Mach Learn (2012) 89:215–231
DOI 10.1007/s10994-012-5304-9

k-Optimal: a novel approximate inference algorithm for ProbLog

Joris Renkens · Guy Van den Broeck · Siegfried Nijssen

Received: 3 February 2012 / Revised: 23 March 2012 / Accepted: 8 June 2012 /
Published online: 27 July 2012

Abstract ProbLog is a probabilistic extension of Prolog. Given the complexity of exact inference under ProbLog's semantics, in many applications in machine learning approximate inference is necessary. Current approximate inference algorithms for ProbLog however require either dealing with large numbers of proofs or do not guarantee a low approximation error. In this paper we introduce a new approximate inference algorithm which addresses these shortcomings. Given a user-specified parameter k, this algorithm approximates the success probability of a query based on at most k proofs and ensures that the calculated probability p is $(1 - 1/e)p^* \le p \le p^*$, where p^* is the highest probability that can be calculated based on any set of k proofs. Furthermore a useful feature of the set of calculated proofs is that it is diverse. Our experiments show the utility of the proposed algorithm.

Keywords Inductive logic programming · Statistical relational learning · Decision theory · Approximative inference

1 Introduction

ProbLog (De Raedt et al. 2007) is a probabilistic extension of Prolog. It has been used to solve learning problems in probabilistic networks as well as other types of probabilistic data (De Raedt et al. 2010). The key feature of ProbLog is its distribution semantics. Each fact in a ProbLog program can be annotated with the probability that this fact is true in a random sample from the program. The success probability of a query is equal to the probability that the query succeeds in a sample from the program, where facts are sampled independently from each other. Each such sample is also called a *possible world*.

The main problem in calculating the success probability of a query in ProbLog is the high computational complexity of exact inference. As multiple proofs for a query can be

Editors: S. Muggleton and J. Chen.

J. Renkens (✉) · G. Van den Broeck · S. Nijssen
Department of Computer Science, Katholieke Universiteit Leuven, Celestijnenlaan 200A,
3001 Heverlee, Belgium
e-mail: joris.renkens@cs.kuleuven.be

true in a possible world, we cannot calculate the success probability of a query based on the probabilities of the independent proofs; we need to deal with a *disjoint sum problem* (De Raedt et al. 2007). This problem becomes worse as the number of proofs grows.

To deal with this computational issue, several approaches have been proposed in the past. De Raedt et al. (2007) proposed to use Binary Decision Diagrams (BDDs) (Bryant 1986) to deal with the disjoint sum problem. BDDs can be seen as a representation of proofs from which the required success probability can be calculated in time polynomial in the size of the BDD. Building a BDD for all proofs can however be intractable. De Raedt et al. showed that for a given desired approximation factor ϵ, an iterative deepening algorithm can be used to approximate the success probability from a subset of proofs. However, to reach reasonable approximation errors in practice, this algorithm still needs to compile large numbers of proofs into a BDD (De Raedt et al. 2007).

A commonly used alternative which does not have this disadvantage is the *k-best* strategy. In this case, the k most likely proofs are searched for, where k is a user-specified parameter; a BDD is constructed based on these proofs only. Whereas this strategy avoids compiling many proofs, its disadvantage is that one has few guarantees with respect to the quality of the calculated success probability: it is not clear whether any other set of k proofs would achieve a better approximation, or how far the calculated probability is from its true value.

A further disadvantage of k-best is found in parameter learning for ProbLog programs (Gutmann et al. 2008, 2011) and the use of ProbLog in solving probabilistic decision problems (Van den Broeck et al. 2010). An example of such a decision problem is targeted advertising in social networks: in order to reduce advertising cost, one wishes to identify a small subset of nodes in a social network such that the expected number of people reached is maximized. Such problems can be solved by defining an optimization problem on top of a set of BDDs. To ensure that these calculations remain tractable, it is important that the BDDs are as small as possible but still represent the most relevant proofs. As we will show, the k-best strategy selects a set of proofs that is not optimal for this purpose. Intuitively the reason is that the k-best proofs can be highly redundant with respect to each other.

In this paper we propose a new algorithm, k-optimal, for finding a set of at most k proofs. The key distinguishing feature with respect to k-best is that it ensures that the set of k proofs found is of provably good quality. In particular, if p^* is the best probability that can be calculated based on k proofs, our algorithm will not calculate a probability that is worse than $(1 - 1/e)p^*$. At the same time, our algorithm ensures that the resulting set of proofs is diverse, i.e., proofs are less likely to use similar facts; the resulting set of proofs carries more of the probability mass of the exact success probability. We will empirically show that using k-optimal leads to significant memory gains and can lead to reduced runtime for similar or better approximations of the true probability. Furthermore we will show that using k-optimal for inference in DTProbLog can improve the results over the use of k-best, by reducing runtime for sufficiently high k values.

The remainder of this paper is organized as follows: Sect. 2 introduces ProbLog and DTProbLog and discusses the drawbacks of k-best; Sect. 3 introduces the new algorithm; Sect. 4 proves the quality of the resulting set of proofs; Sect. 5 reports on some experiments and finally Sect. 6 concludes.

2 Background

We will now introduce ProbLog and describe how to compute the probability of a query using BDDs, both exactly and approximately with the k-best algorithm.

