Relational Data Mining Using Probabilistic Relational Models

Design, Implementation and Test

Aalborg University
Department of Computer Science, Frederik Bajers Vej 7E, DK 9220 Aalborg Øst

Title:

Relational Data Mining Using Probabilistic Relational Models – Design, Implementation and Test

Project period:
Master's Thesis
DAT6, Feb. 14th 2005 – Jun. 9th 2005

Project group:
d640a

Members of the group:
Morten Gade
Michael Gade Nielsen

Supervisor:
Manfred Jaeger

Number of copies: 5

Report – number of pages: 85

Total amount of pages: 85

Abstract

This thesis documents the design, implementation and test of Probabilistic Relational Models (PRMs). PRMs are a graphical statistical approach to modeling relational data using the Relational Language. PRMs consist of two components; the dependency structure and the parameters.

Our design is based on simplicity, flexibility, and performance. We explain the search over possible structures, using a notation of potential parents. The potential parents are used in four different search algorithms; one greedy, two random, and a hybrid between greedy and random. Also, we explicitly explain learning the parameters using sufficient statistics, by considering internal and external dependencies and how to keep track of the context.

We perform five different tests, showing that the penalty term of the score function can be tweaked to control the trade off between maximum likelihood and complexity. Also, our limited scalability test shows that our implementation scales near linear, although our implementation could be further optimized. The search test of the four algorithms show that introducing randomness is beneficial if combined with a greedy approach. The results also show, that although greedy finds the best model, the hybrid approach comes close in less time.

In comparison with our prior work, it is clear that PRMs are a very good alternative to propositional data mining in terms of descriptiveness.

Preface

This thesis documents the design, implementation and test of Probabilistic Relational Models (PRMs). The thesis has been written during DAT6 (8. semester) at Aalborg University, Denmark.

The reader should be familiar with our prior work [10]. A short summary of this, is provided for convince in the introductory parts of the thesis.

A special thanks to Manfred Jaeger for his guidance and assistance during the project.

Aalborg University, June 2005

Morten Gade

Michael Gade Nielsen

Contents

Chapter 1

Introduction

1.1 Introduction

Data mining has been, implicitly, synonymous with propositional data mining for decades. In propositional data mining, we only work with a single *homogeneous* table representation of the data. The term *data set* is often used about this propositional representation format. Most of today's data are, however, stored in relational databases, which is a *heterogeneous* data representation with relations between entities. Propositional data mining techniques cannot be directly applied to a relational database, nor utilize the relational structure. Instead the database must be flattened into a data set. A database is flattened by merging tables of interest into a single table representation. This process, see Figure 1.2, has two main disadvantages:

- Flattening a relational database often results in duplicated information, thus leading to **statistical incorrectness** since the data is skewed and does not represent the *true* distribution. Flattening the database in Figure 1.1(a), results in two entries in the propositional data set if a parent has two children, see Figure 1.2. Thus the information of the parent would be duplicated.
- Before flattening, the **attributes of the propositional data set must be determined**. This reduces the ability to explore new dependencies among relations, since we must determine which relations to explore. E.g.
  - If we flatten a relational database with a recursive relation, such as a parent and its children modeled by a generic person table, see Figure 1.1(b). What attributes should be included, the attribute of the child, its parents, and their parents etc. how far should the relational structure be extended?
  - If we flatten the relational database in Figure 1.1(a), we may include the attributes of a child's siblings, since these may be correlated.

However, this is suboptimal since we introduce duplicate information and null values, see Figure 1.2.

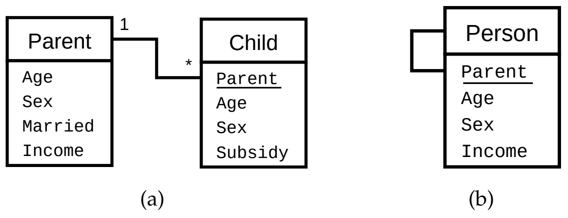
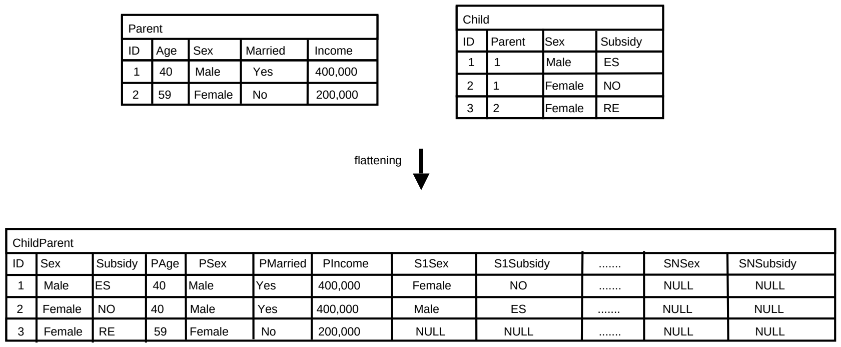


Figure 1.1: (a) A database with two tables, Parent and Child where a child has one parent. (b) A database with a parent-child relation modeled as a recursive relation.



Parent

ID	Age	Sex	Married	Income
1	40	Male	Yes	400,000
2	59	Female	No	200,000

Child

ID	Parent	Sex	Subsidy
1	1	Male	ES
2	1	Female	NO
3	2	Female	RE

flattening

ChildParent

ID	Sex	Subsidy	PAge	PSex	PMarried	PIncome	S1Sex	S1Subsidy	....	SNSex	SNSubsidy
1	Male	ES	40	Male	Yes	400,000	Female	NO	....	NULL	NULL
2	Female	NO	40	Male	Yes	400,000	Male	ES	....	NULL	NULL
3	Female	RE	59	Female	No	200,000	NULL	NULL	....	NULL	NULL

Figure 1.2: The result of flattening the database in 1.1(a). The figure shows how the database structure and its data are merged into a single table. In the single table, we can encounter duplicate information, null values, and we must anticipate which relations to explore. Also how should we determine the number of n siblings?

In many cases a propositional data mining algorithm is sufficient. Although in cases where the above disadvantages are not acceptable, we must resort to relational data mining (RDM), which also considers the relational structure. RDM has become a very widespread research topic, as more and more data are being stored in relational databases and a demand for knowledge extraction methods has emerged.

The next section describes two different approaches to RDM.

1.2 Relational Data Mining

1.2.1 Inductive Logical Programming

Just as many of today's data mining algorithms come from the field of machine learning, many of the RDM algorithms come from the field of inductive logic programming (ILP). In the ILP setting the relational data are expressed as logic programs using clauses, predicates and facts, where predicates corresponds to relations in a relational database. E.g the predicate $lives\_in(X, Y)$, is a relation between a person(X) and a city(Y).

Currently ILP covers the whole spectrum of data mining tasks, such as classification, regression, association rules and clustering [8]. ILP problems dealing with classification tasks typically search the space of possible clauses by using a general-to-specific search. The search is conducted by initially using a very general rule (covering all training examples) and stepwise refine it to only cover the positives training examples (completeness and consistency property). When one such clause is found the search stops. Figure 1.3 shows the search for a set of clauses which is complete and consistent with regard to the training examples. The training examples states whether a child receives an economical subsidy or not.

At first the rule at the root, $Subsidy(X, Y) \leftarrow$, is true for all training examples, hence not complete and consistent with the training examples. The search continues in a breadth/depth first manner, until a clause is met, which only classifies positives instances correctly. One such is found at node $Subsidy(X, Y) \leftarrow Parent(Z, X) \wedge Income(Z, low)$.

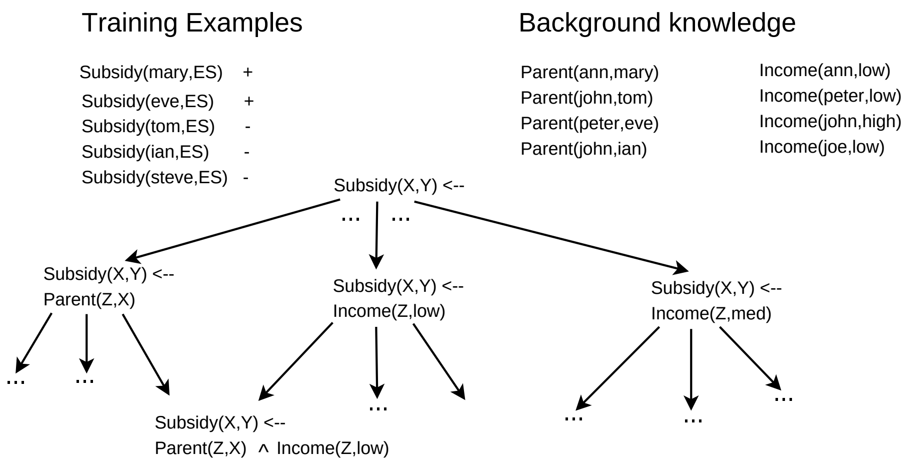


Figure 1.3: The search for a clause which is complete and consistent. The edges in the figure represent different paths in the search tree (the hypothesis space of possible ILP expressions) and nodes are ILP clauses. A + indicates positive training examples, which is the case when $Subsidy(X, Y) \leftarrow Parent(Z, X) \wedge Income(Z, low)$

Several well known learning algorithms have been implemented in the ILP setting and modified for handling relational data structures, e.g. TILDE [3] is an "upgrade" of C4.5 [27] and SCART [19] of CART [4]. The ILP setting seems a feasible approach to RDM, however it has some shortcomings. The most important disadvantage is a higher time and space complexity than other standard machine learning approaches. Traditional search methods in ILP explore the search space in, as shown previously, a general-to-specific or a specific-to-general fashion. More sophisticated search methods have to be derived in order to make ILP efficient when performing data mining on large relational databases [29].

Another weakness, which occurred in the earlier ILP settings, is that a conversion of the original data into equivalent logic expressions has to be made. Thereby the learning is not applied directly on the relational database, but on an extraction and conversion of it. However in today's settings, front-ends are used that cope with this problem.

Although ILP is the most widely studied approach to relational data mining, it is not the only one. Another approach is based on statistical graphical models, which is described below.

1.2.2 Statistical Graphical Models

Statistical graphical modeling of relational data has received much attention, in the last years, especially following the work by D. Koller [18]. In 1998, D. Koller and A. Pfeffer introduced the concept of Probabilistic Frame-Based Systems, which was revised by L. Getoor et. al. in 1999 as Probabilistic Relational Models (PRMs) [9]. PRMs extend the relational model—a commonly used representation for the structure of a database—by introducing concepts from Bayesian networks and learning. Thus, PRMs are a graphical language for modeling dependencies among relational data, just as Bayesian networks is for propositional data.

D. Heckerman provides a survey [14] (2004) of PRMs, plate models, and introduces a third language which is more expressive. Plate models [5] were developed by W. L. Buntine, as a language for compact representation of graphical models in which there are repeated measurements. In a plate model, each entity class is modeled as a *plate* which can intersect or overlap with other plates. An intersection amounts to a relationship between two or more plates.

The third language by D. Heckerman, is the Directed Acyclic Probabilistic Entity-Relationship (DAPER) model [14]. The model is an extension of the entity-relationship model, another common model for the abstract representation of a database structure. The DAPER model is closely related to plate models and PRMs. However, the DAPER model is more expressive than both existing models, and also helps to demonstrate their similarities. As a consequence, D. Heckerman describes a mapping between DAPER and PRMs, and between DAPER and plate models. Since DAPERs is more expressive, the mapping occurs by omitting the features of DAPER that is not covered in the two other models. The superset of features, is for instance the ability to assign constraints in ILP, such as disjunction, conjunction and existence, to dependencies.

Other approaches to statistical modeling of relational data include: Relational Bayesian networks by M. Jaeger [16], and Relational Dependency Net-

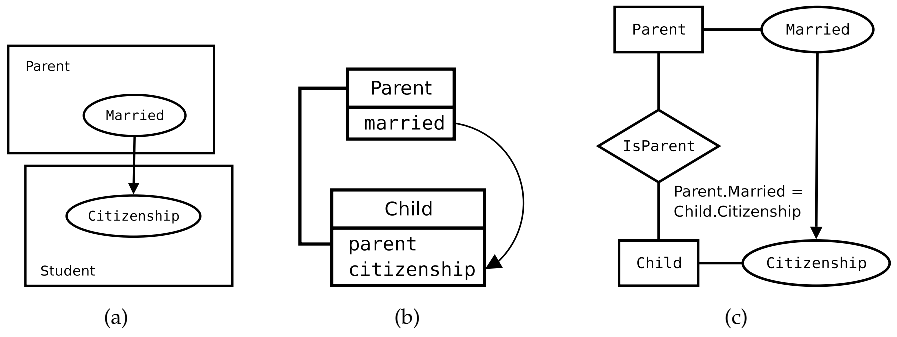


Figure 1.4: Three equivalent models, of three nonequivalent languages for describing probabilistic dependencies of relational data. (a) A Plate, (b) A PRM, and (c) A DAPER model showing the dependency between the citizenship of a child and the marital status of its parent.

works [22], Relational Probability Tress [23], and Relational Simple Bayes Classifier [24], by D. Jensen and J. Neville et. al. These models, are based on concepts from QGRPAH [2] which is a visual language for querying and updating graph databases, also by D. Jensen et. al. Further graph-based relational learning has been researched by L. B. Holder and D. J. Cook [15]—who also explores current and future directions—and by T. Washio and H. Motoda [30].

1.3 Objective

Statistical modeling of relational data, as described in the previous section, is an interesting topic. Although, plate models; PRMS; DAPER; extensions and related subjects have been widely studied, there exists no detailed documentation of implementing these models, which account for some of the novel and subtle concepts. Consequently, the objective of this project—motivated by our prior work [10]—is to implement PRMs and learning them from relational data. We choose PRMs, since they have been widely studied and described in the context of learning [9].

Our objective is as follows:

1. Design an implementation of probabilistic relational models [9], that supports learning the *dependency structure* and the *parameters*.

2. Provide a relational database structure of the problem relating to subsidies and institutions in Herning Kommune, by M. Gade and M. G. Nielsen [10], suitable for relational learning.

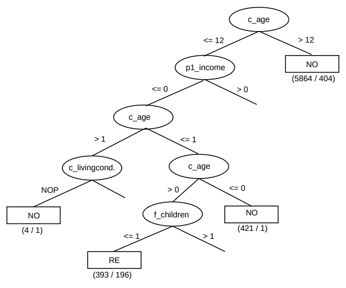
3. Prepare the UCI Movie Database [1] for relational learning, this include prepossessing and cleaning.

4. Perform tests that examine different aspects of the model on the two relational databases, such as execution time, scalability, tweak-ability etc.

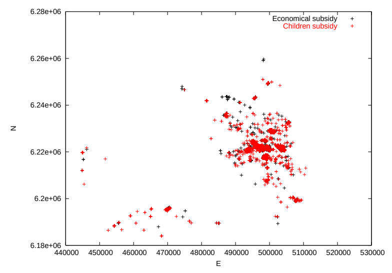
This direction of PRMs is motivated by our prior work, which is summarized in the next section.

1.4 Summary of Prior Work

In our prior work, we took a descriptive approach to data mining of children daycare subsidies and institutions in Herning Kommune. The objectives were related to cutting the cost of subsides, and alleviating an institution capacity problem in Herning Kommune. In particular, the objective was to characterize the families who receive various types of subsides and their choice of institution. Furthermore, we introduced a spatial aspect by retrieving the UTM coordinates of the residents in Herning Kommune. Consequently, we could examine if people in different parts of Herning Kommune were more likely to receive a subsidy, and if parents would choose a near-by institution for their children.



(a)



(b)

Figure 1.5: (a) A partial classification tree using the subsides as the class label. The tree revealed that certain attributes such as income, martial status, living condition were correlated. Unexpectedly, neither citizenship nor the UTM coordinates proved valuable in the classification. (b) The concentration of subsidies in Herning Kommune proved not to be, particular, correlated with the different people in Herning Kommune.

Before the objectives could be investigated, we had to flatten the relational database, provided by Herning Kommune. The database contained information about the children in Herning Kommune, their parents e.g. income, martial status, citizenship and age; which subsides the children had received; and which institution they attend. The database was flattened, and during this process several new attribute were generated, among these were the UTM coordinates and residential taxation values. The data set was subject to classification and clustering, by using different sets of attribute and class labels with different levels of detail. The classification tree was examined, see Figure 1.5(a), and the results were later confirmed by clustering and by simple data exploration, see Figure 1.5(b).

The result of the objectives were limited. No major discovery was done, although various minor discoveries emerged, which confirmed previous beliefs and assumptions by Herning Kommune. For further details, please see "Data Mining for Descriptive Modeling: of Children Daycare Subsidies and Institutions in Herning Kommune" [10].

Chapter 2

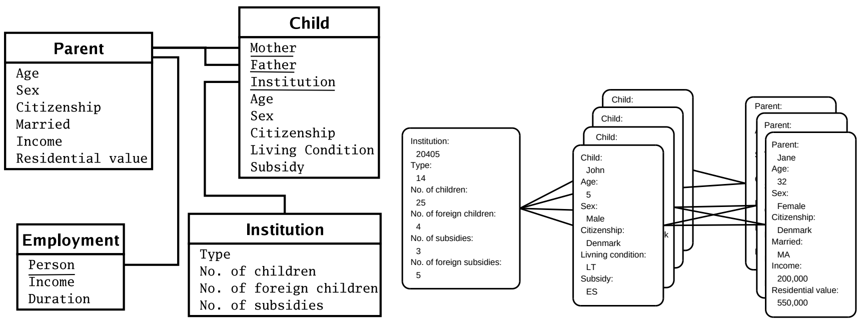
PRMs

In this chapter, we describe Probabilistic Relational Models (PRM). Most of the material is based on and inspired by the work of D. Koller and L. Getoor et al. [9; 11; 12].

2.1 Relational Language

The relational language allows us to specify the classes, i.e. the different kinds of objects, in our domain by means of a schema.

2.1.1 Schema



(a) A Schema (b) An Instance

Figure 2.1: (a) A relational schema for the daycare domain [10]. The schema consists of four classes: Child, Parent, Institution and Employment. A child has two parents, a mother and a father who are employed at a company, and the child attends an institution. (b) Shows an instance of the scheme, though omitting the Employment class, with three parents, four children who all attend the same institution. Two of the children have a mother and a father, while the remaining two only have a single parent.

A relational schema consists of a set of classes, $\mathcal{X} = \{X_1, \ldots, X_n\}$. In our *recurring* example of Figure 2.1(a) the classes are $Child$, $Parent$, $Institution$ and $Employment$. Each class X has a set of attributes, denoted $\mathcal{A}(X)$. Attribute A of class X is denoted $X.A$, and its range of values is denoted $\mathcal{V}(X.A)$. We assume that the range is finite. The value range e.g. of attribute *Child.Subsidy* is $\{SS, ES, CS, DS\}$ which represents a social, economical, children, and a disability subsidy, respectively.

