running time median is between 3 and 4. Note that only the cases where all three algorithms ran successfully are considered.

Figures 4.20 to 4.21 show the performance of Hugin, SIIC and SIIC# algorithms in terms of running time with varying values of the parameters NOC, NON, and NOO (See Appendix C for the analyses based on NOP and NOS shown in Figure C.2).

Figures 4.20a, 4.20b and 4.20c represent the distribution of Hugin, SIIC and SIIC# in terms of running time for increasing values of NOC. All three algorithms' running time increases with the increase in NOC. The median of running time (in log scale) for Hugin and SIIC ranges from 4 to 5 whereas for SIIC# the running time median ranges between 2.75 and 3.5.

Figures 4.20d, 4.20e and 4.20f demonstrate the performance plot of the three algorithms for using the running times with varying NON-values. The running time of the three algorithms increases with the increase in NON except for SIIC and SIIC# when NON = 50. The increasing tendency is not evident there. This discrepancy is observed because for larger values of NON, the cases where Hugin successfully generated JTs, Hugin required a huge amount of time and hence there was a sharp increase in Hugin's distribution. However, for the very few cases where all three algorithms worked with high NON value, SIIC and SIIC# did not require such a long time. Hence, the median value experienced a downfall for NON=50. The median of Hugin and SIIC running time is between 4 and 6, whereas for SIIC# the median is between 2.75 and 4.

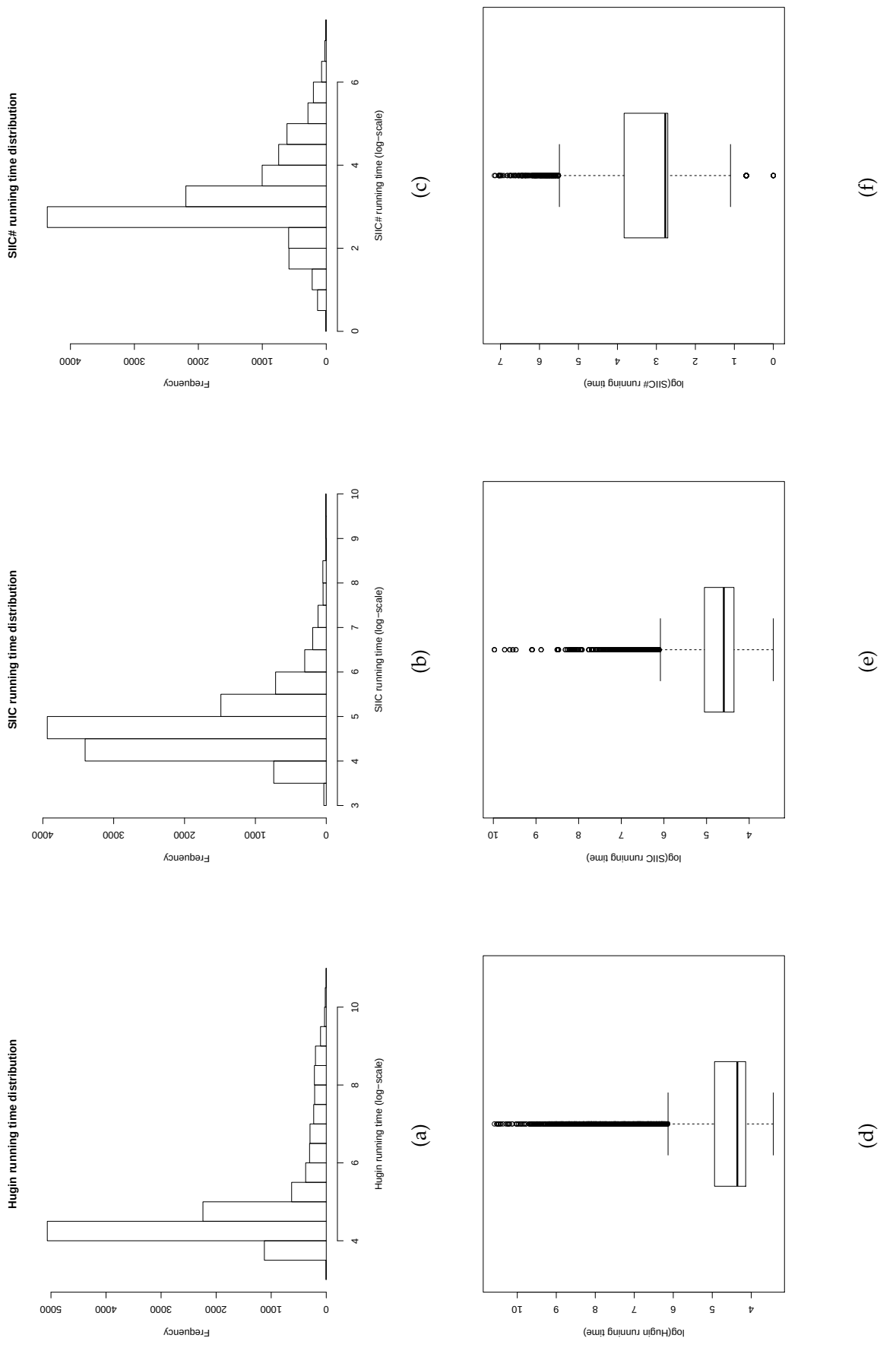


Figure 4.19: Running time distribution: (a) Hugin, (b) SIIC, and (c) SIIC#; Box-plots: (d) Hugin, (e) SIIC, and (f) SIIC#. Normal distributions are observed for all three algorithms running times (a–c). Better performance w.r.t. running time is observed for SIIC# (d–f).

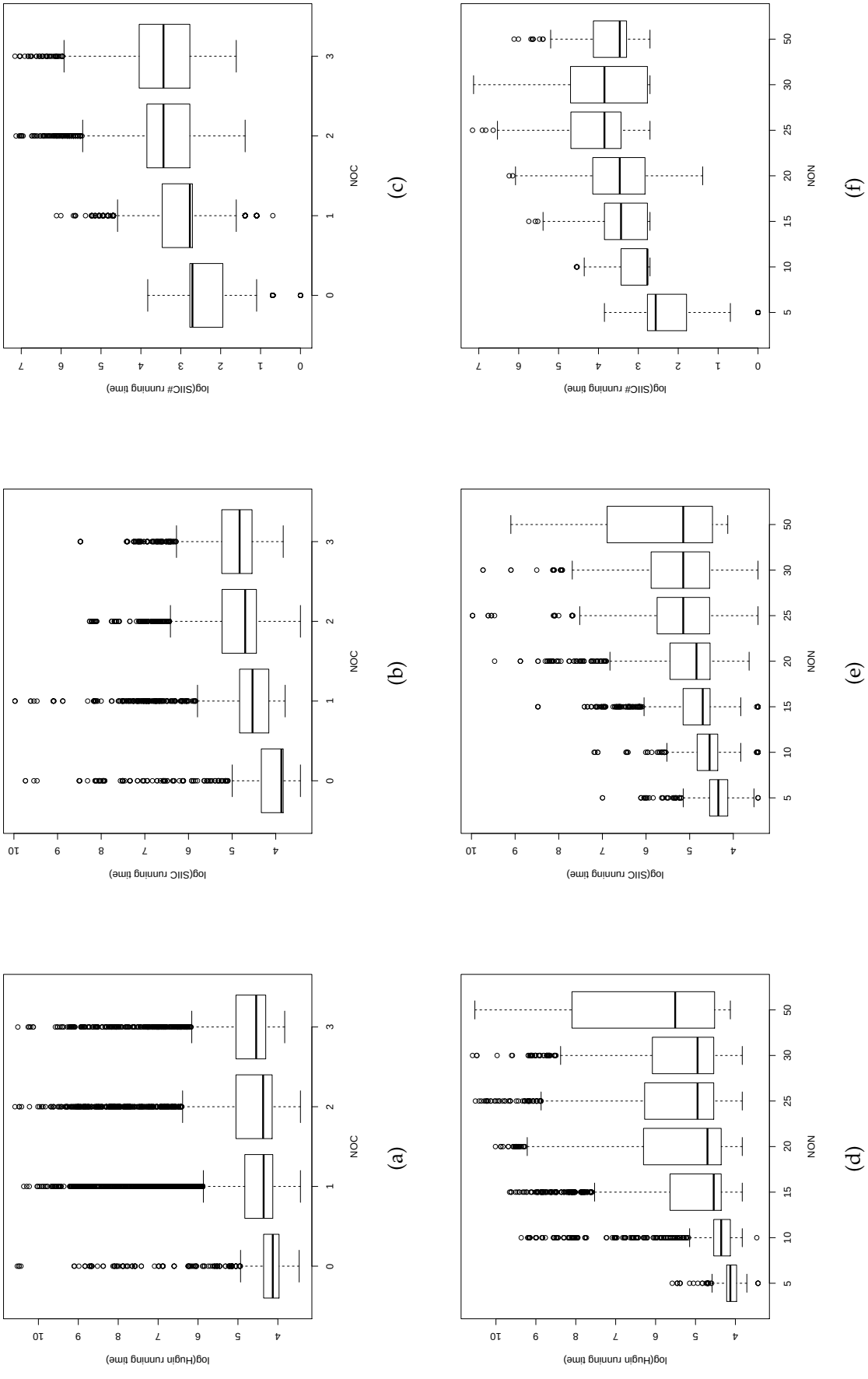


Figure 4.20: Running time w.r.t. NOC: (a) Hugin, (b) SIIC, and (c) SIIC#; Running time w.r.t. NON: (d) Hugin, (e) SIIC, and (f) SIIC#. Running time increases with the increase in both NOC and NON. For both NOC and NON, SIIC# performs better than SIIC and Hugin.

The box-plots in figures 4.21a, 4.21b and 4.21c, respectively, analyses the running time performance of Hugin, SIIC and SIIC# w.r.t. NOO. Though all three algorithms' running time increased with the increase in NOO, the impact of varying NOO was clearly visible in SIIC and SIIC#. Both Hugin and SIIC had the same range of median values, i.e., 4 to 5, but SIIC# had a considerable gain, with the median being between 2.9 and 4. The reason behind this difference in the performance of SIIC and SIIC# is that SIIC may involve generating JTs for each embedded object but SIIC# fetches all JTs of embedded objects from the database. In other words, the JT of a class is created only once and reused as many times as the number of objects are there for the same class in SIIC whereas in SIIC# the step of creating the JT, even for the first time, is not needed. This technique saves the time for flattening and compilation that is required in Hugin.

This analysis has demonstrated the strength of the SIIC and SIIC# algorithms, i.e., the more objects, the better the performance of these algorithms is. Hence, in Section 4.6.3, more analyses were performed for OOBN classes with and without embedded objects.

In Appendix C, Figure C.1, another very essential performance investigation of the algorithms is performed: i.e., running time performance of SIIC, SIIC# and Hugin is plotted against the complexity of the generated OOBNs.

These results are shown in Figure 4.22. It can be inferred from the analyses shown in Figure 4.22 that SIIC performs better than Hugin, even after only considering the cases where both Hugin and SIIC failed. In Figure 4.22a, the distribution of the overall running time difference of Hugin and SIIC is shown. The plot supports the claim mentioned above. For varying values of different parameters, such as NOC, NON, NOO, NOP, and NOS, the running time difference is documented in figures 4.22b, 4.22c, 4.22d, 4.22e, and 4.22f, respectively. For NOP, the median of the difference is almost consistent but not less than 0 which indicates that Hugin requires more time in most cases than SIIC if the number of parents per node increases. For NOC and NOO, the difference decreases but does not become lower than 0. That means for more classes and objects, SIIC's performance is not improving significantly though its performance is still better than Hugin. Finally, for NON and NOS, the greater the parameter values, the higher is the difference between Hugin and SIIC. This is a clear indication that for high NON and NOS values, SIIC performs way better than Hugin. The following table 4.7 summarises the outcome of the plotting shown in Figure 4.22. The reason behind such superior performance in SIIC is that SIIC partitions the whole compilation process and can reuse JTs for embedded objects if there is more than one objects from the same class. In Hugin, an OOBN with more objects needs to be flattened and compiled into a large BN (flattened OOBN class).

Table 4.5 summarizes the analysis of the overall distribution of the running time difference

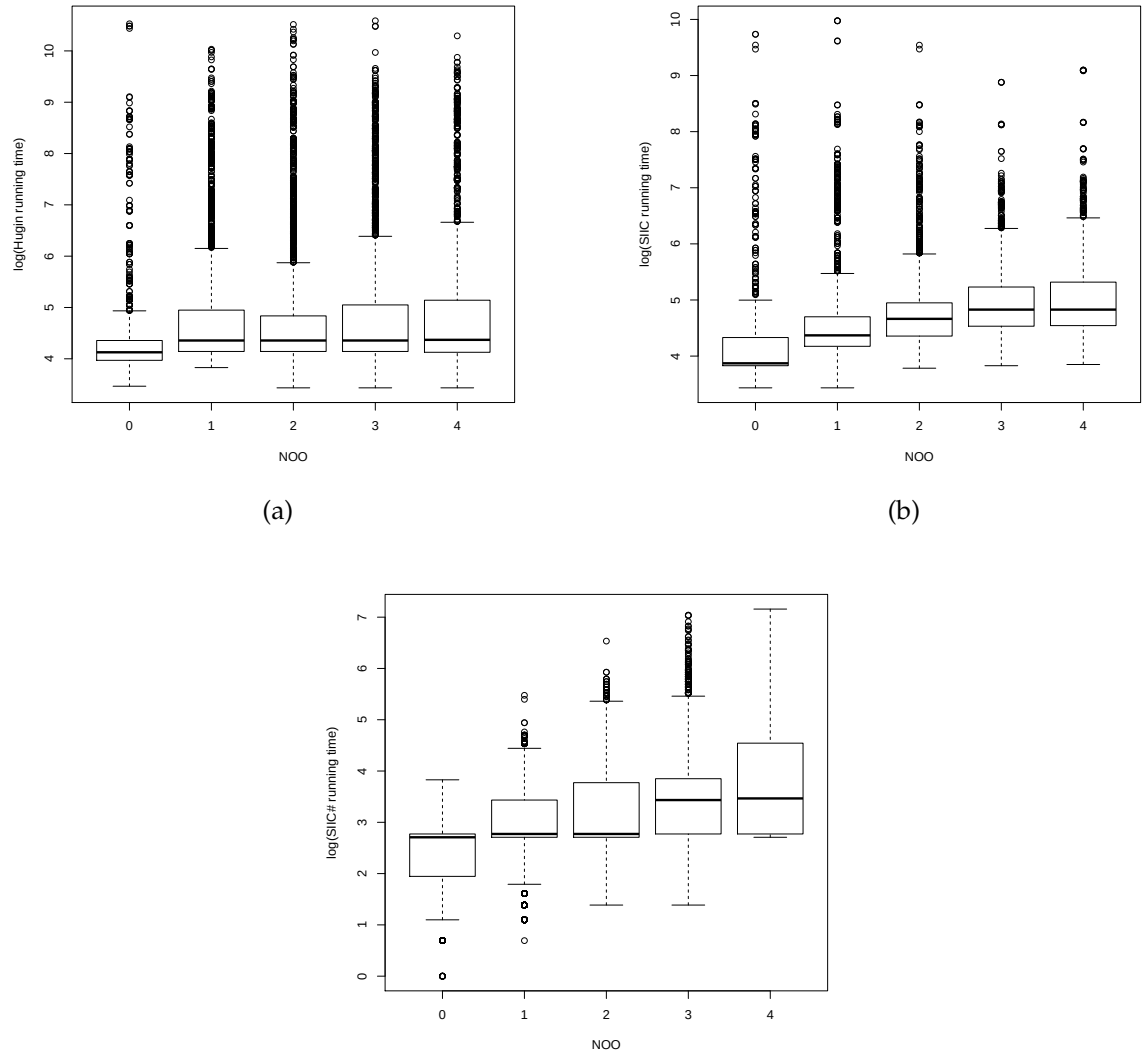


Figure 4.21: Running time w.r.t. NOO: (a) Hugin, (b) SIIC, and (c) SIIC#. All algorithms' running time increases with the increase in NOO and SIIC# has least running time than Hugin and SIIC.

between Hugin and SIIC.

Table 4.5: Frequency of the mid-points of the bars in the histogram for time difference between Hugin and SIIC (UOM = ms)

Mid Points	Counts	Mid Points	Counts	Mid Points	Counts
-3.25	1	-0.25	5386	2.75	181
-2.75	1	0.25	2010	3.25	114
-2.25	10	0.75	524	3.75	86
-1.75	21	1.25	336	4.25	33
-1.25	213	1.75	281	4.75	6
-0.75	1625	2.25	213	5.25	2

Alike analyses were performed to investigate the performance gain of SIIC# over Hugin.

Refer to Figure 4.23 to see the performance comparison of Hugin and SIIC#. The overall distribution (irrespective of any specific parameter) of the running time difference of Hugin and SIIC# is shown in Figure 4.23a. This plot of the performance implies that Hugin is outperformed by SIIC# in almost all cases. The bars in the histograms are higher at around 1 to 3. The distribution is mostly biased towards positive values. Then in figures 4.23b to 4.23f, the performance gain of SIIC# over Hugin is shown with different values of NOC, NON, NOO, NOP, and NOS, respectively. No matter which parameter is considered nor how much its value increases, SIIC# performed better than Hugin. The superior performance of SIIC# is evident because SIIC# partitions the whole compilation process and can reuse JTs for embedded objects created beforehand. Note that the difference decreases for higher values of NOC and NOO, this is because for new foreign classes, SIIC# needs to fetch JTs from storage, which is normally a slow operation. Hence, there is still some scopes to improve SIIC#'s performance by faster accessing the JTs held in disc storage.

The summary of the running time distribution of Hugin, SIIC, SIIC# and the difference between Hugin–SIIC and Hugin–SIIC# is illustrated in Table 4.7. The table can be used for getting further insight into the analysis performed for running time of the algorithms.

Table 4.6 summarizes the analysis of the overall distribution of the running time difference between Hugin and SIIC#.

Table 4.6: Frequency of the mid-points of the bars in the histogram for time difference between Hugin and SIIC#.

Mid points	Counts	Mid points	Counts	Mid points	Counts
-2.25	1	1.25	3110	4.75	207
-1.75	9	1.75	2210	5.25	152
-1.25	26	2.25	1082	5.75	63
-0.75	197	2.75	574	6.25	33
-0.25	385	3.25	425	6.75	9
0.25	542	3.75	242	7.25	7
0.75	1563	4.25	206		

Table 4.7: Summary of the time difference between Hugin–SIIC and Hugin–SIIC#.

Summary	Hugin Time	SIIC Time	SIIC# Time	(Hugin - SIIC) Time	(Hugin - SIIC#) Time
Min	3.434	3.434	0	-3.14745	-2.189
1st Qu	4.143	4.357	2.708	-0.37753	1.043
Median	4.357	4.595	2.773	-0.12401	1.435
Mean	4.871	4.78	3.206	0.09133	1.665
3rd Qu	4.942	5.05	3.829	0.18659	2.105
Max	10.587	9.976	7.158	5.28675	7.325

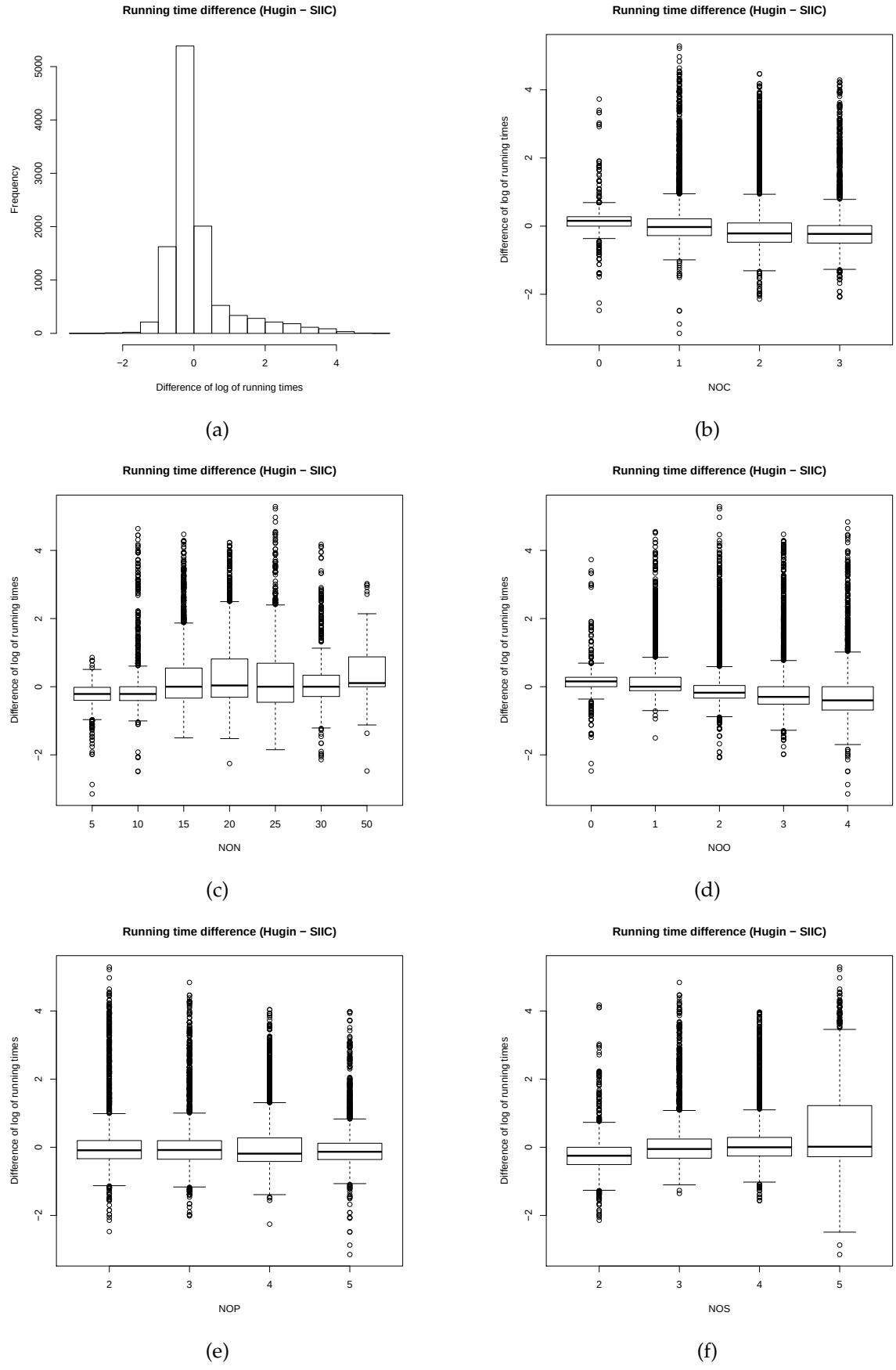


Figure 4.22: Running time difference of Hugin and SIIC: (a) overall distribution; and w.r.t. (b) NOC, (c) NON, (d) NOO, (e) NOP, and (f) NOS. The running time difference exhibits a normal distribution; the difference roams around zero; the time difference is positive (SIIC performs better) for NON, NOP, and NOS; and for other parameters, the difference is negative (Hugin performs better).

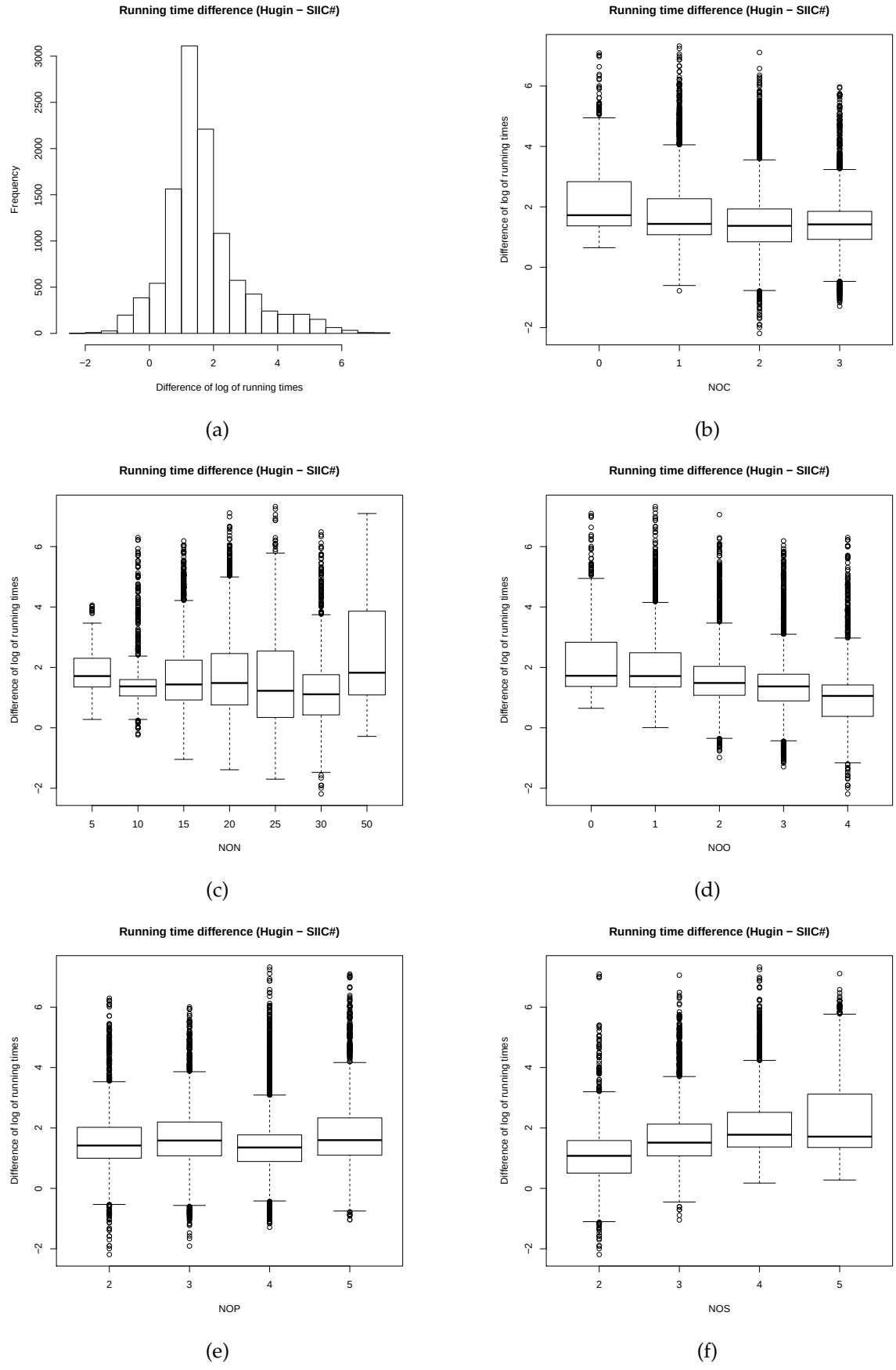


Figure 4.23: Running time difference of Hugin and SIIC#: (a) overall distribution; and w.r.t. (b) NOC, (c) NON, (d) NOO, (e) NOP, and (f) NOS. The running time difference exhibits a normal distribution and the difference is always higher than zero (SIIC# performs better).

4.6.2 Cost comparison of the JTs produced by Hugin and SIIC

In this section, an essential outcome of the compilation algorithms, i.e., JTs are analysed in terms of the cost of the JTs (a measure proposed by Kanazawa [212]). Note that SIIC and SIIC# both produce the same JT and hence the cost of the JT for SIIC and SIIC# are the same. To avoid repetition, SIIC#'s JT cost is not analysed in this thesis.

Figures 4.24a and 4.24b, respectively, show the JT-cost of the JTs created by the algorithms Hugin and SIIC. The distributions of the JT-cost exhibits a normal distribution and does not show any tendency to a particular value or range. As the SIIC is not expected to generate efficient JTs (i.e., the JT-cost of the produced JT is minimal), it is obvious that the JT-cost of SIIC is higher than the JT-cost of Hugin because the clique sizes in the JT created by SIIC are larger than those created by Hugin. The current implementation did not include the thinning process of the cliques in the JT generated by SIIC. This particular crucial task was left as an extension of the implementation, as the main focus was on reduction of compilation time for very large OOBNs. Figure 4.24c shows the distribution of the difference of the JT-costs of Hugin and SIIC. The difference indicates that the cost of the JTs produced by SIIC are mostly higher than for Hugin. To get a better idea about the distribution, Table 4.8 shows the summary of the JT cost distribution for Hugin, SIIC and the difference between the Hugin and SIIC JT cost.

Table 4.8: Summary of the JT cost difference between Hugin and SIIC

Summary	Hugin JT Cost	SIIC Cost JT	(SIIC -Hugin) JT Cost
Min	2.773	2.773	-5.6843
1st Qu	9.566	11.617	0.7882
Median	12.889	16.072	2.3534
Mean	13.018	16.633	3.6143
3rd Qu	16.253	20.756	5.1036
Max	21.889	49.671	31.225

Figures 4.24 to 4.26 show the correlation of the JT-cost of Hugin and SIIC with various parameters listed in Table 4.2.

For the increase in NOC, both the Hugin and SIIC JT-cost increases. However, the increase in SIIC JT-cost is higher than for the Hugin (see figures 4.24d and 4.24e). Hugin JT-cost increases with the increase in NON but SIIC JT-cost increases for lower values of NON and remains stationary for higher NON. Then Figure 4.24f shows the distribution of the differences of Hugin's JT cost and SIIC's JT cost for the same OOBN with varying NOC. The difference increases with the increment of NOC and this is evident in the figure.

In figures 4.25a and 4.25b, the distribution of JT cost of the JTs produced by the algorithms Hugin and SIIC, respectively, are shown with varying values of NON. For the increase in NON, both the Hugin and SIIC JT-cost increases. However, the increase in Hugin JT-cost is

higher than for the SIIC though the overall value of SIIC JT-cost is higher than for Hugin. The Hugin JT-cost increases with the increase in NON but SIIC JT-cost increases for lower values of NON and remains stationary for higher NON values. Then in Figure 4.25c the distribution of the differences of the JT costs of Hugin and SIIC for the same OOBN with varying NON is depicted. The decrease in difference is evident for the higher values of NONs.

For NOO (figures 4.25d and 4.25d) the Hugin and SIIC JT-cost increase. The difference between the JT costs of the two algorithms (Hugin and SIIC) also increases with the increasing value of NOO (refer to Figure 4.25f).

For NOP, Hugin (Figure 4.26a) and SIIC (Figure 4.26b) JT-cost increases up to 4 but starts decreasing where NOP = 5. The reason behind this abnormality is that for higher NOP, Hugin (and hence SIIC also) fail to generate any JT. Since there are very few cases here a JT is created, the box-plot for the JT-cost with NOP= 5 can be ignored. In Figure 4.26c, the depicted distribution indicates that the difference between Hugin's and SIIC's JT cost remain steady but is always greater than zero. The non-zero difference indicates that for varying NOP, the JT cost of SIIC is always higher than Hugin's.

Figures 4.26d to 4.26f, respectively, plot the distribution of the JT costs of Hugin and SIIC algorithms and the difference between the two with respect to increasing value of NOS. The plots indicate that the costs increase with an increasing value of NOS in all cases.

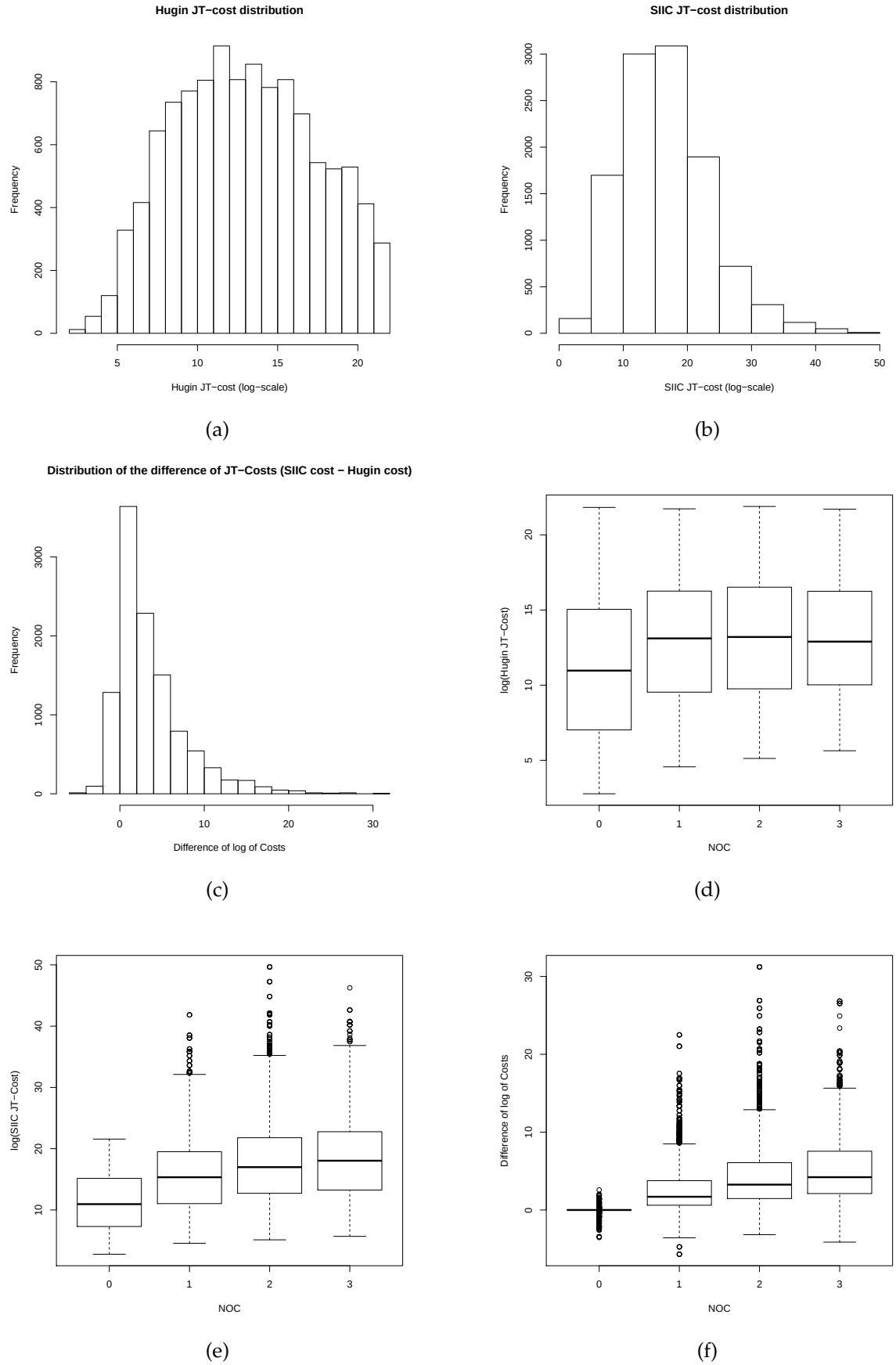


Figure 4.24: JT cost distribution (overall): (a) Hugin, (b) SIIC, and (c) difference of SIIC and Hugin (SIIC - Hugin), normal distributions are observed in a–c; JT cost distribution w.r.t. NOC: (d) Hugin, (e) SIIC, and (f) difference of SIIC and Hugin. JT cost and difference of cost increases with the increase in NOC.

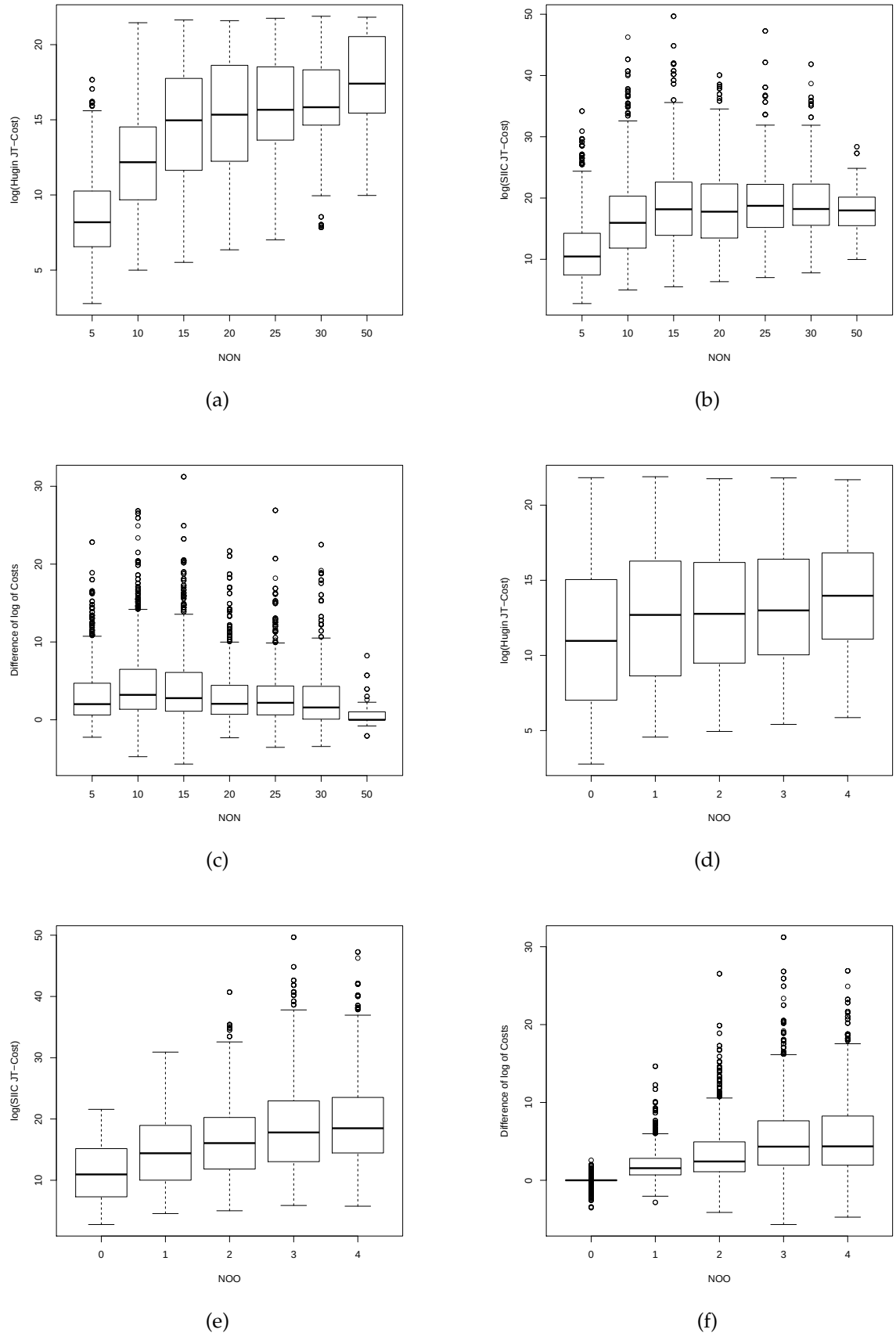


Figure 4.25: JT cost distribution w.r.t. NON: (a) Hugin, (b) SIIC, and (c) difference of SIIC and Hugin (SIIC - Hugin); and w.r.t. NOO: (d) Hugin, (e) SIIC, and (f) difference of SIIC and Hugin. JT cost and difference of cost increases with the increase in NON and NOO.

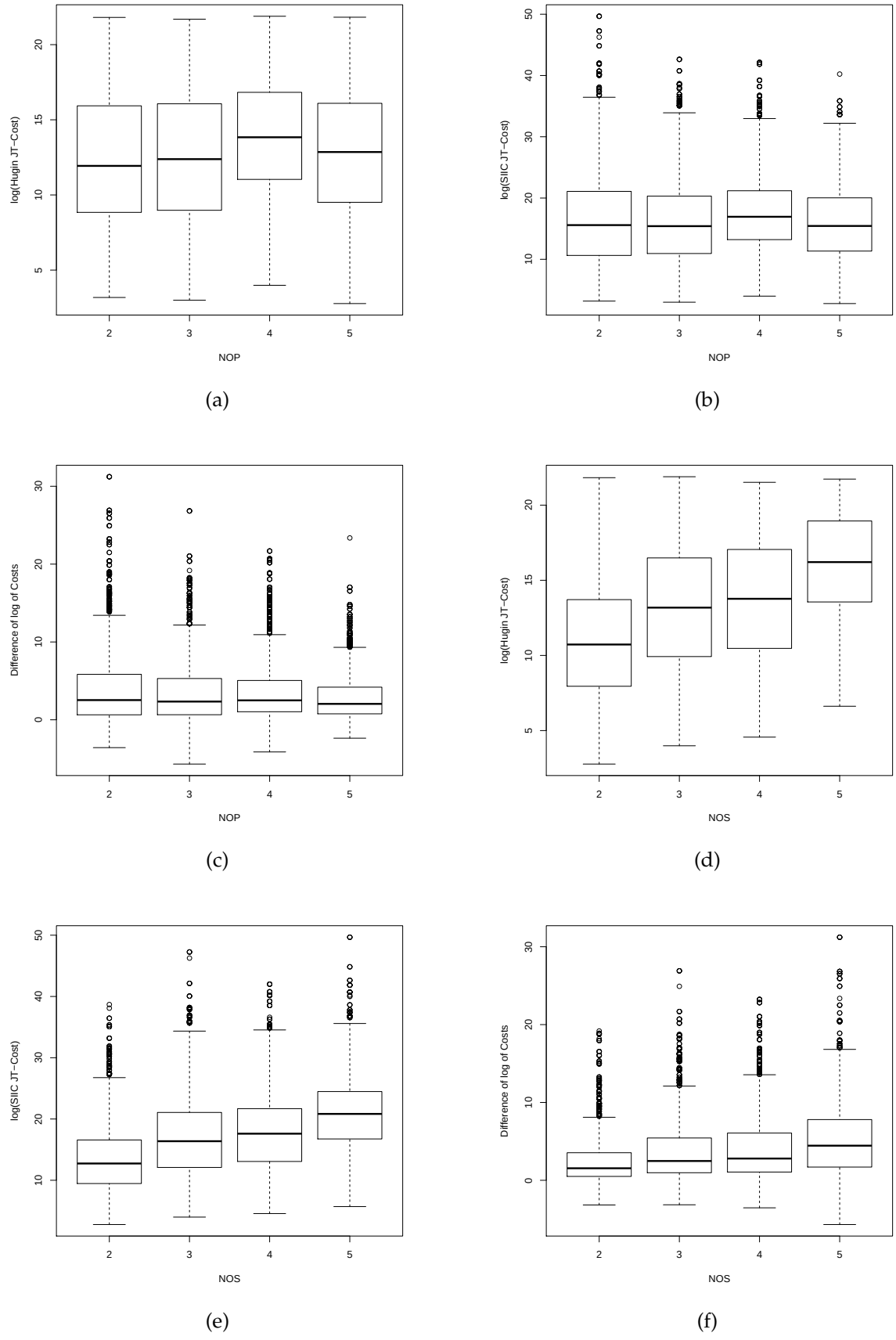
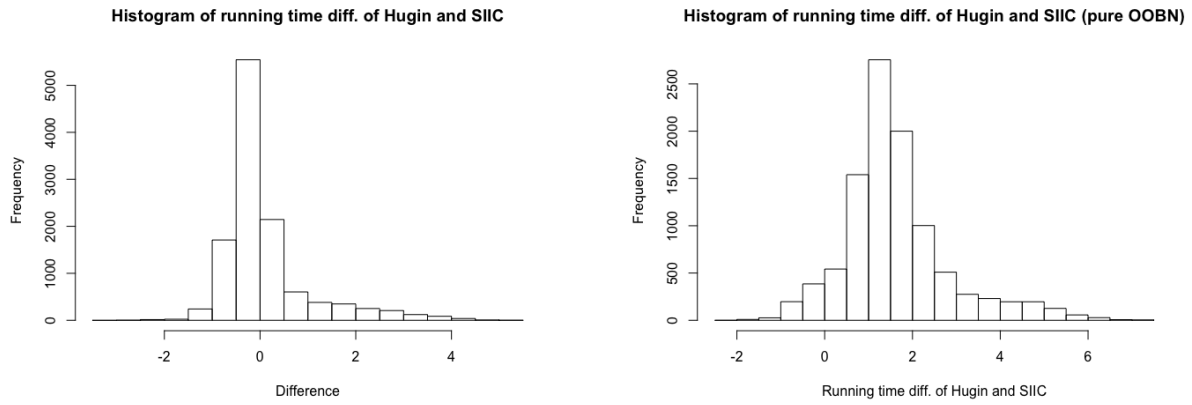


Figure 4.26: JT cost distribution w.r.t. NOP: (a) Hugin, (b) SIIC, and (c) difference of SIIC and Hugin (SIIC - Hugin); and w.r.t. NOS: (d) Hugin, (e) SIIC, and (f) difference of SIIC and Hugin. JT cost and difference of cost increases with the increase in NOS and NOP.