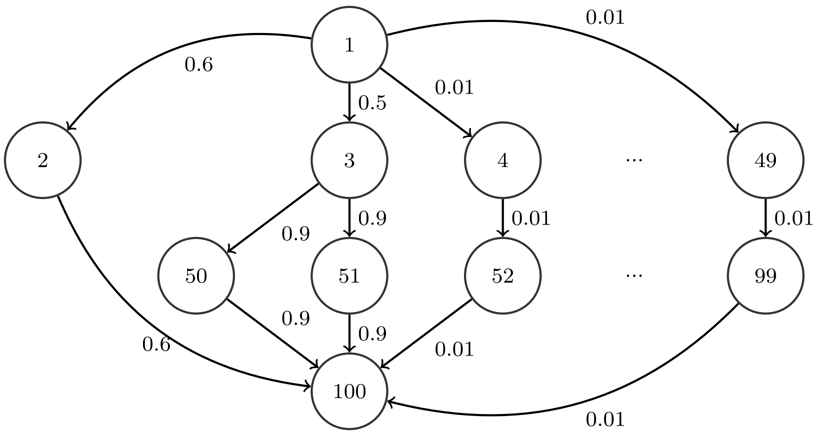


Fig. 1 Probabilistic network example

2.1 ProbLog

A ProbLog program consists of two parts: a set of probabilistic facts $\mathcal{PF}$ and a set of rules $\mathcal{R}$. The rules are written in standard Prolog syntax. The syntax for the probabilistic facts is $p_i :: f_i$, meaning that fact $f_i\theta$ is true with probability p_i for all substitutions θ that ground f_i. Probabilistic facts represent independent random variables. Let $\mathcal{GF}$ be the set of all g_j where g_j is a grounding of some f_i in $\mathcal{PF}$. The ProbLog program represents a probability distribution over logic programs $F \subseteq \mathcal{GF}$, which consist of ground facts.

$$\mathrm{P}(F) = \prod_{f_i \in F} p_i \prod_{f_i \in \mathcal{GF}\setminus F} (1 - p_i) \tag{1}$$

For a fixed set of rules $\mathcal{R}$, each logic program F corresponds to a unique Herbrand interpretation of the logic program $\mathcal{R} \cup F$. Therefore, $\mathrm{P}(F)$ also defines a probability distribution over these Herbrand interpretations. We will assume that $\mathcal{GF}$ is finite. See Sato (1995) for an extension of these semantics to the infinite case.

The *success probability* of a query q is the probability that it succeeds in a randomly sampled logic program. This is defined as

$$\mathrm{P}(q) = \sum_{F \subseteq \mathcal{GF}} \mathbf{1}_q(F) \cdot \mathrm{P}(F), \tag{2}$$

where $\mathbf{1}_q(F) = 1$ if $\mathcal{R} \cup F \models q\theta$ for some substitution θ, and 0 otherwise.

Example 1 This running example shows the implementation of the small probabilistic network of Fig. 1. The network itself is defined by probabilistic facts $\mathcal{PF}$. The definition of a path in the network is encoded by the rules $\mathcal{R}$.

$$\mathcal{PF} = \left\{ \begin{array}{c} \texttt{0.6} :: \texttt{edge(1,2).} \\ \texttt{0.5} :: \texttt{edge(1,3).} \\ \vdots \end{array} \right\} \qquad \mathcal{R} = \left\{ \begin{array}{l} \texttt{path(X,Y) :- edge(X,Y).} \\ \texttt{path(X,Y) :- edge(X,Z),path(Z,Y).} \end{array} \right\}$$

Computing the probability of a query q is performed by finding all proofs for the query using SLD-resolution in the ProbLog program. Each proof relies on the conjunction of probabilistic facts that need to be true to prove q. From now on we will use the term 'proof' also for this conjunction. The disjunction of these conjunctions is a DNF formula that represents all proofs, that is, all conditions under which q is true.

Example 2 In the network of Fig. 1, each proof reflects one set of edges that needs to be present in a possible world in order for a path to exist. Worlds in which a path between node 1 and 100 exists are characterized by the following formula in DNF:

$$\begin{aligned}&\big(\mathtt{e}(1,2) \wedge \mathtt{e}(2,100)\big)\\ &\quad\vee \big(\mathtt{e}(1,3) \wedge \mathtt{e}(3,50) \wedge \mathtt{e}(50,100)\big)\\ &\quad\vee \big(\mathtt{e}(1,3) \wedge \mathtt{e}(3,51) \wedge \mathtt{e}(51,100)\big)\\ &\quad\vee \big(\mathtt{e}(1,4) \wedge \mathtt{e}(4,52) \wedge \mathtt{e}(52,100)\big)\\ &\quad\vee \dots\end{aligned}$$

Here, $\mathtt{e(X,Y)}$ denotes the fact $\mathtt{edge(X,Y)}$ being true in the possible world.

This reduces the problem of computing the probability of the query q to that of computing the probability of a DNF formula, i.e. $\mathrm{P}(q) = \mathrm{P}(\bigvee_{p\in V} p)$, where V is the set of proofs for query q; $\mathrm{P}(\bigvee_{p\in V} p) = \sum_{F\subseteq \mathcal{G}\mathcal{F}} \mathrm{P}(\bigvee_{p\in V} p|F)\cdot \mathrm{P}(F)$. However, because the conjunctions in the different proofs are not mutually exclusive, one cannot compute the probability of the DNF formula as the sum of the probabilities of the different conjunctions as this could lead to values larger than one.

Therefore, the next step in the evaluation addresses this disjoint-sum problem by constructing a Binary Decision Diagram (BDD) (Bryant 1986) that represents the DNF formula. A BDD is an efficient graphical representation for Boolean formulas. It can be seen as a decision tree in which identical subtrees are shared by multiple parents. The probability of a DNF can be calculated from a BDD by traversing the BDD bottom-up. Each internal node represents a probabilistic fact; the probability for a node is computed by combining the probabilities of its children; for details see the paper of De Raedt et al. (2007).

2.2 *k*-Best

Computing the success probability of a query as described in Sect. 2.1 can fail for two reasons. First, finding all proofs for the query can be intractable. Second, compiling the set of proofs into a BDD solves a #P-complete problem, which quickly becomes intractable when the number of proofs increases. The k-best inference algorithm solves these problems by only considering a fixed number of k proofs in the calculation of the approximate success probability. The effect of ignoring some of the proofs is that the query is evaluated to true only in a subset of the possible worlds where the query should succeed. Therefore, the k-best strategy computes a lower bound on the success probability.

