MCMC Estimation of Conditional Probabilities in Probabilistic Programming Languages

Bogdan Moldovan, Ingo Thon, Jesse Davis, and Luc de Raedt

Department of Computer Science, Katholieke Universiteit Leuven, Belgium

Abstract. Probabilistic logic programming languages are powerful formalisms that can model complex problems where it is necessary to represent both structure and uncertainty. Using exact inference methods to compute conditional probabilities in these languages is often intractable so approximate inference techniques are necessary. This paper proposes a Markov Chain Monte Carlo algorithm for estimating conditional probabilities based on sampling from an AND/OR tree for ProbLog, a general-purpose probabilistic logic programming language. We propose a parameterizable proposal distribution that generates the next sample in the Markov chain by probabilistically traversing the AND/OR tree from its root, which holds the evidence, to the leaves. An empirical evaluation on several different applications illustrates the advantages of our algorithm.

1 Introduction

Probabilistic programing languages (PPLs) embed probabilistic concepts into programming languages. They provide high-level constructs for specifying models that can capture both uncertainty and structure. Examples of PPLs include ProbLog [11, 2], PRISM [22], BLOG [14], Church [6], and IBAL [16].

This paper focuses on ProbLog, a probabilistic extension of the logic programming language Prolog, based on Sato's distribution semantics [20]. A ProbLog program represents a distribution over possible worlds. Consequently, unlike in Prolog, the success or failure of a query is not deterministic. A central inference problem is computing the probability that a query succeeds conditioned on some given evidence. Unfortunately, computing such probabilities exactly for high dimensional realistic problems is unfeasible, only approximation techniques providing a polynomial time solution [1]. One of the most popular sampling techniques used by many PPLs [21, 6, 14] is Markov chain Monte Carlo (MCMC) [1].

We present an MCMC approach tailored to computing the conditional probability of a ProbLog query. Computing conditional probabilities in PPLs has, with a few exceptions [3], not yet received much attention in the literature.

Several challenges arise when designing a ProbLog MCMC algorithm. First, as ProbLog is a programming language, the possible worlds can be infinite, making it impossible to sample complete worlds. Our MCMC approach samples partial possible worlds (i.e., assignments to subsets of the random variables in model) which correspond to proofs. Second, ProbLog explicitly deals with the disjoint sum problem in contrast to other PPLs (e.g., Prism) that make the mutually exclusiveness assumption to avoid this NP-hard problem. The disjoint sum problem arises when two proofs overlap. We solve this using the Karp and Luby algorithm [9]. By not making the mutually exclusiveness assumption, a user can write a ProbLog program that more easily models a richer problem setting. Third, only those possible worlds that agree with the evidence are relevant for approximating the conditional probability. We employ an AND/OR tree rooted at the evidence, representing all such possible worlds, and probabilistically traverse the tree to generate only those samples where the evidence holds. The AND/OR tree is needed to deal with ProbLog's underlying non-deterministic nature, also distinguishing our approach from those applied to functional programming languages. Finally, in contrast to some other languages, we also provide support for numeric random variables and discrete distributions.

2 Background

We first review some basic concepts of logic programming: An atom $pred(t_1, ..., t_n)$ consists of a predicate $pred/n$ of arity n and t_i terms. A term is either a (lowercase) *constant*, a (uppercase) *variable*, or a functor $func/n$ applied on n terms. A *definite clause* is an expression of the form $h \leftarrow b_1, ..., b_n$, where h and the b_i are atoms. It states that h is true whenever all b_i are true. If n is 0, we have a fact $f \leftarrow$, which expresses that f is true. A *substitution* $\theta = \{X_1 = t_1, ..., X_n = t_n\}$ maps each variable X_i to a term t_i. Applying a substitution θ to an atom a yields $a\theta$, in which each occurrence of X_i in a is replaced with t_i.

A ProbLog [11, 2] program consists of a set of labeled facts $p_i :: c_i$, where p_i is a probability value and c_i a fact, and a set of definite clauses. Each ground instance of such a fact represents a random variable that is true with probability p_i. We use the following ProbLog program as a running example in the paper:

```
0.05 :: burglary.
0.01 :: earthquake.
0.7 :: hears_alarm(john).
0.6 :: hears_alarm(mary).
```

```
alarm :- burglary.
alarm :- earthquake.
calls(Pers) :- alarm, hears_alarm(Pers).
```

It has the random variables: *burglary*, *earthquake*, $hears\_alarm(john)$ and $hears\_alarm(mary)$, and states that there is an alarm whenever there is burglary or an earthquake. The last clause states that if there is an alarm and a person hears the alarm, that person will call.