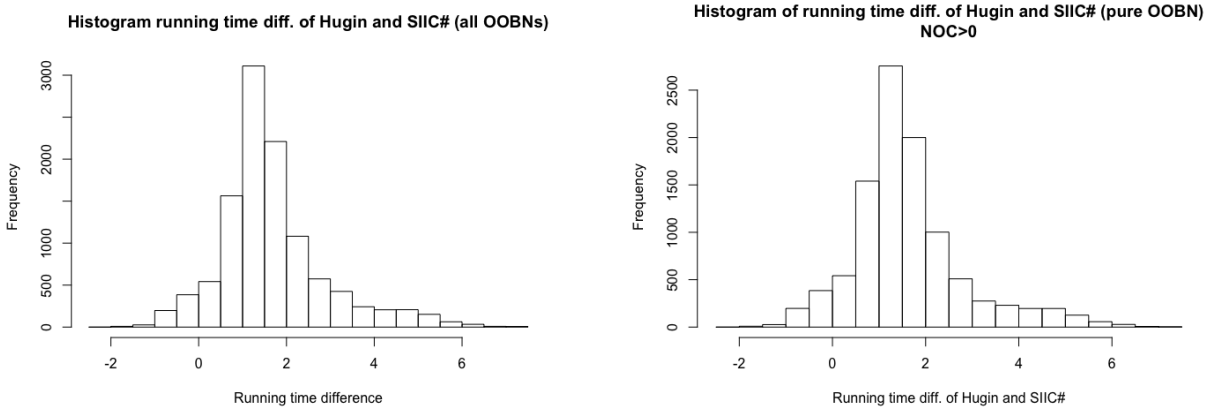
4.6.3 Effect of embedded objects on the performance

SIIC and SIIC# achieve no gain over Hugin in cases where the OOBNs have no embedded objects (non-pure OOBNs). To illustrate how much the performance of SIIC degrades in case of all OOBNs (i.e., both pure and non-pure OOBNs), Figure 4.27a shows the running time difference between Hugin and SIIC for all OOBNs (where $NOC \geq 0$), while Figure 4.27b shows this for pure OOBNs (where $NOC > 0$). For all OOBNs, the difference ranges from -2 to 4 with a high frequency at 0, and for pure OOBNs, the range is from -2 to 6 with a high frequency at 1 to 2. One point needs to be mentioned, namely, that the more positive the difference, the better the performance of the SIIC algorithm is. Hence, in the case of pure OOBNs (OOBNs with embedded objects where reusability can be introduced), SIIC performs better than Hugin and better than models with non-pure OOBNs.



(a) (b)

Figure 4.27: Time difference of Hugin and SIIC (Hugin - SIIC) distribution for: (a) all OOBNs, (b) pure OOBNs. Normal distributions are observed in both cases.



(a) (b)

Figure 4.28: Time difference of Hugin and SIIC# (Hugin - SIIC#) distribution for: (a) all OOBNs, (b) Pure OOBNs. Normal distributions are observed in both cases.

Similarly, to illustrate the performance of SIIC# with and without embedded objects in a class, Figure 4.28a plots the running time difference between Hugin and SIIC# for all OOBNs (where $NOC \geq 0$), while Figure 4.28b plots the time difference of Hugin and SIIC# for pure OOBNs (where $NOC > 0$), respectively. Similar to the difference between Hugin and SIIC, the more positive the difference between Hugin and SIIC#, the better the performance of SIIC#. In both cases, the frequency of the time difference is more positive than negative. This implies that SIIC# performs better than Hugin.

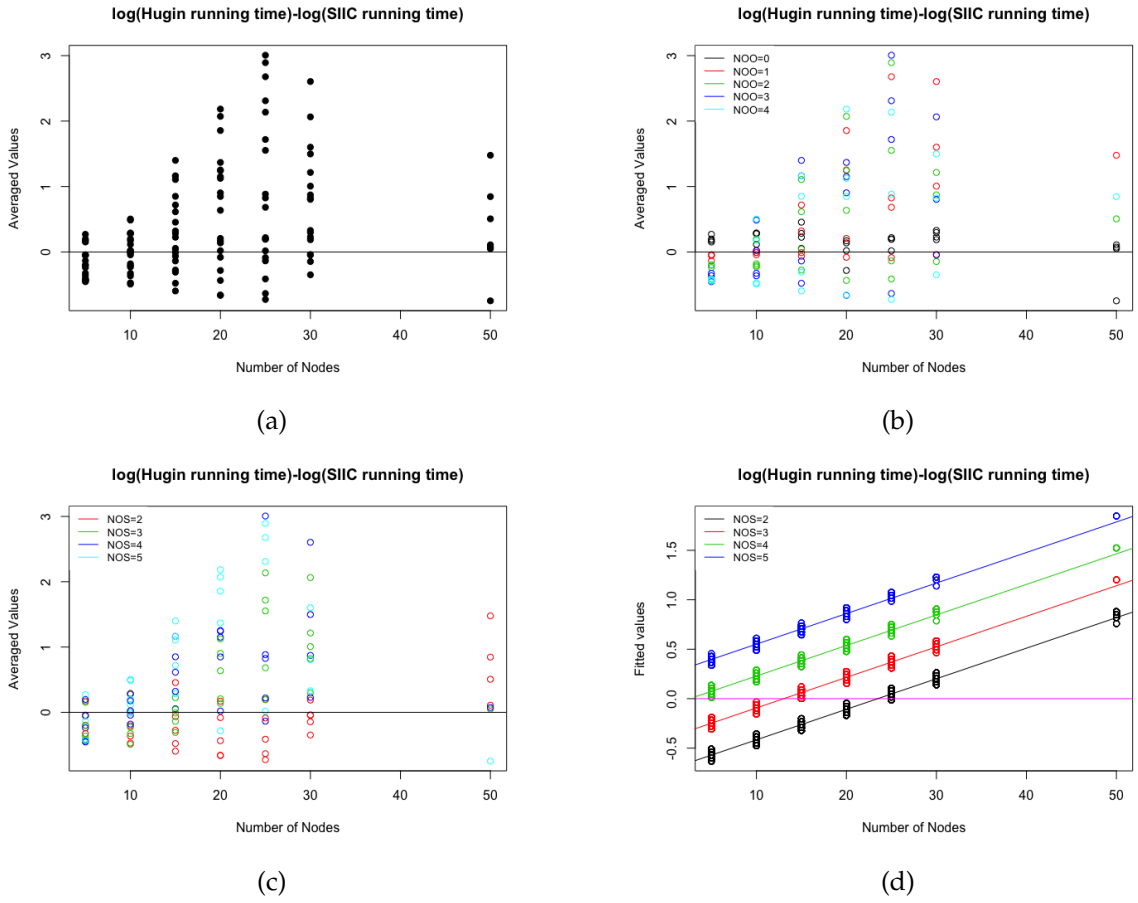


Figure 4.29: (a) NON vs average running time difference: (a) Hugin vs SIIC, (b) Hugin vs SIIC for varying NOO, (c) varying NOS, (d) NON vs fitted (theoretical) values in Model 4 with various NOSs: Hugin vs SIIC

Figures 4.29a, 4.29b and 4.29c illustrate the performance of the algorithms w.r.t. NON, NOO and NOS. The figures plot the average values of the running time difference in the vertical axis. The average running times are plotted against NON (Figure 4.29a). Results indicate that with an increase in NON, the performance of SIIC improves. In figures 4.29b and 4.29c, the average time difference is plotted against NON, but the points with different NOO and NOS values are plotted in different colours. The plots signify that for higher NOO and NOS with increasing NON, the performance of SIIC improves. This analysis is supported by the analysis performed

in Figure 4.29d which shows an increasing trend of fitted values (Model 4 of Table C.1) for different NOS with increasing NOO.

An extensive analysis and comparison of the performance of the proposed algorithms is included in Appendix C, Section C.2.2.

4.6.4 Summary of the experimentation

The three algorithms (Hugin, SIIC and SIIC#) did not successfully generate JTs for all OOBNs. The most significant reasons behind the failure of the algorithms are (i) excessively big BNs/OOBN classes in terms of number of nodes, (2) large numbers of states per node and large numbers of parents per node caused compilation to fail in compiling, and (3) repeated performance of complex operations like triangulation, clique finding, and JT construction. Hugin's performance suffered from all of the aforementioned issues; SIIC avoided some of them and SIIC# could avoid them more successfully than SIIC. In fact, SIIC# never suffered from the third issue because of its ability to reuse JTs. Hence, the success of compilation should also be considered as a performance factor, indicating that SIIC# performed better than SIIC. Table 4.9 shows a summary of these experiments. It has two parts. The upper part contains the statistics of the experimentation showing the total number of cases and configurations used in the study. It shows the number and percentage of cases in which the algorithms ran or failed. The lower part of the table compares the three algorithms (Hugin, SIIC, SIIC#) on their relative performance against each other and shows the number and percentage of wins, losses and tied outcomes for each. In order to make the comparisons statistically significant, the algorithms were compared using the paired t-test for the four runs of each of the trialled and distinct networks.

In order to run the experimentation, the total configurations were 1456 (as in Table 4.2) and for each of the configurations, five OOBNs (a total of 7,280 OOBNs) were generated. Each of the three algorithms (Hugin, SIIC and SIIC#) was tested four times on every single OOBN. That means the experimentation was conducted on a total of 29,120 cases and produced the following outcomes: In 51% of cases, Hugin failed to produce any JT, while SIIC and SIIC# failed in 27% and 17% occasion, respectively. Therefore, to compare the performance further, only the 14,060 cases where all three algorithms were able to generate a JT successfully were considered.

It is worth noting that SIIC and SIIC# failed because Hugin failed to generate any JT for the embedded objects in an OOBN class. Also, where the proposed algorithms for SIIC and SIIC# were implemented to produce JTs without embedded objects, Hugin's JT construction was used. This was justified because SIIC and SIIC# can, in principle, construct JTs for classes

with no embedded objects and no parent classes by using any traditional approach. So, using Hugin, in this case, ensured that any traditional algorithm would work successfully within SIIC and SIIC#. Also, note that the required running time of Hugin for creating the JTs for embedded objects was taken into account in computing the running time of SIIC and SIIC#.

Table 4.9: Experimentation summary

	Count	%
Total Configurations	1456	
Total OOBNs	7280	
Total cases	29120	
Hugin Fails	15060	51.72%
SIIC Fails	7899	27.13%
SIIC# Fails	4991	17.14%
All algo. Pass	14060	48.28%

t-test statistics
Total cases = 3515 (14060 / 4)

	Hugin wins		Hugin loses		Tied	
Hugin vs SIIC	312	8.88%	65	1.84%	3138	89.28%
Hugin vs SIIC#	4	0.11%	762	21.68%	2725	78.21%
	SIIC wins		SIIC loses		Tied	
SIIC vs SIIC#	0	0.00%	1151	32.75%	2340	67.26%

The detailed comparison on how many times Hugin wins and loses against SIIC and SIIC#, how many times the algorithms tie, and how many times SIIC# outperforms SIIC, are listed, as mentioned, in the lower part of the table, where it is clear that even if the 51% of times where Hugin failed are ignored, SIIC# outperformed both Hugin and SIIC. The percentage of cases in which Hugin outperformed SIIC is low, namely 8.88%. In computing the differences and their significance, a "t-test" was used.

4.7 Summary

A new incremental compilation algorithm, SII compilation, has been proposed for OOBNs, that, unlike previous methods, does not require transforming the OOBN into its underlying BN. There are two kinds of reuse: (1) when compiling a subclass, to reuse the JT of its superclass; and (2) when compiling a class with embedded objects, to reuse the JTs from those objects. It has been proven that SIIC compilation produces a valid JT. The JT constructed from the proposed SIIC algorithm may, and in examples to date tends to, contain larger cliques in comparison to the cliques of existing approaches. Nevertheless, after the final thinning step, an example showing that the resultant JT can be more compact has been shown, using a JT-cost measure that captures the complexity of inference on that JT. A future plan is to explore whether further optimization can be done in the algorithm to reduce the clique sizes and

whether there are theoretical results regarding the quality (the maximum number of branches) of the resultant JT.

The extent to which the proposed incremental compilation algorithm reduces the computation obviously depends on the structures of the class being compiled and the classes of any embedded objects, as well as the location, nature and extent of the modifications. Next, an experimental analysis should be conducted across a range of real-world and synthetic classes to explore what reductions can be achieved in practice. In sum, the proposed SIIC and SIIC# algorithms were efficient in compiling complex and very large OOBNs with large numbers of parents per node. Moreover, it was shown that Hugin (the most widely used, most stable commercial BN/OOBN modelling software) is not capable of compiling very complex and large real-life models like the WGR (Western Grassland Reserve project) whereas both SIIC and SIIC# were able to compile the whole project in a reasonable time.

Chapter 5

iOOBN (OOBN) Class Hierarchy Learning

The automated learning of causal structures [1] from data has received attention from researchers for decades [155]. Numerous algorithms have been proposed, and several approaches have been taken to learning the causal structures of BNs [154, 156]. However, to our knowledge there has been no work done to date to develop a technique for fully automated structure learning for OOBNs, although Bangsø et al. [183] proposed a semi-automated OOBN structure learning that relied on experts' opinion or guidance. Some possible reasons are that it is difficult to capture Object-Oriented notions from data and OOBNs are relatively new and still not widely used. Inheritance, one of the most interesting and promising features of OO, has not yet been defined well, nor practised enough in the OOBN paradigm.

With an increasing trend to model real-life large and complex problems using BNs, the value of OOBNs is more in focus. In fact modellers are using OOBNs in most of the recent modelling projects [18, 57, 93]. However, in spite of the massive interest in OOBNs, modellers find it challenging to switch from traditional ordinary BNs to OOBNs. One of the biggest challenges to switching is that of converting an on going and currently in use project that was originally modelled in a BN into an OOBN model. Moreover, an OOBN system may become large and complex to maintain if an inappropriate hierarchy is constructed or an initial hierarchy has been extended by different people at different times.

These facts have motivated the learning of hierarchical relations between BNs or restructuring the OOBN hierarchy by learning a new hierarchy from a set of OOBN classes. Hierarchy learning is a step towards automated learning of OO models from data and provides a novel approach to converting a BN repository into an OOBN repository. This thesis proposes a supergraph-based hierarchy learning for OOBN classes. The algorithm converts ordinary BNs to OOBNs using some simple heuristics, and learns causal structures for iOOBN classes from a constructed hierarchy (note that some classes are the original input classes that

[1] Causal structure is the graphical structure of a BN. It is one of the vital components of a BN. The remaining of a BN are the parameters associated with the nodes and causal connections of the structure.

can be a class built by transforming a BN into the class, and some are inferred intermediate classes) with maximum reusability of the existing/previously formed classes. The proposed algorithm, however, proved suboptimal and deterministic. The efficiency of the algorithm is tested against synthetic (randomly generated) hierarchies by empirical analysis.

5.1 Hierarchy of OOBN Classes

The concept of Hierarchy provides a great deal of meaningful information about a system. Looking at the hierarchy of a system helps in understanding the whole system in a nutshell, the properties of its various components, what comes before what, the level of components and classes, and their similarities and differences. From the dawn of science, scientists have been constructing hierarchies for groups of things such as animals (the animal taxonomy of Aristotle), plants (plant taxonomy, i.e., Historia Plantarum by Theophrastus), objects, names, values and people. The idea of taxonomy has migrated from the biological and natural sciences to the contemporary sciences , for example, in Phylogenetic trees [218–222]; in Bio-informatics and Dendrograms in the agglomerative clustering [223] used in Data Science. Inheritance and hierarchy cover a vast area of computer science, especially software development, through the use of object-orientation (OO).

OO and its features, especially inheritance, provide various facilities in developing large complex systems. Inheritance reduces computation, reduces the effort to build a class by enabling reuse of existing resources (classes). It suggests a hierarchical structure where subclasses derived from a class are placed as children of that class. However, hierarchy and inheritance come with some inherent problems: for example, the Yo-Yo [2] problem [224], imperfect construction of hierarchy, inappropriate use of inheritance, and inconsistencies that occur when a group of people develop a system together.

Manually designing inheritance hierarchies to maximise factoring is very difficult, particularly if the system is large and has been built by different people. Even though a system may be well designed initially, maintenance and evolution cause the hierarchy to degrade to below standard [187]. To address these issues and resolve them, researchers have developed various techniques such as the evolution of Inheritance hierarchy [187], the automatic restructuring of hierarchy [188], the automatic inferring of inheritance [188–195], to mention some common in the software development arena.

Motivated from software development, the iOOBN framework (proposed in Chapter 3), for the first time allows the use of inheritance in an OOBN by constructing an inheritance

[2]The Yo-Yo problem is a special issue of the OO-paradigm where a designer has to work with a class whose inheritance graph is very long and complicated. This requires the designer to keep flipping between many different class definitions. One solution is to make class hierarchies shallow.

hierarchy of classes. Certainly, there is a very high risk that any iOOBN system will suffer from the problems related to inheritance, outlined above. On the other hand, restructuring an already built hierarchy and automated learning-building of hierarchies makes systems more compact, improves consistency, and helps maintain the system with less effort and expense [187]. Although there has been much interesting work done on causal structure learning for BNs from data [150, 152, 225, 226], to the best of our knowledge, for OOBNs, there has been no work on the automated learning of hierarchies or restructuring a hierarchy.

The merits of inheritance in the OO paradigm, the importance of hierarchy in better-utilizing inheritance and managing OO-systems with inheritance, automated learning and the restructuring of a previously built OO-system played an important role in designing and conducting this research. Since iOOBNs support all the features of OOBNs along with some extra facilities, this chapter describes the proposed algorithm of learning iOOBN class structures from a set of BNs or a set of OOBN classes. The proposed approach converts a set of BNs to some form of OOBN classes. The conversion is only done by marking interface nodes using the basic properties of the OOBN input and output nodes. Finally, a hierarchy of the OOBN classes is constructed making it possible to learn iOOBN classes from this hierarchy.

5.2 Terminology

This section contains some relevant terminologies and notations used to present the proposed learning algorithm.

Any BN, OOBN class and iOOBN class, contains a DAG as one of its main components. As discussed in the BN literature, the DAG is a graphical structure required to define the dependency between standard (chance, decision and utility) nodes. OOBN classes and iOOBN classes may also contain complex instance nodes that are replicas of another class. Hence, at its deepest level, the whole class can be represented by a DAG with standard nodes and complex instance nodes. When flattening is performed, the resultant class is a simple DAG with only standard nodes and standard edges (causal edges, information and precedence links).

DEFINITION 5.1 : SUBGRAPH AND SUPERGRAPH

A graph $G' =< V', E' >$ is a **Supergraph** of another graph $G =< V, E >$, iff $V \subseteq V'$ and $E \subseteq E'$ and denoted as $G \subseteq G'$. Note that G is called a subgraph of G'.

The DAGs also have a superDAG and sub-DAG relationship with the same properties as a subgraph and a supergraph (see Definition 5.1). Figure 5.1 shows an example supergraph and an example subgraph of a DAG.

In iOOBN, defining subclasses and the superclass of a class is allowed. Consequently,

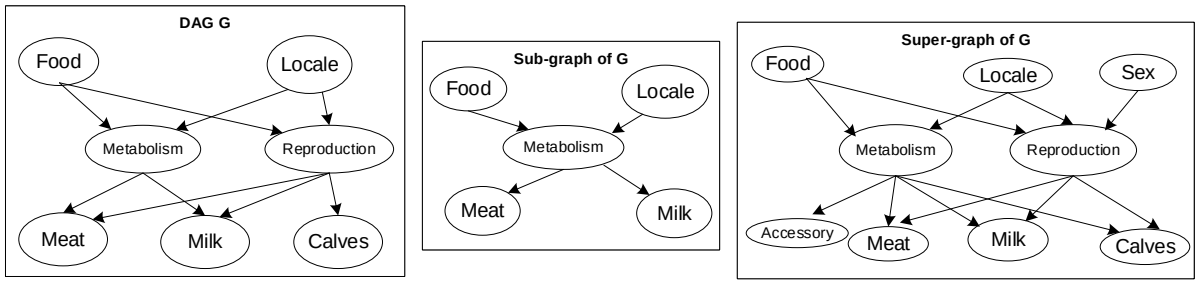


Figure 5.1: An example supergraph and subgraph.

the subclass DAG and superclass DAG have a supergraph and subgraph relationship with the DAG of the class in consideration. To check the subgraph and supergraph relationships between graphs, isomorphism testing (an NP-complete task) needs to be done.

DEFINITION 5.2 : ISOMORPHISM

A graph $G =< V, E >$ is **isomorphic to a subgraph** [227] of a graph $G' =< V', E' >$ denoted by $G \cong S' \subseteq G'$, if there exists a one-to-one function $\varphi : V \rightarrow V'$ such that, for every pair of vertices $v_i, v_j \in V$, if $(v_i \rightarrow v_j) \in E$ then $(\varphi(n_i) \rightarrow \varphi(n_j)) \in E'$.

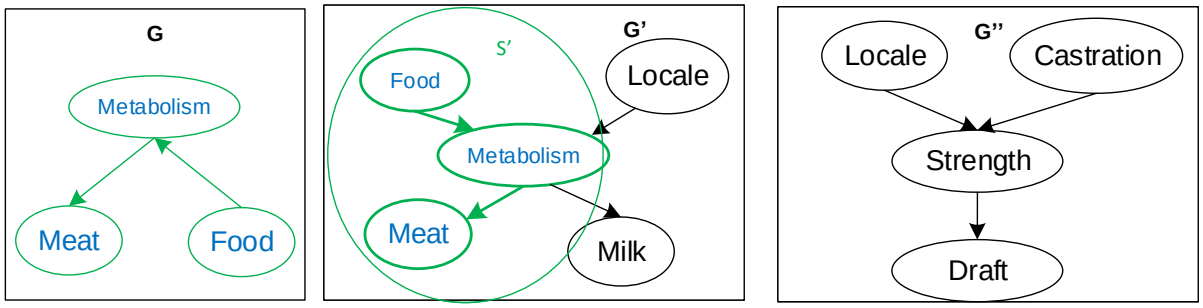


Figure 5.2: Example of subgraph isomorphism and non-isomorphism: G is a isomorphic subgraph of G' (since $G = S'$ and $S' \subset G'$) and not a isomorphic subgraph of G'' (Since, no subgraph of G'' is equal to G).

Figure 5.2 demonstrates subgraph isomorphism with examples. The DAG G shown in the first slot is contained in the graph G' shown in the next slot. Hence, G is an isomorphic subgraph of G'. In the last slot, another graph G'' is shown, which does not contain G as a subgraph. Therefore, G is not an isomorphic subgraph of G''.

DEFINITION 5.3 : COMMON SUBGRAPH

For a set of N graphs $\mathbb{G} = \{G_1, G_2, ..., G_N\}$ where each graph $G_i =< V_i, E_i > \in \mathbb{G}$. The **common subgraph** $G_{com} =< V_{com}, E_{com} >$ of the set of graphs $\mathbb{G}$ is a graph such that $G_{com} \subseteq G_i$ for $1 \leq i \leq N$. $V_{com} \subset V_1 \bigcap V_2 \bigcap ... \bigcap V_N$ and $E_{com} \subset E_1 \bigcap E_2 \bigcap ... \bigcap E_N$.

DEFINITION 5.4 : RESIDUAL GRAPH

Given a set of N graphs $\mathbb{G} = \{G_1, G_2, ..., G_N\}$ where $G_i =< V_i, E_i >$ where $i = 1, 2, ..., N$. Let $G_{com} =< V_{com}, E_{com} >$ be the common subgraph of the set of graphs $\mathbb{G}$. The graph $G_i^r = G_i \setminus G_{com}$ is a **residual graph** of G_i with respect to G_{com} where $G_i^r =< V_i^r, E_i^r >$, $V_i^r = V_i \setminus V_{com}$ and $E_i^r = E_i \setminus E_{com}$.

In the proposed algorithm, all the nodes and edges in a graph are labelled with a unique label associated with the graph. Hence, for a given graph, $G =< V, E >$, where V is a set of **Nodes** and E is a set of **Edges**, each node $v \in V$ and each edge $e \in E$ has a label $lab(G)$. The associated label for a graph G can be obtained by a function, $lab(G)$, that returns a unique label for that graph.

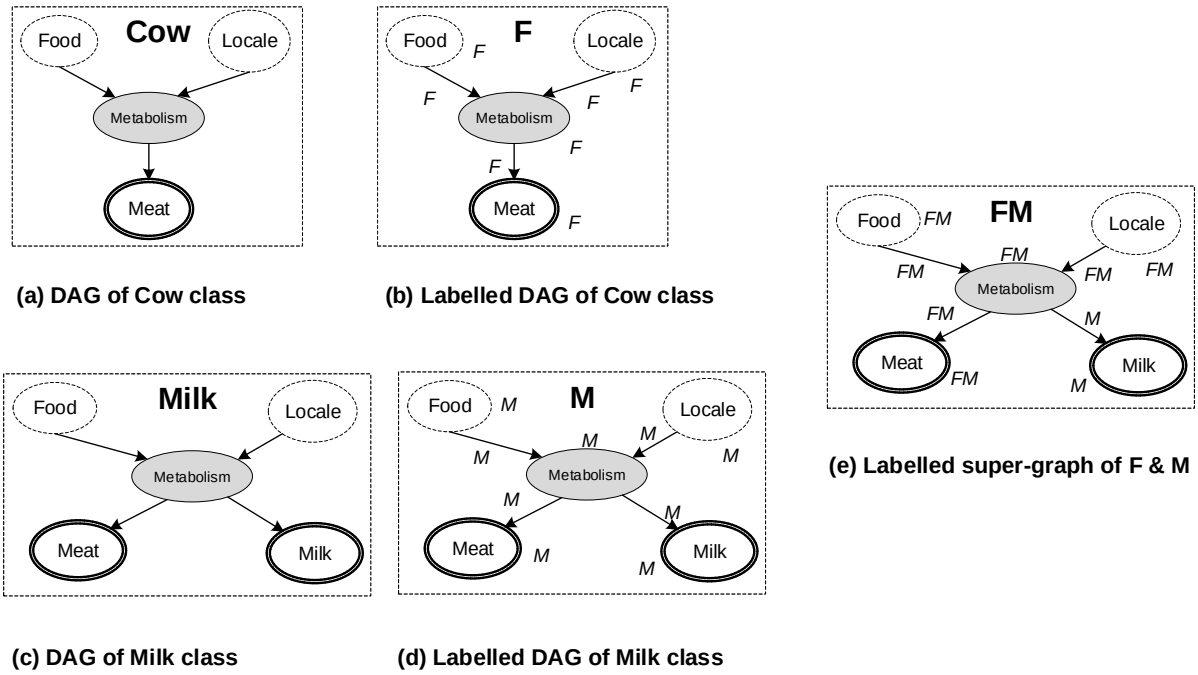


Figure 5.3: (a) A DAG of an OOBN class "Cow", (b) Labelled DAG of Cow class assuming the label of Cow class is "F", (c) A DAG of an OOBN class "Milk", (d) Labelled DAG of Milk class assuming the label of Milk class is "M", (e) Labelled superDAG of labelled "Cow" and "Milk" DAG

As an example, in Figure 5.3, (a) and (c) show the DAGs for "Cow" and "Milk Cow" classes, respectively; (b) and (d) show the labelled version of the DAGs in (a) and (c), respectively. Part (e) depicts a labelled supergraph of the graphs in (b) and (d).

The proposed algorithm forms a supergraph using all the graphs (DAGs) given as input. To keep track of which node and which edge belongs to which DAG, each node and edge of a DAG in the input is labelled with a unique label of the graph.

DEFINITION 5.5 : MAXIMAL PROPER SUBSET

Let $\mathbb{S}$ be a set of subsets. $L_c \in \mathbb{S}$ is a **maximal proper subset** of $L \in \mathbb{S}$ iff $L_c \subset L$ and there exists no label $L' \in \mathbb{S}$ where $L_c \subset L'$ and $L' \subset L$.

As an example, if the set of subsets $\mathbb{S} = \{abcde, abc, ab, a, de, cde\}$, then abc and cde are two **Maximal proper subset**s of $abcde$.

DEFINITION 5.6 : MAXIMAL COMMON SUBGRAPH

For a set of N graphs $\mathbb{G} = \{G_1, G_2, ..., G_N\}$ where each graph $G_i =< V_i, E_i > \in \mathbb{G}$. The **maximal common subgraph** $G_{mc} =< V_{mc}, E_{mc} >$ of the set of graphs $\mathbb{G}$ is a graph such that $G_{mc} \subseteq G_i$ for $1 \leq i \leq N$. $V_{mc} = V_1 \bigcap V_2 \bigcap ... \bigcap V_N$ and $E_{mc} = E_1 \bigcap E_2 \bigcap ... \bigcap E_N$.

DEFINITION 5.7 : HIERARCHY TREE

$\top = < \mathcal{N}, \mathcal{E} >$ is a **Hierarchy Tree** where $\mathcal{N}$ is a set of DAGs and $\mathcal{E}$ is a set of edges between the DAGs such that

1. $\top$ is a tree
2. For every edge $(D' \rightarrow D) \in \mathcal{E}$, $D' \subset D$ where $D, D' \in \mathcal{N}$

DEFINITION 5.8 : LABEL HIERARCHY TREE

Label Hierarchy Tree $\mathcal{T} = < N, E >$ is a 2-tuple, where N is a set of subsets of labels, E is a set of **Edges**, and for each edge $(u \rightarrow v) \in E$, where $u, v \in N$, v is a maximal proper subset of u. As $\mathcal{T}$ is a tree, u is the parent of v and denoted as $u = \mathcal{P}(v)$.

The proposed algorithm constructs a labelled hierarchy tree where each node in the tree has a unique label. A node is said to be a parent of another node (if there is any) if the label of the latter node is a maximal proper subset of the former one. The algorithm also constructs a hierarchy tree where each node of the tree contains a DAG, and the DAG is the maximal common subgraph of its children DAGs.

Following are some terms defined for finding an optimized hierarchy tree that maximises reusability and minimises construction cost. Although the main terms are "reusability" and "construction cost", other terms are required to compute them. Some of the terms, defined in Section 5.5.2, are used for evaluating the efficacy of a constructed hierarchy. Those terms are extensions of the following terms, obtained by necessary modifications for comparing various hierarchy trees.

DEFINITION 5.9 : DERIVING A GRAPH

Given a hierarchy tree, $\top = < \mathcal{N}, \mathcal{E} >$ (Definition 5.7) and a DAG D, D is **derived** from $\top$ if there exists $D' \in \mathcal{N}$ such that $D' \subseteq D$. In other words, D can be constructed by adding some nodes or edges to D'.

DEFINITION 5.10 : SIZE OF A GRAPH

Let $G =< V, E >$ be a graph where V is the set of nodes and E is the set of edges. The size of the graph G is the sum of the number of nodes and number of edges in G and denoted as $|G| = |V| + |E|$.

DEFINITION 5.11 : DERIVATION COST

Given a hierarchy tree, $\top = < \mathcal{N}, \mathcal{E} >$ (Definition 5.7), a graph G is derived from $\top$ and $D \subseteq$ G where D is a DAG and $D \in \mathcal{N}$.

The **Derivation cost** of G with respect to the DAG D is

$$\delta(G, D) = |G| - |D|$$

The **derivation cost** of G with respect to the hierarchy tree $\top$ is,

$$\delta_T(G, \top) = \min_{D \in \mathcal{N}} \delta(G, D)$$

DEFINITION 5.12 : ADDING A CHILD DAG TO A HIERARCHY

Let $\top = < \mathcal{N}, \mathcal{E} >$ be a hierarchy tree constructed by the proposed learning algorithm. **Child-adding** is the procedure of adding a DAG C to the hierarchy tree before adding C, i.e., $\top' = < \mathcal{N} \setminus C, \mathcal{E} \setminus e >$ in order to construct $\top$, where $e = u \to C$, $u \in \mathcal{N} \setminus C$.

DEFINITION 5.13 : CHILD-ADDING COST

Let a hierarchy tree $\top = < \mathcal{N}, \mathcal{E} >$ be obtained by adding a child DAG C (Definition 5.12). The **child-adding cost** of C, to obtain $\top$, is the derivation cost of C with respect to $\top'$ ($\top$ before adding C). It is denoted as,

$$\chi(C, \top) = \min_{D \in \mathcal{N}'} \delta(C, D)$$

DEFINITION 5.14 : HIERARCHY CONSTRUCTION COST

Let, $\top = < \mathcal{N}, \mathcal{E} >$ be a hierarchy tree. The **construction cost** of $\top$ is,

$$Cost(\top) = \sum_{D \epsilon \mathcal{N}} \chi(D, \top)$$

In other words, starting with an empty hierarchy tree, the cost required to add all the DAGs needed in order to construct the hierarchy tree is called the **hierarchy construction cost**.

As an example, in a hierarchy tree, constructed by the proposed learning algorithm, the root graph, say G, has a child-adding cost of $|G|$. That is, all the nodes and edges of G need to be added in the hierarchy tree. Suppose another graph G' is added as a child node of G in the hierarchy tree, then the child-adding cost of G' is $|G' - G|$.

One of the prime reasons for adapting OO-features is the reuse of existing components. Inheritance provides maximal reuse if classes are extended in a well-organized manner. Inheritance allows constructing a hierarchy of a set of classes based on the subgraph–supergraph relationships among the classes. Assessing such hierarchies can be performed based on the number of reused components in the overall hierarchy. In Definition 5.15, such a measure is defined.

DEFINITION 5.15 : REUSABILITY

For a hierarchy tree, $\top = < \mathcal{N}, \mathcal{E} >$ (Definition 5.7), the **reusability** in deriving the set of DAGs $\mathbb{G} = \{G_1, G_2, ..., G_N\}$ is,

$$\rho(\top, \mathbb{G}) = \sum_{i=1}^{N} |G_i| - \left(Cost(\top) + \sum_{i=1}^{N} \delta_T(G_i, \top) \right)$$

DEFINITION 5.16 : REUSABILITY OF HIERARCHY FOREST

Given a set of DAGs $\mathbb{G} = \{G_1, G_2, ..., G_N\}$ and a set of z hierarchy trees (aka a hierarchy forest), $HF = \{\top_1, \top_2, ..., \top_z\}$, constructed using the proposed learning algorithm, the **reusability of** HF is,

$$\rho(HF) = \sum_{i=1}^{z} \rho(\top_i, \mathbb{G}_i)$$

where $\mathbb{G}_i$ is a group of graphs obtained from $\mathbb{G}$ having a non-empty maximal common subgraph, and $\mathbb{G}_i \subseteq \mathbb{G}$.

Measures of reusability, derivation cost and hierarchy tree construction cost are extended in Definition 5.19 to compare the performance of the proposed algorithm with a synthetic

hierarchy. The ratio is taken to get a normalized derivation and construction cost. This measure plays a significant role in providing a fair comparison ground for evaluating and comparing the proposed hierarchy learning algorithm with synthetic hierarchies.

5.3 Converting BNs to OOBN Classes

If the target is that of learning OOBN or iOOBN classes from data using the proposed learning algorithm, a set of BNs from some datasets needs to be learned using any of the existing causal structure learning approaches, such as CaMML [150]. Then the set of BNs needs to be converted to a set of OOBN classes. Some simple heuristics have been used to convert BNs into OOBN classes.

Heuristic 3. *A set of nodes in a BN, having no parent nodes, forms a set of input nodes in the converted OOBN class.*

ALGORITHM 5.1 (FIND INPUT NODES)

Call : $FindInputNodes(DAG) \rightarrow InputNodes$
Input: DAG: a Directed Acyclic Graph
Output: $InputNodes$: a set of input nodes

```
1 begin
2     InputNodes ← ∅
3     V ← NodesInDAG(DAG)
4     // Heuristic : node with no parent is an input node
5     foreach v ∈ V do
6         if v.par == ∅ then
7             InputNodes ← InputNodes ⋃ v
8     return InputNodes
```

Algorithm 5.1 finds a set of potential nodes using Heuristic 3: those could be marked as input nodes in the converted OOBN class without breaking the properties of an OOBN class.

Heuristic 4. *A set of nodes in a BN, having no child nodes, forms a set of output nodes in the converted OOBN class.*

Heuristic 4 is used in Algorithm 5.2 to find a set of potential nodes of a BN to use them as output nodes in the converted OOBN class, and it does not violate any of the properties of OOBN classes.

ALGORITHM 5.2 (FIND OUTPUT NODES)

Call : $FindOutputNodes(DAG) \rightarrow OutputNodes$
Input: DAG: a Directed Acyclic Graph
Output: $OutputNodes$: a set of Output nodes

```
1 begin
2     OutputNodes ← ∅
3     V ← NodesInDAG(DAG)
4     foreach v ∈ V do
5         // Heuristic : node with no child is an output node
6         if v.child == ∅ then
7             OutputNodes ← OutputNodes ⋃ v
8     return OutputNodes
```

Finally, Algorithm 5.3 converts a set of BNs to a set of OOBN classes by marking the input and output nodes of each BN using Algorithm 5.1 and Algorithm 5.2. The set of OOBNs, found from a set of BNs using Algorithm 5.3, could be used to build a hierarchy tree and learn iOOBN structures.

Note that Heuristic 4 has some limitations if applied in real-life applications. In particular, from the definition of OOBNs and iOOBNs, it is quite obvious that a node that has no children nodes in an OOBN may not be an output node. Even in some real-life real-world problems considered in this thesis (e.g., in the DOOBNs of the WGR [3, 57] project and the example OOBNs revisited in Section 3.4 of Chapter 3), this limitation of Heuristic 4 is evident.

ALGORITHM 5.3 (CONVERTING BNS TO OOBNS)

Call : $ConvertBNsToOOBNs(BNs) \rightarrow OOBNStrucs$
Input: BNs: a set of BNs
Output: $OOBNStrucs$: a set of OOBN class structures

```
1 begin
2     DAGs ← extractDAGs(BNs)
3     foreach DAG ∈ DAGs do
4         OOBNStruc.In ← FindInputNodes(DAG)
5         OOBNStruc.Out ← FindOutputNodes(DAG)
6         OOBNStrucs ← OOBNStrucs ⋃ OOBNStruc
7     return OOBNStrucs
```

5.4 A Method for Learning an iOOBN Class Hierarchy from a Set of OOBN Classes

The proposed class hierarchy learning method consists of the following three steps:

1. **Construct a supergraph:** Taking a set of OOBN classes as input, extract a set of DAGs, then build a supergraph by amalgamating all the DAGs.
2. **Construct a label-hierarchy tree:** a tree that represents multiple alternative consistent OOBN class hierarchies, constructed from the set of unique labels of the nodes and edges of the supergraph.
3. **Construct an OOBN class hierarchy:** by traversing the label-hierarchy tree, construct a class hierarchy using the derivation cost to choose which class to extend.

In the following subsections, each of the above-mentioned steps are described in some detail and algorithms are presented in detail with pseudo code for all the steps. The mechanism of the algorithm is also illustrated throughout with a simple example (the Cow example, as used in [201]), and illustrated in Figure 5.6.

5.4.1 Step 1: Construction of supergraph from a set of OOBN classes

Let there be a set of t input OOBN classes $\mathbb{C}_1, \mathbb{C}_2, ..., \mathbb{C}_t$ where $\mathbb{C}_i =< \mathbb{N}_i, \mathbb{O}_i, \mathbb{E}_i, \Pi_i >$ and each node $n_j \in \mathbb{N}_i$, each object $o_k \in \mathbb{O}_i$, each edge $e_l \in \mathbb{E}_i$ has the label $lab(\mathbb{C}_i)$, i.e., $lab(\mathbb{C}_i)$ is considered as a unique identifier for $\mathbb{C}_i$ and the DAG associated with $\mathbb{C}_i$. This identifier can be a any valid string such as number, letter, or name.

ALGORITHM 5.4 (CONSTRUCTION OF A SUPERGRAPH OF A SET DAGS (STEP 1))

Call : $Construct\_SuperGraph(dAGSet) \rightarrow G_s$
Input: $dAGSet$: a set of DAGs, $DAG_1, DAG_2, ..., DAG_t$
Output: G_s: A supergraph

```
1 begin
2     // Construct a supergraph
3     G_s ← < ∅, ∅ >
4     foreach DAG ∈ dAGSet do
5         G_s ← Accumulate(G_s, DAG)
6     return G_s
```

A supergraph is constructed from the set of t DAGs of the aforementioned OOBN classes. Each of the nodes, objects and edges of the supergraph contain a set of labels of the classes

that contain the node, the object or the edge. The nodes and objects of a class form the set of vertices for the DAG and the edges of the class constitute the edges of the DAG. Algorithm 5.4 takes a set of OOBN classes as input and constructs a supergraph by amalgamating the DAGs of the classes, one by one, using Algorithm 5.5.

ALGORITHM 5.5 (ACCUMULATE DAG TO SUPERGRAPH)

Call : $Accumulate(G_s, DAG) \rightarrow G_s$

Input: $DAG =< V, E >$: a set of labelled nodes and a set of labelled directed non-cyclic edges

$G_s =< \mathbf{V}_s, \mathbf{E}_s >$: a supergraph

Output: $G_s =< \mathbf{V}_s, \mathbf{E}_s >$: the accumulated supergraph

```
1  begin
2      foreach v ∈ V do
3          flagFound ← False
4          foreach v' ∈ V_s do
5              if v == v' then
6                  v'.label ← v'.label ⋃ v.label
7                  flagFound ← True
8                  break
9          if flagFound == False then
10             V_s ← V_s ⋃ v
11     foreach e ∈ E do
12         flagFound ← False
13         foreach e' ∈ E_s do
14             if e == e' then
15                 e'.label ← e'.label ⋃ e.label
16                 flagFound ← True
17                 break
18         if flagFound == False then
19             E_s ← E_s ⋃ e
20     return G_s
```

To be more specific, a supergraph is constructed by accumulating each DAG of the classes. Initially, it begins with an empty supergraph, $G_s = < N_s, E_s >$ where $N_s = \varnothing, E_s = \varnothing$. Then for the DAG, G_1 associated with $\mathbb{C}_1$ is added in G_s. The resulting graph $G_s =< N_s, E_s >$ where $N_s = N_1, E_s = E_1$, and each node in N_s and each edge in E_s has a label 1. Now, for $G_2 =< N_2, E_2 >$ of $\mathbb{C}_2$, each node of N_2 and each edge of E_2 is added in G_s. If any

node or edge does not exist in G_s, they are added with label 2; otherwise, label '2' is appended with the previous label of the nodes and edges. This results in a supergraph, $G_s =< N_1 \bigcup N_2, E_1 \bigcup E_2 >$. In the same way, all the DAGs are added which then turns into a supergraph, $G_s =< N_1 \bigcup N_2 \bigcup ... \bigcup N_t, E_1 \bigcup E_2 \bigcup ... \bigcup E_t >$.

5.4.2 Step 2: Construction of a hierarchy tree

The second step in the method involves constructing a hierarchy tree that represents the subgraph–supergraph relationship among the underlying DAGs of a set of OOBN classes. Checking for the subgraph requires a subgraph isomorphism check, which is an asymptotically complex and expensive task. Fortunately, in any OOBN class, the node names, object names, and the edge labels are unique. This fact simplifies the subgraph isomorphism checking and makes it a linear-time operation.

This supergraph is used in finding the hierarchy of the OOBNs. The edges and nodes, having the same set of labels, form a particular DAG of an OOBN class. Another OOBN class is called a child class of the former OOBN class if formed by edges and nodes having a subset of the former label.

Hence, to construct the hierarchy, the unique sets of labels are collected from the supergraph's nodes and edges (as shown in Algorithm 5.6). The set of labels is used to construct a tree structure where each node is a set of labels; its ancestor is a superset of the label, and its descendants are subsets of the label. In Algorithm 5.8, a set of labels is taken as input. The labels are lexicographically sorted and mapped by their length. In the map, all size one sets are put into a list mapped with 1; size two sets are mapped with 2 and so on. This procedure is shown in Algorithm 5.7.

The map facilitates the construction of the hierarchy smoothly and efficiently because the search space of superset–subset finding is clustered and subdivided. For a set of size i (starting with 1), in order to find its parent in the hierarchy tree, all sets of size $i+1$ to n are checked as to whether there exists any set for which the size i set is a subset. The algorithm stops as soon as it obtains an aforementioned superset. If such a set is not found, an empty set $\varnothing$ is assigned as the parent of the set. A set of such nodes with a maximal proper subset and superset relationship inherently captures a tree structure rooted at $\varnothing$.

ALGORITHM 5.6 (EXTRACTING THE SET OF LABELS)

Call : $Extract\_LabelSet(G_s) \rightarrow labSet$

Input: G_s: a supergraph

Output: $labSet$: a set of labels

```
1  begin
2      labSet ← ∅
3      V_s ← NodesInDAG(G_s)
4      E_s ← EdgesInDAG(G_s)
5      foreach node v ∈ V_s do
6          l ← v.label
7          if l ∉ labSet then
8              labSet ← Add(labSet, l)
9      foreach edge e ∈ E_s do
10         l ← e.label
11         if l ∉ labSet then
12             labSet ← Add(labSet, l)
13     labSet ← Lexicographical_Sort(labSet)
14     return labSet
```

ALGORITHM 5.7 (MAP LABELS TO SIZE)

Call : $MapLabelSetsToSize(labSet) \rightarrow MAP < Size, LabelSets >$

Input: $labSet$: a set of subsets of labels

Output: $MAP < Size, LabelSets >$: A data structure that maps a list of labels w.r.t. their size

```
1  begin
2      Map < Size, LabelSets > ← < ∅, {} >
3      foreach label ∈ labSet do
4          s ← size(label)
5          if s ∈ Map.keys() then
6              Map[s] ← Map[s] ⋃ label
7          else
8              Map ← Map ⋃ < s, label >
9      return Map < Size, LabelSets >
```

ALGORITHM 5.8 (LABEL HIERARCHY CONSTRUCTION (STEP 2))

Call : $Construct\_Label\_Hierarchy(labSet) \rightarrow \mathcal{T}$

Input: $labSet$: a set of subset of labels

Output: $\mathcal{T}$: A label hierarchy tree, $\mathcal{T} = < V, E >$

```
1  begin
2      Map < Size, LabSets > = MapLabelSetsToSize(labSet)
3      T = < V = ∅, E = ∅ >
4      // Assuming the map has keys in increasing order
5      minS = min(Map.keys())
6      maxS = max(Map.keys())
7      for (cSize = minS; cSize ≤ maxS; cSize ++) do
8          if cSize ∈ Map.keys() then
9              cSizeLabSet = Map.get(cSize)
10             foreach lab ∈ cSizeLabSet do
11                 T.V = T.addNode(lab)
12                 for (nSize = cSize+1; nSize ≤ maxS; nSize++) do
13                     if nSize ∈ Map.keys() then
14                         nSizeLabSet = Map.get(nSize)
15                         flagMPS = False
16                         foreach nLab ∈ nSizeLabSet do
17                             if lab ⊂ nLab then
18                                 T.V = T.addNode(nLab)
19                                 T.E = T.addEdge(nLab → lab)
20                                 flagMPS = True
21                         if flagMPS == True then
22                             break
23     return T
```

5.4.3 Step 3: Constructing an iOOBN class hierarchy from the hierarchy tree

In this step, a method is presented for extracting a consistent iOOBN class hierarchy from the hierarchy tree, where each class in that hierarchy is either one of the original OOBN classes, or it is a new inferred class.

