

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

Daan Fierens

Jan Ramon

Maurice Bruynooghe

Hendrik Blockeel

Report CW 490, May 2007



Katholieke Universiteit Leuven

Department of Computer Science

Celestijnenlaan 200A – B-3001 Heverlee (Belgium)

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

Daan Fierens

Jan Ramon

Maurice Bruynooghe

Hendrik Blockeel

Report CW490, May 2007

Department of Computer Science, K.U.Leuven

Abstract

There is an increasing interest in upgrading Bayesian networks to the relational case, resulting in so-called directed probabilistic logical models. In this paper we discuss how to learn non-recursive directed probabilistic logical models from relational data. This problem has already been tackled before by upgrading the structure-search algorithm for learning Bayesian networks. In this paper we propose to upgrade another algorithm, namely ordering-search, since for Bayesian networks this was found to work better than structure-search. We experimentally compare the two upgraded algorithms on two relational domains.

Keywords : Statistical Relational Learning, Probabilistic Logical Models, Inductive Logic Programming, Bayesian Networks, Probability Trees, Dstructure Learning

CR Subject Classification : I.2.6

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 describing 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 [9, 12, 15]. Most of these algorithms are essentially upgrades of the traditional *structure-search* algorithm for Bayesian networks [10]. In 2005, an alternative algorithm for learning Bayesian networks was introduced: *ordering-search*. This was found to perform at least as good as structure-search while usually being faster. This motivates us to investigate how ordering-search can be upgraded to the relational case.

The contributions of this work are two-fold. First, we upgrade the ordering-search algorithm towards learning non-recursive directed probabilistic logical models. Second, we experimentally compare the resulting algorithm with the upgraded structure-search algorithm on two relational domains. We use the formalism Logical Bayesian Networks [5] but the proposed approach is also valid for related formalisms such as Probabilistic Relational Models [8, 9], Bayesian Logic Programs [12, 13] and Relational Bayesian Networks [11]. Part of this work has been published before [6].

This paper is structured as follows. We review Logical Bayesian Networks in Section 2. We discuss the setting of learning non-recursive Logical Bayesian Networks in Section 3 and various learning algorithms in Section 4. We experimentally compare these algorithms in Section 5. In Section 6 we briefly discuss learning in the recursive case. Finally, in Section 7 we conclude.

2 Logical Bayesian Networks

We first discuss some preliminaries. Then we review Logical Bayesian Networks by means of an example. Then we briefly discuss the existence of syntactic variants of Logical Bayesian Networks. Finally, we define the notion of non-recursive Logical Bayesian Networks.

2.1 Preliminaries: Bayesian Networks and Logic Programming

A *Bayesian network* is a compact specification of a joint probability distribution on a set of random variables under the form of a directed acyclic graph (the *structure*) and a set of conditional probability distributions (CPDs). When learning 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 [10]).

Logical Bayesian Networks share some terminology with 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, for example *student*/1. Arguments can be constants (denoted by lower-case symbols), variables (denoted by upper-case symbols) or compound objects (not considered further). An *atom* is a predicate together with the right number of arguments, for example *student*(*tim*) or *student*(S). A *literal* is an atom or a negated atom, for example *not*(*student*(*john*)). A *ground atom or literal* is

an atom or literal that does not contain any variables as arguments. 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 (for example the considered set of people).

2.2 Logical Bayesian Networks: Example

A Logical Bayesian Network or LBN [5] is essentially a specification of a Bayesian network conditioned on some logical input predicates describing the domain of discourse. For instance, when modelling the well-known ‘university’ domain [9], 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.

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 clauses 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.

The Structure of the Induced Bayesian Networks 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 clauses determine the dependencies (edges). Together, this fully determines the structure of the induced Bayesian network. We now illustrate this with an example.

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)) <- student(S),course(C),takes(S,C).
random(satisfaction(S,C)) <- student(S),course(C),takes(S,C).
```

In these clauses *random/1* is a special predicate. Informally, the first clause, for instance, should be read as “*intelligence(S)* is a random variable if *S* is a student”. Formally, in the induced Bayesian network, there is a node for each ground probabilistic atom *p* for which *random(p)* holds.

The dependency clauses 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 clause 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 ”. There are also more complex clauses such as the last clause, which should be read as “the popularity of a professor P depends on the rating of a course C if P teaches C ”. In this clause, $popularity(P)$ is called the head, $rating(C)$ the body and $teaches(P, C)$ the context. Formally, in the induced Bayesian network there is an edge from a node p_{parent} to a node p_{child} if there is a ground instance of a dependency clause with p_{child} in the head, with p_{parent} in the body, with true context and with $random(.)$ true for each probabilistic atom in the head and body.

Quantifying the Dependencies To quantify the dependencies specified by the dependency clauses, 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.

In this work 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 [11, 13, 12, 15]). 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 for the tree for a probabilistic atom p_{target} to be consistent with the dependency clauses, the tree can of course only test on probabilistic atoms that are parents of p_{target} according to the dependency clauses. An example of a tree for $satisfaction(S, C)$ is shown in Figure 1. Recall that $satisfaction(S, C)$ depends on $grade(S, C)$ and $ability(P)$ where P teaches C .

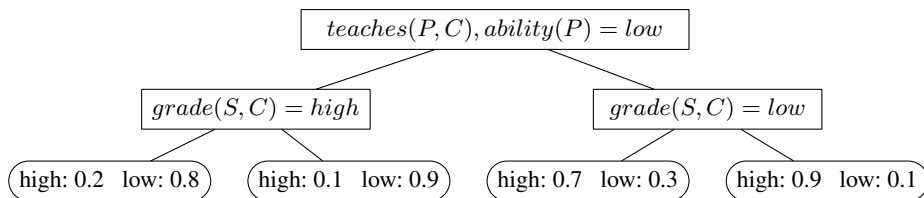


Fig. 1. Example of a logical CPD for $satisfaction(S, C)$. Tests in internal nodes are binary. When a test succeeds the left branch is taken, when it fails the right branch is taken. Note that internally in TILDE, tests like $grade(S, C) = low$ are represented as $grade(S, C, low)$.

Note that we cannot simply use tabular CPDs (i.e. conditional probability tables) in LBNs since they are too restrictive. One problem with tabular CPDs is that they cannot deal with a variable number of inputs (for instance, 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 can be first-order queries. As shown by Van Assche et al. [20], 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.

2.3 Equivalence of Sets of Dependency Clauses

Some LBNs have syntactic variants. The main manifestation of this is the fact that each LBN that contains dependency clauses with multiple atoms in the body is equivalent to an LBN that contains only dependency clauses with one atom in the body (we explain this below). Since syntactic variants could cause redundancy in the learning algorithm, they should be excluded as much as possible. Hence, when learning we only consider LBNs with *dependency clauses with only one probabilistic atom in the body*.

For each LBN with dependency clauses with multiple probabilistic atoms in the body an equivalent LBN can be obtained by replacing each such dependency clause C by a set of equivalent clauses. Concretely, for each probabilistic atom in the body of C we create a new clause with the same head, with only that atom in the body and with as context the context of C plus the condition that all other probabilistic atoms in the body of C are random variables. Rather than formally proving that this yields an equivalent LBN, we illustrate this with two examples.

Example 1 *Consider the university domain from the previous section. The LBN for this domain contains one dependency clauses with multiple atoms in the body, namely the following clause.*