Each class X is associated with a set of (reference) *slots*, denoted $\mathcal{R}(X)$. A slot represents a relation between two classes, such as the relation between a $Parent$ and a $Child$. We use $X.p$ to denote the slot p of X. Hence the class $Child$ has three slots, $Child.Mother$, $Child.Father$, and $Child.Institution$, see Figure 2.1(a) where the slots are underlined. A slot is typed i.e. for each slot ρ in X, $Domain(\rho) = X$ and $Range(\rho) = Y$, where $X, Y \in \mathcal{X}$. For instance, the slot *Child.Mother* has *Child* as its domain type and *Parent* as its range type. Furthermore, each slot ρ has an *inverse slot* ρ^{-1} such that if $Domain(\rho) = X$ and $Range(\rho) = Y$ then $Domain(\rho^{-1}) = Y$ and $Range(\rho^{-1}) = X$. For instance, the inverse slot of *Child.Mother* is $Parent.MotherOf$.

Finally, a *slot chain* is a combination of slots $\rho_1, \ldots, \rho_k$ (inverse or otherwise) such that $Range(\rho_i) = Domain(\rho_{i+1})$. A slot chain τ describes a set of objects from a class, these objects are called *tau-relatives*. E.g. the slot chain $Parent.MotherOf.Institution$ denotes the institution of a mother's child. Generally, we will use Of or At to denote the name of an inverse slot ρ^{-1}, such as $MotherOf$, $InstitutionOf$, and $EmployedAt$ etc.

2.1.2 Instance

An instance $\mathcal{I}$ of a schema specifies the set of objects, $I(X)$, for each class X; a value for each attribute $x.A$ for each object x; and a value y for each slot $x.\rho$, which is an object in the appropriate range type, i.e. $y \in Range(\rho)$. We use $x.A$ as opposed to $X.A$ when x is an object of X. For each object x in the instance, and for each of its attributes A, $\mathcal{I}_{x.A}$ denotes the value of $x.A$ in $\mathcal{I}$. Figure 2.1(b) shows an instance of the relational scheme in Figure 2.1(a).

2.1.3 Cardinality

A slot $X.\rho$ has a cardinality which specifies if object x has multiple related objects y of class $Range(X.\rho)$. The cardinality of a slot and its inverse is either one-to-one or one-to-many, and is domain specific, thus the user must input this information. For instance, the cardinality of slot $Employed$ in $Employment$ is one-to-one whereas the cardinality of its inverse, $EmployedAt$ in $Parent$, is one-to-many. See Figure 2.1(a) (inverse slots are not shown).

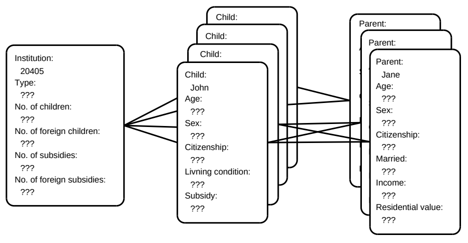
A slot chain τ also has a cardinality, which is determined by its slots $\rho_i \in \tau$. If all the slots are one-to-one, then the cardinality of the slot chain is also one-to-one, however, if the cardinality of one slot ρ_i is one-to-many, then the cardinality of the slot chain is also one-to-many. To appreciate the latter, consider the following slot chain: $A.B.C.D$ where B is one-to-many, and A, C, D are one-to-one. Hence the slot chain $A.B.C.D$ leads to multiple objects, since $A.B$ leads to multiple objects and *each of these* leads to one object in $C.D$.

The importance of cardinality will become apparent, in the next section when we introduce PRMs.

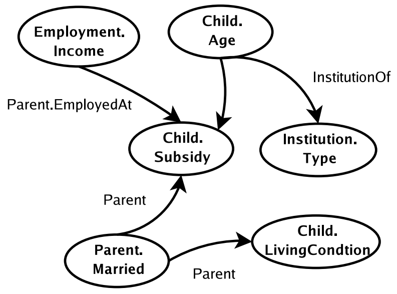
2.2 Probabilistic Relational Models

A PRM defines a probability distribution over a set of instances of a schema. We assume that the set of objects and the relations between them are fixed. These constraints are specified in a *relational skeleton*, which is a partial specification of an instance of a schema. The skeleton distinguishes between two kinds of attributes, *fixed* and *descriptive*, and it only gives the values for the fixed attributes but leaves the values of the descriptive attributes unspecified, see Figure 2.2(a). We want to specify a probability distribution over the descriptive attributes, i.e. over all possible instances of a schema.

Definition 1 (A Relational Skeleton, σ_r) *Is a partial specification of an instance of a schema. It specifies the set of objects $\sigma_r(X_i)$ for each class X_i, the values of the fixed attributes, and the relations that hold between the objects. However, it leaves the values of the descriptive attributes unspecified.*



(a) A Skeleton



(b) A Dependency Structure

Figure 2.2: (a) A relational skeleton and (b) the PRM dependency structure for the daycare domain, the slot chains are specified next to its corresponding parent (edge).

A PRM consists of two components, the dependency structure S and the parameters θ_S associated with it.

2.2.1 Dependency Structure

The dependency structure is defined by associating each descriptive attribute with a set of *formal parents*, $Pa(X.A)$. The formal parents $Pa(X.A)$ are attributes that have a direct influence on $X.A$. The formal parents will be instantiated in different ways for different objects in X, and there are two different types of formal parents:

Internal dependencies, where an attribute $X.A$ depends on another attribute $X.B$, of the same class, which is denoted $X.B \rightarrow X.A$. Hence, for any object $x \in \sigma_r(X)$, $x.A$ will depend probabilistic on $x.B$. For instance, an internal dependency could be $Child.Age \rightarrow Child.Subsidy$, such that $Child.Age$ has an influence on $Child.Subsidy$.

External dependencies, where an attribute $X.A$ depends on attributes of *related object(s)*, which is denoted $X.\tau.B \rightarrow X.A$ where τ is a slot chain. The slot chain τ must point to a class Y where $B \in \mathcal{A}(Y)$. For instance, an external dependency could be $Child.Mother.Employ.Salary \rightarrow Child.Subsidy$ where $Mother.Employ$ is the slot chain leading to $Employment.Salary$ from $Child$. External dependencies are further divided into single-valued and multi-valued dependencies.

- A **single-valued** dependency is where the τ-relatives only consist of one object. For instance, if a parent is only employed at one company.
- In a **multi-valued** dependency, the τ-relatives consist of more than one object, i.e. the slot chain leads to more than one object. For instance, if a parent is employed at more than one company, he would have two object salaries; $companyA.Salary$ and $companyB.Salary$. Thus, if the slot chain is multi-valued we must specify the probabilistic dependency of $x.A$ on the multi-set $\{y.B \mid y \in x.\tau\}$.

One approach is to specify the probabilistic dependency of each object $y.B$ in the multi-set, however with many objects this solution becomes intractable. Instead, we need a compact representation of the multi-set, which *aggregation* from database theory facilitates. Thus $x.A$ will depend probabilistically on some aggregation of the multi-set of τ-relatives. Formally, the aggregator $\gamma(X.\tau.B)$ returns a summary of the attribute B of the τ-relatives. Hence, the attribute $X.A$ can have $\gamma(X.\tau.B)$ as a parent such that for any $x \in X$, $x.A$ will depend on the value of $\gamma(X.\tau.B)$. There are several useful aggregation functions, such as the mode of a set, average, minimum and maximum etc. For instance, a child's subsidy could depend on the average income salary of its mother, $\gamma_{avg}(Child.Mother.Employ.Salary) \rightarrow Child.Subsidy$.

A dependency structure is shown in Figure 2.2(b).

2.2.2 Parameters

The parameters are a local probability model for an descriptive attribute $X.A$. The local probability model is defined, given the parents $Pa(X.A)$, by associating a conditional probability table (CPT) that specifies $P(X.A \mid Pa(X.A))$. Let $\mathbb{U} = Pa(X.A)$ where each of these parents $\mathbb{U}_i$—either an attribute or an aggregated value of τ-relatives—have a set of values $\mathcal{V}(\mathbb{U})$. For each n-tuple of values $u \in \mathcal{V}(\mathbb{U})$, where $n = |Pa(X.A)|$ is the number of parents, we specify a distribution $P(X.A \mid u)$ over $\mathcal{V}(X.A)$. This entire set of parameters comprises θ_S. For instance, we would need to associate a CPT with $Child.Subsidy$ that specifies $P(Child.Subsidy \mid \gamma_{avg}(Child.Mother.Employ.Salary))$.

In summary, these two components leads us to a definition of a PRM.

Definition 2 (A Probabilistic Relation Model, Π) *For each class $X \in \mathcal{X}$ and each descriptive attribute $A \in \mathcal{A}(X)$, in a relational schema R, we have:*

- *A set of parents $Pa(X.A) = \{\mathbb{U}_1, \ldots, \mathbb{U}_n\}$, where each U_i has the form $X.B$ or $\gamma(X.\tau.B)$, where τ is a slot chain and γ is an aggregation of $X.\tau.B$.*
- *A conditional probability table (CPT), $P(X.A \mid Pa(X.A))$.*

2.3 Bayesian Networks and PRMs

As mentioned, PRMs define a probability distribution over possible instances of a schema, by using a relational skeleton. The direct correlation with Bayesian networks occurs when we consider the objects, in the relational skeleton, and their attributes. With these, we can "compile" a Bayesian network, or in other words; for a skeleton σ_r the PRM structure induces a *ground* Bayesian network over the random variables $x.A$. The compilation consists of:

- Creating a random variable $x.A$ for every descriptive attribute $A \in \mathcal{A}(X)$ of every object $x \in \sigma_r(X)$, for each class $X \in \mathcal{X}$.
- Making $x.A$ depend probabilistically on internal $x.B$ and external $x.\tau.B$ parents. If the slot chain τ of $x.\tau.B$ is multi-valued, then the parent $y.B$ is the aggregation, $\gamma(x.\tau.B)$, of the set of random variables $\{y \mid y \in x.\tau\}$.
- Associating the CPT, $P(X.A \mid Pa(X.A)$ with each object $x.A$.

Figure 2.3 is a compilation of a PRM with two external dependencies; $Child$ $.Parent.Employed.Salary \rightarrow Child.Subsidy$ and $Child.Parent.Citizenship \rightarrow$ $Child.Subsidy$. There are three $Employment$ objects, $companyA$, $companyB$ and $companyC$; one $Parent$ object, $john$; and one $Child$ object, $jane$. In Figure 2.3(a) we need to specify the probability $P(jane.Subsidy \mid companyA.Salary, companyB$ $.Salary, companyC.Salary, john.Citizenship)$. In Figure 2.3(b) we apply the average aggregator to the $Employment$ objects, such that we only need to specify the probability $P(jane.Subsidy \mid avg(companyA.Salary, companyB.Salary,$ $companyC.Salary), john.Citizenship)$. The average of the salaries, is inserted as an intermediate *deterministic function* in Figure 2.3(b) to visualize that aggregate functions, such as average, facilitates a compact representation of a multi-set. Hence, the intermediate function need not be learned and facilities that learning the parameters is tractable.

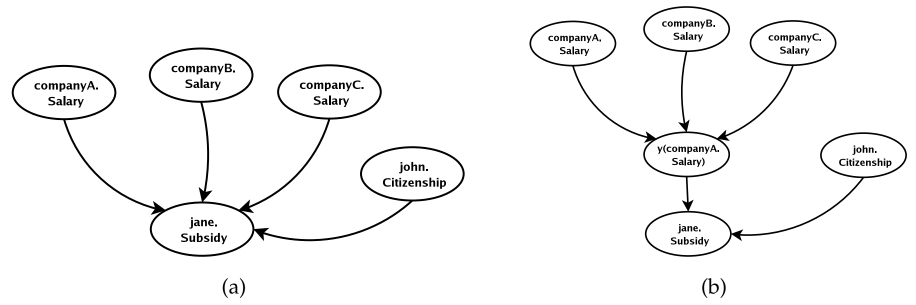


Figure 2.3: (a) The Bayesian network induced by the structure of a PRM and a relational skeleton of three employment objects, a parent and a child object. Without a compact representation of the $Employment.Salary$ objects, we would need to specify the probability of each object. (b) The Bayesian network with a intermediate aggregation object of the salaries.

Chapter 3

Learning PRMs

In this chapter, we describe how the parameters and the structure of PRMs are learned. Again, most of the material presented is based on and inspired by [9; 11; 12].

3.1 Parameter Estimation

We begin with estimating the parameters θ_S when the dependency structure S is known. Our task is to learn the CPTs for each attribute $X.A$ with parents $Pa(X.A)$. Estimating the parameters is based on a complete instance $\mathcal{I}$, which amounts to our training data, and the likelihood of the parameters given the training data, $L(\theta_S \mid \mathcal{I}, \sigma, S) = P(\mathcal{I} \mid \sigma, S, \theta_S)$. We work with the log of this function:

$$l(\theta_S|\mathcal{I},\sigma,S) = log\ P(\mathcal{I}|\sigma,S,\theta_S) = \sum_{X_i} \sum_{A \in \mathcal{A}(X_i)} \sum_{x \in \mathcal{O}^\sigma(X_i)} \log\ P(\mathcal{I}_{x.A} \mid \mathcal{I}_{Pa(x.A)}) \tag{3.1}$$

This equation is very similar to the log-likelihood of the training data given the parameters for a Bayesian network [13]. Actually, Eq. 3.1 is the likelihood function of the *ground* Bayesian network (ignoring the intermediate functions in Section 2.3, since they are deterministic) induced by the structure S given the skeleton σ_r, see Section 2.3. The only difference is that the parameters for different random variables in the network are forced to be identical, thus the parameters for objects $x.A$ of the same class X are identical. To appreciate this, review Section 2.3 which states that we associate the CPT $P(X.A \mid Pa(X.A))$ with each object $x.A$. Also the learned parameters can then be used to reason about other skeletons, which induce completely different Bayesian networks.

As a consequence of this similarity, we can apply the well-known techniques of Bayesian network learning. Such as maximum likelihood estimation, where the goal is to find the parameters θ_S that maximizes the likelihood

$L(\theta_S \mid \mathcal{I}, \sigma, S)$. This estimation can be done via *sufficient statistics* that aim to summarize the underlying distribution of the data. In the case of multinominal CPTs this amounts to frequency counting, which is also used in Bayesian network learning. We let $C_{X.A}(v, \mathbb{U})$ denote the number of times we jointly observe $\mathcal{I}_{X.A} = v$ and $\mathcal{I}_{Pa(X.A)} = \mathbb{U}$ in the training data.

Using the counts of the sufficient statistics and assuming multinominal CPTs, the maximum likelihood parameter setting $\hat{\theta}_S$ is

$$P(X.A = v | Pa(X.A) = \mathbb{U}) = \frac{C_{X.A}(v, \mathbb{U})}{\sum_{v'} C_{X.A}(v', \mathbb{U})}. \tag{3.2}$$

Another approach to parameter estimation, is a Bayesian approach that alleviates the main problem with maximum likelihood, namely *over fitting*. The Bayesian approach uses a prior distribution over the parameters to smooth the irregularities in the training data, making it more robust to noisy or unobserved data. The concept of smoothing is also applicable to maximum likelihood, where we simply add a constant count which represents unobserved data.

3.2 Structure

When the structure is not known prior to parameter estimation, we must first learn the structure and then learn the parameters. Structure learning consists of:

1. Defining the hypothesis space of legal candidate structures.
2. Evaluating different candidate structures, such that we may qualify which structure fits the data best.
3. Searching the hypothesis space for a good structure.

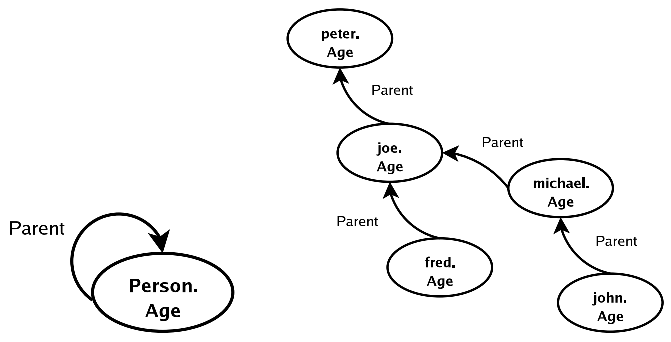
We describe the three aspects in the following sections.

3.2.1 Hypothesis Space

The hypothesis space [9] is defined by structures that are *acyclic*, such that no attribute $X.A$:

- Depends *directly* on itself $X.A \rightarrow X.A$ or $X.\tau.A \rightarrow X.A$
- Or *indirectly* on itself $X.A \rightarrow X_1.A_1 \rightarrow X_2.A_2 \rightarrow X.A$.

If the structure (at the class-level) is acyclic, then the ground Bayesian network (at the object-level) is also acyclic. However, if the structure is cyclic then the ground Bayesian network may not be, see Figure 3.1.



(a) Cyclic Dependency Structure

(b) Acyclic Ground Network

Figure 3.1: (a) The cyclic dependency structure which specifies that a person's age depends on the age of its parent, (b) the ground Bayesian network that resolves the cyclicity, since a parent cannot be a parent of itself.

Although, the cyclicity may be resolved at the object-level of the ground Bayesian network, we will ignore this issue in our design and implementation due to complexity. Since we have *no* way of knowing if the cyclicity resolves itself—different skeletons may introduce a cycle and others may not—we would need to allow the user to enter domain specify information, e.g. that no parent can be a parent of itself as in Figure 3.1, this is referred to as *guaranteed acyclic slots* by L. Getoor [9]. Instead, we will simply define the hypothesis space as structure that are DAGs. Also, the relaxation of guaranteed acyclic slots does not seem to expand the search space of the relational databases in Section 6, as none of these have recursive relations.

3.2.2 Evaluating a Structure

Once the hypothesis space of legal candidate structures has been determined, we need a measure for evaluating candidate structures. A candidate structure S_k is evaluated with regard to how well it fits the data and its complexity. We adopt the Bayesian Information Criteria (BIC)

$$S_{BIC}(S_k, \theta_S, \mathcal{I}, \sigma) = -l(\theta_S | \mathcal{I}, \sigma, S_k) + d_k \, log \, n, \tag{3.3}$$

where d_k is the number of parameters and $n = |\mathcal{I}|$ is the size of the database.

The first term, $-l(\theta_S|\mathcal{I}, \sigma, S_k)$, is the negative log-likelihood of Eq. 3.1. The second term, penalizes the complexity of the model using the number of parameters.

Another score function, used by L. Getoor [9] is the *marginal likelihood*. The marginal likelihood is a complex quantity, also computational wise, and one problem is that it requires a prior probability distribution over the structures. This is complicated by the fact that there exists infinitely many structures, in the presence of a recursive relation. Thus, we will settle for our more simple BIC inspired score function, and refer to L. Getoor [9] for further information on other scoring functions.