Algorithm 5.9 constructs a class hierarchy tree (a single-parent hierarchy tree) from the label hierarchy tree (a multi-parent hierarchy tree) by considering maximum reuse (minimum construction and derivation cost). Note that in the iOOBN framework, as proposed in Chapter 3, multiple class inheritance is not allowed. Hence, a single, best inheritance with maximum reuse (and the minimum derivation cost , $\delta_T(G, \top)$) is chosen here. It uses the formula given in Definition 5.15 to calculate reusability and only keeps a parent with the maximum reusability among all parents of any node in the multi-parent tree.

ALGORITHM 5.9 (CLASS HIERARCHY CONSTRUCTION (STEP 3))

Call : $Construct\_Hierarchy\_SingleParent(\mathcal{T}) \rightarrow HT$
Input: HT: a multi-parent label hierarchy, $\mathcal{T} = < V, E >$
Input: G_s: a supergraph of a set of graphs
Output: HT: A hierarchy tree, $\top = < \mathcal{N}, \mathcal{E} >$

```
1  begin
2      V = DFS_ordered_Nodes(V)
3      HT ← < N = ∅, E = ∅, L = V >
4      foreach node v ∈ V do
5          minDeriveCost ← 0
6          minCostParent ← ∅
7          chDAG ← FormOODAG(G_s, v)
8          HT.N ← HT.addNode(chDAG)
9          foreach edge (u → v) ∈ E do
10             parDAG ← FormOODAG(G_s, u)
11             derivCost ← δ(parDAG, chDAG)
12             if minDeriveCost > derivCost then
13                 HT.E = HT.removeEdge(minCostParent → v) // remove prev. max
                       edge
14                 minDeriveCost ← derivCost
15                 minCostParent ← u
16                 HT.E = HT.addEdge(u → v) // add current max edge
17     return HT
```

An OOBN class $\mathbb{C}$ is constructed from each of the nodes of the label hierarchy tree. It is a DAG derived from the supergraph, G_s, say with label L. Any class $\mathbb{C}'$, derived from G_s using a set of labels $L' \subset L$ which is found in one of the descendant nodes of $\mathcal{N}_{desc} > \mathcal{N}$ in the tree,

is a subclass of $\mathbb{C}$, denoted as $\mathbb{C}' \subset \mathbb{C}$. Algorithm 5.10 depicts all the actions required to derive a DAG from the supergraph for a particular label set L obtained from the hierarchy tree. It derives a DAG for any node in the hierarchy tree with label set L by finding the nodes and edges in the supergraph with a label set of L or a superset of L. These DAGs are used to form OOBN classes later. For each of the nodes of the hierarchy tree, an OOBN class is constructed using Algorithm 5.11. This algorithm traverses the hierarchy tree $\top$ in a DFS (Depth-First Search) manner. While traversing nodes in each level of the tree, it forms DAGs for the children nodes of the nodes in the current level. This strategy makes future reuse of components easy.

ALGORITHM 5.10 (FORM OODAG)

Call : $FormOODAG(G_s, L) \rightarrow OODAG$
Input: G_s: a supergraph of a set of BNs
Input: L: label
Output: $OODAG$: an OODAG

```
1  begin
2      OON ← ∅
3      OOE ← ∅
4      V_s ← NodesInDAG(G_s)
5      E_s ← EdgesInDAG(G_s)
6      foreach node v ∈ V_s do
7          if L ⊆ v.label then
8              OON ← OON ∪ v
9      foreach edge e ∈ E_s do
10         if L ⊆ e.label then
11             OOE ← OOE ∪ e
12     OODAG ← < OON, OOE >
13     return OODAG
```

In an iOOBN, extending subclasses from a class is allowed. As to which class should be used to derive/extend which class with maximum reusability, this kind of information can be easily derived while traversing the hierarchy tree to form OODAGs. If an OODAG is formed using a node in the tree, then the OODAG structure can be reused to form OODAGs for the children nodes of the node in the tree.

Algorithm 5.12 forms OOBN class structures for the DAGs constructed in the algorithms described previously. The formation of an OOBN structure is straightforward. It follows Heuristic 3 and Heuristic 4 in order to find nodes in DAGs without parent nodes or child

ALGORITHM 5.11 (CONSTRUCT ALL OODAGS)

Call : $Construct\_OODAGs(G_s, HT) \rightarrow OODAG$

Input: G_s: a supergraph of a set of BN

Input: HT: hierarchy tree, $< \mathcal{N}, \mathcal{E}, L >$

Output: $OODAGs$: a set of DAGs of OOBN classes

```
1  begin
2      // find root of Hierarchy Tree
3      n_root ← ∅
4      foreach n ∈ HT.N do
5          // root node has no parent
6          if P(n) == ∅ then
7              n_root ← n
8              break
9      Q_N ← n_root // traversing the tree by looking for child nodes of root
10     while Q_N ≠ ∅ do
11         n_root ← Q_N.serve()
12         foreach n ∈ HT.N do
13             if P(n) == n_root then
14                 oodag ← FormOODAG(G_s, L_n)
15                 OODAGs ← OODAGs ⋃ oodag
16     return OODAGs
```

ALGORITHM 5.12 (MAKING AN OOBN STRUCTURE)

Call : $Make\_OOBN\_Structure(DAGs) \rightarrow OOBNStrucs$

Input: $DAGs$: a set of DAGs of OOBN classes

Output: $OOBNStrucs$: a set of OOBN class structures

```
1  begin
2      foreach DAG ∈ DAGs do
3          tempStruc.In ← FindInputNodes(DAG)
4          tempStruc.Out ← FindOutputNodes(DAG)
5          OOBNStrucs ← OOBNStrucs ⋃ tempStruc
6      return OOBNStruc
```

nodes, then marks them as input and output nodes accordingly. Though these heuristics may not cover all the input and output nodes of a valid OOBN classes, more sophisticated heuristics and expert elicitation may help to work them out more effectively.

ALGORITHM 5.13 (CONSTRUCT DAGS FROM OOBN CLASSES)

Call : $constructDAGsOOBNs(OOBNs) \rightarrow dAGSet$
Input: $OOBNs$: a set of OOBN classes
Output: $DAGSet$: a set of DAGs

```
1  begin
2      DAGSet ← ∅                          // a set of DAGs
3      foreach class C ∈ OOBNs do
4          dag ← ∅                         // a DAG, dag =< V, E >
5          dag.v ← C.N ⋃ C.O               // C.N = set of nodes in C,
6                                          // C.O = a set of objects/instance nodes in C
7          dag.e ← C.E_c ⋃ C.E_r           // C.E_c = set of causal edges in C,
8                                          // C.E_r = a set of referential edgesin C
9          DAGSet ← DAGSet ⋃ dag
10     return DAGSet
```

ALGORITHM 5.14 (LEARNING IOOBN)

Call : $Learn\_iOOBN(BNs) \rightarrow OOBNs$
Input: BNs: a set of BNs
Output: $OOBNs$: a set of iOOBN classes

```
1  begin
2      OOBNs ← ConvertBNsToOOBNs(BNs)
3      dAGSet ← constructDAGsOOBNs(OOBNs)
4      G_s ← Construct_SuperGraph(dAGSet)
5      labSet ← Extract_labelSet(G_s)
6      HT ← Construct_Hierarchy(labSet)
7      OODAGs ← Construct_OODAGs(G_s, HT)
8      OOBNStruc ← Make_OOBN_Structure(OODAGs)
9      return OOBNs
```

Algorithm 5.14 describes all the elementary steps and flow of the whole iOOBN learning algorithm. If there is a set of BNs, then the whole procedure of the algorithm can be executed. If a set of OOBNs is available, the first line of the algorithm can be ignored, and the algorithm starts from constructing DAGs from a set of OOBNs using function "constructDAGsOOBNs()" that takes a set of OOBN classes and constructs a set of DAGs. This function considers the objects/instances as simple nodes, ignoring the complex and embedded structures in them. Indeed, this is a limitation of the current version of the algorithm. However, if an efficient

algorithm can be devised that can consider the embedded complex structures then replacing this function would mitigate the limitation. One such approach could be flattening the OOBNs first before using the classes as OOBN and then collapsing (an imaginary inverse operation of flattening) after the algorithm finishes execution. Along with the above pseudo codes, figures 5.4 and 5.5 show the flow diagrams of the proposed learning algorithm to explain the whole procedure.

Theorem 2. *Given a set of DAGs* $\mathbb{G} = \{G_1, G_2, ..., G_N\}$

1. *Algorithm 5.9, produces a Hierarchy Tree* $\top =< \mathcal{N}, \mathcal{E} >$
2. *Every* $G_i \in \mathbb{G}$ *is derived from* $\top$
3. *The root node of* $\top$ *is* $G_1 \bigcap G_2 \bigcap ... \bigcap G_N$
4. *For every node,* $D \in \mathcal{N}$ *there exists* $G_{i_1}, G_{i_2}, ..., G_{i_m}$ *such that* $D = \bigcap_{k=1}^{m} G_{i_k}$*, where* $i_k \in \{1, ..., N\}$ *and* $k = 1, ..., m$.

Proof: Suppose the learning algorithm takes a set of N DAGs $\mathbb{G} = \{G_1, G_2, ..., G_N\}$ and construct a hierarchy tree $\top =< \mathcal{N}, \mathcal{E} >$.

1. $\top$ has a set of nodes, and the nodes are connected via a set of edges. **There is no cycle in** $\top$ as the algorithm starts with a root node in level 0, and in level 1, it generates children nodes of the root node. In the next level (2^{nd} level), it finds the children nodes of these nodes and so on. Hence, the nodes expand to next level children nodes, and the tree grows downward. Hence, there is no chance to add a cycle. The algorithm chooses a single parent from a list of potential parents with the maximum similarity or minimum derivation cost. Thus, **each node has exactly one parent in** $\top$**, the exception being the root node that has 0 parents**.

2. All the input DAGs to the algorithm have at least one non-empty subgraph in $\top$ because from root to leaves, all the nodes in any level are constructed based on the maximal common subgraphs among a group of graphs in that level. Hence, by adding the required nodes and edges to a most suitable (least derivation cost) graph in the tree $\top$, any input graph can be derived.

3. The algorithm takes the maximal common subgraph of all input DAGs to find the root graph in the tree. If the maximal common graph is empty, then it finds z groups of graphs from N graphs. In each of the groups, there is at least a maximal common subgraph G_{mc} with $|G_{mc}| \geq 1$. These z maximal common subgraphs form z root nodes for the

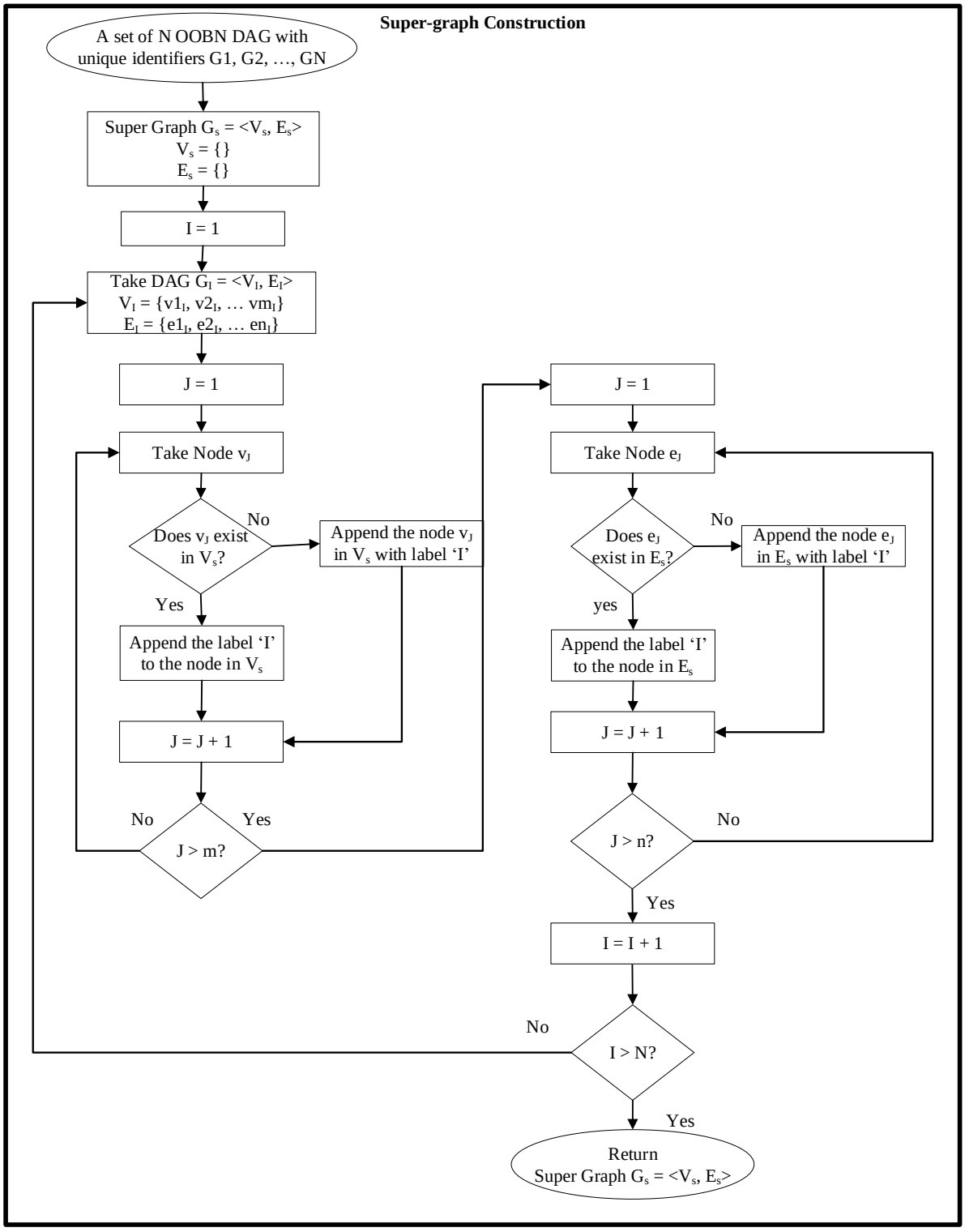


Figure 5.4: Flowchart of the supergraph construction technique

set of z hierarchy trees. If in a group, there are $1 \leq m \leq N$ number of graphs, then a maximal common subgraph for that group is $G_{mc} = \bigcap_{k=1}^{m} G_{i_k}$ where $i_k \in \{1, ..., N\}$ and $k = 1, ..., m$. Hence, if the N input graphs form a single root node then the maximal

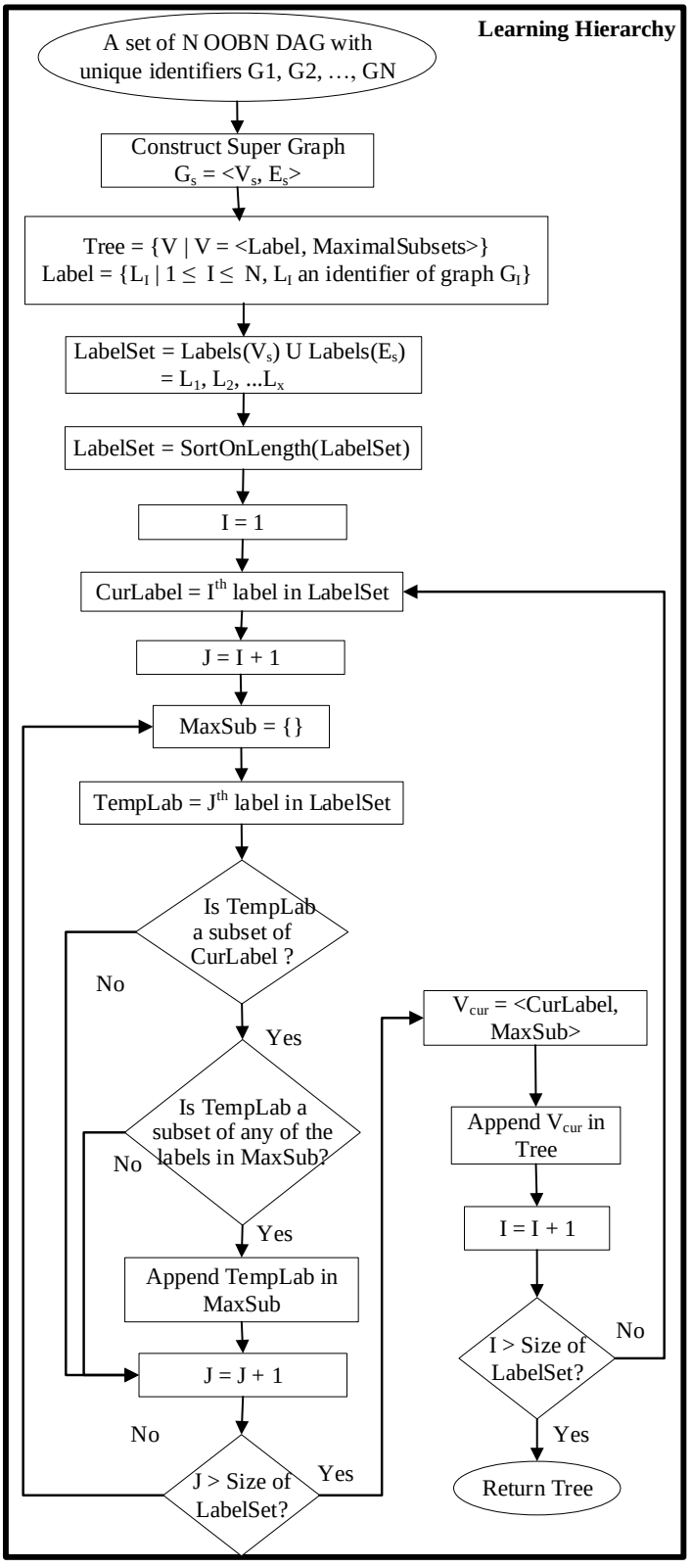


Figure 5.5: Flowchart of the proposed learning algorithm

common subgraph of the input graphs that represent the root node is $G_1 \bigcap G_2 \bigcap ... \bigcap G_N$.

4. Let's assume, there are V_{mc} nodes and E_{mc} edges maximal common among the set of input DAGs $\mathbb{G} = \{G_1, G_2, ..., G_N\}$. Therefore, the hierarchy tree has a root node and N children nodes which implies that the learned hierarchy has an extra DAG, a parent/root DAG, constructed from the V_{mc} common nodes and E_{mc} common edges, denoted with G_{mc}. In the N children DAGs, $G_i^r =< V_i^r, E_i^r >=< V_i - V_{mc}, E_i - E_{mc} >$ for each input DAG G_i where $1 \leq i \leq N$. Hence, to derive the original N DAGs using the hierarchy requires adding G_i^r with common DAG G_{mc}, i.e., $G_i = < V_{mc} + V_i^r, E_{mc} + E_i^r > = < V_{mc} + V_i - V_{mc}, E_{mc} + E_i - E_{mc} > = < V_i, E_i >$.

   Say the algorithm creates a set of residual graphs $\mathbb{G}^r = \{G_1^r, G_2^r, ..., G_N^r\}$ in level 1, then recursively finds maximal common subgraphs in the residual DAGs form z groups of graphs based on some common graphs in each of the groups. Say, the groups are $\mathbb{G}_1, \mathbb{G}_2, ..., \mathbb{G}_z$ with $G_{mc_1}, G_{mc_2}, ..., G_{mc_z}$, respectively, maximal common subgraphs. If they form a set of hierarchy trees $\top_1^r, \top_2^r, ..., \top_z^r$, their root node will be $G_{mc_1}, G_{mc_2}, ..., G_{mc_z}$, where $G_{mc_i} = \bigcap_{k=1}^{m} G_{i_k}$, $1 \leq i \leq z$, $i_k \in \{1, ..., N\}$ and $k = 1, ..., m$.
   The recursions terminate if the residual graph set, $\mathbb{G}^r$, is null or the algorithm has a single graph as input. Hence, from the third property, for every DAG, $D \in \mathcal{N}$, $G_{i_1}, G_{i_2}, ..., G_{i_m}$ where $D = \bigcap_{k=1}^{m} G_{i_k}$, and $i_k \in \{1, ..., N\}$ and $k = 1, ..., m$.

□

Lemma 1. *Let an algorithm "A" for learning hierarchy be optimal, and it produces a hierarchy tree "$\top^*$". Then the minimal derivation cost for a set of DAGs and the construction cost of "$\top^*$" is minimal.*

Proof: Let $\mathbb{G} = \{G_1, G_2, ..., G_N\}$ is a set of N DAGs given as input to the proposed hierarchy learning algorithm. $\top = < \mathcal{N}, \mathcal{E} >$ is the learned hierarchy from the input DAGs. $\top$, and hence the algorithm, is optimal if the reusability, $\rho(\top)$ is maximal.

According to Definition 5.15,

$$\rho(\top, \mathbb{G}) = \sum_{i=1}^{N} |G_i| - \left(Cost(\top) + \sum_{i=1}^{N} \delta_T(G_i, \top) \right)$$

From the equation above, reusability increases if the $Cost(\top)$ and derivation cost of all the input DAGs decreases. In other words, reusability is maximal if the tree construction cost and input DAG derivation costs are minimal.

1. **Minimal tree construction cost:** An algorithm "A" that produces a total order of the input graphs, where all input graphs have a mutual subgraph–supergraph relationship, constructs a hierarchy tree with minimum cost. If such a relationship does not exist

among the input graphs, then some minimal number of intermediate DAGs are inferred by the algorithm, and those along with the input DAGs can form the aforementioned total order. The hierarchy tree constructed using the input DAGs and the intermediate DAGs (if there are any) incur a minimal cost.

2. **Minimal derivation cost:** The more the input DAGs are similar to the DAGs contained in the hierarchy tree nodes, the lower is the derivation cost. The lowest possible derivation cost of a set of DAGs from a hierarchy tree is 0. This condition is only valid in the case where all input DAGs are included in the set of nodes in the hierarchy tree.

Therefore, any algorithm "A" that produces a hierarchy tree with minimal cost and derives all the input DAGs from the hierarchy with minimal derivation cost is optimal. □

Theorem 3. *The proposed hierarchy learning algorithm is suboptimal.*

Proof: The proposed learning algorithm tries to find the best possible hierarchy tree with minimal construction cost. The smallest non-empty common graph among the input DAGs is the root node in the hierarchy. Say, the node is present in all N DAGs. In the learning algorithm, this node is used only once and $N-1$ times this is reused. Similarly, for each level of the tree, each node is constructed based on the similarity of the graphs where, for any number of graphs having the same similar graph segment, the segment is used only once in the tree. This technique ensures maximum reuse and minimal derivation and construction cost.

The hierarchy tree T was constructed by checking the maximum similarity among the graphs in every level using the maximal proper subset and maximal common subgraphs. Based on the similarity found among the DAGs, the algorithm made several clusters and recursively performed the same tasks, like make a root of a subtree using the maximal common subgraph, find residual graphs, and so on, until the system finally produced it ended up with a cluster of size one or a set of residual graphs having no maximal common subgraph in them except an empty graph. Furthermore, while adding a child graph in the tree, if there were multiple options, the algorithm added the child graph to a node which entailed the minimum child-adding cost.

This attempt by the proposed algorithm is not guaranteed to be safe from being stuck in local maxima, as it is a greedy-like algorithm. An overall optimal tree could be constructed by choosing the optimal extension option at each step. However, this is not guaranteed in the case of the proposed tree-searching-based algorithm.

On the other hand, the hierarchy tree T, constructed using the proposed algorithm, has minimal construction costs. □

Theorem 4. *The proposed supergraph based hierarchy learning algorithm is deterministic.*

Proof: In the proposed algorithm, there are three main steps.

1. Supergraph construction
2. Label hierarchy construction
3. Hierarchy tree construction from the Label hierarchy tree
4. Learning the input DAGs from the hierarchy

The proposed algorithm assumes that all the nodes in a DAG are unique, which is always true for Bayesian networks. Hence, all the edges are also distinct in a DAG. So, the construction of a supergraph never involves making any choice from multiple alternatives. Instead, its construction is straightforward and deterministic.

Label hierarchy construction involves the finding of maximal proper subsets, which is also deterministic in nature. The method is used to construct a hierarchy tree for a given set of input DAGs. In the construction process, the algorithm constructs the tree by forming a parent–child relationship among all the DAGs as it finds pairs of graphs with minimum derivation costs. This process is also straightforward and does not involve any non-deterministic action.

The final step, that of learning the input DAGs from the hierarchy, involves a preorder traversal of the hierarchy tree and the extraction of nodes and edges from the supergraph by the checking subset–superset relationships. All these underlying operations and the whole procedure are deterministic.

Therefore, the proposed algorithm is deterministic.

□

5.4.4 Learning an OOBN class hierarchy: an example

As an example, a set of BNs is considered as shown in Figure 5.6, given as input to the algorithm. These may be previously constructed from expert elicitation or learned from a dataset using causal structure learning algorithms. The BNs are first converted to a set of OOBN classes, as shown in Figure 5.6. The conversion is done by the technique documented in Algorithm 5.3 which uses the two heuristics defined in Section 5.3.

From the set of OOBN classes, a set of DAGs is extracted, and a supergraph is constructed by amalgamating all the DAGs in one place. These operations are performed in Algorithm 5.4. The supergraph (as shown in Figure 5.7) contains all the information from the subgraphs that constituted it. All the nodes and edges in the supergraph have labels constructed from the label of the associated graphs (see Figure 5.6). Hence, each vertex and edge has a non-empty set of labels.

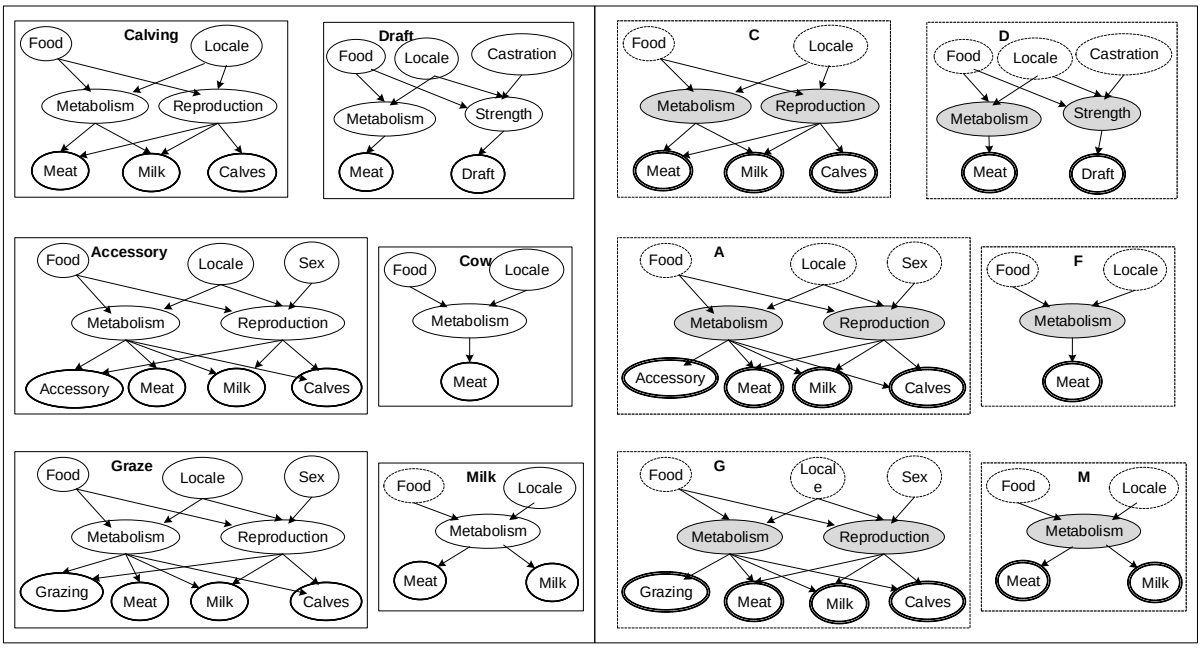


Figure 5.6: The set of BNs found from the OMD farm example used in the class hierarchy in Figure 3.8 of Chapter 3. The set is used here as an input to the proposed hierarchy learning algorithm.

Then the whole supergraph is traversed in order to extract a list of labels of the vertices and edges. The redundant labels are then removed and a set of labels formed from the list of labels. A hierarchy tree is constructed using the labels, where each node in the tree has a label that is a superset of all the labels of its descendant node and a subset of all the labels of its ancestor nodes. The hierarchy for the example OOBNs is shown in the left-most window of Figure 5.8.

Constructing the OOBN classes begins with traversing the hierarchy tree in preorder, i.e., root node > left child node > right child node. The traversal of the hierarchy is marked in iterations 1 to 6 (Figures 5.8 to 5.13). The current node under consideration is marked green, and the path from the root to the current node is marked using red colour in the nodes of the path, though the order or style of traversal makes no changes provided that a top-down traversal is followed.

Accordingly, step 1 of the execution (as shown in Figure 5.8) starts with the root node having the label "ACDFGM". Then the nodes and edges of the supergraph having the label "ACDFGM" or a superset of "ACDFGM" are searched. The DAG obtained in the way mentioned above forms an OOBN/iOOBN class.

In the same way, in steps 2 to 6 (figures 5.9 to 5.13), the hierarchy tree is traversed and, for each of the labels found in the tree nodes, the nodes and edges in the supergraph having the same label or superset of the label are searched. This procedure provides the seven OOBN/iOOBN classes in the six steps described in the figures.

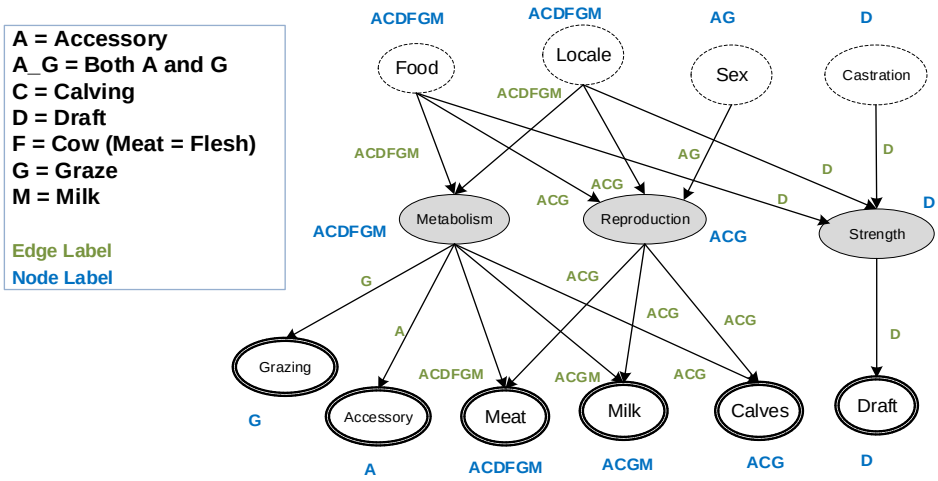


Figure 5.7: The supergraph constructed from the OOBN classes

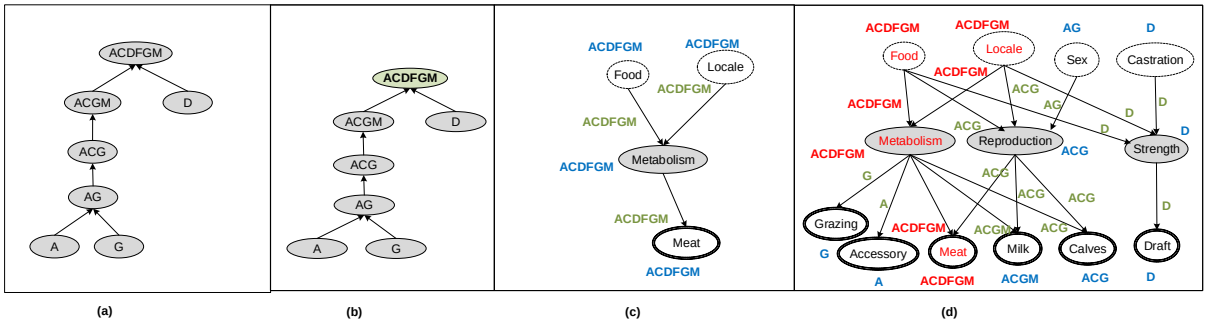


Figure 5.8: Learning iteration (1): (a) A label hierarchy tree constructed from the labels: "ACDFGM", "ACGM", "ACG", "AG", "A", "G", and "D". (b) Start traversing the hierarchy tree from the root node "ACDFGM". (c) The sub-DAG extracted from the supergraph where each node and edge has a label that contains "ACDFGM". (d) The graph segment marked in red-text in the supergraph.

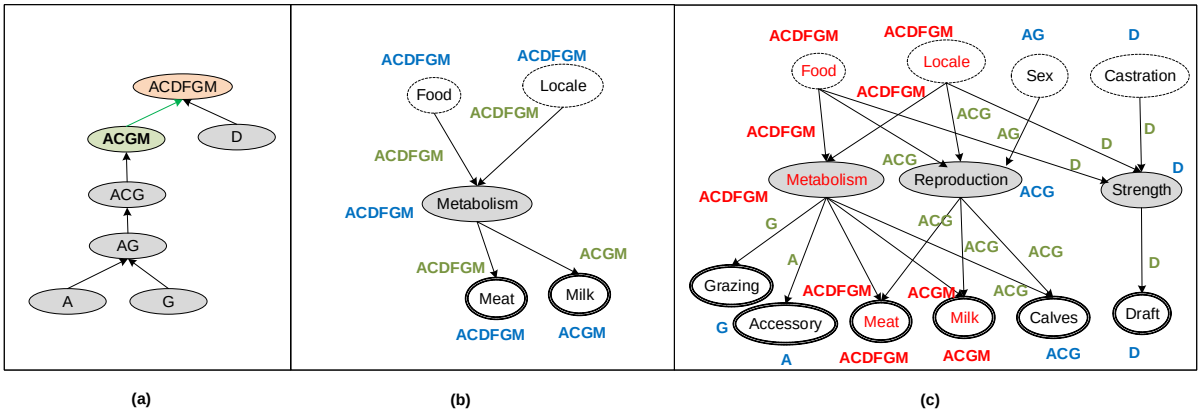


Figure 5.9: Learning iteration (2): (a) Take the left-most non-visited child of "ACDFGM", i.e., "ACGM". (b) The sub-DAG extracted from the supergraph where each node and edge has a label that contains "ACGM". (c) The graph segment marked in red-text in the supergraph.

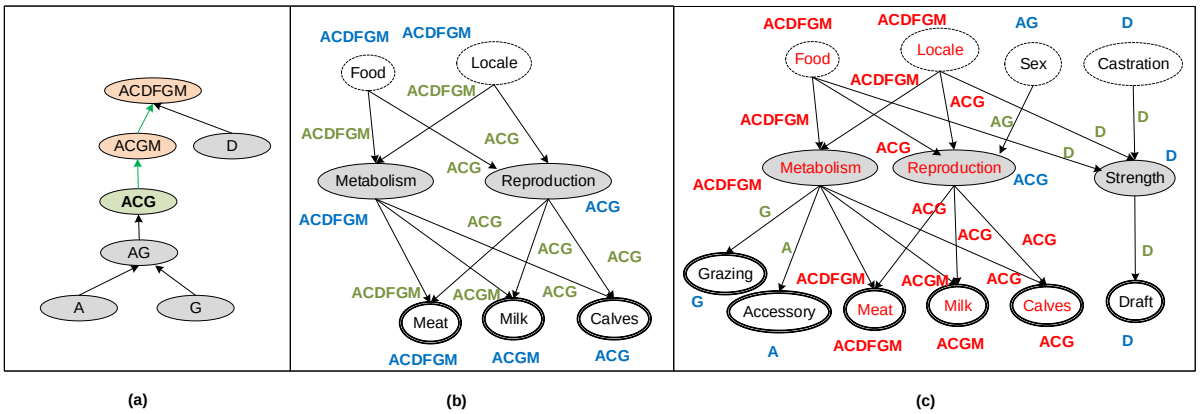


Figure 5.10: Learning iteration (3): (a) Take the left-most non-visited child of "ACGM", i.e., "ACG". (b) The sub-DAG extracted from the supergraph where each node and edge has a label that contains "ACG". (c) The graph segment marked in red-text in the supergraph.

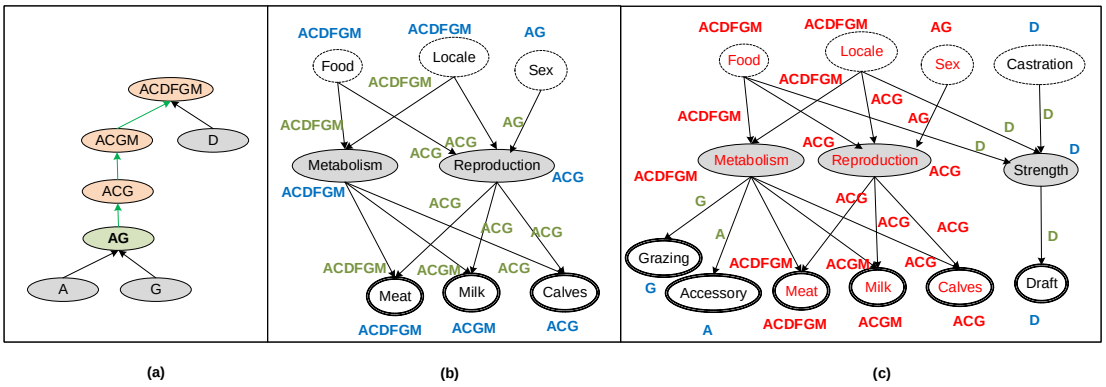


Figure 5.11: Learning iteration (4): (a) Take the left-most non-visited child of "ACG", i.e., "AG". (b) The sub-DAG extracted from the supergraph where each node and edge has a label that contains "AG". (c) The graph segment marked in red-text in the supergraph.

The classes constructed in the proposed learning algorithm also contain hierarchical information derived from the hierarchy tree. If two classes are formed from two nodes in the hierarchy tree, where one of the nodes is a parent (ancestor) node of another node, the class found by the parent (ancestor) node is a parent (ancestor) class of the later one. The whole procedure provides a hierarchy of classes, as shown in Figure 5.14.

Note, in the above running example, each class has exactly one extension option i.e. only one parent class to be extended from. This is not a usual case and hierarchies in real-life problems may not be quite so simple. Hence, how the proposed algorithm deals with choosing an appropriate parent class from multiple parent classes is shown using an extended example in Appendix E.

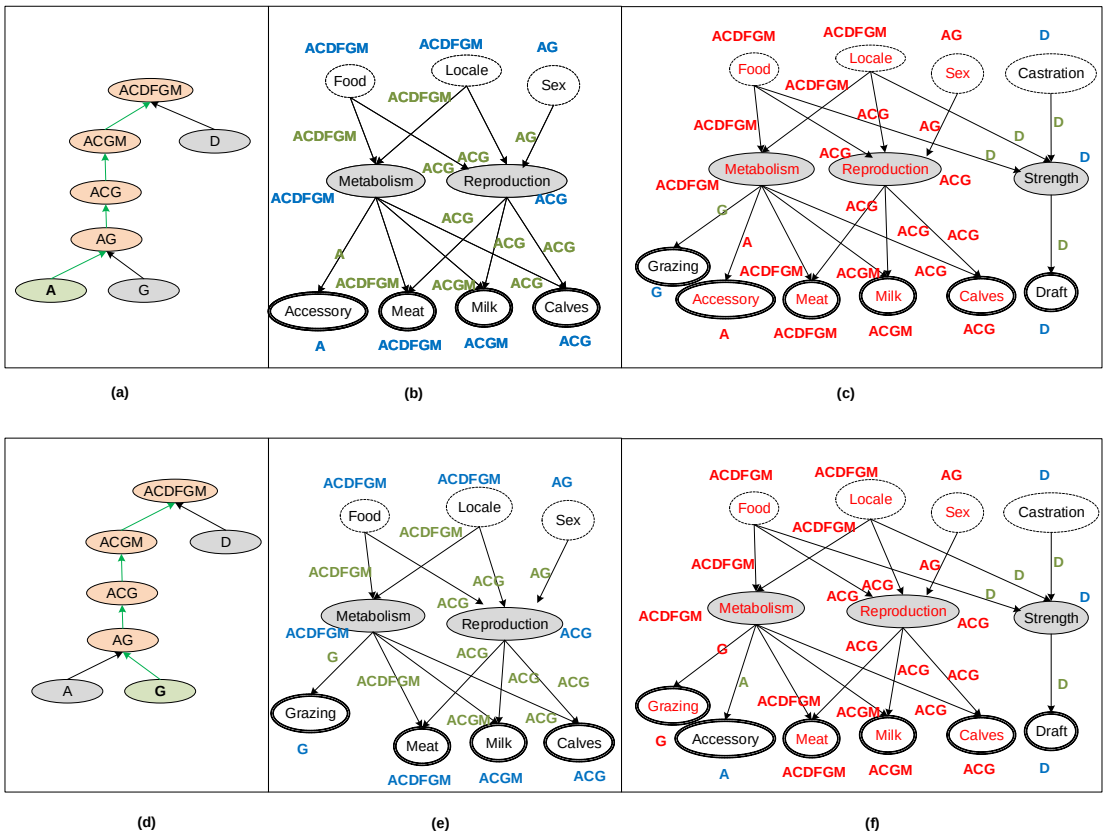


Figure 5.12: Learning iteration (5): (a) Take the left-most non-visited child of "AG", i.e., "A". (b) The sub-DAG extracted from the supergraph where each node and edge has a label that contains "A". (c) The graph segment marked in red-text in the supergraph. (d) Backtrack to the node "AG" and take the next left-most non-visited child of "AG", i.e., "G". (e) The sub-DAG extracted from the supergraph where each node and edge has a label that contains "G". (f) The graph segment marked in red-text in the supergraph.

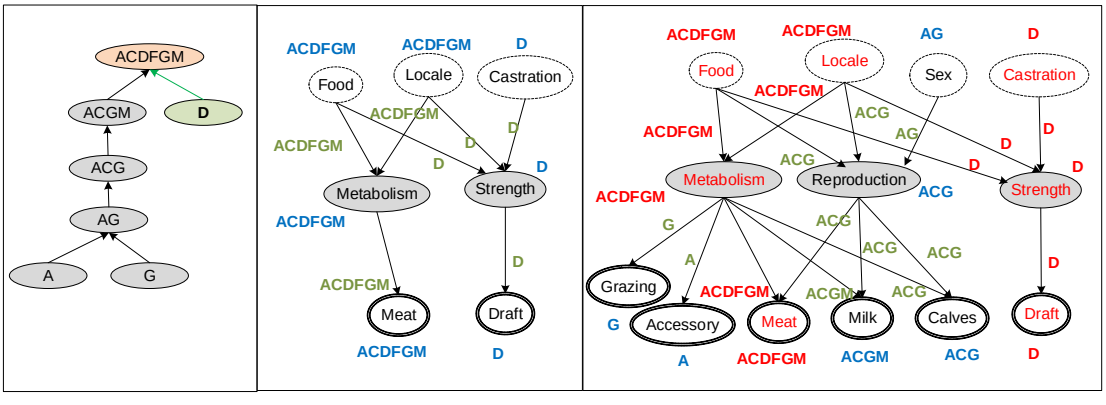


Figure 5.13: Learning iteration (6): (a) Backtrack to the node "AG", then to "ACG" and so on till the original root "ACDFGM" is found that still has a child unvisited. Take the next left-most non-visited child of "ACDFGM", i.e., "D". (b) The sub-DAG extracted from the supergraph where each node and edge has a label that contains "D". (c) The graph segment marked in red-text in the supergraph.