To approximate the exact probability, k-best selects the k most probable proofs. The probability of a proof is the product of the probabilities of the probabilistic fact literals occurring in the proof. Finding this set is done by depth-first branch and bound search on the proof tree induced by SLD-resolution (Algorithm 1). An inner node of the proof tree represents a partial proof, consisting of literals for probabilistic facts. An inner node might

Algorithm 1 *k*-Best

```
1: function K-BEST($k$,$r$)    ▷ number of proofs $k$    ▷ proof tree root node $r$
2:     $A \leftarrow \emptyset$    ▷ set of $k$ best proofs
3:     $p_{min} = 0$    ▷ minimum proof probability
4:     $stack = \{r\}$    ▷ stack of proof tree nodes
5:     while $stack$ not empty do
6:         pop $n$ from $stack$
7:         let $pr_n$ be the (partial) proof in $n$
8:         if $pr_n$ is a complete proof then
9:             $A \leftarrow A \cup \{pr_n\}$
10:            if $|A| > k$ then
11:                $A \leftarrow A \setminus \arg\min_{pr \in A} \mathrm{P}(pr)$
12:                $p_{min} \leftarrow \min_{pr \in A} \mathrm{P}(pr)$
13:        else if $\mathrm{P}(pr_n) > p_{min}$ then
14:            push the child nodes of $n$ onto $stack$
15:    return $A$
```

lead to one or more leaf nodes, which extend the partial proof into a complete proof of the query. *k*-Best avoids searching the space of all proofs by keeping track of the minimum probability p_{min} required for a proof to make it into the set of k best proofs A. The probability of a partial proof gives an upper bound on the probability of any complete proof that extends it. This means we can avoid searching child nodes whose partial proofs have probability below p_{min}.

Example 3 The three best proofs in the network of Example 1 are $(\mathtt{e}(1,2) \wedge \mathtt{e}(2,100))$, $(\mathtt{e}(1,3) \wedge \mathtt{e}(3,50) \wedge \mathtt{e}(50,100))$ and $(\mathtt{e}(1,3) \wedge \mathtt{e}(3,51) \wedge \mathtt{e}(51,100))$, with probabilities $0.6^2 = 0.36$ and $0.5 \cdot 0.9^2 = 0.405$. When the search algorithm reaches node four of the network, p_{min} is 0.36, but the partial proof that leads to node four in the network only has probability 0.01. Therefore, the algorithm does not continue to search for the proof $(\mathtt{e}(1,4) \wedge \mathtt{e}(4,52) \wedge \mathtt{e}(52,100))$.

k-Best's ability to compile small, approximate BDDs is especially useful in the context of DTProbLog and parameter learning for ProbLog programs (Gutmann et al. 2008, 2011). These algorithms build and store BDDs for a large number of queries. It is essential that these BDDs are small for the algorithms to be effective. Although *k*-best guarantees small BDDs, it does not guarantee good approximations. When selecting proofs based on their probability, it is impossible to detect redundancy between the proofs. This can result in the selection of a highly redundant set of proofs which do not approximate the probability in an optimal way.

2.3 DTProbLog

DTProbLog (Van den Broeck et al. 2010) is an extension of ProbLog which was developed to solve probabilistic decision problems. An example of such a decision problem is targeted advertising in social networks: in order to reduce cost, one wishes to identify a small subset of nodes in a social network such that the expected number of people reached is maximized. Probabilistic decision problems for ProbLog can be formalized in the following way:

Given: (1) a set of utility queries $\mathcal{U}$ where each query has a reward given by a function $u : \mathcal{U} \to \mathbb{R}$; (2) a set of boolean decision facts $\mathcal{D}$, for each of which a value from $\{0, 1\}$ has to be determined; (3) a ProbLog program $\mathcal{PF}, \mathcal{R}$.
Find: a decision for each decision fact $s : \mathcal{D} \to \{0, 1\}$.
Such that: the score given by $\sum_{g\in\mathcal{U}} \mathrm{P}(g)u(g)$ is maximal.
Where: the probabilities are calculated using the ProbLog program $\mathcal{PF} \cup \{s(d) :: d | d \in \mathcal{D}\}, \mathcal{R}$; note that we can see binary decisions as zero/one probability assignments.

DTProbLog offers multiple algorithms for finding the solution to these problems. One algorithm which gives the globally optimal solution, calculates the BDDs for each of the utility queries and combines these BDDs into an Algebraic Decision Diagram (ADD) (Van den Broeck et al. 2010). From this ADD, the solution can be found in polynomial time. This approach can become intractable when the BDDs for the utility queries become too large. A second algorithm uses a greedy hill climbing algorithm to find an approximate solution. The calculations are sped up by calculating the BDDs for each utility query once, and reusing them for each evaluation of the score. As only the probabilities of the decision facts change, the structure of the BDDs stays the same; after building the BDDs once, the rest of the score evaluations can be done in polynomial time.

For both algorithms, it is important that the BDDs stay small enough. The construction of the ADD quickly becomes intractable when the BDDs become large and even for the approximate algorithm, the BDDs for the goals need the fit in memory. Small BDDs can be obtained using approximative inference algorithms.

3 *k*-Optimal

We will introduce k-optimal, which is the main contribution of this paper. It is an approximative algorithm for calculating the success probability, similar to k-best. First the intuition behind the algorithm is explained after which a straightforward implementation is given. Later on, optimizations for the implementation are described which speed up k-optimal significantly.