3.2.3 Structure Search

With the score function in place, we can evaluate which candidate structures in the hypothesis space fit the data best, i.e. we can find an *optimal* structure. However, it is intractable to consider *every* candidate structure using a exhaustive search. Finding the optimal structure of a PRM is at least NP-Hard, since a Bayesian network is simply a PRM with one class and no external relations, and for Bayesian networks the optimal structure search is NP-Hard [7].

Consequently, we must resort to search heuristics and rely on them to find a *good* structure. The most straightforward method is a greedy search (hill-climbing), where a current candidate structure is maintained and iteratively modified to improve the structure score. There are two basic operations that modifies the structure, *remove* dependency and *add* dependency. A third operation *reverse* dependency can be derived from these two. The structure score is used as a metric, and the search is e.g. terminated once the score decreases. A random restart could be applied to escape local minima.

When modifying the structure by adding dependencies, we need to know which dependencies to consider. The *potential parents* $Pot_k(X.A)$ of an attribute $X.A$ are the attributes that can be reached by a slot chain of length k. With length 0 only the attributes within the same class are considered, i.e. $Pot_0(X.A) = \mathcal{A}(X)\backslash X.A$ where $X.A$ is excluded as not to introduce a cycle. Given length k the attributes following a slot chain of length less than or equal to k are included, see Figure 3.2.3.

See the Chapter 4 for further details about the search algorithms, potential parents etc.

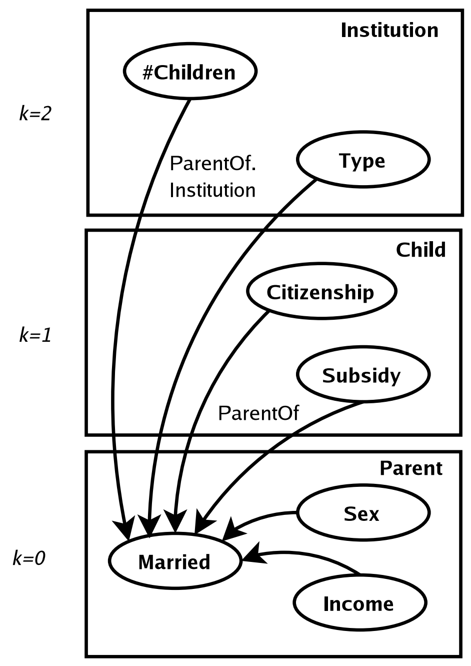


Figure 3.2: The potential parents of Parent.Married, $Pot_k(Parent.Married)$ with $k = 0$, $k = 1$, and $k = 2$, using a subset of the Daycare relational database. Each box represents a class, a node represents an attribute, and edges represents potential parents.

Chapter 4

Design

The previous chapters described the theory behind Probabilistic Relational Models. In this chapter, we present the design of PRMs as described in Chapter 2 and 3. As a consequence, this chapter describes further details, which is necessary for an implementation.

4.1 Design Criteria

In this section, we will shortly discuss the design criteria that are important (in order of priority).

1. Simplicity of the design is very important, since we do not want to increase the complexity of an already complex model, although we need to address certain design issues that may seem insignificant.

2. Flexibility of the design is important since several extensions to PRMs have been described, such as *structural uncertainty* [11] and *hierarchical models* [12; 25], and we want our design to be flexible to these extensions.

3. Performance is important, since we wish to use PRMs to perform relational data mining. Thus, our learning procedure needs to perform well. Also it must be able to handle large amounts of data efficiency, i.e. it must be scalable.

The next section describes the components and classes.

4.2 Overview and Classes

The design is based on the different elements of a PRM, as introduced in Chapter 2. Hence many of the elements of the relational language and PRMs are mapped into a corresponding class. The class diagram in Figure 4.1 shows the various inter-dependencies of the classes, differing between *aggregation*, *inheritance*, and *association*. The classes are clustered together in coherent components, and the following describes the components and classes.

The first component represents the basic PRM structure outlined in Chapter 2:

Prm: This class represents a PRM with a `Graph` as its dependency structure and associated parameters, along with the `Class`es in the relational scheme.

Class: Represents a class X with fixed (`FixedAtt`) and descriptive (`DescAtt`) attributes $\mathcal{A}(X)$, and `Slots` $\mathcal{R}(X)$. A class has a *Name* property, which is used to specify the corresponding table in the database.

Slot: Represents a slot ρ between two classes X and Y. The primary and foreign key of the slot are modeled by two `FixedAtts`. A slot has a cardinality, see Section 2.1.3, which is retrieved with the *Cardinality* property. Furthermore it has a *Name*, *Domain*, and *Range* property as explained in Section 2.1.

SlotChain: Represents a slot chain τ that contains a sequence of `Slots`. A slot chain also has a *Cardinality* property that returns the cardinality of its slots, see Section 2.1.3, which are used to determine which chains are multi-valued. The sufficient statistics needs a unique identification of a slot chain, thus a slot chain has a *UniqueName* property, see Section 4.5.

Att: Represents a generic attribute, which belongs to a `Class`. An `Att` has a *Name* property which specifies a column in the table. It also has a *Range* property which specifies the range type, which is either discreet or continuous. Continuous attributes must be *discretized* as we cannot handle continuous attributes.

DescAtt: Represents a descriptive attribute with a conditional probability table `Cpt`, and a number of parents modeled as `Edge`s.

FixedAtt: Represents a fixed attribute. It does not have a `Cpt` nor any parents, however it can be the parent of a `DescAtt`.

Graph: Represents the dependency structure with dependencies between attributes.

Edge: Represents an internal or external dependency between a `DescAtt` and its parents. An `Edge` has a slot chain if it is external, and an aggregator if the external dependency is multi-valued.

Cpt: Represents the conditional probability table with `CptEntry`s.

CptEntry: Represents an entry of the `Cpt` with values $X.A = v$ and $Pa(X.A) = \mathbb{U}$, probability $P(X.A = v| Pa(X.A) = \mathbb{U})$, and count $C_{X.A}(v, \mathbb{U})$.

The second component represents an instance of a relational scheme, i.e. the database.

Database: Provides the database interface, which facilitates executing SQL queries. It has two functions, *Connect* which establishes a connection to the database, and *GetTable* which retrieves the internal data structure of the `Cpt` from the database.

Statistics: Provides an interface for computing the sufficient statistics for the values of an attribute and its parents. It has two function, *Count* which counts the number of joint values, and *CreateView* which creates a view used to calculate the aggregated values in a multi-valued slot chain, see Section 4.5.

DataGrouping: Represents an abstract class for data grouping.

EWD: Represents data grouping as Equal Width Discretization [32].

EFD: Represents data grouping as Equal Frequency Discretization [32].

The third component represents learning the PRMs, thus most of the classes are purely functional objects.

StrucureLearner : Is responsible for learning the structure of a PRM.

ParameterLearner : Is responsible for learning the parameters of a PRM.

Search: Represents an abstract class for structure search.

Greedy: Represents a greedy approach for searching.

Random: Represents a random approach for searching.

Hybrid: Represents a hybrid approach for searching.

4.3 Dependency Structure Search

The dependency structure S of a PRM is represented by a `Graph` that contains the attributes `Att` and the `Edge` dependencies between them, see Figure 4.2.

Each `Edge` represents a parent which has a `SlotChain`. If the slot chain is multi-valued, it has one of the following discretized aggregators γ:

- γ_{max}, which returns the maximum numerical value of the multi-set.
- γ_{min}, which returns the minimum numerical value of the multi-set.
- γ_{sum}, which returns the sum of all the numerical values in the multi-set.
- γ_{avg}, which returns the average of all the numerical values in the multi-set.

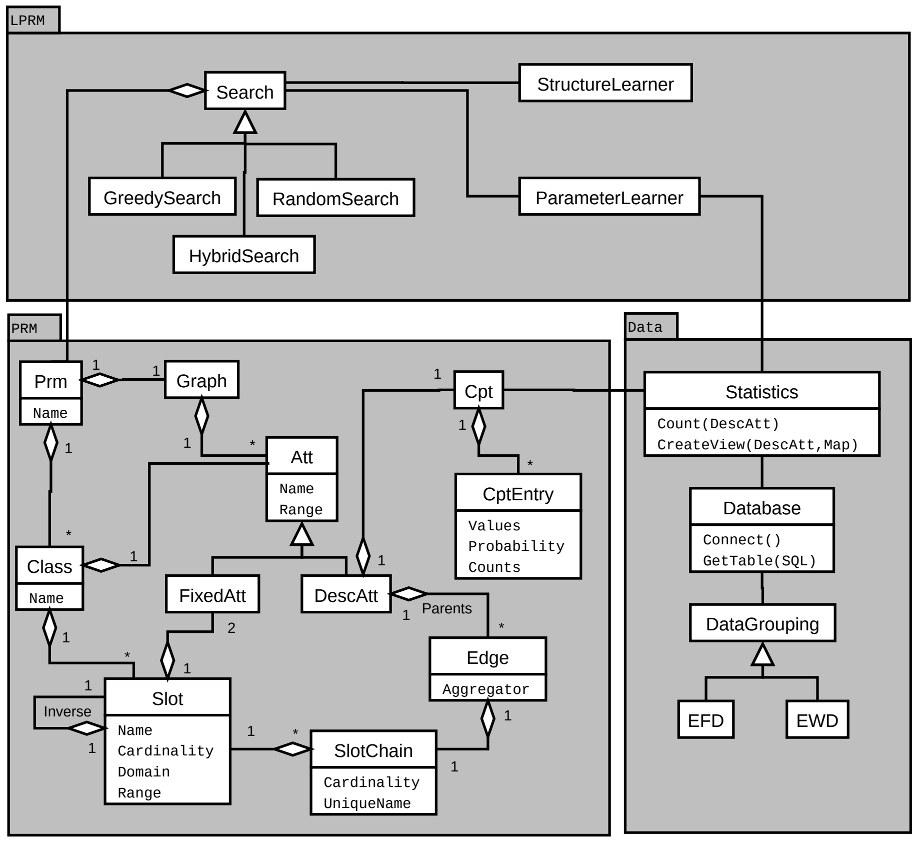


Figure 4.1: The class diagram divided into three clusters: LPRM, PRM and Data. There are three inter-dependencies aggregation, inheritance and association.

- γ_{mod}, which returns the most frequent nominal value (mode) of the multi-set .

Only the γ_{mod} aggregator works on nominal values, the others are restricted to numerical values.

The *potential parents*, as briefly explained in Section 3.2.3, is actually a *search over possible slot chains and aggregators*. Since, the potential parents $Pot_k(X.A)$ are decided by the slots that can be reached from $X.A$ within length k. We can now define a potential parent, which comprises the dependency search space:

Definition 3 (Potential Parent) *The triple $(Y.B, \tau, \gamma)$ is a potential parent to $X.A$ if Y is reached by following τ from X, and γ is the aggregator if τ is multi-valued. $\tau = \emptyset$ if $Y = X$.*

A potential parent is represented by an `Edge` in our design.

The slot chain search is performed by using either a breadth-first or a depth-first inspired search, however, we need to impose some restrictions to avoid cycle slot chains e.g. such as $Institution.InstitutionOf.Institution$ etc. (recall

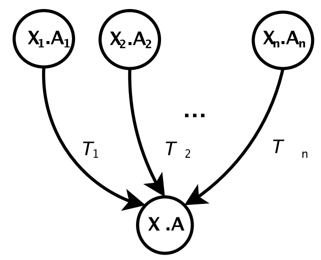


Figure 4.2: An attribute X.A with parents $X_1.A_1, X_2.A_2, \ldots, X_n.A_n$. Each external dependency is routed through a chain of slots, τ. $\tau = \emptyset$ if the dependency is internal.

the slots with the postfix Of and At are inverse slots). Consequently, the solution is to only allow *forward movement* of the slot chain search, such that for a slot chain we will not allow *it* to revisit classes. See Listing 4.1 for algorithmic details of the breadth-first slot chain search, $SlotBFS_k(X.A)$. Hence, if $Father$ is our current slot chain, starting from the class $Child$, the only option for expanding $Father$ is by slot $Employed$ since we cannot revisit the class $Parent$. The result of e.g. $SlotBFS_2(Child.Subsidy)$ of Listings 4.1 is the following slot chains $\{Institution, Father, Mother, Father.EmployedAt, Mother.EmployedAt\}$, see Figure 4.3 and Figure 2.1(a) for the database schema.

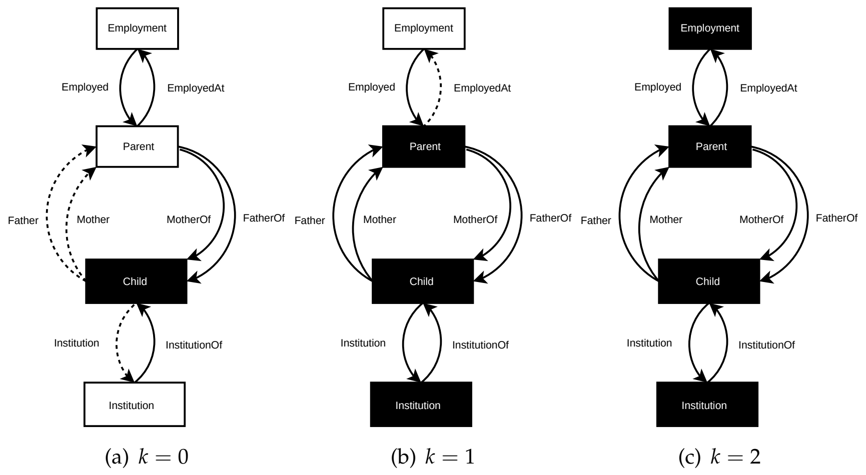


Figure 4.3: A slot chain search from Child, with (a) $k = 0$, (b) $k = 1$, and (c) $k = 2$. A white box represents unvisited classes, whereas a black box represents a visited classes of the breadth-first search. Dashed edges represents possible steps in the slot chain search. The result of e.g. $SlotBFS_2(Child.Subsidy) = \{Institution, Father, Mother, Father.EmployedAt, Mother.EmployedAt\}$.

The slot chains of the search, need to be transformed into potential parents, which $Pot_k(X.A)$ is responsible for, see Listing 4.2. $Pot_k(X.A)$ considers each slot chain τ, from $SlotBFS_k(X.A)$, and follows τ to its last class i.e. $Range(\rho_l)$ where $l = |\tau|$. E.g. the last class of slot chain $Mother.EmployedAt$

Listing 4.1: $SlotBFS_k(X.A)$ – the slot chain search for X.A

```
$chains = \emptyset$
$queue = <>$
for each $\rho \in \mathcal{R}(X)$ // slots in class X
    $\tau = \rho$ // create a slot chain with a single slot
    queue.enqueue($\tau$)

while $|queue| > 0$
    $\tau =$ queue.dequeue()
    $\rho_l =$ last slot of $\tau$

    if $|\tau| > k$
        break
    $chains = chains \cup \tau$

    for each $\rho_i \in \mathcal{R}(Range(\rho_l))$
        if $\rho_i$ results in forward movement from $\rho_l$
            $\tau = \tau.\rho_i$ // expand $\tau$ with $\rho_i$
            queue.enqueue($\tau$)
return $chains$
```

is the class *Employment*. At *Employment* the algorithm adds each attribute $B \in \mathcal{A}(Employment)$ to the list of potential parents. If the slot chain is multi-valued, we explicitly add each parent four times due to the aggregators, to facilitate the search over aggregators (line 9-10 in Listing 4.2).

$Pot_k(X.A)$ initiates the search for potential parents, which uses the result of the slot chain search $SlotBFS_k(X.A)$. The potential parents are used in the search algorithms for adding dependencies (`Edges`). This is the topic of the next section.

4.4 Search Algorithms

The following sections describe the various search algorithms we have applied in our PRM setting.

4.4.1 Greedy

This is the most straight forward algorithm. A current candidate structure is maintained, and dependencies are iteratively added in order to improve the structure score. In each iteration the best dependency is added. The pseudo code for the greedy algorithm is shown in Figure 4.3.

Listing 4.2: $Pot_k(X.A)$ – the potential parents of $X.A$

```
$pot = \emptyset$
$\tau_{X.A}$ = $SlotBFS_k(X.A)$
for each $\tau \in \tau_{X.A}$ // external dependencies
    $Y$ = $Range(\rho_l)$ where $l = |\tau|$ // last class of $\tau$

    for each $Y.B \in \mathcal{A}(Y)$
        if $\tau$ is multi-valued // multi-valued dependencies
            if $\tau$ contains numerical values
                $pot = pot \cup \{(Y.B, \tau, \gamma_{max}), (Y.B, \tau, \gamma_{min}),$
                                $(Y.B, \tau, \gamma_{avg}), (Y.B, \tau, \gamma_{sum})\}$

            else
                $pot = pot \cup \{(Y.B, \tau, \gamma_{mod})\}$
        else // single-valued dependencies
            $pot = pot \cup \{(Y.B, \tau, \gamma_{none})\}$

for each $X.B \in \mathcal{A}(X) \backslash X.A$ // internal dependencies
    $pot = pot \cup \{X.B, \emptyset, \gamma_{none}\}$

return $pot$
```

Listing 4.3: *GreedySearch*(Prm)

```
m = 0 // The current chain length
bestStructureScore // The best structure score achieved
bestCandidate // The best candidate with current chain length.
currentScore // Current score of the Prm
while(doSearch)
  for each descriptive attribute $X.A$
    for each potential parent $Y.B \in Pot_m(X.A)$
      add edge from $Y.B$ to $X.A$
      currentScore = ScoreStructure($Prm$)
      remove edge from $Y.B$ to $X.A$
      if(currentScore > bestCandidate)
        bestCandidate = currentScore
  if(bestCandidate > bestStructureScore)
    bestStructureScore = bestCandidate
    add the best edge to $Prm$
  else
    doSearch = false
  m++
return bestStructureScore
```

The search stops when adding further dependencies does not improve the current structure score. However, a serious disadvantage of this approach, is that the search can be caught in a local maximum and stops.

In our test setting we are trying to overcome this disadvantage by using the following approach. Instead of terminating when no dependency addition increase the score, see Listing 4.3, it is allowed to add k dependencies in order to see if these can improve the score. This is however not shown in the listing.

If an improvement can be obtained within k number of steps, the algorithm continues as the one describe above, otherwise it terminates. By using this strategy a local maximum can be avoided, but not guaranteed. However, the value of k are difficult to determine and must be determined using heuristics.

4.4.2 Random Search

In order to cope with the difficulties of local maxima, we also want to address algorithms that may perform better than greedy.

Subset Random

This algorithm is very simple and can be seen as a subset of greedy. It selects a random attribute $X.A$ and m number of potential parents $Pot_m(X.A)$ and adds, if legal, the best dependency to the current structure. The number m is currently set to 10% of the potential parents. The optimal value of m has to be found using heuristics. If no dependency increases the structure score, it tries m new dependencies. This is allowed for k tries. If still no dependency is found which increases the score, the algorithm terminates.

Because there is a reduced number of score calculation and parameter estimation involved in this search, this algorithm is superior in speed.

