Mach Learn (2015) 100:127–156
DOI 10.1007/s10994-015-5510-3

# Bandit-based Monte-Carlo structure learning of probabilistic logic programs

**Nicola Di Mauro**[1] · **Elena Bellodi**[2] · **Fabrizio Riguzzi**[3]

Received: 10 January 2014 / Accepted: 28 May 2015 / Published online: 16 July 2015


**Abstract** Probabilistic logic programming can be used to model domains with complex and uncertain relationships among entities. While the problem of learning the parameters of such programs has been considered by various authors, the problem of learning the structure is yet to be explored in depth. In this work we present an approximate search method based on a one-player game approach, called LEMUR. It sees the problem of learning the structure of a probabilistic logic program as a multi-armed bandit problem, relying on the Monte-Carlo tree search UCT algorithm that combines the precision of tree search with the generality of random sampling. LEMUR works by modifying the UCT algorithm in a fashion similar to FUSE, that considers a finite unknown horizon and deals with the problem of having a huge branching factor. The proposed system has been tested on various real-world datasets and has shown good performance with respect to other state of the art statistical relational learning approaches in terms of classification abilities.

**Keywords** Statistical relational learning · Structure learning · Distribution semantics · Multi-armed bandit problem · Monte Carlo tree search · Logic programs with annotated disjunctions

Editors: Gerson Zaverucha and Vítor Santos Costa.

✉ Elena Bellodi
elena.bellodi@unife.it

Nicola Di Mauro
nicola.dimauro@uniba.it

Fabrizio Riguzzi
fabrizio.riguzzi@unife.it

[1] Dipartimento di Informatica, University of Bari "Aldo Moro", Via Orabona, 4, 70125 Bari, Italy

[2] Dipartimento di Ingegneria, University of Ferrara, Via Saragat 1, 44122 Ferrara, Italy

[3] Dipartimento di Matematica e Informatica, University of Ferrara, Via Saragat 1, 44122 Ferrara, Italy

## 1 Introduction

Probabilistic Logic Programming (PLP) is gaining popularity due to its ability to represent domains with many entities connected by complex and uncertain relationships. One of the most fertile approaches to PLP is the distribution semantics (Sato 1995), that is at the basis of several languages such as the Independent Choice Logic (Poole 2008), PRISM (Sato 2008), Logic Programs with Annotated Disjunctions (LPADs) (Vennekens et al. 2004) and ProbLog (De Raedt et al. 2007). Various algorithms for learning the parameters of probabilistic logic programs under the distribution semantics have been proposed, such as PRISM (Sato and Kameya 2001), LFI-ProbLog (Gutmann et al. 2011) and EMBLEM (Bellodi and Riguzzi 2013). Fewer systems have been developed for learning the structure of these programs. Among these, SLIPCASE (Bellodi and Riguzzi 2012) performs a beam search in the space of possible theories using the log-likelihood (LL) of the examples as the heuristic. The beam is initialized with a number of simple theories that are repeatedly revised using theory revision operators: the addition/removal of a literal from a rule and the addition/removal of a whole rule. Each refinement is scored by learning the parameters with EMBLEM and using the LL of the examples returned by it. SLIPCOVER (Bellodi and Riguzzi 2014) differs from SLIPCASE because the beam search is performed in the space of clauses. In this way, a set of promising clauses is identified and these are added one by one to the empty theory, keeping each clause if the LL improves.

Since SLIPCASE and SLIPCOVER search space is extremely large, in this paper we investigate the application of a new approximate search method. In particular, we propose to search the space of possible theories using a Monte Carlo Tree Search (MCTS) algorithm (Browne et al. 2012). MCTS has been originally and extensively applied to Computer Go and recently used in Machine Learning in FUSE (Feature UCT Selection) (Gaudel and Sebag 2010), that performs feature selection, and BAAL (Bandit-based Active Learner) (Rolet et al. 2009), that focuses on active learning with small training sets. In this paper, similarly to FUSE, we propose the system LEMUR (*LEarning with a Monte carlo Upgrade of tRee search*) relying on UCT, the tree-structured multi-armed bandit algorithm originally introduced in (Kocsis and Szepesvári 2006).

We tested LEMUR on seven datasets: UW-CSE, Mutagenesis, Hepatitis, Carcinogenesis, IMDB, Mondial and HIV. We compared it with various state of the art systems for structure learning of PLP and Markov Logic Networks. LEMUR achieves higher areas under the Precision Recall and ROC curves in most cases, thus showing its classification abilities. To investigate LEMUR behaviour in modeling distributions, we computed the LL of the test sets and we analyzed its performance on the HIV dataset, where we try to model all the predicates at once. In this case LEMUR is exceeded by five systems out of twelve in terms of LL, thus highlighting an area for improvement.

The paper is organized as follows. Section 2 presents Probabilistic Logic Programming, concentrating on LPADs. Section 3 defines the multi-armed bandit problem while Sect. 4 provides an overview of MCTS algorithms. Section 5 describes the LEMUR system. Section 6 presents related work, Sect. 7 experimentally evaluates our system and Sect. 8 concludes the paper.

## 2 Probabilistic logic programming

We introduce PLP focusing on the distribution semantics (Sato 1995). We use LPADs as the language for their general syntax and we don't allow function symbols; for the treatment of function symbols see Riguzzi and Swift (2013).

LPADs (Vennekens et al. 2004) consist of a finite set of annotated disjunctive clauses $C_i$ of the form $h_{i1} : \Pi_{i1}; \ldots; h_{in_i} : \Pi_{in_i} : -b_{i1}, \ldots, b_{im_i}$, where $h_{i1}, \ldots h_{in_i}$ are logical atoms and $b_{i1}, \ldots, b_{im_i}$ are logical literals. $\{\Pi_{i1}, \ldots, \Pi_{in_i}\}$ are real numbers in the interval $[0, 1]$ such that $\sum_{k=1}^{n_i} \Pi_{ik} \leq 1$. $b_{i1}, \ldots, b_{im_i}$ is called the *body* and is indicated with $body(C_i)$. Note that if $n_i = 1$ and $\Pi_{i1} = 1$ the clause corresponds to a non-disjunctive clause. If $\sum_{k=1}^{n_i} \Pi_{ik} < 1$, the head of the annotated disjunctive clause implicitly contains an extra atom *null* that does not appear in the body of any clause and whose annotation is $1 - \sum_{k=1}^{n_i} \Pi_{ik}$. We denote by $ground(\mathcal{T})$ the grounding of an LPAD $\mathcal{T}$.

An *atomic choice* is a triple $(C_i, \theta_j, k)$ where $C_i \in \mathcal{T}, \theta_j$ is a substitution that grounds $C_i$ and $k \in \{1, \ldots, n_i\}$ identifies a head atom of $C_i$. $C_i\theta_j$ corresponds to a multi-valued random variable $X_{ij}$ and an atomic choice $(C_i, \theta_j, k)$ to an assignment $X_{ij} = k$. A set of atomic choices $\kappa$ is *consistent* if only one head is selected from a ground clause. A *composite choice* $\kappa$ is a consistent set of atomic choices. The *probability* $P(\kappa)$ *of a composite choice* $\kappa$ is the product of the probabilities of the individual atomic choices, i.e. $P(\kappa) = \prod_{(C_i,\theta_j,k)\in\kappa} \Pi_{ik}$. A *selection* $\sigma$ is a composite choice that, for each clause $C_i\theta_j$ in $ground(\mathcal{T})$, contains an atomic choice $(C_i, \theta_j, k)$. A selection $\sigma$ identifies a normal logic program $w_\sigma$ defined as $w_\sigma = \{(h_{ik} \leftarrow body(C_i))\theta_j | (C_i, \theta_j, k) \in \sigma\}$, which is called a *world* of $\mathcal{T}$. Since selections are composite choices, we can assign a probability to worlds: $P(w_\sigma) = P(\sigma) = \prod_{(C_i,\theta_j,k)\in\sigma} \Pi_{ik}$. We denote by $S_\mathcal{T}$ the set of all selections and by $W_\mathcal{T}$ the set of all worlds of a program $\mathcal{T}$. A composite choice $\kappa$ identifies a set of worlds $\omega_\kappa = \{w_\sigma | \sigma \in S_\mathcal{T}, \sigma \supseteq \kappa\}$. We define the set of worlds identified by a set of composite choices $K$ as $\omega_K = \bigcup_{\kappa\in K} \omega_\kappa$.

We consider only *sound* LPADs, where each possible world has a total well-founded model, so $w_\sigma \models Q$ means a query $Q$ is true in the well-founded model of the program $w_\sigma$. The probability of a query $Q$ given a world $w$ is $P(Q|w) = 1$ if $w \models Q$ and 0 otherwise. The probability of $Q$ is then:

$$P(Q) = \sum_{w\in W_\mathcal{T}} P(Q, w) = \sum_{w\in W_\mathcal{T}} P(Q|w)P(w) = \sum_{w\in W_\mathcal{T}: w\models Q} P(w) \tag{1}$$

*Example 1* The following LPAD $\mathcal{T}$ models the fact that if somebody has the flu and the climate is cold, there is the possibility that an epidemic or a pandemic arises:

$$C_1 = epidemic : 0.6; pandemic : 0.3 : -flu(X), cold.$$
$$C_2 = cold : 0.7.$$
$$C_3 = flu(david).$$
$$C_4 = flu(robert).$$

$T$ has 18 instances, the query $Q = epidemic$ is true in 5 of them and its probability is $P(epidemic) = 0.6{\cdot}0.6{\cdot}0.7{+}0.6{\cdot}0.3{\cdot}0.7{+}0.6{\cdot}0.1{\cdot}0.7{+}0.3{\cdot}0.6{\cdot}0.7{+}0.1{\cdot}0.6{\cdot}0.7 = 0.588$.

### 2.1 Inference

Since it is unfeasible to enumerate all the worlds where $Q$ is true, inference algorithms find in practice a covering set of *explanations* for $Q$, i.e. a set of composite choices $K$ such that $Q$ is true in a world $w_\sigma$ iff $w_\sigma \in \omega_K$. For Example 1, a covering set of explanations is $\{\{(C_1, \{X/david\}, 1), (C_2, \emptyset, 1)\}, \{(C_1, \{X/robert\}, 1), (C_2, \emptyset, 1)\}\}$ where $\theta_1 = \{X/david\}, \theta_2 = \{X/robert\}$ and non-disjunctive clauses are omitted.

From the set $K$, the following Boolean function can be built:

$$f_K(\mathbf{X}) = \bigvee_{\kappa\in K} \bigwedge_{(C_i,\theta_j,k)\in\kappa} (X_{ij} = k) \tag{2}$$

where $\mathbf{X} = \{X_{ij} | C_i$ is a clause and $\theta_j$ is a grounding substitution of $C_i\}$ is a set of multi-valued random variables. The domain of $X_{ij}$ is $1, \ldots, n_i$ and its probability distribution is given by $P(X_{ij} = k) = \Pi_{ik}$. The problem of computing the probability $P(Q)$ can be solved by computing the probability that $f_K(\mathbf{X})$ takes on value true. For Example 1, (2) is given by

$$f_K(\mathbf{X}) = (X_{11} \wedge X_{21}) \vee (X_{12} \wedge X_{21}) \tag{3}$$

where $X_{11}$ corresponds to $(C_1, \{X/david\})$, $X_{12}$ corresponds to $(C_1, \{X/robert\})$ and $X_{21}$ corresponds to $(C_2, \emptyset)$.

$f_K(\mathbf{X})$ in (2) can be translated into a function of Boolean random variables by encoding the multi-valued variables. Various options are possible, we found that the following provides good performance (Sang et al. 2005; De Raedt et al. 2008a): for a multi-valued variable $X_{ij}$, corresponding to the ground clause $C_i\theta_j$, having $n_i$ values, we use $n_i - 1$ Boolean variables $X_{ij1}, \ldots, X_{ijn_i-1}$ and we represent the equation $X_{ij} = k$ for $k = 1, \ldots n_i - 1$ by means of the conjunction $\overline{X_{ij1}} \wedge \ldots \wedge \overline{X_{ijk-1}} \wedge X_{ijk}$, and the equation $X_{ij} = n_i$ by means of the conjunction $\overline{X_{ij1}} \wedge \ldots \wedge \overline{X_{ijn_i-1}}$. For Example 1, $X_{11} = 1$ is represented as $X_{111}$ and $X_{11} = 2$ as $\overline{X_{111}} \wedge X_{112}$. Let us call $f'_K(\mathbf{X}')$ the result of replacing multi-valued random variables with Boolean variables in $f_K(\mathbf{X})$.

The probability distribution of the Boolean random variables $X_{ijk}$ is computed from that of multi-valued variables as $\pi_{i1} = \Pi_{i1}, \ldots, \pi_{ik} = \frac{\Pi_{ik}}{\prod_{j=1}^{k-1}(1-\pi_{ij})}$ up to $k = n_i - 1$, where $\pi_{ik}$ is the probability that $X_{ijk}$ is true. With this distribution the probability that $f'_K(\mathbf{X}')$ is true is the same as $f_K(\mathbf{X})$ and thus is the same as $P(Q)$. For Example 1, $f'_K(\mathbf{X}')$ is given by

$$f'_K(\mathbf{X}') = (X_{111} \wedge X_{211}) \vee (X_{121} \wedge X_{211}) \tag{4}$$

Computing the probability that $f'_K(\mathbf{X}')$ is true is a SUM- OF- PRODUCTS problem and it was shown to be #P-hard (see e.g. Rauzy et al. 2003). An approach that was found to give good results in practice is *knowledge compilation* (Darwiche and Marquis 2002), i.e. translating $f'_K(\mathbf{X}')$ to a target language that allows answering queries in polynomial time. A target language often used is that of Binary Decision Diagrams (BDD). From a BDD we can compute the probability of the query with a dynamic programming algorithm that is linear in the size of the BDD (De Raedt et al. 2007). Algorithms that adopt such an approach for inference include Riguzzi (2007b, 2009, 2014); Riguzzi and Swift (2010, 2011).