```
grade(S,C) | intelligence(S), difficulty(C).
```

Following the definition of dependency clauses, the dependency of grade on intelligence and on difficulty only holds for student-course pairs for which intelligence and difficulty are defined (i.e. are random variables). Hence, this clause can be replaced by the following set of equivalent clauses.

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

We can simplify these clauses by using the information provided by the random variable declarations. First we can replace the random(.) literals in the context of the clauses by their definitions. This yields the following clauses.

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

The random variable declarations can be used to simplify these clauses even further. The condition course(C) in the context of the first clause can be dropped since grade(S,C) is only defined if C is a course anyway. Similarly, the condition student(S) in the context of the second clause can be dropped. We finally obtain the following clauses.

```
grade(S,C) | intelligence(S).
grade(S,C) | difficulty(C).
```

Example 2 *As a more advanced example, consider the following LBN.*

```
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).
```

Note that ranking and intelligence are defined for students but thesis-score is defined only for particular students, namely master students. Following the definition of dependency clauses, the dependency of ranking on intelligence and thesis-score only holds for students for which both intelligence and thesis-score are defined. In other words, the dependency only holds for master students. This dependency clause can be replaced by the following set of equivalent clauses.

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

As in the previous example, we can again simplify these clauses by using the random variable declarations. After the first step we get the following clauses.

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

In the second step the condition $student(S)$ in the context of both clauses can be dropped since ranking is only defined for students. We finally obtain the following clauses.

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

Note that the second clause 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 clause only applies to master students anyway.

2.4 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 clause with predicate p_2 in the head and p_1 in the body.

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.

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

We now discuss the problem of learning LBNs from relational data. In the learning setting that we consider, the random variable declarations are given (this is similar to the setting for Probabilistic Relational Models where the relational schema is given [8, 9]). The goal of learning is to find the dependency clauses and logical CPDs that maximize the scoring criterion. We focus on learning *non-recursive* LBNs. We briefly discuss learning recursive models in Section 6.

We assume that we have a dataset of *mega examples* (terminology adapted from Mihalkova et al. [14]). 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 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 [4] and Relational Bayesian Networks [11].

Example 3 *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 in that mega example. 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
```

4 Learning Non-recursive Logical Bayesian Networks: The Algorithms

We first briefly discuss a generic algorithm for learning the structure of LBNs in Section 4.1. In Section 4.2 we discuss how to learn logical CPDs under the form of logical probability trees. In Sections 4.3 and 4.4 we give two instantiations of the generic algorithm for learning LBNs. The first is the upgraded ordering-search algorithm that we introduce in this paper. The second is the existing upgraded structure-search algorithm. To stress the difference with Logical Bayesian Networks we will sometimes refer to ordinary Bayesian networks as ‘propositional’ Bayesian networks.

4.1 A Generic Algorithm for Learning Logical Bayesian Networks

As for propositional Bayesian networks, also for LBNs there exists a generic learning algorithm of which the structure-search and ordering-search algorithms are specific instantiations. The idea is to perform hillclimbing-search through a space of solutions. In the case of structure-search a *solution* is a structure (a set of dependency clauses), while in the case of ordering-search it is an ordering on the set of probabilistic predicates. The generic algorithm is shown in Figure 2. The *neighbourhood* of the current solution is defined as the set of solutions that can be obtained by making a small modification to the current solution. Computing the score of a solution is done by learning logical CPDs for that solution (see Section 4.2). Which probabilistic predicates can be used as input for the logical CPDs depends on the particular solution. When using a decomposable scoring criterion the above algorithm can be implemented quite efficiently (see Appendix A).

```

% find a good solution:
Solcurrent = random solution
compute score(Solcurrent)
repeat until convergence
  for each Solcand ∈ neighbourhood(Solcurrent)
    compute Δscore(Solcand) = score(Solcand) − score(Solcurrent)
  end for
  Solcurrent = argmax(Δscore(Solcand))
end repeat
% extract the dependency clauses from the final solution:
for each probabilistic predicate p
  extract the dependency clauses for p from Solcurrent
end for

```

Fig. 2. Generic hillclimbing algorithm for learning LBNs. In the two instantiations of this generic algorithm that we consider, a solution *Sol* corresponds to respectively a structure or an ordering.

To use the generic algorithm of Figure 2 with a specific kind of solutions (structures, orderings, ...) one has to determine a) how to obtain an initial random solution¹, b) how to define the neighbourhood of a solution, c) how to learn and score logical CPDs, and d) how to extract the dependency clauses from the learned solution. We explain learning logical CPDs in Section 4.2. We explain each of the other issues for ordering-search in Section 4.3 and for structure-search in Section 4.4.

4.2 Learning Logical CPDs

Logical CPDs represented as logical probability trees (such as the tree in Figure 1) can be learned using any of the standard probability tree algorithms in TILDE [7]. Below we discuss the data needed for learning such trees and the scoring criteria that we used.

Data for Learning Logical CPDs 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) R 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 R in m and consists of the part of m that is relevant for R .

Example 4 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* relationships).

Concretely, the simplified mega example given in Example 3 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.

