Annals of Mathematics and Artificial Intelligence manuscript No.
(will be inserted by the editor)

Learning Directed Probabilistic Logical Models: Ordering-Search versus Structure-Search

Daan Fierens · Jan Ramon · Maurice Bruynooghe · Hendrik Blockeel

Received: date / Accepted: date

Abstract We discuss how to learn non-recursive directed probabilistic logical models from relational data. This problem has been tackled before by upgrading the structure-search algorithm initially proposed for Bayesian networks. In this paper we show how to upgrade another algorithm for learning Bayesian networks, namely ordering-search. For Bayesian networks, ordering-search was found to work better than structure-search. It is non-obvious that these results carry over to the relational case, however, since there ordering-search needs to be implemented quite differently. Hence, we perform an experimental comparison of these upgraded algorithms on four relational domains. We conclude that also in the relational case ordering-search is competitive with structure-search in terms of quality of the learned models, while ordering-search is significantly faster.

Keywords statistical relational learning · probabilistic logical models · inductive logic programming · Bayesian networks · probability trees · structure learning

1 Introduction

There is an increasing interest in probabilistic logical models as can be seen from the variety of formalisms that have recently been introduced for representing such models. Many of these formalisms deal with directed models that are upgrades of Bayesian networks to the relational case. Learning algorithms have been developed for several such formalisms [12, 15, 19]. Most of these algorithms are essentially upgrades of the traditional *structure-search* algorithm for Bayesian networks [13].

An alternative algorithm for learning Bayesian networks, more recent than structure-search, is *ordering-search* [22]. Ordering-search is based on the fact that it is relatively easy to learn a Bayesian network if an ordering on the random variables is given (because

D. Fierens
Katholieke Universiteit Leuven, Department of Computer Science, Celestijnenlaan 200A, B-3001 Leuven
Tel.: +32-16-327576
Fax: +32-16-327996
E-mail: daan.fierens@cs.kuleuven.be

J. Ramon, M. Bruynooghe, H. Blockeel
Katholieke Universiteit Leuven, Department of Computer Science, Celestijnenlaan 200A, B-3001 Leuven

this eliminates the possibility of cycles; this was for instance the idea behind the seminal *K2* learning algorithm [3]). However, usually the best ordering is not known in advance. Hence, the idea behind ordering-search is to perform a heuristic search through the space of possible orderings to find the best ordering. Teyssier and Koller [22] experimentally compared ordering-search to structure-search for learning Bayesian networks. They found that ordering-search is competitive with structure-search in terms of quality of the learned Bayesian networks, while ordering-search is usually faster.

Unlike structure-search, ordering-search has not yet been upgraded to the relational case. The good performance of ordering-search for Bayesian networks motivates us to perform this upgrade, and to investigate whether also in that case ordering-search still performs well as compared to structure-search. This is an interesting question since it is non-obvious that the efficiency advantage of ordering-search over structure-search as observed for Bayesian networks also holds in the relational case. The reason for this is that ordering-search needs to be implemented quite differently in the relational case due to the fact that simple conditional probability tables can no longer be used (see Section 6.1.5).

1.1 Contributions

The main contributions of this work are three-fold. First, we upgrade the ordering-search algorithm towards learning non-recursive directed probabilistic logical models. Second, we discuss the relation of the resulting algorithm to the original ordering-search algorithm and to several algorithms for learning recursive dependencies. Third, we experimentally compare our ordering-search algorithm to the upgraded structure-search algorithm on four relational domains. We use the formalism Logical Bayesian Networks [6] but the proposed approach is also applicable to related formalisms such as Probabilistic Relational Models [10, 12], Bayesian Logic Programs [15, 16] and Relational Bayesian Networks [14].

Part of this work has been published before [9]. This paper extends our previous work in three ways. First, we included a discussion of some properties of Logical Bayesian Networks that are relevant for learning, and how to deal with them. Second, we included a discussion of the relation of our ordering-search algorithm for Logical Bayesian Networks to related algorithms (namely the original ordering-search algorithm and several algorithms for learning directed probabilistic logical models with recursive dependencies). Third, we extended the experimental analysis in several respects (we added two real-world datasets, we performed a more detailed analysis of running times of the algorithms, we report learning curves, and we give examples of learned dependencies). Apart from the above extensions, this paper also provides some more details on several issues.

1.2 Structure of the paper

This paper is structured as follows. We first discuss some preliminaries in Section 2. Then we review Logical Bayesian Networks in Section 3 and discuss some relevant properties in Section 4. We discuss the setting of learning non-recursive Logical Bayesian Networks in Section 5 and the corresponding ordering-search and structure-search algorithms in Section 6. We experimentally compare these algorithms in Section 7. In Section 8 we briefly discuss learning recursive models. Finally, in Section 9 we conclude.

2 Preliminaries

We first discuss some preliminaries about Bayesian networks [13] and logic programming [17].

2.1 Bayesian Networks

A *Bayesian network* is a compact specification of a joint probability distribution on a set of random variables (in this paper we only consider discrete random variables). A Bayesian network consists of a qualitative part and a quantitative part. The qualitative part is a *directed acyclic graph*, the so-called 'structure' of the Bayesian network. The nodes in this graph represent random variables, and the directed edges specify (in)dependencies between the random variables: each variable is conditionally independent of its non-descendants given its parents. Essentially, the parents of a variable can be seen as the 'direct influences' of that variable. The quantitative part of a Bayesian network is a set of *conditional probability distributions (CPDs)*. Concretely, each variable X needs a CPD that specifies the probability distribution of X given its parents (i.e., for each joint state of the parents of X, this CPD specifies a probability distribution on the possible values of X). CPDs can be represented in several ways. Two popular formats are *conditional probability tables* [13] and *probability trees* [11].

When learning Bayesian networks from data, the goal is usually to find the structure and CPDs that maximize a certain scoring criterion, such as likelihood or the Bayesian Information Criterion [13].

2.2 Logic Programming

A *predicate* represents a property or relation and is denoted as p/n, where p is the name and n is the number of arguments or arity. Arguments of predicates are called *terms*, and can be constants (denoted by lower-case symbols), variables (denoted by upper-case symbols) or compound objects. An *atom* is a predicate together with the right number of arguments. A *literal* is an atom or a negated atom. A term, atom or literal is *ground* if it does not contain any variables. A (ground) *substitution* is an assignment of (ground) terms to variables and the result of applying a substitution θ to a literal (or conjunction of literals) l is denoted by $l\theta$. An *interpretation* of a set of logical predicates is an assignment of a truth value to each ground atom that is built from these predicates and that has arguments belonging to the considered domain of discourse.

A *definite clause* is of the form $head \leftarrow body$, where $head$ is an atom and $body$ is a conjunction of atoms (all free variables in the clause are implicitly universally quantified). A definite *logic program* is a finite set of definite clauses. The *least Herbrand model* of a definite logic program is an interpretation of all predicates used in the program in which all the clauses of the program are satisfied. Practically speaking, the least Herbrand model captures the semantics of a logic program.

3 Logical Bayesian Networks

We now review Logical Bayesian Networks by means of an example. We also define the notion of recursive and non-recursive Logical Bayesian Networks, and briefly discuss the relation of Logical Bayesian Networks to some other probabilistic logic formalisms.

3.1 Logical Bayesian Networks: Example

A *Logical Bayesian Network* or *LBN* [6] is essentially a specification of a Bayesian network conditioned on some logical input predicates that describe the domain of discourse. For instance, when modelling the well-known 'university' domain [12], we would use predicates $student/1$, $course/1$, $prof/1$, $teaches/2$ and $takes/2$ with their obvious meanings. The semantics of an LBN is that, given an interpretation of these logical predicates, the LBN induces a particular Bayesian network (see below).

In LBNs random variables are represented as ground atoms built from certain special predicates, the *probabilistic predicates*. For instance, if $intelligence/1$ is a probabilistic predicate then the atom $intelligence(ann)$ is called a probabilistic atom and represents a random variable. Apart from sets of logical and probabilistic predicates, an LBN basically consists of three parts: a set of random variable declarations, a set of dependency statements, and a set of logical CPDs. The former two together determine the structure of the induced Bayesian network, while the logical CPDs quantify the dependencies in this structure.

3.1.1 The Structure of the Induced Bayesian Network

For a given interpretation of the logical predicates, the random variable declarations in an LBN determine the set of random variables (nodes) in the induced Bayesian network, while the dependency statements determine the dependencies (edges). Together, this fully determines the structure of the induced Bayesian network.

Random variable declarations are of the form $random(p) \leftarrow body$, where $random/1$ is a logical predicate (which cannot be used outside of the random variable declarations), p is a probabilistic atom and $body$ is a conjunction of logical atoms. For the university domain, the random variable declarations are the following.

```
random(intelligence(S)) <- student(S).
random(ranking(S)) <- student(S).
random(difficulty(C)) <- course(C).
random(rating(C)) <- course(C).
random(ability(P)) <- prof(P).
random(popularity(P)) <- prof(P).
random(grade(S,C)) <- takes(S,C).
random(satisfaction(S,C)) <- takes(S,C).
```

Informally, the first clause, for instance, should be read as "$intelligence(S)$ is a random variable if S is a student". Assuming that the interpretation of the logical predicates is defined by a set of definite clauses, these clauses together with the random variable declarations form a definite logic program, the *random variable declaration program*, that defines the random

variables in the induced Bayesian network[1]. Formally, there is a random variable a for every atom $random(a)$ in the least Herbrand model of the random variable declaration program.

Dependency statements are of the form $head \mid body \leftarrow context$, where $head$ is a probabilistic atom, $body$ is a conjunction of probabilistic atoms and $context$ is conjunction of logical literals. If the context is empty (or 'true') we omit it from the notation and write $head \mid body$. The dependency statements for the university domain are the following.

```
grade(S,C) | intelligence(S), difficulty(C).
ranking(S) | grade(S,C).
satisfaction(S,C) | grade(S,C).
satisfaction(S,C) | ability(P) <- teaches(P,C).
rating(C) | satisfaction(S,C).
popularity(P) | rating(C) <- teaches(P,C).
```

Informally, the first statement, for instance, should be read as "the grade of a student S for a course C depends on the intelligence of S and the difficulty of C" and the last statement as "the popularity of a professor P depends on the rating of a course C if P teaches C". Formally, a dependency statement $a \mid a_1, \ldots a_n \leftarrow context$ specifies that $a_i\theta$ is a parent of $a\theta$ in the induced Bayesian network if θ is a ground substitution for which the conjunction $random(a)\theta, random(a_1)\theta, \ldots random(a_n)\theta, context\theta$ is true in the least Herbrand model of the random variable declaration program. Note that this implies that a dependency statement with multiple probabilistic atoms in the body (such as the first statement) only 'fires' if all atoms in the body are indeed random variables.