Normalized Random

This algorithm uses information about how good the different dependencies are. By recording the structure score that we would obtain by introducing a dependency from $Y.B \rightarrow X.A$ we get this information. This is done for all possible legal dependencies of a given chain length i. The structure scores are normalized and the normalized value become the probability of choosing that dependency.

However there might be some issues with this approach. There are two main issues that we briefly will discuss.

1. **Score Function**

   An optimal structure score in our setting is zero. This effects that the worst dependencies are having larger normalized probabilities. The solution is to record the increment in the structure score when introducing a new dependency, and use these increments in order to normalize. This ensures that we have a higher probability of selecting the best dependencies.

2. **Number of possible edges**
   Assume a dependency $Y.B \rightarrow X.A$ which has a score increase (approaches zero) that is more than a tenfold higher than any other possible score that can be induced by any dependency of the same chain length. Then this dependency is an obvious choice, but this obvious choice can be suppressed by the number of possible dependencies. Assuming that there are 1000 possible dependencies, and the sum of all scores is ten times larger than the score of the dependency $Y.B \rightarrow X.A$. Now there is only 10% chance of selecting the best dependency, even though it is more than tenfold better than any other dependency. So this algorithm can suffer by the fact of many possible dependencies.

   The solution is to only consider the k best dependencies when calculating the normalized probability.

Listing 4.4: *HybridSearch*(Prm)

```
randomStepProb = 0.7 // chance of taking a random step
randomStepRate = 0.5 // decrease random probability rate
maxRandomSteps = 3 //   steps before decreasing randomStepProb
while(doSearch)
  r = random number between 0 and 1
  for(int i = 0; i < maxRandomSteps; i++)
    if(r < randomStepProb)
      makeRandomStep();
    else
      bestCandidate = makeBestStep();
      if(bestCandidate > bestStructureScore)
        bestStructureScore = bestCandidate
      else
        doSearch = false
  randomStepProb *= randomStepRate //decrease randStepProb
return BestStructureScore
```

4.4.3 Hybrid Search

This algorithm tries to combine the nature of greedy search and random search strategies. It is inspired by the Random Subset Search (RSS)[12] algorithm.

In each iteration of the algorithm, it is possible to take a random step. As the number of iterations increases the possibility of taking a random step decreases. This is done in order to say that our starting point is less important, but as the search continues we trust more and more in our best steps. Hopefully this would lead us to a global maximum instead of a local one.

The pseudo code for this algorithm is shown in Listing 4.4. The function *makeRandomStep*() adds an arbitrary dependency, whereas *makeBestStep*() adds the best possible dependency.

Several other hybrid algorithms exist, one of the more noticeable is mentioned by J. D. Nielsen et. al. [26], where a constant k determines the degree of greediness in the algorithm.

4.5 Computing Sufficient Statistics

Learning the parameters θ_S of a structure S, involves querying the database to count the frequency of the joint values of an attribute $X.A$ and its parents $Pa(X.A)$. The count is denoted $C_{X.A}[v, \mathbb{U}]$ which counts the joint frequency of the value v of $X.A$ and the value tuple $\mathbb{U}$ of its parents. This was briefly explained in Section 3.1. However, due to efficiency we will simply count the entire sufficient statistics for an attribute $X.A$, i.e $C_{X.A}$. Next, we need to compose the actual SQL query that computes the count $C_{X.A}$ from the relational database. First, we only consider internal dependencies, then single valued external dependencies, and show how to keep track of the context. Lastly, we show how to handle multi valued external dependencies.

4.5.1 Internal Dependencies

First, we only consider the case where all dependencies are internal, i.e. $X.B \rightarrow X.A$. In this case, we only need to select the attribute $X.A$ and its parents, $Pa(X.A) = \{X.A_1, X.A_2, \ldots, X.A_n\}$. Also, we need to group equal values together such that we can count the frequency. The following SQL query computes the count $C_{X.A}$, see Figure 4.4. Note that X is omitted, since each attribute has this as a prefix.

SELECT $A,\ A_1,\ \ldots,\ A_n,\ count(*)$ **FROM** X
GROUP BY $A,\ A_1,\ A_n$

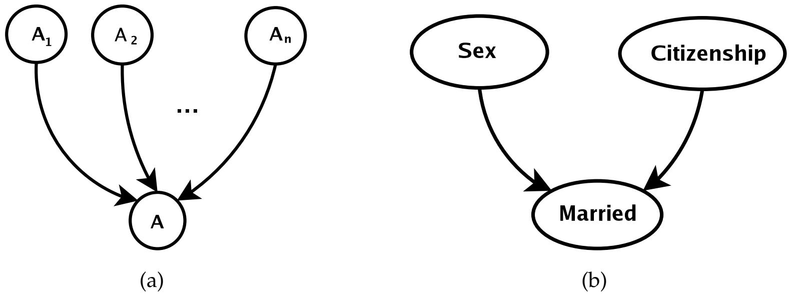


Figure 4.4: (a) A dependency structure that only contains internal dependencies, $X.B \rightarrow X.A$. (b) An example of an internal dependency structure, with a dependency $Parent.Citizenship \rightarrow Parent.Married$ and $Parent.Sex \rightarrow Parent.Married$.

The example in Figure 4.4(b), amounts to the SQL query below, which counts the frequency of $C_{Parent.Married}$.

SELECT $Married, Citizenship, Sex, count(*)$ **FROM** $Parent$
GROUP BY $Married, Citizenship, Sex$

4.5.2 External Dependencies, Single-Valued

Next, we move on to the more challenging problem of considering external dependencies such as $X.\tau.B \rightarrow X.A$, where τ is single-valued. In this case, we need to include the class name in the query, and also perform joins across the various classes (tables) that occur in the dependency of $X.A$. A dependency is routed through a slot chain τ, thus we must join the slots that occur in τ to reach attribute B. To facilitate this, we use $\tau^*(X.A) = \{\tau_1, \tau_2, \ldots, \tau_n\}$ to denote all the slot chains of $X.A$'s parents, see Figure 4.5. We use $\rho^*(X.A)$ to denote an *ordered* set of all the slots in $\tau^*(X.A)$, i.e. ρ^* decomposes all the slot chains into slots, while maintaining the ordering such that if $\tau^*(X.A) = \{\rho_1.\rho_2, \rho_3.\rho_4\}$ then $\rho^*(X.A) = (\rho_1, \rho_2, \rho_3, \rho_4)$. We let $m = |\rho^*(X.A)|$ denote the number of elements in ρ^*.

Furthermore, we need to be able to retrieve the destination class of a slot ρ and its foreign and primary key. These are necessary, since we need two keys to join on. We use $Range(\rho)$ to denote the destination class Y of ρ, see Section 2.1. The primary and foreign key of ρ is denoted $p_{key}(\rho)$ and $f_{key}(\rho)$, respectively. We need to perform a join operation for each slot $\rho \in \rho^*(X.A)$, such that the range and the domain of $X.A$ is joined with regard to the primary and foreign key. Thus, the SQL query for $C_{X.A}$ becomes:

SELECT $X.A, X_1.A_1, X_2.A_2, \ldots, X_n.A_n, count(*)$ **FROM** X
LEFT JOIN $Range(\rho_1)$ **ON** $Range(\rho_1).p_{key}(\rho_1)$ = $Domain(\rho_1).f_{key}(\rho_1)$
LEFT JOIN $Range(\rho_2)$ **ON** $Range(\rho_2).p_{key}(\rho_2)$ = $Domain(\rho_2).f_{key}(\rho_2)$
$\vdots \qquad \vdots \qquad \vdots \qquad \vdots$
LEFT JOIN $Range(\rho_m)$ **ON** $Range(\rho_m).p_{key}(\rho_m)$ = $Domain(\rho_m).f_{key}(\rho_m)$
GROUP BY $X.A, X_1.A_1, \ldots, X_n.A_n$

Where $\rho_i \in \rho^*(X.A)$ and $X_j.A_j \in Pa(X.A)$. The following shows the SQL query for the structure shown in Figure 4.5(b).

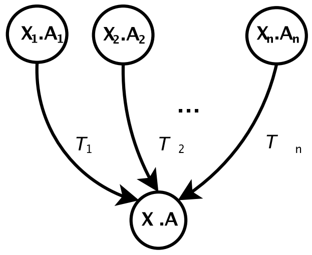
SELECT *Child.Subsidy*, *Institution.Children*, *Employment.Salary*, $count(*)$ **FROM** *Child*
LEFT JOIN *Institution* **ON** *Institution.id* = *Child.institution*
LEFT JOIN *Parent* **ON** *Parent.id* = *Child.mother*
LEFT JOIN *Employment* **ON** *Employment.id* = *Parent.id*
GROUP BY *Child.Subsidy*, *Institution.Children*, *Employment.Salary*

In the example there are two dependencies, both with *Child.Subsidy* as the child, thus $\tau^*(Child.Subsidy) = \{Institution, Mother.EmployedAt\}$ and $\rho^*(Child.Subsidy) = \{Institution, Mother, EmployedAt\}$.

The joins facilitate that the dependencies are routed through the correct slots, such that the context in which an attribute appears is correct. Although, in some cases we need to keep track of dependencies that may end at the same attribute but is routed through different slots. The next section explains this issue.

4.5.3 Tracking Context

If we try to use the SQL query presented in Section 4.5.2 on the dependency structure in Figure 4.6, we get the following result.



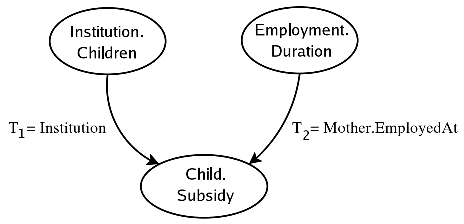


Figure 4.5: (a) A dependency structure that contains external dependencies $X_n.\tau.A_n. \rightarrow X.A$ (b) An example of a dependency structure, where Child.Mother.Employment.Duration $\rightarrow$ Child.Subsidy and Child.Institution.Children $\rightarrow$ Child.Subsidy.

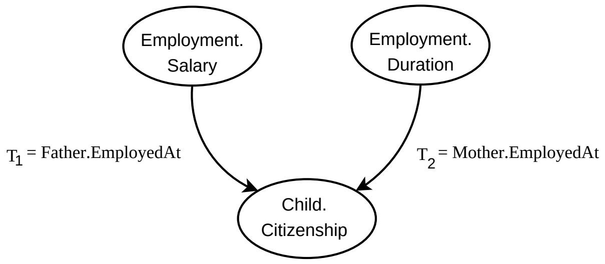


Figure 4.6: A dependency structure where we must be aware of the context of the dependencies, in order to distinguish between the mother and father.

SELECT $Child.Subsidy, Employment.Salary, Employment.Duration, count(*)$ **FROM** $Child$
LEFT JOIN $Parent$ **ON** $Parent.id = Child.mother$
LEFT JOIN $Parent$ **ON** $Parent.id = Child.father$
LEFT JOIN $Employment$ **ON** $Employment.id = Parent.id$
GROUP BY $Child.Subsidy, Parent.Age, Employment.Duration$

This SQL query, poses two related problems. The query is ambiguous and thus its syntax is illegal. The problem is that, there is only one join on the class $Employment$, but what does $Parent.id$ referrer to? Since $Parent$ is seen in two different contexts, a $Father$ and a $Mother$ as determined by the slots. Thus, we must be aware of which parent (slot) we are looking through to make a correct join using the $Employment$ class, otherwise we cannot distinguish between the employment data of the father and mother. The different contexts of a join can be assigned an *alias* using the SQL keyword **AS**. This facilitates that different joins can be distinguish by a unique identifier, hence the correct SQL query of the example is shown below:

SELECT $Child.Subsidy, FE.Salary, ME.Duration, count(*)$ **FROM** $Child$
LEFT JOIN $Parent$ **AS** M **ON** $M.id$ = $Child.mother$
LEFT JOIN $Parent$ **AS** F **ON ON** $F.id$ = $Child.father$
LEFT JOIN $Employment$ **AS** ME **ON** $ME.id$ = $M.id$
LEFT JOIN $Employment$ **AS** FE **ON** $FE.id$ = $F.id$
GROUP BY $Child.Subsidy, FE.Salary, ME.Duration$

Now, the two joins on both $Parent$ and $EmployedAt$ are performed according to the correct context, and the attributes of the select statement are changed to $FE.Salary$ and $ME.Duration$. Still, we need to establish a naming convention for the aliases of the joins. We use $\mu(\rho)$ to denote the unique identifier associated with the slot ρ.

Definition 4 (Unique Slot Identifier, $\mu(\rho)$) *A slot $\rho_i \in \tau$ is uniquely identified by $\mu(\rho_i)$, which is the concatenation of all predecessor slots $\rho_j \in \tau$, i.e. the concatenation $\rho_0\rho_1 \dots \rho_{i-1}\rho_i$.*

For instance, $\mu(Mother.EmployedAt) = MotherEmployedAt$, or simply ME is our short notation in the above SQL query. If $|\tau| = 1$ then $\mu = \rho_0$. This facilities that the joins use the appropriate identifier and the correct context. However, we also need to know μ of a class $X_1, X_2, \dots, X_m$, such that the select and group-by statements match the joins.

Definition 5 (Unique Class Identifier, $\mu(X)$) *A class X_i where $X_i.A_i \in Pa(X.A)$ is uniquely identified by $\mu(X)$, which is the concatenation of all slots ρ in its slot chain τ, which is equal to $\mu(\rho_l)$ where $\rho_l \in \tau$ and $l = |\tau|$.*

Recall, that each parent/dependency is routed through a slot chain, hence $\mu(Employment).Salary = FatherEmployedAt.Salary$ or simply $FE.Salary$ in our short notation. The previous unique slot identifier, $\mu_{-1}(\rho_i)$, is defined as the unique identifier of the previous slot ρ_{i-1}, thus $\mu_{-1}(\rho_i) = \mu(\rho_{i-1})$. The previous unique identifier of the first slot ρ_0 in a slot chain τ is X, i.e. $\mu_{-1}(\rho_0) = X$.

Lastly, we have the final SQL query that considers internal and external dependencies while maintaining the context:

SELECT $X.A, \mu(X_1).A_1, \mu(X_2).A_2, \dots, \mu(X_n).A_n, count(*)$ **FROM** X
LEFT JOIN $Range(\rho_1)$ **AS** $\mu(\rho_1)$ **ON** $\mu(\rho_1).p_{key}(\rho_1)$ = $\mu_{-1}(\rho_1).f_{key}(\rho_1)$
LEFT JOIN $Range(\rho_2)$ **AS** $\mu(\rho_2)$ **ON** $\mu(\rho_2).p_{key}(\rho_2)$ = $\mu_{-1}(\rho_2).f_{key}(\rho_2)$
$\vdots \quad \vdots \quad \vdots \quad \vdots$
LEFT JOIN $Range(\rho_m)$ **AS** $\mu(\rho_n)$ **ON** $\mu(\rho_m).p_{key}(\rho_m)$ = $\mu_{-1}(\rho_m).f_{key}(\rho_m)$
GROUP BY $X.A, \mu(X_1).A_1, \dots, \mu(X_n).A_n$

This naming convention guarantees unique identifiers, hence it is trivial to see, that if we apply this naming convention to the example in Figure 4.6, we will get the correct SQL query.

Lastly, if slots in different slot chains are *equal* with regard to their context, only one should be included in ρ^*. For instance, if we have the two dependencies $Child.Mother.EmployedAt.Duration \rightarrow Child.Subsidy$ and $Child.Mother .EmployedAt.Salary \rightarrow Child.Subsidy$, then $\tau^*(Child.Subsidy) = \{Mother .EmployedAt, Mother.EmployedAt\}$ and $\rho^*(Child.Subsidy) = \{Mother, EmployedAt\}$, because of slot contextual equality.

Definition 6 (Slot Contextual Equality in ρ^*) *Two slots ρ_i and ρ_j is contextual equal if $\mu(\rho_i) = \mu(\rho_j)$.*

This facilitates that we do not perform unnecessary joins.

4.5.4 External Dependencies, Multi-Valued

In Section 4.5.2, we outlined the SQL query for external dependencies that are single-valued. Recall that we differ between two kinds of external dependencies, see Section 2.2.1. In the case of multi-valued external dependencies, $\gamma(X.\tau.B) \rightarrow X.A$, we need to apply the aggregation γ to the objects $\{y.B \mid y \in x.\tau\}$, while applying the concepts in Section 2.2.1 and 4.5.3. The aggregate functions are applied in the select and group by parts of the SQL statement (the joins are omitted, since they are unchanged):

$$\begin{array}{l}
\textbf{SELECT } X.A, \gamma(\mu(X_1).A_1), \ldots, \gamma(\mu(X_n).A_n), count(*) \textbf{ FROM } X \\
\vdots \qquad\qquad\qquad\qquad \vdots \qquad\qquad\qquad \vdots \\
\textbf{GROUP BY } X.A, \gamma(\mu(X_1).A_1), \ldots, \gamma(\mu(X_n).A_n)
\end{array}$$

Unfortunately, this SQL query is not valid, since it is not allowed to group by an aggregated function $\gamma(\mu(X_n).A_n)$. Consequently, such a query must be split into two queries by using a sub-query, a view or a temporary table. The first query computes the aggregate functions of the external multi-valued parents, the second query performs the joins and grouping, by using the aggregated values of the first query. Views seems to be the most straightforward solution, since we do not have to tackle complex embedded sub-queries. With views, only the joins need to be adjusted, such that they use the aggregated value in the view instead of the multi-valued entries in the original table. Basically the views collapse the multi-valued entries into an aggregated value.

The query that creates the view for Figure 4.8 is shown below:

$$\begin{array}{l}
\textbf{CREATE VIEW } v_1 \textbf{ AS} \\
\textbf{SELECT } parent\_id, avg(Duration) \textbf{ AS } avg\_dur, avg(Salary) \textbf{ AS } avg\_salary \\
\qquad\quad \textbf{FROM } Employment \\
\textbf{GROUP BY } parent\_id
\end{array}$$

The result of the view is shown in Table 4.9(a). When creating the views we must group by the foreign key $f_{key}(\rho)$ of the last slot ρ in the slot chain τ. In the example, it is $f_{key}(EmployedAt)$ in $Parent.EmployedAt$ which is $parent\_id$.

id	**parent_id**	**subsidy**
1	1	ES
2	2	ES
3	3	DS

(a) Child Table

id	**age**	**sex**
1	45	M
2	50	M
3	30	F

(b) Parent Table

id	**parent_id**	**years**	**salary**
1	1	10	40000
2	1	10	20000
3	2	10	30000

(c) Employment Table

Figure 4.7: (a) The child table has one slot $Child.Parent$, (b) the parent table has two inverse slots $Parent.ChildOf$ and $Parent.EmployedAt$, (c) the employment table has one slot $Employment.Employed$.