A BDD for a function of Boolean variables is a rooted graph that has one level for each Boolean variable. A node $n$ in a BDD has two children: one corresponding to the 1 value of the variable associated with $n$, indicated with $child_1(n)$, and one corresponding to the 0 value of the variable, indicated with $child_0(n)$. When drawing BDDs, the 0-branch—the one going to $child_0(n)$—is distinguished from the 1-branch by drawing it with a dashed line. The leaves store either 0 or 1.

BDDs can be built by combining simpler BDDs using Boolean operators. While building BDDs, simplification operations can be applied that delete or merge nodes. Merging is performed when the diagram contains two identical sub-diagrams, while deletion is performed when both arcs from a node point to the same node. In this way a reduced BDD is obtained, often with a much smaller number of nodes with respect to the original BDD. The size of the reduced BDD depends on the order of the variables: finding an optimal order is an NP-complete problem (Bollig and Wegener 1996) and several heuristic techniques are used in practice by highly efficient software packages such as CUDD[1]. Alternative methods involve learning variable order from examples (Grumberg et al. 2003). A BDD for function (4) is shown in Fig. 1.

[1] Available at http://vlsi.colorado.edu/~fabio/CUDD/.

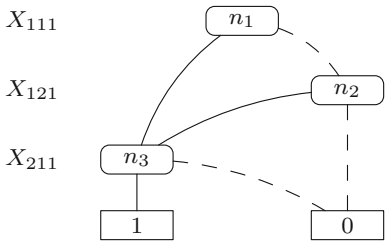


**Fig. 1** BDD for function (4)

## 2.2 Parameter learning

BDDs are employed to efficiently perform parameter learning of LPADs by the system EMBLEM (Bellodi and Riguzzi 2013), based on an Expectation Maximization (EM) algorithm. EMBLEM takes as input a set of interpretations, i.e., sets of ground facts describing a portion of the domain. It is targeted at discriminative learning, since the user has to indicate which predicate(s) of the domain is/are *target*, the one(s) for which we are interested in good predictions. The interpretations must contain also negative facts for target predicates. All ground atoms for the target predicates will represent the positive and negative examples (*queries* $Q$) for which BDDs are built, encoding the disjunction of their explanations. After building the BDDs, EMBLEM then maximizes the LL for the positive and negative target examples with an EM cycle, until it has reached a local maximum or a maximum number of steps is executed. The E-step computes the expectations of the latent variables directly over BDDs and returns the LL of the data that is used in the stopping criterion. For each target fact Q, the expectations are $\mathbf{E}[X_{ijk} = x|Q]$ for all $C_i$s, $k = 1, \ldots, n_i - 1$, $j \in g(i) := \{j|\theta_j$ is a substitution grounding $C_i\}$ and $x \in \{0, 1\}$. $\mathbf{E}[X_{ijk} = x|Q]$ is given by

$$\mathbf{E}[X_{ijk} = x|Q] = P(X_{ijk} = x|Q) \cdot 1 + P(X_{ijk} = (1 - x)|Q) \cdot 0 = P(X_{ijk} = x|Q).$$

From $\mathbf{E}[X_{ijk} = x|Q]$ one can compute the expectations $\mathbf{E}[c_{ik0}|Q]$ and $\mathbf{E}[c_{ik1}|Q]$ where $c_{ikx}$ is the number of times a Boolean variable $X_{ijk}$ takes on value $x$ for $x \in \{0, 1\}$ and for all $j \in g(i)$. The expected counts $\mathbf{E}[c_{ik0}]$ and $\mathbf{E}[c_{ik1}]$ are obtained by summing $\mathbf{E}[c_{ik0}|Q]$ and $\mathbf{E}[c_{ik1}|Q]$ over all examples. $P(X_{ijk} = x|Q)$ is given by $\frac{P(X_{ijk}=x,Q)}{P(Q)}$, where $P(X_{ijk} = x, Q)$ and $P(Q)$ can be computed with two traversals of the BDD built for the query $Q$.

The M-step updates the parameters $\pi_{ik}$ for all clauses as:

$$\pi_{ik} = \frac{\mathbf{E}[c_{ik1}]}{\mathbf{E}[c_{ik0}] + \mathbf{E}[c_{ik1}]}$$

for the next EM iteration.

Other EM-based approaches for parameter learning include PRISM (Sato and Kameya 2001), LFI-ProbLog (Gutmann et al. 2011), ProbLog2 (Fierens et al. 2013) and RIB (Riguzzi and Di Mauro 2012).

PRISM imposes a restriction on the kind of allowed programs: the body of clauses sharing an atom in the head must be mutually exclusive. This restriction allows PRISM to avoid using BDDs but severly limits the class of programs to which it can be applied; Example 1, for instance, does not satisfy this requirement, as there are two clauses in the grounding of the program that share the atom *epidemic* (and *pandemic*) in the head but the bodies are not mutually exclusive.

LFI-ProbLog, the most similar to EMBLEM, also performs EM using BDDs but, while LFI-ProbLog builds a BDD for an interpretation that represents the application of the whole

theory to the interpretation, EMBLEM focuses on a target predicate, the one for which we want to obtain good predictions, and builds BDDs starting from ground atoms for the target predicate. ProbLog2 improves on LFI-ProbLog by using d-DNNFs instead of BDDs, a representation that is more succinct than BDDs, but again considers whole interpretations rather than focusing on target predicates.

RIB uses a specialized EM algorithm but is limited to example interpretations sharing the same Herbrand base.

## 3 Multi-armed bandit problem

A multi-armed bandit problem (see Bubeck and Cesa-Bianchi (2012) for a review of stochastic and adversarial bandits) is a sequential allocation problem characterized by a set of arms (choices or actions)[2]. The process sequentially allocates a unit resource to an action obtaining an observable payoff. The aim is to maximize the total obtained payoff. A bandit problem is a sequential decision making problem with limited information where one has to cope with the *exploration versus exploitation dilemma*, since the player should try to balance the exploitation of already known actions having a high payoff and the exploration of other probable profitable actions. The multi-armed bandit problem represents the simplest instance of this dilemma.

The bandit problem may be defined as follows. Given $K \geq 2$ arms, a $K$-armed bandit problem is defined by random variables $X_{i_1}, \ldots, X_{i_n}$, where $i_t \in \{1, \ldots, K\}$ is the index of an arm and $n$ represents the length of the finite horizon, or the rounds. $X_{i_t}$ is the unknown reward associated with arm $i_t$ in round $t$. Successive plays of arm $i$ yield rewards that are independent and identically distributed according to an unknown probability distribution $\nu_i$ on $[0, 1]$, $X_i \sim \nu_i$, with unknown expectation $\mu_i = \mathbb{E}_{X_i \sim \nu_k}[X_i]$ (mean reward of arm $i$). We denote by $i_t$ the arm the player selected at time step $t$, and by $T_i(t) = \sum_{s=1}^{t} \mathbb{I}(i_s = i)$ the number of times the player selected arm $i$ on the first $t$ rounds, where $\mathbb{I}(x)$ is the indicator function that is 1 if $x$ is true and 0 otherwise.

**Definition 1** *(The stochastic bandit problem)*

**Available parameters:** number of arms $K$ and (possibly) number of rounds $n \geq K$
**Unknown parameters:** probability distributions $\nu_1, \ldots, \nu_K$ on $[0, 1]$
**for** each round $t = 1, 2, \ldots, n$ **do**
the player chooses $i_t \in \{1, \ldots, K\}$
the environment draws the reward $x_{i_t t} \sim \nu_{i_t}$ independently from the past
the player receives the reward $x_{i_t t}$
**Objective:** maximize $x_{i_1 1} + \ldots + x_{i_n n}$

A policy is an algorithm that chooses the next arm to play based on the sequence of past plays and obtained rewards. Knowing in advance the arm distributions an optimal policy corresponds to selecting the single arm with the highest mean at each round, obtaining an expected reward of $n\mu^*$ where $\mu^* = \max_{i=1,\ldots,K} \mu_i$. Since the distributions of the arms are

[2] The term *one-armed bandit* is an American slang for slot-machine, while the term *multi-armed bandit* derives from the scenario of a casino where the player faces a row of slot machines when deciding which machines to play. The name is due to Robbins (1952), who pictured a gambler who has the option to play any of a number of slot machines (one-armed bandits) with unknown reward distributions and who wishes to maximize his total expected gain.

unknown, it is necessary to pull each arm several times (*exploration*) and to pull increasingly often the best ones (*exploitation*).

The (expected) regret from which a bandit algorithm (or a policy) suffers with respect to the optimal arm after $n$ rounds is defined by

$$R_n = n\mu^* - \sum_{i=1}^{K} \mu_i \mathbb{E}[T_i(n)],$$

where $\mathbb{E}[T_i(n)]$ denotes the expected number of plays for arm $i$ in the first $n$ rounds.

This defines the loss resulting from not knowing from the beginning the reward distributions. For bandit problems it is useful to know the *upper confidence bound* (UCB) of the mean reward of an arm (i.e., the upper bound of a confidence interval on the mean reward of each arm). Auer et al. (2002) proposed a simple UCB, called UCB1, given by

$$\text{UCB1} = \overline{X}_i + \sqrt{\frac{2 \ln t}{T_i(t)}}, \tag{5}$$

where $\overline{X}_i$ is the average reward obtained from arm $i$. The algorithm that at trial $t$, after playing each arm once for initialization, chooses the arm $i$ that maximizes UCB1 achieves logarithmic regret uniformly over $n$ rounds without any preliminary knowledge about the reward distributions (apart from the fact that their support is in [0, 1]).

This upper confidence bound is used to cope with the exploration-exploitation dilemma, and the corresponding technique converges to the optimal solution for multi-armed bandit problems (Auer et al. 2002).

## 4 Monte Carlo Tree Search (MCTS)

MCTS (see Browne et al. (2012) for a survey) is a family of algorithms aiming at finding optimal decisions by taking random samples in the decision space and by building a search tree in an incremental and asymmetric manner. In each iteration of the algorithm, first a *tree policy* is used in order to find the most urgent node of the tree to expand, trying to balance exploitation and exploration. Then a *simulation* phase is conducted from the selected node, by adding a new child node (obtained with a move from the selected node) and using a *default policy* that suggests the sequence of actions ("simulation") to be chosen from this new node. Finally, the simulation result is *backpropagated* upwards to update the statistics of the nodes that will inform future tree policy decisions.

In computer game-playing MCTS is used in combination with bandit algorithms to explore more efficiently the huge tree of game continuations after a chosen move. Kocsis and Szepesvári (2006) proposed a MCTS strategy for hierarchical bandits called UCT (UCB applied to Trees), derived from the UCB1 bandit algorithm, that led to a substantial advancement in Computer Go performance (Gelly and Wang 2006).

**Algorithm 1** The general MCTS approach

```
1: function MCTS(v_0)
2:     while within a computational budget do
3:         v_l ← TreePolicy(v_0)
4:         Δ ← DefaultPolicy(v_l)
5:         Backup(v_l, Δ)
       return a(BestChild(v_0))
```

The general MCTS algorithm is shown in Algorithm 1, where $v_0$ is the root node of the search tree, $v_l$ is the last node reached during each tree policy iteration, and $\Delta$ is the reward for the terminal state reached by running the default policy from $v_l$. Finally, the action that leads to the best child of the root node $a(\text{BESTCHILD}(v_0))$ is returned. In particular, the tree policy component of the algorithm starts from the root node $v_0$ and then recursively selects child nodes according to some utility function until a node $v_{l-1}$ is reached that either describes a terminal state or is not fully expanded. Now, an unvisited action $a$ from this state is selected and a new leaf node $v_l$ is added to the tree. In the default policy component, a simulation is executed from the new leaf node $v_l$ in order to obtain a reward value $\Delta$, which is then backpropagated in the backup phase up the sequence of nodes selected during the tree policy (i.e., from the newly added node $v_l$ to the root $v_0$) to update their statistics (i.e., incrementing their visit count and updating their average reward according to $\Delta$). In the simplest case, the default policy is uniformly random.

The goal of a MCTS algorithm is to approximate the true values of the moves that may be taken in a given node of the tree. In Kocsis and Szepesvári (2006) the choice of a child node in the tree policy is treated as a multi-armed bandit problem, i.e. the value of a child node is the expected reward approximated by Monte Carlo simulations. In particular, a child $j$ (an arm) of a node $i$ is selected to maximize the following formula:

$$UCT = \begin{cases} \overline{Q}_j + 2C\sqrt{\frac{2\ln N_i}{N_j}} & \text{if } N_j > 0 \\ \text{FPU}_j & \text{otherwise} \end{cases} \tag{6}$$

where $\overline{Q}_j$ is the average reward from arm $j$, $N_i$ is the number of times the current node $i$ (the parent of node $j$) has been visited, $N_j$ is the number of times child $j$ has been visited and $C > 0$ is a constant. Generally, there is no way of determining the unexplored nodes visiting order and typically UCT visits each unvisited node once in random order before revisiting them. To address this issue, *first-play urgency* (FPU) (Gelly and Wang 2006) is used, that assigns a fixed value $FPU_j$ to unvisited nodes (when $N_j = 0$).

The UCT formula in Equation (6) tries to balance exploitation (the first term of the sum) and exploration (the second term of the sum ensuring that each child has a non-zero probability of selection). The constant $C$ in the formula can be set to control the amount of exploration. The value $C = 1/\sqrt{2}$ was shown by Kocsis et al. (2006) to satisfy the Hoeffding inequality[3] with rewards in the range [0, 1].