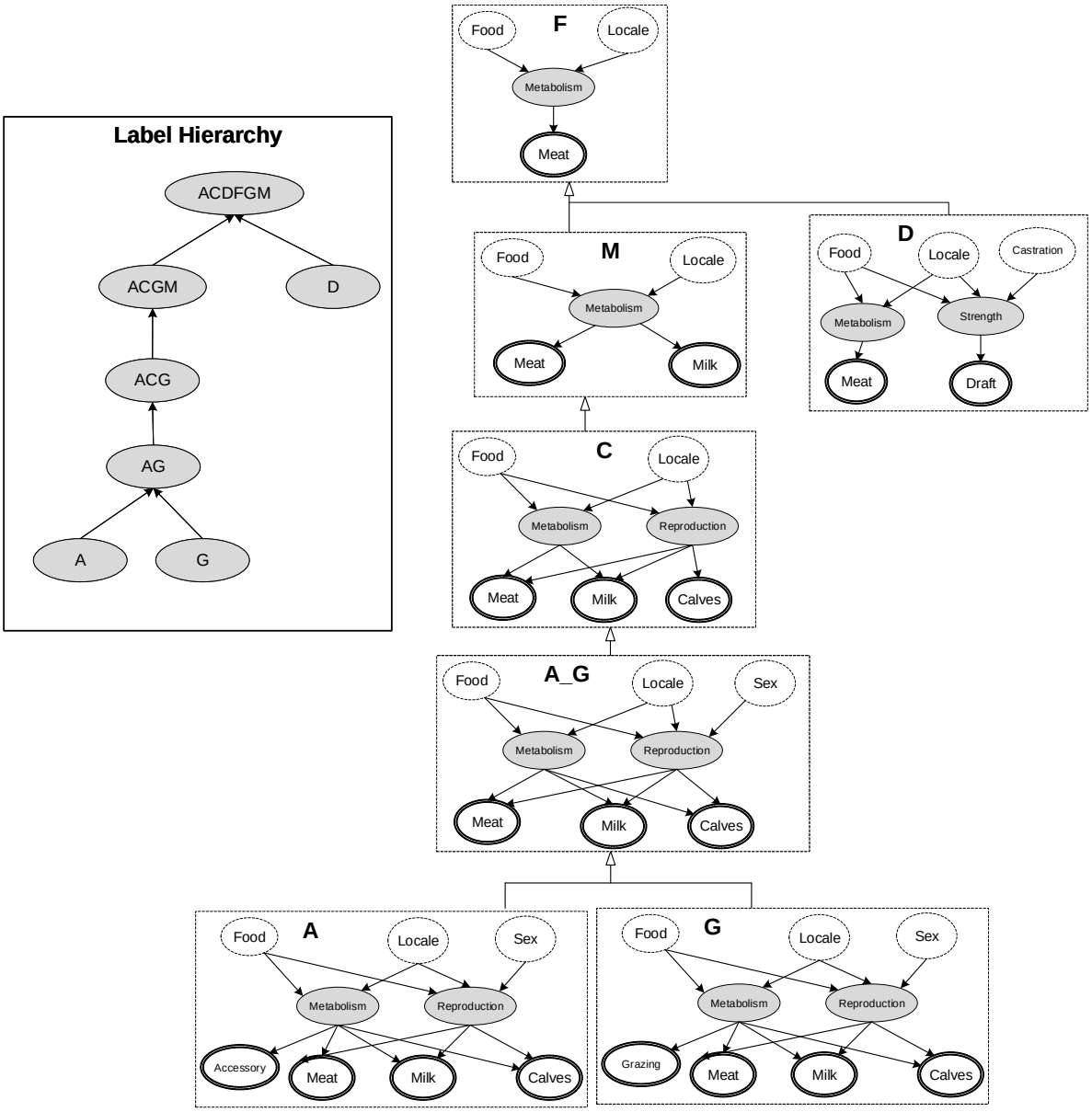


Figure 5.14: The class hierarchy learned by Algorithm 5.14. (This is same as the hierarchy constructed for OMD farm in Figure 3.8 of Chapter 3).

5.5 Evaluation

In this section an experimental analysis on synthetically created class hierarchies is given, using the measures defined above, to assess the efficiency of the learned class hierarchy compared to the original synthetic class hierarchies.

5.5.1 Synthetic OOBN class hierarchy generation

Because there are so few existing OOBN class hierarchies, in order to analyse the proposed learning algorithm and its effectiveness, a novel synthetic hierarchy generator has been devel-

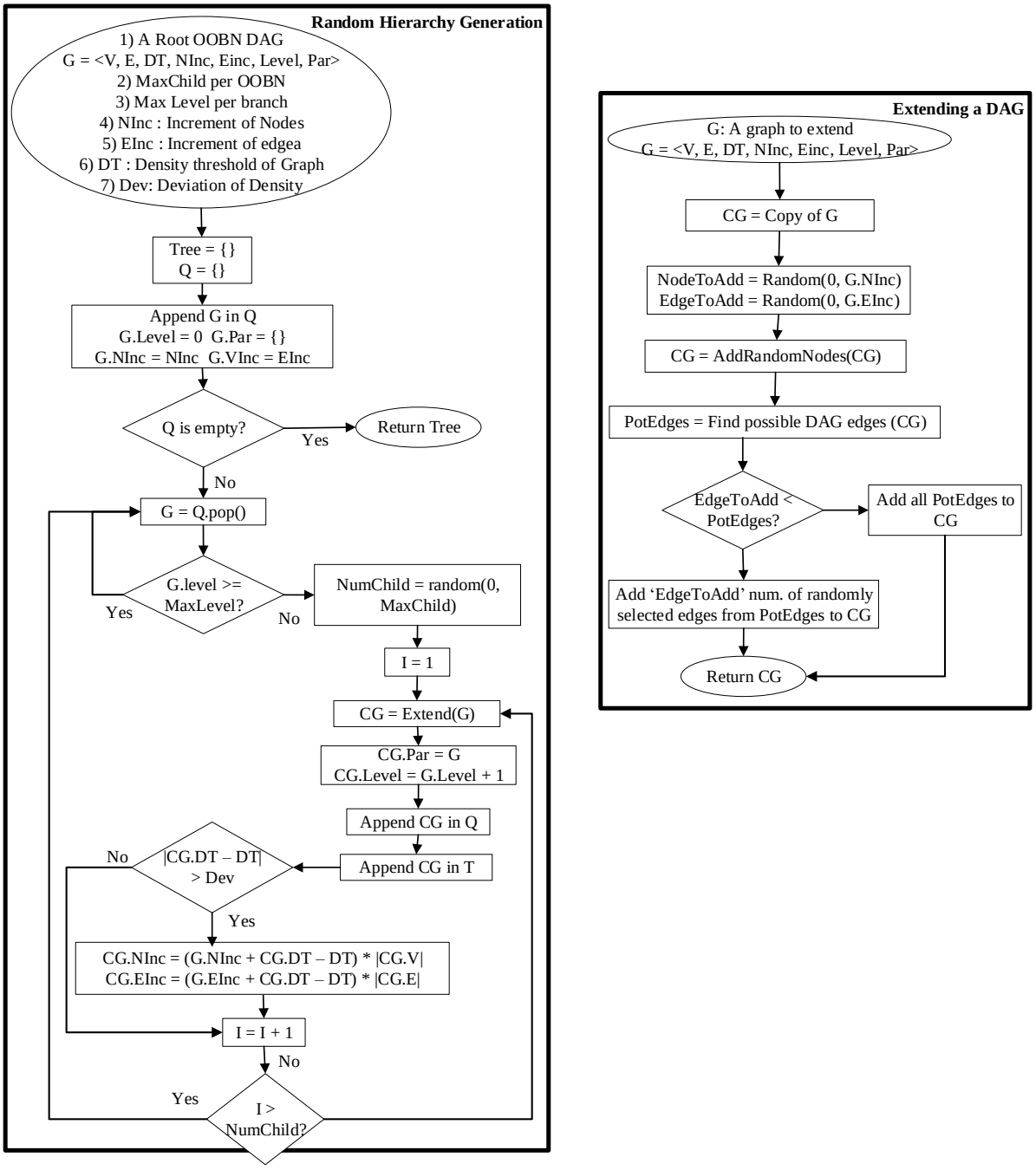


Figure 5.15: Flowchart of the synthetic hierarchy generation process

oped, as shown in Figure 5.15 in the form of a flowchart. The synthetic hierarchy generator starts with an initial DAG (which may be empty, provided by the user, or randomly generated) as the root of the hierarchy. Note, the hierarchy is a tree structure with each node of the tree containing a DAG. Starting with the initial DAG as the root node, some children nodes (each node containing a superDAG of the root node) are added by extending the root node. Then, recursively, each newly generated node (containing a DAG) is considered to extend in order to complete the hierarchy. The recursion terminates when a particular number of children and a

specific depth in the hierarchy is reached. The maximum number of children per hierarchy tree node and the maximum depth for any branch is specified by the user. They can also specify a range for the percentage increase in the number of nodes and edges in a newly generated child class, as well a range for its density (i.e., the number of edges per node). The generator adds a random number of children classes for each node in the hierarchy tree, without violating the overall maximums for number of children and maximum depth of any branch. The synthetic hierarchy generator is implemented in Python 3.7.

The synthetic hierarchy generator randomly selects the number of children to be generated for each node and recursively keeps adding children nodes unless it reaches the maximum number of children or the maximum depth of any branch. For some nodes, there might be no child at all due to a random number of children within a range being generated. For each child of a graph, a random number (chosen from a range) of nodes and edges are added that satisfies the density threshold.

5.5.2 Evaluation measures

In Section 5.2, some costs are defined, namely, the derivation cost, a hierarchy tree construction cost and an overall reusability cost. However, we need to compare the learned hierarchies with the original synthetic hierarchies, which may have different numbers of classes. So, additional measures that give a normalised derivation and construction cost, have been defined to provide more effective comparison.

DEFINITION 5.17 : DERIVATION COST RATIO

Given a hierarchy tree, $\top = < \mathcal{N}, \mathcal{E} >$ (Definition 5.7) and a DAG G, the **derivation cost ratio** of G with respect to $\top$ is,

$$\delta_{T_r}(G, \top) = \min_{D \epsilon \mathcal{N}} \frac{\delta(G, D)}{|D|}$$

DEFINITION 5.18 : CONSTRUCTION COST RATIO

Given a hierarchy tree, $\top = < \mathcal{N}, \mathcal{E} >$ (Definition 5.7) and a DAG G, the **Child-adding cost ratio** of a DAG C in $\mathcal{T}$ is

$$\chi_r(C) = \min_{D \epsilon \mathcal{N}} \frac{\delta(C, D)}{|D|}$$

DEFINITION 5.19 : RATIO COST

Given a hierarchy tree, $\top = < \mathcal{N}, \mathcal{E} >$ (Definition 5.7) and a DAG G, the **Ratio Cost** of $\top$ and $\mathbb{G} = \{G_1, G_2, ..., G_N\}$ is,

$$Cost_r(\top) = \sum_{D \epsilon \mathcal{N}} \chi_r(D) + \sum_{i=1}^{N} \delta_{T_r}(G_i, \top)$$

5.5.3 Experimental analysis

In the experiment to support the theoretical proof, two phases of experimentation were conducted. In each of the phases, 32 synthetic hierarchies were generated with various values of the parameters of the generation method (i.e. maximum number of children–"Max Child", maximum depth–"Max Depth", maximum node increase rate–"Max Inc Nodes", maximum edge increase rate–"Max Inc Edges" and maximum density of a generated DAGs–"Max Density"), with one row for each of the 32 cases, The parameters used to produce the synthetic hierarchies are given in the first five columns; "Number of DAGs generated Randomly" (NODR) shows the number of DAGs (i.e. the number of OOBN classes) generated in the synthetic hierarchy and "Number of DAGs Learned" (NODL) represents the number of DAGs generated by the proposed hierarchy learning algorithm. The table then gives the results of computing the different measures (combined construction and derivation cost, reusability and ratio cost) on the learned hierarchy (LH), the original synthetic hierarchies (SH), and the base case where there is no hierarchy (NH) [3]. For each row, for each measure, the better measure is bolded; lower is better for derivation–construction costs and ratio cost, higher is better for reusability. The row below all 32 cases gives the total count of cases where one hierarchy is better than the other. In other words, it can be seen how many times the learned hierarchy is better than the original synthetic hierarchy.

In both phases of the experimentation, the generation process starts with a randomly chosen DAG (from the six Cow OOBN classes plus the empty DAG). Then the proposed learning algorithm is applied to those 32 sets of classes in each phase. Note that in both the tables the number of DAGs generated in some learned hierarchies is sometimes smaller than the number of DAGs generated randomly for synthetic hierarchies. This is because the learning system is able to detect the similarity of two classes residing in two different branches of the synthetic hierarchy. The proposed algorithm keeps only one copy in the system and suggests that two original classes could be generated from that one single node.

[3]The result for no hierarchy is only given as a base case for the combined derivation and construction cost, D+C, as of course there is no reusability measure, and since for D+C we have found NH is worse in every case, there was no point in computing the ratio cost for NH.

Table 5.2 shows an analysis of the first phase. For the combined construction and derivation cost, and the corresponding reusability, the learned hierarchy is better in one third of the cases (11 out of 32). However, these measures do not consider the size of the DAGs. So to avoid a flat comparison of the derivation and construction cost by the number of nodes and edges added to the children DAGs, the ratio is computed with respect to the size of the parent DAG from which the child DAG is derived. In the Table 5.2 column "RatioCost", the LH and SH hierarchies each win in half the cases for the measures that incorporate ratios for a fairer comparison.

Table 5.1: A list of terms with their full-forms used in Table 5.2

Short term	Full form
MC	**M**aximum number of **C**hildren class per class
MD	**M**aximum **D**epth per branch in the hierarchy
MN	**M**aximum number of **N**odes per class
ME	**M**aximum number of **E**dges per class
MDe	**M**aximum **De**nsity of a DAG of a class
NODR	**N**umber **o**f **D**AGs **R**andomly generated within a hierarchy
NODL	**N**umber **o**f **D**AGs generated within a **L**earned hierarchy
D+C	**D**erivation and **C**onstruction cost
D+C Ratio	Ratio of Derivation and Construction cost over the size of a DAG of a class
LH	**L**earned **H**ierarchy
SH	**S**ynthetic **H**ierarchy
D+C Ratio LH leaves	D+C Ratio for the DAGs in the leaf level of the learned hierarchy tree
D+C Ratio SH leaves	D+C Ratio for the DAGs in the leaf level of the synthetic hierarchy tree

Note, in the toy example (shown in Section 5.4.4), that the learning algorithm may sometimes exactly reconstruct an original hierarchy. It is also true that the learning algorithm may not produce an optimal class hierarchy. Further investigations and changes are required to get an optimal outcome from the algorithm. The above empirical results suggest that, in practice, the algorithm can find class hierarchies that have a higher degree of reusability, i.e. are a more efficient representation.

To support the above assertion, further investigation was conducted. The analyses performed in Phase 1 generated hierarchies where the rate of node and edge increment to extend superclass to subclasses were estimated at 30% and 40%, respectively. Hence, an analysis was performed with node and edge increment rate estimated at 60% and 70%, respectively. A better outcome was observed in this phase of the experimentation. In Table 5.3 the row below all 32 cases shows that 26 times out of 32 learned hierarchies were found better than synthetic hierarchies if reusability and combined derivation and construction cost are considered. Furthermore, 27 times learned hierarchies performed better in terms of ratio cost.

Table 5.2: Comparison of class hierarchies learned from a set of classes, to the original hierarchies, and to no hierarchies. The node and edge increment rate in the subclasses are 30% and 40%, respectively.

Parameters					NODR	NODL	DerCost+HCCost			Reusability (ρ)		RatioCost	
MC	**MD**	**MN**	**ME**	**MDe**			**NH**	**LH**	**SH**	**LH**	**SH**	**LH**	**SH**
3	3	5	15	10	8	12	89	**31**	38	**58**	51	**6.55**	13.75
		10	20	15	10	24	276	134	**128**	142	**148**	**15.36**	29.98
	4	5	15	10	12	17	139	**44**	53	**95**	86	**6.85**	11.76
		10	20	15	9	15	339	161	**153**	178	**186**	**12.62**	26.28
	5	5	15	10	7	9	101	**34**	39	**67**	62	**6.16**	7.36
		10	20	15	21	62	1300	798	**608**	502	**692**	**30.083**	33.9
	6	5	15	10	32	40	702	188	**177**	514	**525**	**14.19**	19.71
		10	20	15	29	72	4897	4331	**3060**	566	**1837**	63.65	**46.7**
4	3	5	15	10	10	14	83	**26**	40	**57**	43	**5.57**	11.93
		10	20	15	16	37	450	226	**197**	224	**253**	**20.97**	32.99
	4	5	15	10	35	37	560	**169**	175	**391**	385	**16.55**	30.15
		10	20	15	34	86	1469	689	**549**	780	**920**	**35.65**	41.45
	5	5	15	10	93	81	1754	632	**482**	1122	**1272**	53.31	**48.95**
		10	20	15	81	149	6013	4042	**2687**	1971	**3326**	132.06	**91.71**
	6	5	15	10	130	114	3587	1258	**788**	2329	**2799**	71.35	**53.6**
		10	20	15	167	361	32212	**16141**	20304	**16071**	11908	390.52	**260.7**
5	3	5	15	10	16	22	235	**72**	100	**163**	135	**9.07**	25.91
		10	20	15	26	40	699	306	**294**	393	**405**	**25.25**	45.23
	4	5	15	10	49	44	694	**249**	**249**	**445**	**445**	**26.9**	32.5
		10	20	15	71	95	3200	1599	**1131**	1601	**2069**	74.11	**70.84**
	5	5	15	10	119	97	2714	1044	**680**	1670	**2034**	75.65	**62.26**
		10	20	15	174	369	12330	9629	**5354**	2701	**6976**	311.82	**143.45**
	6	5	15	10	279	166	8806	3020	**1869**	5786	**6937**	139.04	**88.29**
		10	20	15	517	422	119807	**28891**	78089	**90916**	41718	**567.36**	867.52
6	3	5	15	10	48	50	676	282	**269**	394	**407**	**31.58**	50.11
		10	20	15	58	59	1726	813	**677**	913	**1049**	**54.71**	74.71
	4	5	15	10	161	97	3174	1222	**874**	1952	**2300**	99.19	**92.56**
		10	20	15	185	124	8087	4237	**2881**	3850	**5206**	204.9	**142.83**
	5	5	15	10	475	256	10898	**2714**	4393	**8184**	6505	310.45	**195.94**
		10	20	15	615	380	49502	31803	**22689**	17699	**26813**	942.96	**527.38**
	6	5	15	10	683	356	19618	7894	**4333**	11724	**15285**	428.85	**219.48**
		10	20	15	1492	2434	366973	**51517**	241803	**315456**	125170	13471	2564.62
Count of times a hierarchy is better							0	11	21	11	21	16	16

Notations	**MC:** Maximum number of Children class per class, **MD:** Maximum Depth per branch in the hierarchy, **MN:** Maximum number of Nodes per class, **ME:** Maximum number of Edges per class, **MDe:** Maximum Density of a DAG of a class, **NODR:** Number of DAGs (classes) generated Randomly within a hierarchy, **NODL:** Number of DAGs (classes) generated within a Learned hierarchy, **LH:**Learned Hierarchy, **SH:**Synthetic Hierarchy, **Bold-numbers:** to indicate better

Table 5.3: Comparison of class hierarchies learned from a set of classes, to the original hierarchies, and to no hierarchies. The node and edge increment rate in the subclasses are 60% and 70%, respectively.

Parameters					NODR	NODL	DerCost+HCCost			Reusability (ρ)		RatioCost	
MC	**MD**	**MN**	**ME**	**MDe**			**NH**	**LH**	**SH**	**LH**	**SH**	**LH**	**SH**
3	3	5	15	10	5	8	28	**14**	19	**14**	9	**3.50**	5.60
		10	20	15	6	8	37	**17**	26	**20**	11	**4.63**	6.17
	4	5	15	10	10	21	62	**29**	32	**33**	30	**2.95**	6.20
		10	20	15	7	19	100	**17**	28	**83**	72	**5.26**	14.29
	5	5	15	10	17	28	46	30	**21**	16	**25**	**1.64**	2.71
		10	20	15	20	32	127	**59**	66	**68**	61	**3.97**	6.35
	6	5	15	10	18	33	198	128	**123**	70	**75**	**6.00**	11.00
		10	20	15	20	28	1223	**521**	664	**702**	559	**43.68**	61.15
4	3	5	15	10	12	18	58	41	**33**	17	**25**	**3.22**	4.83
		10	20	15	10	28	192	**120**	123	**72**	69	**6.86**	19.20
	4	5	15	10	25	29	115	**57**	96	**58**	19	**3.97**	4.60
		10	20	15	19	52	455	305	**214**	150	**241**	**8.75**	23.95
	5	5	15	10	45	51	519	**181**	253	**338**	266	**10.18**	11.53
		10	20	15	47	54	1153	**472**	997	**681**	156	**21.35**	24.53
	6	5	15	10	84	77	1723	**419**	675	**1304**	1048	22.38	**20.51**
		10	20	15	101	261	7221	**3167**	6149	**4054**	1072	**27.67**	71.50
5	3	5	15	10	13	19	54	**33**	46	**21**	8	**2.84**	4.15
		10	20	15	14	30	270	154	**140**	116	**130**	**9.00**	19.29
	4	5	15	10	41	49	265	**120**	232	**145**	33	**5.41**	6.46
		10	20	15	36	71	538	**205**	440	**333**	98	**7.58**	14.94
	5	5	15	10	100	75	1099	**355**	615	**744**	484	14.65	**10.99**
		10	20	15	112	149	863	348	**288**	515	**575**	**5.79**	7.71
	6	5	15	10	152	94	3146	**810**	1092	**2336**	2054	33.47	**20.70**
		10	20	15	251	382	15168	**6701**	13866	**8467**	1302	**39.71**	60.43
6	3	5	15	10	19	31	172	**99**	101	**73**	71	**5.55**	9.05
		10	20	15	22	41	323	**181**	227	**142**	96	**7.88**	14.68
	4	5	15	10	69	67	484	**201**	470	**283**	14	7.22	**7.01**
		10	20	15	71	66	2108	**797**	1011	**1311**	1097	**21.94**	29.69
	5	5	15	10	158	84	2320	**680**	1047	**1640**	1273	17.62	**14.68**
		10	20	15	229	129	8533	**3032**	5067	**5501**	3466	**35.15**	37.26
	6	5	15	10	295	168	8531	**2113**	4286	**6418**	4245	**20.78**	28.92
		10	20	15	755	523	43399	**19804**	39613	**23595**	3786	**32.98**	57.48
Count of times a hierarchy is better							0	26	6	26	6	27	5

Notations	**MC:** Maximum number of Children class per class, **MD:** Maximum Depth per branch in the hierarchy, **MN:** Maximum number of Nodes per class, **ME:** Maximum number of Edges per class, **MDe:** Maximum Density of a DAG of a class, **NODR:** Number of DAGs (classes) generated Randomly within a hierarchy, **NODL:** Number of DAGs (classes) generated within a Learned hierarchy, **LH:**Learned Hierarchy, **SH:**Synthetic Hierarchy, **Bold-numbers:** to indicate better

5.5.4 Case study: Learning hierarchical structure in WGR

As a proof-of-concept case study, the proposed learning algorithm was applied to the OOBN classes constructed for a real-world environmental modelling project [3] using the Hugin OOBN software (which does not support inheritance). These consisted of 129 OOBN classes with no reusability of components, giving a combined construction + derivation cost of 3993. The hierarchy learned using the proposed algorithm contained 159 classes with a construction and derivation cost of 2135 (a 46% reduction) and a reusability score of 1858 (a 54% increase). The derivation cost and hierarchy tree construction cost ratio was 309.84 (8%). The hierarchy for WGR classes, learned by the proposed algorithm, is given in Appendix D, together with a mapping between the original class names and the reengineered class names. Again, this suggests that the proposed algorithm could be useful in practice.

5.5.5 Summary of the evaluation

The analyses performed in Section 5.5.3 and the proof-of-concept case study (see Section 5.5.4) identified gains in terms of reusability, combined derivation and construction cost, and ratio cost for the hierarchy learned by the proposed algorithm in comparison with the synthetic hierarchy. In summary, the experiment results support the theoretical analysis that the proposed learning algorithm is suboptimal. Suboptimal means here that the outcome was not the best, a fact that is evident in the experimental analysis and supported by proof of non-optimality (see Theorem 3). However, in practice, the algorithm was useful and it points to a new direction in automated learning of OOBNs from data.

5.6 Summary

This chapter presented the first algorithm for learning an OOBN class hierarchy from a set of OOBN classes. It takes a set of OOBN classes as input and suggests a hierarchy tree that minimises the number of nodes and edges that need to be added in deriving the input DAGs.

The algorithm first constructs a supergraph by amalgamating all the input DAGs, adding a label for each node and edge that keeps track of its source DAGs. Next, it constructs a forest of multi-parent hierarchy trees based on the maximal proper subset property of the labels. Finally, from these hierarchy trees and the supergraph, it constructs hierarchies of OOBN classes, containing all the original classes as well as inferred new classes, where having additional classes reduces the overall derivation and construction costs of the hierarchies.

In order to evaluate the proposed new algorithm, a novel synthetic hierarchy generator was developed, and new measures of derivation cost and reusability were proposed that capture

the relative efficiency of a given class hierarchy. These measures are also used in the learning algorithm, when there is a choice between alternative parent classes in the hierarchy, to choose the one with a minimal derivation cost and hence maximise reusability.

While it is observed that the performance of the algorithm is suboptimal, the experimental results on synthetically generated hierarchies, and on a real-world environmental modelling case study, show that the algorithm is prospectively useful and efficient in practice. The algorithm can be used either to create a class hierarchy from ordinary BNs or from OOBNs that were built in a software that does not support inheritance, or it can be used to restructure an existing hierarchy to make it more compact and efficient.

One limitation of the current algorithm is that an embedded object is treated as a node; the algorithm could be improved if these objects could be treated as instances of the classes. A future extension is to investigate combining this new class hierarchy learning with learning the OOBN classes themselves from data. Further, there is potential for inheritance in object-oriented programming to be reshaped or learned using a similar approach to the algorithm proposed here, provided the programs can be represented graphically.

Conclusions

6.1 Research Contributions

The Bayesian network (BN) is one of the most suitable tools for making decisions under uncertainty. However, ordinary BNs present various limitations, and hence several techniques to overcome the limitations have been proposed. Among these techniques, object-oriented Bayesian (decision) networks (OOBNs) are the most promising. The thesis address a vital issue in the current state-of-the-art tools and frameworks for developing OOBNs and suggest an important solution with required demonstrations. The solution is a new OOBN framework, "iOOBN", that exhibits most of the OO features; such as encapsulation, inheritance, abstraction, polymorphism, type checking, and typecasting. This new framework provides flexibility in modelling, robustness in extending/reusing existing models and scalability in modelling large-scale real-life applications. iOOBN offers the potential for better reuse of existing components (classes, structures and parameters) by utilising inheritance, which builds a hierarchy internally. Proper treatment of the inheritance hierarchy facilitates propagating changes among classes that will in turn support better reuse of the existing components. A prototype of the framework was implemented to check the correctness of the functionalities included in the theoretical framework. As a proof-of-concept case study, a real-life project, WGR was reengineered using the developed tool, "iOOBN". The proposed iOOBN framework is backward compatible. In order to prove compatibility, the toy example problems that existing frameworks had considered are revisited, and possible extensions of the models are listed. The extensions demonstrate that iOOBN allows incremental modelling. The thesis also includes a short architectural view of the developed tool.

To date, there is no inference (actually compilation, a vital step of inference) technique proposed and developed that directly works on an OOBN. The present research study offers a compilation technique that not only directly works on the OOBN (i.e., avoids the need for flattening the OOBN into a BN) but also allows the reuse of already built junction trees (JTs). This reuse helps subclasses to use the JT of their parent classes and classes to reuse the JTs of embedded objects. The efficiency of the algorithm was analysed asymptotically, mathematically

and experimentally. The proposed algorithm was found efficient in subsequent analyses. In order to analyse the algorithm experimentally, synthetic OOBNs were generated using a wide range of varying parameters. The proposed SIIC algorithm performs better when there are embedded objects in a class, and the performance gets better when the number of objects per instantiated class increases. Furthermore, the SIIC algorithm was able to successfully compile the whole WGR project, whereas Hugin (a well-known and widely used OOBN modelling software) failed to compile the project due to its enormous size. Instead the WGR modellers had to write a stochastic simulation inference algorithm to compute the posterior probabilities and expected utilities for a 20 year scenario projection.

This thesis proposes a new algorithm for automated learning of class hierarchy from a set of OOBN classes. Learning the hierarchy of OOBN classes has the potential to play a significant role in the OO-arena, because, in the OO paradigm, re-shaping of inheritance hierarchy is a vital task that contributes to refactoring the components and helps to maximise opportunity for reuse. The new hierarchy learning algorithm is deterministic and suboptimal. In order to determine the efficiency of the learning algorithm empirically, a novel synthetic hierarchy generator was proposed and developed. Using the generator, a set of synthetic hierarchies (and hence, a set of OOBNs in each hierarchy) was generated. Some evaluation measures are proposed to evaluate the efficacy of hierarchies built by both approaches (i.e., learned by proposed algorithm and built by the synthetic hierarchy generator). The hierarchies learned by the proposed algorithm were compared with the synthetic hierarchies in terms of the proposed measures such as reusability and in most of the cases, the reusability of the hierarchies learned by the proposed algorithm performed better than synthetic hierarchies. Another case study was conducted on the real-life project WGR by reengineering the original model using the proposed learning algorithm. An inheritance hierarchy was constructed from the original OOBN classes and the new classes formed from the hierarchy. This reengineering achieved more scalability in terms of reusability in comparison to the original model. The empirical and case study results demonstrate that the hierarchy learning algorithm will be useful in future real-life applications, such as to reshape the hierarchy of an existing model or to ensure better reuse of existing components by building a better hierarchy of classes. Hierarchy learning is also a step towards automated learning of OOBN classes from data; this algorithm could be used in future algorithms to learn the hierarchy from a set of data by integrating any of the existing algorithms to learn BNs from a set of data.

6.2 Future Works

There are a number of promising directions for extending the research presented in this thesis.

Improving the SIIC Algorithm: The SIIC algorithm could be improved by adding an additional post-processing step to thin the cliques. Experimental results have also shown that the JT cost from using SIIC is sometimes higher than from using Hugin, which suggests there is scope for incorporating further heuristics to reduce those JT costs. Further, SIIC, like Hugin, currently only works for ordinary OOBNs. There is potential for extending these to Jensen's [228] strong JTs, which incorporate decision and utility nodes. The current SIIC also assumes that all nodes and edges in a superclass exist in the subclass (not a constraint in iOOBNs). Thus the SIIC could be extended to handle classes where edges and nodes are deleted in the inheritance hierarchy. Finally, the SIIC algorithm is defined in this thesis for OOBNs where only class inheritance is allowed. Extending the SIIC algorithm to work in an iOOBN system that allows interface inheritance could be a topic for future research work.

Extending iOOBN learning: The current hierarchy learning algorithm is suboptimal because sometimes it gets stuck in local optima. An optimal learning algorithm, possibly based on dynamic programming, is a plausible future research topic. In addition, the class hierarchy learning only identifies inheritance involving structure changes. A natural and simple extension would be to identify classes that vary in parameters only.

The current algorithm is limited to learning a hierarchy with no embedded objects in the input classes. Even if there were embedded objects, the proposed algorithm considered them as simple nodes. Therefore, an extension of the algorithm to learn iOOBN classes with embedded structures is a potential direction for future research.

As surveyed in Chapter 2, there are many existing large BN models. A promising further direction, (and useful for building up a library of iOOBN classes for reuse), is to develop a method for automated decomposition of a large BN into simpler and smaller BNs that could be treated as embedded objects of OOBN/iOOBN classes.

The proposed method works on an existing set of BNs or OOBN classes. It would be more interesting to adapt existing causal discovery algorithms to learn OOBNs directly from data.

Knowledge engineering in iOOBNs: In this thesis, pre-existing OOBN models were used for reengineering and testing purposes. An obvious next direction is to work directly with domain experts to build some iOOBNs from scratch. The learning from these new case studies should inform the development of an iOOBN knowledge engineering methodology or modelling guidelines, possibly extending the methodology proposed by Boneh [33].

The nature and scope of the concept of "class" in the iOOBN allows the incorporation/embedding of expert knowledge. In the reengineering of existing models (see Chapter 3), sharedness associated with the different examples (e.g. different vehicles) was illustrated. This

sharedness links to the Bradford Hill properties of causation. One can elicit people's causal knowledge from some generic reasoning and then project that into an instance through the iOOBN hierarchy. Such a property of "class" could be formulated to embed expert knowledge and to formalise the idea of causation.

iOOBN software: It would be beneficial for both the BN research community, and for BN modellers, to develop/implement a functional and independent version of iOOBN (not dependent on the Hugin decision engine) that supports all OO features, all type of nodes and all edges/links.

A long-term vision would be to generate a substantial set of iOOBN classes (possibly using the learning methods outlined above), and make these publicly available, to help bootstrap modelling for new iOOBN modelling applications.

iOOBN Software Development

In Chapter 2, Section 2.6, a comparative study of the existing popular tools for modelling is presented. Each has some common features, some special features, and associated limitations. It is clear that very few tools support OO-features, and they are either not cost-free or not well documented. Finding the best software that allows proper support of the extended features of OOBNs is very hard. The journey towards developing the software "iOOBN" that supports OO features, especially inheritance, its challenges and overcoming those challenges, features and limitations of the current version of the software is described in this Appendix.

A.1 iOOBN Software

The importance of the Object-Orientated (OO) features in modelling, especially the inheritance, is significant. The limitations of currently available tools to provide these features hinder the capability of modellers and discourage people from using them. Moreover, for large real-life applications, scalability becomes a significant challenges for existing tools. At the same time we know that the scalability issue can be resolved better with OO features. As an example, WGR [3] is a large real-life project: it fails to compile in Hugin. However, the reengineered version of the WGR successfully compiles and runs to perform inference.

Developing a complete, standalone tool for modelling with OO-features is challenging and requires much time, effort and funding. Therefore, the target of the project was to develop a prototype version and to test its functionality and feasibility. Instead of starting from scratch, the project aimed to build the tool on an existing framework (referred to as the "Skeleton" framework) that had the maximal required functionality to create the target tool. Onto this framework, we added the required components to support the existing OOBN features.

This appendix presents the challenges faced during the development of this tool and an architectural view of the software following the "4+1 view model" [7] (Figure A.1). A preliminary version of the software architecture can be found in the technical report [229] of the "iOOBN tool". The appendix concludes with a note about the limitations and some future development

plans for this project.

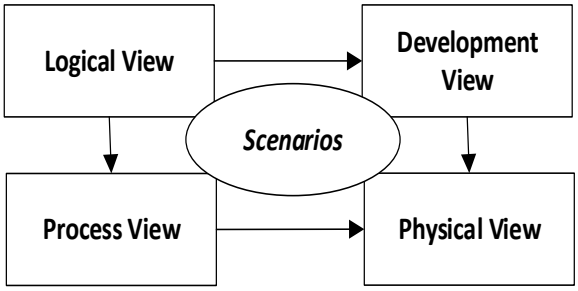


Figure A.1: The "view" model of software architecture [7].

A.2 Targeted Features for the Developed Framework

Existing frameworks have various functionalities and features that support Bayesian network modelling. Nevertheless, some well-known limitations of the existing tools necessitate a new OOBN tool that supports most of the OO features. Among the limitations, lack of support for inheritance is most significant. Other limitations include lack of provision for reuse of existing components, lack of strong type checking, the need for proper abstraction of classes and reusable-incremental compilation.

Therefore, the target of the new software is to address all the aforementioned limitations and overcome them, along with supporting the main OO features, i.e. inheritance, abstraction, polymorphism, and encapsulation.

A.3 Challenges Faced

According to a principle of software engineering, replicating previous work should be avoided when developing a new system, or software or framework. Since there is plenty of software available for BN modelling, the first challenge is to find the most appropriate skeleton framework to develop the target software.

A non-exhaustive list of challenges faced during the development of the software is as follows:

- Starting from scratch or choosing a skeleton framework or API
- Finding the right API
- Accessing the code base of the API to add features
- Accessing through an alternative door, i.e., NET grammar

- Reengineering the grammar and adding features through ".net" file creation
- Developing NPP grammar for supporting extended features
- Developing a compiler to translate the ".net" code to and from the ".NPP" code
- Adding a GUI and interfacing with Hugin

A.3.1 Choosing the right skeleton framework

A list of desired software features was drawn up and software was tested and compared to find the best backbone framework. The comparison suggested the most popular and free open source framework UnBBayes. However, the UnBBayes core systems have many shortcomings due to problems in resolving dependency and their lack of backward compatibility. Moreover, proper documentation in the English language was difficult to manage in developing a plugin for UnBBayes.

Finally, Hugin Expert (another popular framework providing OOBN features) was chosen. However, the free edition (Hugin Lite) has only limited facilities and is not suitable for developing powerful features. A licence for two users of the Hugin Researcher package was therefore purchased to start the development.

A.3.2 Accessing the code base and adding backward compatibility

Hugin Experts provide only an API (Application Programming Interface) to use the functionalities and GUI (Graphical User Interface) to model applications using Bayesian theory. The owners of the system do not disclose or share their code base with others. Even several attempts to collaborate with them in the R&D (Research and Development) process failed to convince them.

Since HUGIN was chosen for the core of the decision engine, and the proposed framework has some significantly improved OO-features compared with HUGIN, backward compatibility became an issue. To overcome this issue and to develop an iOOBN tool based on Hugin, it was necessary to analyse the file structure and contents structure of the text file, the language to define networks for the Hugin API. The language follows a grammar called NET grammar. Parsing Hugin text files that contain NET language codes required using NET grammar.

A.3.3 Deciphering the codes used in Hugin: reengineering NET grammar and parsing NET language codes

NET grammar (used to define the textual definition of OOBN classes in HUGIN) is not publicly available, which posed another challenge. Hence, NET language specification had to be

studied extensively, and around three hundred codes were analysed from various real-life applications: these were modelled using Hugin (as in the WGR project [3]), and then the codes were reengineered into a grammar that resembles the NET grammar used in Hugin and NET language.

Next, a parser had to be either developed from scratch or borrowed from existing systems. The latter method was chosen to avoid re-inventing the wheel. Initially, PEP [230] (an Earley recursive parser) was chosen, which took a considerable amount of time (approximately 50 seconds) to parse an average of 400 lines of NET code. (The reason behind this slowness is that it generates all possible parse trees for given inputs).

Finally, the ANTLR 4.6 [231] parser (an LL(*) parser) was adopted, which performed much better than PEP. ANTLR requires about 800 milliseconds (0.8 seconds) to handle an average of 1000 lines of NET code.

A.3.4 Developing NPP grammar and NPP language: a dedicated back-end grammar and language for iOOBN

The target was to provide extended OO-facilities, NET language and NET grammar not being sufficient. Hence, a dedicated grammar development was needed. An extended version of NET grammar (refer to Textbox A.3.1 and Textbox A.3.2 for NPP grammar) has been developed. It is called Net Plus Plus (NPP), and the language that follows this grammar is called NPP language.

TEXTBOX A.3.1 (CONTEXT FREE GRAMMAR : NET PLUS PLUS)

iOOBN	:	concClass | abstractClass | interface
concClass	:	**class** name parClass parInterface '{' elem* '}'
abstractClass	:	**abstract class** name parClass parInterface '{' elem* '}'
interface	:	**interface** name parInterface '{' interfaceElem* '}'
interfaceElem	:	basicNode | attrib+ | classInstance
parClass	:	**extends** name | ϵ
parInterface	:	**implements** interfaceList | ϵ
interfaceList	:	name (',' name)*
elem	:	domainElem | attrib+ | classInstance
domElem	:	basNode | ptnt
classInstance	:	**instance** name ':' cName '(' binds ')' '{' iAttr '}'
iAttr	:	label | pos | attr
binds	:	inBinds | inBinds ';' outBinds | ';' outBinds | ϵ

```
inBinds      : inBind ( ',' inBind )*
inBind       : formalName '=' actualName
outBinds     : outBind ( ',' outBind )*
outBind      : formalName '=' actualName
basicNode    : node ndName '{' nodeAttrib* '}'
             | nodeType node ndName '{' nodeAttrib* '}'
             | decision ndName '{' nodeAttrib* '}'
             | utility ndName '{' nodeAttrib* '}'
nodeType     : discrete | continuous
nodeAttrib   : state | label | pos | attr | sType
state        : states '=' '(' STR* ')' ';'
label        : label '=' STR ';'
pos          : position '=' '(' xCoord yCoord ')' ';'
```

TEXTBOX A.3.2 (CONTEXT FREE GRAMMAR : NET PLUS PLUS)

```
sType        : subtype '=' boolean ';'
             | subtype '=' label ';'
             | subtype '=' number ';' stVals
             | subtype '=' interval ';' stVals
stVals       : state_values '=' '(' NUM* ')' ';'
potential    : potential edgeInfo '{' potentialAttrib* '}'
edgeInfo     : '(' childNodes ')' | '(' childNodes | parNodes ')'
childNodes   : ID+
parNodes     : ID+
data         : data '=' '(' tuple ')' ';'
tuple        : NUM | '(' tuple ')' | tuple NUM | '(' tuple ')' tuple
potentAttrib : data | modelAttribs
attrib       : attribName '=' attribValue ';'
modelAttribs : model_data '=' stmt ';' | attrib | model_nodes '=' '(' ID* ')' ';'
stmt         : '(' stmt ')' | expr ',' stmt | expr | ϵ
attribValue  : STR | NUM | '(' NUM+ ')' | '(' ID+ ')' | '(' ')'
func_call    : ID '(' ')' | ID '(' parameters ')'
params       : expr | params ',' expr
expr         : sumExpr ( logicExpr sumExpr )*
```

```
logicExpr   :  '<'  |  '<='  |  '>'  |  '>='  |  '=='  |  '!='
sumExpr     :  prodExpr  ( ( '-'  |  '+' )  prodExpr )*
prodExpr    :  primary  ( ( '/'  |  '*' )  primary )*
primary     :  literal  |  funct_call  |  '('  formula  ')'
literal     :  true  |  false  |  NUM  |  STR  |  ID
```

The overall plan was to open for use the full functionalities of the OO paradigm for BN, i.e., inheritance, polymorphism, encapsulation, and abstraction. However, for the computation of belief, inference, and reasoning, the system relies on the Hugin decision engine. Thus, the next challenge was to develop a syntax converter for translating NPP to NET code.

A.3.5 Developing a syntax translator, code optimiser and code generator for NET

Although there are lots of similarities in the reengineered NET grammar and NPP grammar, there are significant differences too, especially in their principles. For example, NET language has no facility to share or reuse code as NPP has. The NET language cannot provide abstraction, type checking, overloading, overriding or polymorphism facilities, all of which NPP language can offer.

Hence, a syntax translator was developed that translates NPP code into NET, after performing intermediate optimized code generation.

A.3.6 Interfacing with Hugin API and providing GUI facilities

Figure A.2 illustrates the process of interfacing with the Hugin Decision Engine (HDE) given that the built-in GUI and API of Hugin Expert developer package (as explained above) are not able to be modified or extended. The figure also shows the NET language code, which is parsed by the HDE in order to generate BN components for real-life applications. Hence, two components have been developed, that is, an NPP language code generator and compiler for NPP code in order to produce NET code to be parsed by HDE.

Writing code in the NPP language with use of NPP grammar is a tedious and laborious job that would certainly discourage users from using iOOBN software. A GUI system would make it much easier for users to model a BN/OOBN. Thus users would not need to follow the structure or grammar or NPP rules explicitly, i.e., they would not need to memorize and write code as in programming. For those reasons, a GUI was added to the iOOBN to allow easy interfacing with the Hugin API within modelling environment. Hence, in the current iOOBN system, users can model in BN/OOBN without directly writing Net or NPP code, instead

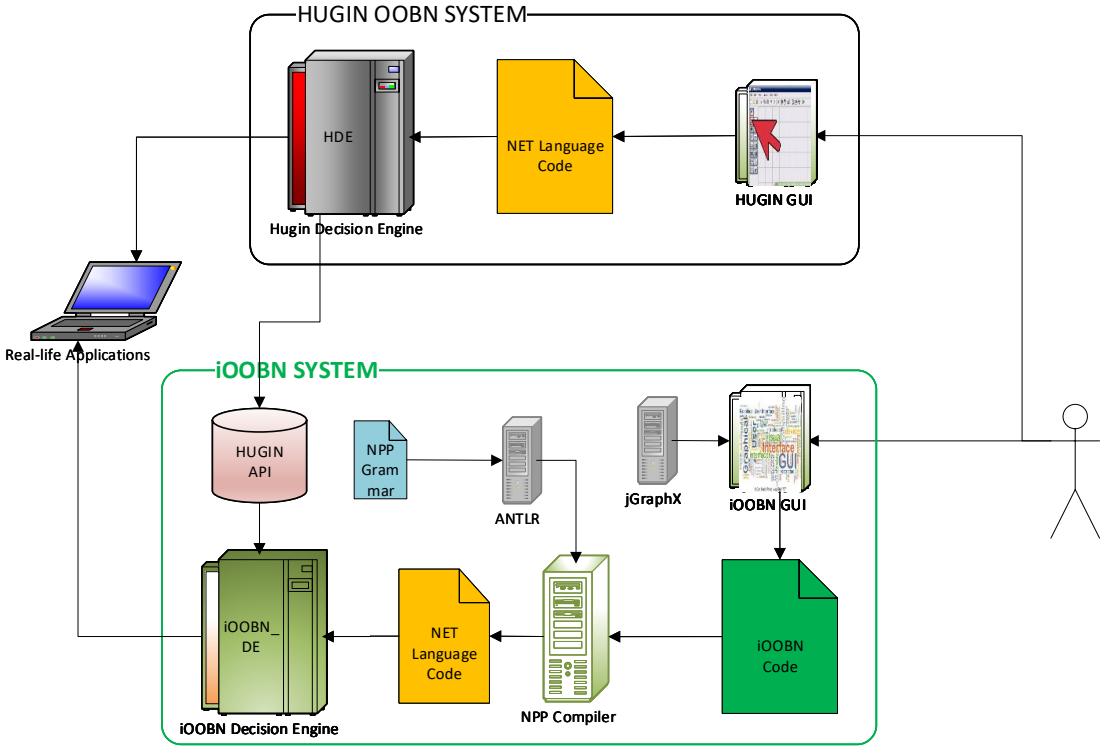


Figure A.2: Interfacing with the Hugin engine

using the developed GUI to suit their modelling requirements. The system then internally generates NPP code. This code is fed to the iOOBN core engine that converts the NPP code to Net code and feeds the net code to the HDE by using the Hugin API for inference and reasoning.