To make this more concrete, consider the following interpretation of the logical predicates (specified as a set of facts for these predicates).

```
student(mike).      student(emma).
prof(john).
course(ai).         course(ml).
takes(emma,ai).     takes(emma,ml).     takes(mike,ai).
teaches(john,ai).   teaches(john,ml).
```

Given an LBN with the above random variable declarations and dependency statements, the structure of the induced Bayesian network for this interpretation is shown in Figure 1. The random variables (nodes) in this network are determined by the random variable declarations, the dependencies (edges) by the dependency statements.

3.1.2 Quantifying the Dependencies

To quantify the dependencies specified by the dependency statements, LBNs associate with each probabilistic predicate a so-called logical CPD. These logical CPDs can be used to determine the CPDs in the induced Bayesian network.

We represent logical CPDs under the form of logical probability trees in TILDE [7] (as an alternative to the combining rules used in some other formalisms [14, 16, 15, 19]). The leaves of the tree for a probabilistic atom p_{target} contain probability distributions on the values of p_{target}. The internal nodes of the tree contain a) tests on the values of a probabilistic atom, b) conjunctions of logical literals or c) combinations of the two. In order

[1] This can be extended to clauses and random variable declarations with *negative* literals in the body if care is taken that the resulting program has a unique two valued model, i.e., that the well-founded model is two valued.

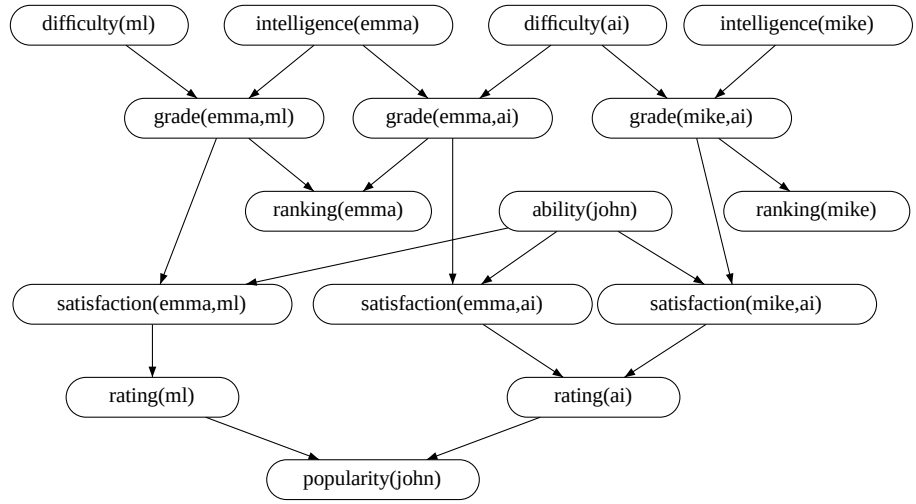

Fig. 1 The structure of the induced Bayesian network for our running example.

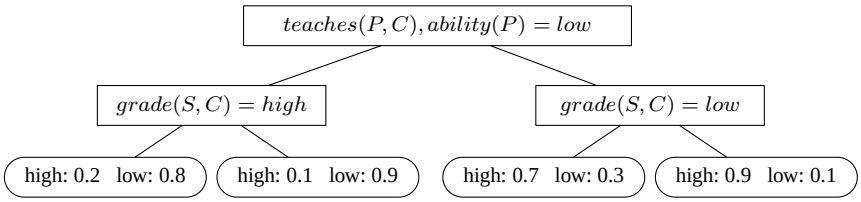

Fig. 2 Example of a logical CPD for $satisfaction(S, C)$. Tests in internal nodes are binary. When a test succeeds the left branch is taken, when it fails the right branch is taken. Note that internally in TILDE, tests like $grade(S, C) = low$ are represented as $grade(S, C, low)$.

for the tree for a probabilistic atom p_{target} to be consistent with the dependency statements, the tree can of course only test on probabilistic atoms that are parents of p_{target} according to the dependency statements. An example of a tree for $satisfaction(S, C)$ is shown in Figure 2. Recall that according to the dependency statements, $satisfaction(S, C)$ depends on $grade(S, C)$, and on $ability(P)$ where P teaches C.

Note that we cannot simply use conditional probability tables as a format for representing CPDs in LBNs since they are too restrictive. One problem is that conditional probability tables cannot deal with a variable number of inputs (as occurs for instance when the ranking of a student depends on his grades for all courses taken, and the number of courses per student can vary). Logical probability trees in TILDE can deal with a variable number of inputs because the tests in the internal nodes are first-order queries. As shown by Van Assche et al. [23], this makes it possible to express selection (for instance, does there exist a course for which the student has a high grade), aggregation (for instance, is the average of all grades of the student high) and combinations of the two.

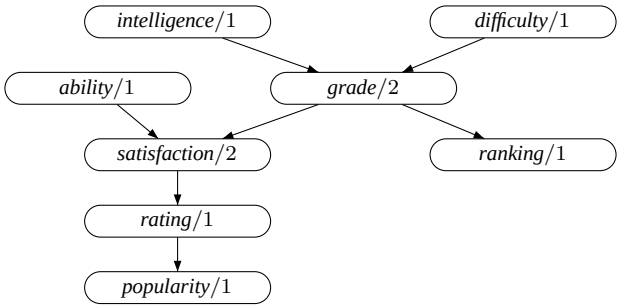

Fig. 3 The predicate dependency graph of the LBN for the university domain.

3.2 Recursive and Non-recursive Logical Bayesian Networks

The *predicate dependency graph* of an LBN is the graph that contains a node for each probabilistic predicate and an edge from a node p_1 to a node p_2 if the LBN contains a dependency statement with predicate p_2 in the head and p_1 in the body. The predicate dependency graph of the LBN for the university domain is shown in Figure 3.

An LBN is called *non-recursive* if its predicate dependency graph is acyclic and *recursive* otherwise. Note that for non-recursive LBNs the induced Bayesian network (for any interpretation of the logical predicates) is always acyclic. For recursive LBNs, the induced network can be cyclic or acyclic depending on the interpretation of the logical predicates. If it is cyclic then the semantics of the LBN with respect to that interpretation is left undefined.

3.3 Related Probabilistic Logic Formalisms

The formalism of LBNs is closely related to other probabilistic logic formalisms that are based on Bayesian networks. Some of the many such formalisms are Probabilistic Relational Models [12], Bayesian Logic Programs [16] and Relational Bayesian Networks [14]. Like LBNs, most of these formalisms make a distinction between a qualitative part (such as the random variable declarations and dependency statements in LBNs), and a quantitative part (such as the logical CPDs in LBNs). For each of these formalisms, the representation for the *qualitative part* is somewhat different. For some more details on this issue, we refer to Fierens et al. [6]. However, the main difference between LBNs and other probabilistic logic formalisms based on Bayesian networks might be in the *quantitative part*. To the best of our knowledge, LBNs is the only such formalism that uses *logical probability trees* to quantify the dependencies. Most other formalisms instead use some kind of combining rules (such as noisy-or) [16] or combination functions [14]. Our motivation for using logical probability trees in LBNs is the success of probability trees as a format for specifying CPDs in Bayesian networks [11]. The advantage of probability trees is that they can compactly represent CPDs that exhibit context-specific independencies, which makes it possible to learn more accurate Bayesian networks [11] and speed up probabilistic inference [1]. By using logical probability trees in LBNs, we essentially upgrade this approach to the relational case.

4 Properties of Logical Bayesian Networks

In this section we discuss two properties of LBNs that have to do with cases in which dependency statements can be rewritten into other dependency statements, leading to an equivalent LBN. (We call two LBNs equivalent if, for any possible interpretation of the logical predicates, their induced Bayesian network for that interpretation is the same.)

1. A dependency statement with multiple atoms in the body can be rewritten into a set of dependency statements each with only one atom in the body. We call this process 'decomposing' the dependency statement.
2. Some dependency statements are redundant or contain redundant literals in the context. Such redundant statements or literals can be removed.

Both these properties are *relevant with respect to learning*.

1. The first property implies that we can restrict the hypothesis space to LBNs with dependency statements with only one atom in the body (since any other LBN is equivalent to such an LBN).
2. The second property is useful to simplify learned dependency statements.

We will make use of these two observations in Section 6.

Below we only discuss these two properties for LBNs in which the *random variable declarations are disjoint* (the head of any random variable declaration does not unify with the head of any other random variable declaration). The reason for this is that these properties are most easily formulated for such LBNs and that only considering such LBNs is no restriction: any set of non-disjoint random variable declarations can be translated into an equivalent set of disjoint random variable declarations by introducing an extra logical predicate.

Example 1 (Disjoint random variable declarations) The following two random variable declarations are not disjoint.

```
random(ranking(S)) <- bachelor_student(S).
random(ranking(S)) <- master_student(S).
```

If we include the following clauses

```
student(S) <- bachelor_student(S).
student(S) <- master_student(S).
```

as background knowledge in the random variable declaration program, then we can replace the two original random variable declarations by the following one.

```
random(ranking(S)) <- student(S).
```

□

4.1 Decomposing Dependency Statements

We now show that a dependency statement with multiple atoms in the body can be decomposed into an equivalent set of dependency statements each with only one atom in the body.

Theorem 1 (Decomposing dependency statements) *Consider an LBN $\mathcal{L}_1$ containing a dependency statement of the form*

$$a \mid a_1, \ldots, a_n \leftarrow c. \qquad n \geq 2.$$

Let $\mathcal{L}_2$ be the LBN obtained by replacing this dependency statement by the following set of dependency statements

$$\begin{cases} a \mid a_1 \leftarrow c, b_1. \\ \vdots \\ a \mid a_i \leftarrow c, b_i. \\ \vdots \\ a \mid a_n \leftarrow c, b_n. \end{cases}$$

where b_i is the conjunction of the bodies of the random variable declarations with the atoms $a_1, \ldots, a_{i-1}, a_{i+1}, \ldots, a_n$ in the head.

Then $\mathcal{L}_1$ and $\mathcal{L}_2$ are equivalent.

We prove this theorem in Appendix A.1.

The intuition behind this decomposition is best explained in two steps.