3.1 Intuition

The problem for which our algorithm finds an approximation is the following:

Given: the collection V of all possible proofs for a goal and a maximum number k of proofs which can be used.
Find: a collection $A = \arg\max_{B\subseteq V, |B|\le k} \mathrm{P}(\bigvee_{p\in B} p)$.

The algorithm that we propose is an implementation of a simple greedy algorithm. This algorithm constructs a set of k proofs by starting with $A = \emptyset$ and iteratively adding the proof that maximizes the *added probability*, which is the probability $\mathrm{P}(A \cup \{pr\}) - \mathrm{P}(A)$ (Algorithm 2). As $\mathrm{P}(A)$ is a constant in each iteration, this is equivalent to greedily adding the proof that achieves the highest $\mathrm{P}(A \cup \{pr\})$.

Example 4 We compare the results obtained by 2-best and 2-optimal on the network of Example 1. 2-Best selects proofs $(\mathtt{e}(1, 3) \wedge \mathtt{e}(3, 50) \wedge \mathtt{e}(50, 100))$ and $(\mathtt{e}(1, 3) \wedge \mathtt{e}(3, 51) \wedge \mathtt{e}(51, 100))$ because they have the highest proof probability. This results in a probability of $\mathrm{P}(\mathtt{e}(1, 3) \wedge \mathtt{e}(3, 50) \wedge \mathtt{e}(50, 100) \vee \mathtt{e}(1, 3) \wedge \mathtt{e}(3, 51) \wedge \mathtt{e}(51, 100)) = 0.405$. The first iteration of 2-optimal will select $(\mathtt{e}(1, 3) \wedge \mathtt{e}(3, 50) \wedge \mathtt{e}(50, 100))$ because in this iteration,

Algorithm 2 *greedy_solve*(V)

```
$A \leftarrow \emptyset$
for $i = 1..k$ do
    $A \leftarrow A \cup \arg\max_{pr \in V} \mathrm{P}(A \cup \{pr\})$
return $A$
```

the added probability is equal to the proof probability. In the second iteration however, a different proof will be selected. The added probability of $(\mathtt{e}(1,2) \wedge \mathtt{e}(2,100))$ is higher than the added probability of $(\mathtt{e}(1,3) \wedge \mathtt{e}(3,51) \wedge \mathtt{e}(51,100))$ even though the proof probability is lower. This results in a probability of $\mathrm{P}(\mathtt{e}(1,3) \wedge \mathtt{e}(3,50) \wedge \mathtt{e}(50,100) \vee \mathtt{e}(1,2) \wedge \mathtt{e}(2,100)) = 0.48195$ which is higher than the one calculated by 2-best.

There are several problems that complicate implementing this algorithm. These are addressed in the following section.

3.2 Implementation

Collecting all possible proofs before starting the greedy algorithm is undesirable, as collecting these proofs can be intractable. In k-best this problem is solved by a branch and bound search of the SLD-tree. We can use a similar solution in the k-optimal greedy algorithm. In each iteration we start a branch-and-bound search with a modified scoring function. Compared to 1-best, we now optimize $\mathrm{P}(A \cup \{pr\})$ instead of $\mathrm{P}(pr)$. Because for any pair of (partial) proofs $pr \subset pr'$, $\mathrm{P}(A \cup \{pr\}) \geq \mathrm{P}(A \cup \{pr'\})$, we eliminate branches for which $\mathrm{P}(A \cup \{pr\})$ is not sufficiently high. Hence, 1-best is modified by replacing the calculation of $\mathrm{P}(pr)$ in lines 11 to 13 of Algorithm 1 with a calculation of $\mathrm{P}(A \cup \{pr\}) - \mathrm{P}(A)$.

During each iteration, the added probability for many proofs in the SLD search tree has to be calculated. The main task that needs to be addressed is hence the efficient calculation of $\mathrm{P}(A \cup \{pr\})$ where pr is a conjunction of probabilistic facts. In a naïve approach this would involve building the BDD for $A \cup \{pr\}$ from scratch for each pr. Fortunately, we can avoid this. Let *dnf* represent the DNF formula $\bigvee_{pr' \in A} pr'$ for the set of proofs A and let $pr \equiv f_1 \wedge \cdots \wedge f_n$. Then

$$\begin{aligned}\mathrm{P}(f_1 \wedge \cdots \wedge f_n \vee \mathit{dnf}) = {} & \mathrm{P}(f_1 \wedge \cdots \wedge f_n) + \mathrm{P}(\neg f_1 \wedge \mathit{dnf}) \\ & + \mathrm{P}(f_1 \wedge \neg f_2 \wedge \mathit{dnf}) + \cdots \\ & + \mathrm{P}(f_1 \wedge \cdots \wedge f_{n-1} \wedge \neg f_n \wedge \mathit{dnf}).\end{aligned}$$

Because the probabilistic facts in ProbLog are all assumed to be independent, all the terms can be decomposed into smaller parts. The first term can be calculated as the product of the probabilities of the individual used facts. The other terms in this sum can be calculated as follows:

$$\mathrm{P}(f_1 \wedge \cdots \wedge f_{i-1} \wedge \neg f_i \wedge \mathit{dnf}) = \mathrm{P}(f_1) \ldots \mathrm{P}(f_{i-1}) \big(1 - \mathrm{P}(f_i)\big) \mathrm{P}(\mathit{dnf} | f_1 \wedge \cdots \wedge f_{i-1} \wedge \neg f_i),$$

where we still need to specify how to calculate $\mathrm{P}(\mathit{dnf} | f_1 \wedge \cdots \wedge f_{i-1} \wedge \neg f_i)$. Fortunately, this term can be calculated efficiently if we already have the BDD for the *dnf* formula. We reuse the algorithm for calculating $\mathrm{P}(\mathit{dnf})$ from a BDD (De Raedt et al. 2007), where during the traversal of the BDD, we assume that $\mathrm{P}(f_j) = 1$ for the conditional facts $j < i$ while setting $\mathrm{P}(f_i) = 0$.