¹ The initial solution might influence the final result since the algorithm only converges to a local optimum. Hence, we experimented with random restarts (multiple runs with different initial random solutions) but we found this not to be significantly better than a single run.

```

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

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

```

Scoring Criteria In this paper we consider two ways of learning and scoring trees. A straightforward approach is to learn and score trees using the Bayesian Information Criterion (BIC) [7, 10]. We also looked for an alternative approach since BIC is known to perform poorly when the cardinality (number of possible values) of the random variables is rather high. In this alternative approach we learn trees using randomization test pruning [7] since this works well even when the cardinality is high. However, randomization is a pruning criterion that can only be used to learn trees but not to score them. Of course, we cannot use BIC to score the trees learned using randomization since this would not solve our original problem. As a simple solution, we instead use log-likelihood as a scoring criterion, although this has the drawback of not penalizing extra dependency clauses which could potentially lead to overfitting. We refer to this alternative approach as “randomization/likelihood”.

4.3 Ordering-search

We now discuss ordering-search. First we briefly discuss the propositional case. then we discuss the case of LBNs and the differences between the two.

Ordering-search for Propositional Bayesian Networks Ordering-search is based on the fact that it is relatively easy to learn a Bayesian network if an ordering on the set of random variables is given [19]. Such an ordering eliminates the possibility of cycles. This makes it possible to decide for each variable X separately which variables, from all variables preceding it in the ordering, are its parents. This can simply be done by learning a CPD for X under the assumption that ‘selective’ CPDs are used, i.e. CPDs that select from all candidate inputs the relevant inputs (for instance conditional probability tables with a bound on the number of effective inputs [19]). However, the score of the Bayesian network that is learned in this way depends heavily on the quality of the ordering that is used. Hence, the idea of ordering-search is to perform hillclimbing through the space of possible orderings, in each step applying the above procedure.

Teyssier and Koller [19] experimentally compared ordering-search to 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 costly acyclicity checks.

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.

The above conclusions from the propositional case motivated us to investigate this upgrade. 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 set of probabilistic predicates* is given. We can then learn an LBN simply by learning for each probabilistic predicate p a logical probability tree with as inputs all predicates preceding p in the ordering. Ordering-search corresponds to applying the generic algorithm of Figure 2 with orderings as solutions. This is basically hillclimbing through the space of orderings. As an initial ordering we use any random ordering. The score of an ordering is defined as the score of the LBN that is learned for that ordering. The neighbourhood of an ordering O is defined as the set of orderings that can be obtained by swapping a pair of adjacent predicates in O (this is similar to what is done for propositional Bayesian networks [19]). Note that this implies that the size of the neighbourhood is linear in the number of probabilistic predicates. Once we found the optimal ordering and the logical CPDs for this ordering, we still need to extract the dependency clauses from these logical CPDs. We now discuss this in more detail.

Extracting the Dependency Clauses from the Logical CPDs Below we explain how to extract the dependency clauses from a logical probability tree. To obtain an LBN, this procedure has to be applied to the probability tree for each probabilistic predicate.

When extracting dependency clauses from a logical probability tree with as target the probabilistic atom p_{target} , we want to find the most specific set of clauses that is consistent with the tree. With *consistent* we mean that the tree should never test any probabilistic atom that is not a parent of p_{target} according to the set of clauses.

When extracting dependency clauses from a tree, we create a clause 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 clause 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 it were the logical literals that caused the failure and not the probabilistic tests).

Example 5 Consider the probability tree shown in Figure 1. 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 clauses 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).
```

Note that in principle the second clause can be dropped (it is a special case of the third). However, such a form of post-processing is currently not implemented in our learning system.

Note that in the above procedure for extracting the dependency clauses 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 the context of a dependency clause is not allowed to contain probabilistic atoms. This implies that generally not all independence information specified in a logical probability tree can be captured by the dependency clauses. This is of course not surprising since this 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 [3]).

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 [19] 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 tabular CPDs. Recall that for LBNs we cannot use simple tabular CPDs since they are too restrictive (see Section 2.2). However, for propositional Bayesian networks it would be possible to use (propositional) probability trees instead of tabular CPDs. Such an approach would have both an advantage and a disadvantage with respect to efficiency.

The advantage of using probability trees instead of tabular CPDs is that the effective parents of a random variable X can then be determined from the set of candidate parents of X (i.e. all variables preceding X in the current ordering) by learning only a single CPD for X . This is because probability trees are selective, so we can simply learn a tree using all candidate parents as input and determine the effective parents as all random variables that are used in the learned tree. In contrast, tabular CPDs 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; the lower k , the more selective the approach is. Teyssier and Koller compute the score of all CPDs for X having at most k candidate parents as inputs and then determine the effective parents as all random variables that are used in the highest scoring CPD for X . Of course, this is only feasible for small k (Teyssier and Koller use at most $k = 4$).

The disadvantage of using probability trees instead of tabular CPDs is that a tabular CPD can be learned more efficiently than a probability tree, especially if the number of inputs for the CPD is bounded by a small number k . This 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 the best of our knowledge, ordering-search for propositional Bayesian networks has not yet been used with probability trees. Hence, it is unclear whether the above advantage of probability trees over tabular CPDs outweighs its disadvantage or vice versa. Note, however, that this issue is only relevant in the propositional case and not in the case of LBNs since there tabular CPDs are too restrictive.

4.4 Structure-search

Structure-search (also known as DAG-search) is the most traditional and most straightforward approach for learning propositional Bayesian networks [10]. It is essentially hillclimbing through the space of possible structures. The neighbourhood of the current structure typically consists of all acyclic structures that can be obtained by adding, deleting or reversing an edge. This algorithm has already been upgraded to the relational case for several formalisms [8, 9, 12, 13, 15]. The algorithm that we use for LBNs is very similar to these existing upgrades.

To derive a concrete structure-search algorithm for LBNs from the generic algorithm of Figure 2, we have to define the notion of a neighbourhood and define how an initial structure is obtained (note that for structure-search the final step in the generic algorithm, extracting clauses from the solution, is not needed since a solution is a set of dependency clauses). We define the neighbourhood of the current structure as the set of all non-recursive structures that can be obtained by adding a dependency clause, deleting a clause or swapping the head and body of a clause in the current structure. Note that this implies that the size of the neighbourhood is quadratic in the number of probabilistic predicates. To find an initial set of clauses we borrow some elements from the ordering-search algorithm. Specifically, we generate a random initial ordering, learn logical CPDs for this ordering and apply the procedure for extracting dependency clauses from logical CPDs. 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.

We implemented one extension to the above structure-search algorithm for LBNs: *lookahead*. When using lookahead we not only try adding one clause but also adding two clauses with the same head during a single refinement step. This is useful when adding only one of the two clauses does not change the score, which is the case when the parent introduced by one clause is not predictive for the head in itself but only together with the parent introduced by the other clause. In the propositional case, this happens for instance when the head is the XOR of both parents. In the relational case, this happens for instance when the function of one predicate inside the logical CPD for the head is only to introduce a new logical variable used by another predicate in the logical CPD (this is a standard case of lookahead in inductive logic programming [1, 16]). Of course, in order for lookahead at the level of dependency clauses to have an effect, lookahead also needs to be applied at the level of logical CPDs. In our case this means that the logical probability trees should be learned using lookahead [1].

5 Experiments

We first discuss the datasets used and the experimental setup. Then we discuss our experimental results: first we compare ordering-search to structure-search, next we analyze runtimes of both algorithms and finally we briefly investigate the influence of the scoring criterion.

5.1 Datasets

We perform experiments on two relational domains: the synthetic university domain and the real-world UWCSE dataset.

For the synthetic university domain we generated datasets with a varying number of mega examples from the LBN given in Section 2. 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 used datasets of 5, 10, 15, 31, 62, 125 and 250 mega examples.

The UWCSE dataset is a real-world dataset that is currently a popular benchmark in the field of statistical relational learning [18]. This dataset was constructed by extracting information about graduate students, professors and courses from the web pages of a computer science department. The dataset consists of 5 mega examples, each corresponding to a specific research area. Since in this dataset relations are of special importance², we decided to incorporate them into the probabilistic model, leading to what Getoor et al. call *relational uncertainty* [9]. In LBNs this can be accomplished by simply modelling the relations as probabilistic predicates. We only used four logical predicates: *student/1*, *prof/1*, *course/1* and *same_area/2*. The random variable declarations look as follows.