To model univariate discrete distributions (e.g., uniform, Poisson), we also allow for discrete distribution probabilistic facts $X \sim \phi :: f$. X is a logical variable appearing in atom f and ϕ a probability density function. Currently only the uniform and Poisson distributions are implemented. For example, $X \sim uniform(7) :: apples(X)$ specifies that $apples(X)$ is true with X sampled from the set of integers between 1 and 7 with equal probability. Only for the sampled value of X will $apples(X)$ be true. Each grounding of all the variables (except X) in f denotes a random variable. In ProbLog, all random variables (discrete distributions or probabilistic facts) are assumed marginally independent.

The semantics of the ProbLog program is then given by probability distributions over subsets of the facts f_i (called subprograms) and sample values for the numeric variables in the uniform and Poisson distributions. Each ground probabilistic fact $p :: f$ specifies an atomic choice, i.e., we can choose to include f as a fact (with probability p) or its negation $\overline{f}$ (with probability $1-p$), where $\overline{f}$ is the predicate denoting the explicit negation of f. These negated predicates may also occur in the background knowledge, allowing us to deal with explicit negation on probabilistic facts. For a uniform distribution, X will be sampled from the discrete uniform distribution and $f(x)$ will be included as a fact, where x is the sampled value for X. Poisson distributions are treated similarly.

The resulting set of facts is called a *total choice* [17] when we have included a fact for *all* random variables, and a *partial choice* otherwise. To each total or partial choice we can associate a probability. This is simply the product of the probabilities of the atoms chosen for inclusion in the total or partial choice, as these random variables are marginally independent. For example, the probability of the total choice $T_1 = \{burglary, \overline{earthquake}, hears\_alarm(john), \overline{hears\_alarm(mary)}\}$ is $0.05 \times .99 \times .7 \times .4$.

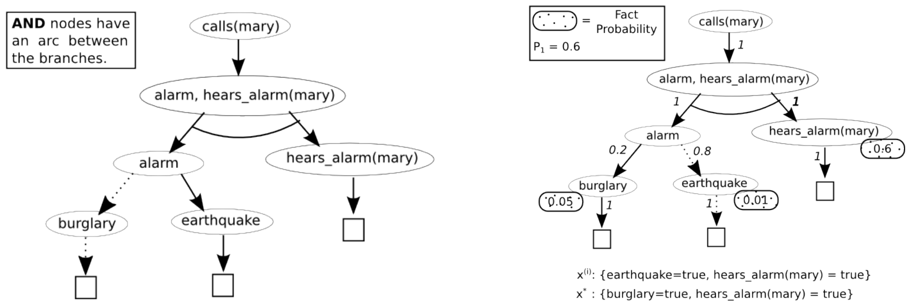
The distribution over total choices induces a probability distribution P over possible worlds, which also defines the (success) probability $P_s(q)$ of a query q (conjunction of atoms) as $P_s(q) = P(\{w|q$ is true in the possible world $w\})$. Continuing our example, the probability of *alarm* is equal to the probability that it is true in the 2^4 possible worlds. Rather than enumerating these worlds explicitly, one would compute the proofs of the query and observe that *alarm* is true exactly when *earthquake* or *burglary* is true. The partial choices corresponding to the two proofs are sometimes called *explanations.* So:

$$P_s(alarm) = P_s(burglary \vee earthquake)$$
$$= P_s(burglary \vee (\overline{burglary} \wedge earthquake)) = 0.05 + (.95 \times .01)$$

This derivation also illustrates the *disjoint sum* problem, as we have to make the two arguments of the disjunction *mutually exclusive* before we can correctly compute the probability of the query. This is a #P complete problem [23].

3 AND/OR Trees

Our MCMC algorithm relies on the notion of an AND/OR tree for definite programs [18]. Let T be a definite clause program and $?-e$ an evidence query. The AND/OR tree for ProbLog $pTree(e)$ of the given query is a tree with root e whose nodes are divided into two disjunctive sets, the set of *AND* nodes and the set of *atomic choice* nodes. Each node contains a query. Leafs of $pTree(e)$ are either an empty clause ($\Box$) or a failure (for leaf $?-a_{leaf}$ no clause head in T unifies with a_{leaf}). The nodes $?-a_1, ..., ?-a_n$ constitute the children of an AND node $?-a_1, ..., a_n$. An atomic choice node can be of three types: exclusive OR-nodes, probabilistic atoms and discrete distribution atoms. An OR node $?-a$ has a child $?-(a_1, ..., a_n)\theta$ if and only if there is a definite clause $a \leftarrow a_1, ..., a_n$ in T and a substitution θ such that $a'\theta = a$. An atomic choice node $?-a$ (or $?-\overline{a}$) for a probabilistic atom $p :: a'$ adds $a'\theta$ (or $\overline{a'}\theta$) with $a\theta = a'\theta$ as a fact to T, and imposes the constraint that $\overline{a'}\theta$ (or $a'\theta$) will never be added to T. Similarly, an atomic choice node $?-a$ for a discrete distribution atom $X \sim \phi :: f$ adds $a\theta\{X = v\}$ as a fact to T for one possible value v in the distribution ϕ with $a\theta\{X = v\} = f\theta\{X = v\}$, and imposes the contraint that facts will not be added for any other value than v. Since these facts are now added to T, they prove the node containing these probabilistic atomic choices, thus the child of this node is $\square$. Figure 1a illustrates $pTree(e)$ on our running example for $e = calls(mary)$. An AND/OR tree is obtained by starting with the root e and recursively expanding each node for the definite clause program.