1. Recall that a dependency statement of the form

   $$a \mid a_1, \ldots, a_n \leftarrow c. \qquad n \geq 2$$

   only 'fires' if all atoms in the body are defined as random variables (Section 3.1.1). Hence, such a dependency statement can be replaced by the following set of 'decomposed' dependency statements.

   $$\begin{cases} a \mid a_1 \leftarrow c, random(a_2), \ldots, random(a_n). \\ \vdots \\ a \mid a_i \leftarrow c, random(a_1), \ldots, random(a_{i-1}), random(a_{i+1}), \ldots, random(a_n). \\ \vdots \\ a \mid a_n \leftarrow c, random(a_1), \ldots, random(a_{n-1}). \end{cases}$$

   The function of the extra $random/1$ atoms in the context of these decomposed dependency statements is to ensure that these statements fire if and only if the original dependency statement fires too.
2. For each of these decomposed dependency statements, we can replace all the $random/1$ atoms in the context by their definition as given by the random variable declarations (in logic programming terminology, this is called *unfolding* the $random/1$ atoms). The result of this step is the set of dependency statements given by Theorem 1.

   We now illustrate this using two examples.

Example 2 (Decomposing dependency statements) Consider again the LBN for our running example, in particular the following random variable declarations and dependency statement.

```
random(intelligence(S)) <- student(S).
random(difficulty(C)) <- course(C).
random(grade(S,C)) <- takes(S,C).
grade(S,C) | intelligence(S), difficulty(C).
```

In a first step, this dependency statement can be decomposed into the two following statements.

```
grade(S,C) | intelligence(S) <- random(difficulty(C)).
grade(S,C) | difficulty(C) <- random(intelligence(S)).
```

In a second step, we can replace the $random/1$ atoms in the context by their definition as given by the random variable declarations.

```
grade(S,C) | intelligence(S) <- course(C).
grade(S,C) | difficulty(C) <- student(S).
```

□

Example 3 (Decomposing dependency statements (2)) As a more advanced example, consider the following random variable declarations and dependency statement.

```
random(ranking(S)) <- student(S).
random(intelligence(S)) <- student(S).
random(thesis_score(S)) <- student(S), in_master(S).
ranking(S) | intelligence(S), thesis_score(S).
```

The dependency statement only fires if both intelligence and thesis-score are defined as random variables. Note that ranking and intelligence are defined for students but thesis-score is defined only for particular students, namely master students. Hence, the dependency statement only fires for master students.

In a first step, this dependency statement can be decomposed as follows.

```
ranking(S) | intelligence(S) <- random(thesis_score(S)).
ranking(S) | thesis_score(S) <- random(intelligence(S)).
```

In a second step, we can again replace the $random/1$ atoms by their definition.

```
ranking(S) | intelligence(S) <- student(S), in_master(S).
ranking(S) | thesis_score(S) <- student(S).
```

In the next section we show how these statements can be further simplified. □

4.2 Redundancy in Dependency Statements

We now show that some LBNs contain redundant dependency statements, or redundant literals in the context of the dependency statements. The definitions below are relative to some set of random variable declarations, but we leave this set implicit.

Definition 1 (Redundant dependency statement) We call a dependency statement of the form

$$a \mid a_1, \ldots, a_n \leftarrow c_1.$$

redundant with respect to another dependency statement with the same head and body but context c_2 if the conjunction $random(a), random(a_1), \ldots, random(a_n), c_1$ implies c_2.

The intuition behind this definition is the following. The condition in this definition is such that the second dependency statement (with c_2 in the context) fires whenever the first statement fires. Since both statements have the same head and body, this makes the first statement redundant.

Definition 2 (Redundant context literal) We call a literal l in the context of a dependency statement of the form

$$a \mid a_1, \ldots, a_n \leftarrow c, l.$$

(with c being a conjunction of logical literals) *redundant* if the conjunction $random(a)$, $random(a_1), \ldots, random(a_n), c$ implies l.

The following theorem shows that removing redundant dependency statements or redundant literals preserves the semantics of an LBN.

Theorem 2 (Redundancy in dependency statements) *Consider an LBN $\mathcal{L}_1$ and an LBN $\mathcal{L}_2$ obtained by removing redundant dependency statements and/or redundant context literals from $\mathcal{L}_1$. Then $\mathcal{L}_1$ and $\mathcal{L}_2$ are equivalent.*

We prove this theorem in Appendix A.2.

We now illustrate this by continuing Example 3.

Example 4 (Redundant context literal) Consider the random variable declarations and dependency statements obtained in Example 3.

```
random(ranking(S)) <- student(S).
random(intelligence(S)) <- student(S).
random(thesis_score(S)) <- student(S), in_master(S).
ranking(S) | intelligence(S) <- student(S), in_master(S).
ranking(S) | thesis_score(S) <- student(S).
```

In both dependency statements, the literal $student(S)$ is redundant. The intuition is that these statements only fire if $ranking(S)$ is a random variable, which requires that S is a student (according to the random variable declaration of $ranking(S)$). Hence we can remove these literals from the context.

```
ranking(S) | intelligence(S) <- in_master(S).
ranking(S) | thesis_score(S).
```

Note that the second statement does not specify in the context that S should be a master student. This is indeed not needed since thesis-score is only defined for master students and hence the statement only fires for master students anyway. □

In this example only redundant literals occurred. An example where redundant statements occur can be found later in this paper (Section 6.1.4).

5 Learning Non-recursive Logical Bayesian Networks: The Learning Setting

We now discuss the problem of learning LBNs from relational data. In this paper we focus on learning *non-recursive* LBNs. We briefly discuss learning recursive models in Section 8.

The learning task that we consider can be summarized as follows.

- **Given:**
  - a set of random variable declarations,
  - a scoring criterion,
  - a dataset.
- **Find:** the LBN (i.e., dependency statements and logical CPDs) that maximizes the score on the dataset.

Note that we assume that the random variable declarations are given. This is similar to the learning setting for Probabilistic Relational Models, where the relational schema is given [10, 12]. As a scoring criterion we use the *Bayesian Information Criterion (BIC)* [7, 13], but our algorithms can be used with any other decomposable scoring criterion as well (in terms of LBNs, a scoring criterion is decomposable if the score of an LBN can be written as the sum of local scores for each of the logical CPDs in that LBN).

The data that we learn from is a dataset of *mega examples* (terminology adopted from Mihalkova et al. [18]). Each mega example is a set of connected pieces of information. For instance, in a dataset about the inheritance of genes among family members, each mega example would correspond to one particular family; in a dataset for the university domain, each mega example corresponds to one particular collection of students, professors and courses with all their relations and properties. We assume that mega examples are mutually independent. Learning from a dataset consisting of independent mega examples (as opposed to learning from a single relational database) is useful for instance for cross validation. We use the term 'mega example' rather than simply 'example' because, as we will see later, each mega example can give rise to multiple smaller examples for learning logical CPDs.

In our learning setting, each mega example consists of two parts: a logical part and a probabilistic part. The logical part consists of an interpretation of the logical predicates. The probabilistic part consists of an assignment of values to all ground random variables (as determined by the random variable declarations). This is similar to the data used for learning Bayesian Logic Programs [5] or Relational Bayesian Networks [14].

Example 5 Consider a simplified variant of the university domain in which we consider only students and courses (but no professors) and use only four probabilistic predicates ($ranking/1$, $difficulty/1$, $rating/1$ and $grade/2$). The logical part of a mega example would then specify all students and courses, and which students take which courses. This could for instance look as follows.

```
student(s1).        student(s2).
course(c1).         course(c2).
takes(s1,c1).       takes(s2,c1).       takes(s2,c2).
```

The probabilistic part of a mega example specifies a value for all random variables for that mega example. This could for instance look as follows.

```
ranking(s1)=high    ranking(s2)=mid
difficulty(c1)=mid  difficulty(c2)=high
rating(c1)=low      rating(c2)=mid
grade(s1,c1)=high   grade(s2,c1)=mid   grade(s2,c2)=low
```

□

6 Learning Non-recursive Logical Bayesian Networks: The Algorithms

We now discuss the algorithms for learning non-recursive LBNs. We first show how to upgrade the ordering-search algorithm for Bayesian networks towards non-recursive LBNs. Next, we briefly discuss a structure-search algorithm for LBNs that is similar to existing learning algorithms for other probabilistic logic formalisms. Finally, we briefly discuss how both algorithms can be implemented in an efficient way. To stress the difference with Logical Bayesian Networks we will sometimes refer to ordinary Bayesian networks as 'propositional' Bayesian networks.

6.1 Ordering-search

First we briefly discuss ordering-search for the propositional case. Then we discuss the case of LBNs, and the differences between the two.

6.1.1 Ordering-search for Propositional Bayesian Networks

Ordering-search for propositional Bayesian networks [22] is based on two observations.

- It is relatively easy to learn a Bayesian network if an ordering on the random variables is given.
- Usually the best ordering is not known in advance. Hence a search through the space of possible orderings needs to be carried out to find the best ordering.

We now explain this further.

Given an ordering on the random variables, it is relatively easy to learn the best Bayesian network consistent with that ordering (by consistent we mean that the parents of a variable X should all precede X in the ordering). Given such an ordering, and provided that the scoring criterion used is decomposable, the learning task decomposes: to find the Bayesian network with the highest score we simply need to find for each random variable separately the CPD with the maximal local score. Note that the function of the ordering is to eliminate the possibility of cycles (if we would not take into account an ordering, learning each CPD separately would very likely lead to cycles in the network, which is not allowed). Note that in principle the parents for a variable X could be all the variables that precede X in the ordering. However, this would lead to a 'fully-connected' network, which is undesirable. Hence, the approach of Teyssier and Koller [22] is to look for the best k parents for X (they use at most k=4). They do this by considering all possible parent sets of size k (i.e., all subsets of size k of the set of variables that precede X in the ordering). For each such set they compute the score of the resulting CPD. They then select the highest scoring CPD, and use this as the final CPD for X. The parents of X are then the variables that occur in that CPD. This procedure is applied in turn to each random variable X, yielding a complete Bayesian network.

Using the above strategy, the score of the resulting network depends heavily on the ordering that is used. However, usually *the optimal ordering is not known in advance*. Hence, the idea of ordering-search is to search through the space of orderings, for each ordering applying the above procedure for finding the CPDs (we discuss the search process in more detail for LBNs below). At the end, the best ordering is retained, and the Bayesian network for that ordering is returned as the final network.

Teyssier and Koller [22] experimentally compared ordering-search and structure-search for propositional Bayesian networks and found that ordering-search is always at least as good and usually faster. As an explanation they note that the space of orderings is smaller than the space of structures, and that ordering-search does not need acyclicity tests, which are costly if there are many variables.

6.1.2 Ordering-search for Logical Bayesian Networks

Until now ordering-search has not yet been upgraded to the case of non-recursive directed probabilistic logical models[2]. The above conclusions from the propositional case motivated

[2] Note that ordering-search is mainly relevant for directed models, and not for undirected models such as Markov Logic Networks [21] (since the point of ordering-search is to avoid directed cycles).

us to consider this. We now show how to upgrade ordering-search towards learning non-recursive LBNs.