A.4 Features of iOOBN Software

The developed iOOBN system uses all the facilities provided by Hugin, as it was developed on the skeleton framework of Hugin. Capabilities of the developed software are listed as follows:

1. Users can create interfaces, abstract and concrete classes using GUI with a ".ioobn" extension as follows:

    - An interface can have input and output nodes with no CPTs defined in it.
    - An abstract class can have input-output nodes, embedded nodes, objects, and edges. There can be fully defined, partially defined, or non-defined CPTs for nodes.
    - A concrete class can have input, output, embedded nodes, objects and edges. There can be fully defined CPTs for embedded nodes.
    - Abstract classes and interfaces are not eligible for inference, i.e., cannot be instantiated. They are used mainly for inheritance purpose and facilitating placeholders

for concrete classes.

- An interface can implement multiple interfaces.
- A class can implement multiple interfaces but can extend a maximum of one class, that is, a single parent class is allowed.

2. The "compile" function can be run to compile/convert a file with extension ".ioobn" to create ".oobn" files. This step will actually convert an iOOBN code (also known as NPP code) into a Net language code (native code in Hugin). Hence, the source file/code, i.e., iOOBN code (also known as NPP code) will be converted to an equivalent OOBN/Net language code, i.e. the target code. Note that "iOOBN code" (also known as NPP code) contains some texts defining the formation of iOOBN components like nodes, edges, CPTs, classes, and interfaces. It may be hand-written by a modeller or generated by iOOBN software by converting the drawings (drawn by a modeller) in the GUI of the iOOBN software.

3. The "Belief Propagation" function can perform the computation of beliefs and inference. This step will simply use the functionalities provided by Hugin through Hugin's APIs.

4. With the GUI, users can define various nodes, edges and CPTs as well as inherit attributes like nodes, edges and CPTs from their parent interfaces (only interface nodes, i.e., IO nodes) and parent classes (nodes, edges and CPTs).

5. Users are also allowed to change/alter/edit any component as per their requirements or redefine any component inherited from parent classes or parent interfaces. This alteration of inherited components is called overriding.

6. Sharing interface allows users to work in a parallel manner, independently in a group or interact remotely. As well, this allows multiple forms of definition with unique signatures for any components like classes and interfaces. This facility is introduced to facilitate polymorphism.

7. Abstract classes provide flexibility to define or redefine CPTs and classes as per requirements and to amplify code sharing facilities among components.

8. Abstract classes and interfaces allow abstraction of components.

9. The software allows dynamically changing objects by abstraction classes through interfaces and abstract classes and the overriding facilities of specific components.

10. Instances of classes can be utilised to define time-slice representation.

A.4.1 Comparison of iOOBN features with existing software

Bayesia Lab [105], UnBBayes [107], Hugin [103, 232], Netica [29] and GeNIe [35] are state-of-the-art and widely used BN modelling tools. They support particular features very well, but some features are not well supported or not supported at all. Table A.1 compares the features of existing tools and the new iOOBN tool.

Table A.1: Supported features of BN/OOBN modelling tools

Features	**Netica**	**GeNIe**	**UnBBayes**	**HUGIN**	**iOOBN**
DBNs	✓✓	✓✓✓	✓	✓	✓✓✓
Submodels	×	✓✓✓	✓ ✓✓	✓ ✓✓	✓✓✓
OOBNs	×	×	✓✓✓	✓✓✓	✓✓✓
Supports inheritance	×	×	×	×	✓✓✓
Supports polymorphism	×	×	×	×	✓✓✓
Supports typecasting	×	×	×	×	✓✓✓
Incremental compilation	×	×	×	×	✓✓✓
Reuse of JT	×	×	×	×	✓✓✓
Sensitivity to findings	✓✓	✓✓	✓✓	✓✓	✓✓
Sensitivity to parameters	×	×	✓✓	✓✓	✓✓
Continuous nodes	✓	✓✓	✓✓	✓✓✓	✓✓✓
Equations	✓✓	✓	✓	✓✓✓	✓✓✓
Learn parameters from data	✓✓✓	✓✓	✓✓	✓	×
Learn structure from data	✓	✓✓	✓✓	✓	×
Ease of use	✓✓✓	✓✓	✓✓	✓	✓✓✓
Cost	✓✓	✓✓✓	*open source*	✓	*free*

✓✓✓ : High | ✓✓ : Medium | ✓ : Low | × : Not available [a]

[a] Ticks and crosses are derived from the survey of user experience performed in a workshop of the 2016 ABNMS conference and user feedback provided on various web pages relating to the software packages.

A.5 Prototype Implementation

A prototype iOOBN implementation was carried out using Java programming language and the Hugin BN software API [27]. This prototype is based on a newly defined language NET++ (NPP), an extended version of the NET code that is used in Hugin. The interpreter integrated into iOOBN converts the iOOBN definitions of nodes, edges, classes (abstract and concrete) and the interfaces defined in NPP language to the equivalent NET language. The HUGIN engine is then used to perform all the standard belief updating and other BN operations. The grammar for "NET Plus Plus" is given in the following text boxes.

A.5.1 Design goals

The design priorities of iOOBN software are presented before the design is presented.

The design priorities for the $iOOBN$ software are:

- The design was targeted at minimizing the complexity. "Re-inventing the wheel" was avoided by making the best use of existing resources.
- The $iOOBN$ tool was developed as part of this PhD research project and mainly designed for a prototype implementation. The development was done in a well-coordinated fashion by a very tiny group of people. Hence, the development process was well-organized with no chance of miscommunication or mismanagement.
- The overall system is designed to be extendable. Adding features to the existing system without breaking the working status is straightforward.
- All the standards used to design and develop the basic software were followed to make it easier for users to operate it with minimum effort and also for developers to develop and extend it easily.

A.5.2 System behaviour and use case view

Use case view of a system is used to drive the design phase and validate the output of the design phase. The architecture description presented in Figure A.3 for the software iOOBN starts with a review of the expected system behaviour.

The diagram consists of four main components or subsystems, namely

1. GUI editor
2. Hugin Integration
3. SIIC Compilation
4. Learning iOOBN

The overall iOOBN system is also divided into the aforementioned four subsystems. Users can directly interact with the system through the GUI editor. They can create a class (abstract or concrete) or an interface, and save/open/close new or existing files containing definitions of iOOBN components (class/interface). They can also instruct the system to perform inference as well as to learn iOOBN classes from a set of data or a set of OOBN classes. In other words, they can convert an existing OOBN repository into an iOOBN repository using the learning module of iOOBN software.

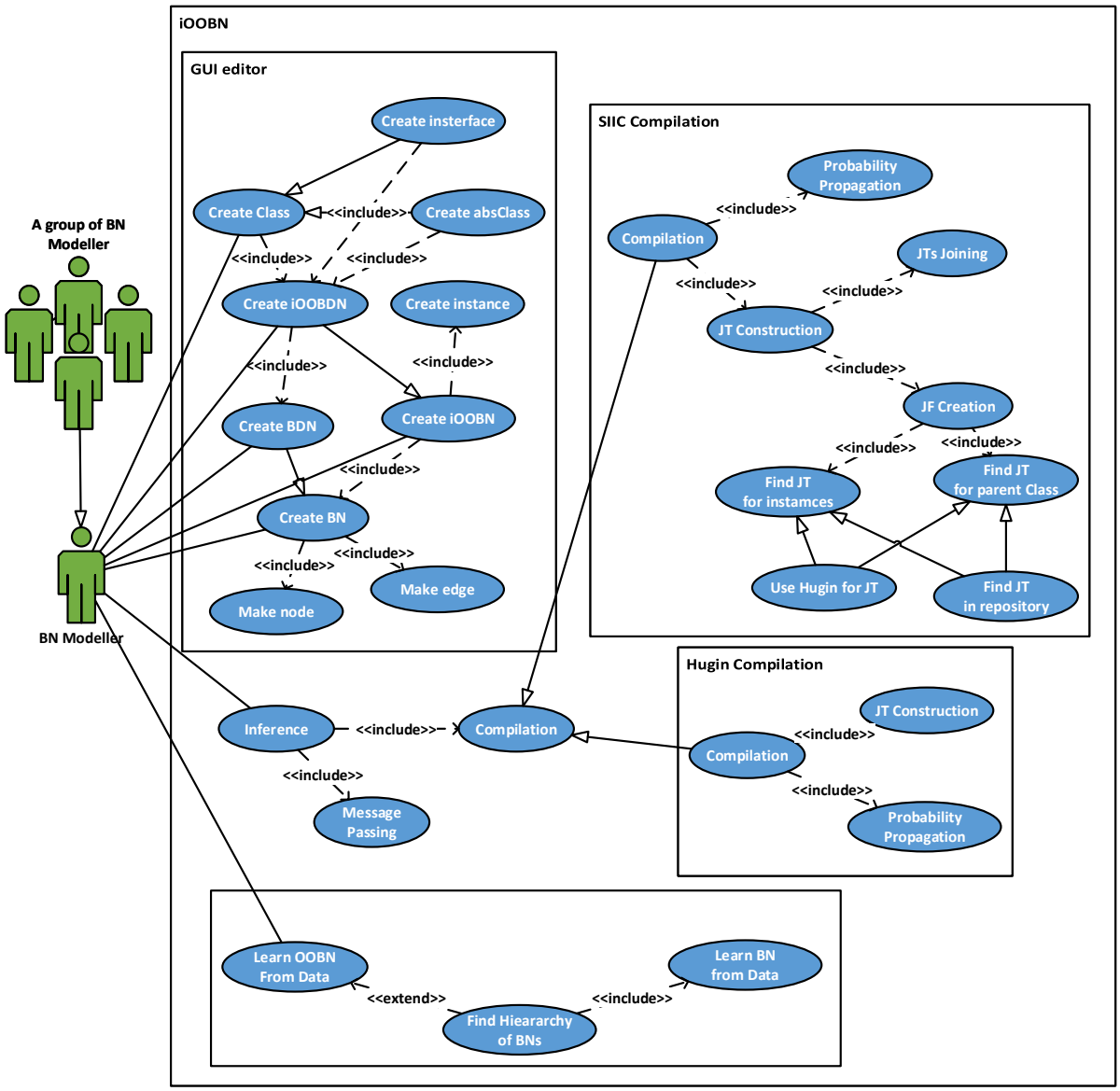


Figure A.3: Use case Diagram of iOOBN software.

A.5.3 Logical view

The main functional components of a system are described in logical view. The components include modules, the static relationships between modules, and their dynamic patterns of interaction.

In this view, the modules of a system are expressed by detailed components and classes that come with specific attributes and operations. However, an in-detail class diagram seems beyond the scope of the thesis and unnecessarily increases the word count. Therefore, class diagram details are kept to a minimum.

A.5.3.1 Detailed class design

This section presents class diagrams of various packages of the software iOOBN.

Figure A.4 depicts the class diagram of the package iOOBN GUI editor that is an essential part of the user interface. In Figure A.5 the class diagram of the package "Graph panel" is shown, which is mostly borrowed from jGraphX [233] framework that deals with graph editing. The functionalities of the package are modified and extended as per the requirement of iOOBN. This module is one of the submodules of the iOOBN editor module, whereas Figure A.6 depicts the class diagram of the package "Frames", another submodule of the iOOBN editor. It mostly contains the frames to take user inputs, such as name, attributes, CPTs and other necessary inputs.

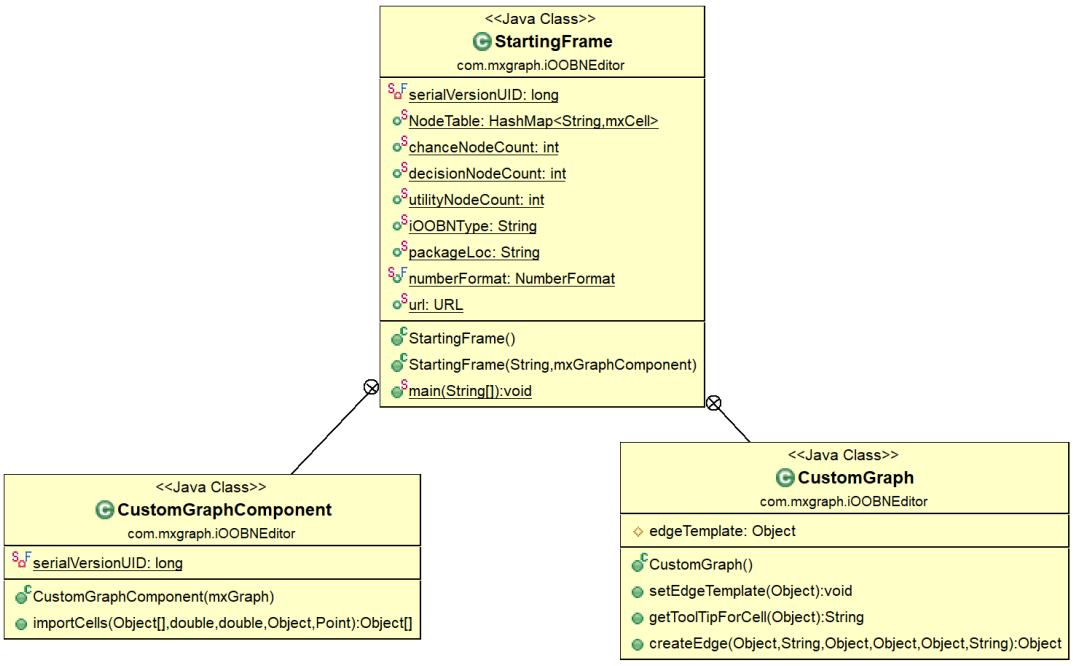


Figure A.4: Class diagram of iOOBN Editor package

Figure A.7 illustrates the class diagram of the package iOOBN GUI editor package, another submodule of the iOOBN editor. It contains all the classes necessary to facilitate iOOBN-specific operations/actions such as super (class or interface) adding, instance adding, compiling, and code generation. Figure A.8 shows the class diagram of the package "components". This package consists of classes representing attribute storage and functionalities of the main iOOBN components such as class (abstract and concrete), interface, nodes, and potentials.

Figure A.9 shows the class diagram of the package ANTLR [231]. This package contains the required classes to use the facility of ANTLR API to develop the core compiler techniques. Figure A.10 represents the class diagram of the package Hugin API [27] Integration. This package is made up of the classes that handle communication between the iOOBN and Hugin and

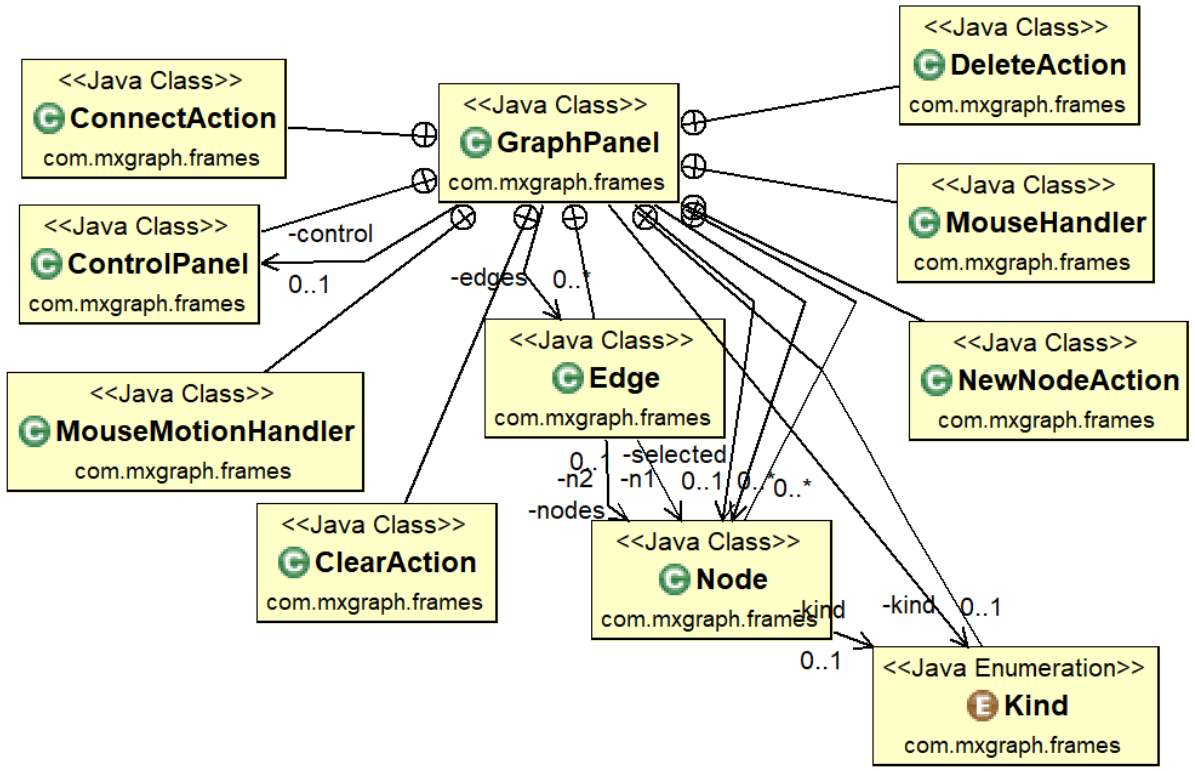


Figure A.5: Class diagram of Graph Panel package

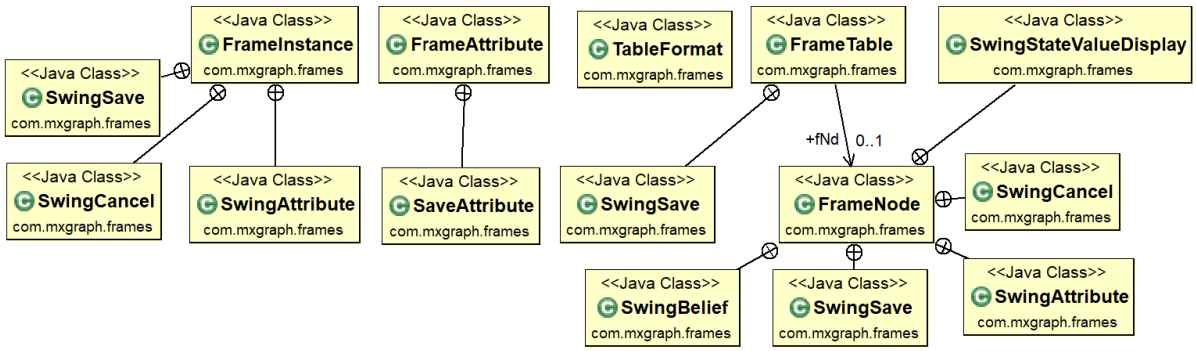


Figure A.6: Class diagram of Frames package

utilization of Hugin API. Figure A.11 presents a class diagram of the package "Learn OOBN" that contains the classes required to learn OOBN and iOOBN classes from a set of data. This feature is unique to iOOBN software, and the first-ever in the literature on BNs.

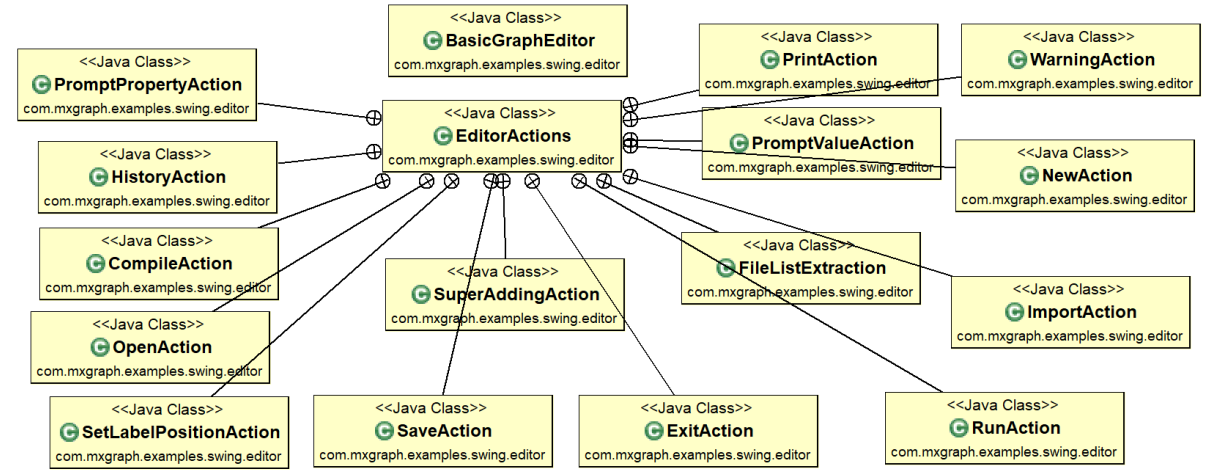


Figure A.7: Class diagram of GUI editor package

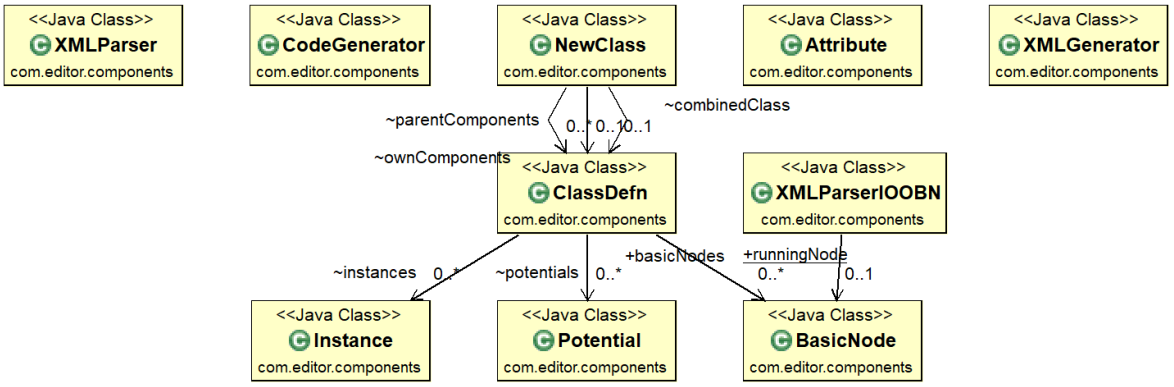


Figure A.8: Class diagram of OOBN components package

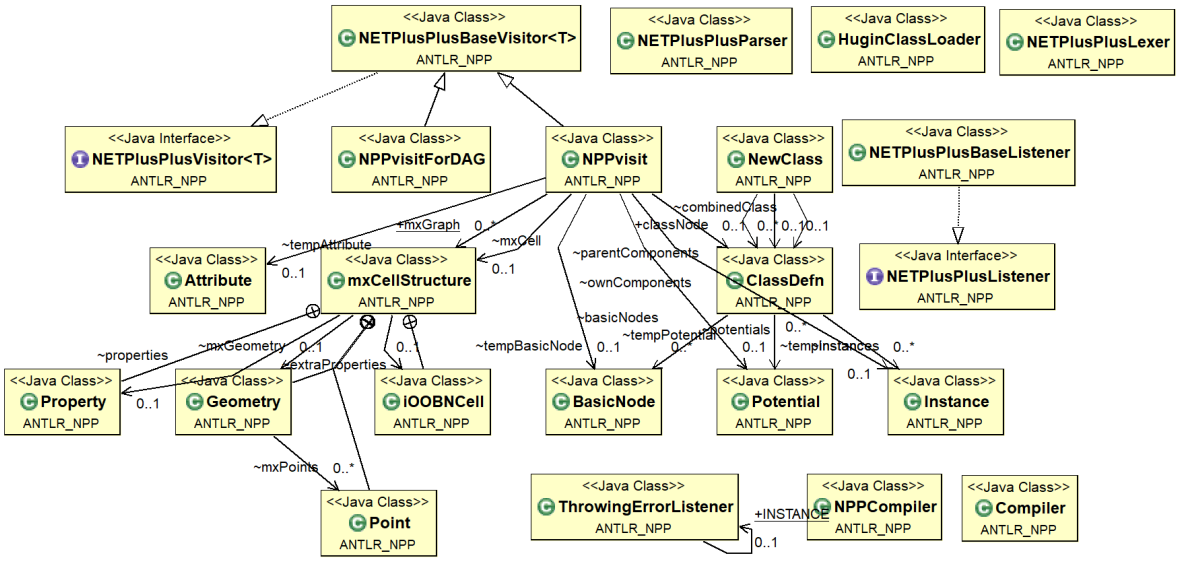


Figure A.9: Class diagram of ANTLR Net Plus Plus package

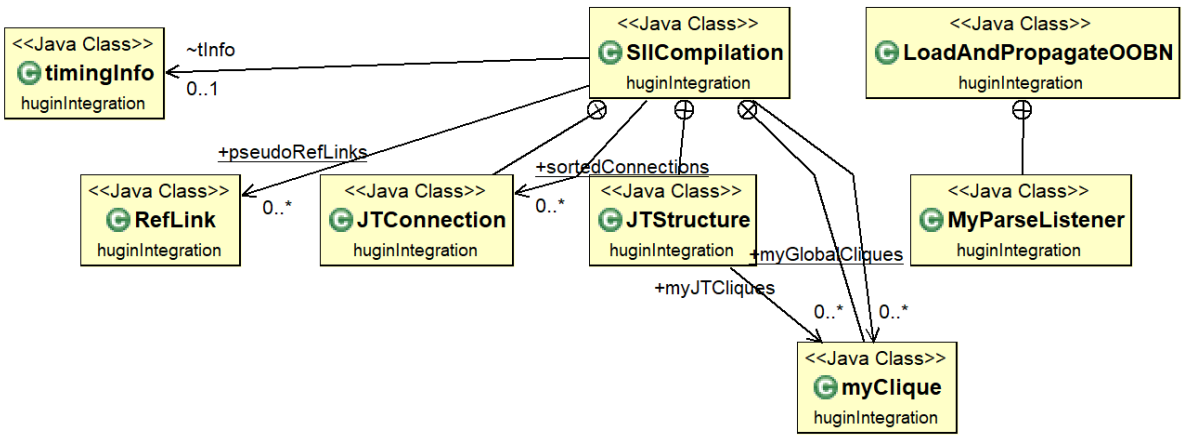


Figure A.10: Class diagram of the package to integrate with Hugin

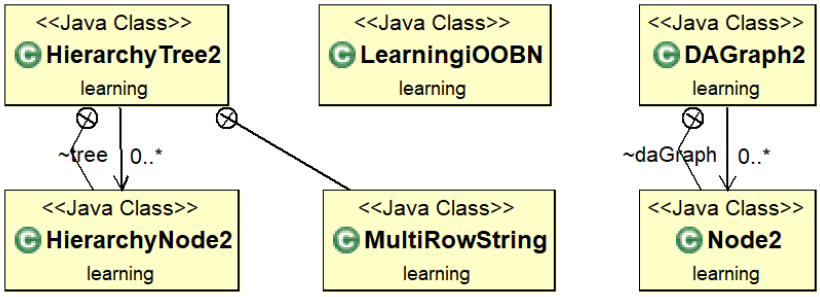


Figure A.11: Class diagram of Learning OOBN package

A.5.4 Process view

The process view is essential in understanding how the separate components and subcomponents communicate with each other in a concurrent application. However, iOOBN software is a single process and single-threaded system. The "process view" is omitted, as there is no multithreading or multiprocessing and no need for interprocess, or interthread communication.

A.5.5 Development view

The iOOBN can be viewed as a high-level compiler, working in the back-end of a GUI editor. The whole system stores the drawn and constructed network in two different formats. The former file type has an extension ".ioobn" containing all the GUI components' definitions and the latter file type has one of the following extensions: ".class", ".absClass", or ".interface" containing all the iOOBN components' (nodes, edges, and instances) definitions and their properties such as potentials and states.

Once any of the iOOBN definitions (abstract class, concrete class or interface) is found, the OOBN (HUGIN [27] compatible) code is generated by translating the iOOBN definition using the reengineered NPP grammar shown in Textbox A.3.1 and Textbox A.3.2. The OOBN code is

then used in the Hugin decision engine to perform inference if SIIC (proposed in Chapter 4) is not used for compilation.

In the iOOBN tool (see Figure A.2), in order to build the compiler module, the ANTLR [231] framework is used; this uses NPP grammar, as defined in the framework, to generate the compiler for the iOOBN. This compiler produces Hugin-compatible net code from iOOBN-compatible NPP code so that the HUGIN API can be used without redefining features (e.g., message passing, and JT creation) from scratch for the iOOBN.

The graphical editor of the iOOBN is built on top of the jGraphX [233] framework, which is an open source graph editing tool. This popular and widely used graph editing framework is used to avoid having to redevelop the GUI.

In the iOOBN tool, an iOOBN code (also known as NPP code) is generated from the GUI file that is generated by users using the GUI editor. Then lexical analysis and syntax analysis are performed. If no problem is found in the code, or a valid code is generated, a symbol table and a syntax tree are constructed. The symbol table and the syntax tree are then used to generate an optimized and correct Hugin OOBN code to best utilize the functionality of the Hugin decision engine. The compilation system is depicted in Figure A.12. If the iOOBN encounters an OOBN file, it can pass it directly to either the SIIC inference engine or the Hugin decision engine to perform the inference. Otherwise, if it encounters an iOOBN file, it first generates an OOBN from the source iOOBN file using the integrated compiler system and then works on it as it does for an input OOBN file.

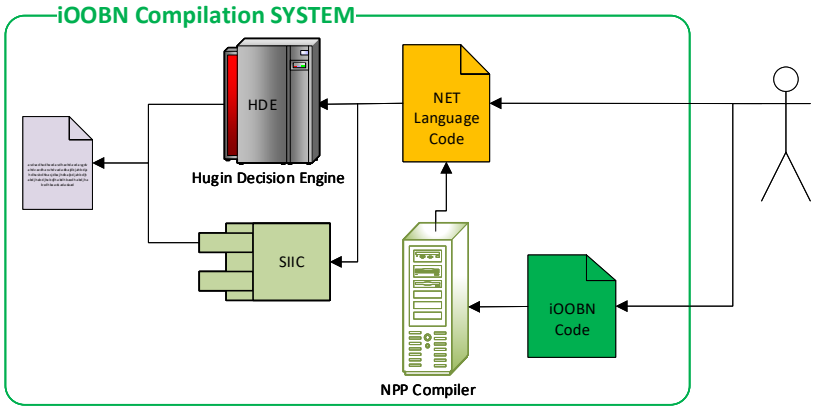


Figure A.12: How iOOBN software compiles both an OOBN and iOOBN file.

The compilation follows all the steps of a standard compiler system, like preprocessing, lexical analysis, syntax analysis or parsing, semantic analysis, and target code generation. The intermediate code generation and code optimization steps, however, are skipped. The flowchart in Figure A.13b illustrates how the integrated compiler generates an OOBN code from a given iOOBN code (also known as NPP code).

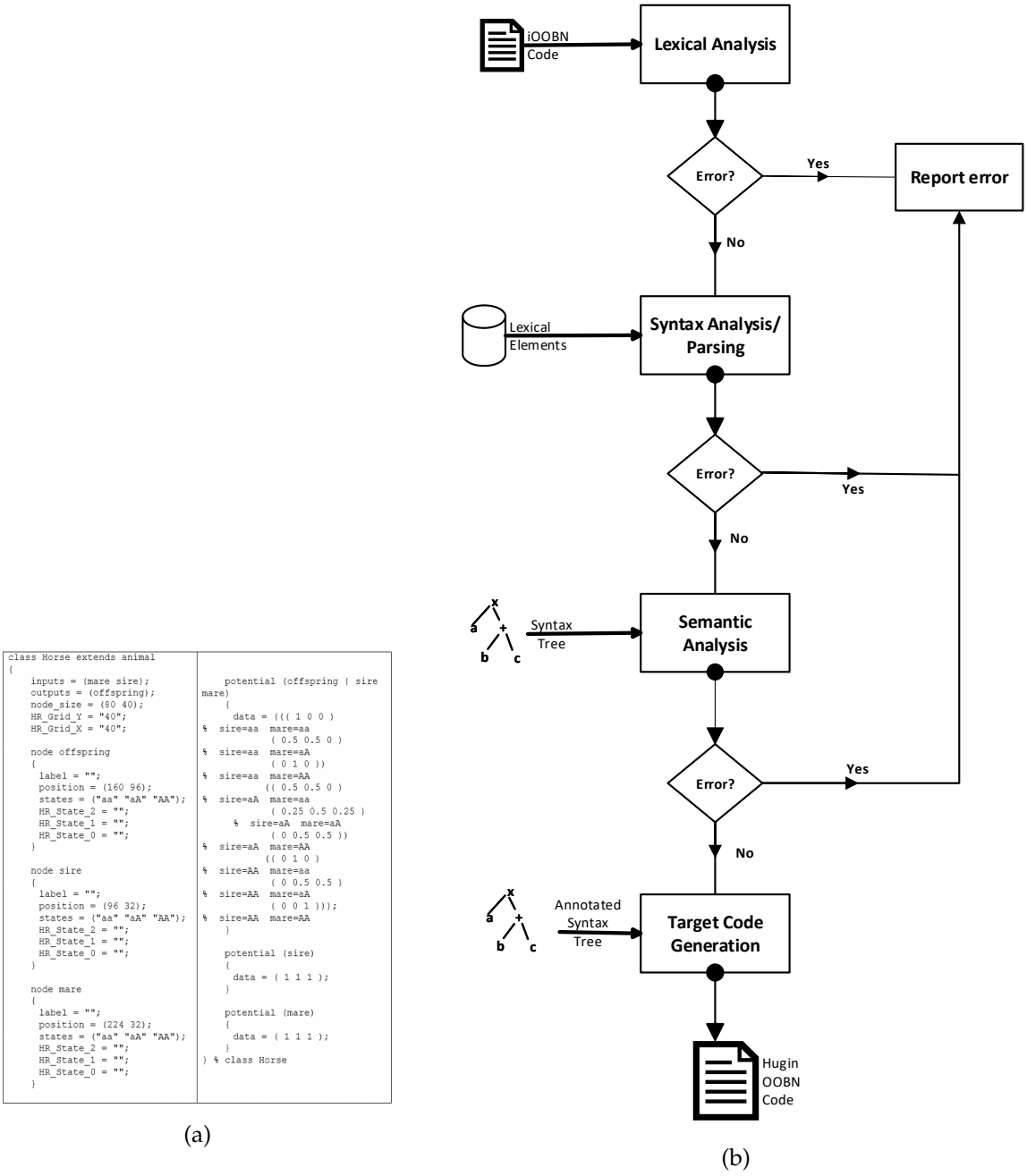


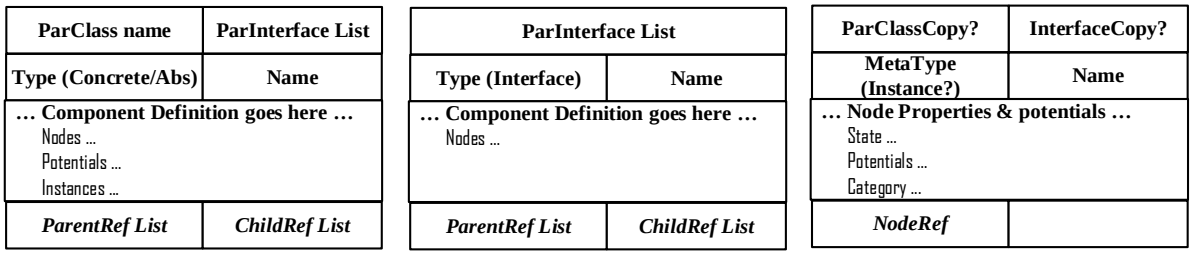
Figure A.13: (a) An example iOOBN code snippet: A "horse" iOOBN class that extends an animal iOOBN class, and (b) The steps involved in target code (NET language Code) generation from source code (iOOBN code, also known as NPP code).

An iOOBN class (abstract or concrete) contains a set of nodes, potentials and instances definition in its body. In the header, it contains a unique and valid name, an optional set of iOOBN interfaces names and an optional parent class name. A high-level view of the class structure is shown in Figure A.14a. Each line may have optional comments that begin with a "//" symbol (single line comment marker).

Similarly, an iOOBN interface contains a set of input and output node definitions in its body. In the header of an interface, there has to be a unique and valid name for the interface

and an optional set of parent interface names. A high-level view of the interface structure is presented in Figure A.14b.

In sum, the main difference between an OOBN and an iOOBN class is the header section.



(a) (b) (c)

Figure A.14: Meta Node structure of: (a) iOOBN class (abstract or concrete), (b) iOOBN interface; (c) Node data structure of iOOBN

Table A.2 represents the hierarchy of classes that was shown in Figure 3.8 graphically. Here the tabular format is the underlying representation of this hierarchy in iOOBN.

Table A.2: Hierarchy table: A tabular representation of hierarchy tree.

Ind	Name	Meta Node Ref	Child Index	Parent Index
0	Cow	MetaNodeCow	1, 2	∅
1	MilkCow	MetaNodeMilk	3	0
2	DraftingCow	MetaNodeDraft	∅	0
3	CalvingCow	MetaNodeCalve	∅	1
4	.	.	.	.
.	.	.	.	.
.	.	.	.	.
.	...	...	...	...

In the table, Ind is the index of a node in the tree to track the parent-child relationship ($Child-index$ and $Parent-index$) between nodes. The $Name$ has to be a unique, valid name for a class/interface where $MetaNode\ Ref$ is a reference to the instance of the structure as shown in Figure A.14a for classes and Figure A.14b for interfaces.

Table A.3 is an analogy to the so-called "symbol table" of a typical compiler. It stores static, dynamic and all other required information to execute a set of instructions in a traditional compiler. Here, a node table serves almost the same purpose but one more relevant to inference. In Table A.3, an example of a node table is shown for a class "MilkCow" and how the inheritance of each component in a class are tracked. Here, Id is a unique identifier generated by iOOBN internally, and $Name$ is a valid unique name given by users for each component (nodes, instances, potentials) of an iOOBN meta component (class/interface).

$NodeRef$ in Table A.3 is a reference to the instance of the structure that stores necessary contents and information for an element of a BN or OOBN/iOOBN class or interface. Fig-

Table A.3: Symbolic node table used in iOOBN compiler.

Id	Name	Node Ref	Node Src	Instance?	Overridden?
id0	Food	NodeFood	par	×	×
id1	Locale	NodeLocale	par	×	×
id2	Sex	NodeSex	self	×	NA
id3	Metabolism	NodeMetabolism	par	×	×
id4	Milk	NodeMilk	self	×	NA
id5	Cow	NodeCow	par	×	×
.	.	.	.	.	.
.	.	.	.	.	.
.	...	...	...	...	...

ure A.14c depicts a very high-level view of such a structure. The column $NodeSrc$ in the table is for tagging whether a node is a copy/reference of a parent class/interface or an element of its own class/interface. The column $Instance?$ marks whether a node is a complex instance node or a simple decision/utility/chance node. Finally, the column $overridden?$ tags whether a node is inherited from parent class/interface or changed in the child class/interface.

Finally, the high-level architecture of the learning iOOBN class element (which is integrated into the iOOBN) is presented. Figure A.15 shows a block diagram of the learning system. iOOBN classes can be learned from a set of data or a set of BNs or a set of OOBN classes. If data or BNs are given, they are automatically converted to OOBN classes and sent to the main learning engine. The engine extracts a set of DAGs and finds a hierarchy (as proposed in Chapter 5) of the OOBNs. This hierarchy helps in designing or building iOOBN classes in a more efficient and cost-effective way by allowing maximum reusability of existing resources.

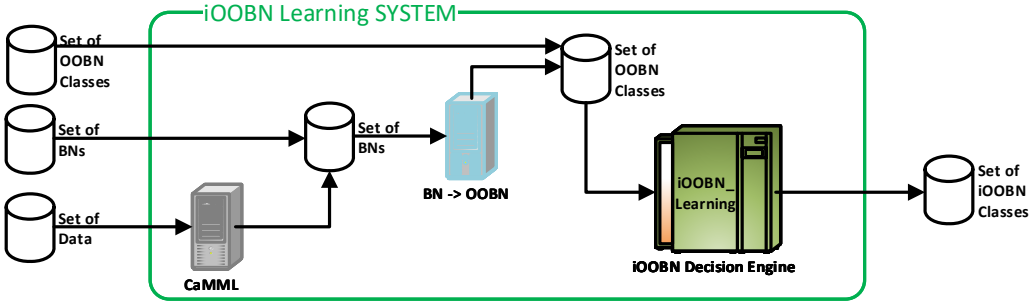


Figure A.15: Flow Diagram of the iOOBN hierarchy learning system.

A.5.6 Physical view

iOOBN software is a single-processor based system, and neither parallel processing nor parallel-distributed computing has yet been added. Hence, this aspect is not relevant to the present discussion.

A.6 Missing Features

In the current version, there are limited features. Some noteworthy omissions need to be addressed to make the software usable by general users.

- Addition of structure and parameter learning is not possible at this stage.
- OO-structure and parameter learning are missing. Since no relevant algorithm has been devised to date, it was not possible to add this feature.
- User documentation is not complete.
- Error and exception handling is yet to be properly done.
- Multiple file opening at one time in a single window is not possible.
- The internal inheritance tree is not capable of propagating changes (pushing downward or pulling upward in the hierarchy) between subclasses and superclasses.
- Sensitivity analysis is yet to be added.

Most of the limitations can be overcome by adding all of the supported features of Hugin to iOOBN. The current implementation is easily extensible. Therefore, it is quite straightforward to add more facilities to the prototype version of "iOOBN". A set of proper GUI buttons/options need to be added, and the action-listener can be used to call up appropriate functions from Hugin API functions.

A.7 Summary

The current implementation is not complete. It has a lot of issues and limitations. Exception handling needs special attention to make the user experience smooth. The whole system needs thorough testing. All the features that would make it a standalone tool are yet to be added. The next step towards the improvement of the tool is to add all the facilities provided by the Hugin API. Making the system more user-friendly would add great value to the software in the modelling arena. Developing a decision engine to avoid dependency on the Hugin tool could also be a future project.

Case Study: Western Grassland Reserve Project

This appendix contains a brief introduction to the Western Grassland Reserve (WGR) project and the figures show the hierarchies (and mapping between the classes) constructed in the iOOBN system to reengineer the "WGR" (Western Grassland Reserve) Project.

B.1 WGR Problem Domain

The State Government of Victoria in Australia is reserving 15,000 hectares of land to protect native grasslands to the west of Melbourne, to be managed by the Department of Environment and Primary Industries (DEPI). The WGR currently contains a mixture of high quality native grasslands, degraded grasslands and non-native vegetation, including improved pasture and cultivated pasture (cropland). Managing these areas for conservation will require a complex management approach involving weed control, biomass management using fire and grazing, pasture resting and restoration involving the re-introduction of native plants and their seeds. The reserve must be managed as soon as land is required, but the best management techniques are largely unknown. Thus an adaptive management approach has been taken – where management and monitoring are adjusted over time as understanding of the ecosystem's needs are better known. In order to assist adaptive management, BN technology was chosen to model ecological change in a grassland ecosystem, to provide probabilistic predictions to evaluate management actions (e.g. weed control, fire) and to justify the choice of actions to be trialled within the reserve [57].

B.2 The WGR OOBN Model

The WGR model is a complex BN, employing a number of extensions to the basic BN structure. The original system is a dynamic BN, representing the change in state variables over time, with seasonal time steps, rolled-out for a 30-year prediction window. It act as a decision network,

with decision nodes representing management options grouped into management strategies, which are sequences of actions across seasons, and utility nodes which represent the costs associated with interventions and the environmental value of the site. It needs an object-oriented model, to manage the complexity of the number of species and seasonal transitions. The WGR dynamic object-oriented BN model is presented here. The model is now deployed and being used by DEPI to: make predictions about changes in the grassland ecosystem; act as a repository of knowledge, to be updated as understanding of the grassland ecosystem changes or improves; quantitatively evaluate the ecological and financial consequences of management actions; and rank management options with the highest probability of success for trialling [57]. In Chapter 3, Section 3.5.1, a brief introductory description of the over all project and the components of the model is given. Figure B.3 shows a list of names of the 129 DOOBN classes of the original WGR project. It also shows a mapping between original WGR DOOBN classes with the first reengineered model iOOBN classes.

B.3 Reengineering the WGR OOBN in iOOBN

In order to validate the models built to trial the new iOOBN, WGR was chosen for reengineering. The reengineered WGR model was built as a component of this PhD research. The aims of the reengineering can be classified broadly as follows:

1. Fully automated learning
2. Semi-automated modelling (reengineering with background knowledge and expert elicitation)
3. Manually built model with expert elicitation

The outcomes of the last two approaches were quite identical and hence only one of them is explicated in this thesis.

The reengineered models for the WGR were implemented directly using iOOBN software (see Appendix A for the details of iOOBN software) and a GUI designed specifically for the iOOBN.

The reengineered model of WGR version-1 (figures B.1 and B.2) includes inheritance, abstraction and encapsulation. First, using the hierarchy learning approach (proposed in Chapter 5), available as a function/feature in iOOBN software, a hierarchy of the classes was learned. Initially, all 129 DOOBN classes of the original WGR were input into the hierarchy learning function. This function then constructed a hierarchy of classes (129 original classes),

abstract classes (inferred), and interfaces (inferred). This hierarchy diagram and the hierarchical relation among the iOOBN classes and interfaces were used to specify the classes needed for WGR.