Overall, a basic strategy is hence to compile the BDD for the set of proofs A at the beginning of each search iteration. For each visited node of the SLD-tree we calculate the added probability, $\mathrm{P}(A \cup \{p\})$, where p is the partial proof that is used to reach the node. Every calculation of the added probability requires the calculation of n conditional probabilities.

Even though we avoid building a BDD in every node of the search tree, calculating the conditional probability remains a time consuming operation. The next section introduces optimizations to reduce the number of conditional probabilities that need to be calculated.

3.3 Optimizations

A first optimization results in the calculation of only one conditional probability per node of the SLD-tree. We can make the following observation: when the added probability for the (partial) proof $f_1 \wedge \cdots \wedge f_n \wedge f_{n+1}$ needs to be calculated, the added probability for $f_1 \wedge \cdots \wedge f_n$ has already been calculated in the parent of the node. This means that the conditional probabilities $\{\mathrm{P}(dnf|f_1 \wedge \cdots \wedge f_{i-1} \wedge \neg f_i)|1 \leq i \leq n\}$ have already been calculated and the only conditional probability that needs to calculated in order to calculate the added probability is $\mathrm{P}(dnf|f_1 \wedge \cdots \wedge f_n \wedge \neg f_{n+1})$.

Bounding the SLD-tree based on the added probability of a partial proof, will prune a node as soon as it is known that the proofs below that node will not achieve a high enough added probability. However, in Sect. 5 it will be shown experimentally that this approach entails a significant overhead in terms of added probability calculations. In practice it is actually more efficient to use the bound of k-best for incomplete proofs and only calculate added probability once a complete proof is found. This means that only in lines 11 and 12 of Algorithm 1 the calculation of $\mathrm{P}(pr)$ is replaced with a calculation of $\mathrm{P}(A \cup \{pr\}) - \mathrm{P}(A)$. Intuitively this is explained by the fact that no expensive operations will have to be calculated for nodes of the SLD-tree which do not result in a proof later on. The correctness of this strategy follows from the observation that

$$\forall A \subseteq V : \mathrm{P}(pr) \geq \mathrm{P}\big(A \cup \{pr\}\big) - \mathrm{P}(A);$$

hence $\mathrm{P}(p)$ will never underestimate the quality of the solution that can still be reached. Using the bound of k-best will reduce the computational overhead but also has some disadvantages. Because the proof probability is never smaller than the added probability, less pruning of the search space is possible.

This modified strategy requires more extensive book-keeping, as we can no longer assume that we have calculated an added probability for the parent of a complete proof in the search tree. By caching the results of intermediate added probability calculations, however, we can still share the calculations between multiple complete proofs.

Furthermore we can optimize the calculation by changing the order in which facts in a proof are considered. When the fact f_n does not occur in the BDD, $\mathrm{P}(dnf|f_1 \wedge \cdots \wedge f_n) = \mathrm{P}(dnf|f_1 \wedge \cdots \wedge f_{n-1})$. Moving these facts to the front of the proof will result in the first conditional probabilities being equal to $\mathrm{P}(dnf)$, which has been calculated in a previous iteration.

As in any branch-and-bound algorithm it is important to quickly find a good bound on the best added probability that can be reached. In k-optimal, this is solved in the following way. In the first iteration we reuse the strategy of k-best. In further iterations, the bound is initialized based on the proofs that have been encountered in previous iterations, as follows. In each iteration, whenever the added probability for a complete proof is calculated, the proof as well as its added probability are stored. Before starting the SLD-tree search in the next iteration, we traverse these stored proofs in decreasing order of added probability in the previous iteration. For each proof we recalculate its added probability in the new iteration (caching any intermediate results for later use during the SLD search). During this traversal we in practice quickly find increasingly better added probability thresholds, which allow us to eliminate later proofs in the list as well as proofs later in the SLD search. Here we exploit the observation that the added probability for a proof pr can not increase in a later iteration. Although proofs are stored, SLD search is still necessary in each iteration as partial proofs which previously have been pruned can be relevant in later iterations.

3.4 k-θ-Optimal

Both k-best and k-optimal suffer from the following problem. Before the approximate probability is calculated, the user needs to specify the number of proofs that have to be used to approximate the probability. When enough proofs are present k proofs will always be used, even when a lot of these are insignificant. However, there is no way of knowing the number of significant proofs; we would need to start k-best with a sufficiently high parameter. This is undesirable as the complexity of computing the BDD for a collection of proofs can increase exponentially with the number of proofs.

In the case of k-best, it is only possible to leave proofs out based on their individual proof probabilities. However, even proofs with a high proof probability can be insignificant when combined with other proofs. Within k-optimal, we can detect insignificant proofs by imposing a threshold θ on the added proof probability. We stop the algorithm before k proofs are selected when no more proofs with an added probability higher than θ can be found. The resulting algorithm is called k-θ-optimal and is obtained by initializing the bound with at least θ in the beginning of each iteration.

Example 5 The network of Example 1 contains 49 proofs for a path between node 1 and node 100. Of these proofs only $(\mathtt{e}(1,2) \wedge \mathtt{e}(2,100))$, $(\mathtt{e}(1,3) \wedge \mathtt{e}(3,50) \wedge \mathtt{e}(50,100))$ and $(\mathtt{e}(1,3) \wedge \mathtt{e}(3,51) \wedge \mathtt{e}(51,100))$ are significant as the other proofs have a probability equal to 0.00001. When we run 10-optimal, 10 proofs will be used to construct the BDD. This is not the case with 10-θ-optimal when $\theta > 0.000001$ because the added probability of the insignificant proofs will never be high enough to be selected. As a result, the constructed BDD will be much smaller without calculating a much lower probability.