Similar to the case of propositional Bayesian networks, it is easy to learn a non-recursive LBN when an *ordering on the probabilistic predicates* is given. Again the learning task decomposes: we can learn for each probabilistic predicate separately the logical CPD. To learn the logical CPD for a predicate p, we learn a logical probability tree for predicting p. This procedure is applied in turn to each probabilistic predicate p, yielding a set of logical CPDs. As we explain below, the dependency statements can be extracted from these logical CPDs in a post-processing step. Recall from the previous section that in the propositional case some care needs to be taken to ensure that the resulting network is not fully-connected. In our learning algorithm for LBNs, this is accomplished in a quite simple way. This is due to the fact that we use logical probability trees as CPDs, and that learning algorithms for decision trees are typically 'selective' (if we learn a decision tree with as input a number of predicates, then the learned tree will typically only contain tests on some of these predicates). Hence, to learn the tree for a predicate p, we simply supply all predicates preceding p in the ordering as inputs to the learning algorithm, and let the learning algorithm select which of these predicates are really relevant for predicting p. The selectivity of the tree learning algorithm then ensures that we do not obtain a 'fully-connected' LBN.

The above strategy requires an ordering on the probabilistic predicates. When the optimal ordering is not known in advance, we need to search over possible orderings. Obviously exhaustive search is infeasible, so some kind of heuristic search is needed. Like Teyssier and Koller [22] in the propositional case, we essentially perform *hill-climbing through the space of all orderings*. Concretely, we start from a random ordering and compute the score for this ordering (i.e., the score of the LBN learned using this ordering). Then we consider all candidate orderings in the neighbourhood of the initial ordering, and select the ordering with the highest score. Using this ordering as the new ordering, we repeat the same procedure, and so on until we obtain no more improvements. We then use the logical CPDs learned for the final ordering as the final logical CPDs[3]. This algorithm is summarized in Figure 4.

```
% start with a random ordering:
O_current = random ordering on the probabilistic predicates
compute score(O_current)
% search for a better ordering:
repeat until convergence
    for each O_cand ∈ neighbourhood(O_current)
        compute Δscore(O_cand) = score(O_cand) − score(O_current)
    end for
    if max(Δscore(O_cand)) > 0
        O_current = argmax(Δscore(O_cand))
    end if
end repeat
% extract the dependency statements from the logical CPDs learned for the final ordering:
for each probabilistic predicate p
    extract dependency statements from the logical CPD for p learned using O_current
end for
```

Fig. 4 The ordering-search algorithm for learning LBNs.

[3] Since the initial ordering might influence the final result (because the algorithm only converges to a local optimum), it could be useful to perform random restarts, i.e., multiple runs with different initial random orderings. However, in our experiments we found the gain of this to be very small. This not only holds for the above ordering-search algorithm, but also for the structure-search algorithm that we discuss later.

In the above algorithm the neighbourhood of an ordering is defined as the set of orderings that can be obtained by swapping a pair of adjacent predicates in that ordering (this is similar to what is done for propositional Bayesian networks [22]). Note that the size of the neighbourhoods is $n-1$, with n the number of probabilistic predicates (because there are $n-1$ different possibilities for swapping adjacent predicates in an ordering). Since the size of the neighbourhoods is equal to the branching factor of the search, this implies that *the branching factor of ordering-search is linear in the number of probabilistic predicates*. This will turn out to be important in our experiments.

Below we explain how to learn and score logical CPDs (when we know the set of predicates that are used as in input for the logical CPD), and how to extract dependency statements from a logical CPD. At the end, we also discuss some differences between our ordering-search algorithm for LBNs and the algorithm for propositional Bayesian networks.

6.1.3 Learning Logical CPDs

We represent logical CPDs as logical probability trees like the tree in Figure 2 (p. 6). Such trees can be learned using any of the standard probability tree algorithms in TILDE [7]. The only two issues are which scoring criterion to use for the trees, and how to construct the datasets for learning the trees. In this paper we use the Bayesian Information Criterion (BIC) [7, 13] for scoring the trees. We now explain how we construct the datasets.

To learn a logical CPD for a target predicate p_{target} we need a dataset of labelled examples which can be derived from the mega examples in the original dataset. In general, a single mega example can give rise to multiple examples in the dataset for the logical CPD since there can be multiple ground atoms for the predicate p_{target} in the mega example. Concretely, each random variable (ground probabilistic atom) X built from p_{target} in each mega example m leads to one example e in the dataset for the logical CPD. This example e is labelled with the value of X in m and consists of the part of m that is relevant for X.

Example 6 Consider the logical CPD for the probabilistic predicate $difficulty/1$ in the university domain. Note that each random variable for $difficulty/1$ in each mega example corresponds to a particular course. Hence, each course C in each mega example m gives rise to another example e in the dataset for this logical CPD. Such an example e contains all information from the mega example m that is about the course C or about a student linked to C (for instance through the $takes/2$ relation).

Concretely, the simplified mega example given in Example 5 contains two courses and hence leads to two examples in the dataset for the logical CPD for $difficulty/1$. The first example is about course $c1$, is labelled with 'difficulty=mid' and looks as follows.

```
course(c1).
student(s1).          student(s2).
takes(s1,c1).         takes(s2,c1).

ranking(s1)=high      ranking(s2)=mid
rating(c1)=low
grade(s1,c1)=high     grade(s2,c1)=mid
```

The second example is about course $c2$, is labelled with 'difficulty=high' and looks as follows.

```
course(c2).
```

```
student(s2).
takes(s2,c2).

ranking(s2)=mid
rating(c2)=mid
grade(s2,c2)=low
```

□

6.1.4 Extracting Dependency Statements from a Logical CPD

The result of the search over orderings is the set of logical CPDs that was learned for the final ordering. To obtain a complete LBN, we still need to determine the dependency statements. It turns out that the dependency statements can be extracted from the logical CPDs (this is a generalization of the fact that the directed acyclic graph of a Bayesian network can be extracted from the CPDs in that network). Below we explain how to do this for a logical CPD specified as a logical probability tree. To obtain a complete LBN, this procedure needs to be applied to the logical probability tree for each probabilistic predicate.

When extracting dependency statements from a logical probability tree with as target the probabilistic atom p_{target}, we want to find a set of statements that is consistent with the tree (i.e., the tree should never test any probabilistic atom that is not a parent of p_{target} according to the set of statements). We do this by creating a dependency statement for each test on a probabilistic atom in each internal node of the tree. Call the atom that is tested p_{test} and the node N. In the most general case, apart from the test on p_{test}, the node N can contain a number of tests on other probabilistic atoms and a conjunction l of logical literals. We then create a dependency statement of the form $p_{target} \mid p_{test} \leftarrow l, path(N)$, where $path(N)$ is a conjunction of logical literals that describes the path from the root to N. Each node on this path can contribute a number of logical literals to $path(N)$. A succeeded node (i.e., a node for which the succeeding branch of the tree was chosen in the path) contributes all logical literals that it contains. A failed node that does not contain any tests on probabilistic atoms contributes the negation of all its logical literals. A failed node that contains a test on a probabilistic atom does not contribute to the path (letting such a node contribute the negation of its logical literals could be inconsistent since we cannot be sure that the logical literals caused the failure, rather than the probabilistic tests).

After we applied the above procedure to extract all dependency statements, we simplify those statements by removing redundant literals and redundant statements (Section 4.2).

Example 7 Consider the probability tree shown in Figure 2 (p.6). For this tree, p_{target} is $satisfaction(S,C)$. For the root node, p_{test} is $ability(P)$, l is $teaches(P,C)$ and the path is empty. For the internal node below the root to the left, p_{test} is $grade(S,C)$, l is empty and the path is $teaches(P,C)$. For the node below the root to the right, p_{test} is $grade(S,C)$ and l and the path are both empty. The three resulting dependency statements for these nodes are respectively the following.

```
satisfaction(S,C) | ability(P) <- teaches(P,C).
satisfaction(S,C) | grade(S,C) <- teaches(P,C).
satisfaction(S,C) | grade(S,C).
```

The second statement can be removed since it is redundant with respect to the third statement. □

Note that with the above approach we never learn dependency statements with multiple atoms in the body. This is no restriction since we have shown in Section 4.1 that each such LBN is equivalent to an LBN with dependency statements with only one atom in the body, which can be learned. Also note that in the above procedure for extracting the dependency statements from a logical probability tree, the probabilistic atoms in the internal nodes never contribute to $path(N)$ for a node N. The reason for this is that $path(N)$ is the context of the dependency statement and the context can only contain logical literals but no probabilistic atoms. This implies that generally not all independence information specified in a logical probability tree can be captured by the dependency statements. This is not surprising: it also holds in the propositional case (for instance, a CPD under the form of a probability tree can capture context-specific independence while the structure of a Bayesian network cannot [1]).

6.1.5 Differences between Ordering-search for LBNs and Propositional Ordering-search

Two obvious differences between ordering-search for LBNs and ordering-search as proposed by Teyssier and Koller [22] for propositional Bayesian networks are that for LBNs we use orderings on the set of probabilistic predicates (instead of on the set of random variables), and the extraction of the structure from the CPDs is more complex.

A third and more important difference is that we use **logical probability trees** as CPDs whereas Teyssier and Koller use **conditional probability tables**. Recall that for LBNs we cannot use conditional probability tables since they are too restrictive (Section 3.1.2). However, for propositional Bayesian networks it would be possible to use (propositional) probability trees instead of conditional probability tables. Such an approach would have both an advantage and a disadvantage with respect to efficiency:

- *The advantage of using probability trees is that we need to learn fewer CPDs than when using conditional probability tables. This is a consequence of the fact that probability trees are selective but conditional probability tables are not.*
  Suppose that we are given an ordering and a random variable X for which we need to learn the CPD. Using probability trees, we simply learn a CPD with as input all random variables preceding X in the ordering, and we let the decision tree learning algorithm select from all these variables the relevant ones. Using tables, selecting from all the variables the relevant ones is more complex because tables are not selective. Hence, the approach of Teyssier and Koller is to put an upper bound k on the number of inputs for the CPDs. Concretely, they compute the score of all CPDs for X that have at most k random variables (from all the variables preceding X in the ordering) as inputs, and then determine the parents of X as all random variables that are used in the highest scoring CPD. The drawback of this is that there are many such CPDs. Hence this is computationally only feasible for small k (Teyssier and Koller use at most $k = 4$).
- *The disadvantage of using probability trees is that learning a single CPD is computationally less efficient than when using conditional probability tables (with an upper bound on the number of inputs).*
  Using tables with an upper bound k on the number of inputs allows Teyssier and Koller to compute beforehand the sufficient statistics for all CPDs that could ever be needed during the search over orderings. As a consequence, the actual search over orderings becomes very fast. In contrast, when using probability trees it is not efficient to learn all CPDs beforehand. Hence, in our ordering-search algorithm for LBNs we do not learn probability trees beforehand but learn them on the fly as they are needed.