Each node $v$ is characterized by two values, updated every time $v$ takes part of a tree policy simulation from the root: the number $N_v$ of times it has been visited and a value $Q_v$ that corresponds to the total reward of all *playouts*[4] that passed through the node. Thus, $\overline{Q}_v = Q_v/N_v$ is the average reward obtained from node $v$ and represents an approximation of the theoretic value of $v$.

## 5 LEMUR: learning LPADs as a multi-armed bandit problem

When applying the UCT algorithm for learning the structure of LPADs, we consider each clause to be added to the theory as a bandit problem, where each legal clause revision[5] is

[3] Hoeffding's inequality gives us the upper bound on the probability that the sum of random variables deviates from its expected value.

[4] As in Browne et al. (2012), we understand the terms playout (or simulation) to mean "playing out the task to completion according to the default policy", i.e. the sequence of actions chosen after the tree policy steps of selection and expansion have been completed.

[5] In this paper we consider specializations only as revisions.

an arm with unknown reward. We start from the empty theory and then we iteratively add clauses to it. At each iteration $i$ we start a new execution of a MCTS to find the clause $C$ to be added to the theory $\mathcal{T}_{i-1}$. In particular, each node of the search tree is a clause and its children are its revisions that are selected according to the UCT formula (6). The reward of a clause/arm is computed by learning the parameters of the current theory plus the clause with EMBLEM and using the resulting log-likelihood (LL), appropriately scaled so that it belongs to [0, 1]. During the search, both in the tree and in the default policy, the best clause in terms of LL is included in the current theory. The approach is shown in Algorithm 2.

LEMUR takes as input four parameters: the maximum number $K$ of LPAD clauses to be learned, the number $L$ of UCT rounds, the constant $C$ used in the UCT formula and the maximum random specialization steps $S$ of the default policy. LEMUR starts from an empty theory $\mathcal{T}^*$ (line 1) and then iteratively adds at most $K$ clauses to $\mathcal{T}^*$ (lines 2–16). Each iteration $i$ corresponds to an application of a UCT procedure in order to extend the current best theory $\mathcal{T}^*$ with another clause. The iterative process can be stopped when adding a new clause does not improve the LL. The computational budget for each UCT application corresponds to the execution of $L$ playouts (line 6).

The tree policy in LEMUR (lines 17–20) is implemented as follows. During each iteration, LEMUR starts from the root of the tree corresponding to a clause with an empty body. At each node, LEMUR selects one move by calling the BESTCHILD function (lines 33–34). The move corresponds to a possible clause revision, according to the UCT formula (6). LEMUR then descends to the selected child node and selects a new move until it reaches a leaf. The tree policy part ends by calling the EXPAND function (lines 35–43) that computes all the children of the leaf.

We implemented a first play urgency approach by assigning the LL of the parent node to score all the expanded children nodes. EXPAND then considers the first child of the leaf, it computes its true LL by parameter learning and returns the couple (node, LL). This LL is compared with the current best (line 8) in order to check whether a new optimum has been reached.

Then LEMUR starts the default policy (lines 21–32) consisting of a random sequence of revisions applied to the new leaf node $v_l$ until a *finite unknown horizon* is reached: LEMUR stops the simulation after $k$ steps, where $k$ is a uniformly sampled random integer smaller than the input parameter $S$. Once the horizon is reached, LEMUR produces a reward value $\Delta$ corresponding to the best LL of the theory visited during this random simulation. Logic theories are scored by learning their parameters with EMBLEM and by using the resulting LL.

This reward value $\Delta$ is then backpropagated by calling the procedure BACKUP (lines 44–49) along the sequence of nodes selected for this iteration in order to update their statistics. The LL $LL_{\mathcal{T}}$ of a theory $\mathcal{T}$ computed by EMBLEM is normalized as $Q_{\mathcal{T}} = 1/(1 - LL_{\mathcal{T}})$ in order to keep the values of $\Delta$ and thus of $Q_j$ for each node $j$ within [0, 1].

Clause revisions are performed using a *downward refinement operator* (Nienhuys-Cheng and de Wolf 1997): a function $\rho$ that takes as argument a clause $C$ and returns a set of clauses $\rho(C)$ containing only refinements of $C$ according to theta subsumption, the usual generality order in ILP.

**Algorithm 2** LEMUR($K, L, C, S$)

**Input:** $K$: maximum number of clauses; $L$: number of UCT rounds; $C$: UCT constant; $S$: maximum random specialization steps in the default policy.
**Output:** $\mathcal{T}^*$: the best LPAD Theory

$\mathcal{T}^* \leftarrow \emptyset$; $LL \leftarrow -\infty$
**for** $i = 1, \ldots, K$ **do** ▷ K UCT iterations
  create a root node $v_0$ with a clause with an empty body
  $BC^* \leftarrow v_0$ ▷ best clause
  $LL^* \leftarrow$EMBLEM$((\mathcal{T}^* \cup \{v_0\}))$ ▷ best clause's LL
  **for** $j = 1, \ldots, L$ **do**
    $(w, LL_w) \leftarrow$ TreePolicy$(v_0, \mathcal{T}^*, C)$ ▷ a clause and its corresponding LL
    **if** $LL_w > LL^*$ **then**
      $LL^* \leftarrow LL_w$
      $BC^* \leftarrow w$
    $(BC^*, \Delta) \leftarrow$ DefaultPolicy$(w, \mathcal{T}^*, S, BC^*, LL^*)$
    Backup$(w, \Delta)$
    CheckAMAF$(BC^*, w, \Delta)$
    $LL^* = \Delta$
  **if** $(LL^* - LL) > \epsilon$ **then**
    $\mathcal{T}^* \leftarrow \mathcal{T}^* \cup \{BC^*\}$; $LL \leftarrow LL^*$

**function** TreePolicy$(v, \mathcal{T}, C)$
  **while** $v$ is not a leaf node **do**
    $v \leftarrow$ BestChild$(v, C)$
  **return** Expand$(v, \mathcal{T})$

**function** DefaultPolicy$(v, \mathcal{T}, S, BC, LL)$
  $BC^* \leftarrow BC$
  $\Delta \leftarrow LL$
  $k \leftarrow$ rand$(1, S)$ ▷ finite horizon
  **for** $i = 1 \ldots k$ **do**
    select a revision $r_v$ of $v$ uniformly at random
    $LL =$EMBLEM$(\mathcal{T}^* \cup \{r_v\})$
    **if** $LL > \Delta$ **then**
      $\Delta \leftarrow LL$
      $BC^* \leftarrow r_v$
    $v \leftarrow r_v$
  **return** $(BC^*, \Delta)$

**function** BestChild$(v, C)$
  **return** $\underset{w \in \text{children of } v}{\text{argmax}} \frac{Q_w}{N_w} + 2C\sqrt{\frac{2 \ln N_v}{N_w}}$

**function** Expand$(v, \mathcal{T})$
  add all the child nodes (revisions) to $v$ in the tree
  **for** each child $w$ of $v$ **do**
    $N_w = 1$ ▷ initializations for the FPU
    $Q_w = 1/(1 - LL_v)$
    $LL_w = LL_v$
  $x =$the first child of $v$
  $LL_x =$EMBLEM$(\mathcal{T} \cup \{x\})$
  **return** $(x, LL_x)$

**procedure** Backup$(v, \Delta)$
  **while** $v$ is not the root of the tree **do**
    $N_v \leftarrow N_v + 1$
    $Q_v \leftarrow Q_v + 1/(1 - \Delta)$
    $v \leftarrow$ parent of $v$
  $N_v \leftarrow N_v + 1$ ▷ updates the visits of the root node

**procedure** CheckAMAF$(BC, v, \Delta)$
  **for** each node $u$ of the tree not ancestor of $v$ **do**
    **if** the clause corresponding to $u$ subsumes $BC$ **then**
      $N_u \leftarrow N_u + 1$
      $Q_u \leftarrow Q_u + 1/(1 - \Delta)$

The refinement operator $\rho$ that we use adds a literal to a clause and selects the literal according to a language bias specified in terms of mode declarations. Following Muggleton (1995), a *mode declaration m* is either a head declaration $modeh(r, s)$ or a body declaration $modeb(r, s)$, where $s$, the *schema*, is a ground literal and $r$ is an integer called the *recall*. A schema is a template for literals in the head or body of a clause and can contain special placemarker terms of the form `#type`, `+type` and `-type`, which stand, respectively, for ground terms, input variables and output variables of a type. An input variable in a body literal of a clause must be either an input variable in the head or an output variable in a preceding body literal in the clause. If $M$ is a set of mode declarations, $L(M)$ is the *language of* $M$, i.e. the set of clauses $\{C = h_1; \ldots; h_n :- b_1, \ldots, b_m\}$ such that the head atoms $h_i$ (resp. body literals $b_i$) are obtained from some head (resp. body) declaration in $M$ by replacing all + (resp. -) placemarkers with input (resp. output) variables. Differently from Muggleton (1995), the mode declarations are not used to build a bottom clause from which to extract the literals but these are directly obtained from $L(M)$.

Note that in our case the problem we are solving with UCT can be represented as a directed acyclic graph, since similar clauses can be reached through different sequences of revisions. In other words, our operator is not optimal, in the usual ILP terminology (Nienhuys-Cheng and de Wolf 1997). However, we can not expect to do better as optimal refinement operators do not exist for the language we chose (Nienhuys-Cheng and de Wolf 1997). A way to solve this problem is to consider the enhancement *All Moves As First* (AMAF) first proposed in Gelly and Silver (2007). The AMAF algorithm treats all moves played during a tree policy as if they were played on a previous tree policy. This means that the reward estimate for an action $a$ from a state $s$ is updated whenever $a$ is encountered during a playout, even if $a$ was not the actual move chosen from $s$ (i.e., $a$ is not actually traversed in the selected playout). The AMAF approach implemented in LEMUR is reported in procedure CHECKAMAF (lines 50–54) that updates the statistics of each node $i$ of the tree whose corresponding clause $C_i$ subsumes the one returned by the default policy $C$, since $C$ can be reached also from the node $i$ after a given number of specializations. The algorithm implements the general AMAF procedure, without considering its variants. Furthermore, the independence assumption made for the rewards yielded by the arms is mitigated by the adopted AMAF approach, as reported in the following explanatory example.

Suppose that we have three predicates `a/2`, `p/1` and `q/1`, and we want to predict the predicate `a/2`. The language bias contains a *modeh* declaration, `modeh(*,a(+1,+1))`, and *modeb* declarations such as `modeb(*,p(+1))` and `modeb(*,q(+1))`.

Figure 2 shows snapshots of LEMUR's learning states. Each node in the figure is labeled with its corresponding literal plus the cumulative reward over the number of visits. Figure 2a reports the selection process of the Tree Policy phase where, starting from the root, a leaf is reached by making the best choice for each node, according to the UCT formula. When on a leaf node, the expansion process is executed (Fig. 2b) and the simulation phase follows (Fig. 2c). By looking at Fig. 2d, the expansion has generated two nodes, $q(A)$ and $p(B)$. As we already said, we implemented a first-play urgency (FPU) approach by using the LL of the parent node to score all the expanded children nodes that are not used to start the simulation. Now, the reward value corresponding to the best LL of the theory visited during the simulation (the downward zig zag arrow starting from $p(B)$ in Fig. 2c) is backpropagated to the parent nodes. The clause corresponding to the last leaf node from which the backpropagation started is subsumed by other clauses in the tree and hence the AMAF procedure updates their values.

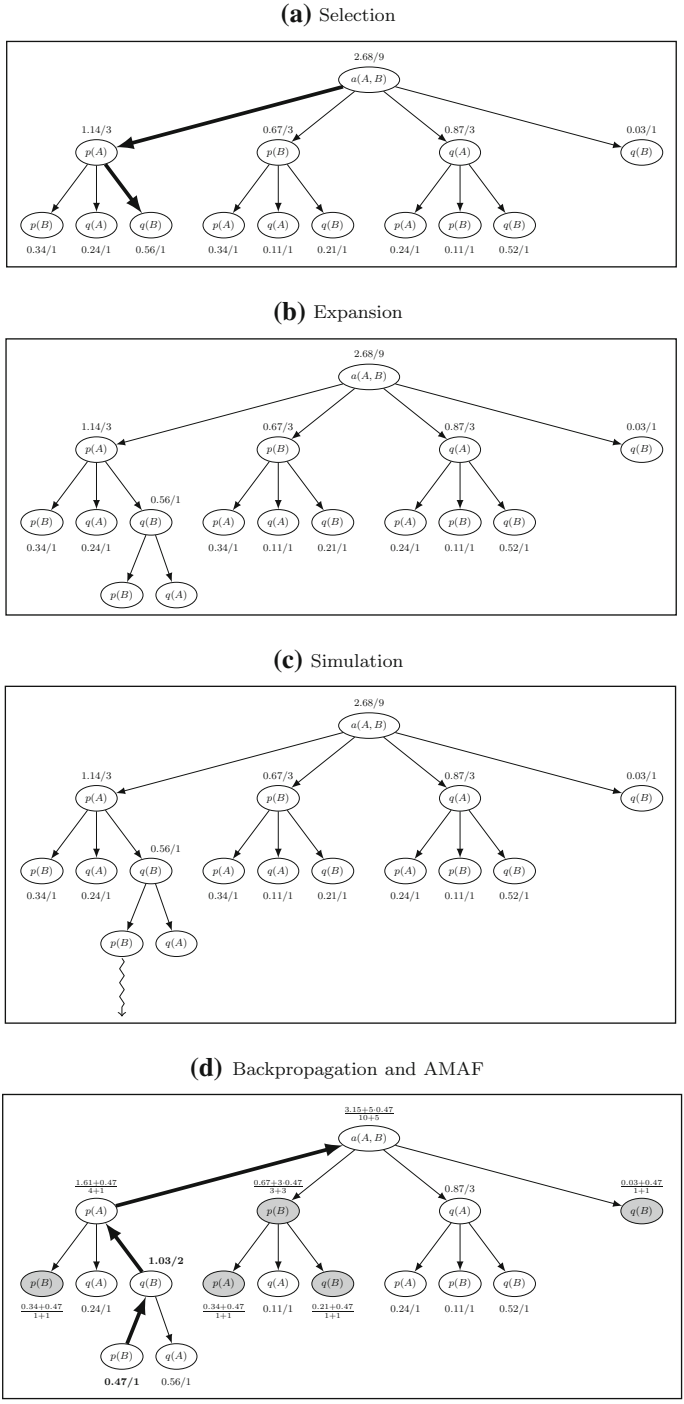


**Fig. 2** States of LEMUR's Tree Search. **a** Selection. **b** Expansion. **c** Simulation. **d** Backpropagation and AMAF

### 5.1 Execution example

We now show an execution example for the UW-CSE dataset, used in the experiments discussed in Sect. 7. UW-CSE describes the Computer Science department of the University of Washington with 22 different predicates, such as `advisedby/2`, `yearsinprogram/2` and `taughtby/3`. The aim is to predict the predicate `advisedby/2`, namely the fact that a person is advised by another person. The language bias contains *modeh* declarations such as `modeh(*,advisedby(+person,+person))` and *modeb* declarations such as `modeb(*,courselevel(+course, -level))`.