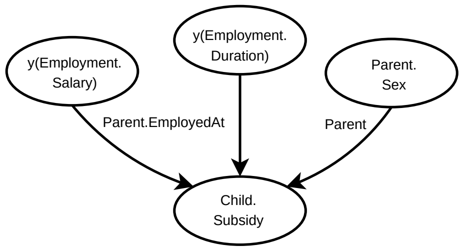


Figure 4.8: The parents of $Child.Subsidy$. Two multi-valued dependencies $\gamma(Employment.Salary)$ and $\gamma(Employment.Duration)$ both through slot chain $Parent.EmployedAt$. $Parent.Sex$ is a single-valued dependency through slot $Parent$. These slots are reflected in the primary and foreign keys of the tables in Figure 4.7.

If the multi-valued dependencies are not from the same class, we would need additional views (one for each class).

The SQL query that computes the sufficient statistics is as in Section 2.2.1, however the views are applied instead of the original tables of the multi-valued parents, see the result in Table 4.9(b):

SELECT $Child.Subsidy, PE.avg\_years, PE.avg\_salary$
    $P.Sex, count(*)$ **AS** $Count$ **FROM** $Child$
LEFT JOIN $Parent$ **AS** P **ON** $P.id = Child.parent\_id$
LEFT JOIN $v1$ **AS** PE **ON** $PE.parent = P.id$
GROUP BY $Child.Subsidy, PE.avg\_emp, PE.avg\_sal, P.Sex$

parent_id	avg_yea	avg_sal
1	10	$30,000$
2	10	30.000

(a) The Aggregated View

Child.Subsidy	PE.avg_yea	PE.avg_sal	P.Sex	Count
ES	10	30.000	*M*	2
DS	*NULL*	*NULL*	*F*	1

(b) $C_{Child.Subsidy}$

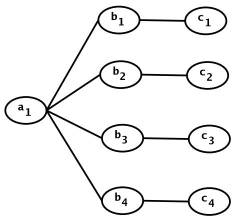
Figure 4.9: (a) The view containing the aggregated values of the tables in Figure 4.7, (b) the result of the count $C_{Child.Subsidy}$.

An interesting aspect is how objects in a multi-valued slot chain are counted. Figure 4.10 depicts two different scenarios; Figure 4.10(a) where four objects point to four different objects, and Figure 4.10(b) where four objects point to the same object. In the first case, it is clear that we count the objects reached by following the slot chain as distinct objects, thus the count is four. However, in the second case it is not clear whether we get a count of one or four. Indeed the count is four, since the SQL query returns a *bag* of objects, thus we get four "copies" of the same object although we are not aware of this. Actually, this is quite reasonable since the four objects have one parent each, which just happen to be the same object.

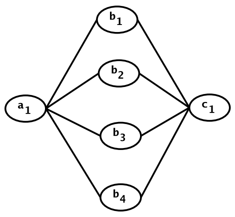
4.6 Calculating the Probabilities

In the previous section the joins which produce the sufficient statistics were described, however we still need to specify how to calculate the probabilities in a CPT. The result of executing a query is shown in Table 4.0(a), and is stored in main memory. The number of rows in the result from the query, is the cross product of $V(X.A) \times V(X_1.A_1) \times V(X_2.A_2) \times \ldots \times V(X_n.A_n)$.

We also need to calculate the frequency of the joint values that the parents



(a) Different Objects



(b) Same Object

Figure 4.10: The nodes represent objects and the edges represent relations between objects. (a) The objects b_1, b_2, b_3 and b_4 point to four different c objects, (b) the objects b_1, b_2, b_3 and b_4 point to one object c_1.

(a) Result of Executing a Query

$X.A$	$X_1.A_1$	$X_2.A_2$	$\cdots$	$X_n.A_n$	$Count$
$\ldots$	$\ldots$	$\ldots$	$\ldots$	$\ldots$	$\ldots$

(b) Calculating the Sum

Child.Subsidy	*Parent.Married*	*Parent.Citizenship*	*Child.Age*	*Count*	*Sum*
ES	*MA*	*Denmark*	3	6	10
ES	*WI*	*Sri Lanka*	6	3	3
RE	*MA*	*Denmark*	3	4	10

Table 4.1: Result of Executing a Query

can take. This is equal to the denominator in Equation 3.2 on page 22. Since our table is already grouped as a result of the join, we simply sum the *count* column where the rows of $X_1.A_1$ to $X_n.A_n$ are equal. This value is stored in a new column called *sum*. An example is shown in Table 4.1.

The probability is computed in each row by using the two columns *count* and *sum*, $P(X.A \mid Pa(X.A)) = \frac{count}{sum}$.

Chapter 5

Implementation

Our design has been implemented on the .NET platform using the language C# with 4 327 lines of code. A screenshot of the application is shown in Figure 5.1.

The next sections shows how to specify a database schema and starting the learning procedure, using our implementation.

5.1 Specifying a Database Schema

A subset of the code that specifies the schema for the movie database is shown in Listing 5.1. The slots and their cardinality have to be entered explicitly. This also implies specifying the cardinality of inverse slots.

Listing 5.1: Specifying the Database Schema

```
Prm p = new Prm("imdb");

Class movie = new Class("movies",p);
Class director = new Class("director",p);
Class director = new Class("studio",p);

FixedAtt primaryDirectorID = new FixedAtt("directorid",director,p);
FixedAtt primaryStudioID = new FixedAtt("name",studio,p);
FixedAtt primaryMovieID = new FixedAtt("filmid",movie,p);

FixedAtt foreignDirectorID = new FixedAtt("dirid",movie,p);
FixedAtt foreignStudioID = new FixedAtt("studio",movie,p);

DescAtt m_year = new DescAtt("year",movie,RangeType.Continuous);
DescAtt m_award = new DescAtt("awtype",movie,RangeType.Discrete);

Slot slot_director = movie.AddSlot("Director",primaryDirectorID, foreignDirectorID,
                                      Cardinality.OneToOne, Cardinality.OneToMany);

Slot slot_studio = movie.AddSlot("Studio",primaryStudioID,foreignStudioID,
                                    Cardinality.OneToOne, Cardinality.OneToMany);
```

The method *Class.AddSlot* is shown in Listing 5.2.

Listing 5.2: Creating ASlot

```
public Slot AddSlot(string name, FixedAtt primaryKey, FixedAtt foreignKey,
                    Cardinality car, Cardinality inverseCar)
{
   Slot slot = new Slot(name, primaryKey, foreignKey, car, inverseCar);
   slot.CreateInverse();

   foreignKey.Class.Slots[slot.Name] = slot;
   primaryKey.Class.Slots[slot.Inverse.Name] = slot.Inverse;

   return slot;
}
```

A new slot is created by assigning a name, a primary key, a foreign key, the cardinality, and the inverse cardinality. The slot is created on the class of the foreign key. Once a slot has been created, we automatically create the inverse slot, on the class which contains the primary key, see Listings 5.3.

Listing 5.3: Creating An Inverse Slot

```
public void CreateInverse()
{
  inverse = new Slot(name + "Of", to, from);
  inverse.Inverse = this;
  inverse.isInverse = true;
  inverse.Cardinality = this.tmpCardinality;
}
```

Notice, that we add the postfix *Of* to inverse slots, and also that ${\rho^{-1}}^{-1} = \rho$, since we set *inverse.Inverse* to the current object.

5.2 The Learning Procedure

Once the database schema has been established, the learning procedure can be initiated, see Listing 5.4.

Listing 5.4: The Search Procedure

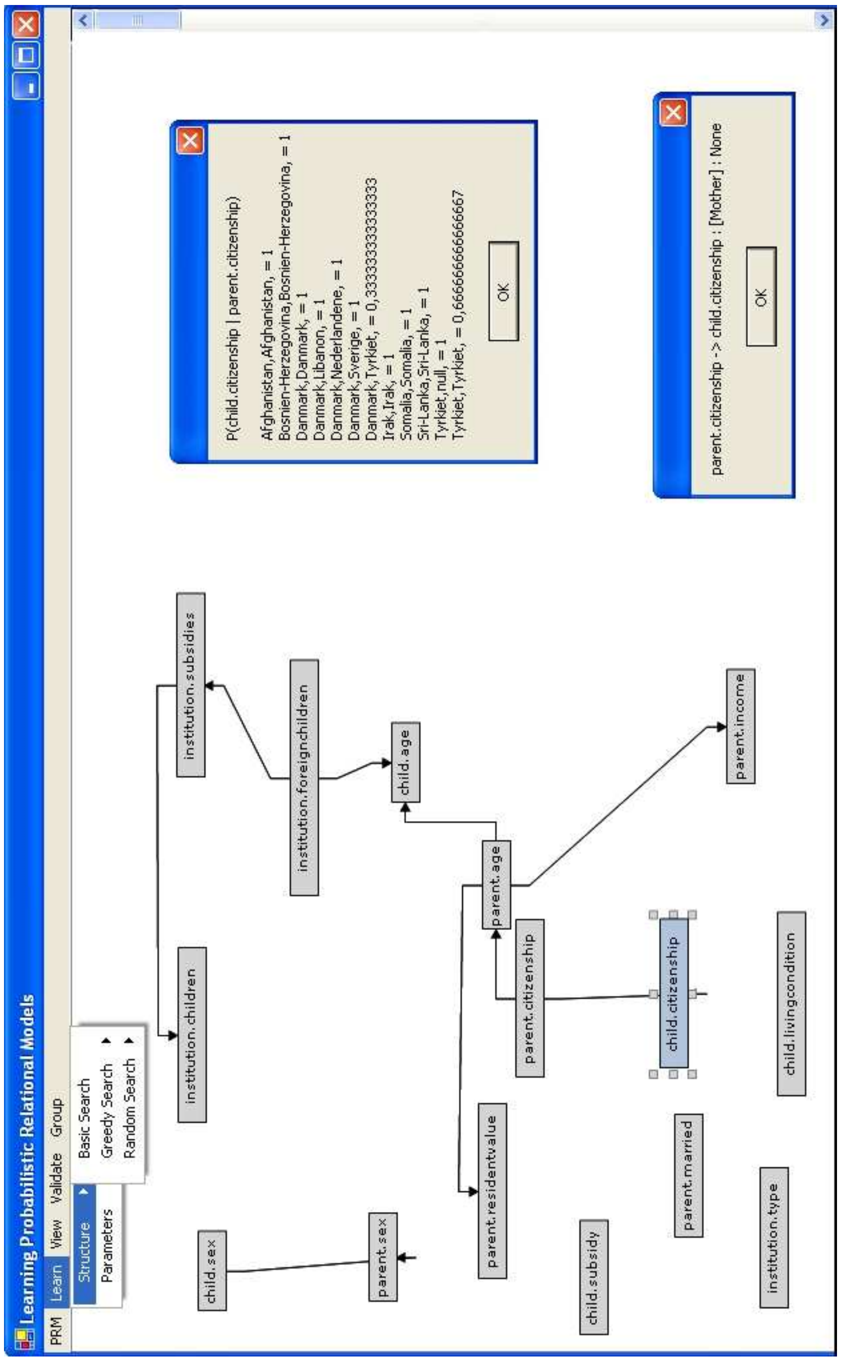
```
Prm p = Schema.IMDB();

StructureLearner structure = p.StructureLearner;
structure.Method = SearchMethod.Greedy;

Search search = structure.Search;
search.Mult = 3;

structure.Learn();
```

First, a schema is chosen and the appropriate settings of the search, e.g. the penalty multiplier is assigned in $search.Mult = 3$. $structure.Learn()$ initiates the search for a dependency structure.



```
P(child.citizenship | parent.citizenship)

Afghanistan,Afghanistan, = 1
Bosnien-Herzegovina,Bosnien-Herzegovina, = 1
Danmark,Danmark, = 1
Danmark,Libanon, = 1
Danmark,Nederlandene, = 1
Danmark,Sverige, = 1
Danmark,Tyrkiet, = 0,3333333333333333
Irak,Irak, = 1
Somalia,Somalia, = 1
Sri-Lanka,Sri-Lanka, = 1
Tyrkiet,null, = 1
Tyrkiet,Tyrkiet, = 0,6666666666666667
```

```
parent.citizenship -> child.citizenship : [Mother] : None
```

Figure 5.1: A screenshot of our application. The box to the right shows the CPT for the attribute Child.Citizenship. The slot chain is also shown, which indicates that the dependency uses the Mother slot. Notice that this screenshot is from a small sample database, otherwise the CPT would be too large.

Chapter 6

Databases

6.1 Relational Daycare Database

This database originates from daycare admission of children, in Herning Kommune, Denmark. The database contains information about children, their parents and which institution the children attend. The database was the subject of data mining in our earlier work [10].This database is a revised version of the database presented, see [10] for further information.

The following section briefly introduces the domain and background for the data.

6.1.1 Daycare Admission of Children

Herning Kommune, along with other municipalities in Denmark, coordinates a number of institutions that provide daycare activities for children e.g. nursery, kindergarten etc. These institutions offer a number of available seats for children, which parents can apply for. Once an application has been processed, the child either receives a seat at an institution or is placed on a waiting list if the municipality has capacity problems.

A seat costs a monthly fee, which is payed by the parents. This type of seat is known as a regular seat, **RE**. There exist subsidies for these regular seats, such that the municipality pays a percentage of the parents monthly institution fee. In the following we describe the different kinds of subsidies given by the municipality.

Economical Subsidy (ES): Parents with a low income is eligible for an economical subsidy. The assessment is based on income intervals, with a corresponding percentage price cut.

Social Subsidy (SS): Parents with children who have social problems, due to difficulties at home or at school, may be eligible for a social subsidy. The assessment is based on a subjective evaluation by a social worker.

Disability Subsidy (DS): Parents with disabled children is eligible for a disability subsidy. The assessment is based on documentation of the child's disability.

Children Subsidy (CS): Parent with more than two children attending daycare is eligible for a children subsidy. The subsidy is given automatically.

In the following section a brief documentation of database schema is given. Notice, the *Employment* table is missing, since this only is included in order to address important theoretical issues.

6.1.2 Database Schema

Figure 6.1 shows the schema and keys of the relational daycare database.

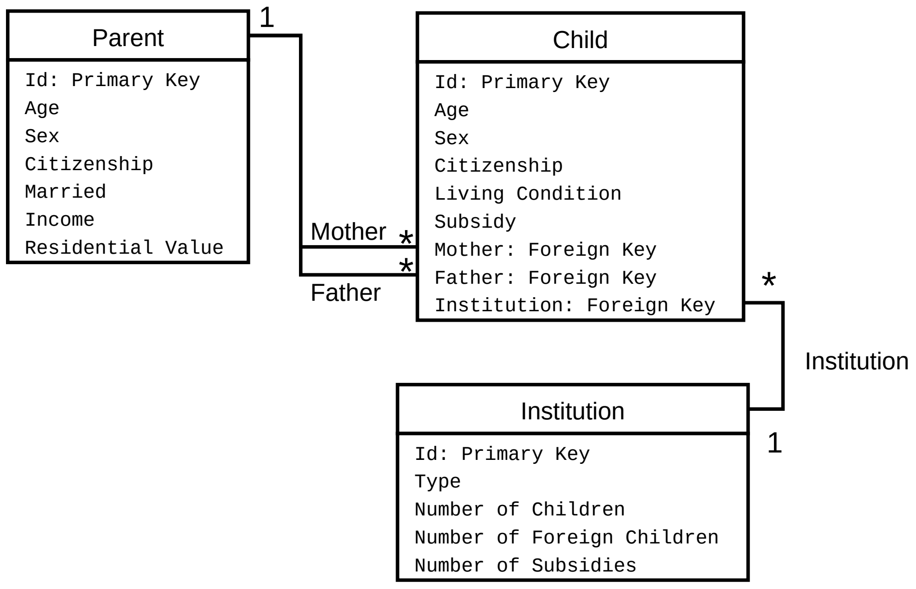


Figure 6.1: Schema for the daycare database

Child Table

The child table contains three slots, five attributes, and has 14 295 rows.

Slots:

Mother: The mother of the child (one-to-one).

Father: The father of the child (one-to-one).

Institution: The institution of the child (one-to-one).

Attributes:

Age: The age of the child (numeric).

Sex: The sex of the child (nominal, binary).

Citizenship: The citizenship of the child e.g. Denmark, Sweden etc. (nominal, 48 states).

LivingCondition: The living condition of the child e.g. livening together, only with father, only with mother etc. (nominal, 6 states).

Subsidy: The type of subsidy granted e.g. no subsidy, social subsidy, see Section 6.1.1 (nominal, 6 states).

Parent Table

The parent table has five attributes, two slots, and 15 249 rows.

Slots:

MotherOf: The children of the mother (one-to-many).

FatherOf: The children of the father (one-to-many).

Attributes:

Age: The age of the parent (numeric).

Citizenship: The citizenship of the parent e.g. Denmark, Sweden etc. (nominal, 68 states).

Married: The marital status of the parent e.g. Married, Widow etc. (nominal, 5 states).

Income: The average income of the parent (numeric).

ResidentialValue: The Residential value of the parent (numeric).

Institution Table

The institution table has three attributes, one slot and 69 rows.

Slots:

InstitutionOf: The children of the institution (one-to-many).

Attributes:

Type: The type of institution e.g. kindergarten, daycare, integrated etc. (nominal, 7 states).

Children: The number of children in the institution (numeric).

ForeignChildren: The number of foreign children in the institution (numeric).

Subsidies: The number of subsidies granted in the institution (numeric).

6.2 Relational Movie Database

This database contains information about movies and their relations. The main table is *movies* and related data such as information about *studios*, *actors* and *directors* can be reached by following relations from the movie table. The schema is shown in Figure 6.2.

This relational database from UCI Knowledge Discovery in Databases (KDD)[1] has been well studied for several different tasks, such as: classification, regression and clustering. David Jensen and Jennifer Neville gives a survey of classification using this database [17].

6.2.1 Converting to a Useful Relational Schema

Unfortunately the condition of the database from the website is not a standard file that can be downloaded, but expressed in XML and HTML language. This means that a conversion from HTML or XML to relational database form is needed. However the information about the database structure from the UCI website can easily be misinterpreted. This results in difference databases when made by different people, so a directly comparison is difficult.

We have followed the original description of the database given by Gio Wiederhold [31] in order to construct the most correct relational database. However there are some flaws in the documentation, with regard to types and duplicates in data tables and issues that easily be questioned. We have cleared the database of any duplicates, such it now only contains non-redundant information.

Several improvements has been made in order to correct the design, this includes:

Primary keys: No duplicates are allowed. The XML and HTML pages are not always consistent with regard to this restriction.

Indexes: Indexes are made on all foreign keys such that any join on these columns is improved.

Belows is only shown the attributes that we use in our setting,although the database contains much more information about movies, actors and directors. Information such as movie title, actors first name, actors family name etc. are omitted.

For a full description please see [31].

Director Table

The director table has one slot, seven attributes, and 3 471 rows.

Slots:

$DirectorOf$**:** The movies of the director (one-to-many).

Attributes:

Pcode: Indicates which type of director e.g. director, producer, writer etc. (nominal, 127 states).

CareerStart: Year the director started (numeric).