To summarize, using probability trees instead of conditional probability tables has two opposing effects: fewer CPDs need to be learned, but learning a single CPD is less efficient. It is unclear what the combined effect of these two opposing effects is since this has not yet been studied. Although using conditional probability tables in LBNs is not an option (since they are too restrictive), this issue is nevertheless relevant for our work: it implies that it is non-obvious that the efficiency advantage of ordering-search over structure-search that was observed by Teyssier and Koller [22] also holds in the case of LBNs. This is part of our motivation for experimentally comparing ordering-search and structure-search for LBNs in Section 7.

6.2 Structure-search

We now discuss structure-search. First we briefly discuss the propositional case, then we discuss the case of LBNs.

6.2.1 Structure-search for Propositional Bayesian Networks

Structure-search (also known as DAG-search) is the most well-known and most straightforward approach for learning propositional Bayesian networks [13]. It is based on two observations.

- It is relatively easy to learn the CPDs in a Bayesian network if the structure of the network (the directed acyclic graph) is given.
- If the best structure is not known in advance, a search through the space of possible structures needs to be carried out to find the best structure.

6.2.2 Structure-search for Logical Bayesian Networks

The structure-search algorithm for propositional Bayesian networks has already been upgraded to the relational case for several formalisms [10, 12, 15, 16, 19]. The algorithm that we use for LBNs is very similar to these existing upgrades, in specific the algorithm for learning Probabilistic Relational Models [10, 12].

One possible approach to structure-search for LBNs would be to define refinement operators for sets of dependency statements, and use these operators to organize the search. To avoid double computations in the learning process, we would then have to take into account possible redundancy in the dependency statements (see Section 4.2). However, for learning non-recursive models, a more simple approach can be taken: we can search in the space of predicate dependency graphs instead of in the space of dependency statements. Recall from Section 3.2 that a predicate dependency graph specifies for each predicate on which other predicates it depends but not exactly how. Hence, when searching in the space of predicate dependency graphs, redundancy is less of a problem, and hence the search process is easier to implement.

Our structure-search algorithm for LBNs is basically hill-climbing in the space of predicate dependency graphs. To learn the logical CPD for a predicate p, given a predicate dependency graph S, we simply learn a logical probability tree for predicting p with as input all predicates that are parents of p in S. When we found the final predicate dependency graph and the corresponding logical CPDs, we extract the dependency statements from these logical CPDs in exactly the same way as in our ordering-search algorithm (Section 6.1.4). The resulting structure-search algorithm for LBNs is summarized in Figure 5.

```
% start with a random structure:
S_current = random predicate dependency graph
compute score(S_current)
% search for a better structure:
repeat until convergence
    for each S_cand ∈ neighbourhood(S_current)
        compute Δscore(S_cand) = score(S_cand) − score(S_current)
    end for
    if max(Δscore(S_cand)) > 0
        S_current = argmax(Δscore(S_cand))
    end if
end repeat
% extract the dependency statements from the logical CPDs learned for the final structure:
for each probabilistic predicate p
    extract dependency statements from the logical CPD for p learned using S_current
end for
```

Fig. 5 The structure-search algorithm for learning LBNs.

To find the initial predicate dependency graph in the above algorithm, we borrow some elements from our ordering-search algorithm. Specifically, we generate a random initial ordering, learn logical CPDs for this ordering, and extract the predicate dependency graph from these logical CPDs (this amounts to simply checking which predicates are used inside which logical CPDs)[4].

In the above algorithm the neighbourhood of a predicate dependency graph S is defined as the set of all acyclic graphs that can be obtained by adding, deleting or reversing an edge in S. Note that the number of possibilities to add an edge in a predicate dependency graph is in general quadratic in the number of probabilistic predicates. Hence *the branching factor of structure-search is quadratic in the number of probabilistic predicates*. Recall that it was linear for ordering-search (Section 6.1.2, p. 13).

We also implemented an extension to the above structure-search algorithm for LBNs, namely lookahead: with lookahead we not only try adding one edge but also adding two edges with the same head during a single refinement step. The experimental results were rather discouraging, however: lookahead does not significantly improve the quality of the learned models (in terms of test log-likelihood or number of dependency statements), but significantly slows down the algorithm [8]. Hence, we do not consider this extension further.

6.3 Efficiently Implementing the Algorithms

We now briefly show some optimizations that can be used to implement the above algorithms for learning LBNs efficiently, provided that the scoring criterion is decomposable. These optimizations apply equally well to the propositional case as to the case of LBNs. In fact, in the propositional case they are entirely standard [13].

When using a decomposable scoring criterion, the ordering-search and structure-search algorithms can be implemented quite efficiently because decomposability has two beneficial effects on the computation of the score-change for a candidate ordering ($\Delta score(O_{cand})$ in the algorithm of Figure 4) or candidate structure ($\Delta score(S_{cand})$ in Figure 5).

[4] In our experiments, we use the same random initial ordering for ordering-search and structure-search. Hence, both algorithms always start from the same point. This ensures that an experimental comparison of both algorithms evaluates the search process itself and not the starting point of the search.

– *Locality*: the score-change for a candidate ordering/structure only depends on the score of the logical CPDs that are different for the candidate ordering/structure than for the current ordering/structure. For ordering-search and structure-search there are at most two such logical CPDs[5]. Hence, score-changes can be computed quite efficiently.
– *Reusability*: many of the score-changes that are computed during one iteration of the loop (the "repeat" in Figure 4 or 5) are still valid during the next iteration and can be reused. To be precise, a score-change due to a modification m_1 is still valid after a modification m_2 to the current ordering/structure if and only if the set of probabilistic input predicates that is changed by m_1 was not changed by m_2.

7 Experiments

We now experimentally compare the above algorithms for learning LBNs (ordering-search and structure-search). We discuss the datasets, the experimental setup and the results.

7.1 Datasets

We perform experiments on four relational domains: the synthetic university domain used before in our examples, and three real-world datasets (IMDB, UWCSE and WebKB) that are popular benchmarks in the field of statistical relational learning.

For the *synthetic university domain* we generated datasets with a varying number of mega examples from the LBN given in Section 3. The logical part of each mega example was specified by hand (it contains 20 students, 10 courses, 5 professors and their relationships). The probabilistic part of each mega example was constructed by sampling from the given LBN. Each mega example corresponds to 230 random variables. We generated datasets of 5, 10, 15, 31, 62, 125 and 250 mega examples (we refer to these datasets as 'Univ5' to 'Univ250').

The *UWCSE dataset* [21] contains information about 140 graduate students, 52 professors and 132 courses at a computer science department. The dataset consists of five mega examples, each corresponding to a specific research area. Since in this dataset relations are of special importance[6], we incorporate them into the probabilistic model. In LBNs this can be accomplished by simply modelling them as probabilistic predicates. In total, we use three logical predicates ($student/1$, $prof/1$ and $course/1$) and ten probabilistic predicates with the following random variable declarations.

```
random(phase_in_PhD(S)) <- student(S).
random(year_in_PhD(S)) <- student(S).
random(student_nb_publications(S)) <- student(S).
random(position(P)) <- prof(P).
random(prof_nb_publications(P)) <- prof(P).
```

[5] For *ordering-search* there are *exactly two* such logical CPDs since a new candidate ordering is obtained by swapping two adjacent predicates in the current ordering and this influences only the logical CPDs for these two predicates. For *structure-search* there are *at most two* such logical CPDs since a new candidate predicate dependency graph is obtained by adding/deleting/reversing an edge in the current graph, and adding/deleting an edge influences only the logical CPD for the predicate that the edge points to, while reversing an edge influences the logical CPDs for both predicates involved.

[6] For instance, this dataset has been used for supervised learning with the 'advised by' relation as the target [21].

Table 1 Dataset characteristics: total number of random variables, number of mega examples, number of probabilistic predicates.

Dataset	#Variables	#MegaExamples	#ProbPredicates
Synthetic Univ	1150 to 57500	5 to 250	8
IMDB	2852	5	7
UWCSE	9607	5	10
WebKB	78132	4	5

```
random(level(C)) <- course(C).
random(teaches(P,C)) <- prof(P), course(C).
random(assistant(S,C)) <- student(S), course(C).
random(advised_by(S,P)) <- student(S), prof(P).
random(co_author(S,P)) <- student(S), prof(P).
```

Of course, since this is a real-world dataset the true dependency statements are unknown.

The *IMDB dataset* was extracted from the internet movie database (`www.imdb.com`). The dataset that we use (by Mihalkova et al. [18]) contains information about 20 movies, 236 actors and 32 directors. This information is divided over five mega examples. We use three logical predicates ($actor/1$, $director/1$ and $movie/1$) and seven probabilistic predicates with the following random variable declarations.

```
random(gender(A)) <- actor(A).
random(comedy(D)) <- director(D).
random(crime(D)) <- director(D).
random(drama(D)) <- director(D).
random(worked_for(A,D)) <- actor(A), director(D).
random(acts(A,M)) <- actor(A), movie(M).
random(directs(D,M)) <- director(D), movie(M).
```

The *WebKB dataset* is about people (faculty or students), courses and projects at the computer science departments of four universities [4]. The dataset that we use [18] contains information about 746 people, 163 courses and 80 projects, divided over four mega examples (each corresponding to a different university). We use three logical predicates ($person/1$, $project/1$ and $course/1$) and five probabilistic predicates with the following random variable declarations.

```
random(faculty(P)) <- person(P).
random(student(P)) <- person(P).
random(has_project(P,Pr)) <- person(P), project(Pr).
random(assistant(P,C)) <- person(P), course(C).
random(prof(P,C)) <- person(P), course(C).
```

The main characteristics of the above datasets are summarized in Table 1.

7.2 Experimental Setup

For all experiments we performed five-fold cross validation (except for WebKB we performed four-fold cross validation since this dataset contains only four mega examples). For the synthetic university domain, mega examples were divided over equal-sized folds randomly. For the real world datasets, each fold corresponds to one mega example. We report

the average results over the folds and use two-tailed paired t-tests (with α=0.05) to assess the significance of differences between two algorithms.

We use four evaluation criteria: *normalized test log-likelihood* (the log-likelihood on the test data divided by the number of mega examples), *normalized train score* (the score on the training data divided by the number of mega examples; while not important in itself it can give some insight into the degree of overfitting of an algorithm), *number of dependency statements learned* (smaller is usually better because of ease of interpretation) and *running time*.