(a) A solution tree (solid line arrows) (b) Highlighted solution tree for $x^{(i)}$.

Fig. 1: Example AND/OR tree for the evidence: $calls(mary)$.

A *solution tree* S in the AND/OR tree $pTree(e)$ is a subtree such that 1) e is the root of S, 2) the children of all *AND* nodes that are in S are also in S, 3) all *OR* nodes that are in S have exactly one child that is also in S, and 4) all the leaves are $\square$. A solution tree is consistent with regard to random variables and atomic choices (e.g., it will not contain two atoms a and $\overline{a}$ and there cannot be two different values assigned to the same discrete distribution atom). A solution tree represents one particular proof of e. Since e is true with respect to every solution tree in $pTree(e)$, every solution tree implies a model of the evidence e. Figure 1a highlights with solid line arrows the solution tree corresponding to the partial possible world $\{earthquake = true, hears\_alarm(mary) = true\}$.

4 MCMC Algorithm Overview

Computing the conditional probability of a query in ProbLog is defined as:

Given: a ProbLog program T, a set of observed (evidence) atoms e, a query q
Do: Calculate $P(q|e)$

As it is often intractable to compute $P(q|e)$ exactly, we propose an MCMC [1, 8, 13] approach. Each state in the Markov Chain is a (partial) possible world. The estimate of $P(q|e)$ is obtained by dividing the number of partial possible worlds where e is true and q is entailed by the number of partial possible worlds where e is true. Two key challenges arise when designing the MCMC algorithm.

The first challenge is designing a proposal distribution that, as often as possible, constructs states that agree with e, as only these are relevant for estimating

$P(q|e)$. We exploit the fact that each partial possible world meeting this criteria corresponds to a solution tree in $pTree(e)$. Given the previous solution tree, our proposal distribution builds a new one to propose as the candidate next state.

Secondly, two partial possible worlds can overlap, i.e., sampling their unassigned variables can lead to the same full possible world. If two such overlapping partial worlds are counted as distinct then that full possible world would be overcounted, skewing the probability estimate. We adapt ideas from the Karp and Luby algorithm [9] to identify overlapping worlds.

Our algorithm is similar to the standard MCMC algorithm in [1]. Until a stop criteria is met, each iteration proposes a candidate state, which is checked for overlap with previously seen states. If there is no overlap, or it can be resolved, we calculate the acceptance probability, and advance to the next state accordingly.

4.1 Proposing a New State

Our Markov chain samples solution trees from $pTree(e)$. We exploit the intuition that small changes in the solution tree are more likely to lead to another solution than a big jump by probabilistically favouring reusing parts of the current proof for e. Each proof requires making decisions at OR and atomic choice nodes. We stay close to the previous state by (1) following the same branch at an OR node with probability P_1, and (2) making the same atomic choice with probability P_2.

P_1 and P_2 are user defined parameters; higher values encourage more reuse between consecutive solution trees. Parameter choice depends on the problem. If $pTree(e)$ contains many solution trees with few shared branches, lower parameter values are better to encourage faster solution space exploration. If $pTree(e)$ contains only few solution trees or they share many branches, higher values are better to favour reuse. If solution trees are more evenly spead out in $pTree(e)$, parameters have smaller impact. Any non-zero values lead to eventual exploration of all solution trees in $pTree(e)$. Only these are relevant to estimate $P(q|e)$.

Algorithm 1 outlines the recursive procedure *prove* for proposing a new solution. Its parameters are: N (current node), S_{old} (previous solution tree), and S_{new} (tree under construction). It begins at the root node e and, depending on the type of the current node N, it recursively traverses $pTree(e)$ as follows:

AND node: Recursively call *prove* on each of the node's children because proving e requires proving each child. Return the conjunction of the results.