Version-2 of the reengineered model (Figure B.4) used expert elicitation, background and domain knowledge with iOOBN features, such as encapsulation, abstraction, and inheritance. The inheritance hierarchy was built based on background knowledge of the project. For every different group of species of plants, separate hierarchies were created and these hierarchies (referred to as subhierarchies) were later combined in to a single hierarchy based on the similarities of the root classes of the subhierarchies. The resulting hierarchy was validated by the WGR experts and also validated by comparing it with the manual version of the hierarchy built by expert elicitation.

The hierarchies learned in the above-mentioned ways have their own merits and demerits. However, there is a common factor in both hierarchies; i.e., the automated one places emphasis on the flat similarities of the attributes of plants/classes whereas the manual one is based on expert knowledge of plant species, where attributes and genetic similarities of particular species are well understood. Hence, there were no remarkable differences between the hierarchies except for the part of the hierarchies where classes relate to managerial actions/decisions/interventions. This occurred because, in the automatic learned hierarchies, some decision network segments were classified similarly to some plant classes, as they shared some attributes in common.

The BN model describes in some detail the species composition of the grassland and how this composition is predicted to change under different management regimes. Within this adaptive management approach, the BN has been used for three primary purposes: [3]

- To make predictions about the effects of management (including cost-benefit considerations), thereby allowing us to select a small set of promising options to trial in the field from the vast array of possible management options.
- To make detailed assumptions about grassland ecology and management explicit, and thus open to criticism and improvement.
- As a tool to help in learning and a framework in which to house that learning. The BN parameters can be regularly updated as we learn more about grasslands and their management and hence strengthen the knowledge base.

In Figure B.4, another reengineered version of the WGR project is shown. This version was built after discussing the reengineering project with WGR experts from the Bayesian In-

telligence Pte Ltd (BAIPL) [1] and was also based on background knowledge of vegetation and agriculture. The modelling also exhibits most of the OO features provided by an iOOBN. This allowed comparison with the earlier version that was built from the knowledge and advice provided by WGR DELWP experts.

The learning algorithm not only constructs a hierarchical tree that has classes (abstract and concrete) and interfaces as nodes but also sketches the structure of the classes and components of the interfaces. In order to implement the reengineered WGR model, the GUI designed for the of iOOBN software was used to construct the class and interface structures. The software also has the facility to add parameters manually and share among classes. The iOOBN parameter adding/sharing facility is used to complete the class definitions. Finally, the compilation facility converts iOOBN classes into OOBN classes. The OOBN classes can then be used as input to the Hugin decision engine in order to perform inference. iOOBN software facilitates the inference by using Hugin APIs.

A point worth noting is that in all the hierarchies (figures B.1, B.2 and B.4), the actual concrete classes are not shown. These are the concrete classes where the parameters differed from the superclass (shown as leaf nodes in the hierarchy). When an iOOBN model is built for a specific problem, it will be this concrete class that is used in the overall model. In the case of the reengineered WGR, these concrete classes are the same 129 classes that existed in the original OOBN. Therefore, the need for validation is obsolete. More importantly, the reengineering is all about safe reuse of existing classes/components (structures and parameters) through the use of inheritance, encapsulation, polymorphism and type checking/typecasting.

[1] www.bayesian-intelligence.com

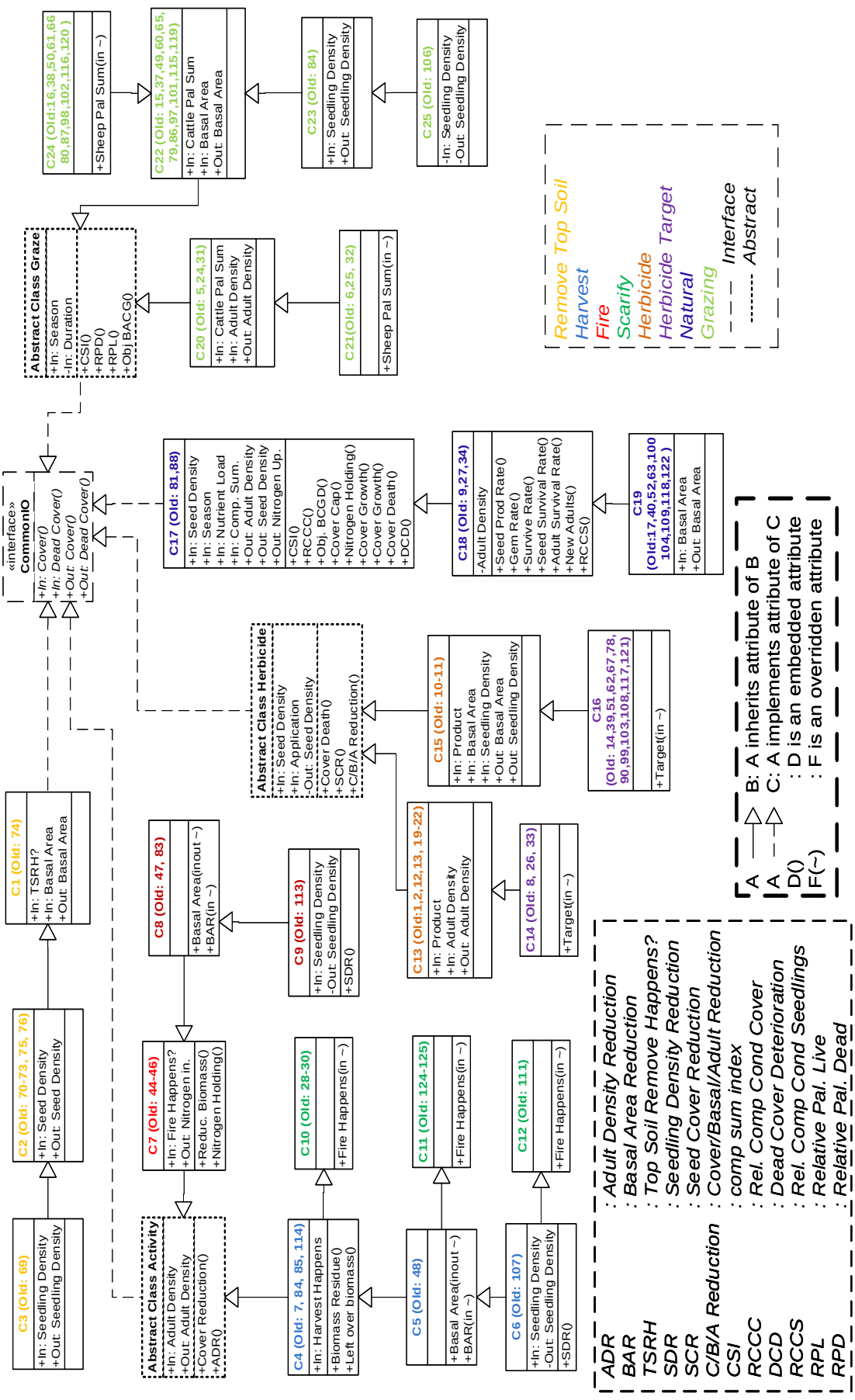


Figure B.1: Class hierarchy of the WGR reengineered (iOOBN) system learned by automated hierarchy construction algorithm

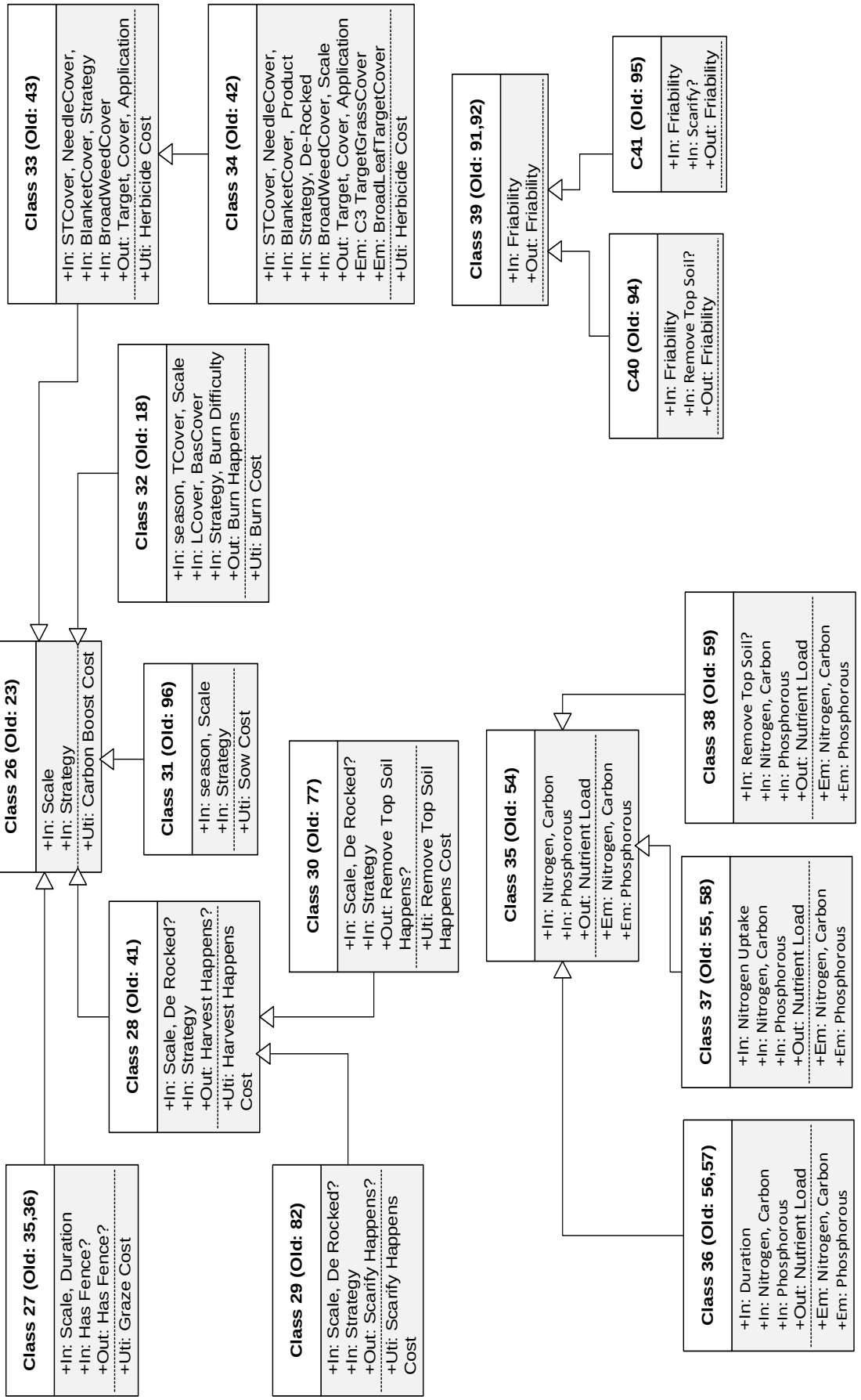


Figure B.2: Class hierarchy of the WGR reengineered (iOOBN) system - Decision and utility nodes learned by automated hierarchy construction algorithm

Mapping of class names:

New column represents the class names/numbers in **reverse engineered version** and (**Old** + **Name**) columns represent the class names in **original WGR version**
Table with **Blue** Header represents constructs those are only available **IOOBN. Dark** rows indicates "Classes **not available** in the hierarchy".

New	Old	Name
1	74	Remove Top Soil 6
2	70	Remove Top Soil 2
	71	Remove Top Soil 3
	72	Remove Top Soil 4
	73	Remove Top Soil 5
	75	Remove Top Soil 7
	76	Remove Top Soil 8
3	69	Remove Top Soil 1
4	7	Blanket Harvest
	84	Sensitive Harvest 1
	85	Sensitive Harvest 2
	114	Tolerant Harvest
5	48	Moderate Harvest
6	107	Themeda Harvest
7	44	Killed Fire1
	45	Killed Fire2
	46	Killed Fire3
8	47	Moderate Fire
	83	Sensitive Fire
9	113	Tolerant Fire
10	28	Fragile Scarify 1
	29	Fragile Scarify 2
	30	Fragile Scarify 3
11	124	Tenacious Scarify 1
	125	Tenacious Scarify 2
12	111	Themeda Scarify
13	1	Annual Grass Herbicide 1
	2	Annual Grass Herbicide 2
	12	Broad Leaf Target Herbicide 1
	13	Broad Leaf Target Herbicide 2
	19	C3 Grass Herbicide
	20	C3 Grass Target Herbicide 1
	21	C3 Grass Target Herbicide 2
	22	C4 Grass Herbicide
14	8	Blanket Herbicide Target

New	Old	Name
14	26	Exotic Annual Herbicide Target
	33	Grain Herbicide Target
15	10	Broad Leaf Herbicide 1
	11	Broad Leaf Herbicide 2
16	14	Broad Weed Herbicide Target
	39	Hardy Native Herbicide Target
	51	Needle Herbicide Target
	62	Onion Herbicide Target
	67	Red Leg Herbicide Target
	78	Ruder Herbicide Target
	90	Sensi Native Herbicide Target
	99	Spear Herbicide Target
	103	ST Herbicide Target
	108	Themeda Herbicide Target
	117	Wallaby Herbicide Target
	121	Windmill Herbicide Target
17	81	Ruder Natural
	88	Sensi Native Natural
18	9	Blanket Natural
	27	Exotic Annual Natural
	34	Grain Natural
19	17	Broad Weeds Natural
	40	Hardy Native Natural
	52	Needle Natural
	63	Onion Natural
	68	Red Leg Natural
	100	Spear Natural
	104	ST Natural
	109	Themeda Natural
	118	Wallaby Natural
	122	Windmill Natural
20	5	Blanket Graze Cattle
	24	Exotic Annual Graze Cattle
	31	Grain Graze Cattle

New	Old	Name
21	6	Blanket Graze Sheep
	25	Exotic Annual Graze Sheep
	32	Grain Graze Sheep (*)
22	15	Broad Weed Graze Cattle
	37	Hardy Native Graze Cattle
	49	Needle Graze Cattle
	60	Onion Graze Cattle
	65	Red Leg Graze Cattle
	79	Ruder Graze Cattle
	86	Sensit Native Graze Cattle
	97	Spear Graze Cattle
	101	ST Graze Cattle
	115	Wallaby Graze Cattle
	119	Windmill Graze Cattle
23	105	Themeda Graze Cattle
24	16	Broad Weed Graze Sheep
	38	Hardy Native Graze Sheep
	50	Needle Graze Sheep
	61	Onion Graze Sheep
	66	Red Leg Graze Sheep
	80	Ruder Graze Sheep
	87	Sensit Native Graze Sheep
	98	Spear Graze Sheep
	102	ST Graze Sheep
	116	Wallaby Graze Sheep
	120	Windmill Graze Sheep
25	106	Themeda Graze Sheep

Constructs	Name
Interface	CommonIO
Abstract Classes	Activity
	Graze
	Herbicide

New	Old	Name
26	23	Carbon Boost
27	35	Graze Intervention Cattle
	36	Graze Intervention Sheep
28	41	Harvest Intervention
29	82	Scarify Intervention
30	77	Remove Top Soil Intervention
31	96	Sow Intervention
32	18	Burn Intervention
33	43	Herbicide Intervention Target
34	42	Herbicide Intervention
35	54	Nitrogen Carbon Boost
36	56	Nutrient Graze Cattle
	57	Nutrient Graze Sheep
37	55	Nutrient Fire
	58	Nutrient Natural
38	59	Nutrient Remove Top Soil
39	91	Soil Graze Cattle
	92	Soil Graze Sheep
40	94	Soil Remove Top Soil
41	95	Soil Scarify
42	3	Basal Adult Cover Graze (BACG)
43	4	Biomass Natural
44	64	Plant Intervention
45	53	Nui Adder
46	89	Sensitive Native Sow
47	93	Soil Natural
48	110	Themeda Plant
49	112	Themeda Sow
50	123	Basal Cover Growth Death (BCGD)
51	126	Biomass Summaries
52	127	Main
53	128	EnvValue
54	129	Nutrient Harvest [Found Blank]

Figure B.3: Mapping of the WGR (original) classes and the WGR reengineered (iOOBN) classes

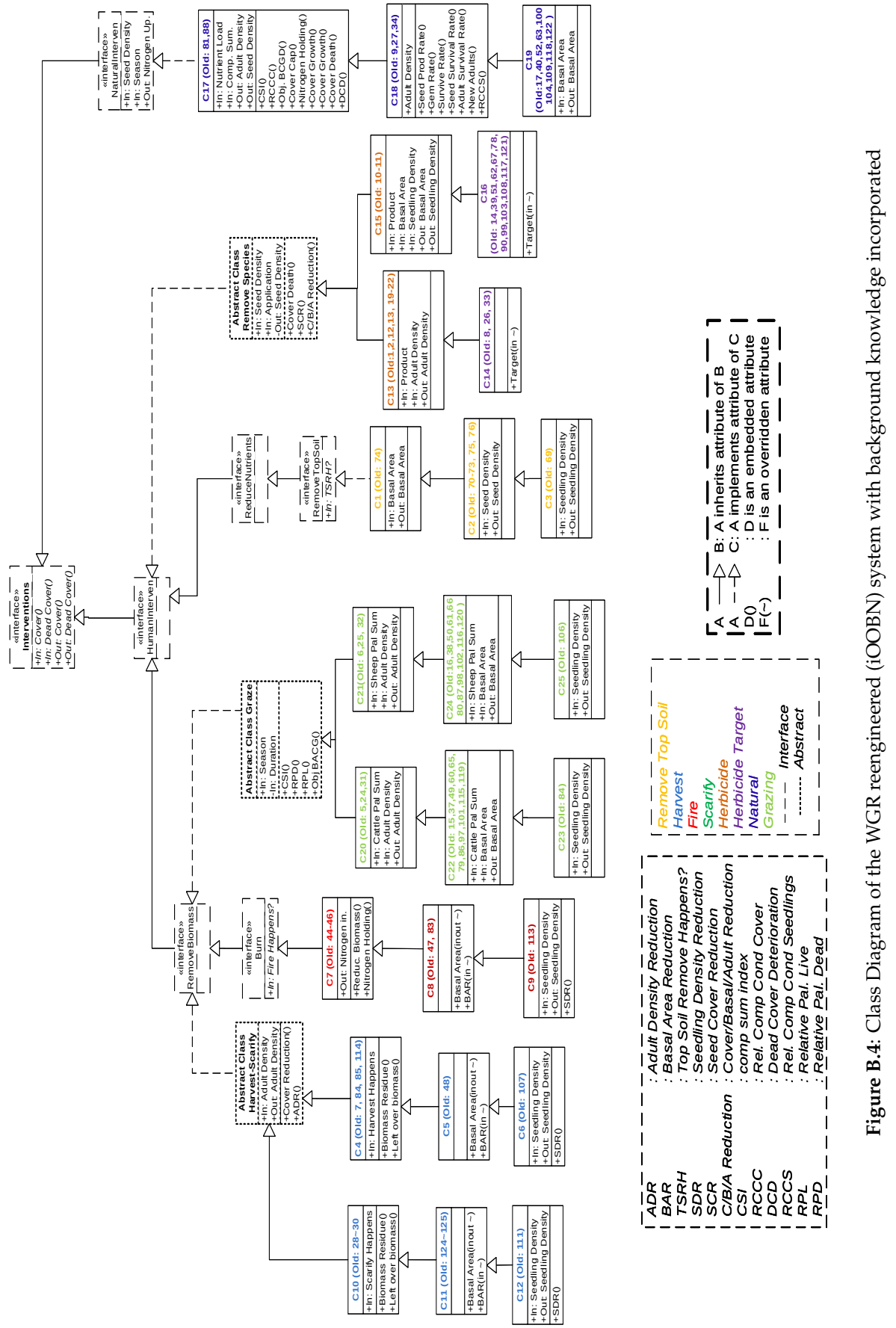


Figure B.4: Class Diagram of the WGR reengineered (iOOBN) system with background knowledge incorporated

Extended Experiments of Compilation Algorithms

C.1 Performance Analysis of the Proposed Algorithm

This section outlines the extended experimental comparison of the existing and proposed compilation algorithms. To analyse the performance of the proposed algorithms, linear regression models, ANOVA, Mean, and the Standard deviation of several runs and other standard statistical approaches were applied. The following two subsections contain, respectively, an analysis of the algorithms on various combinations of parameters and a comparison of the performance of the algorithms according to various factors.

Table C.1: The models built to perform analysis

#Model	Details	Parameters
1	Linear Regression Model For all OOBNs $Diff \sim NOO + NON + NOC + NOS + NOP + NOPAvg$	(1) $Diff = log(HuginTime) - log(SIICTime)$ (2) NOO (3) NOC (4) NON (5) NOS (6) NOP (7) NOPAvg
2	Linear Regression Model For all OOBNs $Diff \sim NOO + NON + NOS$	(1) $Diff = log(HuginTime) - log(SIICTime)$ (2) NOO (3) NON(4) NOS
3	Linear Regression Model For all OOBNs $Diff \sim NON + NOS$	(1) $Diff = log(HuginTime) - log(SIICTime)$ (2) NON (3) NOS
4	Linear Regression Model For pure OOBNs $Diff \sim NOO + NON + NOC + NOS + NOP + NOPAvg$	(1) $Diff = log(HuginTime) - log(SIICTime)$ (2) NOO (3) NOC (4) NON (5) NOS (6) NOP (7) NOPAvg
5	Linear Regression Model For pure OOBNs $Diff \sim NOO + NON + NOC + NOS + NOP$	(1) $Diff = log(HuginTime) - log(SIICTime)$ (2) NOO (3) NOC (4) NON (5) NOS (6) NOP
6	Linear Regression Model For all OOBNs $Diff \sim NOO + NON + NOC + NOS + NOP + NOPAvg$	(1) $Diff = log(HuginTime) - log(SIIC\#Time)$ (2) NOO (3) NOC (4) NON (5) NOS (6) NOP (7) NOPAvg
7	Linear Regression Model For all OOBNs $Diff \sim NOO + NOC + NOS + NOP + NOPAvg$	(1) $Diff = log(HuginTime) - log(SIIC\#Time)$ (2) NOO (3) NOC (4) NOS (5) NOP (6) NOPAvg
8	Linear Regression Model For pure OOBNs $Diff \sim NOO + NON + NOC + NOS + NOP + NOPAvg$	(1) $Diff = log(HuginTime) - log(SIIC\#Time)$ (2) NOO (3) NOC (4) NON (5) NOS (6) NOP (7) NOPAvg
9	Linear Regression Model For pure OOBNs $Diff \sim NOO + NOC + NOS + NOP + NOPAvg$	(1) $Diff = log(HuginTime) - log(SIIC\#Time)$ (2) NOO (3) NOC (4) NOS (5) NOP (6) NOPAvg

The nine linear models; built for statistical analysis of the parameters NON, NOO, NOS, NOP, $NOPAvg$ (see Table 4.2 for the full terms), and outcomes of the algorithm's JT (generated by Hugin and SIIC) cost, complexity of the OOBN class that was compiled and the running times of SIIC, SIIC#, and Hugin; are listed in Table C.1. ANOVA was applied on the following pairs of models: Model 1, Model 2; Model 2, Model 3; Model 4, Model 5; Model 6, Model 7; and Model 8, Model 9. The outcome of ANOVA was carefully observed to see the effect of input parameters on the outputs. The parameters and the outcomes were correlated significantly and then analysed and plotted as illustrated in the following subsections.

C.2 Performance of Hugin, SIIC and SIIC# Algorithms

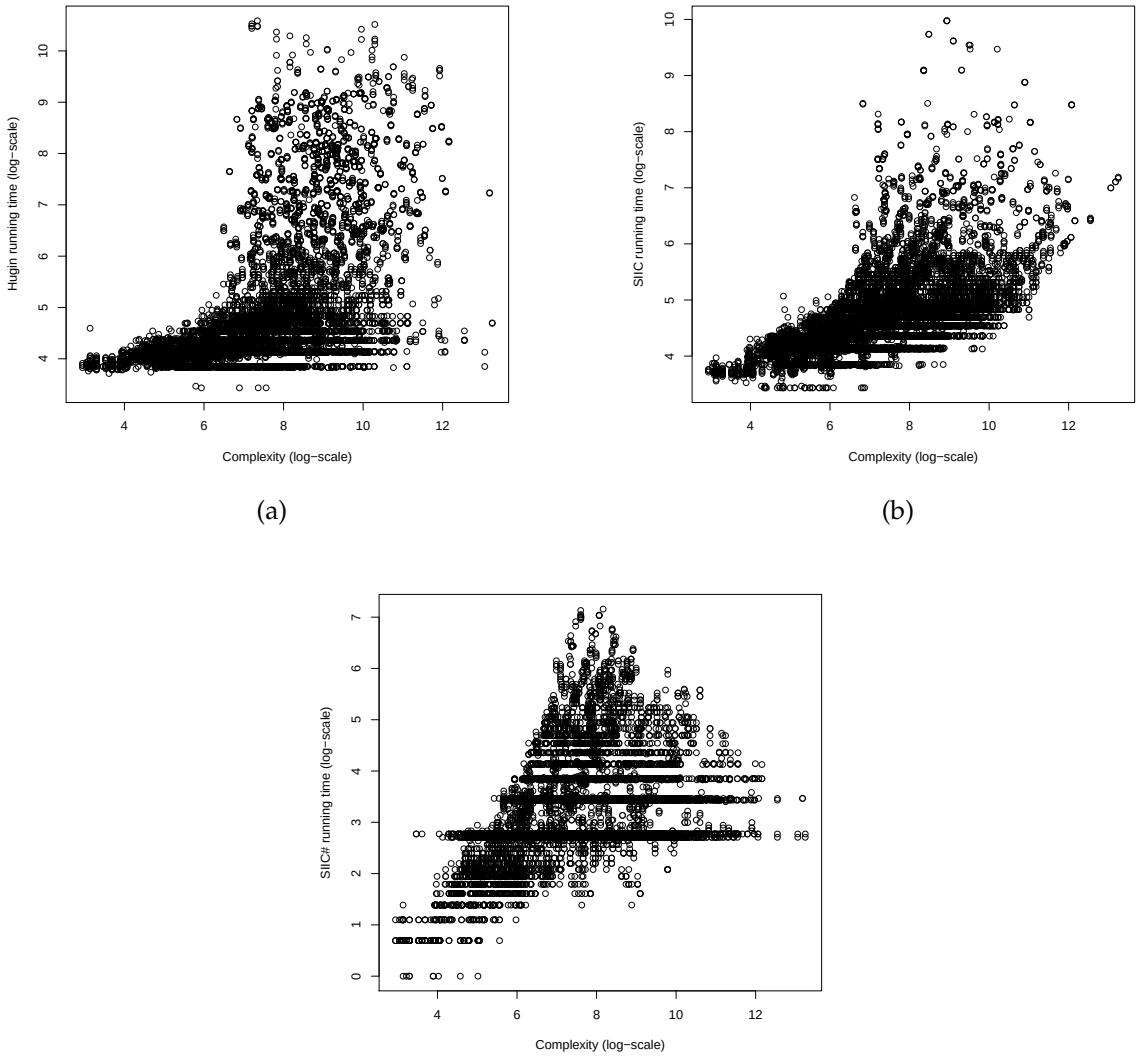
This section outlines the extended experimental comparison of the existing (Hugin JT construction) and proposed compilation (SIIC and SIIC#) algorithms. The contents and figures of those that have less evidence of correlation to the performance comparison are placed in this section.

C.2.1 Time required to compile OOBNs

The running times of the compilation algorithms have a strong correlation with NOP of the OOBN classes. If NOP increases, the complexity of the classes, and hence the running time of the compilation, increases. However, the experimentation exhibits a somewhat strange effect of NOP on the running time, as shown in figures C.2a, C.2b and C.2c, respectively, for Hugin, SIIC and SIIC#. For SIIC, the impact is almost consistent; however, for Hugin and SIIC#, there is a discrepancy in the median of the running time. The reason for this inconsistency is that NOP is the maximum number of parents in a synthetic OOBN class which does not guarantee that all the nodes have the same maximum number of parents. The synthetic OOBN class may also have different density, and hence have a different degree of complexity, which could contribute to the inconsistency.

The complexity (a term defined in [216] for BN and generalized in Section 4.5.1 for OOBNs) of an OOBN class plays a significant role in the running time of the compilation algorithms. Figures C.1a, C.1b and C.1c show performance analyses of the algorithms in terms of running time with respect to OOBN complexity. The plots indicate that Hugin running time increased sharply with increase in complexity; SIIC running time also increased, though the increment was not as sharp as for Hugin. The increase of SIIC#'s running time with respect to an increase in complexity was significantly less than for Hugin or SIIC.

Figures C.2d, C.2e and C.2f demonstrate the performance of Hugin, SIIC and SIIC# al-



(a) (b)

(c)

Figure C.1: Complexity vs Runtime: (a) Hugin, (b) SIIC, and (c) SIIC#

gorithm in terms of running time with respect to varying NOS. In Hugin the running time increased with the increase in NOS. For SIIC, the NOS had almost no impact on the running time and the running time decreased a little in SIIC#. This strange strange outcome is found because for SIIC and SIIC# when the value of NOS goes up, the algorithms fail to compile because their "helper" Hugin compilation fails. The notion of helper means, as discussed earlier, that SIIC and SIIC# use Hugin for compiling classes with no embedded objects. Therefore, for the observed experimental results, Hugin can show a true correlation, but SIIC and SIIC# show a false correlation due to having little evidence for higher NOS values.

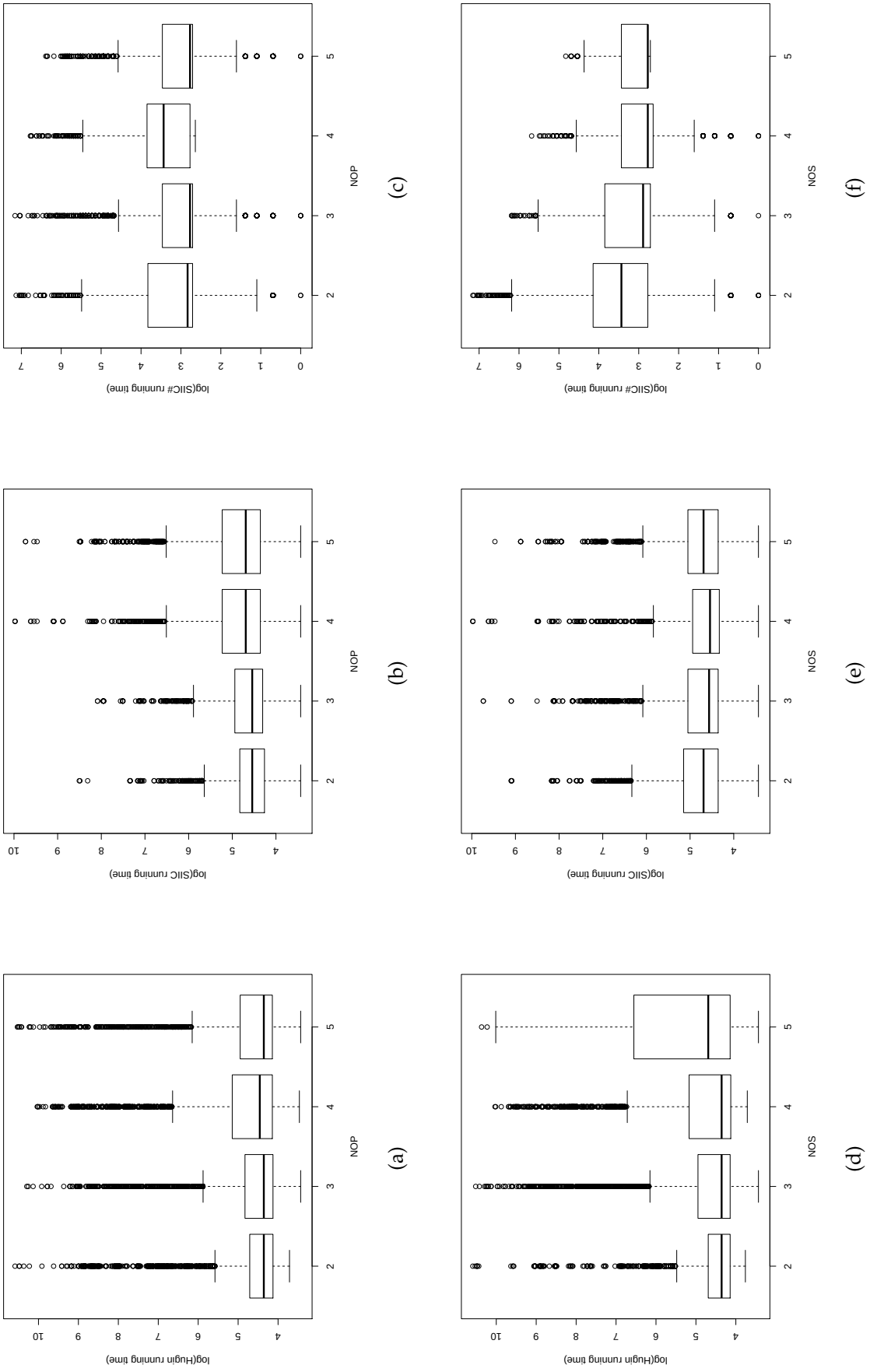


Figure C.2: Running time w.r.t. NOP: (a) Hugin, (b) SIIC, and (c) SIIC#; Running time w.r.t. NOS: (d) Hugin, (e) SIIC, and (f) SIIC#

C.2.2 Comparing costs of the JTs produced by Hugin and SIIC

Figure C.3a compares the running time of Hugin with SIIC. It is easy to infer that SIIC is significantly faster than Hugin. For ease of understanding, the reference line is drawn in pink.

Figures C.3b to C.3f compare the running time of Hugin and SIIC with respect to varying values of NOC (in Figure C.3b) , NON (in Figure C.3c), NOO (in Figure C.3d), NOP (in Figure C.3e), and NOS (in Figure C.3f). The aforementioned figures stipulate that for increasing value of NOC, NON, NOO, NOP and NOS, SIIC is superior to Hugin in terms of running time.

Figure C.4a compares the running time of Hugin with SIIC#. The comparison indicates that SIIC# is significantly better than Hugin in terms of running time. For ease of understanding, the reference line is drawn in pink.

The running time performance of SIIC# and Hugin algorithms are compared in terms of varying values of NOC, NON, NOO, NOP and NOS, respectively, in figures C.4b to C.4f. The fact stipulated from the aforementioned analyses is that for increasing value of NOC, NON, NOO, NOP and NOS, SIIC# performs way better than Hugin in terms of running time.

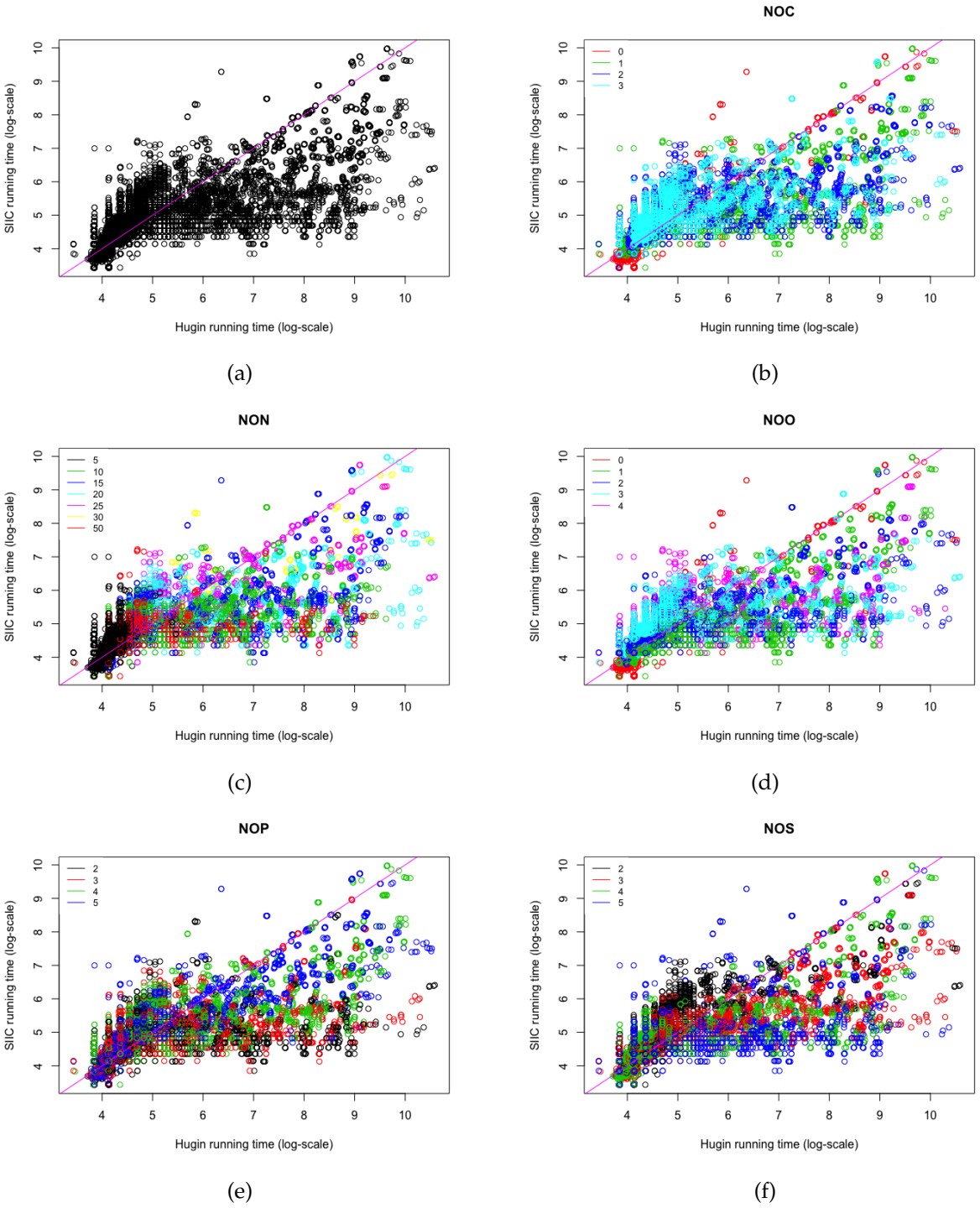


Figure C.3: Hugin vs SIIC running time, (a) overall distribution; and w.r.t. (b) NOC, (c) NON, (d) NOO, (e) NOP, and (f) NOS.

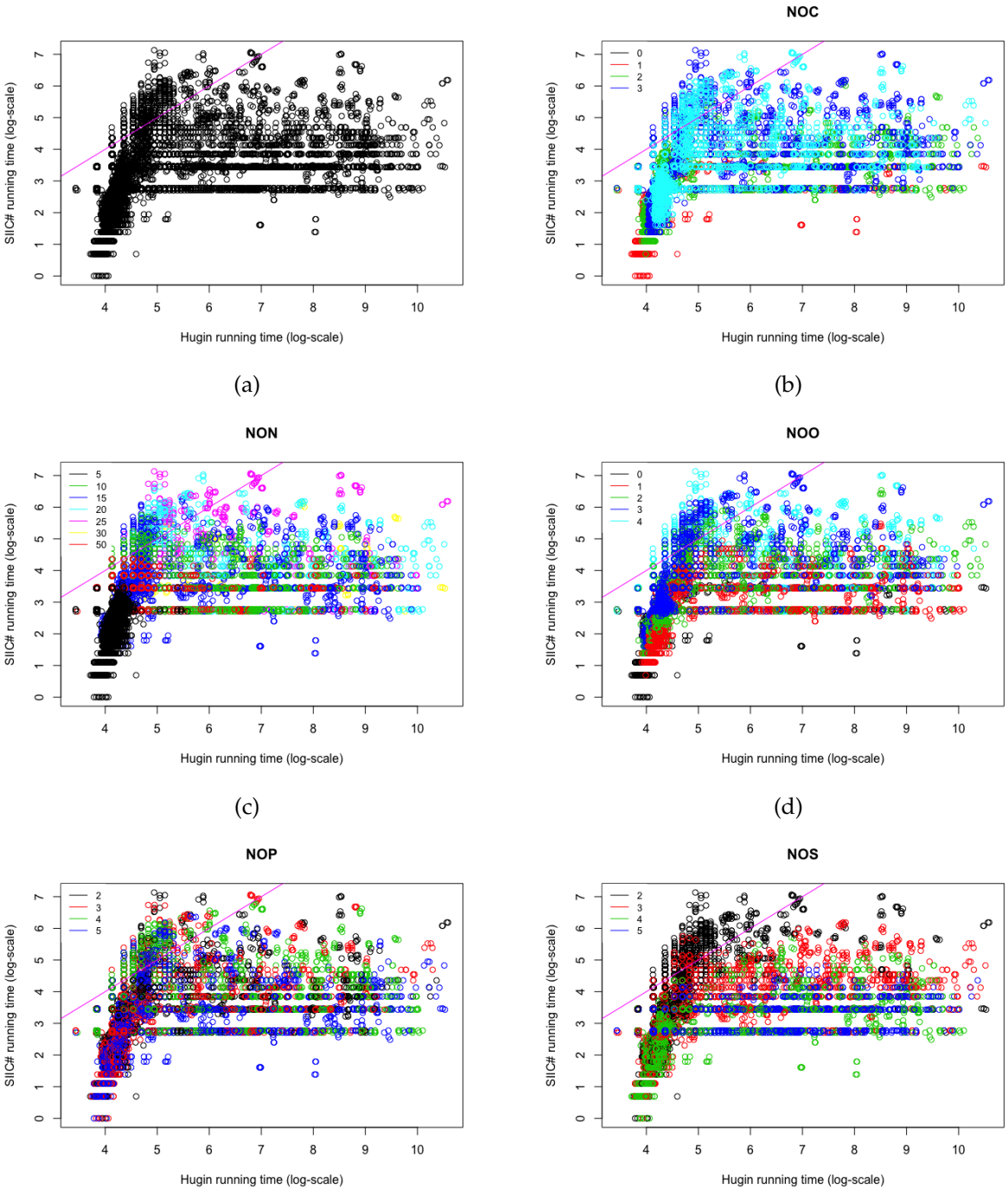


Figure C.4: Hugin vs SIIC# running time, (a) overall distribution; and w.r.t. (b) NOC, (c) NON, (d) NOO, (e) NOP, and (f) NOS.

Appendix D

Hierarchy Learning Case Study: Western Grassland Reserve Project

As a proof-of-concept case study, the real-life project "WGR" (Western Grassland Reserve in Melbourne, Australia) [3], used for the iOOBN framework viability checking in Section 3.5 of Chapter 3, was chosen again.

There are 129 OOBN classes in WGR. The extracted DAGs of the classes contain a total of 3993 nodes and edges. The proposed learning algorithm (in Chapter 5) constructs a hierarchy (as shown in Figure D.1) taking the 129 DAGs as input. The hierarchy suggests 159 DAGs each for a potential "iOOBN" class with a total number of 2135 nodes and edges, provided that, inheritance and maximum reusability are considered.

Reusability and inheritance not only provide scalability in modelling large applications but also reduces the time required for compilation and inference, because the SIIC compilation algorithm (proposed in Chapter 4) allows reusing the junction tree of parent-classes or super-classes as well as replicating JTs for instances of classes. The approach saves time by avoiding redundant construction of JTs.

Figure D.2 presents a mapping between the input DAGs to the learning algorithm (first column) and symbols used by the algorithm to represent the DAGs in constructing a supergraph (second column). The supergraph is used by the learning algorithm to build a hierarchy, as shown in Figure D.1. The DAGs in the hierarchy are represented by another set of symbols. Figure D.3 shows a map between the DAGs in the learned hierarchy represented by uppercase symbols (first column) and a set of symbols of the input DAGs (second column). The set of symbols in the second column represents a DAG derived from the supergraph. Since the WGR is a project owned by DELWP, Australia, the structures or parameters of the classes are not publicly available and cannot be shared. Therefore, the actual structural information of the DAGs, the super-DAG and the parameters of the WGR or reengineered WGR classes are not specified in this thesis.