For the synthetic university domain we know the true LBN that generated the data. Hence, we can use as a fifth evaluation criterion the degree to which a learned LBN matches this true LBN. A simple measure for this would be the number of dependency statements that the true LBN has in common with the learned LBN. However, the problem with this is that generally it is even in theory not possible to learn the true direction of each dependency statement from data[7]. Hence, instead we measure the maximum overlap between the true LBN and the Markov equivalence class $\mathcal{C}_{learned}$ of the learned LBN (we look for the LBN in $\mathcal{C}_{learned}$ that has the most directions in common with the true LBN). We refer to this as the *number of correct dependencies learned*.

For all evaluation criteria we report the results for the two algorithms. For test log-likelihood and train score we additionally report the results for the 'empty LBN' as a baseline. With an 'empty LBN' we mean an LBN with no dependency statements, this is the LBN according to which all random variables are independent.

7.3 Experimental Results

We now report our experimental results. First we focus on the comparison of the two algorithms, ordering-search (OS) and structure-search (SS). Then we analyze the running times of both algorithms in more detail. Finally we briefly show some of the dependencies learned on the real-world datasets.

7.3.1 Ordering-Search versus Structure-Search

Our experimental results for OS and SS are given in Table 2 and summarized in Table 3. An entry in the latter table (for a particular evaluation criterion and dataset) has the following meaning: if one of the two algorithms is significantly better than the other we show the best algorithm; if there is no statistically significant difference between the two algorithms we fill in "/". Note that for the real-world datasets we cannot measure the number of correct dependencies learned since we do not know the true LBN for these datasets. For the synthetic university domain we also plotted the results as a function of dataset size in Figure 6.

In terms of quality of the learned LBNs, the main conclusion from our results is that OS and SS are competitive with each other.

– For none of the datasets there is a significant difference in **test log-likelihood** between OS and SS (see Table 3). In terms of train score, SS performs significantly better than

[7] With the direction of a dependency statement we mean which atom is in the head and which in the body. The reason why this cannot always be learned from data is similar to the reason why the true direction of an edge in a Bayesian network cannot always be learned (each Bayesian network has a *Markov equivalence class*, which is a set of networks that all have the same score but different directions for some of the edges [2].

Table 2 Detailed experimental results. For each dataset the best results are shown in bold.

Dataset	Method	LogLik(Test)	Score(Train)	#Statements	#CorrectDepend	Time
Univ5	OS	-1.3789	-1.3485	**9.0**	3.4	**35s**
Univ5	SS	**-1.3750**	**-1.3365**	9.8	**4.4**	137s
Univ5	empty	-1.4799	-1.4989	-	-	-
Univ10	OS	-1.3669	-1.3524	9.8	4.6	**42s**
Univ10	SS	**-1.3461**	**-1.3410**	**9.4**	**5.0**	134s
Univ10	empty	-1.4722	-1.4880	-	-	-
Univ15	OS	-1.3444	-1.3415	**8.6**	4.4	**51s**
Univ15	SS	**-1.3328**	**-1.3305**	9.6	**5.0**	164s
Univ15	empty	-1.4697	-1.4792	-	-	-
Univ31	OS	-1.3083	-1.3135	**8.8**	5.6	**55s**
Univ31	SS	**-1.3023**	**-1.3060**	9.6	**6.6**	168s
Univ31	empty	-1.4575	-1.4647	-	-	-
Univ62	OS	-1.3051	-1.3097	11.2	5.8	**78s**
Univ62	SS	**-1.2973**	**-1.3012**	**10.4**	**6.6**	262s
Univ62	empty	-1.4554	-1.4595	-	-	-
Univ125	OS	-1.2905	-1.2959	8.8	5.8	**120s**
Univ125	SS	**-1.2828**	**-1.2861**	**8.2**	**6.8**	407s
Univ125	empty	-1.4562	-1.4586	-	-	-
Univ250	OS	-1.3001	-1.3035	9.8	5.6	**194s**
Univ250	SS	**-1.2894**	**-1.2913**	**7.6**	**6.8**	586s
Univ250	empty	-1.4561	-1.4573	-	-	-
IMDB	OS	**-0.7975**	**-0.7620**	7.8	-	**95s**
IMDB	SS	-0.8782	-0.7785	**7.2**	-	378s
IMDB	empty	-0.9265	-0.8975	-	-	-
UWCSE	OS	-0.4288	-0.3539	15.2	-	**135s**
UWCSE	SS	**-0.4160**	**-0.3489**	**14.6**	-	535s
UWCSE	empty	-0.4631	-0.3961	-	-	-
WebKB	OS	**-0.0702**	-0.0694	**7.3**	-	**251s**
WebKB	SS	-0.0709	**-0.0693**	**7.3**	-	565s
WebKB	empty	-0.0808	-0.0791	-	-	-

Table 3 Significance of differences between results for OS and SS. Both OS and SS have significantly better test log-likelihood and train score than the empty LBN in all cases (this is not shown in the table).

Dataset (#Vars)	LogLik(Test)	Score(Train)	#Statements	#CorrectDepend	Time
Univ5	/	/	/	/	OS
Univ10	/	/	/	/	OS
Univ15	/	/	/	/	OS
Univ31	/	/	/	/	OS
Univ62	/	/	/	/	OS
Univ125	/	/	/	SS	OS
Univ250	/	/	/	SS	OS
IMDB	/	/	/	-	OS
UWCSE	/	SS	/	-	OS
WebKB	/	/	/	-	OS

OS in one case (the UWCSE dataset), but this difference on training data does not carry over to the test data. As expected, OS and SS always perform better than the empty LBN (in terms of test log-likelihood and train score).

The evolution of test log-likelihood and train score as a function of the dataset size for the synthetic university domain (Figure 6) is as expected: both improve rapidly when initially increasing the dataset size but this improvement slows down when moving to bigger datasets, and likelihood and score seem to saturate. The figure also shows that

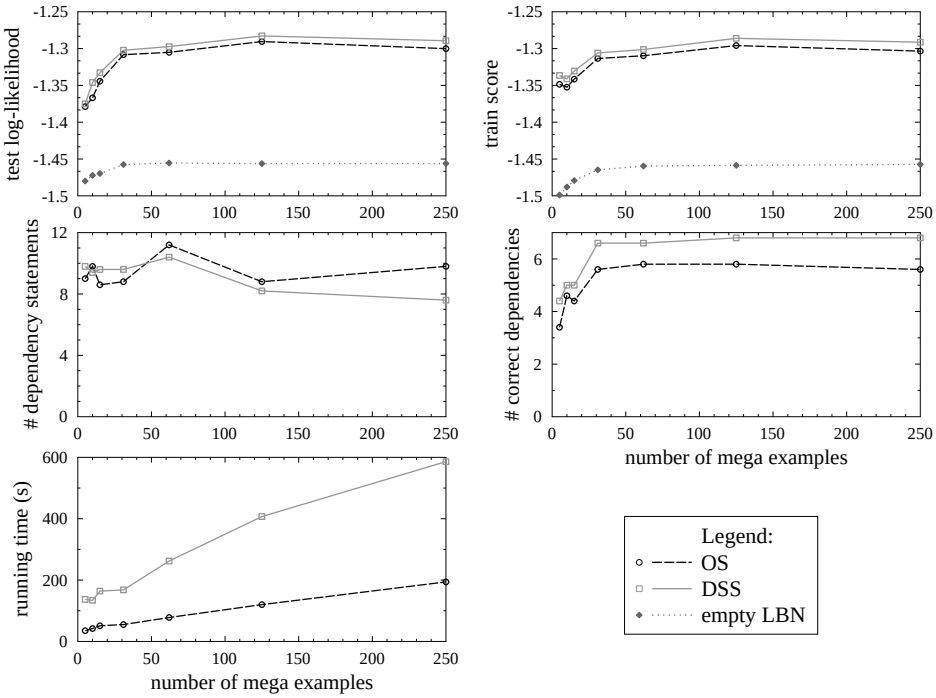

Fig. 6 Results for OS and SS on synthetic university datasets of varying size. For train score and test log-likelihood we also show the results for the empty LBN.

the differences between OS and SS are very small as compared to the differences with the empty LBN.

– For none of the datasets there is a significant difference in the **number of dependency statements** learned by OS and SS (see Table 3). Also, the evolution of the number of dependency statements as a function of the dataset size (Figure 6) does not show any clear trends.
– Since for the synthetic university domain we know the true LBN, we can measure the **number of correct dependencies** learned by OS and SS. SS learns significantly more correct dependencies in two cases while the opposite never occurs. The evolution of the number of correct dependencies learned as a function of the dataset size (Figure 6) is as expected: it increases rapidly when initially increasing the dataset size and then saturates. For sufficiently big datasets, SS learns nearly all seven true dependencies while OS does slightly worse with on average 5.7 true dependencies. We had a closer look at the experiments with the biggest dataset sizes in which not all seven true dependencies were learned and found that the problem was always in having the wrong direction for a dependency but never in 'missing' a dependency (i.e., all true 'undirected' dependencies are always learned but sometimes a dependency is learned in the wrong direction even though the two directions are not Markov equivalent). This can also explain why the above differences in the number of correct dependencies learned between OS and SS do not lead to significant differences in test log-likelihood.

The results for **running time** show that OS is always significantly faster than SS. On the synthetic university domain, running time behaves linearly in the dataset size for both

OS and SS (Figure 6), but nevertheless running time is always significantly smaller for OS than for SS, with differences between a factor 3.0 and 3.9. Also on the real-world datasets the running time is always significantly smaller for OS, with differences between a factor 2.3 and 4.0. We analyze the reason for these differences between OS and SS in more detail in the next section.

Since OS is competitive with SS in terms of quality of the learned LBNs, and OS is significantly faster, we conclude that overall OS is preferable to SS for learning non-recursive LBNs.

7.3.2 Analysis of Running Times

In this section we analyze the running time of both algorithms in more detail by decomposing it into the running times of the main different steps in the algorithms. Such an analysis has not been made before for ordering-search (also not in the propositional case).

The total running time T_{total} of the ordering-search and structure-search algorithms can be decomposed as follows[8]

$$T_{total} = T_{init} + T_{first} + T_{rest},$$

where T_{init} denotes the initialization time (the time for learning and scoring all logical CPDs for the initial ordering/structure), T_{first} denotes the time for the first iteration (i.e., the first execution of the repeat loop of the algorithms) and T_{rest} denotes the time for all other iterations. The reason for considering the first iteration separately is that it typically takes a lot longer than any of the other iterations since all the score-changes needed in the first iteration effectively have to be computed, while in the next iterations most of them can be reused without extra computation (see Section 6.3). Let I denote the number of iterations not including the first one. If we define the average time per iteration (not including the first one) as $T_{avg} = T_{rest}/I$, we can rewrite the total running time as follows.

$$T_{total} = T_{init} + T_{first} + T_{avg} \times I$$

Our experimental results for each of the above measures are shown in Table 4. Note that T_{init} is the same for both algorithms. Hence, below we only discuss T_{total}, T_{first}, T_{avg} and I.