4 Analysis

First, we would like to recall that calculating the probability for an arbitrary set of proofs can be a hard problem in itself: given that they solve a #P problem, all current algorithms are exponential in k, where k is the size of the set of proofs. To gain a better understanding in the k-optimal problem itself, we hence limit ourselves in the remainder of this analysis to instances of the k-optimal problem where probabilities can be calculated in polynomial time.

To show the hardness of the k-optimal problem, we first introduce the *β-k-optimal problem*, which is a decision version of the k-optimal problem defined as follows.

Given the collection V of all possible proofs for a goal q in a ProbLog program, a maximum number k of proofs that can be used, and a probability threshold β.
Decide is there a collection $B \subseteq V$ with $|B| \leq k$ and $\mathrm{P}(\bigvee_{p \in B} p) \geq \beta$?

We can show that under certain restrictions this problem is NP-complete. To this aim, let us first introduce β-polynomial ProbLog goals. A goal in a ProbLog program is called β-*polynomial* if the probability $\mathrm{P}(B)$ can be calculated in polynomial time for any set of proofs B for this goal with $\mathrm{P}(B) \geq \beta$.

Theorem 1 *The β-k-optimal decision problem for β-polynomial goals is NP-complete.*

Proof First, we observe that the decision problem is in NP: by assumption, for each subset B we can verify in polynomial time whether a solution is valid. Note that without our assumption, computing the probability of a goal may be exponential in k; hence, in this case the problem is only fixed-parameter tractable.

Next, we show that the *3-set packing problem*, which is known to be NP-complete (Hazan et al. 2006), can be reduced to the k-optimal decision problem for β-polynomial goals. The 3-set packing problem is defined as follows. Given is a universe of elements X, and a collection V of subsets of X, where each subset is of size 3. The problem is to decide if there exists a subset V' of V of size k such that each pair of elements in V' is disjoint.

We can encode this problem as follows.

– For each element x in the universe X we define a probabilistic fact $\alpha :: \mathtt{p(x)}$ with some probability α which is the same for all facts;
– For each subset $\{x_1, x_2, x_3\}$ in the collection we define a clause

$$\mathtt{q :- p(x_1), p(x_2), p(x_3).}$$

Hence, there are multiple proofs for q, each corresponding to one subset of the collection. In the context of this theorem, we assume that all these proofs are given; note however that they can be calculated in polynomial time.

Then we can show the following:

– If we can find a subset of proofs of size k, all of which have disjoint sets of probabilistic facts, the probability of this set is

$$1 - \left(1 - \alpha^3\right)^k;$$

since each proof has probability α^3 and the k proofs are independent. Note that since the probability for independent proofs can be calculated in polynomial time, the goal q is β-polynomial.
– If a set of proofs of size k has two proofs which share a probabilistic fact, the probability will be lower than this value.

Hence, to decide whether the 3-set packing problem can be solved we have to decide whether the k-optimal problem can be solved for the above Problog program and parameter $\beta = 1 - (1 - \alpha^3)^k$. □

In many cases the assumption that the probability for a set of proofs can be calculated in polynomial time is not valid; however, our theorem shows that even where this is the case, the k-optimal problem is hard to solve.

The theorem hence leads to the following more general observation.

Theorem 2 *The k-optimal optimization problem is NP-hard.*

Proof This follows from the fact that we can use a k-optimal optimizer to solve NP-complete β-k-optimal decision problems. □

Nevertheless our algorithm calculates good solutions. The quality of the result of our algorithm follows from the fact that the function P(.) is *submodular* and *monotone*.

Definition 1 Given a collection S and a function $f : 2^S \rightarrow \mathbb{R}$. Function f is called *submodular* when $\forall A \subseteq B \subseteq S, \forall x \in S : f(A \cup \{x\}) - f(A) \geq f(B \cup \{x\}) - f(B)$. Function f is called *monotone* when $\forall A, B \subseteq S : A \subseteq B \rightarrow f(A) \leq f(B)$.

It can easily be seen that the P(.) function is submodular and monotone: adding a proof to a larger set of proofs will not increase its impact on the overall probability; more possible worlds will already be covered, and a larger set of proofs will result in a larger probability.

Submodular scoring functions allow for a simple greedy approximation algorithm, as shown by Cornuejols et al. (1977).

Theorem 3 *Assume given a submodular and monotone function* $f : 2^S \rightarrow \mathbb{R}$ *on subsets of the set* S. *Let* B *be the set of size* k *for which* $f(B)$ *is maximal. Let* A *be the set of size* k *identified by the following greedy algorithm*:

```
$A \leftarrow \emptyset$
for $i = 1..k$ do
    $A \leftarrow A \cup \{\arg\max_{x \in S} f(A \cup \{x\})\}$
```

Then for the solution found by this algorithm it holds that

$$f(A) \geq \left(1 - \frac{1}{e}\right) f(B).$$

From this theorem it follows that the probability computed by our greedy algorithm for a fixed proof set size is not worse than $1 - \frac{1}{e}$ times the probability of the optimal solution.

Clearly, this theorem does *not* state how far we are from the probability calculated on all proofs. Calculating an upper-bound on the probability for all proofs is a significantly more challenging task. One possibility is the following. When the SLD procedure is running in the last iteration, certain incomplete proof nodes will be pruned as their (added) probability is not sufficiently promising. A useful observation is that if we were to add all complete proofs below this node in our proof set, this can never increase the probability more than the added probability for this incomplete proof (see De Raedt et al. 2007 for a similar observation). Hence, by summing up all added probabilities of pruned incomplete proofs, we obtain an upper-bound on the probability for all proofs. Note, however, that this bound may very well be higher than 1; this bound is not likely to be as accurate as the lower-bound that is calculated based on the submodularity property.