OR node: (At least) one of the children $c_1, \ldots, c_n$ of N needs to be proved in order for e to be true. To favour reuse, if N occurs in S_{old}, then pick the same child c as in S_{old} with probability P_1 and return *prove*(c). Otherwise, pick c_i uniformly at random between $c_1, \ldots, c_n$ and return *prove*(c_i).

Atomic choice node: To favour reuse, if N occurs in S_{old}, then with probability P_2 pick the same value for the random variable as in S_{old}. Otherwise pick a value for it from its probability distribution. For probabilistic atoms, only one value (either true or false) makes e true, so we are forced to pick this value for the proof to succeed. Add the atom in N to T and return true.

Empty clause: Return true.

Failure: No clause head in T unifies with the atom in the node. Return false.

Function *prove* returns true if it finds a solution tree, and false otherwise. If *prove* returns true, the partial possible world associated with S_{new} is the candidate next state. Otherwise, the candidate next state is identical to the current state: $x^* = x^{(i)}$, where we know e is entailed. Subsection 4.3 shows why this is advantageous.

Algorithm 1: bool $prove(N, S_{old}, var\ S_{new})$

```
   Require: Global vars: $pTree(e)$
 1 Add node $N$ to $S_{new}$
 2 if $N$ is AND node $?-a_1,...,a_n$ then return $\bigwedge_i prove(a_i, S_{old}, S_{new})$
 3 else if $N$ is OR node with $n$ children then
 4     if $N$ is in $S_{old}$ then
 5         $c_{old}$ = child of $N$ in $S_{old}$
 6         with prob. $P_1$: return $prove(c_{old}, S_{old}, S_{new})$
 7     let $c_1,...,c_n$ be the children of $N$ in $pTree(e)$
 8     pick $i$ uniformly from $[1...n]$
 9     return $prove(c_i, S_{old}, S_{new})$
10 else if $N$ is atomic choice then
11     if $N$ is in $S_{old}$ then
12         with prob. $P_2$: pick same value for random variable as in $S_{old}$
13     else pick value randomly from its prob distribution
14     add atom $N$ to ProbLog program
15     return true
16 else if $\square$ then return true
17 else return false
```

4.2 Handling Overlapping Partial Worlds

S_{new} represents one proof or explanation for e. Two different explanations for e are not necessarily mutually exclusive (i.e., they overlap). This occurs if, in both explanations, there exists a setting to the unassigned variables that produces the same *full* possible world. This is known as the disjoint sums of product problem.

We illustrate this problem on our example with e={*alarm*=*true*}. There are two solution trees corresponding to the partial possible worlds {*burglary*=*true*} and {*earthquake*=*true*}. Each partial possible world represents a *set* of full possible worlds. The partial world {*burglary*=*true*} represents the two full worlds: {*burglary*=*true*, *earthquake*=*false*} and {*burglary*=*true*, *earthquake*=*true*}. Similarly, the partial world {*earthquake*=*true*} represents the two full worlds: {*burglary*=*false*, *earthquake*=*true*} and {*burglary*=*true*, *earthquake*=*true*}. The full world {*burglary*=*true*, *earthquake*=*true*} is represented by both these two partial worlds. Two partial worlds overlap if they both can represent the same full world. Treating them as distinct (i.e., non-overlapping) will cause this full world to be counted twice, leading to an incorrect probability estimate.

To solve the disjoint sums problem we use the idea from the Karp and Luby algorithm [9]. Each possible world is assigned to exactly one of its explanations. This assignment is defined as positive, leading to the world being accepted. We then use sampling of the unassigned variables in a partial world to resolve overlap. When a candidate sample is proposed, we assign it to its explanation represented by S_{new}, obtaining a pair of a possible world and an explanation. For each possible world, only one such pair is positive. As samples are obtained, we build a list of unique positive pairs, and check new candidate samples against this. If the sample overlaps with a previous one from the list assigned to a different explanation, we attempt to remove overlap as follows. We pick a variable from the previous possible world which is unassigned in the proposed world. Then we extend the proposed world by setting this variable to a value drawn from its distribution. We repeat this procedure until (1) we arrive at a world with no overlap which we save in the list and propose as the candidate state, or (2) no variable in the previous world is unassigned in our proposed world and there is still overlap. In the second case, we reject the sample and propose the current state instead. Intuitively, we reject sample contributions from the overlapping world. It was shown [9] that this results in an accurate estimate for $P(e)$.