```
random(phase_in_PhD(S)) <- student(S).
random(year_in_PhD(S)) <- student(S).
random(nb_publications(S)) <- student(S).
random(position(P)) <- prof(P).
random(nb_publications(P)) <- prof(P).
random(level(C)) <- course(C).
random(teaches(P,C)) <- prof(P),course(C),same_area(P,C).
random(assistant(S,C)) <- student(S),course(C),same_area(S,C).
random(advised_by(S,P)) <- student(S),prof(P),same_area(S,P).
random(co_author(S,P)) <- student(S),prof(P),same_area(S,P).
```

In total the dataset contains 140 students, 132 courses and 52 professors, leading to 9607 random variables. Of course, since this is a real-world dataset the true dependency clauses are unknown.

² For instance, this dataset has been used for supervised learning with the ‘advised by’ relation as the target [18].

5.2 Experimental Setup

For all experiments we performed 5-fold cross validation. For the synthetic university domain, mega examples were divided over folds randomly. For UWCSE, each fold corresponds to one mega example. We report the average results over the folds and use two-tailed paired t-tests (with $\alpha=0.05$) 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 clauses learned* (smaller is usually better because of ease of interpretation) and *runtime*.

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 the learned LBNs matches this true LBN. A simple measure for this degree of overlap would be the number of dependency clauses in the true LBN that are also in the learned LBN. However, the problem with this is that it is in general not possible to learn the true direction (which atom is in the head and which in the body) of each dependency clause from data. This is similar to the case of propositional Bayesian networks. The reason for this is that 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]. Hence, instead of directly comparing the learned LBN and the true LBN, we measure the overlap between the Markov equivalence class of the learned LBN and the true LBN. We do this by finding the LBN in the equivalence class of the learned LBN which has the most directed edges in common with the true LBN. We refer to this number as the *number of correct dependencies learned*.

For all evaluation criteria we report the results for the various 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 clauses, this is the LBN according to which all random variables are independent. We report results both with BIC as a scoring criterion and with randomization/likelihood.

In the following sections we only give summaries of our experimental results. The full results are given in Appendix B.

5.3 Ordering-search versus Structure-search

We now discuss the results of our experimental comparison of ordering-search and structure-search. We first discuss our results on the synthetic university domain and then our results on the UWCSE dataset.

The Synthetic University Domain We tried two algorithms on the datasets for this domain: ordering-search (OS) and structure-search without lookahead (SS). We did not try structure-search with lookahead on these datasets since lookahead is not needed to learn the target LBN for this domain (there are no probabilistic predicates whose only function is to introduce a new logical variable).

The results are summarised in Table 1. An entry in this table (for a particular evaluation criterion and dataset size) has the following meaning: if one of the two algorithms (OS or SS) 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 “/”. We also plotted the results as a function of dataset size in Figure 3 (we only show the results with BIC as a scoring criterion since the results with randomization/likelihood look very similar).

Table 1. Significance of differences between results of OS and of SS on synthetic university datasets of varying size (upper half of table: BIC; lower half: randomization). 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).

#MegaEx (#Vars)	LogLik(Test)	Score(Train)	#Clauses	#Correct	Depend	Time
5 (1150)	/	/	/	/	/	OS
10 (2300)	/	/	/	/	/	OS
15 (3450)	/	/	/	/	/	OS
31 (7130)	/	/	/	/	/	OS
62 (14260)	/	/	/	/	/	OS
125 (28750)	/	/	/	SS	/	OS
250 (57500)	/	/	/	SS	/	OS
5 (1150)	/	SS	OS	/	/	OS
10 (2300)	/	/	OS	/	/	OS
15 (3450)	/	SS	/	/	/	OS
31 (7130)	/	/	/	SS	/	OS
62 (14260)	/	/	/	/	/	OS
125 (28750)	/	SS	OS	/	/	OS
250 (57500)	/	/	/	/	/	OS

We can see from Table 1 that the results for test log-likelihood and runtime are consistent over various datasets sizes and scoring criteria (BIC or randomization/likelihood). The results for the number of dependency clauses and the number of correct dependencies learned are less clear.

OS and SS always obtain significantly better *test log-likelihood* and train score than the empty LBN. Table 1 shows that SS in some cases obtains significantly better train score than OS but this does not carry over to the test data: there is never a significant difference in test log-likelihood between OS and SS. The evolution of test log-likelihood and train score as a function of the dataset size (Figure 3) 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 the differences between OS and SS are small as compared to the differences with the empty LBN.

The results for the *number of dependency clauses* are not very clear. OS learns significantly less clauses than SS (i.e. learns more compact models) on three cases with

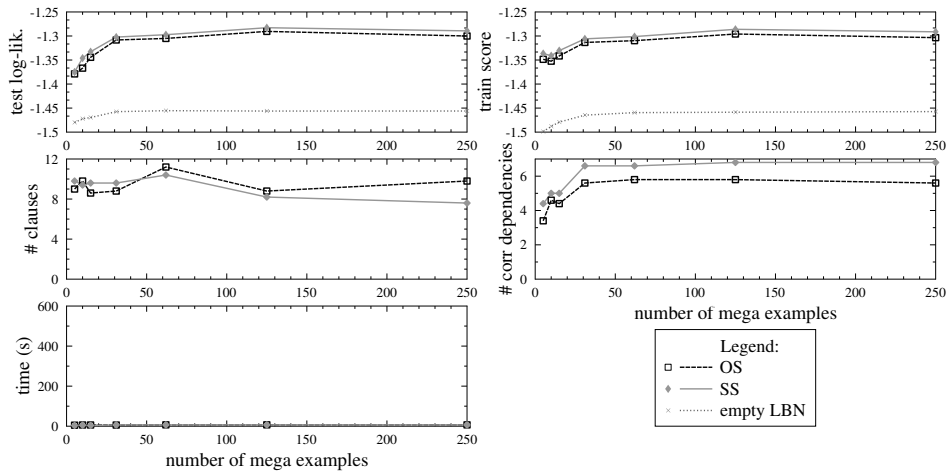


Fig. 3. Results for OS and SS on university datasets of varying size (with BIC as a scoring criterion). For train score and test log-likelihood we also show the results for the empty LBN.

randomization/likelihood as a scoring criterion but there is never a significant difference with BIC as a scoring criterion. Also, the evolution of the number of dependency clauses as a function of the dataset size (Figure 3) does not show any clear trends.