The first clause may be obtained by the first UCT application (line 2 of Algorithm 2 ($i = 1$)) as follows. LEMUR starts with a tree having an empty body clause as the root node. The first application of the TREEPOLICY function (line 7 of Algorithm 2) corresponds to the application of the EXPAND function on the empty clause and then to the execution of the DEFAULTPOLICY, which returns the following new best clause with LL −246.51:

```
advisedby(A, B):0.5 :- professor(B).
```

In a further application (line 7–16 of Algorithm 2) of both the TREEPOLICY and DEFAULTPOLICY functions, the node of the tree corresponding to the following clause having a LL equal to −215.46 could be reached:

```
advisedby(A, B):0.1089 :- professor(B), student(A).
```

No more clauses with a better LL are found in this first iteration.

The second iteration (line 2 of Algorithm 2 ($i = 2$)) then starts with a theory containing the best clause obtained in the previous one. Applications of the TREEPOLICY and the DEFAULTPOLICY functions find a clause that when added to the current theory gives the following one with a LL equal to −189.50:

```
advisedby(A,B):0.11536 :- professor(B), student(A).
advisedby(A,B):0.285709 :- student(A), publication(C,A),
                           professor(B), publication(C,B).
```

## 6 Related work

The idea of applying a MCTS algorithm to Machine Learning problems is not new. Indeed, MCTS has been recently used by Gaudel and Sebag (2010) in their FUSE (Feature Uct SElection) system to perform feature selection, and by Rolet et al. (2009) in BAAL (Bandit-based Active Learner) for active learning with small training sets. Gaudel and Sebag (2010) firstly formalize feature selection as a Reinforcement Learning (RL) problem and then provide an approximation of the optimal policy by casting the RL problem as a one-player game whose states are all possible subsets of features and whose actions consist of choosing a feature and adding it to a subset. The problem is then solved with the UCT approach leading to the FUSE algorithm. Rolet et al. (2009) focus on Active Learning (AL) with a limited number of queries. The authors formalized AL under bounded resources as a finite horizon RL problem. Then they proposed an approximation of the optimal policy leading to the BAAL algorithm that combines UCT and billiard algorithms (Rujan 1997).

Previous work on learning the structure of probabilistic logic programs includes Kersting and De Raedt (2008), that proposed a scheme for learning both the probabilities and the structure of Bayesian logic programs by combining techniques from the learning from interpretations setting of ILP with score-based techniques for learning Bayesian networks. We share with this approach the scoring function, the LL of the data given a candidate structure, and the greedy search in the space of structures.

Paes et al. (2005) perform theory revision of Bayesian logic programs using a variety of heuristic functions, including the LL of the examples. LEMUR differs from this work because it searches the clause space rather than the theory space.

Early systems for learning the structure of LPADs are LLPAD (Riguzzi 2004) and its successor ALLPAD (Riguzzi 2007a, 2008) that however are restricted to learning ground programs with mutually exclusive clauses.

De Raedt et al. (2008b) presented an algorithm for performing theory compression on ProbLog programs. Theory compression means removing as many clauses as possible from the theory in order to maximize the likelihood w.r.t. a set of positive and negative examples. No new clause can be added to the theory.

SEM-CP-logic (Meert et al. 2008) learns parameters and structure of ground CP-logic programs. It performs learning by considering the Bayesian networks equivalent to CP-logic programs and by applying techniques for learning Bayesian networks. In particular, it applies the Structural Expectation Maximization (SEM) algorithm (Friedman 1998): it iteratively generates refinements of the equivalent Bayesian network and it greedily chooses the one that maximizes the BIC score (Schwarz 1978). LEMUR differs from SEM-CP-logic because it searches the clause space instead of the theory space and it refines clauses with standard ILP refinement operators, which allows it to learn non ground theories.

More recently, SLIPCASE (Bellodi and Riguzzi 2012) can learn probabilistic logic programs without these restrictions. It is based on a simple beam search strategy in the space of possible theories, that refines LPAD programs by trying all possible theory revisions. It exploits the LL of the data as the guiding heuristics. The beam is initialized with a number of trivial theories that are repeatedly revised using theory revision operators: the addition/removal of a literal from a clause and the addition/removal of a whole clause. Each refinement is scored by learning the parameters with EMBLEM. LEMUR differs from SLIPCASE because it searches the space of clauses and does it using an approximate search method.

SLIPCOVER (Bellodi and Riguzzi 2014) learns the structure of probabilistic logic programs with a two-phase search strategy: (1) beam search in the space of clauses in order to find a set of promising clauses and (2) greedy search in the space of theories. In the first phase, SLIPCOVER generates refinements of a *single* clause at a time starting from a bottom clause built as in Progol (Muggleton 1995), which are evaluated through LL. In the second phase, the search in the space of theories starts from an empty theory which is iteratively extended with one clause at a time from those generated in the previous beam search. Background clauses, the ones with a non-target predicate in the head, are treated separately, by adding them en bloc to the best theory for target predicates. A further parameter optimization step is executed with EMBLEM and clauses that are never involved in a target predicate goal derivation are removed. LEMUR differs from SLIPCOVER for the use of a MCTS search strategy rather than a beam search. Moreover, there is no separate search for clauses and for theories, since clauses are learned one by one adding each one to the current theory. Thus, clauses are not evaluated in isolation as in SLIPCOVER but are scored together with the current theory. The only random component of SLIPCOVER is the selection of the seed example for building the bottom clauses, while randomization is a crucial component of LEMUR default policy.

Structure learning has been thoroughly investigated for Markov Logic. Mihalkova and Mooney (2007) proposed a bottom-up algorithm (BUSL) for learning Markov Logic Networks (MLNs) that is based on relational pathfinding: paths of true ground atoms that are linked via their arguments are found and generalized into first-order rules. Huynh and Mooney (2008) introduced a two-step method (ALEPH++ExactL1) for inducing the structure of

MLNs: (1) learning a large number of promising clauses through a specific configuration of Aleph[6] (ALEPH++), followed by (2) the application of a new discriminative MLN parameter learning algorithm. This algorithm differs from the standard weight learning one (Lowd and Domingos 2007) in the use of an exact probabilistic inference method and of a L1-regularization of the parameters, in order to encourage assigning low weights to clauses. Kok and Domingos (2010) presented the algorithm "Learning Markov Logic Networks using Structural Motifs" (LSM). It is based on the observation that relational data frequently contain recurring patterns of densely connected objects called structural motifs. LSM limits the search to these patterns. LSM views a database as a hypergraph and groups nodes that are densely connected by many paths and the hyperedges connecting the nodes into a motif. Then it evaluates whether the motif appears frequently enough in the data and finally it applies relational pathfinding to find rules. This process, called *createrules*, is followed by weight learning with the Alchemy system.

A set of recent boosting approaches have been proposed to reduce structure learning of probabilistic relational models to relational regression (Khot et al. 2011; Natarajan et al. 2012). Khot et al. (2011) turned the problem of learning MLNs into a series of relational functional approximation problems, using two kinds of representations for the gradients on the pseudo-likelihood: clause-based (MLN-BC) and tree-based (MLN-BT). At each gradient step, the former version simply learns a set of Horn clauses with an associated regression value, while the latter version views MLNs as a set of relational regression trees, in which each path from the root to a leaf can be seen as a clause and the regression values in the leaves are the clause weights. The goal is to minimize the squared error between the potential function and the functional gradient over all training examples. Natarajan et al. (2012) turned the problem of learning Relational Dependency Networks (RDNs) into a series of relational function approximation problems using Friedman's functional gradient-based boosting. The algorithm is called RDN-B. RDNs approximate the joint distribution of a relational model as a product of conditional distributions over ground atoms. They consider the conditional probability distribution of each predicate as a set of relational regression trees each of which approximates the corresponding gradient. These regression trees serve as the individual components of the final potential function. They are learned such that at each iteration the new set of regression trees aims at maximizing the likelihood. The different regression trees provide the structure of the conditional distributions while the regression values at the leaves form the parameters of the distributions.

Poor scalability in problems with large search spaces and many examples are two challenges faced by many ILP systems employing deterministic search methods. These challenges have been successfully addressed by randomizing the search with Stochastic Local Search (SLS) procedures (Hoos and Stützle 2004) as in Paes et al. (2008); Železný et al. (2002) and Zelezný et al. (2006). In a SLS algorithm, one starts by selecting an initial candidate solution and then iteratively moving from one candidate solution to a neighboring candidate solution. The decision at each search step is based on local knowledge. Both the decisions as well as the search initializations can be randomized. In particular, Paes et al. (2008) apply Stochastic Local Search for first-order theory revision from examples, starting from the implementation of the FORTE system (Richards and Mooney 1995). SLS relies on randomized decisions while searching for solutions: stochastic or greedy moves are chosen according to a fixed probability p. Stochastic moves include randomization of antecedent search (for clause-level refinement) and randomization of revision search (for theory-level refinement).

[6] http://www.cs.ox.ac.uk/activities/machlearn/Aleph/.

Železný et al. (2002) propose to randomize the lattice search of clauses: for a maximum number of tries (restarts), the algorithm begins by randomly selecting a starting clause (seed), generating its subsequent deterministic refinements through a non-traditional refinement operator producing a "radial" lattice, and returning the first clause satisfying conditions on the minimal accuracy. This implementation is called randomized rapid restarts (RRR).

In order to be informative during the search, a local search algorithm should use an evaluation function mapping each search position onto a real number. In case of ILP this corresponds to use a well-known expensive evaluation function for clauses/theories.
In contrast, MCTS yields a larger and unbiased sample of the search neighborhood, and requires state evaluations only at the endpoints of each playout. Its evaluation function depends only on the observed outcomes of the playout, and continues to improve from additional playouts (the lack of need for domain-specific knowledge is one of its most significant benefits). In our specific case, each move is not estimated with a local inspection of the neighborhood that may lead to a local optimum, but with a deep inspection of the tree.

MCTS develops in a highly selective, best-first manner, expanding promising regions of the search space much more deeply. Instead of randomly searching promising solutions, MCTS tries to discover the optimal policy (the revision steps in our case) leading to the optimal solutions.

Furthermore, SLS algorithms often suffer from getting trapped in a local optimum (*local optimum stagnation*), a problem non existing for systematic search algorithms. A common solution to the stagnation problem is to restart the algorithm when no progress is observed (Martí et al. 2010). On the other side the UCT algorithm for MCTS uses an upper confidence bound exploration/exploitation technique. This upper confidence bound technique converges to the optimum solution for multi-armed bandit problems (Auer et al. 2002).

## 7 Experimental validation

The proposed LEMUR system has been implemented in Yap Prolog (Santos Costa et al. 2012) and has been compared with SLIPCASE, SLIPCOVER (for PLPs), BUSL, LSM, ALEPH++ExactL1, MLN-BC, MLN-BT (for MLNs), RDN-B (for RDNs), SLS and RRR (for structure learning by randomized search techniques). All experiments were performed on GNU/Linux machines with an Intel Core 2 Duo E6550 (2,333 MHz) processor and 4 GB of RAM. The systems have been tested on the following seven real world datasets: UW-CSE, Mutagenesis, Hepatitis, Carcinogenesis, IMDB, Mondial and HIV. To evaluate the performance, we computed the log likelihood over the test examples and we drew Precision-Recall curves and Receiver Operating Characteristics curves, computing the Area Under the Curve (AUC-PR and AUC-ROC respectively) with the methods reported in Davis and Goadrich (2006); Fawcett (2006). For each dataset we report the results for those systems that completed the task successfully. The missing results indicate an out-of-memory error. Statistics on all the domains are reported in Table 1. The number of negative testing examples is sometimes different from that of negative training examples because, while for training we explicitly provide negative examples, for testing we consider all the ground instantiations of the target predicates that are not positive as negative.