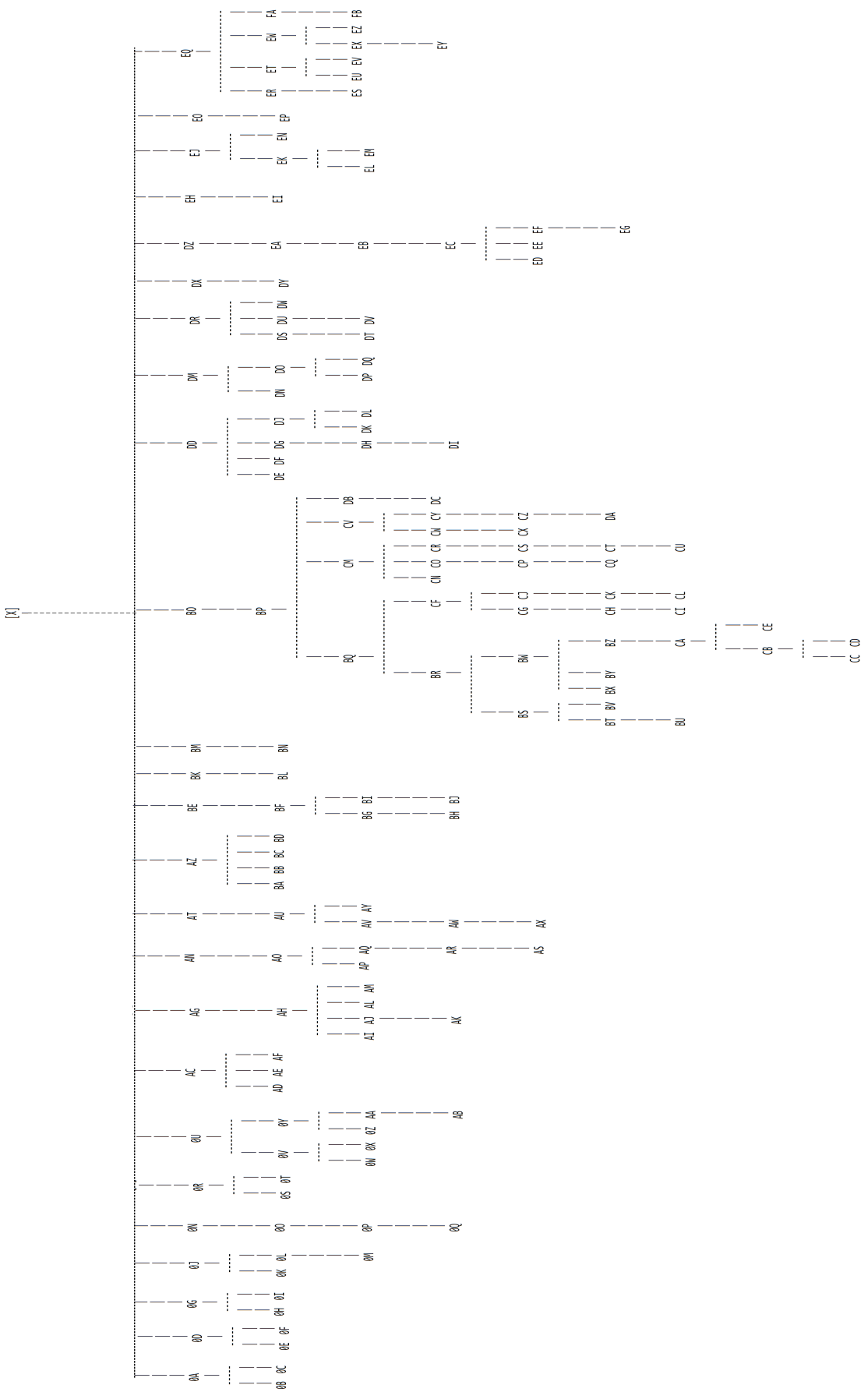


Figure D.1: Learning outcome: The class hierarchy of WGR

DAG Label	Input File Name	DAG Label	Input File Name	DAG Label	Input File Name	DAG Label	Input File Name	DAG Label	Input File Name
0a	dag_AnnualGrassHerbicide1	ac	dag_ExoticAnnualHerbicideTarget	be	dag_NuiAdder	cg	dag_RuderHerbicideTarget	di	dag_ThemedaGrazeSheep
0b	dag_AnnualGrassHerbicide2	ad	dag_ExoticAnnualNatural	bf	dag_NutrientCarbonBoost	ch	dag_RuderNatural	dj	dag_ThemedaHarvest
0c	dag_BasalAdultCoverGraze	ae	dag_FragileScarify1	bg	dag_NutrientFire	ci	dag_ScarifyIntervention	dk	dag_ThemedaHerbicideTarget
0d	dag_BasalCoverGrowthDeath	af	dag_FragileScarify2	bh	dag_NutrientGrazeCattle	cj	dag_SensitiveFire	dl	dag_ThemedaNatural
0e	dag_BiomassNatural	ag	dag_FragileScarify3	bi	dag_NutrientGrazeSheep	ck	dag_SensitiveHarvest	dm	dag_ThemedaPlant
0f	dag_BiomassSummaries	ah	dag_GrainGrazeCattle	bj	dag_NutrientHarvest	cl	dag_SensitiveHarvest2	dn	dag_ThemedaScarify
0g	dag_BlanketGrazeCattle	ai	dag_GrainGrazeSheep	bk	dag_NutrientNatural	cm	dag_SensitiveNativeSow	do	dag_ThemedaSow
0h	dag_BlanketGrazeSheep	aj	dag_GrainHerbicideTarget	bl	dag_NutrientRemoveTopSoil	cn	dag_SensitNativeGrazeCattle	dp	dag_TolerantFire
0i	dag_BlanketHarvest	ak	dag_GrainNatural	bm	dag_OnionGrazeCattle	co	dag_SensitNativeGrazeSheep	dq	dag_TolerantHarvest
0j	dag_BlanketHerbicideTarget	al	dag_GrazeInterventionCattle	bn	dag_OnionGrazeSheep	cp	dag_SensitNativeHerbicideTarget	dr	dag_WallabyGrazeCattle
0k	dag_BlanketNatural	am	dag_GrazeInterventionSheep	bo	dag_OnionHerbicideTarget	cq	dag_SensitNativeNatural	ds	dag_WallabyGrazeSheep
0l	dag_BroadleafHerbicide1	an	dag_HardyNativeGrazeCattle	bp	dag_OnionNatural	cr	dag_SoilGrazeCattle	dt	dag_WallabyHerbicideTarget
0m	dag_BroadleafHerbicide2	ao	dag_HardyNativeGrazeSheep	bq	dag_PlantIntervention	cs	dag_SoilGrazeSheep	du	dag_WallabyNatural
0n	dag_BroadleafTargetHerbicide1	ap	dag_HardyNativeHerbicideTarget	br	dag_RedLegGrazeCattle	ct	dag_SoilNatural	dv	dag_WindmillGrazeCattle
0o	dag_BroadleafTargetHerbicide2	aq	dag_HardyNativeNatural	bs	dag_RedLegGrazeSheep	cu	dag_SoilRemoveTopSoil	dw	dag_WindmillGrazeSheep
0p	dag_BroadWeedsGrazeCattle	ar	dag_HarvestIntervention	bt	dag_RedLegHerbicideTarget	cv	dag_SoilScarify	dx	dag_WindmillHerbicideTarget
0q	dag_BroadWeedsGrazeSheep	as	dag_HerbicideIntervention	bu	dag_RedLegNatural	cw	dag_SowIntervention	dy	dag_WindmillNatural
0r	dag_BroadWeedsHerbicideTarget	at	dag_HerbicideInterventionTarget	bv	dag_RemoveTopSoil1	cx	dag_SpearGrazeCattle		
0s	dag_BroadWeedsNatural	au	dag_KilledFire1	bw	dag_RemoveTopSoil2	cy	dag_SpearGrazeSheep		
0t	dag_BurnIntervention	av	dag_KilledFire2	bx	dag_RemoveTopSoil3	cz	dag_SpearHerbicideTarget		
0u	dag_C3GrassHerbicide	aw	dag_KilledFire3	by	dag_RemoveTopSoil4	da	dag_SpearNatural		
0v	dag_C3GrassTargetHerbicide1	ax	dag_main	bz	dag_RemoveTopSoil5	db	dag_STGrazeCattle		
0w	dag_C3GrassTargetHerbicide2	ay	dag_ModerateFire	ca	dag_RemoveTopSoil6	dc	dag_STGrazeSheep		
0x	dag_C4GrassHerbicide	az	dag_ModerateHarvest	cb	dag_RemoveTopSoil7	dd	dag_STHerbicideTarget		
0y	dag_CarbonBoostIntervention	ba	dag_NeedleGrazeCattle	cc	dag_RemoveTopSoil8	de	dag_STNatural		
0z	dag_EnvValue	bb	dag_NeedleGrazeSheep	cd	dag_RemoveTopSoilIntervention	df	dag_TenaciousScarify1		
aa	dag_ExoticAnnualGrazeCattle	bc	dag_NeedleHerbicideTarget	ce	dag_RuderGrazeCattle	dg	dag_TenaciousScarify2		
ab	dag_ExoticAnnualGrazeSheep	bd	dag_NeedleNatural	cf	dag_RuderGrazeSheep	dh	dag_ThemedaGrazeCattle		

Figure D.2: Mapping of WGR class names and labels in the hierarchy

Notation	Label Name	Notation	Label Name	Notation	Label Name	Notation	Label Name	Notation	Label Name
0A	aq, de, du, bd, 0s, ad, bu, dy, ak, 0k, da, bp	AN	bb, dh, di, bm, bn, an, ao, dr, ds, br, bs, dv, dw, cx, cy, db, ba, dc	BS	cw, cm, do	CZ	dd, bc, de, 0r, bd, df, 0s, 0u, 0v, dj, 0w, dk, 0x, dn, dp, bo, bp, dt, du, bt, bu, dx, dy, ac, cg, aj, cp, ap, aq, 0a, 0b, cz, 0j, 0k, 0l, 0m, da, 0n, 0o	DW	dd, bc, 0r, bd, dg, 0u, 0v, 0w, dk, 0x, dl, dn, bo, bp, dt, bt, dx, dy, ac, ae, cg, af, ag, aj, ak, cp, cq, ap, aq, 0a, 0b, cz, 0j, 0l, 0m, 0n, 0o
				BT	cm				
0B	ar, cd, as, 0t, cw, ci, ax, 0y, al, am, bq	AO	bg	BU	0e, 0f				
		AP	dd, bc, 0r, bd, df, 0s, dg, 0u, 0v, 0w, dk, 0x, dl, dn, bo, bp, dt, du, bt, bu, dx, dy, ac, ae, cg, af, ch, ag, aj, cp, cq, ap, aq, 0a, 0b, cz, 0j, 0k, 0l, 0m, da, 0n, 0o	BV	0f, ax				
0C	dh, di			BW	cc, bv, bw, bx, by, bz, cb	DA	bg, bk		
0D	ck, 0i, cl			BX	bb, 0q, ab, cf, di, ai, bn, co, ao, ds, bs, dw, 0e, cy, 0h, dc	DB	aq, du, bd, 0s, bu, ch, dy, dl, 0k, da, bp, cq	DX	aq, de, du, dy, dl, bp
0E	df, dn							DY	dd, bc, 0r, ac, cg, dk, aj, bo, cp, ap, dt, as, bt, at, dx, cz, 0j
0F	cc, by, bz	AQ	0y	BY	0g, 0h	DC	aq, de, du, bp		
0G	ag, cj, ay, az, dn, dp	AR	dd, bc, 0r, ac, cg, dk, aj, bo, cp, ap, dt, bt, dx, cz, 0j	BZ	ct	DD	aa, ab, ah, ai	DZ	dj, 0i, cl, dq
0H	de, du, bd, ad, bu, dy, 0k, da			CA	cu	DE	de, bd, 0s, ad, ch, dl, ak, bp, cq, aq, du, bu, 0e, dy, 0k, da	EA	0f, ax, 0z
0I	as, at, 0f, ax, 0z	AS	0z	CB	de, bd, 0s, ad, bf, bg, ch, bh, bi, dl, bk, ak, bl, bp, cq, aq, du, bu, dy, 0f, ax, 0k, da			EB	au, av, dj, cj, ay, 0i, cl, dp
0J	0p, 0q, ce, cf, cn, co	AT	bl			DF	au, av, cj, ay, dp	EC	dd, bc, 0r, 0u, 0v, 0w, dk, 0x, bo, dt, bt, dx, ac, cg, aj, cp, ap, 0a, 0b, cz, 0j, 0l, 0m, 0n, 0o
0K	de, du, bd, da, bp	AU	dj, az, dq			DG	0p, bb, 0q, dh, di, bm, bn, dr, ds, br, bs, dv, dw, aa, ab, ce, cf, ah, ai, cn, co, an, ao, 0c, cx, cy, 0g, 0h, db, ba, dc		
0L	dd, bc, 0r, 0u, 0v, 0w, dk, 0x, bo, dt, bt, dx, ac, cg, aj, cp, ap, 0a, 0b, as, at, cz, 0j, 0l, 0m, 0n, 0o	AV	ad, ak	CC	0a, 0b, 0u, 0v, 0w, 0x, 0l, 0m, 0n, 0o				
		AW	bq	CD	cv			ED	bv, bw, bx, ca, cb
		AX	df, dg, ae, af, ag, dj, cj, ck, cl, dn, dp, dq, 0c, au, av, aw, ay, 0i, az	CE	cw			EE	0d, dl
0M	dd, bc, 0r, df, dh, 0u, di, 0v, dj, 0w, dk, 0x, dl, dm, dn, dp, bo, dt, bt, bv, dx, cg, cp, ap, cz, 0l, 0m, 0o			CF	dg, ae, au, af, av, aw, ck, 0i, cl	DH	dy	EF	0p, bb, 0q, de, bd, 0s, dh, di, dl, bm, bn, bp, dr, ds, br, bs, du, dv, bu, dw, dy, aa, ab, ce, ad, cf, ch, ah, ai, ak, cn, co, an, ao, cq, aq, cx, cy, 0g, 0h, 0k, da, db, ba, dc
0N	aq, de, du, dy, bp	AY	de, dh, di, dl, dr, ds, du, dv, dw, dy, aa, ab, ad, ah, ai, ak, an, ao, aq, 0d, 0f, 0g, 0h, 0k, ba, 0p, bb, 0q, bd, 0s, 0t, bm, bn, bp, br, bs, bu, ce, cf, ch, cn, co, cq, ct, cw, cx, cy, da, db, dc	CG	0p, bb, 0q, dh, di, bh, bi, bm, bn, dr, ds, br, bs, dv, dw, aa, ab, ce, cf, ah, ai, al, cn, am, co, an, ao, 0c, cx, cy, 0g, 0h, db, ba, dc	DI	dd, bc, 0r, bd, df, 0s, dg, 0u, 0v, 0w, dk, 0x, dl, dn, bo, bp, dt, du, bt, bu, dx, dy, ac, ad, ae, cg, af, ch, ag, aj, ak, cp, cq, ap, aq, 0a, 0b, cz, 0j, 0k, 0l, 0m, da, 0n, 0o		
0O	de, bd, 0s, ad, ch, cj, dl, ak, dp, bp, cq, aq, du, bu, au, av, dy, aw, ay, 0k, da			CH	aq, de, du, bd, 0s, bu, dy, da, bp			EG	ax, dl
		AZ	bu	CI	0t, 0f			EH	bd, da
0P	0p, aa, ce, dh, ah, bm, cn, an, dr, br, dv, cx, 0g, db, ba	BA	0t, au, av, bg, aw, cj, ay, dp	CJ	0t, ax	DJ	dd, bc, 0r, df, dh, 0u, di, 0v, dj, 0w, dk, 0x, dm, dn, dp, bo, dt, bt, bv, dx, cg, cp, ap, cz, 0l, 0m, 0o	EI	aq, 0s, bp
		BB	bv	CK	dd, de, df, dj, dk, dq, dr, ds, dt, du, dv, dw, dx, an, ao, ap, aq, 0l, 0m, 0o, ba, 0p, bb, 0q, bc, 0r, 0u, 0v, 0w, 0x, bm, bn, bo, bp, br, bs, bt, ce, cf, cg, ch, cn, co, cp, cx, cy, cz, db, dc			EJ	cs, cu, cv, cr
0Q	ak	BC	aq, bd, dy, bp					EK	dd, bc, de, 0r, bd, 0s, 0u, 0v, 0w, dk, 0x, dl, dn, do, bo, bp, dt, du, bt, bu, bv, dx, bw, dy, bx, by, bz, cb, cc, ac, ad, cg, aj, ak, cp, ap, aq, 0a, 0b, cz, 0j, 0k, 0l, 0m, da, 0n, 0o
0R	cj, ay, dp	BD	dj, ck, 0i, az, cl, dq			DK	dj, dn, dp		
0S	dg, by, bz, aa, cc, ab, ac, ad, ae, af, ah, ai, ck, aj, cl, ak, 0a, 0b, au, av, aw, 0g, 0h, 0i, 0j, 0k, 0n	BE	al, am			DL	aq, cq		
		BF	de, bd, 0s, ad, ch, dl, ak, bp, cq, aq, du, bu, dy, 0k, da			DM	du, bd, da, bp		
				CL	de	DN	aw, cl		
0T	ar	BG	ad, 0k	CM	df	DO	dd, de, df, dh, di, dj, dk, dl, dn, dp, dq, dr, ds, dt, du, dv, dw, dx, dy, ag, an, ao, ap, aq, ay, az, 0l, 0m, 0o, ba, 0p, bb, 0q, bc, 0r, bd, 0s, 0u, 0v, 0w, 0x, bm, bn, bo, bp, br, bs, bt, bu, bv, bw, bx, ca, cb, ce, cf, cg, ch, cj, cm, cn, co, cp, cq, cx, cy, cz, da, db, dc	EL	au, av, aw, cj, ay, dp
0U	as	BH	0p, aa, ce, dh, ah, bm, cn, an, dr, br, dv, 0e, cx, 0g, db, ba	CN	dd, de, df, dg, dh, di, dj, dk, dl, dn, dp, dq, dr, ds, dt, du, dv, dw, dx, dy, ac, ae, af, ag, aj, an, ao, ap, aq, 0a, 0b, au, av, aw, 0g, 0h, ay, 0i, az, 0j, 0k, 0l, 0m, 0n, 0o, ba, 0p, bb, 0q, bc, 0r, bd, 0s, 0u, 0v, 0w, 0x, bm, bn, bo, bp, br, bs, bt, bu, bv, bw, bx, by, bz, ca, cb, cc, ce, cf, cg, ch, cj, ck, cl, cn, co, cp, cq, cx, cy, cz, da, db, dc			EM	cc, bv, bw, bx, by, bz, ca, cb
0V	df, ag, dn							EN	dg, ae, af
0W	de, bd, 0s, ad, ch, dl, bp, cq, aq, du, bu, dy, 0k, da	BI	aq, du, 0s, dl, ak, 0k, cq					EO	0p, bb, 0q, dh, di, bm, bn, dr, ds, br, bs, dv, dw, ce, cf, cn, co, an, ao, cx, cy, 0g, 0h, db, ba, dc
0X	0c	BJ	0p, bb, 0q, dh, di, bm, bn, dr, ds, br, bs, dv, dw, aa, ab, ce, cf, ah, ai, cn, co, an, ao, cx, cy, 0g, 0h, db, ba, dc						
0Y	at								
0Z	0d	BK	dj, ck, 0i, az					EP	be, bg, bk
AA	0e	BL	aq, de, du, bd, 0s, ad, bu, dy, dl, 0k, da, bp			DP	dd, bc, de, 0r, bd, 0s, 0u, 0v, 0w, dk, 0x, dl, dn, do, bo, bp, dt, du, bt, bu, bv, dx, bw, dy, bx, by, bz, cb, cc, ac, ad, cg, aj, cp, ap, aq, 0a, 0b, cz, 0j, 0k, 0l, 0m, da, 0n, 0o	EQ	de, bd, 0s, ad, ch, ak, bp, cq, aq, du, bu, dy, 0k, da
AB	ct, 0t, 0f, 0z								
AC	0f	BM	cd	CO	dj			ER	df, cv, dg, ae, af, ag, ci, dn
AD	aw	BN	ct, dl, cm	CP	ar, dj, ck, 0i, az, cl, dq			ES	aq, de, du, bd, 0s, ad, bu, dy, 0k, da, bp
AE	ax	BO	ch	CQ	dl				
AF	bb, 0q, ab, cf, di, ai, bn, co, ao, ds, bs, dw, cy, 0h, dc	BP	as, at	CR	aq, de, du, bd, 0s, bu, ch, dy, 0k, da, bp, cq			ET	bd, 0s, ad, ch, dl, ak, bp, cq, aq, du, bu, dy, 0k, da
		BQ	ci						
AG	dd, bc, 0r, dg, 0u, 0v, 0w, dk, 0x, dn, bo, dt, bt, dx, ac, ae, cg, af, ag, aj, cp, ap, 0a, 0b, cz, 0j, 0l, 0m, 0n, 0o	BR	dd, de, df, dg, dh, di, dj, dk, dl, dn, dp, dq, dr, ds, dt, du, dv, dw, dx, dy, aa, ab, ac, ad, ae, af, ag, ah, ai, aj, ak, an, ao, ap, aq, 0a, 0b, au, av, aw, 0g, 0h, ay, 0i, az, 0j, 0k, 0l, 0m, 0n, 0o, ba, 0p, bb, 0q, bc, 0r, bd, 0s, 0u, 0v, 0w, 0x, bm, bn, bo, bp, br, bs, bt, bu, bv, bw, bx, by, bz, ca, cb, cc, ce, cf, cg, ch, cj, ck, cl, cn, co, cp, cq, cx, cy, cz, da, db, dc	CS	aq, bd, 0s, ak, 0k, cq	DQ	de, bd, 0s, ad, ch, cj, ak, dp, bp, cq, aq, du, bu, au, av, dy, aw, ay, 0k, da	EU	aq, 0s, ad, ch, ak, bp, cq
				CT	ad, ak, 0k			EV	aq, bd, ad, dy, bp, cq
AH	du, bu, dy, dl, 0k					DR	au, av, aw	EW	cc, cd, cu, bv, bw, bx, by, bz, bl, ca, cb
AI	de, bd, da			CU	de, bd, 0s, ad, dl, ak, bp, aq, du, bu, dy, 0k, da	DS	0a, 0b, as, 0u, 0v, 0w, 0x, 0l, 0m, 0n, 0o	EX	ac, dg, ae, af, ck, aj, ak, 0a, 0b, au, av, 0i, 0k
AJ	aq, bd, ch, dy, dl, bp			CV	do	DT	bf, bg, bh, bi, bk, bl		
AK	be			CW	dp	DU	df, ag, dj, cj, ay, az, dn, dp, dq	EY	ct, 0t, 0f
AL	0s			CX	bh, bi	DV	ar, cd, as, ci, ax, 0z	EZ	aq, de, du, ch, bp
AM	0t			CY	bu, 0k			FA	aq, bd, 0k, bp

Figure D.3: Mapping of reengineered hierarchy classes with WGR class labels

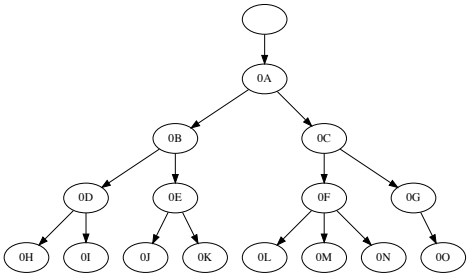
Hierarchy Learning: An Extended Example

In Chapter 5 an approach of hierarchy learning from a set of iOOBN classes was proposed. The running example chosen there is simple and illustrative. However, it was decided to also construct a synthetic hierarchy to better explain the process of extracting a single parent class hierarchy from a multiparent label hierarchy to investigate how reusability and construction costs might be used to identify best potential parent classes when there are multiple options to select.

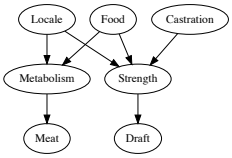
The synthetic hierarchy and class generator, as mentioned in Chapter 5, was created with the maximum number of children = 3, maximum depth = 4, maximum nodes per class = 15, maximum edges per class = 20, maximum density = 10. This results in the hierarchy shown in Figure E.1a and the classes shown in figures E.1b to E.1p.

The classes shown in figures E.1b to E.1p are provided as input to the hierarchy learning algorithm. The algorithm constructs a supergraph and then extracts a set of labels associated with the nodes and edges of the supergraph. The labels are used by the algorithm to construct a label (multiparent) hierarchy as shown in Figure E.2a and E.2b. Here, most of the terminal nodes (the leave nodes representing the original classes) have more than one parent. This is a tricky situation, where we need to choose a parent node that maximises reusability and also minimises construction cost of the hierarchy. Hence, derivation cost and reusability are used to choose the best of the possible available parents. The table in Figure E.2c shows a mapping between the original (synthetic) classes and the learned classes, constructed using the proposed algorithm. The mapping also shows which parent nodes were chosen to extend classes that represent the original synthetic classes. As an instance, "0K" is derivable from three different nodes in the learned hierarchy. The learning algorithm takes the 18^{th} node with parent "0K, 0O" as having maximum reusability and minimum derivation cost among the three available options.

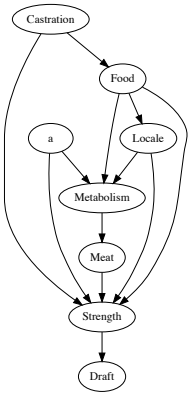
Figures E.3a to E.3q show all the classes generated by the learning algorithm. These classes include classes equivalent to the original classes and additional inferred classes.



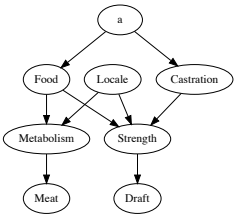
(a) A synthetically generated hierarchy where Maximum number of child = 3, Maximum depth = 4, Maximum nodes per class = 15, Maximum edges per class = 20, Maximum density = 10.



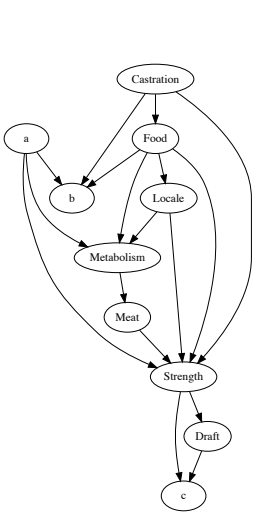
(b) Class 0A



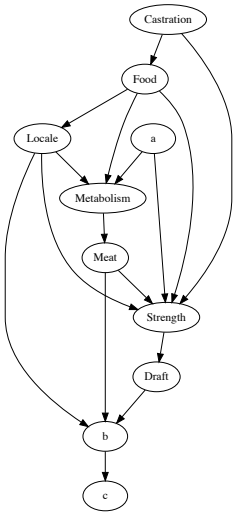
(c) Class 0B



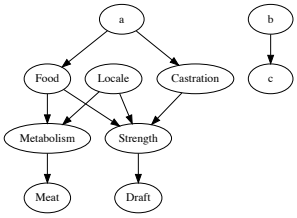
(d) Class 0C



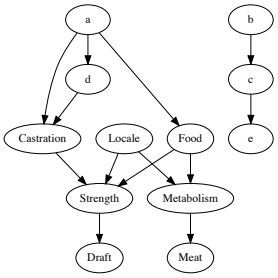
(e) Class 0D



(f) Class 0E

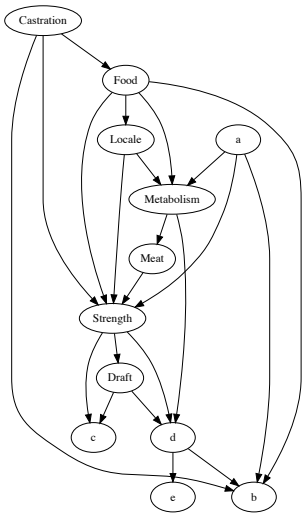


(g) Class 0F

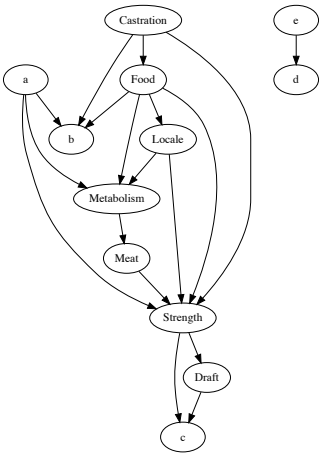


(h) Class 0G

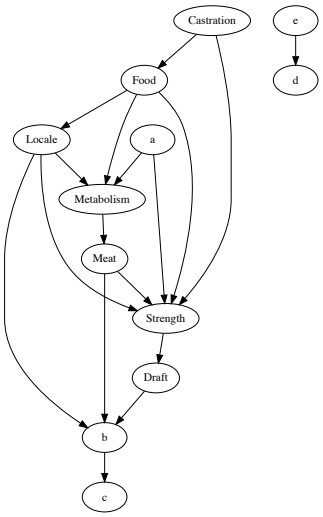
Figure E.1: Classes in the Synthetic hierarchy (contd...)



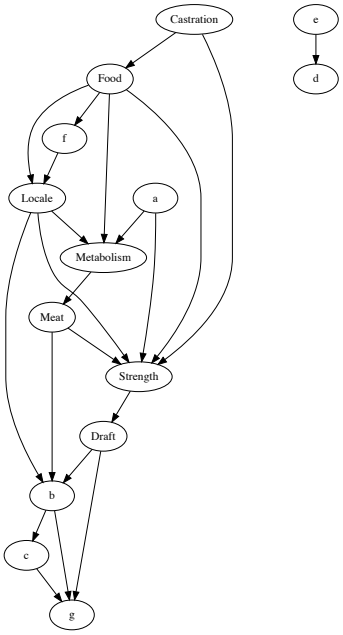
(i) Class 0H



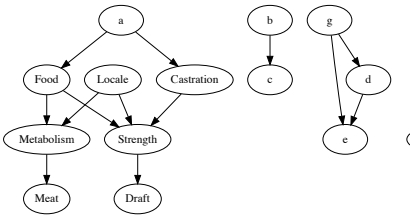
(j) Class 0I



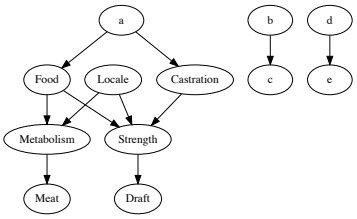
(k) Class 0J



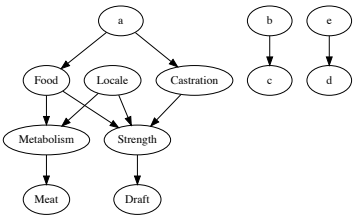
(l) Class 0K



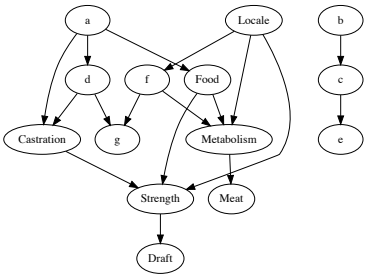
(m) Class 0L



(n) Class 0M

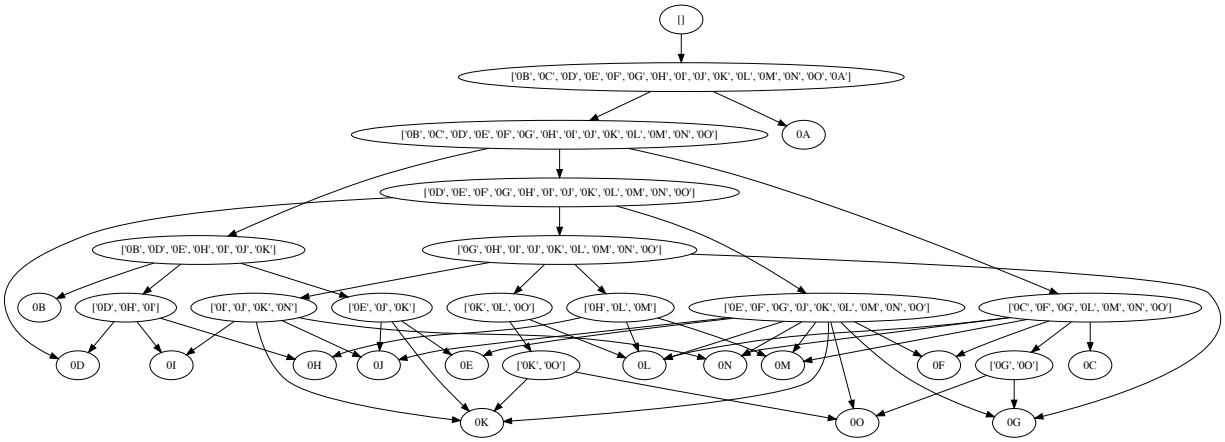


(o) Class 0N

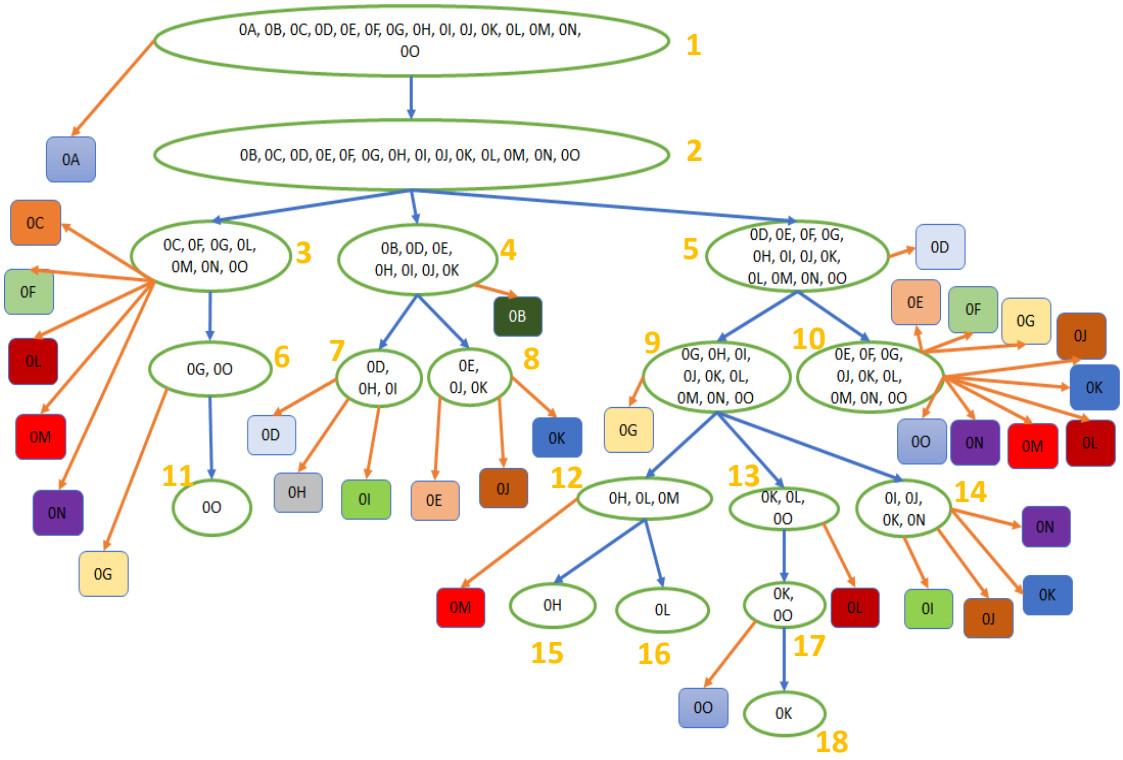


(p) Class 0O

Figure E.1: Classes in the Synthetic hierarchy



(a) The label (multi-parent) hierarchy constructed using the learning algorithm.

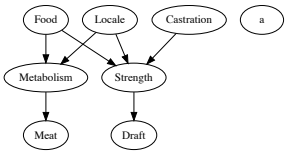


(b) The class (single parent) hierarchy

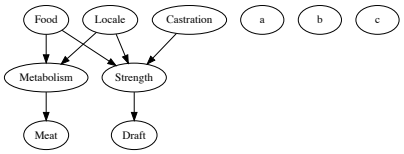
Graph Label	Occurrences (with max matching %)
0A	0A, 0B, 0C, 0D, 0E, 0F, 0G, 0H, 0I, 0J, 0K, 0L, 0M, 0N, 0O (1)
0B	0B, 0D, 0E, 0H, 0I, 0J, 0K (4)
0C	0C, 0F, 0G, 0L, 0M, 0N, 0O (3)
0D	0D, 0H, 0I (7)
0E	0E, 0J, 0K (8)
0F	0E, 0F, 0G, 0J, 0K, 0L, 0M, 0N, 0O (10)
0G	0G, 0O (6)
0H	0D, 0H, 0I (7)
0I	0D, 0H, 0I (7)
0J	0E, 0J, 0K (8)
0K	0K (18)
0L	0K, 0L, 0O (13)
0M	0H, 0L, 0M (12)
0N	0I, 0J, 0K, 0N (14)
0O	0K, 0O (17)

(c) Mapping between learned and original hierarchy classes constructed by taking maximum reusability and minimum derivation-construction cost

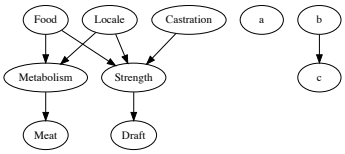
Figure E.2: The learned hierarchy and mapping with the original hierarchy classes



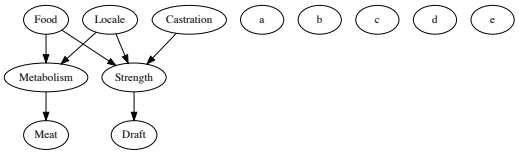
(a) Class 0B_0C_0D_0E_0F-_0G_0H_0I_0J_0K_0L_0M-_0N_0O



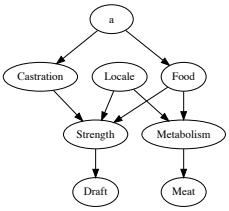
(b) Class 0D_0E_0F_0G_0H_0I_0J-_0K_0L_0M_0N_0O



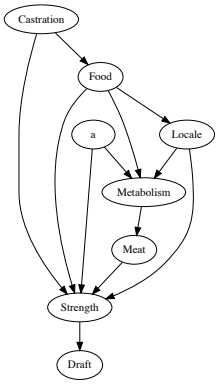
(c) Class 0E_0F_0G_0J_0K_0L-_0M_0N_0O



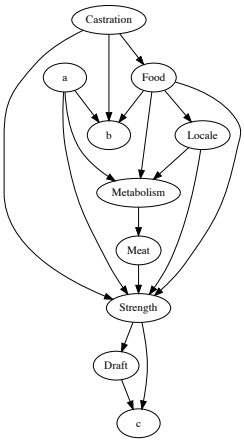
(d) Class 0G_0H_0I_0J_0K_0L_0M_0N_0O



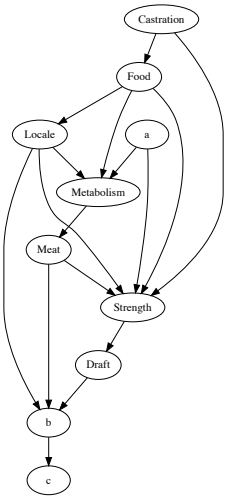
(e) Class 0C_0F_0G_0L-_0M_0N_0O



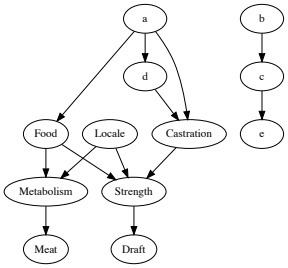
(f) Class 0B_0D_0E_0H-_0I_0J_0K



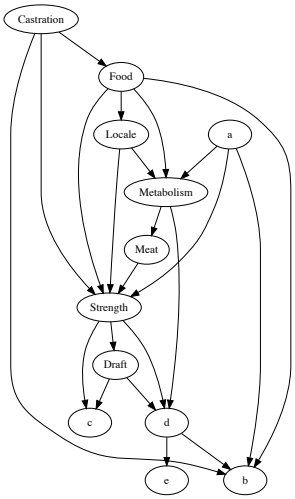
(g) Class 0D_0H_0I



(h) Class 0E_0J_0K

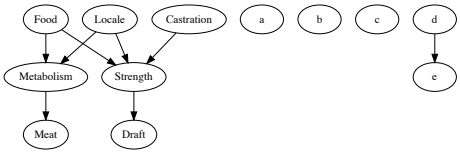


(i) Class 0G_0O

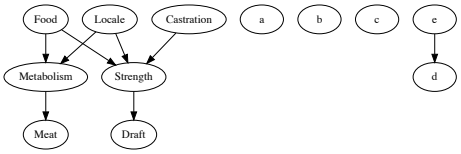


(j) Class 0H

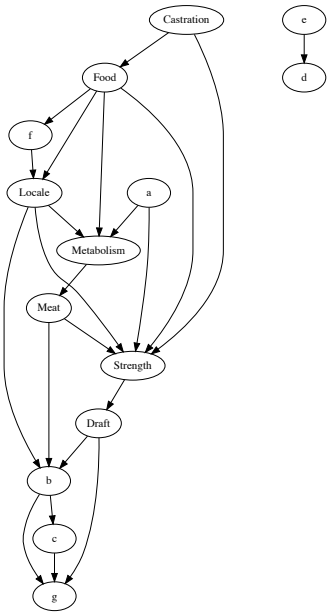
Figure E.3: Classes in the learned hierarchy (contd...)