The *number of correct dependencies learned* is significantly higher for SS than for OS in three cases (the opposite never occurs). The evolution of the number of correct dependencies learned as a function of the dataset size (Figure 3) 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 size in which not all seven true dependencies were learned and found that the problem was always in having the wrong direction for a dependency (in other words, 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.

For both OS and SS *runtime* appears to be linear in the dataset size (Figure 3). Nevertheless, runtime is always significantly smaller for OS than for SS with differences being between a factor 3.0 and 4.1. We analyze runtimes in more detail in Section 5.4.

We conclude that in this domain OS is competitive with SS in terms of quality of the learned LBNs but OS is significantly faster.

The UWCSE Dataset We tried three algorithms on this dataset: OS, SS and SSL (structure-search with lookahead).

The results are summarized in Table 2. In the discussion below we first focus on the comparison between OS and SS. Next we briefly discuss the usefulness of lookahead for structure-search (i.e. we compare SS and SSL).

Table 2. Significance of differences between results for each pair of methods on the UWCSE dataset (upper half of table: BIC; lower half: randomization). OS, SS and SSL all have significantly better train score and test log-likelihood than the empty LBN (this is not shown in the table).

Algorithms	LogLik(Test)	Score(Train)	#Clauses	Time
OS vs SS	/	SS	/	OS
OS vs SSL	/	SSL	/	OS
SS vs SSL	/	/	/	SS
OS vs SS	SS	SS	SS	OS
OS vs SSL	/	SSL	OS	OS
SS vs SSL	/	/	SS	SS

OS and SS always obtain significantly better *test log-likelihood* and train score than the empty LBN. Table 2 shows that the train score is always significantly better for SS than for OS. This difference on training data only carries over to the test data with randomization/likelihood as a scoring criterion; with BIC as a scoring criterion, there is no significant difference in test log-likelihood between OS and SS.

Also for the *number of dependency clauses* the results depend on the scoring criterion. With BIC there is no significant difference between OS and SS, whereas with randomization/likelihood the number of dependency clauses is significantly lower for SS than for OS.

It is worth noting that although SS with randomization/likelihood performs better than OS with randomization/likelihood (significantly better test log-likelihood and significantly more compact models), it does not perform better than OS with BIC (no significant differences in test log-likelihood or compactness) and even performs worse than SS with BIC (no significant difference in test log-likelihood but significantly less compact models)³. Hence, BIC is clearly preferable over randomization/likelihood on this dataset. In that sense the above results with randomization/likelihood are somewhat less important than the results with BIC.

Runtime is always significantly smaller for OS than for SS with differences being between a factor 3.6 and 4.0.

We conclude that on the UWCSE dataset SS is a safer option than OS with respect to quality of the learned LBNs in the sense that with one scoring criterion OS is competitive with SS but with another, albeit worse, scoring criterion SS is superior to OS. The advantage of OS is that it is significantly faster than SS.

When comparing SS and SSL we see that there is never a significant difference in test log-likelihood or train score. SSL sometimes learns a significantly higher number of

³ These conclusions are based on additional t-tests not shown in Table 2.

dependency clauses (i.e. learns a significantly less compact model) while the opposite never occurs. SSL is significantly slower than SS with differences being between a factor 1.9 and 2.4. Hence there is no reason to prefer SSL over SS so we conclude that lookahead in structure-search has no added value on this dataset.

Finally, we investigated the learned LBNs to gain some more insight into this dataset. Most of the dependencies that were frequently learned (for the various algorithms and folds) confirm our intuitions about this dataset. Some examples of such dependencies and their common sense interpretations (obtained by investigating the corresponding logical CPDs) are the following.

- $phase_in_PhD(S)$ depends on $year_in_PhD(S)$ (the logical CPD specified that a student is more likely to be in a later phase if he is in a higher year)
- $nb_publications(S)$ depends on $year_in_PhD(S)$ (a student is more likely to have many publications if he is in a higher year)
- $nb_publications(P)$ 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 particular course if he is advised by a professor who teaches that course)

5.4 Analysis of Runtimes

In this section we analyze the runtime of the various algorithms in more detail by looking at the runtimes of the main different steps in the algorithms. Such an analysis has not been made before for ordering-search (also not for the propositional case).

The total runtime T_{total} of any concrete algorithm derived from the generic algorithm of Figure 2 can be decomposed as follows⁴

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

where T_{init} denotes the initialization time (the time for learning and scoring all logical CPDs for the initial random solution), T_{first} denotes the time for the first iteration (i.e. the first execution of the repeat loop in the generic algorithm) 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 (i.e. $T_{first} > T_{avg}$) 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 Appendix A). 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 runtime as follows

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

We now discuss our experimental results for each of the above measures. Note that T_{init} is the same for all algorithms that we considered. Hence we only discuss T_{first} , T_{avg} and I . First we compare OS and SS, then we briefly compare SS and SSL.