DirectorStart: Year the director produced the first movie (numeric).

CareerEnd: Year the director stopped (numeric).

DateOfBirth: Year of birth (numeric).

DateofDeath: Year of death (numeric).

Background: The country the director comes from e.g. USA, England etc. (nominal, 88 states).

Movie Table

The movie table has three slots, three attributes, and 12 046 rows.

Slots:

Director: The director of the movie (one-to-one).

Studio: The studio that produced the movie (one-to-one).

MovieOf: The cast of the movie (one-to-many).

Attributes:

Year: Year the film was produced (numeric).

AwardType: Type of award e.g. Oscar (nominal, 42 states).

Author : *michael*: The author of the manuscript (nominal, 1060 states).

Studio Table

The studio table has five attributes, one slot, and 186 rows.

Slots:

StudioOf: The name of the studio (one-to-many).

Attributes:

City: The city in which the studio resides e.g. New York, London etc. (nominal, 56 states).

Country: The country e.g. USA, England etc. (nominal, 20 states).

Founder: The name of the founder of the studio (nominal, 27 states).

FirstYear: The first year a film was made in this studio (numeric).

LastYear: The year for the recent film produced in this studio (numeric).

Casts Table

The casts table has two slots, only one attribute, and 11 731 rows. The table serves as a many-to-many relation between the movie and actor table.

Slots:

Movie: The movie of the cast (one-to-one).

Actor: The actor of the cast (one-to-one).

Attribute:

Genre: The genre of the movie e.g action, comedy, etc. (nominal, 45 states).

Actor Table

The actor table has one slot, nine attributes, and 6 714 rows.

Slots:

ActorOf: The cast of the actor (one-to-many).

Attribute:

Start: The year the actor started performing (numeric).

End: The year the actor ended acting (numeric).

Sex: The sex of the actor (nominal, binary).

DateOfBirth: Year the actor was born (numeric).

DateOfDeath: Year the actor died (numeric).

Origin: The citizenship of the actor (numeric).

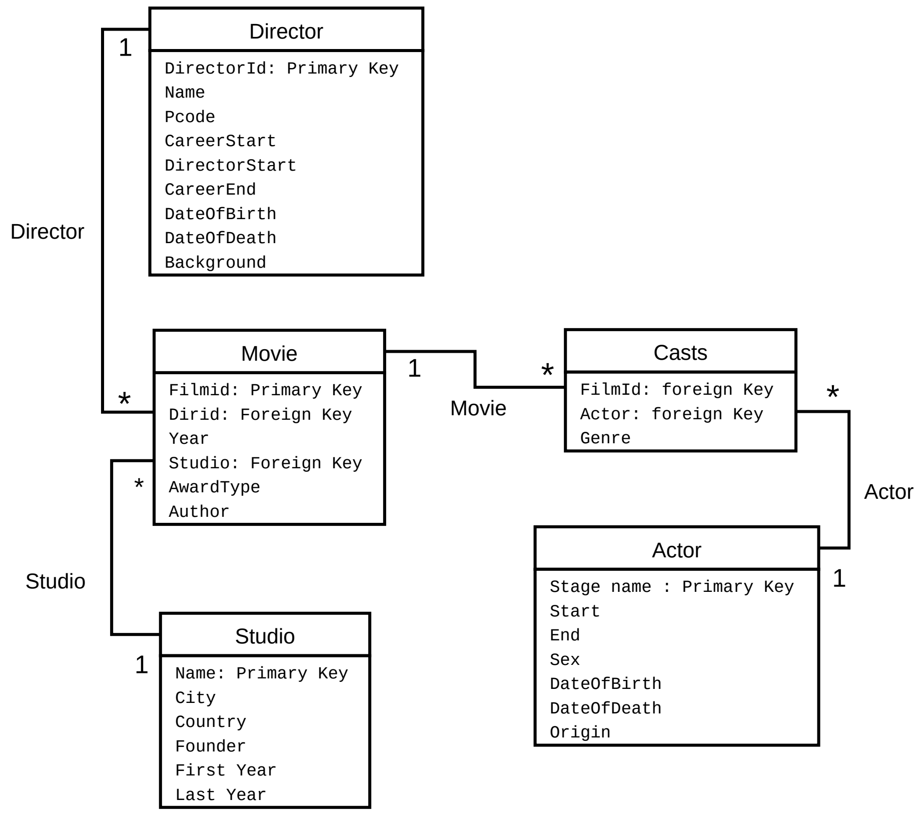


Figure 6.2: Schema for the movie database

Chapter 7

Tests

This chapter describes the various experimental tests that were applied to our implementation.

An important note, is that the running time of tests *across sections* are not comparable, e.g. Section 7.1 and Section 7.2, since they were run on different machines.

7.1 Complexity Penalty Test

The penalty term, $d_k\ log\ n$, of the score function in Eq. 3.3, can be adjusted by inserting a multiplier factor α, such that the penalty term becomes $\alpha\ d_k\ log\ n$. In this test, we will increase the value of α from its default value of 1 to 10, and see how it affects the search procedure (assuming a greedy search approach). The expectation is that increasing α will yield:

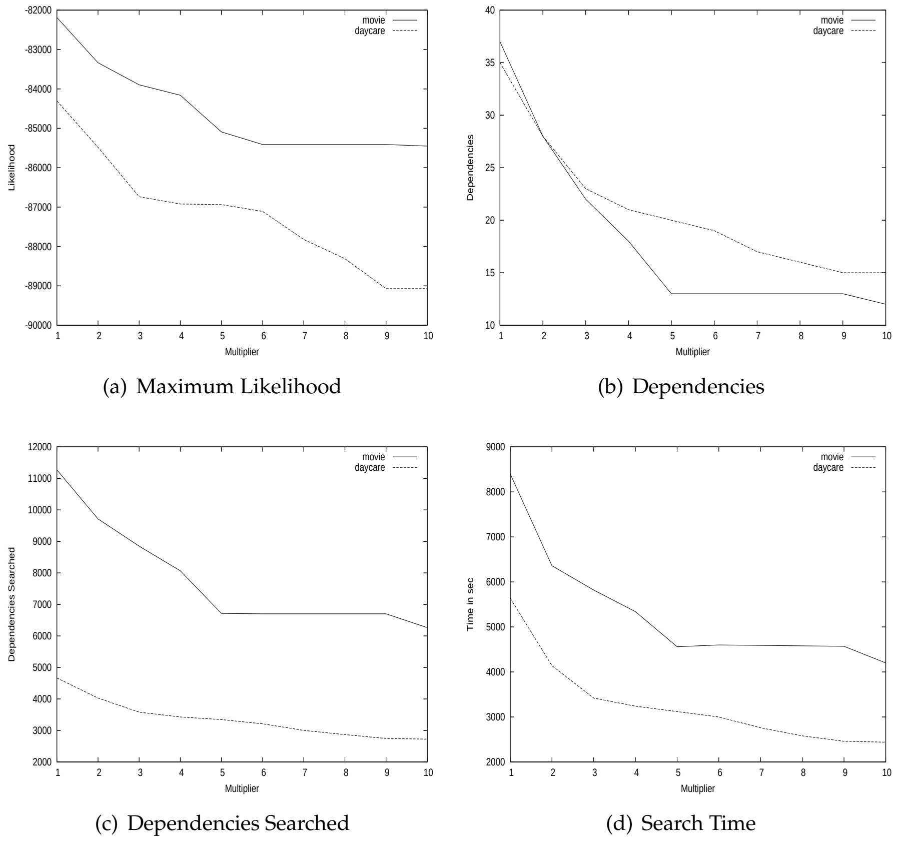
- Fewer dependencies in the final structure.
- Fewer dependencies searched, thus faster execution time.

As a consequence, the maximum likelihood also decreases as α is increased.

7.1.1 Results and Evaluation

Figure 7.1 shows the test results with the multiplier α paired with the likelihood, number of dependencies, number of dependencies searched, and the execution time of the search. The results show that with the daycare database, we get a lower maximum likelihood, fewer dependencies, and lower search time as the multiplier increases—as expected. However, with the movie database the results are, basically, equal from $\alpha = 5$ and $\alpha = 9$, which indicates that all dependencies are so good in terms of improving the likelihood score, that none can be removed before $\alpha = 10$.

The results clearly show that it is a trade off between *fewer dependencies and lower maximum likelihood*. Figure 7.1(a) and Figure 7.1(b) show that we can give up a relative little amount of maximum likelihood, for significantly fewer dependencies. More specifically, we can give up about 3% of the maximum likelihood for about 34% fewer dependencies, between $\alpha = 1$ and $\alpha = 3$. Although, care should be taken when comparing absolute and relative numbers of maximum likelihood, since it is not as comprehensible as e.g. the number of dependencies.



(a) Maximum Likelihood (b) Dependencies

(c) Dependencies Searched (d) Search Time

Figure 7.1: The penalty multiplier α and (a) the maximum likelihood, (b) the number of dependencies in the structure, (c) the number of total dependencies searched, (d) and the execution time of the search.

Two overlay structures of the 10 different runs on the daycare and the movie database, are shown in Figure 7.2 and Figure 7.3, respectively. The thickness of the dependencies represent the threshold to which α had to be increased to, before that dependency was removed due to penalization. Hence the thickness can be interpreted as significance. We will evaluate the dependencies of the structures in Section 7.5.

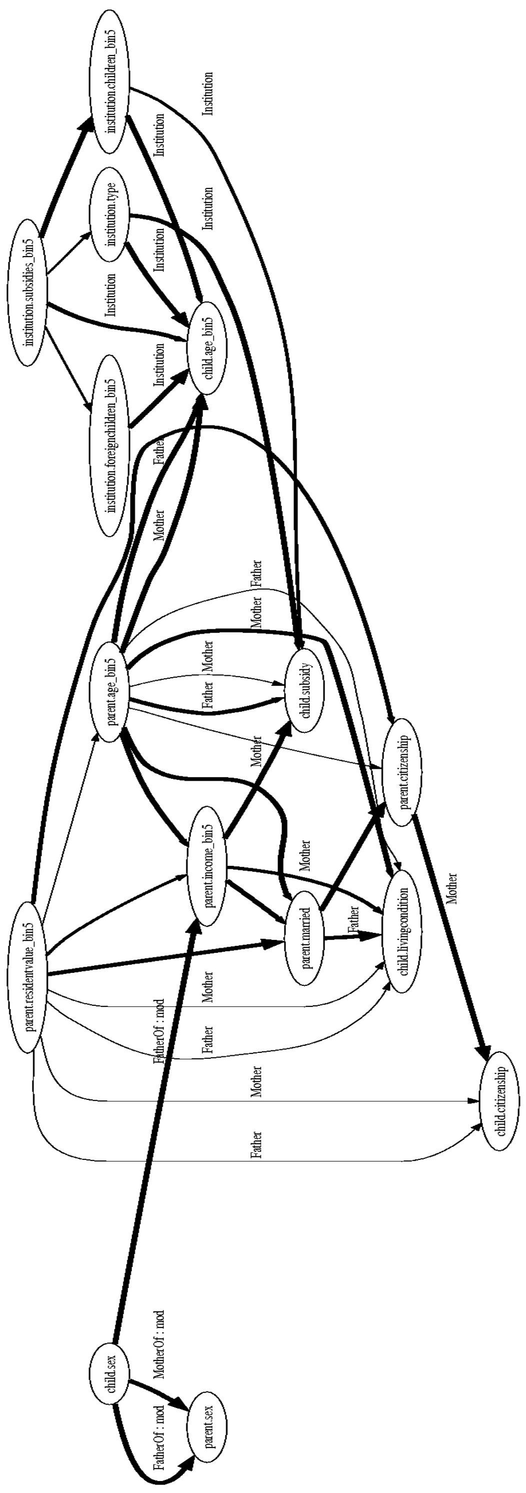


Figure 7.2: The overlay structure for the daycare database.

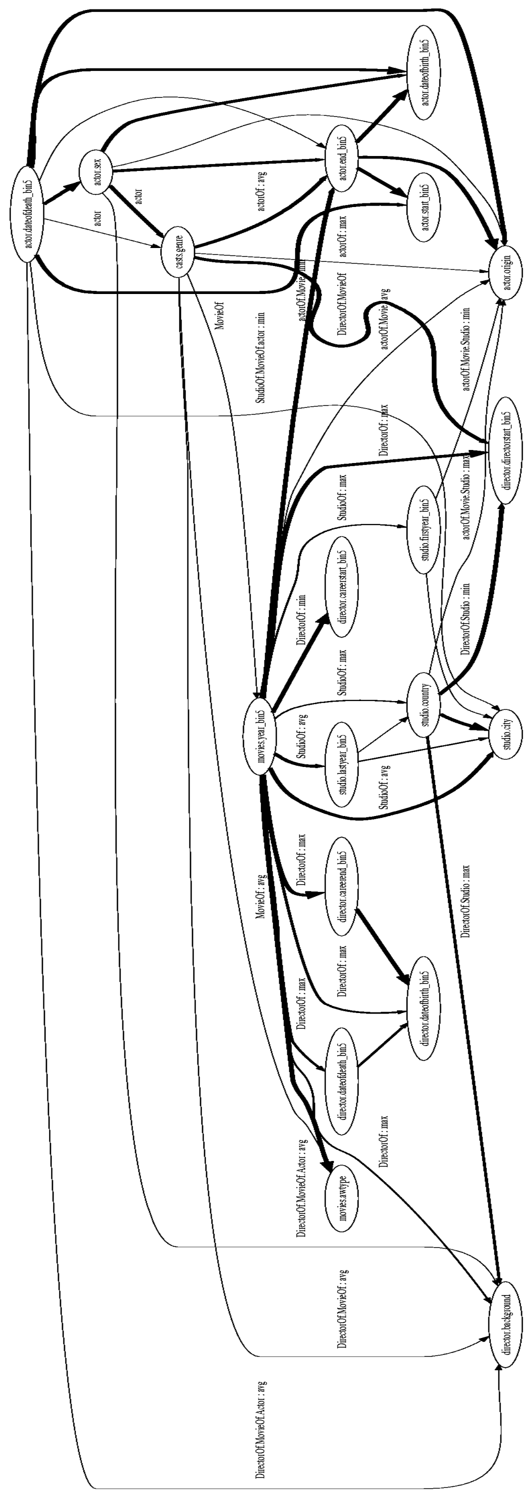


Figure 7.3: The overlay structure for the movie database.

Consequently, α is a useful parameter in controlling the search. Although, there is no all-around good value of α; different values have to be tried in order to get a model that is satisfiable in terms of complexity, execution time, and data fit.

7.2 Search Test

The choice of search algorithm for exploring the hypothesis space of legal models, has a great impact on the structure of our final model. Different algorithms explore the space in different manners. In order to see the effect of the selected search algorithm for exploring our hypothesis space, the daycare database has been tested using the following algorithms, see Section 4.4 on page 32:

- Greedy
- Subset Search
- Normalized Random
- Hybrid

The movie database has been tested using greedy and hybrid, in order to see if any major difference in search behavior occurs.

The following sections show the results given the different search algorithms. Notice, all tests have been conducted with $\alpha = 1.0$.

7.2.1 Greedy Search Result

The result of the greedy search is shown in Figure 7.4. It shows that large score improvements are made in the beginning, but later on the improvements deteriorates. This is as expected, hence as time goes we get a more and more complex model, because more dependencies are added and the penalty becomes larger.

The time taken for learning the structure with greedy search is approximately 2 900 seconds for the daycare database and 7 800 seconds for the movie database. These tests were conducted using an Intel Pentium 4, 2.6 GHZ processor with 512 MB of RAM.

However we do not know if the models found are the optimal ones in the hypothesis space of legal structures, since the most greedy step in each iteration does not automatically lead to an optimal structure. Therefore several randomized algorithms have been proposed in Section 4.4 in order to cope with the problem of local maximas. We expect that some of the most randomized algorithms are much faster than greedy, even though greedy is reasonable fast. Other algorithms are slower, but we focus on increment the structure score as opposed to improving running time. However, the latter we still wish to keep as low as possible. These tests have only been conducted using the daycare database.

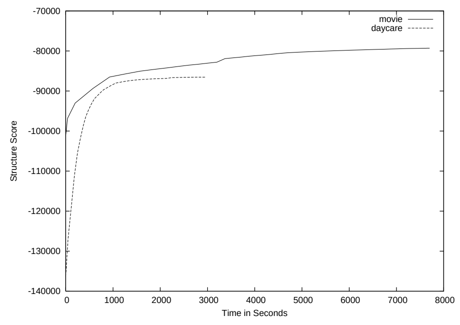


Figure 7.4: Greedy search using a multiplier of 1.0. On the horizontal axis is the time in seconds and on the vertical axis is the structure score.

7.2.2 Random Search

Since randomness is a factor in these tests, they are restarted several times. Two different random algorithms are tested; the random subset and normalized random algorithms as described in Section 4.4.2.

In these tests we should expect the normalized algorithm to do much better than the subset random one, since there is a higher chance of selecting the best dependencies using the normalized algorithm. On the other hand, the running time of the subset search should be much faster than the normalized one, since much less computation is needed.

Subset Random Result

The search has been restarted 5 times and the result is shown in Figure 7.5(a). The random subset approach results in large score improvements very fast, but unfortunately the search stalls very quickly and from that point in time only small improvements are made, if any at all. We can see in Figure 7.5(a), that in *run 3* the algorithm terminates quite fast, and does not seem to try different subsets as intended. This is because when the subset is selected, all possible dependencies are considered and one is chosen. Then a check is made in or to see if the dependency introduce cycles. If not, it is include in the set, otherwise it is dropped. As the search continues the probability of selecting dependencies that are illegal increases, and the subset of useful dependencies decreases. If no dependency within the subset is legal the search is terminated. This e.g. happened in *run 3*. We could make the acyclic check before the dependency is considered to be included in the subset. Then it must be reconsidered when the search should terminate, since the risk for a large increase in time is present, if we are to try all possible dependencies by selecting them randomly among all dependencies.

The subset search algorithm performs very badly compared to greedy search in terms of the final structure score. So this very fast approach is not further pursued.

Normalized Random Result

The result using this search algorithm is shown in Figure 7.5(b). Here we see, as expected, a much better structure score is obtained. It is actually almost as good as the greedy search. However we would like to get a better structure score in order to see if a better approach than greedy exists, and disregard the increment in running time.

The results could suffer from the second statement in Section 4.4.2, where the normalization screws the probability of selecting very good dependencies, if many possible dependencies are present. However, preliminary tests have shown that even though we only considering the 20 best dependencies before normalizing it does not give any better results. The only difference is that the slope in the beginning increases more, but the best structure score does not improve and does not become better than greedy.

An improvement, which could be pursued, is to allow dependencies that decreases the structure score. It could be done like the k tries of m steps in the subset random algorithm. This has though not been explored.