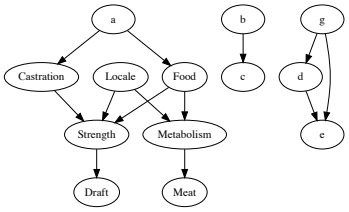
(k) Class 0H_0L_0M



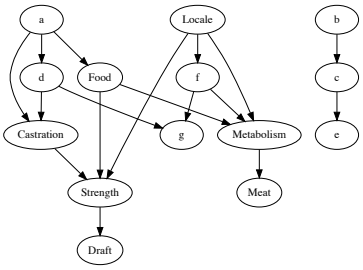
(l) Class 0I_0J_0K_0N



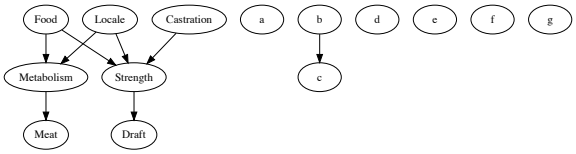
(m) Class 0K



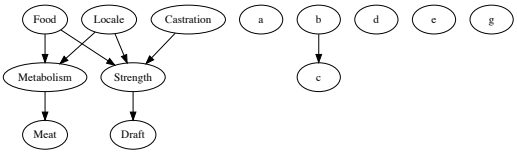
(n) Class 0L



(o) Class 0O



(p) Class 0K_0O



(q) Class 0K_0L_0O

Figure E.3: Classes in the learned hierarchy

Bibliography

[1] K. B. Korb and A. E. Nicholson, *Bayesian Artificial Intelligence*. CRC Press, 2010. xvii, 2, 4, 13, 14, 21, 22, 24, 25, 26, 35, 36, 41, 51, 52, 60, 74, 116, 133

[2] M. Flores, J. Gámez, and K. Olesen, "Incremental compilation of Bayesian networks in practice," *Proceedings of the Fourth International Conference On Intelligent Systems Design and Applications (ISDA 2004)*, pp. 843–848, 2004. [Online]. Available: http://conf.uni-obuda.hu/isda2004/144{_}ISDA2004.pdf xvii, 3, 42

[3] Owen Woodberry, Jessica Millett-Riley, Ann Nicholson and Steve Sinclair, "A Bayesian network model to assist grassland vegetation management," Arthur Rylah Institute for Environmental Research, Department of Environment, Land, Water and Planning, Tech. Rep. 1, 6 2016. xix, 59, 104, 105, 107, 108, 109, 173, 200, 206, 209, 228, 241

[4] M. J. Flores, J. A. Gámez, and K. G. Olesen, "Incremental compilation of Bayesian networks," in *Proc. of 19th Int'l Conf of Uncertainty in Artificial Intelligence UAI '03*, 2003, pp. 233–240. xix, 3, 9, 46, 50, 116, 118, 119

[5] O. Bangsø, M. Flores, and F. Jensen, "Plug and Play Object Oriented Bayesian Networks," *Current Topics in Artificial Intelligence*, vol. 3040, no. Ic, pp. 457–467, 2004. xix, 9, 36, 47, 49, 64, 108, 116, 119, 120

[6] C. Merten, "Incremental Compilation of Object-Oriented Bayesian Networks," pp. 1–8, 2005. xix, 9, 33, 119, 120

[7] P. B. Kruchten, "The 4+ 1 view model of architecture." *IEEE software*, vol. 12, no. 6, pp. 42–50, 1995. xxiii, 206, 207

[8] E. Charniak, "Bayesian networks without tears." *AI magazine*, vol. 12, no. 4, pp. 50–50, 1991. 1, 8

[9] D. Nikovski, "Constructing Bayesian networks for medical diagnosis from incomplete and partially correct statistics," *IEEE Transactions on Knowledge and Data Engineering*, vol. 12, no. 4, pp. 509–516, 2000. 1

[10] J. Pearl, *Probabilistic reasoning in intelligent systems: networks of plausible inference*. Elsevier, 2014. 1, 18, 52

[11] ——, "Causality: models, reasoning and inference," *Econometric Theory*, vol. 19, no. 675-685, p. 46, 2003. 1, 18

[12] J. Pearle, "Probabilistic reasoning in intelligent systems," 1988. 2, 17

[13] F. V. Jensen and T. D. Nielsen, *Bayesian Networks and Decision Graphs*, 2nd ed. New York: Springer Verlag, 2007. 2, 19, 61, 115, 116

[14] T. Charitos, L. van der Gaag, S. Visscher, K. Schurink, and P. Lucas, "A dynamic Bayesian network for diagnosing ventilator-associated pneumonia in ICU patients," *Expert Systems with Applications*, vol. 36, no. 2, pp. 1249–258, 2009. 2

[15] C. Conati, A. Gertner, K. VanLehn, and M. Druzdzel, "On-Line Student Modeling for Coached Problem Solving Using Bayesian Networks," in *UM97 – Proc. of the 6th Int'l Conf. on User Modeling*, 1997, pp. 231–242. 2

[16] K. Kristensen and I. Rasmussen, "The use of a Bayesian network in the design of a decision support system for growing malting barley without use of pesticides," *Computers and Electronics in Agriculture*, vol. 33, no. 3, pp. 197–217, 2002. 2

[17] P. A. Aguilera, A. Fernández, R. Fernández, R. Rumí, and A. Salmerón, "Bayesian Networks in Environmental Modelling," *Environmental Modelling & Software*, vol. 26, no. 12, pp. 1376–1388, 2011. 2

[18] R. Baker, A. Battisti, J. Bremmer, M. Kenis, J. Mumford, F. Petter, G. Schrader, S. Bacher, P. DeBarro, P. Hulme, O. Karadjova, A. Lansink, O. Pruvost, P. Pysek, A. Roques, Y. Baranchikov, and J.-H. Sun, " PRATIQUE: a research project to enhance pest risk analysis techniques in the European Union," *EPPO Bulletin*, vol. 39, no. 1, pp. 87–93, 2009. 2, 164

[19] S. Mascaro, K. Korb, and A. Nicholson, "Anomaly Detection in Vessel Tracks using Bayesian Networks." *Int'l Journal of Approximate Reasoning. Elsevier Science*, vol. 55, no. 1, pp. 84–96, 2011. 2

[20] L. Falzon, "Using Bayesian network analysis to support centre of gravity analysis in military planning," *European Journal of Operational Research*, vol. 170, no. 2, pp. 629–643, 2006. 2

[21] T. Boneh, G. Weymouth, P. Newham, R. Potts, J. Bally, A. Nicholson, and K. Korb, "Fog forecasting for Melbourne Airport using a Bayesian network." *Weather and Forecasting*, vol. 30, no. 5, pp. 1218–1233, 2015. 2

[22] A. Tang, A. Nicholson, Y. Jin, and J. Han, "Using Bayesian belief networks for change impact analysis in architecture design," *Journal of Systems and Software*, vol. 80, no. 1, pp. 127–148, 2007. 2

[23] S. L. Lauritzen and D. J. Spiegelhalter, "Local computations with probabilities on graphical structures and their application to expert systems," *Journal of the Royal Statistical Society. Series B (Methodological)*, pp. 157–224, 1988. 2, 20, 117

[24] G. F. Cooper, "The computational complexity of probabilistic inference using Bayesian belief networks," *Artificial intelligence*, vol. 42, no. 2-3, pp. 393–405, 1990. 3, 41, 118

[25] A. L. Madsen and F. V. Jensen, "Lazy Propagation in Junction Trees," in *Proc. of 14th Int'l Conf of Uncertainty in Artificial Intelligence UAI '98*, 1998, pp. 362–369. 3, 42, 50

[26] M. Mezzini and M. Moscarini, "Simple algorithms for minimal triangulation of a graph and backward selection of a decomposable Markov network," *Theory of Comput. Sci.*, vol. 411, no. 7-9, pp. 958–966, 2010. 3, 43, 46, 118, 132

[27] F. Jensen, "Hugin API Reference guide," Hugin Experts, Tech. Rep. 8, 2 2016. 3, 17, 65, 73, 115, 117, 119, 214, 217, 220

[28] BayesFusion.com. (2018) GeNIe Modeler: Complete Modeling Freedom. [Online; accessed 19-February-2019]. [Online]. Available: https://www.bayesfusion.com/genie/ 3, 28, 39, 75

[29] N. S. Corp., "Netica Application for Belief Networks and Influence Diagrams," Norsys Software Corp, Vancouver, BC, Canada, Tech. Rep. 5, 3 1997. 3, 4, 39, 214

[30] P. Spirtes, C. Glymour, and R. Scheines, *Causation, Prediction, and Search, Second Edition*, ser. Adaptive computation and machine learning. MIT Press, 2000. 4, 54

[31] D. Heckerman, D. Geiger, and D. M. Chickering, "Learning Bayesian Networks: The combination of knowledge and statistical data," *Machine learning*, vol. 20, no. 3, pp. 197–243, 1995. 4, 55

[32] K. B. Laskey and S. M. Mahoney, "Network Fragments: Representing Knowledge for Constructing Probabilistic Models," in *Proc. of the 13th Conf. on Uncertainty in Artificial Intelligence (UAI), USA, 1997*, 1997, pp. 334–341. 4, 26, 28, 74

[33] T. Boneh, "Ontology and Bayesian Decision Networks for Supporting the Meteorological Forecasting Process," Ph.D. dissertation, Clayton School of Information Technology, Monash University, 2010. 4, 14, 20, 52, 204

[34] N. Fenton and M. Neil, "Building large-scale Bayesian networks," *The Knowledge Engineering Review*, vol. 15, no. 3, pp. 257–284, 2000. 4, 26, 28

[35] M. Horny, "Bayesian networks," School of Public Health, Department of Health Policy and Management, Boston University, Tech. Rep. 5, 4 2014. 4, 20, 214

[36] D. Koller and A. Pfeffer, "Object-Oriented Bayesian Networks," in *Proc. of the 13th Conf. on Uncertainty in Artificial Intelligence (UAI), USA, 1997*, 1997, pp. 302–313. 4, 7, 9, 33, 34, 35, 36, 71, 85, 87

[37] O. Bangsø, M. J. Flores, and F. V. Jensen, "Plug & Play Object Oriented Bayesian Networks," in *Current Topics in Artificial Intelligence, 10th Conf. of the Spanish Association for Artificial Intelligence, CAEPIA 2003*, 2003, pp. 457–467. 4, 7, 33, 36, 37, 85

[38] C. Meek and D. Heckerman, "Structure and Parameter Learning for Causal Independence and Causal Interaction Models," in *Proc. of the 13th Conf. on Uncertainty in Artificial Intelligence (UAI), USA, 1997*, 1997, pp. 366–375. 4, 26, 54, 55

[39] E. Segal, D. Pe'er, A. Regev, D. Koller, and N. Friedman, "Learning Module Networks," in *UAI '03, Proc. of the 19th Conf. in Uncertainty in Artificial Intelligence, Acapulco, Mexico, August 7-10 2003*, 2003, pp. 525–534. 4, 26

[40] D. Heckerman, C. Meek, and D. Koller, "Probabilistic entity-relationship models, PRMs, and plate models," in *Introduction to statistical relational learning*, 2007, pp. 201–238. 4, 26, 29

[41] K. B. Laskey, "MEBN: A language for first-order Bayesian knowledge bases," *Artif. Intell.*, vol. 172, no. 2-3, pp. 140–178, 2008. 4, 17, 26, 27

[42] D. P. Xiang, Yang and M. P. Beddoes., "Multiply Sectioned Bayesian Networks and Junction Forests for Large Knowledge-Based Systems," *Computational Intelligence*, vol. 9, no. 2, pp. 171–220, 1993. [Online]. Available: http://www.blackwell-synergy.com/doi/abs/10.1111/j.1467-8640.1993.tb00306.x 4, 17, 27, 41

[43] D. Koller and N. Friedman, *Probabilistic Graphical Models - Principles and Techniques*. MIT Press, 2009. 4, 16, 26, 29

[44] B. J. Cox and A. J. Novobilski, *Object-oriented programming - an evolutionary approach (2. ed.)*. Addison-Wesley, 1991. 4

[45] J. Huang, *A brief history of object-oriented programming*. University of Tennessee Department of Electrical Engineering and Computer . . . , 2013. 4

[46] D. Koller and A. Pfeffer, "Object-oriented Bayesian networks," in *Proc. of the* 13^{th} *conf. on Uncertainty in artificial intelligence*. Morgan Kaufmann Publishers Inc., 1997, pp. 302–313. 5, 17, 26, 29, 64, 108

[47] M. Cartwright and M. Shepperd, "An empirical investigation of an object-oriented software system," *IEEE Transactions on software engineering*, vol. 26, no. 8, pp. 786–796, 2000. 6, 35

[48] L. C. Briand and J. Wüst, "Empirical studies of quality models in object-oriented systems," in *Advances in computers*. Elsevier, 2002, vol. 56, pp. 97–166. 6, 35

[49] R. Jabangwe, J. Börstler, D. Šmite, and C. Wohlin, "Empirical evidence on the link between object-oriented measures and external quality attributes: a systematic literature review," *Empirical Software Engineering*, vol. 20, no. 3, pp. 640–693, 2015. 6, 35

[50] T. Xuan Hoang, "Inheritance in Object Oriented Bayesian Network," Master's thesis, Clayton school of IT, Monash University, Australia, 10 2013. 7, 37, 108

[51] O. Bangsø and P.-H. Wuillemin, "Top-down Construction and Repetitive Structures Representation in Bayesian Networks," in *Thirteenth International Florida Proceedings of the Artificial Intelligence Research Society Conference*, 2000, pp. 282–286. 7

[52] ——, "Object Oriented Bayesian Networks A Framework for Topdown Specification of Large Bayesian Networks and Repetitive Structures," Department of Computer Science, AALBORG University, Tech. Rep. 1, 4 2000. 9

[53] A. Cano, "Elvira: Home Page for users of the Elvira System," http://leo.ugr.es/elvira/, [Online; accessed 07-May-2019]. 9, 39

[54] D. Chickering, D. Geiger, and D. Heckerman, "Learning Bayesian networks: Search methods and experimental results," in *proceedings of fifth conference on artificial intelligence and statistics*, 1995, pp. 112–128. 10

[55] D. M. Chickering, "A transformational characterization of equivalent Bayesian Network structures," in *Proceedings of the Eleventh conference on Uncertainty in artificial intelligence*. Morgan Kaufmann Publishers Inc., 1995, pp. 87–98. 10

[56] ——, "Learning Bayesian Networks is NP-Complete," in *Learning from Data - Fifth International Workshop on Artificial Intelligence and Statistics, AISTATS 1995, Key West, Florida, USA, January, 1995. Proceedings.*, 1995, pp. 121–130. 10, 54

[57] O. Woodberry, J. Millett-Riley, A. Nicholson, and S. Sinclair, "An Object-Oriented Dynamic Bayesian Decision Network Model for Grasslands Adaptive Management (Abstract Only)," in *the 11th UAI Bayesian Modeling Applications Workshop (BMAW 2014), Eds. Kathryn B. Laskey, James Jones and Russell Almond. CEUR Workshop Proc., 1218*, 2014. 10, 59, 105, 164, 173, 226, 227

[58] F. N. David, *Games, gods and gambling: The origins and history of probability and statistical ideas from the earliest times to the Newtonian era.* Hafner Publishing Company, 1962. 13

[59] M. H. Degroot and M. J. Schervish, *Probability and statistics*. Pearson Education, 2012. 13

[60] S. M. Stigler, *The history of statistics: The measurement of uncertainty before 1900*. Harvard University Press, 1986. 13

[61] F. N. David, *Games, gods, and gambling: A history of probability and statistical ideas*. Courier Corporation, 1988. 13

[62] T. Bayes, "An essay towards solving a problem in the doctrine of chances." *Philosophical Transactions of the Royal Society of London*, vol. 45, no. 3, pp. 243–315, 1958. 13

[63] F. P. Ramsey, "Truth and probability," in *Readings in Formal Epistemology*. Springer, 2016, pp. 21–45. 14

[64] V. J. Easton and J. H. McColl. (1997) Statistics Glossary. [Online; accessed 15-July-2019]. [Online]. Available: http://www.stats.gla.ac.uk/steps/glossary/ 16

[65] W. L. Buntine, "Operations for Learning with Graphical Models," *CoRR*, vol. abs/1105.2519, 2011. [Online]. Available: http://arxiv.org/abs/1105.2519 16

[66] R. G. Cowell, A. P. Dawid, and D. J. Spiegelhalter, "Sequential Model Criticism in Probabilistic Expert Systems," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 15, no. 3, pp. 209–219, 1993. 16

[67] R. D. Shachter and C. R. Kenley, "Gaussian influence diagrams," *Management Science*, vol. 35, no. 5, pp. 527–550, 1989. 17

[68] R. D. Shachter, "Evaluating Influence Diagrams," *Operations Research*, vol. 34, no. 6, pp. 871–882, 1986. 17

[69] D. J. Spiegelhalter, A. P. Dawid, S. L. Lauritzen, and R. G. Cowell, "Bayesian analysis in expert systems," *Statistical science*, vol. 8, no. 3, pp. 219–247, 1993. 17

[70] D. Koller, "Probabilistic Relational Models," in *Int'l Conf. on Inductive Logic Programming*. Springer, 1999, pp. 3–13. 17, 18, 26, 27, 29

[71] L. Breiman, *Classification and regression trees*. Routledge, 2017. 17

[72] R. L. Rivest, "Learning Decision Lists," *Machine Learning*, vol. 2, no. 3, pp. 229–246, 1987. 17

[73] W. Buntine, "Representing learning with graphical models," Artificial Intelligence Research Branch, NASA Ames Research Centre, Technical Report FIA-94-14, 1994. 17

[74] P. McCullagh, *Generalized linear models*. Routledge, 2019. 17

[75] M. Frydenberg, "The chain graph Markov property," *Scandinavian Journal of Statistics*, pp. 333–353, 1990. 17

[76] L. Torti, P.-H. Wuillemin, and C. Gonzales, "Reinforcing the Object-Oriented aspect of probabilistic relational models," in *European Workshop on Probabilistic Graphical Models*, 2010, pp. 273–280. 17, 26, 29, 85, 100

[77] M. Jaeger, "Relational Bayesian Networks," *CoRR*, vol. abs/1302.1550, 2013. 17

[78] M. B. Ishak, "Probabilistic relational models: learning and evaluation. (Les modèles probabilistes relationnels : apprentissage et évaluation)," Ph.D. dissertation, University of Nantes, France, 2015. [Online]. Available: https://tel.archives-ouvertes.fr/tel-01179501 18, 27, 54

[79] M. Cossalter, O. Mengshoel, and T. Selker, "Visualizing and understanding large-scale Bayesian networks," in *Workshops at the Twenty-Fifth AAAI Conference on Artificial Intelligence*, 2011. 20

[80] S. Psillos, "An Explorer upon Untrodden Ground: Peirce on Abduction." 2011. 21

[81] P. Dagum, A. Galper, and E. Horvitz, "Dynamic network models for forecasting," in *Uncertainty in artificial intelligence*. Elsevier, 1992, pp. 41–48. 26

[82] A. E. Nicholson, "Monitoring discrete environments using dynamic belief networks (robotics)," Ph.D. dissertation, PhD thesis, University of Oxford (United Kingdom), 1992. 26

[83] U. Kjærulff, "A computational scheme for dynamic Bayesian networks," 1993. 26

[84] T. L. Dean and K. Kanazawa, "Probabilistic Temporal Reasoning." in *AAAI*, 1988, pp. 524–529. 26

[85] D. Koller, "Probabilistic Relational Models," in *Inductive Logic Programming, 9th International Workshop, ILP-99, Bled, Slovenia, June 24-27, 1999, Proceedings*, 1999, pp. 3–13. 27

[86] N. F. D. K. A. P. Getoor, Lise and B. Taskar, "Probabilistic relational models. Introduction to statistical relational learning," in *Introduction to statistical relational learning*, vol. 8, 2007. 27

[87] J. Neville and D. D. Jensen, "Dependency Networks for Relational Data," in *Proceedings of the 4th IEEE International Conference on Data Mining (ICDM 2004), 1-4 November 2004, Brighton, UK*, 2004, pp. 170–177. 27

[88] M. Jaeger, "Relational Bayesian Networks," in *UAI '97: Proceedings of the Thirteenth Conference on Uncertainty in Artificial Intelligence, Brown University, Providence, Rhode Island, USA, August 1-3, 1997*, 1997, pp. 266–273. 27

[89] M. Grzegorczyk, "An introduction to Gaussian Bayesian networks," in *Systems Biology in Drug Discovery and Development*. Springer, 2010, pp. 121–147. 27

[90] C. Gonzales, L. Torti, M. Chopin, and P.-H. Wuillemin, "aGrUM: A GRaphical Universal Modeler," https://forge.lip6.fr/projects/agrum, [Online; accessed 27-June-2017]. 30, 40

[91] N. Mitchell and G. Sevitsky, "LeakBot: An Automated and Lightweight Tool for Diagnosing Memory Leaks in Large Java Applications," in *ECOOP 2003 - Object-Oriented Programming, 17th European Conference, Darmstadt, Germany, July 21-25, 2003, Proceedings*, 2003, pp. 351–377. 32

[92] E. Kindler and I. Krivý, "Object-oriented simulation of systems with sophisticated control," *Int. J. General Systems*, vol. 40, no. 3, pp. 313–343, 2011. 32

[93] Y. E. Chee, L. Wilkinson, A. E. Nicholson, P. F. Quintana-Ascencio, J. E. Fauth, D. Hall, K. J. Ponzio, and L. Rumpff, "Modelling spatial and temporal changes with GIS and Spatial and Dynamic Bayesian Networks," *Environmental Modelling & Software*, vol. 82, pp. 108–120, 2016. 34, 164

[94] B. Bruegee and A. Dutoit, *Object-Oriented software engineering: Practical software development using UML, Patterns and Java*, 3rd ed. Pearson, 2009. 34

[95] N. Shiratori and N. Okude, "Bayesian Networks layer model to represent anesthetic practice," in *Proc. of the IEEE Int'l Conf. on Systems, Man and Cybernetics, Montréal, Canada, 7-10 October 2007*, 2007, pp. 674–679. 35

[96] M. R. Krebs, E. H. Bromley, and A. M. Donald, "The binding of thioflavin-T to amyloid fibrils: localisation and implications," *Artif. Intell.*, vol. 149, no. 1, pp. 30–37, 2005. 35

[97] F. V. Jensen and T. D. Nielsen, "Probabilistic decision graphs for optimization under uncertainty," *Annals OR*, vol. 204, no. 1, pp. 223–248, 2013. 35, 36

[98] Renninger, Harald and Hermann von Hasseln, "Object-Oriented Dynamic Bayesian Network-Templates for Modelling Mechatronic Systems," in *DAIMLER CHRYSLER AG STUTTGART (GERMANY)*, 2002. 35

[99] W. Wiegerinck, W. Burgers, and B. Kappen, "Bayesian Networks, Introduction and Practical Applications," in *Handbook on Neural Information Processing*, 2013, pp. 401–431. 35

[100] R. S. Kenett, "Applications of Bayesian networks," *Available at SSRN 2172713*, 2012. 35

[101] D. TEAM. (2019) Top 10 Real-world Bayesian Network Applications – Know the importance! [Online; accessed 14-October-2019]. [Online]. Available: https://data-flair.training/blogs/bayesian-network-applications/ 35

[102] Pourret, Olivier and Naïm, Patrick and Marcot, Bruce, *Bayesian networks: a practical guide to applications*. John Wiley & Sons, 2008. 35

[103] A. L. Madsen, M. Lang, U. B. Kjærulff, and F. Jensen, "The Hugin Tool for Learning Bayesian Networks," *Learning*, pp. 594–605, 2003. 37, 39, 50, 55, 214

[104] N. S. Corporation. (1995) Netica Application. [Online; accessed 16-August-2017]. [Online]. Available: http://www.norsys.com/netica.html 39

[105] S. Conrady and L. JOUFFE, *Bayesian Networks and Bayesia Lab*. Bayesia Lab., 2007, vol. 1, no. 1. 39, 214

[106] ProbaYes.com. (2019) ProBT. [Online; accessed 29-August-2019]. [Online]. Available: http://www.probayes.com/fr/recherche/probt/ 39

[107] S. Matsumoto, R. N. Carvalho, M. Ladeira, P. C. G. da Costa, L. L. Santos, D. Silva, M. Onishi, E. Machado, and K. Cai, "UnBBayes: a java framework for probabilistic models in AI," *Java in academia and research*, p. 34, 2011. 39, 40, 214

[108] K. Murphy, "The bayes net toolbox for matlab." *Computing science and statistics*, vol. 33, no. 2, pp. 1024–1034, 2001. 39

[109] M. Scutari. (2018) bnlearn - an R package for Bayesian network learning and inference. [Online; accessed 02-November-2019]. [Online]. Available: http://www.bnlearn.com/research/ 39

[110] R. C. for Intelligent Decision-Support Systems, "OpenMarkov: A open source tool for Probabilistic Graphical Models," http://www.openmarkov.org/docs/tutorial/openmarkov-tutorial.pdf, National University for Distance Education (UNED), [Online; accessed 27-June-2017]. 39

[111] K. S. University. (2013) Bayesian Network tools in Java (BNJ). [Online; accessed 02-November-2017]. [Online]. Available: https://sourceforge.net/projects/bnj/ 39

[112] M. U. (Quebec). (2019) ProbRem. [Online; accessed 21-July-2019]. [Online]. Available: https://www.cs.mcgill.ca/~fkaeli/probrem/ 40

[113] U. of Washington. (2018) Alchemy: Open Source AI. [Online; accessed 02-August-2018]. [Online]. Available: http://alchemy.cs.washington.edu 40

[114] K. C. J. S. M. T. Manfred Jaeger, Mark Chavira and A. G. Collado, "Primula," 2009, [Online; accessed 19-September-2017]. [Online]. Available: http://people.cs.aau.dk/~jaeger/Primula/primula-manual2.2.pdf 40

[115] ——. (2002) Primula. [Online; accessed 12-February-2017]. [Online]. Available: http://people.cs.aau.dk/~jaeger/Primula/ 40

[116] M. Inc. (2019) BLOG programming Language. [Online; accessed 17-June-2019]. [Online]. Available: https://bayesianlogic.github.io 40

[117] U. of Massachusetts. (2018) Proximity. [Online; accessed 27-August-2019]. [Online]. Available: http://kdl.cs.umass.edu/software.html 40

[118] agrum.org. (2018) aGrUM a GRaphical Universal Model. [Online; accessed 21-March-2019]. [Online]. Available: https://docs.agrum.org/aGrUM/latest/index.html 40

[119] A. Pfeffer, "IBAL: A Probabilistic Rational Programming Language," in *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence, IJCAI 2001, Seattle, Washington, USA, August 4-10, 2001*, 2001, pp. 733–740. 40

[120] P. O'Rourke and J. Josephson, "Automated abduction: inference to the best explanation," 1997. 41

[121] S. K. Andersen and F. Jensen, "Approximations in Bayesian belief universes for knowledge based systems," in *Proceedings of the Sixth Conference on Uncertainty in Artificial Intelligence, 1990, Cambridge, Ma, Usa*, 1990, pp. 162–169. 41

[122] M. Henrion, "Propagating uncertainty in Bayesian networks by probabilistic logic sampling," in *Machine Intelligence and Pattern Recognition*. Elsevier, 1988, vol. 5, pp. 149–163. 41

[123] A. L. Madsen and F. V. Jensen, "LAZY Propagation: A Junction Tree Inference Algorithm Based on Lazy Evaluation," *Artif. Intell.*, vol. 113, no. 1-2, pp. 203–245, 1999. 42

[124] A. Cano and S. Moral, "Heuristic Algorithms for the Triangulation of Graphs," in *Advances in Intelligent Computing - IPMU'94, 5th International Conference on Processing and Management of Uncertainty in Knowledge-Based Systems, Paris, France, July 4-8, 1994, Selected Papers*, 1994, pp. 98–107. 43

[125] P. Heggernes, J. A. Telle, and Y. Villanger, "Computing Minimal Triangulations in Time $O(n^{\text{alpha}} \log n) = o(n^{2.376})$," *SIAM J. Discrete Math.*, vol. 19, no. 4, pp. 900–913, 2005. 43

[126] M. Shindo and E. Tomita, "A Simple Algorithm for Finding a Maximum Clique and Its Worst-Case Time Complexity," *Systems and Computers in Japan*, vol. 21, no. 3, pp. 1–13, 1990. 43

[127] P. Larrañaga, C. M. H. Kuijpers, M. Poza, and R. H. Murga, "Decomposing Bayesian networks: triangulation of the moral graph with genetic algorithms," *Statistics and Computing*, vol. 7, no. 1, pp. 19–34, 1997. 43

[128] P. Heggernes, "Minimal triangulations of graphs: A survey," *Discrete Mathematics*, vol. 306, no. 3, pp. 297–317, 2006. 43

[129] A. Berry, J. R. S. Blair, and P. Heggernes, "Maximum Cardinality Search for Computing Minimal Triangulations," in *Graph-Theoretic Concepts in Computer Science, 28th International Workshop, WG 2002, Cesky Krumlov, Czech Republic, June 13-15, 2002, Revised Papers*, 2002, pp. 1–12. 43

[130] A. Berry and R. Pogorelcnik, "A simple algorithm to generate the minimal separators and the maximal cliques of a chordal graph," *Inf. Process. Lett.*, vol. 111, no. 11, pp. 508–511, 2011. 43

[131] F. V. Jensen and F. Jensen, "Optimal Junction Trees," *CoRR*, vol. abs/1302.6823, 2013. 43

[132] D. J. Rose, R. E. Tarjan, and G. S. Lueker, "Algorithmic Aspects of Vertex Elimination on Graphs," *SIAM J. Comput.*, vol. 5, no. 2, pp. 266–283, 1976. 43

[133] E. Dahlhaus, "Minimal Elimination Ordering Inside a Given Chordal Graph," in *Graph-Theoretic Concepts in Computer Science, 23rd International Workshop, WG '97, Berlin, Germany, June 18-20, 1997, Proceedings*, 1997, pp. 132–143. 43

[134] M. J. Flores and J. A. Gámez, "Triangulation of Bayesian networks by retriangulation," *Int. J. Intell. Syst.*, vol. 18, no. 2, pp. 153–164, 2003. 43

[135] R. E. Neapolitan *et al.*, *Learning Bayesian Networks*. Pearson Prentice Hall, 2004, vol. 38. 43, 46

[136] K. G. Olesen and A. L. Madsen, "Maximal prime subgraph decomposition of Bayesian networks," *IEEE Trans. Systems, Man, and Cybernetics, Part B*, vol. 32, no. 1, pp. 21–31, 2002. 46, 118

[137] ——, "Maximal prime subgraph decomposition of Bayesian networks," *IEEE Transactions on Systems, Man, and Cybernetics, Part B: Cybernetics*, vol. 32, no. 1, pp. 21–31, 2002. 46

[138] D. Wu, "Maximal prime subgraph decomposition of Bayesian networks: A relational database perspective," *International Journal of Approximate Reasoning*, vol. 46, no. 2, pp. 334–345, 2007. 46

[139] D. Wu and S. K. M. Wong, "Maximal Prime Subgraph Decomposition of Bayesian Networks: A Relational Database Perspective," in *Proceedings of the Eighteenth International Florida Artificial Intelligence Research Society Conference, Clearwater Beach, Florida, USA*, 2005, pp. 793–798. 46

[140] M. Samiullah, T. X. Hoang, D. Albrecht, A. Nicholson, and K. Korb, "iOOBN: a Bayesian Network Modelling Tool using Object Oriented Bayesian Networks with Inheritance," in *Proc. of 29th IEEE ICTAI, BOSTON, MA, USA, November 6-8*, 2017, pp. 1218–1225. 50, 60, 116, 119

[141] Oboler, Andre, "The kebn process: A new approach to knowledge engineering with Bayesian nets," Technical report, Monash University, Tech. Rep., 2002. 51

[142] L. M. Davidson, *Knowledge extraction technology for terminology.* University of Ottawa (Canada), 1998. 51

[143] K. B. Laskey and S. M. Mahoney, "Network engineering for agile belief network models," *IEEE Transactions on knowledge and data engineering*, vol. 12, no. 4, pp. 487–498, 2000. 52

[144] D. Heckerman, "Probabilistic similarity networks," *Networks*, vol. 20, no. 5, pp. 607–636, 1990. 52

[145] A. Zagorecki and M. J. Druzdzel, "Knowledge engineering for Bayesian Networks: how common are noisy-max distributions in practice?" in *ECAI*, 2006, p. 482. 52

[146] ——, "Knowledge engineering for Bayesian networks: How common are noisy-MAX distributions in practice?" *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, vol. 43, no. 1, pp. 186–195, 2012. 52

[147] M. J. Druzdzel and F. J. Diez, "Criteria for combining knowledge from different sources in probabilistic models," in *Working Notes of the UAI 2000 Workshop on Domain Knowledge with Data for Decision Support*, 2000. 52, 53

[148] S. Monti and G. Carenini, "Dealing with the expert inconsistency in probability elicitation," *IEEE Transactions on Knowledge and Data Engineering*, vol. 12, no. 4, pp. 499–508, 2000. 52

[149] R. L. Keeney and H. Raiffa, "Decision analysis with multiple conflicting objectives," *Wiley& Sons, New York*, 1976. 52

[150] C. S. Wallace, K. B. Korb, and H. Dai, "Causal Discovery via MML," in *Machine Learning, Proceedings of the Thirteenth International Conference (ICML '96), Bari, Italy, July 3-6, 1996*, 1996, pp. 516–524. 52, 54, 166, 172

[151] S. Geman and D. Geman, "Stochastic relaxation, Gibbs distributions, and the Bayesian restoration of images," *IEEE Transactions on pattern analysis and machine intelligence*, no. 6, pp. 721–741, 1984. 53

[152] R. T. O'Donnell, L. Allison, and K. B. Korb, "Learning Hybrid Bayesian Networks by MML," in *AI 2006: Advances in Artificial Intelligence, 19th Australian Joint Conference on Artificial Intelligence, Hobart, Australia, December 4-8, 2006, Proceedings*, 2006, pp. 192–203. 53, 166

[153] U. Nodelman, C. R. Shelton, and D. Koller, "Learning Continuous Time Bayesian Networks," in *UAI '03, Proceedings of the 19th Conference in Uncertainty in Artificial Intelligence, Acapulco, Mexico, August 7-10 2003*, 2003, pp. 451–458. 53

[154] C. S. Wallace and K. B. Korb, "Learning linear causal models by MML sampling," in *Causal models and intelligent data management. Springer, Berlin, Heidelberg*, 1999, pp. 89–111. 53, 164

[155] W. Lam and F. Bacchus, "Learning Bayesian Belief Networks: An Approach Based on the MDL Principle," *Computational Intelligence*, vol. 10, pp. 269–294, 1994. 53, 164

[156] P. Spirtes and C. Meek, "Learning Bayesian Networks with Discrete Variables from Data," in *Proceedings of the First International Conference on Knowledge Discovery and Data Mining (KDD-95), Montreal, Canada, August 20-21, 1995*, 1995, pp. 294–299. 53, 164

[157] J. Abellán, M. Gómez-Olmedo, and S. Moral, "Some Variations on the PC Algorithm," in *Third European Workshop on Probabilistic Graphical Models, 12-15 September 2006, Prague, Czech Republic. Electronic Proceedings.*, 2006, pp. 1–8. 53

[158] C. M. Heckerman, David and G. Cooper, "A Bayesian Approach to Casual Discovery," Microsoft Research, Advanced Technology Division, Microsoft Corporation, One Microsoftway, Redmond, WA 98052, Tech. Rep. 19, February 1999. 53

[159] D. Heckerman and D. Geiger, "Learning Bayesian Networks: A Unification for Discrete and Gaussian Domains," *CoRR*, vol. abs/1302.4957, 2013. 54

[160] R. Cui, P. Groot, and T. Heskes, "Learning causal structure from mixed data with missing values using Gaussian copula models," *Statistics and Computing*, vol. 29, no. 2, pp. 311–333, 2019. 54

[161] S. Lee and V. G. Honavar, "On Learning Causal Models from Relational Data," in *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence, February 12-17, 2016, Phoenix, Arizona, USA.*, 2016, pp. 3263–3270. 54

[162] H. Dai, K. B. Korb, C. S. Wallace, and X. Wu, "A Study of Causal Discovery With Weak Links and Small Samples," in *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence, IJCAI 97, Nagoya, Japan, August 23-29, 1997, 2 Volumes*, 1997, pp. 1304–1309. 54

[163] M. Teyssier and D. Koller, "Ordering-Based Search: A Simple and Effective Algorithm for Learning Bayesian Networks," *CoRR*, vol. abs/1207.1429, 2012. 54

[164] J. R. Neil, C. S. Wallace, and K. B. Korb, "Learning Bayesian Networks with Restricted Causal Interactions," *CoRR*, vol. abs/1301.6727, 2013. 54

[165] D. Margaritis, "Learning Bayesian network model structure from data," Ph.D. dissertation, Carnegie-Mellon University, Pittsburgh Pa School of Computer Science, Pittsburgh, PA 15213, USA, 2003. 54

[166] L. Getoor, N. Friedman, D. Koller, and B. Taskar, "Learning Probabilistic Models of Link Structure," *J. Mach. Learn. Res.*, vol. 3, pp. 679–707, 2002. 54

[167] T. Verma and J. Pearl, "Equivalence and synthesis of causal models," in *UAI '90: Proceedings of the Sixth Annual Conference on Uncertainty in Artificial Intelligence, MIT, Cambridge, MA, USA, July 27-29, 1990*, 1990, pp. 255–270. 54

[168] C. N. G. Spirtes, Peter and R. Scheines, "Causality from probability." *Laboratory for Computational Linguistics*, vol. 112, 1989. 54

[169] G. F. Cooper and E. Herskovits, "A Bayesian Method for the Induction of Probabilistic Networks from Data," *Machine Learning*, vol. 9, pp. 309–347, 1992. 54

[170] D. M. Chickering, "Learning Equivalence Classes of Bayesian-Network Structures," *J. Mach. Learn. Res.*, vol. 2, pp. 445–498, 2002. 54

[171] C. K. Chow and C. N. Liu, "Approximating discrete probability distributions with dependence trees," *IEEE Trans. Information Theory*, vol. 14, no. 3, pp. 462–467, 1968. 54

[172] M. Singh and M. Valtorta, "Construction of Bayesian network structures from data: A brief survey and an efficient algorithm," *Int. J. Approx. Reasoning*, vol. 12, no. 2, pp. 111–131, 1995. 54

[173] G. M. Provan and M. Singh, "Learning Bayesian Networks Using Feature Selection," in *Learning from Data - Fifth International Workshop on Artificial Intelligence and Statistics, AISTATS 1995, Key West, Florida, USA, January, 1995. Proceedings.*, 1995, pp. 291–300. 54

[174] S. Acid and L. M. de Campos, "BENEDICT: An algorithm for learning probabilistic belief networks." in *International Conference on Information Processing and Management of Uncertainty in Knowledge-based Systems*, 1996, pp. 979–984. 54

[175] Z. Ji, Q. Xia, and G. Meng, "A review of parameter learning methods in Bayesian network," in *International Conference on Intelligent Computing*. Springer, 2015, pp. 3–12. 55

[176] N. A. Furlotte, D. Heckerman, and C. Lippert, "Quantifying the uncertainty in heritability," *Journal of human genetics*, vol. 59, no. 5, p. 269, 2014. 55

[177] D. Titterington *et al.*, "Bayesian methods for neural networks and related models," *Statistical Science*, vol. 19, no. 1, pp. 128–139, 2004. 55

[178] S. L. Lauritzen, "The EM algorithm for graphical association models with missing data," *Computational Statistics & Data Analysis*, vol. 19, no. 2, pp. 191–201, 1995. 55

[179] S. Geman and D. Geman, "Stochastic Relaxation, Gibbs Distributions, and the Bayesian Restoration of Images," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 6, no. 6, pp. 721–741, 1984. 55

[180] D.-x. Niu, H.-f. Shi, and D. D. Wu, "Short-term load forecasting using Bayesian neural networks learned by Hybrid Monte Carlo algorithm," *Applied Soft Computing*, vol. 12, no. 6, pp. 1822–1827, 2012. 55

[181] S. J. Russell and P. Norvig, *Artificial intelligence: a modern approach*. Malaysia; Pearson Education Limited,, 2016. 55

[182] H. Langseth and O. Bangsø, "Parameter learning in object-oriented Bayesian networks," *Annals of Mathematics and Artificial Intelligence*, vol. 32, no. 1-4, pp. 221–243, 2001. 55

[183] O. B. H. Langseth and T. Nielsen, "Structural learning in object oriented domains," in *Proceedings of the Fourteenth Florida Artificial Intelligence Research Society Conference*, 2001, pp. 340–344. 55, 164

[184] M. Elseidy, E. Abdelhamid, S. Skiadopoulos, and P. Kalnis, "GRAMI: Frequent Subgraph and Pattern Mining in a Single Large Graph," *PVLDB*, vol. 7, no. 7, pp. 517–528, 2014. 56

[185] M. Kuramochi and G. Karypis, "Finding Frequent Patterns in a Large Sparse Graph*," *Data Min. Knowl. Discov.*, vol. 11, no. 3, pp. 243–271, 2005. 56, 57

[186] X. Yan and J. Han, "gSpan: Graph-Based Substructure Pattern Mining," in *Proceedings of the 2002 IEEE International Conference on Data Mining (ICDM 2002), 9-12 December 2002, Maebashi City, Japan*, 2002, pp. 721–724. 56, 57

[187] M. I. Wood, L. Ivanov, and Z. Lamprou, "An Analysis of Inheritance Hierarchy Evolution," in *Proceedings of the Evaluation and Assessment on Software Engineering, EASE 2019, Copenhagen, Denmark, April 15-17, 2019*, 2019, pp. 24–33. 56, 57, 165, 166

[188] I. Moore, "Automatic Inheritance Hierarchy Restructuring and Method Refactoring," in *Proceedings of the 1996 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA '96), San Jose, California, USA, October 6-10, 1996.*, 1996, pp. 235–250. 57, 165

[189] E. Casais, "An Incremental Class Reorganization Approach," in *ECOOP '92, European Conference on Object-Oriented Programming, Utrecht, The Netherlands, June 29 - July 3, 1992, Proceedings*, 1992, pp. 114–132. 57, 165

[190] H. Dicky, C. Dony, M. Huchard, and T. Libourel, "On Automatic Class Insertion with Overloading," in *Proceedings of the 1996 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA '96), San Jose, California, USA, October 6-10, 1996.*, 1996, pp. 251–267. 57, 165

[191] W. F. Opdyke and R. E. Johnson, "Creating Abstract Superclasses by Refactoring," in *Proceedings of the ACM 21th Conference on Computer Science, CSC '93, Indianapolis, IN, USA, February 16-18, 1993*, 1993, pp. 66–73. 57, 165

[192] I. Moore and T. Clement, "A Simple and Efficient Algorithm for Inferring Inheritance Hierarchies," 1996. 57, 165

[193] G. W. Mineau and R. Godin, "Automatic Structuring of Knowledge Bases by Conceptual Clustering," *IEEE Trans. Knowl. Data Eng.*, vol. 7, no. 5, pp. 824–828, 1995. 57, 165

[194] E. Casais, "Managing class evolution in object-oriented systems," in *Object-Oriented Software Composition*, 1995, pp. 201–244. 57, 165

[195] W. W. Pun and R. L. Winder, "Automating class hierarchy graph construction," Bayesian Intelligence, Technical Report RN 89/23, 1989. 57, 165

[196] H. Bunke, X. Jiang, and A. Kandel, "On the Minimum Common Supergraph of Two Graphs," *Computing*, vol. 65, no. 1, pp. 13–25, 2000. 57

[197] X. Yan, P. S. Yu, and J. Han, "Graph Indexing: A Frequent Structure-based Approach," in *Proceedings of the ACM SIGMOD International Conference on Management of Data, Paris, France, June 13-18, 2004*, 2004, pp. 335–346. 57

[198] Y. Chen, X. Zhao, X. Lin, Y. Wang, and D. Guo, "Efficient Mining of Frequent Patterns on Uncertain Graphs," *IEEE Trans. Knowl. Data Eng.*, vol. 31, no. 2, pp. 287–300, 2019. 57

[199] A. Inokuchi, T. Washio, and H. Motoda, "An Apriori-Based Algorithm for Mining Frequent Substructures from Graph Data," in *Principles of Data Mining and Knowledge Discovery, 4th European Conference, PKDD 2000, Lyon, France, September 13-16, 2000, Proceedings*, 2000, pp. 13–23. 57

[200] U. B. Kjærulff and A. L. Madsen, "Bayesian networks and influence diagrams," *Springer*, vol. 200, p. 114, 2008. 59, 116, 124

[201] H. Langseth and O. Bangsø, "Parameter Learning in Bayesian Networks," *Annals of Mathematics and AI*, pp. 221–243, 2001. 61, 174

[202] M. J. Flores, A. E. Nicholson, and R. F. Ropero, "Dynamic OOBNs applied to water management in dams," in *2016 IEEE International Conference on Knowledge Engineering and Applications (ICKEA)*. IEEE, 2016, pp. 255–260. 67

[203] A. J. Pfeffer, "Probabilistic Reasoning for Complex Systems," Ph.D. dissertation, Stanford University, Stanford, CA, USA, 2000, aAI9961943. 85, 93

[204] J. Pearl, *Probabilistic reasoning in intelligent systems: networks of plausible inference*. Elsevier, 2014. 115

[205] F. V. Jensen, S. L. Lauritzen, and K. G. Olesen, "Bayesian updating in causal probabilistic networks by local computations," *Computational Statistics Quarterly*, vol. 4, pp. 269–282, 1990. 116, 118

[206] H. Guo and W. Hsu, "A survey of algorithms for real-time Bayesian network inference," in *Join Workshop on Real Time Decision Support and Diagnosis Systems*, 2002. 118

[207] J. Cheng and M. J. Druzdzel, "Confidence inference in Bayesian networks," *arXiv preprint arXiv:1301.2260*, 2013. 118

[208] R. G. Cowell, P. Dawid, S. L. Lauritzen, and D. J. Spiegelhalter, *Probabilistic networks and expert systems: Exact computational methods for Bayesian networks*. Springer Science & Business Media, 2006. 118

[209] J. Cheng, "Efficient stochastic sampling algorithms for Bayesian networks," *University of Pittsburgh*, 2001. 118

[210] F. G. Cozman *et al.*, "Generalizing variable elimination in Bayesian networks," in *Workshop on probabilistic reasoning in artificial intelligence*. Editora Tec Art São Paulo, Brazil, 2000, pp. 27–32. 118

[211] A. Darwiche, "Recursive conditioning," *Artificial Intelligence*, vol. 126, no. 1-2, pp. 5–41, 2001. 118

[212] K. Kanazawa, "Probability, time, and action," Ph.D. dissertation, PhD thesis, Brown University, Providence, RI, 1992. 133, 138, 146, 154

[213] P. P. Shenoy, "Binary Join Trees," in *UAI '96: Proc. of the 12th Int'l Conf. on Uncertainty in Artificial Intelligence, Reed College, Portland, Oregon, USA, August 1-4, 1996*, 1996, pp. 492–499. 133

[214] ——, "Binary join trees for computing marginals in the Shenoy-Shafer architecture," *Int. J. Approx. Reasoning*, vol. 17, no. 2-3, pp. 239–263, 1997. 133

[215] J. S. Ide and F. G. Cozman, "Random Generation of Bayesian Networks," in *Advances in Artificial Intelligence, 16th Brazilian Symposium on Artificial Intelligence, SBIA 2002, Porto de Galinhas/Recife, Brazil, November 11-14, 2002, Proceedings*, 2002, pp. 366–375. 136, 138

[216] A. Nicholson and J. Flores, "Combining state and transition models with dynamic Bayesian networks," *Journal of Ecological Modelling*, vol. 222(3), pp. 555–566, 2011. 136, 235

[217] J. A. Legge. (2017) Statistics Canada: Constructing box and whisker plots. [Online; accessed 13-February-2020]. [Online]. Available: https://www150.statcan.gc.ca/n1/edu/power-pouvoir/ch12/5214889-eng.htm 143

[218] W. F. Doolittle, "The attempt on the life of the Tree of Life: science, philosophy and politics." *Biology & Philosophy*, vol. 25, no. 4, pp. 455–473, 2010. 165

[219] B. Larget and D. L. Simon, "Markov chain Monte Carlo algorithms for the Bayesian analysis of phylogenetic trees," *Molecular biology and evolution*, vol. 16, no. 6, pp. 750–759, 1999. 165

[220] R. D. Page, "Tree View: An application to display phylogenetic trees on personal computers." *Bioinformatics*, vol. 12, no. 4, pp. 357–358, 1996. 165

[221] F. Ronquist and J. P. Huelsenbeck, "MrBayes 3: Bayesian phylogenetic inference under mixed models," *Bioinformatics*, vol. 19, no. 12, pp. 1572–1574, 2003. 165

[222] R. D. Page, "Space, time, form: viewing the Tree of Life." *Trends in ecology & evolution*, vol. 27, no. 2, pp. 113–120, 2012. 165

[223] J. Han, M. Kamber, and J. Pei, *Data Mining: Concepts and Techniques, 3rd edition*. Morgan Kaufmann, 2011. [Online]. Available: http://hanj.cs.illinois.edu/bk3/ 165

[224] D. H. Taenzer, M. Ganti, and S. Podar, "Problems in Object-Oriented Software Reuse," in *ECOOP '89: Proceedings of the Third European Conference on Object-Oriented Programming, Nottingham, UK, July 10-14, 1989.*, 1989, pp. 25–38. 165

[225] R. T. O'Donnell, A. E. Nicholson, B. Han, K. B. Korb, M. J. Alam, and L. R. Hope, "Causal Discovery with Prior Information," in *AI 2006: Advances in Artificial Intelligence, 19th Australian Joint Conference on Artificial Intelligence, Hobart, Australia, December 4-8, 2006, Proceedings*, 2006, pp. 1162–1167. 166

[226] K. B. Korb, "Learning Graphical Models," in *Encyclopedia of Machine Learning and Data Mining*, 2017, pp. 715–723. 166

[227] R. C. Read and D. G. Corneil, "The graph isomorphism disease," *Journal of Graph Theory*, vol. 1, no. 4, pp. 339–363, 1977. 167

[228] F. Jensen, F. V. Jensen, and S. L. Dittmer, "From influence diagrams to junction trees," in *Uncertainty Proceedings 1994*. Elsevier, 1994, pp. 367–373. 204

[229] M. Samiullah, D. Albrecht, and A. Nicholson, "Supplementary materials: iOOBN framework and case study," http://bayesian-intelligence.com/publications/TR2017_1_iOOBN_Supp.pdf, Bayesian Intelligence, Technical Report TR 2017/1, 2017. 206

[230] NGINX. (2001) PEP is an Earley Parser. [Online; accessed 02-February-2016]. [Online]. Available: http://www.coffeeblack.org/ 209

[231] T. Parr. (1989) Another Tool for Language Recognition. [Online; accessed 21-December-2016]. [Online]. Available: http://www.antlr.org/ 209, 217, 221

[232] F. V. Jensen and T. D. Nielsen, "Bayesian Networks and Decision Graphs," *Knowledge Eng. Review*, vol. 23, no. 4, p. 413, 2008. 214

[233] JGraph Ltd. (2017) JGraphX (JGraph 6) User Manual. [Online; accessed 02-February-2019]. [Online]. Available: https://jgraph.github.io/mxgraph/docs/manual_javavis.html/ 217, 221