⁴ The time needed for the final step of extracting the dependency clauses from the logical CPDs can be ignored since it is very small (it does not depend on the dataset size).

Recall from the previous section that in our experiments the total runtime T_{total} was always significantly lower for OS than for SS with differences being between a factor 3.0 and 4.1. 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 solution have to be scored and the size of the neighbourhood is smaller for OS than for SS (linear in the number of probabilistic predicates for OS but quadratic for SS, see Section 4). In our experiments the difference in T_{first} between OS and SS goes from a factor 2.8 to 4.6.
- 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 1.7 to 3.0.
- The conclusions about the number of iterations I are less clear: in 10 cases I is significantly lower for OS than for SS, while in the remaining 6 cases there is no significant difference.

We now briefly discuss the influence of lookahead on the efficiency of structure-search. Recall from the previous section that in our experiments on the UWCSE dataset the total runtime was always significantly higher for SSL than for SS. When analyzing runtimes we found that T_{first} and T_{avg} are always significantly higher for SSL than for SS. This was expected since the neighbourhood for SSL is bigger than for SS. However, we also found that the number of iterations is not significantly different for SSL than for SS. This was somewhat against our expectations. One would expect SSL to need less iterations than SS to convergence since SSL can take bigger steps in the search space (in a single step SSL can not only add one dependency clause but can also add two dependency clauses with the same head) but apparently this is not the case.

5.5 Influence of the Scoring Criterion

In this section we briefly compare the results for the two scoring criteria that we used: BIC and randomization/likelihood. The main results are summarized in Table 3. This table is based on all results for OS and SS (we left out SSL since we already concluded that it does not have any advantages over SS). An entry like “5/8/1” in this table means that in 5 cases BIC is significantly better, in 8 cases there is no significant difference and in 1 case randomization/likelihood is significantly better.

The main conclusion from Table 3 is that BIC is preferable over randomization/likelihood both in terms of test log-likelihood, number of dependency clauses learned and runtime. In fact it appears that randomization/likelihood overfits. This hypothesis is based on two observations. First, although randomization/likelihood is always significantly better than BIC on training data, it is never significantly better and sometimes significantly worse than BIC on test data. Second, randomization/likelihood often learns significantly more clauses than BIC while the number of correct clauses is never significantly higher than for BIC. In other words, the extra clauses learned by randomization/likelihood are mainly redundant clauses. Upon closer inspection, we found this

Table 3. Influence of the scoring criterion: significant wins/ties/losses of BIC versus likelihood/randomization.

Data	LogLik(Test)	Score(Train)	#Clauses	#CorrectDepend	Time
Univ	4/10/0	0/0/14	5/8/1	0/14/0	6/8/0
UWCSE	0/2/0	0/0/2	2/0/0	-	0/2/0
All	4/12/0	0/0/16	7/8/1	0/14/0	6/10/0

overfitting behaviour to be the strongest on the bigger datasets for the university domain (62 to 250 mega examples).

In a previous study on pruning criteria for probability trees we concluded that BIC is inferior to randomization [7]. The fact that in our present experiments on learning LBNs (using probability trees as logical CPDs) BIC performs better than randomization/likelihood can be explained in two ways. First, BIC has the advantage that it can be used both to learn and to score trees, while randomization is a pruning criterion that can only be used to learn trees but not to score them. Hence, when learning trees using randomization we have to use another criterion to score the trees, such as likelihood. However, the drawback of likelihood is that it does not penalize extra dependency clauses which could indeed lead to overfitting. Second, in the previous study we concluded that learning probability trees using BIC works quite well as long as the cardinality (number of possible values) of the target attribute is small. In the experiments in this paper this is indeed the case since all random variables have cardinality two or three. It is very well possible that in domains with higher cardinalities, randomization/likelihood would perform better than BIC.

6 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 several scenarios depending on how much prior knowledge is available about the presumed recursive dependencies.

6.1 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. took this perspective when developing the learning algorithm for Probabilistic Relational Models [9]. To accommodate for dependencies that are cyclic at the predicate level but acyclic at the ground level they let the user define *guaranteed acyclic relationships*. For instance, to allow that a property of a person depends on this

property for his parents the parent relation has to be defined as guaranteed acyclic. Similarly, to allow that the topic of a paper depends on the topic of the papers published earlier by the same author, the published-earlier relation has to be defined as guaranteed acyclic. Getoor et al. then apply structure-search and use the information about the guaranteed acyclic relationships during the acyclicity checks to deduce that certain cycles at the predicate level are legal.