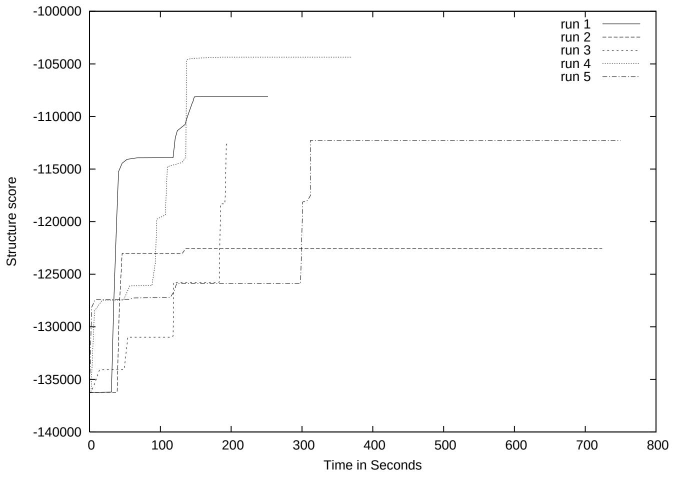
7.2.3 Hybrid Search Result

When applying the hybrid search algorithm we actually get quite good results when speaking in terms of both speed and structure score. This holds for both the daycare and movie database. The search trace for 5 runs is shown in Figure 7.6. Here we see that in the beginning random steps are taken, and the structure score improves rapidly, but as time progresses they all, more or less, take the best step all the time and the improvements deteriorates. The final structure score is very similar in all test runs. However on the movie database, in Figure 7.6(b), we see a larger difference between the runs.

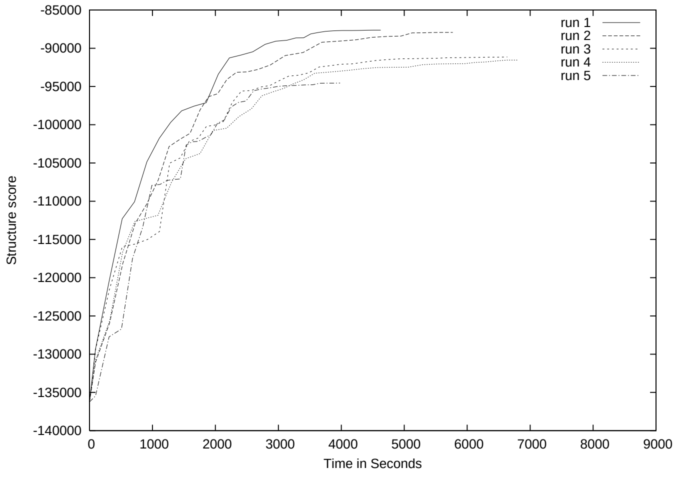
It is worth noticing that the running time of this search algorithm is slightly better than both greedy search and normalized random, the latter with a factor of two. The reason is straight forward; by introducing random steps, fewer calculations are needed.

7.2.4 Evaluation

Figure 7.7 shows a comparison of the four search algorithms on the daycare database. The best test run for the three random algorithms; subset, normalized and hybrid is shown together with greedy search.

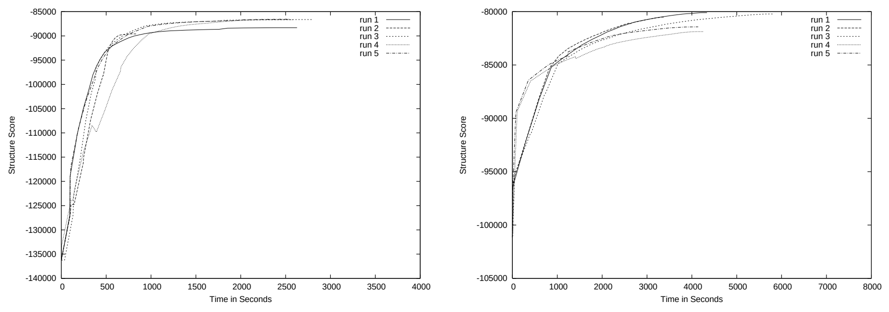


(a) Subset Random Search on the daycare database



(b) Normalized Random Search on the daycare database

Figure 7.5: (a) On the vertical axis is the final structure score and on the horizontal axis the running time in seconds. It is easy to see that the best dependencies are only selected in some, but far from all, cases.(b) The axis' are the same as in a). By using this normalized algorithm we get a better search, in terms of structure score, than using the random subset approach.



(a) Hybrid search on the daycare database (b) Hybrid search on the movie database

Figure 7.6: (a) The hybrid search on the daycare database. All runs are almost identical with respect to score and running time.(b) The hybrid search on the movie database. There is a larger differences between the runs than in the daycare database.

The tests shows that no algorithm finds a better structure than greedy does, but the hybrid get the same structure score in less time. The normalized algorithm takes twice the time for an almost identical score.

In Figure 7.8, greedy and the hybrid search have been run on both databases. As mentioned earlier there is not much difference between hybrid and greedy on the daycare database. However on the movie database the hybrid search find an almost identical score to greedy in half the time. So on this database hybrid is very superior in speed, but does not find a better structure than greedy. However it must be mentioned that the standard deviations from greedy are larger on the movie database than the daycare database, so more tests are needed in order to ensure consistency.

So according to our tests, greedy performs best in terms of structure score. Hence, greedy is in our setting a reasonable choice, but other approaches could be tried, which may yield a better score, although with a much higher running time. Different algorithms could be tried as e.g Best First Search which is similar to hill climbing, but is exhaustive in its search, so it would eventually try all possibilities.

In order to get more different search results between greedy and hybrid in our tests, the variables deciding the degree of randomness can be tweaked in the hybrid approach, see Listing 4.4 on page 36. This tweaking of the variables could result in a structure with score improvement and could be guided by the approach mentioned in [26], where a constant k is used to determine the degree of randomness. Results from this paper shows that a degree of randomness outperforms, in terms of score, the greedy approach in 66% of all cases. However we have not experienced similar results.

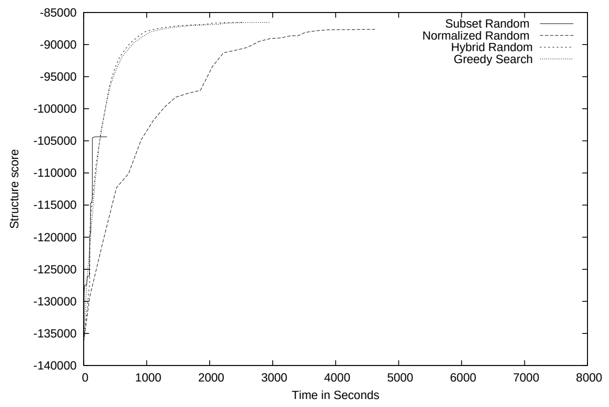


Figure 7.7: Comparison of four search algorithms. There is no search algorithm that performs better in score than greedy, but notice that the hybrid approach get accurate the same score as the greedy algorithm in less time.

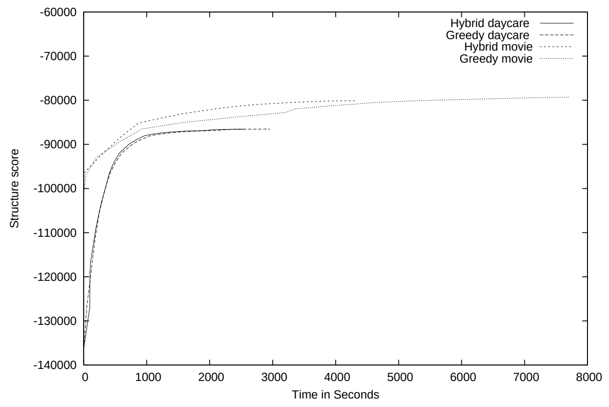


Figure 7.8: Greedy and random search on the two different databases. On the daycare database more equal results are obtained, whereas in the movie domain the difference is more noticeable.

7.3 Scalability Test

The scalability of the search for the best structure is of great importance when working on large databases. Therefore we test the scalability of our design and implementation. *The scalability of our implementation* is measured in time taken for learning the structure that fits the data best. Usually we can create several sample sets of different sizes in order to measure the scalability of a model, by simply notice the increment in running time. This is often done in propositional data mining. Similar sample sets can be made from a relational database, using a slightly different approach. By selecting instances from one table and following all relations we get a subset of the relations contained in the real database. However this is not always quite so easy. We must accept some flaws such as sample incorrectness and omitting related data.

1. **Sample Incorrectness**
   When taking a small subset of children from our daycare database, we statistically also want a small subset of the institutions. But because of the large number of children compared to the number of institutions, we get a screwed subset when following the relation from child to institution. Suppose we want 10% of the children, then there is a great chance of getting more than 10% of the institutions. It is more likely to get 90% of the institutions.

2. **Omitting Related Data**
   We want to create small data samples, which contain a fixed number, k, of children from our daycare database. Besides determining the value of k, we must determine when to omit some relations

   In our samples, we take k children and following the relation to $mother, father$ and $institution$. However we not guaranteed that siblings to specific children are automatically included in the sample set, which amounts to omitting related data.

   The same happens in the movie database, when following all relations from $movies$.

7.3.1 Test and Evaluation

Keeping these obstacles in mind, we have created subsets of the daycare and the movie database, in order to evaluate the scalability of our implementation. The sample sizes are 3 000, 6 000, 9 000, 12 000 and 15 000. The result is shown in Figure 7.9. The figure shows that the scalability of our implementation is almost linear in the number of training examples when looking at the daycare database. A small increment in the running time is noticeable as the sample size increases.

However when looking at the movie database we see a quite different result. Here there is a large increase in running time from the two first samples, and from the second to the third an improvement in running time is made. This could indicate that our samples have not been constructed with equal relations. The improvement in running time could be because there are many useful relations in the additional 3000 samples when going from 6000 to 9000 samples such that the search finds an optimal model more quickly. The difference in time taken for the same sample size on the two database can not directly be compared, since it is on two different databases. However it could indicate that there are many multi-valued slot chains, and longer chains in the movie database.

These results show, as expected, that the approach shown in Figure 7.9 is very limited. Hence things, as mentioned above, have an impact on the running time and hence must be investigated.

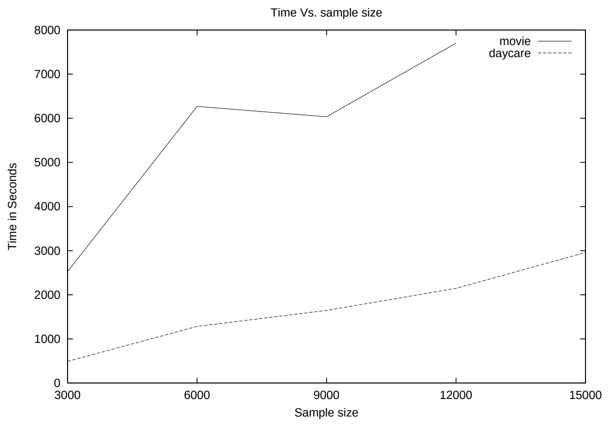


Figure 7.9: A very general scalability tests of our model.

- **Slot Chain Length**
  The maximum length of slot chains that can be constructed may have a high impact on the scalability of our implementation. That is, when retrieving data following a slot chain τ with k different classes, we need to do k joins in order to retrieve the correct information. This is computational heavy, and the impact of $|\tau|$ has to be studied further.

- **Evaluating Queries**
  When working on data that contain multi-valued slot chains the running time increases, since many objects have to be aggregated. A problem is, like in Bayesian networks, that an descriptive attribute can have many parents. If all attributes are good indicates for the same attribute, except the attribute itself, the running time would increase significantly, especially if multi-valued slot chains are present.

7.4 Profile Test

The purpose of the profile test is to uncover potential performance bottlenecks, i.e., which components that slow down the overall performance while learning PRMs. The profiler [21] shows the total percentage of time used *in* and *under* a method. It does, however, not record the time since profiling can add a factor of 20 to the running time.

Our exceptation is that the majority of the time will be spend on querying the database.

7.4.1 Result and Evaluation

Figure 7.10 and Figure 7.11 show the result of profiling the structure learning procedure with greedy search. The results are depicted as call graphs, with method name and percentage. Figure 7.10 shows the call graph for the daycare database, whereas Figure 7.11 shows the call graph for the movie database.

The figures show three interesting results:

- The majority of the total time, about 99% are used in scoring the candidate models (learning the parameters), as expected.
- About $86 - 89\%$ are used in maintaing the internal structure of the CPTs.
- $9 - 13\%$ are used in database querying. Actually, this is quite unexpected.

It is unexpected that only about $9 - 13\%$ are used in database querying, which is a consequence of the large percentage used in maintaining the CPTs. Section 4.6 described how the CPTs were constructed, and the reason for the large percentage of *MakeStatistics* is, apparently, that we perform *un-indexed* in-memory data queries using a select statement. Hence, the queries perform poorly while selecting the values of $X.A$s parents. One approach of alleviating this problem, is to create the appropriate indices. This also takes time, but vastly outweighs selecting on un-indexed columns. However, the indices must be repeatedly created during search, when new dependencies are added. We may have to look else where for a possible optimal solution.

Also, the memory footprint of the implementation is very small; depending on the size of the CPTs. In our tests on the daycare and movie database, the footprint was only about $20 - 30$ megabytes. Which suggests, that we can use more memory for speed optimizations, such as creating indices.

7.5 Structure Evaluation Test

In the following two sections, we discuss the various dependencies in the learned structure of the daycare and movie database.

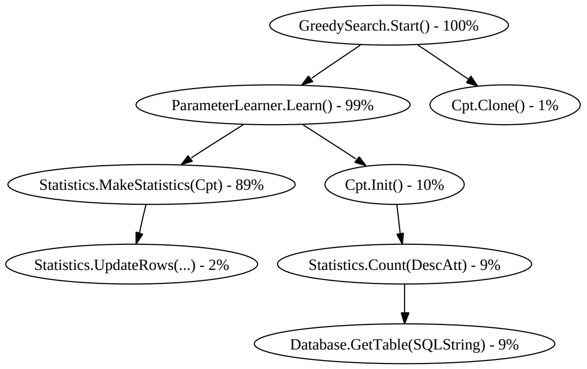

GreedySearch.Start() - 100%
ParameterLearner.Learn() - 99%
Cpt.Clone() - 1%
Statistics.MakeStatistics(Cpt) - 89%
Cpt.Init() - 10%
Statistics.UpdateRows(...) - 2%
Statistics.Count(DescAtt) - 9%
Database.GetTable(SQLString) - 9%

Figure 7.10: The call graph of the structure learning procedure on the daycare database. Methods of the .NET framework are not included.

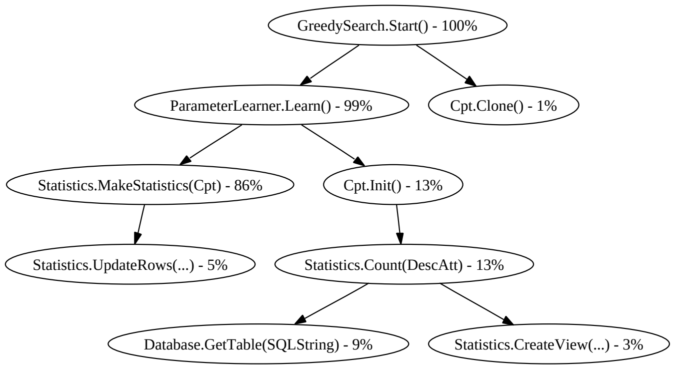

GreedySearch.Start() - 100%
ParameterLearner.Learn() - 99%
Cpt.Clone() - 1%
Statistics.MakeStatistics(Cpt) - 86%
Cpt.Init() - 13%
Statistics.UpdateRows(...) - 5%
Statistics.Count(DescAtt) - 13%
Database.GetTable(SQLString) - 9%
Statistics.CreateView(...) - 3%

Figure 7.11: The call graph of the structure learning procedure on the movie database. Methods of the .NET framework are not included.

$P(Child.Subsidy$ $\mid Parent.Income)$	**-1**	**1**	**2**	**3**	**4**	**5**
CS	0.108	0.315	0.106	0.045	0.026	0.188
DS	0.005	0.037	0.008	0	0	0.004
ES	0.218	0.111	0.357	0.455	0.349	0.003
SS	0.020	0.037	0.022	0.031	0.019	0.003
NO	0.422	0.185	0.322	0.393	0.550	0.495
RE	0.228	0.315	0.184	0.076	0.056	0.307

Table 7.1: The $P(Child.Subsidy \mid Parent.Income)$ with 5 bins, where -1 indicates a missing relation, 1 is a low income, and 5 is a high income.

In the daycare structure, we expect to find a superset of the correlated propositional attributes presented in our prior work [10]. In the movie structure, no previous references are available. Although, the movie domain is well-studied, the lack of consistence between applied databases makes it difficult to correlate different results. Consequently, we simple expect to find reasonable dependencies in the movie structure.

7.5.1 Daycare Structure

The daycare structure, see Figure 7.12, is constructed with greedy search and $\alpha = 6$, since we get a reasonable number of dependencies and likelihood.

In the following we discuss some of the dependencies which seems reasonable.

Child.Age depends on *Institution.Type*, because the different institution types have different age ranges, such that kindergarten is for middle age children between 2 and 6 years old. Also it depends on *Parent.Age* via *Mother* which suggests that the mother's age is better at predicting the age of the child, than the father's age.

Child.Subsidy depends on the number of children in the child's institution, *Institution.Children*, which makes sense since a lower number of children decreases the change of getting a child with a subsidy. It also depends on *Parent.Income* through *Mother*, since the income usually is registered on the mother in a marriage.

The CPT for $P(Child.Subsidy \mid Parent.Income)$ with *Institution.Children* marginalized and 5 bins in *Parent.Income*, is shown in Table 7.5.1. The probabilities for **DS** and **SS** are very small due to few instances in the database, which was also the case for our prior work. Also, the probabilities for **ES** is very low if the income is high (5), which also correlates which our prior findings.

Child.Livingcondition depends on *Parent.Married* and *Parent.Income*, because the marital status is reflected in the living condition of the child and the income of the parent, apparently, also has a influence.

Parent.Married depends on *Parent.Age* which indicates that there may be certain age groups that have more frequent marital statuses, than other age groups.

Parent.Income depends on *Parent.ResidentialValue*, because the income often determines the size and price of the resident. Also, it depends on *Child.Sex* which is probability an indirect dependency on *Parent.Sex*, since the sex of the parent usually has influence on the income.

Parent.Citizenship depends on *Parent.ResidentialValue* and *Parent.Married*, since e.g. Somalians often have a low residential value and are always married in the database.

Parent.Sex depends on *Child.Sex* through two inverse slots, *MotherOf* and *FatherOf*. At first glance, this may seem as an odd dependency. But *Child.Sex* is a good indicator for *Parent.Sex*, since if we follow the slot *Father* from child we will always have $Parent.Sex = Male$ and for the slot *Mother*, $Parent.Sex = Female$. Likewise following the inverse slot *FatherOf* or *MotherOf*, we will always end up with a child or an undefined value (null) if the father or mother have no children.