### 7.1 Parameter settings

LEMUR requires four input parameters: the maximum number $K$ of clauses to be learned, the number $L$ of UCT rounds, the UCT constant $C$ and the maximum number $S$ of random

**Table 1** Characteristics of the datasets for the experiments: target predicates (Target), number of constants (C), of predicates (P), of tuples (T) (i.e., ground atoms), of positive (TrPEx) and negative training and testing (TrNEx and TeNEx) examples for target predicate(s), of folds (F)

| Dataset | Target | C | P | T | TrPEx | TrNEx | TeNEx | F |
|---|---|---|---|---|---|---|---|---|
| UW-CSE | Advisedby | 1323 | 15 | 2673 | 113 | 4079 | 16601 | 5 |
| Mutagen | Active | 7045 | 20 | 15249 | 125 | 63 | 63 | 10 |
| Carcinogen | Active | 13835 | 36 | 24533 | 182 | 136 | 19 | 1 |
| Hepatitis | Type | 6491 | 19 | 71597 | 500 | 500 | 500 | 5 |
| IMDB | WorkedUnder | 316 | 6 | 1540 | 382 | 14236 | 14236 | 5 |
| Mondial | ChristianReligion | 3012 | 11 | 10985 | 572 | 308 | 308 | 5 |
| HIV | 41L,67N,70R, 210W,215FY,219EQ | 0 | 6 | 2184 | 590 | 1594 | 1594 | 5 |

The number of tuples includes the target positive examples

specialization steps in the default policy. For all the datasets we set the $C$ constant to $0.7 \approx 1/\sqrt{2}$ as indicated in Kocsis et al. (2006). The other parameters were set as $S = 8$, $L = 200$ and $K = 20$. These values were chosen in order to allow for a sufficiently deep search and a sufficiently complex target theory while containing the computation time. We also experimented with other parameter values and we obtained similar results, showing that LEMUR is not extremely sensitive to parameter values.

SLIPCASE requires the following parameters: *NIT*, the number of theory revision iterations, *NR*, the maximum number of rules in a learned theory, *NB*, the size of the beam, *NV*, the maximum number of variables in a rule, $\epsilon_s$ and $\delta_s$, respectively the minimum difference and relative difference between the LL of the theory in two refinement iterations. We set $\epsilon_s = 10^{-4}$ and $\delta_s = 10^{-5}$ in all experiments except Mutagenesis, where we used $\epsilon_s = 10^{-20}$ and $\delta_s = 10^{-20}$. Moreover, we set $NIT = 10$, $NB = 20$, $NV = 4$ or 5, $NR = 10$ in all experiments except Carcinogenesis, where we allowed a greater beam ($NB = 100$) to take into account more rules for the final theory.

SLIPCOVER offers the following parameters: the number *NInt* of mega-examples on which to build the bottom clauses, the number *NA* of bottom clauses to be built for each mega-example, the number *NS* of saturation steps (for building the bottom clauses), the maximum number *NI* of clause search iterations, the size *NB* of the beam, the maximum number *NV* of variables in a rule, the maximum numbers *NTC* and *NBC* of target and background clauses respectively. We set $NInt = 1$ or 4, $NS = 1$, $NA = 1$, $NI = 10$, $NV = 4$ or 5, $NB = 10$, 20 or 100, $NTC = 50$, 100, 1000 or 10000, $NBC = 50$ (only UW-CSE) according to the dataset.

LEMUR, SLIPCASE and SLIPCOVER share the parameter learning algorithm EMBLEM. For the EM cycle performed by EMBLEM, we set the maximum number of iterations NEM to $\infty$ (since we observed that it usually converged quickly) and the stopping criteria $\epsilon$ to $10^{-4}$ and $\delta$ to $10^{-5}$. EMBLEM stops when the difference between the LL of the current iteration and the previous one drops below the threshold $\epsilon$ or when this difference is below a fraction $\delta$ of the current LL.

As regards LSM, we used the default parameters in the structure learning phase and the discriminative algorithm in the weight learning phase on all datasets except for HIV, where the generative option was set; we specified the target predicates (see Table 1) as the non-evidence predicates. Only for IMDB we set the $\pi$ parameter to 0.1 as indicated in Kok and Domingos (2010). For testing, we applied the MC-SAT algorithm in the inference phase by specifying the target predicates as the query predicates.

For BUSL, we specified the target predicates as the non-evidence predicates in the learning step and the `startFromEmptyMLN` flag to start structure learning from an empty MLN. Moreover, we set the `minWeight` parameter to 0.5 for IMDB and UW-CSE as indicated in Mihalkova and Mooney (2007); for Hepatitis only the mandatory parameters were used. For testing we applied the same inference method as LSM.

For ALEPH++ExactL1 we used the `induce_cover` command and the parameter settings for Aleph specified in Huynh and Mooney (2008) on the datasets on which Aleph could return a theory.

For MLN-BT we used the `trees` parameter (number of boosting trees) and for MLN-BC the `trees`, `numMLNClause` (number of clauses learned during each gradient step) and `mlnClauseLen` (length of the clauses) parameters with the values indicated in Khot et al. (2011).

For RDN-B, we set the depth of the trees, the number of leaves in each tree, the maximum number of literals in the nodes as indicated in Natarajan et al. (2012) for the common datasets (UW-CSE, IMDB), while we increased the tree depth for Carcinogenesis, Mondial, Mutagenesis for building longer clauses. We present results for all the boosting methods both with sampling of negative examples (twice the positives) and without it. Execution with sampling is indicated with "sam." in the Tables.

For RRR, we used the ILP system Aleph (Srinivasan (2012)) with the appropriate configuration for randomized search (`search` parameter is set to `rls` (randomized local search) and `rls_type` is set to `rrr` (randomized rapid restarts)). Moreover, we set the parameters `tries` (maximum number of restarts) to 10, `evalfn` (evaluation function) to `accuracy`, `minacc` (lower bound on the minimum accuracy of an acceptable clause) to 0.7 and 0.9, `clauselength` (number of literals in an acceptable clause) to 5 and `minpos` to 2 (lower bound on the number of positive examples to be covered by an acceptable clause; it prevents Aleph from adding ground unit clauses to the theory) as indicated in Železný et al. (2002). Results for the two `minacc` values are reported as "RRR 0.7" and "RRR 0.9" in the Tables.

For SLS, we implemented hill-climbing stochastic local search in Yap Prolog, for both antecedent and revision search using accuracy as the evaluation function, as described in Paes et al. (2008). All stochastic parameters were set to 0.5. The starting theory is composed of one definite clause with the target predicate in the head and an empty body.

Since RRR and SLS return non-probabilistic theories, for testing we annotated the head of each learned clause with probability 0.5, in order to turn the sharp logical classifier into a probabilistic one and to assign higher probability to those examples that have more successful derivations.

### 7.2 Results

In the following we report the results obtained for each dataset together with a brief description of them. Tables 2, 3, 4, 5, 6, 7 and 8 show the results in terms of average AUC-PR and AUC-ROC for all datasets. Table 9 shows the average log likelihood over the test examples. Table 10 shows LEMUR performance as the depth of exploration S varies. Tables 11 and 12 show the $p$ value of a paired two-tailed $t$-test of the difference in AUC-PR and AUC-ROC between LEMUR and the other systems on all datasets (except Carcinogenesis where we did not apply cross-validation).

**Table 2** Results of the experiments in terms of average AUC-PR, AUC-ROC and execution time (in seconds) on the UW-CSE dataset

| System | AUC-PR | AUC-ROC | Time(s) |
|---|---|---|---|
| RDN-B | $0.276 \pm 0.065$ | $0.957 \pm 0.015$ | 372.3 |
| RDN-B sam. | $0.265 \pm 0.100$ | $0.957 \pm 0.011$ | 79.5 |
| LEMUR | $0.230 \pm 0.078$ | $0.944 \pm 0.023$ | 667.6 |
| MLN-BT | $0.179 \pm 0.066$ | $0.930 \pm 0.063$ | 1233.1 |
| MLN-BT sam. | $0.151 \pm 0.126$ | $0.931 \pm 0.018$ | 386.7 |
| RRR 0.7/0.9 | $0.135 \pm 0.127$ | $0.575 \pm 0.059$ | 220.9 |
| MLN-BC sam. | $0.117 \pm 0.120$ | $0.949 \pm 0.015$ | 49.9 |
| SLIPCOVER | $0.113 \pm 0.083$ | $0.949 \pm 0.012$ | 143.3 |
| LSM | $0.071 \pm 0.053$ | $0.845 \pm 0.177$ | 9951.9 |
| MLN-BC | $0.061 \pm 0.015$ | $0.872 \pm 0.077$ | 337.1 |
| ALEPH++ExactL1 | $0.047 \pm 0.011$ | $0.580 \pm 0.057$ | 218.2 |
| SLIPCASE | $0.035 \pm 0.005$ | $0.894 \pm 0.025$ | 64.8 |
| BUSL | $0.015 \pm 0.012$ | $0.621 \pm 0.151$ | 53790.8 |
| SLS | $0.007 \pm 0.002$ | $0.500 \pm 0.000$ | 0.592 |

The standard deviations are also shown

### *7.2.1 UW-CSE*

The UW-CSE dataset[7] (Kok and Domingos 2005) contains information about the Computer Science department of the University of Washington, and is split into five mega-examples, each containing facts for a particular research area. The goal is to predict the target predicate `advisedby(person1,person2)`, namely the fact that a person is advised by another person. We employed a five-fold cross validation where we learn from four areas and predict on the remaining area.

Table 2 reports the AUC-PR and AUC-ROC, along with the training time taken by each system averaged over the five folds.

LEMUR is only outperformed by RDN-B in AUC-PR, and by MLN-BC (with sampling), RDN-B and SLIPCOVER in AUC-ROC, with non-significant differences in both cases. The two systems closer to LEMUR in terms of AUC-PR are the boosting approaches.

SLS low performance is due to the fact that it returns empty theory on this dataset.

### *7.2.2 Mutagenesis*

The Mutagenesis dataset[8] (Srinivasan et al. 1996) contains information about a number of aromatic and heteroaromatic nitro drugs, including their chemical structures in terms of atoms, bonds and a number of molecular substructures such as five- and six-membered rings, benzenes, phenantrenes and others. The fundamental Prolog facts are `bond(compound,atom1,atom2,bondtype)`, stating that in the `compound` a bond of type `bondtype` can be found between the atoms `atom1` and `atom2`, and `atm(compound,atom,element,atomtype,charge)`, stating that a `compound`'s `atom` is of element `element`, is of type `atomtype` and has partial charge `charge`. From

[7] http://alchemy.cs.washington.edu/data/uw-cse.

[8] http://www.doc.ic.ac.uk/~shm/mutagenesis.html.

**Table 3** Results of the experiments in terms of average AUC-PR, AUC-ROC and execution time (in seconds) on the Mutagenesis dataset

| System | AUC-PR | AUC-ROC | Time(s) |
|---|---|---|---|
| RDN-B | $0.964 \pm 0.026$ | $0.920 \pm 0.060$ | 70.0 |
| RDN-B sam. | $0.964 \pm 0.034$ | $0.928 \pm 0.060$ | 67.5 |
| LEMUR | $0.952 \pm 0.062$ | $0.935 \pm 0.048$ | 11229.9 |
| SLIPCOVER | $0.951 \pm 0.050$ | $0.885 \pm 0.141$ | 75327.7 |
| ALEPH++ExactL1 | $0.949 \pm 0.043$ | $0.905 \pm 0.050$ | 198.1 |
| MLN-BT | $0.922 \pm 0.087$ | $0.867 \pm 0.070$ | 175.2 |
| SLIPCASE | $0.921 \pm 0.087$ | $0.873 \pm 0.141$ | 5135.3 |
| RRR 0.7 | $0.891 \pm 0.079$ | $0.777 \pm 0.120$ | 28.2 |
| MLN-BT sam. | $0.872 \pm 0.111$ | $0.823 \pm 0.102$ | 181.237 |
| RRR 0.9 | $0.862 \pm 0.077$ | $0.750 \pm 0.111$ | 29.26 |
| MLN-BC sam. | $0.831 \pm 0.111$ | $0.741 \pm 0.123$ | 44.6 |
| MLN-BC | $0.690 \pm 0.201$ | $0.553 \pm 0.162$ | 54.5 |
| SLS | $0.665 \pm 0.122$ | $0.500 \pm 0.000$ | 1.5 |

The standard deviations are also shown

these facts many elementary molecular substructures can be defined, and we used the tabulation of these, available in the dataset, rather than the clause definitions based on `bond/4` and `atm/5`. This greatly sped up learning.

The problem here is to predict the mutagenicity of the drugs. The prediction of mutagenesis is important as it is relevant to the understanding and prediction of carcinogenesis. The subset of the compounds having positive levels of log mutagenicity are labeled *active* and constitute the positive examples, the remaining ones are *inactive* and constitute the negative examples. The dataset is split into two subsets (188+42 examples). We considered the first one, composed of 125 positive and 63 negative compounds. The goal is to predict if a drug is active, so the target predicate is `active(drug)`. We employed a ten-fold cross validation.

Table 3 presents the AUC-PR and AUC-ROC, along with the training time taken by each system averaged over the ten folds. LEMUR is only outperformed by RDN-B in AUC-PR with non-significant difference, and reaches the highest AUC-ROC value with a significant difference in many cases.

### *7.2.3 Hepatitis*

The Hepatitis dataset[9] (Khosravi et al. 2012) is derived from the ECML/PKDD 2002 Discovery Challenge Workshop held during the 13th ECML/6th PKDD conference. It contains information on the laboratory examinations of hepatitis B and C infected patients. Seven tables are used to store this information.

The goal is to predict the type of hepatitis of a patient, so the target predicate is `type(patient,type)` where `type` can be `b` or `c`. Positive examples for a type are considered as negative examples for the other type. We employed a five-fold cross validation.

As can be seen from Table 4, LEMUR performs better than the other learning techniques both for AUC-PR and AUC-ROC and the difference is statistically significant in all cases. RRR low performance is due to the fact that it returns empty theory on this dataset.

[9] http://www.cs.sfu.ca/~oschulte/jbn/dataset.html.