Assume $\{earthquake{=}true\}$ is the first sampled possible world, assigned to the same explanation. If $\{burglary{=}true\}$ is the next sample, we identify an overlap and draw a value for *earthquake*. If *earthquake*=*true*, the full world overlaps with the first sample and we reject it. If $earthquake{=}false$, the overlap is eliminated. We then propose $\{burglary{=}true,\ earthquake{=}false\}$ and assign the world to the explanation $\{burglary{=}true\}$.

4.3 Computing the Acceptance Probability

The Markov chain advances by accepting a candidate state x^* with probability $A = min\{1, \frac{P(x^*)Q(x^{(i)}|x^*)}{P(x^{(i)})Q(x^*|x^{(i)})}\}$, and otherwise remains in the same state ($x^{(i+1)} = x^{(i)}$) [1]. $P(\cdot)$ is the probability of a state (i.e., partial possible world), and $Q(\cdot|\cdot)$ is the probability of transitioning from one state to another. We illustrate these calculations using the example in Figure 1b, where each choice branch is labeled with its probability of being selected in x^* given $x^{(i)}$.

Computing $P(\cdot)$: The probability of a partial world $w = \{c_1 = v_1, \ldots, c_n = v_n\}$ is: $P_{world}(w) = \prod_{i=1}^{n} P_{s_i}$, where P_{s_i} is probability that fact c_i takes on value v_i. Thus $P(\{burglary = true, hears\_alarm(mary) = true\}) = 0.05 \times 0.6 = 0.03$.

Computing $Q(\cdot|\cdot)$: $Q(x^*|x^{(i)})$ is the product of the probabilities of all the choices made when constructing S_{new} from S_{old} since the choices at each node type are made independently. Algorithm 2 shows *computeQ*, a recursive algorithm similar in structure to Algorithm 1, with S_{old} (previous solution tree) and S_{new} (proposed solution tree) as parameters. To compute $Q(x^{(i)}|x^*)$ we call *computeQ* and swap the order of the parameters. In our example in Figure 1b, at the top OR node the probability of the choice is 1 (node has only one child). Next, at the AND node, we multiply the probabilities obtained by recursively calling *computeQ* on each child. The atomic choice *hears_alarm(mary)* must be true for the proof to succeed, so there is no choice and we return 1. At the OR node *alarm*, given parameter $P_1 = 0.6$, the probability of picking the child *burglary* is $\frac{1-0.6}{2} = 0.2$. We reach the atomic choice *burglary*, and we return 1. The product of all the choices made is: $Q(x^*|x^{(i)}) = 1 \times ((1) \times (0.2 \times 1)) = 0.2$.

Algorithm 2: float $computeQ(N, S_{old}, S_{new})$

```
1  if $N$ is AND node $?-a_1,...,a_n$ then return $\prod_i computeQ(a_i, S_{old}, S_{new})$
2  else if $N$ is OR node with $n$ children then
3      $c_{new}$ = child of $N$ in $S_{new}$
4      if $N$ and $c_{new}$ are in $S_{old}$ then
5          return $(P_1 + \frac{1-P_1}{n}) * computeQ(c_{new}, S_{old}, S_{new})$
6      else if $N$ is in $S_{old}$ but $c_{new}$ is not in $S_{old}$ then
7          return $(\frac{1-P_1}{n}) * computeQ(c_{new}, S_{old}, S_{new})$
8      else return $(\frac{1}{n}) * computeQ(c_{new}, S_{old}, S_{new})$;        // none in $S_{old}$
9  else if $N$ is atomic choice then
10     Let $P_{new}$ be the prob of the value of random variable in $S_{new}$
11     if $N$ is in $S_{old}$ with same sampled value then
12         return $P_2 + (1 - P_2) * P_{new}$
13     else if $N$ is in $S_{old}$ with different sampled value then
14         return $(1 - P_2) * P_{new}$
15     else return $P_{new}$;                                         // $N$ is not in $S_{old}$
16 else return 1;                                                     // if $\square$
```

Computing A: In our running example, the acceptance probability will be: $A = min\{1, \frac{0.03 \times 0.2}{0.006 \times 0.2}\} = 1$ and the proposed sample will be accepted.

If a traversal does not reach a solution, or if overlap cannot be resolved, the proposed state is the current state. This greatly simplifies the algorithm. In this case, computing $Q(x^*|x^{(i)})$ would have needed to sum over the probabilities of all paths in $pTree(e)$ where e is not entailed. However, the ratio $\frac{Q(x^{(i)}|x^*)}{Q(x^*|x^{(i)})} = 1$ (since $x^* = x^{(i)}$). Thus $A = 1$ and the MCMC chain advances with $x^{(i+1)} = x^{(i)}$.