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 guaranteed acyclic relationships 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. When learning the logical CPD for a predicate p , the probabilistic predicates used as input would not only be all predicates that precede p in the current ordering but also p itself and possibly also all predicates following p in the ordering, with the restriction that p can only depend on itself and on the latter predicates through a guaranteed acyclic relationship. This actually requires no changes to our ordering-search algorithm but only requires to extend the declarative bias for the logical CPDs.

6.2 No Prior Knowledge

When one does not have enough prior knowledge about the data to find a set of guaranteed acyclic relationships, more complicated approaches have to be taken. When applying structure-search, one can use the techniques developed for Bayesian Logic Programs [12, 13]. 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. The main drawback of this approach is that the acyclicity checks might 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 a candidate model only needs to be checked once and at the predicate level.

An alternative approach is to use *generalized ordering-search* [17], an algorithm we recently developed to learn recursive dependencies. Generalized ordering-search considers orderings on ground probabilistic atoms instead of on predicates (as the ordering-search algorithm proposed 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 clauses 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.

7 Conclusion

We upgraded the ordering-search algorithm for propositional Bayesian networks towards non-recursive directed probabilistic logical models. We experimentally compared the resulting algorithm with the existing upgraded structure-search algorithm on two relational domains. The results show that ordering-search is competitive with structure-search in terms of quality of the learned models but is significantly faster than structure-search. 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”.

References

1. H. Blockeel, and L. De Raedt. Lookahead and discretization in ILP. In *Proceedings of the 7th International Workshop on Inductive Logic Programming (ILP)*, volume 1297 of *Lecture Notes in Computer Science*, pages 77–85. Springer Verlag, 1997.
2. D.M. Chickering. Learning equivalence classes of Bayesian network structures. *Journal of Machine Learning Research*, volume 2, pages 445–498, 2002.
3. C. Boutilier, N. Friedman, M. Goldszmidt and D. Koller. Context-specific independence in Bayesian networks. In *Proceedings of the 12th conference on Uncertainty in AI (UAI)*, pages 115–123. 1996.
4. L. De Raedt and K. Kersting. Probabilistic Inductive Logic Programming. In *Proceedings of the 15th International Conference on Algorithmic Learning Theory (ALT)*, volume 3244 of *Lecture Notes in Computer Science*, pages 19–36. Springer, 2004.
5. D. Fierens, H. Blockeel, M. Bruynooghe, and J. Ramon. Logical Bayesian networks and their relation to other probabilistic logical models. In *Proceedings of the 15th International Conference on Inductive Logic Programming (ILP)*, volume 3625 of *Lecture Notes in Computer Science*, pages 121–135. Springer, 2005.
6. D. Fierens, J. Ramon, M. Bruynooghe, and H. Blockeel. Learning Directed Probabilistic Logical Models: Ordering-search versus Structure-search. In *Proceedings of the 18th European Conference on Machine Learning (ECML)*, volume 4701 of *Lecture Notes in Computer Science*, Springer, 2007. To appear.
7. D. Fierens, J. Ramon, H. Blockeel, and M. Bruynooghe. A comparison of pruning criteria for learning trees. Technical Report CW 488, Department of Computer Science, Katholieke Universiteit Leuven, April 2007.
8. N. Friedman, L. Getoor, D. Koller, and A. Pfeffer. Learning Probabilistic Relational Models. In *Proceedings of the 16th International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1300–1307, Morgan Kaufmann, 1999.
9. L. Getoor, N. Friedman, D. Koller, and A. Pfeffer. Learning Probabilistic Relational Models. In S. Dzeroski and N. Lavrac, editors, *Relational Data Mining*, pages 307–334. Springer-Verlag, 2001.

10. D. Heckerman, D. Geiger, and D.M. Chickering. Learning Bayesian networks: The combination of knowledge and statistical data. *Machine Learning*, volume 20, pages 197–243, 1995.
11. M. Jaeger. Parameter learning for relational Bayesian networks. In *Proceedings of the 24th International Conference on Machine Learning (ICML)*, 2007.
12. K. Kersting and L. De Raedt. Towards combining inductive logic programming and Bayesian networks. In *Proceedings of the 11th International Conference on Inductive Logic Programming (ILP)*, volume 2157 of *Lecture Notes in Computer Science*, pages 118–131. Springer-Verlag, 2001.
13. K. Kersting and L. De Raedt. Basic principles of learning Bayesian logic programs. Technical Report No. 174, Institute for Computer Science, University of Freiburg, June 2002.
14. L. Mihalkova, T. Huynh and R. Mooney. Mapping and revising Markov logic networks for transfer learning. In *Proceedings of the 22nd Conference on Artificial Intelligence (AAAI)*, 2007.
15. S. Natarajan, W. Wong, and P. Tadepalli. Structure refinement in First Order Conditional Influence Language. In *Proceedings of the ICML workshop on Open Problems in Statistical Relational Learning (SRL)*, 2006.
16. R. Quinlan. Learning logical definitions from relations. *Machine Learning*, volume 5, pages 239–266, 1990.
17. J. Ramon, T. Croonenborghs, D. Fierens, H. Blockeel, and M. Bruynooghe. Generalized ordering-search for learning directed probabilistic logical models. 2007. Conditionally accepted for *Machine Learning*, special issue on Inductive Logic Programming.
18. M. Richardson and P. Domingos. Markov logic networks. *Machine Learning*, volume 62(1–2), pages 107–136, 2006.
19. M. Teyssier and D. Koller. Ordering-based search: A simple and effective algorithm for learning Bayesian networks. In *Proceedings of the 21st Conference on Uncertainty in AI (UAI)*, pages 584–590. AUAI Press, 2005.