**Table 4** Results of the experiments in terms of average AUC-PR, AUC-ROC and execution time (in seconds) on the Hepatitis dataset

| System | AUC-PR | AUC-ROC | Time(s) |
|---|---|---|---|
| LEMUR | $0.905 \pm 0.031$ | $0.890 \pm 0.031$ | 3342.7 |
| RDN-B | $0.879 \pm 0.006$ | $0.877 \pm 0.004$ | 310.6 |
| RDN-B sam. | $0.805 \pm 0.023$ | $0.793 \pm 0.024$ | 165.4 |
| SLIPCOVER | $0.799 \pm 0.011$ | $0.743 \pm 0.013$ | 264 |
| MLN-BC | $0.786 \pm 0.021$ | $0.749 \pm 0.015$ | 9569.7 |
| MLN-BC sam. | $0.786 \pm 0.012$ | $0.749 \pm 0.017$ | 4563.7 |
| MLN-BT | $0.779 \pm 0.022$ | $0.760 \pm 0.021$ | 682.3 |
| MLN-BT sam. | $0.772 \pm 0.051$ | $0.765 \pm 0.036$ | 1107.9 |
| SLIPCASE | $0.712 \pm 0.056$ | $0.665 \pm 0.054$ | 132.9 |
| SLS | $0.586 \pm 0.029$ | $0.567 \pm 0.023$ | 0.735 |
| LSM | $0.531 \pm 0.053$ | $0.526 \pm 0.038$ | 89997.7 |
| BUSL | $0.513 \pm 0.028$ | $0.527 \pm 0.020$ | 34771.8 |
| RRR 0.7/0.9* | $0.500 \pm 0.000$ | $0.500 \pm 0.000$ | 248 |

The standard deviations are also shown

### *7.2.4 Carcinogenesis*

This dataset[10] describes more than 300 compounds that have been shown to be carcinogenic or otherwise in rodents (Srinivasan et al. 1997). In particular, it is composed of 182 positive and 155 negative compounds. The chemicals were selected on the basis of their carcinogenic potential (for example, positive mutagenicity tests) and of evidence of substantial human exposure. Similarly to the Mutagenesis dataset, the background knowledge describes molecules in terms of their atoms and bonds, chemical features and three-dimensional structure; in particular the predicates `atm/5, bond/4, gteq/2, lteq/2, =/2` are common to both domains. Moreover, it contains the results of bio-assays about genotoxicity of the chemicals.

The goal is to predict the carcinogenic activity of the compounds, so the target predicate is `active(drug)`. In this case we did not apply cross-validation but we kept the partition into training and test sets already present in the original data.

Table 5 shows that LEMUR is only outperformed by SLS in AUC-PR and by ALEPH++ExactL1 in AUC-PR and AUC-ROC.

### *7.2.5 IMDB*

This is a standard dataset[11] (Mihalkova and Mooney 2007) describing a movie domain. It contains six predicates: `actor/1`, `director/1`, `movie/2`, `genre/2`, `gender/2` and `workedUnder/2`. We used `workedUnder/2` as target predicate. Since the predicate `gender(person,gender)` can take only two values, we converted it to a single argument predicate `female(person)`. We omitted the four equality predicates and we performed a five-fold cross-validation using the five available folds, then we averaged the results over all the folds.

[10] http://www.cs.ox.ac.uk/activities/machlearn/cancer.html.

[11] http://alchemy.cs.washington.edu/data/imdb/.

**Table 5** Results of the experiments in terms of AUC-PR, AUC-ROC and execution time (in seconds) on the Carcinogenesis dataset

| System | AUC-PR | AUC-ROC | Time(s) |
|---|---|---|---|
| ALEPH++ExactL1 | 0.738 | 0.725 | 146.7 |
| SLS | 0.709 | 0.643 | 1.085 |
| LEMUR | 0.691 | 0.721 | 23435.5 |
| MLN-BC sam. | 0.663 | 0.641 | 709.3 |
| SLIPCASE | 0.628 | 0.618 | 1.1 |
| MLN-BC | 0.619 | 0.632 | 45.3 |
| RRR 0.7 | 0.606 | 0.604 | 90.6 |
| SLIPCOVER | 0.600 | 0.676 | 17586.8 |
| RRR 0.9 | 0.594 | 0.588 | 100.4 |
| RDN-B sam. | 0.555 | 0.520 | 82.8 |
| RDN-B | 0.550 | 0.525 | 84.1 |
| MLN-BT | 0.503 | 0.361 | 114.1 |
| MLN-BT sam. | 0.494 | 0.441 | 358.9 |

**Table 6** Results of the experiments in terms of average AUC-PR, AUC-ROC and execution time (in seconds) on the IMDB dataset

The standard deviations are also shown

| System | AUC-PR | AUC-ROC | Time(s) |
|---|---|---|---|
| LEMUR | $1.000 \pm 0.000$ | $1.000 \pm 0.000$ | 1781 |
| ALEPH++ExactL1 | $1.000 \pm 0.000$ | $1.000 \pm 0.000$ | 8.5 |
| LSM | $1.000 \pm 0.000$ | $1.000 \pm 0.000$ | 971.3 |
| RDN-B | $1.000 \pm 0.000$ | $1.000 \pm 0.000$ | 198.8 |
| RDN-B sam. | $1.000 \pm 0.000$ | $1.000 \pm 0.000$ | 43.4 |
| SLIPCOVER | $1.000 \pm 0.000$ | $1.000 \pm 0.000$ | 89.8 |
| SLIPCASE | $1.000 \pm 0.000$ | $1.000 \pm 0.000$ | 64.4 |
| RRR 0.7 | $1.000 \pm 0.000$ | $1.000 \pm 0.000$ | 19.7 |
| RRR 0.9 | $1.000 \pm 0.000$ | $1.000 \pm 0.000$ | 20.1 |
| MLN-BT | $0.977 \pm 0.047$ | $0.999 \pm 0.003$ | 459.2 |
| MLN-BT sam. | $0.977 \pm 0.047$ | $0.999 \pm 0.003$ | 130.4 |
| MLN-BC sam. | $0.968 \pm 0.005$ | $0.998 \pm 0.063$ | 93.5 |
| MLN-BC | $0.942 \pm 0.071$ | $0.996 \pm 0.005$ | 266.2 |
| BUSL | $0.030 \pm 0.012$ | $0.487 \pm 0.019$ | 35124.4 |
| SLS | $0.025 \pm 0.012$ | $0.500 \pm 0.000$ | 1.8 |

Table 6 presents the results in terms of AUC-PR and AUC-ROC values for the target predicate `workedUnder/2`, showing that LEMUR achieves perfect classification as all other systems except MLB-BT, MLN-BC, BUSL and SLS, against which the difference is significant in almost all cases. SLS low performance is due to the fact that it returns empty theory on this dataset.

### *7.2.6 Mondial*

This dataset contains geographical data from multiple web sources (May 1999). The dataset features information regarding geographical regions of the world, including pop-

**Table 7** Results of the experiments in terms of average AUC-PR, AUC-ROC and execution time (in seconds) on the Mondial dataset

| System | AUC-PR | AUC-ROC | Time(s) |
|---|---|---|---|
| ALEPH++ExactL1 | $0.867 \pm 0.068$ | $0.777 \pm 0.078$ | 15.1 |
| LEMUR | $0.864 \pm 0.066$ | $0.782 \pm 0.038$ | 22 |
| SLIPCOVER | $0.862 \pm 0.074$ | $0.797 \pm 0.112$ | 15.7 |
| RRR 0.9 | $0.833 \pm 0.048$ | $0.729 \pm 0.036$ | 8.9 |
| RRR 0.7 | $0.831 \pm 0.059$ | $0.731 \pm 0.049$ | 8.7 |
| MLN-BT sam. | $0.781 \pm 0.100$ | $0.662 \pm 0.091$ | 2368 |
| RDN-B | $0.768 \pm 0.067$ | $0.668 \pm 0.060$ | 90.3 |
| RDN-B sam. | $0.744 \pm 0.093$ | $0.652 \pm 0.071$ | 103.2 |
| MLN-BC sam. | $0.742 \pm 0.085$ | $0.594 \pm 0.059$ | 64.2 |
| MLN-BT | $0.735 \pm 0.100$ | $0.601 \pm 0.079$ | 1438.3 |
| SLIPCASE | $0.650 \pm 0.065$ | $0.500 \pm 0.000$ | 0.9 |
| SLS | $0.624 \pm 0.096$ | $0.445 \pm 0.071$ | 0.148 |
| MLN-BC | $0.585 \pm 0.088$ | $0.390 \pm 0.057$ | 65.2 |

The standard deviations are also shown

ulation size, political system and the country border relationship. We used a subset of the tables and features as in Schulte and Khosravi (2012). We predicted the predicate `christianReligion(country)` and we employed a five-fold cross validation.

Table 7 shows a good performance for LEMUR (is only outperformed by ALEPH++-ExactL1 in AUC-PR and by SLIPCOVER in AUC-ROC), with the differences being statistically significant w.r.t all systems both for AUC-PR and AUC-ROC, as reported in Tables 11 and 12.

### *7.2.7 HIV*

The HIV dataset[12] (Beerenwinkel et al. 2005) records mutations in HIV's reverse transcriptase gene in patients that are treated with the drug zidovudine. It contains 364 examples, each of which specifies the presence or not of six classical zidovudine mutations, denoted by the atoms: `41L`, `67N`, `70R`, `210W`, `215FY` and `219EQ`. These atoms indicate the location where the mutation occurred (e.g., *41*) and the amino acid to which the position mutated (e.g., *L* for Leucine).

The goal is to discover causal relationships between the occurrences of mutations in the virus, so all the predicates are set as target. We employed a five-fold cross validation. In testing, we computed the probability of each atom in turn given the others as evidence.

Table 8 shows that LEMUR and SLIPCOVER get the highest results. The differences between LEMUR and all the other systems except for SLIPCOVER are statistically significant. SLS low performance is due to the fact that it returns the initial theory on this dataset.

### *7.2.8 Summarizing remarks*

On the whole, LEMUR achieves very good results that are always comparable or superior with respect to the other best systems, in terms of both AUC-PR and AUC-ROC. *T*-tests show

[12] Kindly provided by Wannes Meert.

**Table 8** Results of the experiments in terms of average AUC-PR, AUC-ROC and execution time (in seconds) on the HIV dataset

| System | AUC-PR | AUC-ROC | Time(s) |
|---|---|---|---|
| LEMUR | 0.830 ± 0.050 | 0.945 ± 0.013 | 1290.2 |
| SLIPCOVER | 0.824 ± 0.053 | 0.947 ± 0.013 | 415.9 |
| SLIPCASE | 0.777 ± 0.047 | 0.926 ± 0.012 | 43.8 |
| RRR 0.9 | 0.606 ± 0.039 | 0.505 ± 0.002 | 2.2 |
| RRR 0.7 | 0.605 ± 0.040 | 0.505 ± 0.002 | 2.7 |
| MLN-BC | 0.512 ± 0.041 | 0.305 ± 0.040 | 124.7 |
| BUSL | 0.381 ± 0.026 | 0.647 ± 0.025 | 14678.8 |
| LSM | 0.369 ± 0.023 | 0.601 ± 0.022 | 10.31 |
| RDN-B sam. | 0.311 ± 0.017 | 0.520 ± 0.034 | 57.1 |
| MLN-BC sam. | 0.302 ± 0.047 | 0.513 ± 0.039 | 66.7 |
| MLN-BT | 0.288 ± 0.037 | 0.510 ± 0.047 | 277.7 |
| RDN-B | 0.284 ± 0.057 | 0.483 ± 0.067 | 68.5 |
| MLN-BT sam. | 0.278 ± 0.027 | 0.502 ± 0.016 | 276.9 |
| SLS | 0.272 ± 0.032 | 0.501 ± 0.022 | 1.6 |

The standard deviations are also shown

**Table 9** Average log likelihood of the test set over all datasets

| System | Uw-cse | Mutag. | Carcin. | Hepat. | IMDB | Mondial | HIV |
|---|---|---|---|---|---|---|---|
| LEMUR | **−108.06** | −11.94 | −186.88 | **−88.06** | −9.76E-5 | −26.81 | −110.52 |
| SLIPCASE | −178.22 | −28.58 | **−25.91** | −119.89 | **0** | −28.72 | −120.73 |
| SLIPCOVER | −119.22 | −25.28 | −29.84 | −104.04 | **0** | **−25.01** | −107.9 |
| BUSL | −148.98 | - | - | −273.28 | −440.43 | - | −261.17 |
| LSM | −163.45 | - | - | −153.58 | −23.55 | - | −314.66 |
| ALEPH++ | −339.71 | −9.36 | −38.93 | - | −0.08 | −32.18 | - |
| RDN-B | −227.59 | −7.02 | −31.28 | −126.55 | −114.78 | −28.47 | **−40.92** |
| RDN-B sam. | −622.76 | **−6.86** | −31.45 | −139.21 | −125.44 | −28.89 | −45.80 |
| MLN-BC | −640.12 | −25.71 | −34.56 | −354.60 | −381.96 | −56.57 | −48.08 |
| MLN-BC sam. | −2944.44 | −20.34 | −30.43 | −306.41 | −1199.68 | −46.84 | −58.87 |
| MLN-BT | −196.15 | −15.15 | −32.96 | −147.42 | −124.24 | −93.33 | −305.30 |
| MLN-BT sam. | −488.43 | −17.15 | −33.02 | −150.19 | −128.977 | −81.76 | −449.82 |

In bold the highest LLs

that area differences between LEMUR and the other systems are statistically significant in its favor in all cases for the Hepatitis, IMDB, Mondial and HIV datasets, and in half of the cases for UW-CSE and Mutagenesis. These experiments show that LEMUR is able to perform well discriminative structure learning.