5 Experiments

Three types of experiments have been conducted. In Sect. 5.1 we evaluate the improvements to the implementation of the k-optimal algorithm that are described in Sect. 3.3. We ask the following question: **(Q1)** Is scoring partial proofs based on their added probability slower than using the proof probability as an upper bound? Section 5.2 discusses the difference

Fig. 2 Average runtime of k-optimal in milliseconds, when using added probability or proof probability for pruning

	Added probability	Proof probability
$k = 1$	6226	151
$k = 2$	19116	1113
$k = 3$	51687	2384

in individual performance between k-best, k-optimal and k-θ-optimal. The following questions are answered: **(Q2)** Do k-optimal and k-θ-optimal achieve better approximations than k-best? **(Q3)** Do k-optimal and k-θ-optimal calculate smaller BDDs? **(Q4)** What is the difference in runtime between the different algorithms? To conclude we test the effect of using k-optimal and k-θ-optimal incorporated in more complex inference systems. Section 5.3 investigates the following questions: **(Q5)** What is the effect on the utility calculated by DTProbLog? **(Q6)** How does this difference in calculated utility translate to runtime performance?

For all the experiments, the probabilistic network constructed by Ourfali et al. (2007) is used, which contains 15147 bidirectional edges, 5568 unidirectional edges and 5313 nodes. This biological network represents the regulatory system of a yeast cell; biologists are interested in pathways that explain the effect of one protein in this network on the expression of another. For this purpose, the connection probability for many pairs of nodes needs to be calculated. Because of the sheer size of the network, the probability cannot be calculated exactly and needs to be approximated.

5.1 Implementation of k-optimal

It was argued in Sect. 3.3 that it is more efficient to use the proof probability of partial proofs instead of the added probability to prune the search tree. The advantage of the proof probability is that computing the heuristic is much more efficient because it does not need to evaluate a BDD. The disadvantage is that less pruning of the search tree is possible, and more partial proofs need to be evaluated. Note that in both cases, the objective function is still the added probability, and both implementations are instances of k-optimal.

(Q1) *Is scoring partial proofs based on their added probability slower than using the proof probability as an upper bound?* Both approaches have been implemented and their runtime has been tested on computing the probability of a path, with a maximal length of three edges, between 669 pairs of nodes in the biological network. These queries have been run for k equal to 1, 2 and 3 and the average runtime can be found in Fig. 2. The results show that the implementation which uses the proof probability for pruning performs an order of magnitude better.

5.2 Comparing different approximations

In this section, we approximate the probability of a path between 5371 pairs of nodes using k-best, k-optimal and k-θ-optimal with k values ranging between one and twenty. More pairs are considered than in the previous section as these experiments have been run with a maximal path length equal to four edges and we are able to connect more pairs of nodes.

(Q2) *Do k-optimal and k-θ-optimal achieve better approximations than k-best?* Figure 3 shows the results of the experiments that compare the probabilities obtained by k-optimal and k-best for values of k equal to 1, 6, 11 and 16.

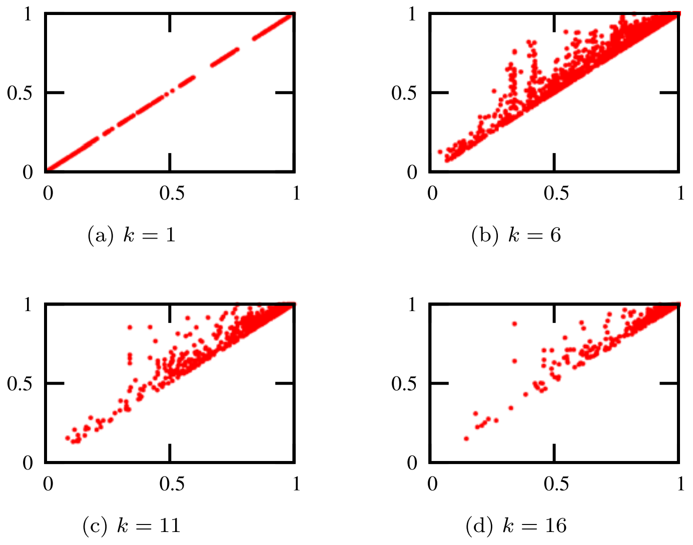


Fig. 3 Ratio of probabilities by k-optimal (y-axis) and k-best (x-axis)

The x-value is equal to the probability achieved using k-best. The y-value is equal to the probability achieved using k-optimal. Only the pairs that have more than k proofs are shown. When this is not the case, all the proofs are selected and k-best and k-optimal achieve the same result.

k-Optimal achieves at least as good results as k-best for all queries and all k values. When k is equal to one, there is no difference between k-best and k-optimal because when selecting the first proof, the added probability of a proof is exactly the same as the probability of the proof itself. In the other cases, k-optimal performs at least as well as k-best. When k is low compared to the number of available proofs ($k = 6$), the difference in calculated probability is the biggest because the selection problem becomes more important. When k is almost equal to the number of proofs ($k = 16$) the remaining proofs are almost completely redundant and it does not matter which proof is left out.

Figure 4 shows the comparison between k-optimal and k-θ-optimal with $\theta = 0.01$. The x-axis contains the probability calculated using k-θ-optimal while the y-axis contains the probability calculated using k-optimal. k-θ-Optimal will always achieve a lower probability than k-optimal because it uses the same selection criterion, but it can stop early when the added probability becomes too low. Although k-θ-optimal will never achieve better results than k-optimal in terms of calculated probabilities, the difference in calculated probability is not as large as the differences observed in the comparison with k-best. Because of this, we can conclude that k-θ-optimal successfully avoids adding insignificant proofs.