5 Related Work

The use of MCMC techniques is popular in the literature on PPLs and statistical relational learning. Many languages (e.g., Blog [14], Church [6], Alchemy [12], Prism [22]) offer an inference algorithm based on MCMC. Our MCMC approach has the important difference that it needs to deal with the disjoint sum problem. The above mentioned techniques assume that the probability of a function or predicate call can be approximated by counting/weighting the number of succesful execution traces of the program. Doing this in the ProbLog context will lead to overcounting of partial worlds and possibly incorrect probability values larger than one. In Blog or Church this is a valid assumption as the underlying programming language is functional (i.e., deterministic). In Prism, one assumes mutually exclusive explanations so the problem does not arise. We solve this using the Karp and Luby algorithm [9], previously used in the ProbLog context in DNF sampling [10], but not with an MCMC approach. Aditionally, by proposing states probabilistically, we eventually fully explore the state space and can tackle a bigger set of problems, while Alchemy and MLNs [12, 19] combine MCMC with satisfiability testing to have an MC-SAT algorithm that can also tackle problem domains with deterministic or near-deterministic dependencies [19].

Wingate [24] proposed a general MCMC technique for obtaining a probability distribution over program execution traces, together with a general method of transforming arbitrary programming languages into PPLs. To compute a conditional probability, one would need to do rejection sampling on all these execution traces sampled from the unconditioned program. By comparison, our AND/OR tree based approach for estimating conditional probabilities attempts to guide the Markov chain towards the solution space of the conditioning query.

We want to stress that the use of AND/OR trees here is completely different than in the work by Dechter & probabilistic theorem proving (PTP) [4, 5], in that, we employ the traditional trees used in theorem proving, whereas Dechter & PTP employ the special data structures used in a knowledge compilation setting. These data structures impose different requirements than the AND/OR trees and are aimed at optimizing some operations (e.g., weighted model counting).

6 Empirical Evaluation

The goal of the experimental evaluation is to explore how our MCMC approach:
Goal 1: compares to existing ProbLog inference techniques
Goal 2: compares to other PPLs when faced with hard constraints
Goal 3: copes with Poisson and uniform distributions

Implementation was in Yap-6 Prolog. Experiments were run on computers with Intel Core $i7-2600$ $3.4GHz$ processors, $8MB$ cache, and $16GB$ memory. Parameters were set to $P_1 = 0.6$, $P_2 = 0.4$, but varying them had minimal impact on performance on the three considered problem domains.

Goal 1: Comparison to the Following Existing ProbLog Inference Algorithms:

ProbLog Exact is the current exact inference implementation for ProbLog, which can scale to tens of thousands of proofs [11].

ProbLog MC is a naive Monte-Carlo method that samples possible worlds for a ProbLog program[11]. We reject the ones where the evidence does not hold.

ProbLog MC-SAT is the state-of-the-art approach to approximate inference in ProbLog [3]. It converts a ProbLog program to a CNF theory and then runs the MC-SAT inference algorithm [15].

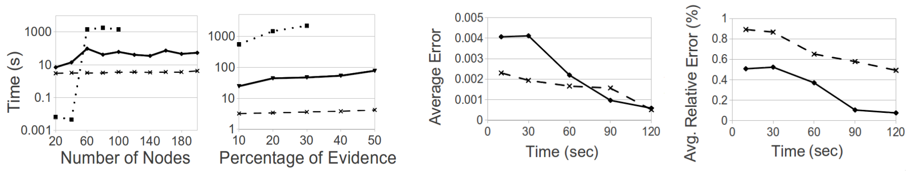
We use WebKB[1], a large data set about university webpages. The knowledge base (KB) contains deterministic knoweldge about the set of words present on the pages and links between pages. We only consider the overall 20 most commonly occuring words in all documents, not including stem words (e.g., the, of, a). The query is a ground wordclass/3 atom. For each setting we randomly generate 20 KBs and average results. Timeout is 1 hour, and this value is used in case of a timeout when computing average runtime. The lines in the graphs stop when more than half the runs time out. We run MCMC and MC-SAT for $100,000$ samples. ProbLog MC is setup with a 95% confidence interval width of 0.01.

We first compare how run time varies with the number of pages in the domain. We vary the number of pages from 20 to 200. For each page, with 20% probability,

[1] http://www.cs.cmu.edu/∼webkb/

we include its true class (e.g. course, staff, etc.) in the evidence. Figure 2a(left) shows the results, where ProbLog MC is not included as it cannot solve any task. ProbLog Exact cannot solve domains with more than 100 pages. MCMC and MC-SAT can solve any WebKB graph size, MC-SAT being faster.