In order to clarify its ability to model distributions, we can consider Table 9 that shows the average log likelihood of the test set. LEMUR achieves the highest or a comparable value of LL except for Carcinogenesis and HIV. The latter dataset is particularly interesting because it is the only one where all predicates where declared as target, thus representing a generative learning problem. On HIV LEMUR has the best AUC-PR and the second best AUC-ROC

**Table 10** LEMUR performance in terms of average AUC-PR, AUC-ROC and execution time (in seconds) on the Hepatitis dataset as $S$, the depth of the exploration, varies.

| $S$ | PR | ROC | Time |
|---|---|---|---|
| 2 | 0.831 | 0.787 | 1476 |
| 3 | 0.910 | 0.897 | 2192 |
| 4 | 0.905 | 0.893 | 2444 |
| 5 | 0.915 | 0.904 | 2658 |
| 6 | 0.907 | 0.892 | 4196 |
| 7 | 0.967 | 0.965 | 3715 |
| 8 | 0.910 | 0.894 | 3504 |
| 9 | 0.890 | 0.880 | 3935 |
| 10 | 0.899 | 0.878 | 4860 |

**Table 11** $p$ values of a $t$-test when comparing the AUC-PR of LEMUR with respect to the other systems

| System | Dataset | | | | | |
|---|---|---|---|---|---|---|
| | UW-CSE | Mutagen. | Hepatitis | IMDB | Mondial | HIV |
| ALEPH++ | 1.19E-2 ● | 8.63E-1 | - | Undefined | 1.76E-2 ● | - |
| BUSL | 1.21E-2 ● | - | 8.37E-6 † | 7.88E-9 † | - | 2.09E-4 † |
| LSM | 5.46E-2 | - | 5.32E-5 † | Undefined | - | 3.1E-5 † |
| MLN-BC | 1.92E-2 ● | 4.9E-3 ◇ | 3.8E-3 ◇ | 9.52E-6 † | 7.1E-4 † | 1.21E-5 † |
| MLN-BC sam. | 2.82E-1 | 1.72E-2 ● | 3.1E-3 ◇ | 3.73E-1 | 3.7E-3 ◇ | 4.97E-6 † |
| MLN-BT | 4.93E-1 | 4.92E-1 | 1.54E-4 † | 3.33E-5 † | 6.1E-3 ◇ | 2.62E-5 † |
| MLN-BT sam. | 4.02E-1 | 1.05E-1 | 2.66E-4 † | 3.73E-1 | 1.19E-2 ● | 2.81E-6 † |
| SLIPCASE | 8E-3 ◇ | 2.63E-1 | 1.6E-3 ◇ | Undefined | 4.28E-4 † | 1.03E-2 ● |
| SLIPCOVER | 1.85E-1 | 9.78E-1 | 4.4E-3 ◇ | Undefined | 2.04E-2 ● | 1.06E-1 |
| RDN-B | 5E-1 | 6.35E-1 | 3.89E-4 † | Undefined | 2.3E-3 ◇ | 2.88E-5 † |
| RDN-B sam. | 6.91E-1 | 6.29E-1 | 3.89E-4 † | Undefined | 5.3E-3 ◇ | 2.14E-5 † |
| RRR 0.7 | 3.61E-1 | 4.15E-2 ● | 1.22E-5 † | Undefined | 4.5E-3 ◇ | 2.8E-3 ◇ |
| RRR 0.9 | 3.61E-1 | 3.1E-3 ◇ | 1.22E-5 † | Undefined | 2.2E-3 ◇ | 2.7E-3 ◇ |
| SLS | 4.9E-3 ◇ | 8.04E-7 † | 1.62E-4 † | 7.62E-9 † | 1.5E-3 ◇ | 3.05E-6 † |

The symbols ●, ◇ and † indicate, respectively, that the significance level is smaller than 0.05, 0.01 and 0.001

but its LL is lower than that of five other systems. These results show that LEMUR is more targeted to discriminative learning.

Experiments also show that, when able to complete learning, the MLN structure learning systems LSM and BUSL take the largest time, ranging from 0.27 h to 250 h for the former and from 9.6 h to 15 h for the latter.

Scatter plots of AUC-PR and AUC-ROC versus time are shown in Figs. 3 and 4 respectively. Each point corresponds to an algorithm and a dataset.

These pictures show that LEMUR achieves its results within a time that is comparable with that of the other systems, thus achieving a good time/performance trade-off: while deep exploration is costly when compared to other depth-limited search strategies, LEMUR naturally deeply explores promising regions. Indeed, as we can see in Table 10, we executed LEMUR on the Hepatitis dataset by varying the depth of exploration ($S$) from 2 to 10, having

**Table 12** *p* values of a *t*-test when comparing the AUC-ROC of LEMUR with respect to the other systems

| System | Dataset | | | | | |
|---|---|---|---|---|---|---|
| | UW-CSE | Mutagen. | Hepatitis | IMDB | Mondial | HIV |
| ALEPH++ | 3.54E-4 † | 9.6E-3 ◇ | - | Undefined | 4.6E-3 ◇ | - |
| BUSL | 1.43E-2 ● | - | 7.7E-6 † | 6.6E-7 † | - | 5.66E-5 † |
| LSM | 3.24E-1 | - | 3.64E-4 † | Undefined | - | 2.94E-5 † |
| MLN-BC | 9.65E-2 | 1.24E-4 † | 3.5E-3 ◇ | 1.81E-5 † | 1.58E-4 † | 2.92E-5 † |
| MLN-BC sam. | 7.51E-1 | 1.1E-3 ◇ | 2.8E-3 ◇ | 3.73E-1 | 1.58E-4 † | 2.22E-5 † |
| MLN-BT | 7.2E-1 | 6.4E-2 | 6.66E-5 † | 2.01E-5 † | 1.7E-3 ◇ | 3.75E-5 † |
| MLN-BT sam. | 1.44E-1 | 2.73E-2 ● | 7.34E-5 † | 3.73E-1 | 1.7E-3 ◇ | 5.95E-8 † |
| SLIPCASE | 2.68E-4 † | 2.33E-1 | 8.32E-4 † | Undefined | 0 † | 2.39E-2 ● |
| SLIPCOVER | 7.39E-1 | 3.16E-1 | 1.3E-3 ◇ | Undefined | 2.2E-2 ● | 5.81E-1 |
| RDN-B | 3.37E-1 | 5.7E-1 | 7.65E-4 † | Undefined | 5.91E-4 † | 1.7E-4 † |
| RDN-B sam. | 4.2E-4 † | 7.8E-1 | 7.65E-4 † | Undefined | 5.91E-4 † | 2.05E-5 † |
| RRR 0.7 | 4.2E-4 † | 7.8E-3 ◇ | 1.54E-5 † | Undefined | 3.7E-4 † | 2.42E-7 † |
| RRR 0.9 | 4.2E-4 † | 2.5E-3 ◇ | 1.54E-5 † | Undefined | 1.15E-4 † | 2.42E-7 † |
| SLS | 2.54E-6 † | 5.6E-10 † | 3.92E-4 † | 0 † | 9.81E-5 † | 3.46E-7 † |

The symbols ●, ◇ and † indicate, respectively, that the significance level is smaller than 0.05, 0.01 and 0.001

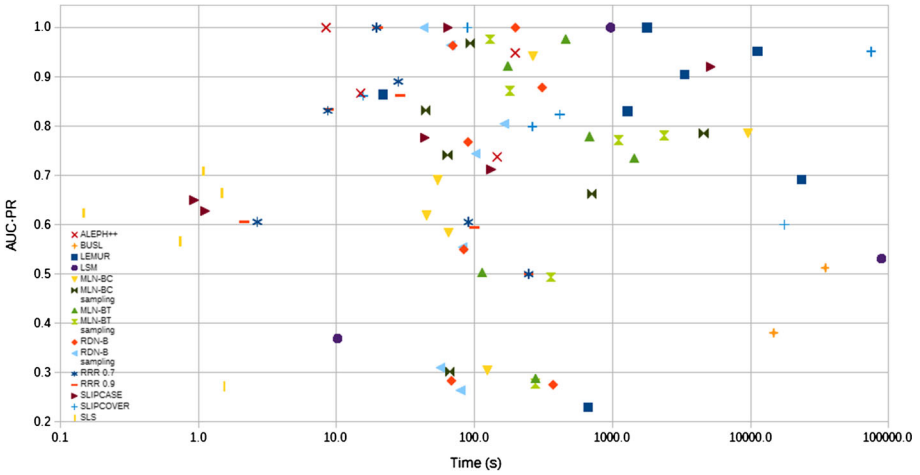


**Fig. 3** Scatter plot of AUC-PR versus Time (in logscale) in seconds

a roughly linear increase of the execution time without a significant effect on the quality of the solution.

The comparison with SLS and RRR shows that LEMUR improvement is not due to randomization only, but rather to the specific features of MCTS.

The benefits of MCTS are testified by the comparison with SLIPCASE/SLIPCOVER that use the same modeling language and differ from LEMUR in the search algorithm: both SLIPCASE and SLIPCOVER achieve smaller areas and lower LL. A possible reason for this is that LEMUR is able to perform a deeper lookahead thus bypassing possible plateaux of the heuristic function: often more than one literal must be added to a clause in order to improve

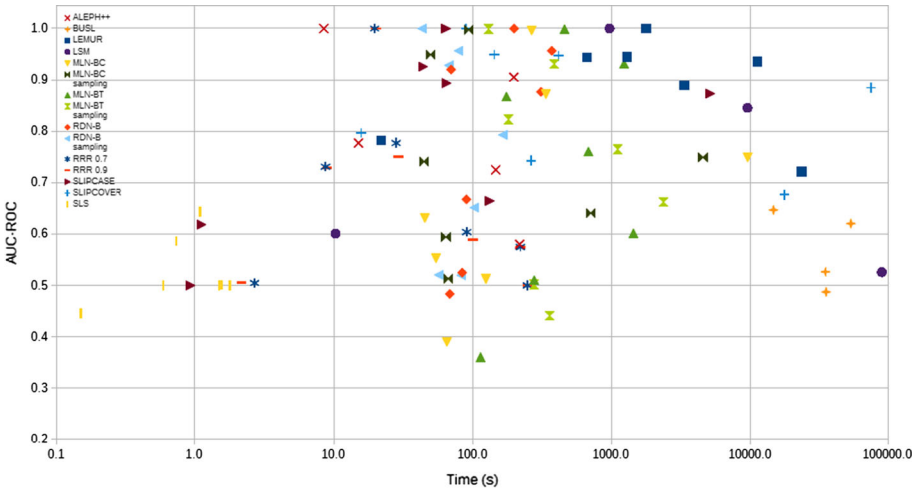


**Fig. 4** Scatter plot of AUC-ROC versus Time (in logscale) in seconds

the heuristic. Most ILP systems allow for the specification of a lookahead in the language bias by indicating which other literals can be added to a clause together with a specific literal but they are often limited to one or two extra literals, while LEMUR can consider adding several literals at once because each move is not estimated with a local inspection of the neighborhood but with a deep inspection of the tree.

Finally, it should be noted that a direct comparison of systems based on different modeling languages is difficult. We tried to make the comparison as fair as possible by providing the systems with similar or equal learning settings, even if the different encodings did not allow a perfect match. Taking into account these intrinsic differences, we have shown that LEMUR can be a competitive approach to other SRL techniques, especially for performing discriminative learning.

## 8 Conclusions

We have presented the system LEMUR that applies Monte Carlo Tree Search to the problem of learning probabilistic logic programs. LEMUR sees the problem of adding a new clause to the current theory as a tree search problem in which it solves a multi-armed bandit problem to choose the clause.

We have tested LEMUR on seven datasets and compared it with four systems for learning probabilistic and non-probabilistic logic programs, with various statistical relational systems that learn Markov Logic Networks and with stochastic search algorithms. We have compared the quality of the results in terms of AUC-PR and AUC-ROC and found that LEMUR can achieve comparable or larger areas in most cases, with the differences often statistically significant. Thus LEMUR is an appealing alternative for discriminative structure learning. As regards generative learning, LEMUR achieves good testing LL. However, on HIV, LEMUR is exceeded by five systems out of twelve in terms of LL, thus showing that it is currently better at discriminative learning. LEMUR, together with the code and the data for all the experiments, is available at http://sites.unife.it/ml/lemur.

Current challenges for structure learning include the investigation of new refinement operators such as one using a bottom clause in the style of Progol (Muggleton 1995) for

limiting the number of revisions, as in Duboc et al. (2009). Another significant challenge is to scale the system to larger datasets exploiting modern computing facilities such as clusters of computers with multiple processors per computer and multiple cores per processor. In particular, we plan to parallelize LEMUR using MapReduce (Dean and Ghemawat 2008) by dividing the search space among Map workers and by collecting the clauses' scores with Reduce workers.

**Acknowledgments** Nicola di Mauro would like to acknowledge the support of the European Commission through the project MAESTRA - Learning from Massive, Incompletely annotated, and Structured Data (Grant number ICT-2013-612944).

## References

Auer, P., Cesa-Bianchi, N., & Fischer, P. (2002). Finite-time analysis of the multiarmed bandit problem. *Machine Learning*, *47*(2–3), 235–256.

Beerenwinkel, N., Rahnenführer, J., Däumer, M., Hoffmann, D., Kaiser, R., Selbig, J., et al. (2005). Learning multiple evolutionary pathways from cross-sectional data. *Journal of Computational Biology*, *12*, 584–598.

Bellodi, E., & Riguzzi, F. (2012). Learning the structure of probabilistic logic programs. In: S. Muggleton, A. Tamaddoni-Nezhad, F. Lisi, (Eds.). *Inductive logic programming, LNCS, vol 7207* (pp. 61–75). Springer, Berlin, Heidelberg.

Bellodi, E., & Riguzzi, F. (2013). Expectation maximization over binary decision diagrams for probabilistic logic programs. *Intelligent Data Analysis*, *17*(2), 343–363.

Bellodi, E., & Riguzzi, F. (2014). Structure learning of probabilistic logic programs by searching the clause space. *Theory and Practice of Logic Programming FirstView Articles*. doi:10.1017/S1471068413000689.