20. A. Van Assche, C. Vens, H. Blockeel and S. Džeroski. First order random forests: learning relational classifiers with complex aggregates. *Machine Learning*, volume 64(1–3), pages 149–182, 2006.

A Efficiently Implementing the Algorithms

In this appendix we briefly discuss how to efficiently implement ordering-search and structure-search. The optimizations discussed below apply equally well to the propositional case as to the case of LBNs. In fact, in the propositional case these optimizations are standard [10].

A *decomposable* scoring criterion is a criterion for which the score of a solution is the sum of the scores of all logical CPDs for that solution. Both scoring criteria used in this paper, BIC and log-likelihood, are decomposable. When using a decomposable scoring criterion, the generic algorithm of Figure 2 and the ordering-search and structure-search algorithms derived from it can be implemented quite efficiently. To be precise, there are two important insights.

First, the score-change for a candidate solution ($\Delta score(Sol_{cand})$ in the algorithm of Figure 2) can be computed efficiently. This is because it only depends on the score of the logical CPDs that are different for the candidate solution and the current solution. For ordering-search there are only two such logical CPDs for each candidate solution

(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 also at most two such logical CPDs (a new candidate structure is obtained by adding/deleting/swapping a dependency clause in the current structure and adding/deleting a dependency clause influences only the logical CPD for the predicate in the head of the clause while swapping a clause influences the logical CPDs for both predicates in the clause).

Second, many of the score-changes that are computed during one iteration of the repeat loop 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 solution if and only if the set of potential parents that is changed by m_1 was not changed by m_2 .

B Detailed Experimental Results

This appendix contains detailed results for the experiments discussed in Section 5.

Table 4. Experimental results on the UWCSE dataset (upper half of table: BIC; lower half: randomization). The best results are shown in bold.

Method	LogLik(Test)	Score(Train)	#Clauses
OS	-0.4288	-0.3539	15.2
SS	-0.4160	-0.3489	14.6
SSL	-0.4156	-0.3484	16.2
empty	-0.4631	-0.3961	-
OS	-0.4318	-0.3343	20.6
SS	-0.4227	-0.3301	18.6
SSL	-0.4251	-0.3286	22.4
empty	-0.4631	-0.3865	-

Table 5. Detailed timings for the UWCSE dataset (upper half of table: BIC; lower half: randomization).

Method	T_{total}	T_{init}	T_{first}	T_{rest}	I	T_{avg}
OS	135s	27s	61s	46s	2.4	20s
SS	535s	27s	279s	229s	4.6	52s
SSL	1287s	27s	806s	453s	5.4	87s
OS	208s	35s	69s	104s	4.2	23s
SS	738s	35s	309s	394s	6.2	62s
SSL	1428s	35s	724s	669s	6.8	97s

Table 6. Experimental results on synthetic university datasets of varying size (upper half of table: BIC; lower half: randomization). The best results are shown in bold.

#MegaEx (#Vars)	Method	LogLik(Test)	Score(Train)	#Clauses	#CorrectDepend
5 (1150)	OS	-1.3789	-1.3485	9.0	3.4
5 (1150)	SS	-1.3750	-1.3365	9.8	4.4
5 (1150)	empty	-1.4799	-1.4989	-	-
10 (2300)	OS	-1.3669	-1.3524	9.8	4.6
10 (2300)	SS	-1.3461	-1.3410	9.4	5.0
10 (2300)	empty	-1.4722	-1.4880	-	-
15 (3450)	OS	-1.3444	-1.3415	8.6	4.4
15 (3450)	SS	-1.3328	-1.3305	9.6	5.0
15 (3450)	empty	-1.4697	-1.4792	-	-
31 (7130)	OS	-1.3083	-1.3135	8.8	5.6
31 (7130)	SS	-1.3023	-1.3060	9.6	6.6
31 (7130)	empty	-1.4575	-1.4647	-	-
62 (14260)	OS	-1.3051	-1.3097	11.2	5.8
62 (14260)	SS	-1.2973	-1.3012	10.4	6.6
62 (14260)	empty	-1.4554	-1.4595	-	-
125 (28750)	OS	-1.2905	-1.2959	8.8	5.8
125 (28750)	SS	-1.2828	-1.2861	8.2	6.8
125 (28750)	empty	-1.4562	-1.4586	-	-
250 (57500)	OS	-1.3001	-1.3035	9.8	5.6
250 (57500)	SS	-1.2894	-1.2913	7.6	6.8
250 (57500)	empty	-1.4561	-1.4573	-	-
5 (1150)	OS	-1.3606	-1.2963	5.2	3.4
5 (1150)	SS	-1.3806	-1.2633	8.2	3.8
5 (1150)	empty	-1.4799	-1.4571	-	-
10 (2300)	OS	-1.3665	-1.3149	6.8	3.8
10 (2300)	SS	-1.3636	-1.2878	10.4	5.2
10 (2300)	empty	-1.4722	-1.4639	-	-
15 (3450)	OS	-1.3427	-1.3041	10.6	4.4
15 (3450)	SS	-1.3390	-1.2791	12.0	5.0
15 (3450)	empty	-1.4697	-1.4619	-	-
31 (7130)	OS	-1.3205	-1.2849	12.4	5.0
31 (7130)	SS	-1.3128	-1.2764	13.0	6.4
31 (7130)	empty	-1.4575	-1.4552	-	-
62 (14260)	OS	-1.3046	-1.2936	12.8	5.2
62 (14260)	SS	-1.3061	-1.2869	15.8	5.8
62 (14260)	empty	-1.4554	-1.4542	-	-
125 (28750)	OS	-1.2920	-1.2870	13.2	6.0
125 (28750)	SS	-1.2877	-1.2763	17.0	6.2
125 (28750)	empty	-1.4562	-1.4557	-	-
250 (57500)	OS	-1.2971	-1.2925	15.6	6.0
250 (57500)	SS	-1.2928	-1.2856	18.0	6.4
250 (57500)	empty	-1.4561	-1.4557	-	-

Table 7. Detailed timings for synthetic university datasets of varying size (upper half of table: BIC; lower half: randomization).

#MegaEx (#Vars)	Method	T_{total}	T_{init}	T_{first}	T_{rest}	I	T_{avg}
5 (1150)	OS	35s	7s	15s	13s	2.4	6s
5 (1150)	SS	137s	7s	60s	70s	5.8	12s
10 (2300)	OS	42s	7s	20s	15s	2.0	7s
10 (2300)	SS	134s	7s	69s	57s	4.2	13s
15 (3450)	OS	51s	8s	18s	24s	3.0	8s
15 (3450)	SS	164s	8s	73s	83s	5.8	14s
31 (7130)	OS	55s	10s	24s	20s	2.2	9s
31 (7130)	SS	168s	10s	85s	73s	4.6	16s
62 (14260)	OS	78s	13s	31s	34s	2.8	12s
62 (14260)	SS	262s	13s	101s	147s	6.2	24s
125 (28750)	OS	120s	18s	45s	57s	3.2	19s
125 (28750)	SS	407s	18s	161s	228s	4.4	56s
250 (57500)	OS	194s	33s	89s	71s	2.2	33s
250 (57500)	SS	586s	33s	246s	307s	4.8	66s
5 (1150)	OS	31s	7s	17s	7s	1.2	6s
5 (1150)	SS	120s	7s	64s	49s	4.0	13s
10 (2300)	OS	42s	8s	20s	15s	2.0	8s
10 (2300)	SS	167s	8s	70s	89s	6.0	14s
15 (3450)	OS	56s	10s	25s	22s	2.6	8s
15 (3450)	SS	202s	10s	79s	11s4	6.8	17s
31 (7130)	OS	74s	11s	28s	35s	2.8	13s
31 (7130)	SS	256s	11s	92s	152s	6.8	21s
62 (14260)	OS	124s	19s	36s	68s	4.6	14s
62 (14260)	SS	415s	19s	124s	272s	8.2	33s
125 (28750)	OS	171s	31s	58s	82s	4.2	21s
125 (28750)	SS	692s	31s	192s	470s	10.2	46s
250 (57500)	OS	321s	56s	102s	164s	3.8	43s
250 (57500)	SS	1179s	56s	358s	765s	8.6	88s