In a second task, we compare how run time varies with the amount of evidence. We keep the number of pages constant at 100, but vary the probability that page's class is included in e from 10% to 50%. ProbLog MC cannot solve this task either, ProbLog exact can only solve settings with a smaller amount of evidence, while MCMC and MC-SAT can consistently solve any setting, as shown in Figure 2a(right). MC-SAT is also faster on this setting.



(a) Runtime comparisons (b) Error comparisons

Fig. 2: WebKB: MCMC is continuous line, Exact dotted, MC-SAT dashed.

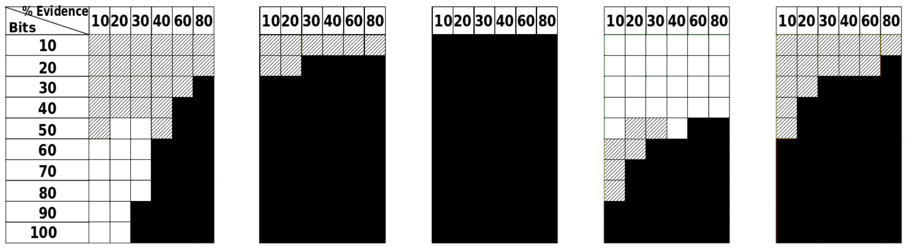
Since MC-SAT is faster than MCMC in producing the same number of samples, we investigate their accuracy next. We use the first task for the largest subset of pages (100) where we can obtain Exact probabilities. We run both methods for the same amount of time (i.e., MC-SAT produces more samples) and compare the errors in the predicted probabilities, as shown in Figure 2b. After the burn-in period, MCMC average error is about the same as MC-SAT, but its average relative error with respect to the exact probability is smaller.

Goal 2: Comparison with other inference engines: the **Bher** implementation of Church [6] and the **Alchemy** [12] implementation of Markov logic [19].

As a test domain, we use Hamming codes, a family of linear error-correcting codes [7] containing data and parity bits. Instead of considering error correction or detection, we predict the values of certain bits given other bits as evidence. This is intended as an illustration of how algorithms cope with hard constraints (PPLs are not necessarily the best way to infer missing bit values). Hard constraints are important in PPLs, yet many approaches struggle.

We vary the number of bits in randomly produced Hamming codes from 10 to 100, and the percentage of bits included in the evidence from 10% to 80%. The query is one of the data bits. We run all sampling algorithms for $100,000$ samples. Figure 3 shows a runtime comparison against the inference engines mentioned above, also including ProbLog Exact and MC. White means the method is the fastest, striped that it solves the problem but is not the fastest, and black means timeout (1 hour) or invalid answer. For smaller domains MCMC run time versus Exact is overestimated as we converge faster than $100,000$ samples.

Bher's MCMC algorithm has difficulty with the hard constraints in this problem and cannot switch between the two non-zero probability states, returning a probability of either 0 or 1. Solving this problem requires running inference multiple times and averaging results (100 times $1,000$ samples). For Alchemy with the MC-SAT inference algorithm, the CNF conversion times out for any domain with more than 9 bits. MCMC can solve more problems than any of the other four approaches. In this task, we outperform them because we propose states probabilistically which eventually allows full exploration of the state space.



(a) MCMC (b) Bher (c) Church (d) Exact (e) MC

Fig. 3: Runtime: White=fastest, Striped=solves problem, Black=timeout/error.

Goal 3: Poisson and Uniform Distributions We model a single server queue, showing a practical problem using these distributions. We assume that (1) the expected number of customer arrivals is 4 (i.e., Poisson distribution with $\lambda = 4$), and (2) the number of customers served is uniformly distributed between 1 and 8. At time t_0 the number of customers in the queue is 10. At $t_5 = t_0 + 5$, we observe (i.e., e) 12 customers. We want to find the posterior distributions of the number of customers in the queue at t_2 and number of customers served at t_3.

We ran our MCMC algorithm 20 times, each with $500,000$ samples. The average runtime was 12 minutes. Figure 4 shows the prior and posterior distributions for the two queries. The posterior puts more weight on a higher number of customers in the queue at t_2 and a smaller number of customers served at t_3.

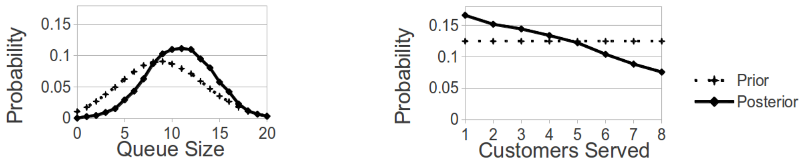


Fig. 4: Different Probability Distributions: at t_2 (left), at t_3 (right).