Recall from the previous section that the total running time, T_{total}, was always significantly lower for OS than for SS with differences being between a factor 2.3 and 4.0. This can be explained as follows.

- The **time for the first iteration**, T_{first}, is always significantly lower for OS than for SS. This was expected since in the first iteration all elements of the neighbourhood of the initial ordering/structure have to be scored and the size of the neighbourhood, and hence *the branching factor of the search, is smaller for OS than for SS* (linear in the number of probabilistic predicates for OS but quadratic for SS, see Sections 6.1 and 6.2). In our experiments the difference in T_{first} between OS and SS goes from a factor 1.7 to 4.1.
- The **average time per iteration** (not including the first one), T_{avg}, is also always significantly lower for OS than for SS. This was expected for the same reasons as for T_{first} above. In our experiments the difference in T_{avg} between OS and SS goes from a factor 1.8 to 3.8.

[8] The time needed for the final step of extracting the dependency statements from the logical CPDs can be ignored since it is very small (it does not depend on the dataset size).

Table 4 Detailed timings.

Dataset	Method	T_{total}	T_{init}	T_{first}	T_{rest}	I	T_{avg}
Univ5	OS	35s	7s	15s	13s	2.4	6s
Univ5	SS	137s	7s	60s	70s	5.8	12s
Univ10	OS	42s	7s	20s	15s	2.0	7s
Univ10	SS	134s	7s	69s	57s	4.2	13s
Univ15	OS	51s	8s	18s	24s	3.0	8s
Univ15	SS	164s	8s	73s	83s	5.8	14s
Univ31	OS	55s	10s	24s	20s	2.2	9s
Univ31	SS	168s	10s	85s	73s	4.6	16s
Univ62	OS	78s	13s	31s	34s	2.8	12s
Univ62	SS	262s	13s	101s	147s	6.2	24s
Univ125	OS	120s	18s	45s	57s	3.2	19s
Univ125	SS	407s	18s	161s	228s	4.4	56s
Univ250	OS	194s	33s	89s	71s	2.2	33s
Univ250	SS	586s	33s	246s	307s	4.8	66s
IMDB	OS	95s	25s	52s	18s	2.0	9s
IMDB	SS	378s	25s	214s	139s	4.0	34s
UWCSE	OS	135s	27s	61s	46s	2.4	20s
UWCSE	SS	535s	27s	279s	229s	4.6	52s
WebKB	OS	251s	60s	58s	133s	4.0	33s
WebKB	SS	565s	60s	98s	407s	5.8	71s

– The conclusions about the **number of iterations** I are less clear. In six cases I is significantly lower for OS than for SS, while in the remaining four cases there is no significant difference.

We conclude that the main reason why OS is faster than SS, is that OS has a smaller branching factor (linear in the number of probabilistic predicates, while it is quadratic for SS).

7.3.3 Learned Dependencies on Real-World Datasets

For each real-world dataset we investigated the learned dependency statements. Most of the dependencies that were frequently learned (for both algorithms and the various folds in the cross validation) confirm our intuitions about these datasets (the true dependencies are of course unknown for these datasets). Some examples of such dependencies, and their common sense interpretations obtained by investigating the corresponding logical CPDs, are the following.

– IMDB dataset:
  – $comedy(D)$ depends on $drama(D)$:
    The corresponding logical CPD specifies that, if a director is into drama, he is less likely to be into comedy.
  – $acts(A, M)$ depends on $worked\_for(A, D)$ and $directs(D, M)$:
    An actor is likely to be in a movie if he worked for a director who directs that movie.
  – $gender(A)$ depends on $worked\_for(A, D)$ and $drama(D)$:
    An actor is more likely to be male if he worked for a director who is into drama.
– UWCSE dataset:
  – $student\_nb\_publications(S)$ depends on $year\_in\_PhD(S)$:
    A student is more likely to have many publications if he is in a higher year.
  – $prof\_nb\_publications(P)$ depends on $advised\_by(S, P)$:
    A professor is more likely to have many publications if he advises more students.

  - $assistant(S, C)$ depends on $advised\_by(S, P)$ and $teaches(P, C)$:
    A student is more likely to be the teaching assistant for a course if he is advised by a professor who teaches that course.
  - $assistant(S, C)$ depends on $level(C)$:
    A student is more likely to be the teaching assistant for a course if the level of the course is lower (the possible levels are undergraduate, advanced undergraduate and graduate). This can be explained by the fact that the lower the course level, the more teaching assistants a course has on average (this is indeed the case for the courses in the dataset).
- WebKB dataset:
  - $faculty(P)$ depends on $student(P)$:
    If a person is not a student, then he is very likely to be faculty.
  - $student(P)$ depends on $has\_project(P, Proj)$:
    The more projects a person has, the less likely he is to be a student (and hence the more likely he is to be faculty).

8 Learning Recursive Directed Probabilistic Logical Models

In this paper we focussed on learning non-recursive models. We now briefly discuss some of the approaches that can be taken if one presumes that the data contains recursive dependencies. A major concern when learning recursive directed models is to ensure that, although the model is cyclic at the predicate level, it is always acyclic at the ground level. We can distinguish two scenarios depending on how much prior knowledge is available about the presumed recursive dependencies.

8.1 Prior Knowledge about Guaranteed Acyclic Relationships

When one knows that there are recursive dependencies and has some prior knowledge or assumptions about which relations (modelled as logical predicates) determine the recursive dependencies, learning can actually be very similar to learning in the non-recursive case. Getoor et al. [12] took this perspective when developing the learning algorithm for Probabilistic Relational Models. To accommodate for dependencies that are cyclic at the predicate level but acyclic at the ground level they let the user define a *guaranteed acyclic relationship (GAR)*. For instance, to allow that some properties of a person depend on these properties for his ancestors, the ancestor relation should be defined as the GAR. Similarly, to allow that some properties of a paper (for instance the topic or length) depend on these properties for the papers published earlier by the same author, the published-earlier relation should be defined as the GAR. Getoor et al. then apply structure-search and use the information about the GAR during the acyclicity checks to deduce that certain cycles at the predicate level are legal [12].

The algorithm of Getoor et al. is very similar to the structure-search algorithm used for LBNs in this paper. Hence the approach of using a GAR can be directly applied to our structure-search algorithm as well. Moreover, the same approach can also be applied to our ordering-search algorithm for LBNs. This actually requires no changes to our ordering-search algorithm but only requires to adapt the declarative language bias for the logical CPDs. When learning the logical CPD for a predicate p, the probabilistic input predicates for the CPD would be

- all predicates that precede p in the current ordering, with the restriction that p must not depend on these predicates through the inverse of the GAR (for instance, if the GAR is the published-earlier relation, then the inverse is the published-later relation),
- p itself and all predicates that follow p in the ordering, with the restriction that p can only depend on these predicates through the GAR.

8.2 No Prior Knowledge

When one does not have enough prior knowledge about the data to find a GAR, more complicated approaches need to be taken. When applying structure-search, one can use the techniques developed for Bayesian Logic Programs [15, 16]. The idea is that the learning algorithm searches itself for the logical relations that determine the recursion, and that acyclicity of a candidate model is checked at the ground level for each example (i.e., acyclicity is checked for the induced Bayesian network for each example). The main drawback of this approach is that the acyclicity checks can be computationally very expensive since the cost depends on the number of examples and the size of the examples. This is different from the non-recursive case where acyclicity of each candidate model only needs to be checked once at the predicate level, and hence the cost is independent of the number or size of the examples.

An alternative approach is to use *generalized ordering-search* [20], an algorithm that we developed especially to learn recursive dependencies. In generalized ordering-search we use orderings on ground probabilistic atoms (instead of on predicates, as we do in this paper). In principle, generalized ordering-search can also be used to learn *non-recursive* LBNs but in this respect it has a number of disadvantages as compared to the algorithm proposed in this paper. One disadvantage is that it does not learn an LBN 'in closed form': it does not learn a set of dependency statements but rather a procedural description of how to determine the induced Bayesian network given any possible interpretation of the logical predicates. Another disadvantage is that it deviates quite far from the propositional ordering-search algorithm. For instance, when applied on propositional data generalized-ordering search does not correspond to the original propositional ordering-search algorithm, while this is the case for the algorithm in this paper. This might make generalized-ordering search harder to understand for people familiar with the propositional ordering-search algorithm.

9 Conclusion

We upgraded the ordering-search algorithm for Bayesian networks towards non-recursive directed probabilistic logical models. We experimentally compared the resulting algorithm with a more traditional structure-search algorithm on four relational domains. The results show that ordering-search is competitive with structure-search in terms of quality of the learned models. Also, ordering-search is significantly faster than structure-search due to a smaller branching factor. We conclude that ordering-search is a good alternative to structure-search for learning non-recursive directed probabilistic logical models.

Acknowledgements Daan Fierens is supported by the Institute for the Promotion of Innovation by Science and Technology in Flanders (IWT Vlaanderen). Jan Ramon and Hendrik Blockeel are post-doctoral fellows of the Research Foundation-Flanders (FWO Vlaanderen). This research is also supported by GOA 2003/08 "Inductive Knowledge Bases" and GOA/08/008 "Probabilistic Logic Learning".