Bollig, B., & Wegener, I. (1996). Improving the variable ordering of OBDDs is NP-complete. *IEEE Transactions on Computers*, *45*(9), 993–1002.

Browne, C., Powley, E. J., Whitehouse, D., Lucas, S. M., Cowling, P. I., Rohlfshagen, P., et al. (2012). A survey of Monte Carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in Games*, *4*(1), 1–43.

Bubeck, S., & Cesa-Bianchi, N. (2012). Regret analysis of stochastic and nonstochastic multi-armed bandit problems. *Foundations and Trends in Machine Learning*, *5*(1), 1–122.

Darwiche, A., & Marquis, P. (2002). A knowledge compilation map. *Journal of Artificial Intelligence Research*, *17*, 229–264.

Davis, J., & Goadrich, M. (2006). The relationship between Precision-Recall and ROC curves. In: *Proceedings of the 23rd international conference (ICML-2006) machine learning, ACM* (pp. 233–240).

De Raedt, L., Kimmig, A., & Toivonen, H. (2007). ProbLog: A probabilistic Prolog and its application in link discovery. In: *International joint conference on artificial intelligence* (pp. 2462–2467). AAAI Press.

De Raedt, L., Demoen, B., Fierens, D., Gutmann, B., Janssens, G., & Kimmig, A. (2008a). Towards digesting the alphabet-soup of statistical relational learning. In: *N*IPS*2008 workshop on probabilistic programming.

De Raedt, L., Kersting, K., Kimmig, A., Revoredo, K., & Toivonen, H. (2008b). Compressing probabilistic Prolog programs. *Machine Learning*, *70*(2–3), 151–168.

Dean, J., & Ghemawat, S. (2008). Mapreduce: Simplified data processing on large clusters. *Communication of the ACM*, *51*(1), 107–113. doi:10.1145/1327452.1327492.

Duboc, A., Paes, A., & Zaverucha, G. (2009). Using the bottom clause and mode declarations in fol theory revision from examples. *Machine Learning*, *76*(1), 73–107. doi:10.1007/s10994-009-5116-8.

Fawcett, T. (2006). An introduction to ROC analysis. *Pattern Recognition Letters*, *27*, 861–874.

Fierens, D., den Broeck, G.V., Renkens, J., Shterionov, D.S., Gutmann, B., & Thon, I. et al. (2013). Inference and learning in probabilistic logic programs using weighted boolean formulas. In: *Theory and practice of logic programming firstview articles (CoRR abs/1304.6810).*

Friedman, N. (1998). The Bayesian structural EM algorithm. In: *Proceedings of the 14th conference on uncertainty in artificial intelligence* (pp. 129–138). Morgan Kaufmann.

Gaudel, R., & Sebag, M. (2010). Feature selection as a one-player game. In: *Proceedings of the 27th international conference on machine learning* (pp. 359–366).

Gelly, S., & Silver, D. (2007). Combining online and offline knowledge in UCT. In: *Proceedings of the 24th international conference on machine learning* (pp. 273–280). ACM.

Gelly, S., & Wang, Y. (2006). Exploration exploitation in Go: UCT for Monte-Carlo Go. In: *NIPS on-line trading of exploration and exploitation workshop*.

Grumberg, O., Livne, S., & Markovitch, S. (2003). Learning to order BDD variables in verification. *Journal of Artificial Intelligence Research*, *18*, 83–116.

Gutmann, B., Thon, I., & De Raedt, L. (2011). Learning the parameters of probabilistic logic programs from interpretations. In: D. Gunopulos, T. Hofmann, D. Malerba, M. Vazirgiannis (Eds.). *European conference on machine learning and knowledge discovery in databases, LNCS, vol 6911* (pp. 581–596). Springer.

Hoos, H. H., & Stützle, T. (2004). *Stochastic local search: Foundations & applications*. Amsterdam: Elsevier.

Huynh, T.N., & Mooney, R.J. (2008). Discriminative structure and parameter learning for markov logic networks. In: W.W. Cohen , A. McCallum, S.T. Roweis (Eds.). *Proceedings of the 25th international conference on machine learning* (pp. 416–423). ACM.

Kersting, K., & De Raedt, L. (2008). Basic principles of learning Bayesian Logic Programs. In: *Probabilistic inductive logic programming, LNCS, vol 4911* (189–221). Springer.

Khosravi, H., Schulte, O., Hu, J., & Gao, T. (2012). Learning compact Markov logic Networks with decision trees. *Machine Learning*, *89*(3), 257–277.

Khot, T., Natarajan, S., Kersting, K., & Shavlik, J.W. (2011). Learning Markov Logic Networks via functional gradient boosting. In: Cook DJ, Pei J, W.W. 0010, O.R. Zaane, X. Wu (Eds.). *Proceedings of the 11th IEEE international conference on data mining* (pp. 320–329).

Kocsis, L., & Szepesvári, C. (2006). Bandit based Monte-Carlo planning. In: *Proceedings of the 17th European conference on machine learning* (pp. 282–293). Springer.

Kocsis, L., Szepesvári, C., & Willemson, J. (2006). Improved Monte-Carlo search. Tech. rep., Univ. Tartu, Estonia.

Kok, S., & Domingos, P. (2005). Learning the structure of Markov Logic Networks. In: *Proceedings of the 22nd international conference on machine learning* (pp. 441–448). ACM.

Kok, S., & Domingos, P. (2010). Learning Markov Logic Networks using structural motifs. In: J. Fürnkranz, T. Joachims (Eds.). *Proceedings of the 27th international conference on machine learning* (pp. 551–558). Omnipress.

Lowd, D., & Domingos, P. (2007). Efficient weight learning for Markov logic networks. In: *Proceedings of the 11th European conference on principles and practice of knowledge discovery in databases* (pp. 200–211). Springer.

Martí, R., Moreno, J. M., & Duarte, A. (2010). *Advanced multi-start methods*. Berlin: Springer.

May, W. (1999). Information extraction and integration: The mondial case study. Tech. rep., Universitat Freiburg, Institut für Informatik.

Meert, W., Struyf, J., & Blockeel, H. (2008). Learning ground CP-Logic theories by leveraging Bayesian network learning techniques. *Fundamenta Informaticae*, *89*(1), 131–160.

Mihalkova, L., & Mooney, R.J. (2007). Bottom-up learning of markov logic network structure. In: *Proceedings of the 24th international conference on machine learning* (pp. 625–632). ACM.

Muggleton, S. (1995). Inverse entailment and Progol. *New Generation Computing*, *13*, 245–286.

Natarajan, S., Khot, T., Kersting, K., Gutmann, B., & Shavlik, J. (2012). Gradient-based boosting for statistical relational learning: The relational dependency network case. *Machine Learning*, *86*(1), 25–56.

Nienhuys-Cheng, S. H., & de Wolf, R. (1997). *Foundations of inductive logic programming, LNCS*. Berlin: Springer.

Paes, A., Revoredo, K., Zaverucha, G., & Costa, V.S. (2005). Probabilistic first-order theory revision from examples. In: *Proceedings of the 15th international conference on inductive logic programming, LNCS, vol 3625* (pp. 295–311). Springer. doi:10.1007/11536314_18.

Paes, A., Zaverucha, G., & Costa, V.S. (2008). Revising first-order logic theories from examples through stochastic local search. In: *Proceedings of the 17th international conference on inductive logic programming, ILP'07* (pp. 200–210). Springer, Berlin, Heidelberg. http://dl.acm.org/citation.cfm?id=1793494.1793518

Poole, D. (2008). The independent choice logic and beyond. In L. De Raedt, P. Frasconi, K. Kersting, & S. Muggleton (Eds.), *Probabilistic inductive logic programming, LNCS* (Vol. 4911, pp. 222–243). Berlin, Heidelberg: Springer.

Rauzy, A., Châtelet, E., Dutuit, Y., & Bérenguer, C. (2003). A practical comparison of methods to assess sum-of-products. *Reliability Engineering and System Safety*, *79*(1), 33–42.

Richards, B. L., & Mooney, R. J. (1995). Automated refinement of first-order horn-clause domain theories. *Machine Learning*, *19*(2), 95–131. doi:10.1007/BF01007461.

Riguzzi, F. (2004). Learning logic programs with annotated disjunctions. In: A. Srinivasan, R. King (Eds.). *Proceedings of the 14th international conference on inductive logic programming, LNCS, vol. 3194* (pp. 270–287). Springer. doi:10.1007/978-3-540-30109-7_21.

Riguzzi, F. (2007a). ALLPAD: Approximate learning of logic programs with annotated disjunctions. In: S. Muggleton, R. Otero (Eds.). *Proceedings of the 16th international conference on inductive logic programming, LNAI, vol. 4455* (pp. 43–45). Springer. doi:10.1007/978-3-540-73847-3_11.

Riguzzi, F. (2007b). A top down interpreter for LPAD and CPlogic. In: *Proceedings of the 10th congress of the Italian association for artificial intelligence, LNAI, vol. 4733* (pp. 109–120). Springer. doi:10.1007/978-3-540-74782-6_11.

Riguzzi, F. (2008). ALLPAD: Approximate learning of logic programs with annotated disjunctions. *Machine Learning*, *70*(2–3), 207–223. doi:10.1007/s10994-007-5032-8.

Riguzzi, F. (2009). Extended semantics and inference for the Independent Choice Logic. *Logic Journal of the IGPL*, *17*(6), 589–629. doi:10.1093/jigpal/jzp025.

Riguzzi, F. (2014). Speeding up inference for probabilistic logic programs. *The Computer Journal*, *57*(3), 347–363. doi:10.1093/comjnl/bxt096.

Riguzzi, F., & Di Mauro, N. (2012). Applying the information bottleneck to statistical relational learning. *Machine Learning*, *86*(1), 89–114. doi:10.1007/s10994-011-5247-6.

Riguzzi, F., & Swift, T. (2010). Tabling and Answer Subsumption for Reasoning on Logic Programs with Annotated Disjunctions. In: *Technical communications of the 26th int'l. conference on logic programming (ICLP'10), Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, Leibniz international proceedings in informatics (LIPIcs), vol. 7* (pp. 162–171). doi:10.4230/LIPIcs.ICLP.2010.162.

Riguzzi, F., & Swift, T. (2011). The PITA system: Tabling and answer subsumption for reasoning under uncertainty. In: *Theory and practice of logic programming, 27th international conference on logic programming (ICLP'11)* special issue, Lexington, Kentucky 6–10 July 2011 *11*(4–5), 433–449. doi:10.1017/S147106841100010X.

Riguzzi, F., & Swift, T. (2013). Welldefinedness and efficient inference for probabilistic logic programming under the distribution semantics. In: *Theory and practice of logic programming 13 (Special Issue 02–25th Annual GULP Conference)* (pp. 279–302). doi:10.1017/S1471068411000664.

Robbins, H. (1952). Some aspects of the sequential design of experiments. *Bulletin of the American Mathematics Society*, *58*, 527–535.

Rolet, P., Sebag, M., & Teytaud, O. (2009). Boosting active learning to optimality: A tractable Monte-Carlo, billiard-based algorithm. In W. Buntine, M. Grobelnik, D. Mladeni, & J. Shawe-Taylor (Eds.), *Proceedingf of the European conference on machine learning and knowledge discovery in databases, LNCS* (Vol. 5782, pp. 302–317). Berlin, Heidelberg: Springer.

Rujan, P. (1997). Playing billiards in version space. *Neural Computation*, *9*(1), 99–122.

Sang, T., Beame, P., & Kautz, H. A. (2005). Performing bayesian inference by weighted model counting. In M. M. Veloso & S. Kambhampati (Eds.), *National conference on artificial intelligence* (pp. 475–482). Cambridge: AAAI Press / The MIT Press.

Santos Costa, V., Rocha, R., & Damas, L. (2012). The YAP Prolog system. *Theory and Practice of Logic Programming*, *12*(1–2), 5–34.

Sato, T. (1995). A statistical learning method for logic programs with distribution semantics. In: *International conference on logic programming* (pp. 715–729). MIT Press.

Sato, T. (2008). A glimpse of symbolic-statistical modeling by PRISM. *Journal of Intelligent Information Systems*, *31*(2), 161–176.

Sato, T., & Kameya, Y. (2001). Parameter learning of logic programs for symbolic-statistical modeling. *Journal of Artificial Intelligence Research*, *15*, 391–454.

Schulte, O., & Khosravi, H. (2012). Learning graphical models for relational data via lattice search. *Machine Learning*, *88*(3), 331–368.

Schwarz, G. (1978). Estimating the dimension of a model. *The Annals of Statistics*, *6*(2), 461–464.

Srinivasan, A. (2012). Aleph. http://www.cs.ox.ac.uk/activities/machlearn/Aleph/aleph.html.

Srinivasan, A., Muggleton, S., Sternberg, M. J. E., & King, R. D. (1996). Theories for mutagenicity: A study in first-order and feature-based induction. *AI*, *85*(1–2), 277–299.

Srinivasan, A., King, R. D., Muggleton, S., & Sternberg, M. J. E. (1997). Carcinogenesis predictions using ILP. In N. Lavrac & S. Dzeroski (Eds.), *7th International workshop on inductive logic programming, LNCS* (pp. 273–287). Springer.

Vennekens, J., Verbaeten, S., & Bruynooghe, M. (2004). Logic programs with annotated disjunctions. *International conference on logic programming, LNCS, vol. 3131* (pp. 195–209). Springer.

Železný, F., Srinivasan, A., & Page, C.D. (2002). Lattice-search runtime distributions may be heavy-tailed. In: *Proceedings of the 12th international conference on inductive logic programming*, Springer.

Zelezný, F., Srinivasan, A., & Page, C. D, Jr. (2006). Randomised restarted search in ILP. *Machine Learning*, *64*(1–3), 183–208.