7 Conclusion

We presented an MCMC algorithm for estimating the conditional probability of a query given evidence in ProbLog. Our proposal distribution proposes candidate states by sampling solution trees from an AND/OR tree. Handling potential overlap between partial worlds is solved by employing ideas from the Karp and Luby algorithm. We provide support for Poisson and uniform distributions. We outperform existing ProbLog inference techniques on the considered tasks.

Acknowledgements. Bogdan Moldovan is supported by the IWT (agentschap voor Innovatie door Wetenschap en Technologie). This work is supported by the European Community's 7th Framework Programme, grant agreement First-MM-248258.

References

1. C. Andrieu, N. de Freitas, A. Doucet, and M. I. Jordan. An introduction to MCMC for machine learning. *Machine Learning*, 50, 2003.
2. L. De Raedt, A. Kimmig, and H. Toivonen. Problog: A probabilistic Prolog and its application in link discovery. In *IJCAI*, pages 2462–2467, 2007.
3. D. Fierens, G. Van den Broeck, I. Thon, B. Gutmann, and L. De Raedt. Inference in probabilistic logic programs using weighted CNF's. In *UAI*, 2011.
4. V. Gogate and R. Dechter. AND/OR importance sampling. In *UAI*, 2008.
5. V. Gogate and P. Domingos. Probabilistic theorem proving. *CoRR*, abs/1202.3724, 2012.
6. N. Goodman, V. K. Mansinghka, D. M. Roy, K. Bonawitz, and J. B. Tenenbaum. Church: a language for generative models. In *UAI*, pages 220–229, 2008.
7. R. W. Hamming. Error detecting and error correcting codes. *Bell System Technical J.*, 29:147, April 1950.
8. W. K. Hastings. Monte Carlo sampling methods using Markov chains and their applications. *Biometrika*, 57:97–109, 1970.
9. R. M. Karp and M. Luby. Monte-carlo algorithms for enumeration and reliability problems. In *FOCS*, pages 56–64. IEEE Computer Society, 1983.
10. A. Kimmig. *A Probabilistic Prolog and its Applications.* PhD thesis, Informatics Section, Department of Computer Science, KU Leuven, Belgium, November 2010.
11. A. Kimmig, B. Demoen, L. De Raedt, V. Santos Costa, and R. Rocha. On the implementation of the probabilistic logic programming language ProbLog. *Theory and Practice of Logic Programming*, 11:235–262, 2011.
12. S. Kok, M. Sumner, M. Richardson, P. Singla, H. Poon, D. Lowd, J. Wang, A. Nath, and P. Domingos. The alchemy system for statistical relational AI. Technical report, Dept. of Computer Science and Engineering, U. of Washington, WA, 2010.
13. N. Metropolis, A. W. Rosenbluth, M. N. Rosenbluth, A. H. Teller, and E. Teller. Equations of state calculations by fast computing machine. *Journal of Chemical Physics*, 21:1087–1091, 1953.
14. B. Milch, B. Marthi, S. Russell, D. Sontag, D. L. Ong, and A. Kolobov. BLOG: Probabilistic models with unknown objects. In *IJCAI*, pages 1352–1359, 2005.
15. J. D. Park. Using weighted MAX-SAT engines to solve MPE. In *AAAI/IAAI*, pages 682–687, Menlo Parc, CA, USA, 2002. AAAI Press.
16. A. Pfeffer. IBAL : A probabilistic rational programming language. In *IJCAI*, 2001.
17. D. Poole. The independent choice logic for modelling multiple agents under uncertainty. *Artif. Intell*, 94(1-2):7–56, 1997.
18. K. Kersting, L. De Raedt. Bayesian logic programs. *CoRR*, cs.AI/0111058, 2001.
19. M. Richardson and P. Domingos. Markov logic networks. *Machine Learning*, 62(1-2):107–136, 2006.
20. T. Sato. A statistical learning method for logic programs with distribution semantics. In *In ICLP*, pages 715–729. MIT Press, 1995.
21. T. Sato. A general MCMC method for bayesian inference in logic-based probabilistic modeling. In *IJCAI*, pages 1472–1477. IJCAI/AAAI, 2011.
22. T. Sato and Y. Kameya. PRISM: A language for symbolic-statistical modeling. In *IJCAI*, pages 1330–1339, 1997.
23. L. G. Valiant. The complexity of enumeration and reliability problems. *SIAM Journal on Computing*, 8:410–421, 1979.
24. D. Wingate, A. Stuhlmüller, and N. D. Goodman. Lightweight implementations of probabilistic programming languages via transformational compilation. *Journal of Machine Learning Research - Proceedings Track*, 15:770–778, 2011.