References

1. Boutilier, C., Friedman, N., Goldszmidt, M., Koller, D.: Context-specific independence in Bayesian Networks. In: Proceedings of the 12th Conference on Uncertainty in Artificial Intelligence, pp. 115–123 (1996)
2. Chickering, D.: Learning equivalence classes of Bayesian-network structures. Journal of Machine Learning Research **2**, 445–498 (2002)
3. Cooper, G., Herskovits, E.: A Bayesian method for the induction of probabilistic networks from data. Machine Learning **9**, 309–347 (1992)
4. Craven, M., DiPasquo, D., Freitag, D., McCallum, A., Mitchell, T.M., Nigam, K., Slattery, S.: Learning to extract symbolic knowledge from the world wide web. In: Proceedings of the 15th National Conference on Artificial Intelligence, pp. 509–516 (1998)
5. De Raedt, L., Kersting, K.: Probabilistic inductive logic programming. In: Proceedings of the 15th International Conference on Algorithmic Learning Theory, *Lecture Notes in Computer Science*, vol. 3244, pp. 19–36. Springer (2004)
6. Fierens, D., Blockeel, H., Bruynooghe, M., Ramon, J.: Logical Bayesian networks and their relation to other probabilistic logical models. In: Proceedings of the 15th International Conference on Inductive Logic Programming, *Lecture Notes in Computer Science*, vol. 3625, pp. 121–135. Springer (2005)
7. Fierens, D., Ramon, J., Blockeel, H., Bruynooghe, M.: A comparison of pruning criteria for probability trees. Tech. Rep. CW 488, Department of Computer Science, Katholieke Universiteit Leuven (2007). `http://www.cs.kuleuven.ac.be/publicaties/rapporten/cw/CW488.abs.html`
8. Fierens, D., Ramon, J., Bruynooghe, M., Blockeel, H.: Learning directed probabilistic logical models from relational data. Tech. Rep. CW 490, Department of Computer Science, Katholieke Universiteit Leuven (2007) `http://www.cs.kuleuven.ac.be/publicaties/rapporten/cw/CW490.abs.html`
9. Fierens, D., Ramon, J., Bruynooghe, M., Blockeel, H.: Learning directed probabilistic logical models: Ordering-search versus structure-search. In: Proceedings of 18th European Conference on Machine Learning, *Lecture Notes in Artificial Intelligence*, vol. 4701, pp. 567–574. Springer (2007)
10. Friedman, N., Getoor, L., Koller, D., Pfeffer, A.: Learning probabilistic relational models. In: Proceedings of the 16th International Joint Conference on Artificial Intelligence, pp. 1300–1307. Morgan Kaufmann, Stockholm, Sweden (1999)
11. Friedman, N., Goldszmidt, M.: Learning Bayesian networks with local structure. In: M. Jordan (ed.) Learning in Graphical Models, pp. 421–459. Kluwer Academic Publishers (1998)
12. Getoor, L., Friedman, N., Koller, D., Pfeffer, A.: Learning Probabilistic Relational Models. In: S. Dzeroski, N. Lavrac (eds.) Relational Data Mining, pp. 307–334. Springer-Verlag (2001)
13. Heckerman, D., Geiger, D., Chickering, D.: Learning Bayesian networks: The combination of knowledge and statistical data. Machine Learning **20**, 197–243 (1995)
14. Jaeger, M.: Parameter learning for relational Bayesian networks. In: Proceedings of the 24th International Conference on Machine Learning (2007)
15. Kersting, K., De Raedt, L.: Towards combining inductive logic programming and Bayesian networks. In: Proceedings of the 11th International Conference on Inductive Logic Programming, *Lecture Notes in Computer Science*, vol. 2157, pp. 118–131. Springer-Verlag (2001)
16. Kersting, K., De Raedt, L.: Basic principles of learning Bayesian logic programs. Tech. Rep. 174, Institute for Computer Science, University of Freiburg, Germany (2002)
17. Lloyd, J.: Foundations of Logic Programming, 2nd edn. Springer-Verlag (1987)
18. Mihalkova, L., Huynh, T., Mooney, R.: Mapping and revising Markov logic networks for transfer learning. In: Proceedings of the 22nd Conference on Artificial Intelligence (2007)
19. Natarajan, S., Wong, W., Tadepalli, P.: Structure refinement in First Order Conditional Influence Language. In: Proceedings of the ICML workshop on Open Problems in Statistical Relational Learning (2006)
20. Ramon, J., Croonenborghs, T., Fierens, D., Blockeel, H., Bruynooghe, M.: Generalized ordering-search for learning directed probabilistic logical models. Machine Learning **70**(2-3), 169–491 (2008)
21. Richardson, M., Domingos, P.: Markov logic networks. Machine Learning **62**(1–2), 107–136 (2006)
22. Teyssier, M., Koller, D.: Ordering-based search: A simple and effective algorithm for learning Bayesian networks. In: Proceedings of the 21st conference on Uncertainty in Artificial Intelligence, pp. 584–590. AUAI Press (2005)
23. Van Assche, A., Vens, C., Blockeel, H., Džeroski, S.: First order random forests: Learning relational classifiers with complex aggregates. Machine Learning **64**(1-3), 149–182 (2006)

A Proofs for Section 4

In this appendix we prove the theorems given in Section 4.

A.1 Decomposing Dependency Statements

We now prove Theorem 1 (p. 9) about the decomposition of dependency statements with multiple atoms in the body. As explained in Section 4.1, this decomposition can be accomplished in two steps. Since the correctness of the second step (replacing $random/1$ atoms by their definitions) is straightforward, we only prove the correctness of the first step.

Let $\mathcal{L}_1$ be an LBN, and let D be a dependency statement in $\mathcal{L}_1$ of the form $a \mid a_1, \ldots a_n \leftarrow c$, with $n \geq 2$. Let $\mathcal{L}_2$ be the LBN obtained by replacing D in $\mathcal{L}_1$ by the following set of decomposed dependency statements.

$$\begin{cases} a \mid a_1 \leftarrow c, random(a_2), \ldots, random(a_n). \\ \qquad\qquad\qquad\qquad\qquad\qquad \vdots \\ a \mid a_i \leftarrow c, random(a_1), \ldots, random(a_{i-1}), random(a_{i+1}), \ldots, random(a_n). \\ \qquad\qquad\qquad\qquad\qquad\qquad \vdots \\ a \mid a_n \leftarrow c, random(a_1), \ldots, random(a_{n-1}). \end{cases}$$

Call this set $\mathcal{D}_{decomp}$. We need to prove that $\mathcal{L}_1$ and $\mathcal{L}_2$ are equivalent. This means that, for any interpretation of the logical predicates, the set of random variables, the parent relation and the CPDs are the same for $\mathcal{L}_1$ as for $\mathcal{L}_2$. Note that

- The set of random variables is the same since it only depends on the random variable declarations and these are common to both LBNs.
- A random variable a_{par} is a parent of a_{child} only if it is 'caused' to be a parent by some dependency statement. Hence, proving that the parent relation is the same for both LBNs requires proving that the original dependency statement D causes a_{par} to be a parent of a_{child} if and only if some decomposed dependency statement in $\mathcal{D}_{decomp}$ causes this.
- If the parent relation is the same, it follows that the CPDs are also the same (since, given the parent relation, the CPDs only depend on the logical CPDs and these are common to both LBNs).

The above means that we need to prove that the original dependency statement D causes a_{par} to be a parent of a_{child} if and only if some decomposed dependency statement in $\mathcal{D}_{decomp}$ causes this. We now prove this in the two directions. We use $r(.)$ as shorthand notation for $random(.)$.

- Assume that D causes a_{par} to be a parent of a_{child}. This implies that there exists a grounding substitution θ such that $D\theta$ is of the form $a_{child} \mid a'_1, \ldots, a'_{i-1}, a_{par}, a'_{i+1}, \ldots, a'_n \leftarrow c'$ for which c' is true and for which $r(.)$ is true for all ground probabilistic atoms in the head and body. Hence, in the ground statement $a_{child} \mid a_{par} \leftarrow c', r(a'_1), \ldots, r(a'_{i-1}), r(a'_{i+1}), \ldots, r(a'_n)$ the context is true and $r(.)$ is true for the ground probabilistic atoms in the head and body. This ground statement is an instance of the dependency statement $a \mid a_i \leftarrow c, r(a_1), \ldots, r(a_{i-1}), r(a_{i+1}), \ldots, r(a_n)$ under the substitution θ, which is in D_{decomp}. Hence D_{decomp} causes a_{par} to be a parent of a_{child}.
- Assume that a dependency statement D_{decomp} in $\mathcal{D}_{decomp}$ causes a_{par} to be a parent of a_{child} and that D_{decomp} is of the form $a \mid a_i \leftarrow c, r(a_1), \ldots, r(a_{i-1}), r(a_{i+1}), \ldots, r(a_n)$. This implies that there exists a grounding substitution θ such that $D_{decomp}\theta$ is of the form $a_{child} \mid a_{par} \leftarrow c', r(a'_1), \ldots, r(a'_{i-1}), r(a'_{i+1}), \ldots, r(a'_n)$ for which $r(a_{child})$ is true, $r(a_{par})$ is true and the context $c', r(a'_1), \ldots, r(a'_{i-1}), r(a'_{i+1}), \ldots, r(a'_n)$ is true. Hence, in the ground statement $a_{child} \mid a'_1, \ldots, a'_{i-1}, a_{par}, a'_{i+1}, \ldots, a'_n \leftarrow c'$, the context is true and $r(.)$ is true for all ground probabilistic atoms in the head and body. This ground statement is an instance of the dependency statement D under the substitution θ. Hence D causes a_{par} to be a parent of a_{child}.

A.2 Removing Redundancy in Dependency Statements

We now prove Theorem 2 (p. 11) about removing redundancy in dependency statements. First we prove that removing a redundant dependency statement from an LBN yields an equivalent LBN. Next we prove that the same holds for removing a redundant context literal in a dependency statement. We again use $r(.)$ as shorthand notation for $random(.)$.

– Redundant dependency statement:
Consider an LBN that contains a dependency statement of the form

$$a \mid a_1, \ldots, a_n \leftarrow c_1$$

which is redundant with respect to another dependency statement in the LBN with the same head and body but context c_2. This first dependency statement (with context c_1) fires if the conjunction $r(a), r(a_1), \ldots, r(a_n), c_1$ is true. According to the definition of redundancy, this implies that c_2 is true, and hence also the conjunction $r(a), r(a_1), \ldots, r(a_n), c_2$ is true. This implies that the second dependency statement fires. This proves that, whenever the first dependency statement fires, the second fires as well. Since both statements have the same head and body, this makes the first statement obsolete. Hence removing the first dependency statement from the LBN yields an equivalent LBN.
– Redundant literal:
Consider a dependency statement of the form

$$a \mid a_1, \ldots, a_n \leftarrow c, l$$

with c a conjunction of logical literals and l a logical literal. Suppose that l is redundant in this dependency statement. We refer to the dependency statement obtained by removing l from the context as the 'reduced' statement. This reduced statement fires if the conjunction $r(a), r(a_1), \ldots, r(a_n), c$ is true. According to the definition of redundancy, this implies that l is true, and hence also the conjunction $r(a), r(a_1), \ldots, r(a_n), c, l$ is true, and the original statement fires too. This shows that if the reduced statement fires, then the original statement fires too. Obviously, if the original statement fires, then the reduced statement fires too (since the context of the reduced statement is a subset of the context of the original statement). Hence, we conclude that the reduced statement fires if and only if the original statements fires. Since both statements also have the same head and body, they are equivalent. Hence removing the redundant literal from the original statement yields an equivalent LBN.