The above only contains a subset of the dependencies in Figure 7.12, which we found interesting. We did find a superset of dependencies of our prior work, although it could be interesting to perform inference in the ground Bayesian network, we will leave this as future work.

7.5.2 Movie Structure

The final structure of the movie database is shown in Figure 7.13 with $\alpha = 3$.

Generally, there are many dependencies connecting attributes that contain information about years, e.g. *Studio.Firstyear* depends on *Movie.Year* and *Director.Directorstart* depends on *Movies.Year*. These dependencies are trivial and not surprising, but however indicates that our implementation is working correctly and that the final structure is reasonable.

In the following we discuss some of the dependencies which we find most interesting.

Movies.Year depends on *Casts.Genre*, which makes sense since in different decades different genres becomes popular.

Director.Directorstart depends on *Casts.Genre*, which also strengthen our belief that different movie genres are more popular in some decades.

$P(Director.Background$ $\vert Movies.Year)$	**-1**	**1**	**2**	**3**	**4**	**5**
American	0.424	0.496	0.0.512	0.568	0.476	0.621
British	0.060	0.076	0.112	0.091	0.071	0.040
Italian	0.064	0.106	0.058	0.061	0.124	0.105
France	0.031	0.047	0.059	0.044	0.072	0.030

Table 7.2: The $P(Director.Background \mid Movies.Year)$ with 5 bins, where -1 indicates a missing relation, 1 indicates an old movie, and 5 indicates a recent movie.

$Casts.Genre$ depends on $Actor.Sex$ which could be plausible since more female actors would be casted to romance and comedy movies. The other way around with action movies.

$Director.Background$ depends on $Studio.Country$, which seems correct since both attributes express information about citizenship. It is likely that a director often chooses a studio in the same country he comes from. It also depends on $Movies.Year$, which could indicate that in different decades different nationalities where more attractive. However this dependency seems not so trustworthy as the other, so a fraction of the CPT is shown in Table 7.5.2. Parts of the CPT has been omitted, since there are almost 50 different values of $Director.Background$. Only the four largest entries are shown.

It shows that in all values of $Movies.Year$ an American director is most likely. This do not seem surprising. The dependency from $Movies.Year$ to $Director.Background$ could be because the proportion of American directors is different in the different time periods. If the values were equal in all periods then this dependency would be wrong, since $Director.Background$ would be independent of $Movies.Year$.

$Studio.City$ depends on $Studio.Country$, which is trivial since different cities resides in different countries. It further depends on $Studio.Lastyear$ which indicates that over time different studios located in different cities became attractive.

$Movies.Awtype$ depends on $Movies.Year$, because in different time periods different awards were given. It could be the fact that the different awards wear out over time.

The structure of the movie database indeed contains reasonable dependencies and confirms prior beliefs about the movie domain.

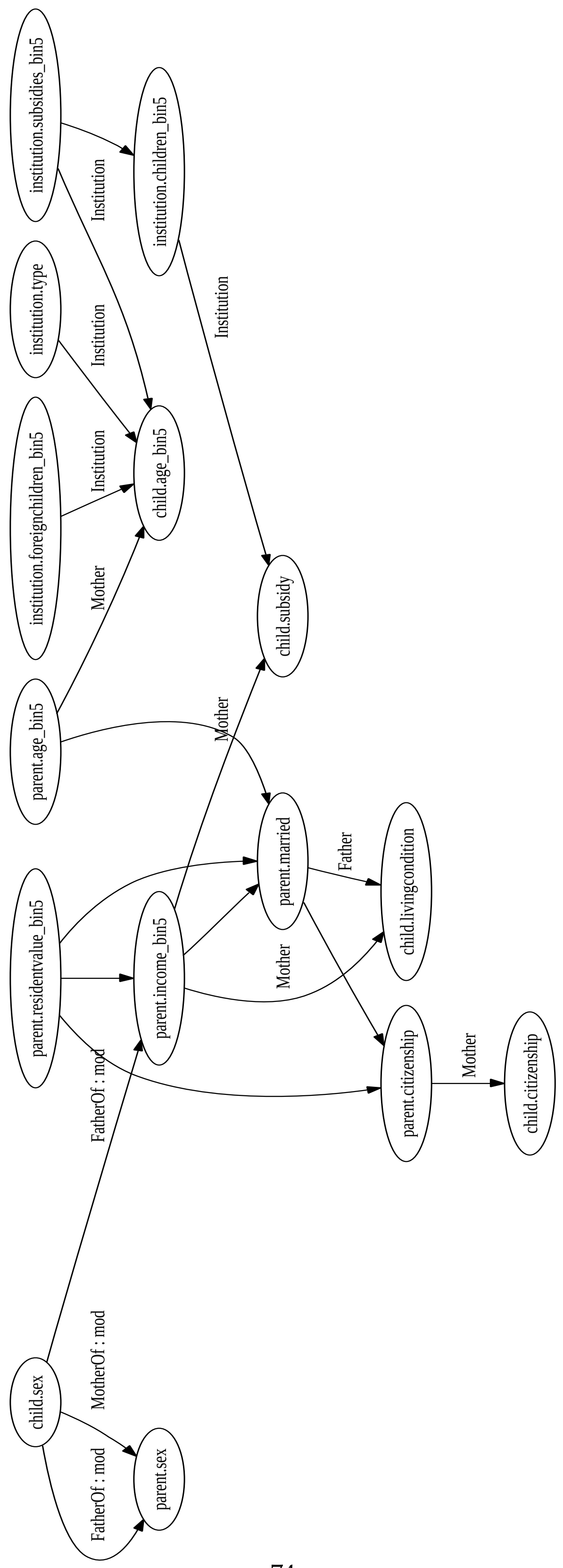


Figure 7.12: The dependency structure of the relational daycare database with $\alpha = 6$.

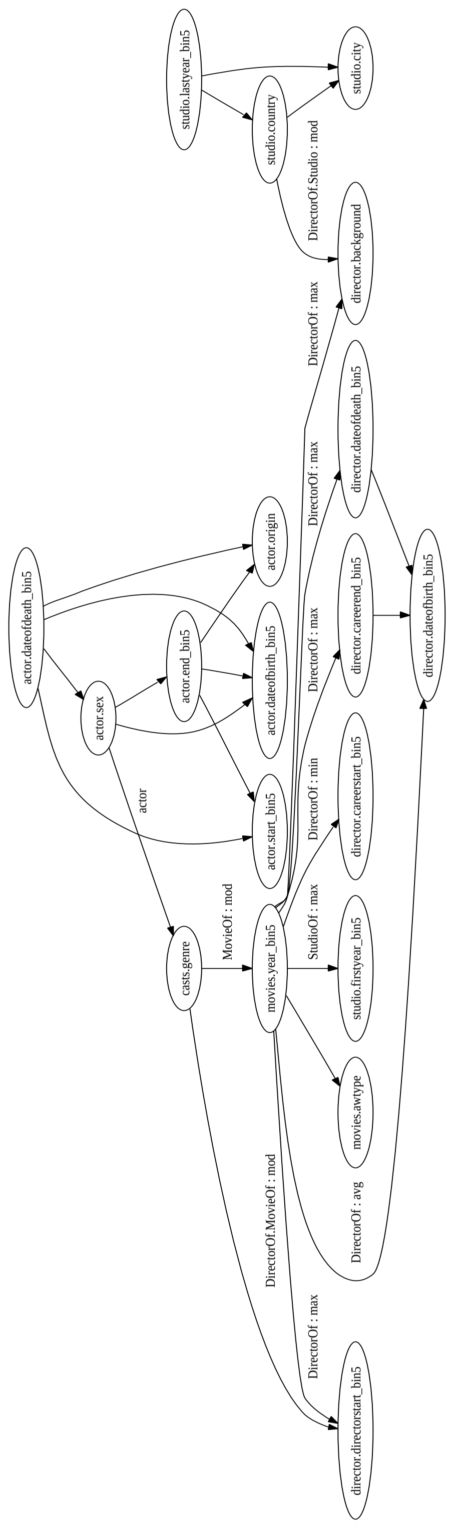


Figure 7.13: The dependency structure of the relational movie database with $\alpha = 3$.

Chapter 8

Conclusion

This chapter presents the conclusion of our work, and presents some areas of future work.

8.1 Conclusion

Our design of PRMs, which supported learning the parameters and the dependency structure, had three design criteria:

Simplicity: The object-oriented design is very simple, we only introduced classes directly correlated with the syntax of the relational language and entities in PRMs, and functional classes for database querying and learning PRMs.

Flexibility: The flexibility of the design is yet to be determined, however, since the design is simple it should only be a matter of extending the classes through inheritance. This facilitates, that it should be relative easy to extend the design with e.g. link uncertainty [11].

Performance: Performance considerations, resulted e.g. in local structure change and only querying the database once per attribute, see Chapter 5. Although, these optimizations came short while profiling the implementation, which showed a significant design flaw, performance wise, see Section 7.4. However, since we prioritized simplicity over performance we settled with the current design, and left any further optimizations as further work.

Testing our design, uncovered the following aspects:

Complexity Penalty Test: Showed that by introducing a penalty multiplier, α, we could tweak the search procedure by deciding the trade off between *likelihood* and *complexity*. Hence, α also influences the running time of searching for a structure using a greedy approach. See Section 7.1.

Search Test: The search test showed that introducing randomness was beneficially, although the pure random algorithms did not perform satisfiable. Hence, the hybrid search algorithm is a good alternative to greedy, although it did not surpass greedy in achieving a better model (though close), it did surpass it in speed. A best-first approach could be applied, if a higher score could be archived since the learning procedure is *relative fast*. With about one hour for learning the daycare database, and two hours for learning the movie database.

Scalability Test: The limited scalability test showed a near linear scalability on samples of the daycare database. The scalability of the movie database, showed that it is not always the case that increased data leads to higher running time, see Section 7.3.

Profile Test: The profile test, showed a design flaw which greatly increased the running time of our implementation, which would be redeemed by using indicies. Otherwise, the profiler showed a very small memory footprint at about 20-30 megabytes. See Section 7.4.

Structure Evaluation Test: We found a superset of the correlated daycare propositional attributes of our prior work. An interesting new dependency was the impact of the residential value, which was not present in our prior work. The structure of the movie database showed many plausible dependencies, also through longer slot chains, which provided much knowledge about the movie domain e.g. that the genre of casts depends on the actors in it. See Section 7.5.

In summary, our contributions are:

- A simple design and implementation of PRMs. The design specifies the search for potential parents, four algorithms for finding a good dependency structure, and explicitly shows how to compute the sufficient statistics for learning the parameters.
- Two relational databases which PRMs can be directly applied.
- Tests that show aspects of the dependency structure search, tweaking the score function, scalability, performance usage of our implementation, and structure evaluation.
- Dependencies of the movie and daycare databases, which may be referred by other literature on relational data mining.

Overall, PRMs seems to be a very good alternative to propositional data mining, instead of applying flattening.

8.2 Future Work

Our design and tests could be expanded in various ways:

- Using PRMs with link uncertainty [11] facilitates that we can specify the probability that certain objects are linked. Instead of explicitly specifying the probabilities, the approach relies on certain attribute partitions such as the type of a movie, and then specifies the probabilities over these attribute partitions that certain objects are linked. Our design could support this, by creating the classes `PRMLink.PartitionAtt`, `PRMLink.DescAtt`, `PRMLink.PartititonAtt` etc. where *PRMLink* is the namespace.

- The importance of numeric values in relational databases makes discretization a necessity. Instead of applying a fixed discretization method, the search for potential parents could be expanded to include this, hence a potential parent would be $(Y.B, \tau, \gamma, \lambda)$ where λ is a discretization method.

- Synthetic relational data would allow us to assess the validity of our learning procedure, by sampling data according to the specified probabilities and dependencies. This "gold standard" would then be compared with the learned dependencies and parameters of our learning procedure. Also, this would allow us to perform further scalability tests.

- A domain especially suited for relational data mining, is social networks. Social networks have a recursive nature, where a friend is a friend of a friend etc. Extensive studies with propositional and relational techniques [6; 28] have been applied, it could be interesting to apply PRMs to this domain, with e.g. data from Orkut [20].

- Compiling a PRM to a ground Bayesian network, would allow us to perform inference in the ground network. The compilation would result in a quite large Bayesian network, which might require approximate inference instead of exact inference.

Chapter 9

Appendix A - Summary

When propositional data mining techniques, applied to relational databases, is not adequate because of the disadvantage of flattening, relational data mining is an alternative. Flattening is the process of transforming a relational database to a propositional data set. We presented two disadvantage of flattening, namely, statistical incorrectness and attributes must be fixed. These disadvantages can be avoided by using relational mining techniques such as ILP or statistical graphical models.

We presented Probabilistic Relational Models (PRMs), a statistical graphical model, in Chapter 2. PRMs extend the concept of Bayesian networks with the relational language, such that is considers relations between objects. As in a Bayesian network, PRMs consists of two components; the dependency structure and its parameters. The dependency structure consists of internal and external dependencies, where the latter is either single-valued or multi-valued. The parameters are the conditional probabilities of an attribute given its dependencies. The multi-valued external dependencies would result in many dependencies, thus, we explained how a compact representation based on aggregation could be used to specify the probabilities of these multi-valued dependencies.

Chapter 3, further explained PRMs by introducing the concept of learning the structure and parameters of a PRM. We presented a simple approach based on maximum likelihood for learning the parameters of a PRM, using sufficient statistics. In learning the dependency structure, we defined three concepts; that define the legal structure, how to evaluate them, and finally the search heuristics. We defined a legal structure as a DAG, although this condition can be relaxed. Our score function for evaluating a structure, was based the Bayesian Information Criteria, and the search heuristics use a notation of potential parents.

The design, in Chapter 4, presented a simple design of PRMs with four different search algorithms for finding a good dependency structure. In our design the dependency structure is based on potential parents, which we defined as a triple containing an attribute, a slot chain, and an aggregator. Furthermore, we presented algorithmic details of the search for potential parents and a dependency structure. Lastly, the chapter explicitly described how to compute the sufficient statistics using SQL queries to the relational database.

In Chapter 5, we presented two relational databases; the daycare database and movie database. The schema for each, were outlined after we had selected the attributes of interest and applied prepossessing.

The implementation, in Chapter 6, were submitted to five different tests, in Chapter 7, ranging from scalability to model evaluation. The scalability test showed a near linear scalability on sampled databases, although further tests would be necessary to say anything conclusive about scalability. The four different search algorithms were applied and both greedy and a hybrid approach gave good results. Using the search algorithms, we evaluated the learned structures of the daycare and movie database. Both structure provided important insight into the two domains.

We concluded that PRMs seemed to be a very good alternative to propositional data mining, instead of applying flattening.

Bibliography

[1] C.L. Blake and C.J. Merz. UCI repository of machine learning databases, 1998.

[2] H. Blau, N. Immerman, and D. Jensen. A visual language for querying and updating graphs. Technical Report 2002-037.

[3] Hendrik Blockeel and Luc De Raedt. Top-down induction of first-order logical decision trees. *Artificial Intelligence*, 101(1-2):285–297, 1998.

[4] L. Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone. Classification and regresion trees, 1984.

[5] Wray L. Buntine. Operations for learning with graphical models. *Journal of Artificial Intelligence Research*, 2:159–225, 1994.

[6] C. Butts. Network inference, error, and informat (in)accuracy: a bayesian approach, social networks. 2003.

[7] D. Chickering, D. Geiger, and D. Heckerman. *Learning Bayesian Networks is NP-Hard*. MSR-TR-94-17, Microsoft Research.

[8] Sašo Džeroski. Multi-relational data mining: an introduction. *SIGKDD Explor. Newsl.*, 5(1):1–16, 2003.

[9] Nir Friedman, Lise Getoor, Daphne Koller, and Avi Pfeffer. Learning probabilistic relational models. In *IJCAI*, pages 1300–1309, 1999.

[10] Morten Gade and Michael Gade Nielsen. Data mining for descriptive modeling: Daycare subsides and institutions in herning kommune, 2005.

[11] L. Getoor, N. Friedman, D. Koller, and B. Taskar. Learning probabilistic models of link structure, 2002.

[12] Lise Getoor. *Learning Statistical Models from Relational Data. Ph.D dissertation*. Computer Science Department, Stanford University, 2002.

[13] David Heckerman. A tutorial on learning with bayesian networks.

[14] David Heckerman, Christopher Meek, and Daphne Koller. Probabilistic models for relational data. Technical Report MSR-TR-2004-30.

[15] Lawrence B. Holder and Diane J. Cook. Graph-based relational learning: current and future directions. *SIGKDD Explor. Newsl.*, 5(1):90–93, 2003.

[16] Manfred Jaeger. Relational Bayesian networks. In Morgan Kaufmann, editor, *Proceedings of the 13th Conference on Uncertainty in Artificial Intelligence*, pages 266–273, 1997.

[17] D. Jensen and J.Neville. Correlation and sampling in relational datamining. In *Proceedings of the 33rd Symposium on the Interface of Computing Science and Statistics*, 2001.

[18] Daphne Koller and Avi Pfeffer. Probabilistic frame-based systems. In *AAAI/IAAI*, pages 580–587, 1998.

[19] S. Kramer and G. Widmer. Inducing classification and regression trees in first order logic.

[20] Orkut.com LLC. Orkut – the social network. `http://www.orkut.com`.

[21] M. Mastracii. Nprof: The .net profiler application and api. `http://nprof.sourceforge.net/Site/SiteHomeNews.html`.

[22] J. Neville and D. Jensen. Collective classification with relational dependency networks. In *Proceedings of the 2nd Multi-Relational Data Mining Workshop, 9th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2003.

[23] J. Neville, D. Jensen, L. Friedland, and M. Hay. Learning relational probability trees. In *Proceedings of the 9th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2003.

[24] J. Nevillea, D. Jensen, and B. Gallagher. Simple estimators for relational bayesian classifers. In *Proceedings of The Third IEEE International Conference on Data Mining*, 2003.

[25] J. Newton and R. Greiner. Hierarchical probabilistic relational models for collaborative filtering. *xx*, 0:00, 00.

[26] Jens D. Nielsen, Tomáš Kočka, and Jose M. Peña. On local optima in learning bayesian networks. In *Proceedings of the Nineteenth Conference on Uncertainty in Artificial Intelligence (UAI)*, 2003.

[27] J. R. Quinlan. *C4.5: programs for Machine Learning*. Morgan Kaufmann, 1993.

[28] P. Smyth. Statistical modeling of graph and network data. *Information and Computer Science, University of California, Irvine*.

[29] Lappoon R. Tang. Statement of research summary and current directions.

[30] Takashi Washio and Hiroshi Motoda. State of the art of graph-based data mining. *SIGKDD Explor. Newsl.*, 5(1):59–68, 2003.

[31] Gio Wiederhold. Movies database documentation. `http://kdd.ics.uci.edu/databases/movies/doc.html`.

[32] Ying Yang and Geoffrey I. Webb. A comparative study of discretization methods for naive-bayes classifiers.