(Q3) *Do k-optimal and k-θ-optimal calculate smaller BDDs?* Figure 5a shows the average BDD size and the probability obtained by the three algorithms. Each point represents the average over all path queries for a single k value. The results clearly show that k-optimal achieves better approximations of the probabilities with the same BDD size. This means that the increase in probability is preserved when comparing the BDD size and not the number of proofs.

k-Optimal also achieves better results than k-θ-optimal with $\theta = 0.01$, so stopping early is not reflected in a lower BDD size as a function of the probability. However, k-θ-optimal

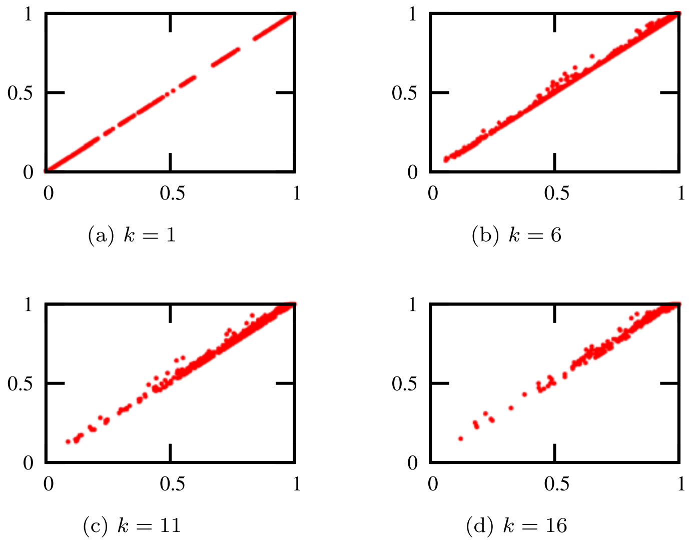


Fig. 4 Ratio of probabilities by k-optimal (y-axis) and k-θ-optimal (x-axis)

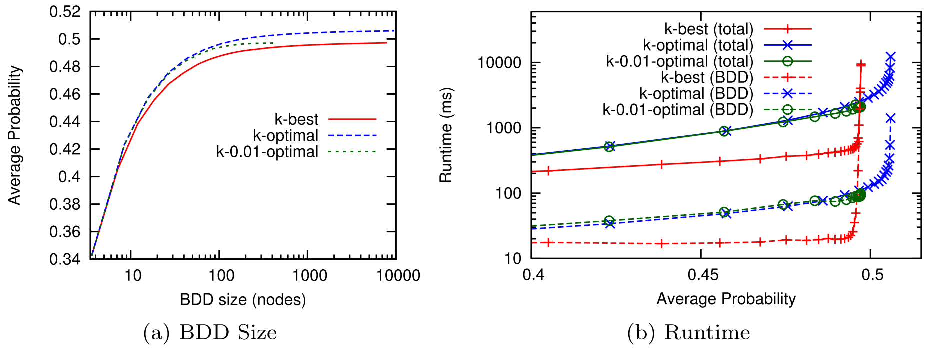


Fig. 5 Average probability for increasing BDD size and average runtime as a function of the average achieved probability with varying k values. Both the runtime for building the BDDs and the total runtime are shown

does manage to limit the BDD size for all pairs. For high k values, the average BDD size remains relatively small when we compare it to the BDD size for k-optimal and k-best. Hence k-θ-optimal is useful when small BDDs need to be guaranteed.

(Q4) *What is the difference in runtime between the different algorithms?* Figure 5b shows the average computation time and BDD construction time (both in ms) as a function of the average probability for varying k values. Each point represents the averaged results for one k value. When we are using low k values, the time that is needed to compute the BDDs is not dominant and k-best achieves better results due to lower search time. However, with higher k values, the BDD construction time grows exponentially; as k-optimal constructs smaller BDDs for similar calculated probabilities, it has an advantage for such values.

5.3 DTProbLog

The previous section showed that k-optimal can achieve better results than k-best when approximating the success probability. In Sect. 2.3 DTProbLog is introduced and it is argued that approximating algorithms need to be used when the problem is large. We will compare the use of k-best and k-optimal for approximative inference in DTProbLog.

For this purpose the problem described by Ourfali et al. (2007) is implemented in DTProbLog. In this problem a probabilistic network is given, as well as a set of cause-effect pairs; each cause-effect pair consists of two nodes of the network and a sign ($+1$ or -1), but is not stored as an edge in the network itself. Based on the set of cause-effect pairs, signs ($+1$ or -1) have to be assigned to the edges in the network. The goal is to maximize the expected number of connected cause-effects pairs, where a path is valid only when the product of the sign of its edges is equal to the sign of the cause-effect pair.

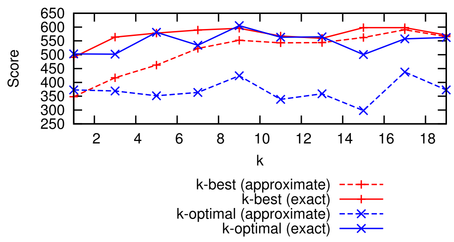
The network and cause-effect pairs are the same as those used in the previous experiments. Inference is done using k-best and k-optimal with varying k-values $(1, 3, \ldots, 19)$. For the optimization step a random restart greedy hill climbing algorithm is used with five restarts.

(Q5) *What is the effect on the utility calculated by DTProbLog?* Figure 6a shows the score obtained with k-best and k-optimal as inference algorithms with k values equal to $1, 3, \ldots, 19$. The dotted line shows the calculated score (which is a lower bound on the actual score of the solution). The full line shows the exact score of the solution; calculating the exact score is feasible for the final solution as the number of edges that gets a non-zero decision is small. In terms of the exact score, k-best achieves better results in most cases; the differences are however not significant, and in some cases k-optimal performs better. This is not the case in terms of the approximated score. An explanation for this can be found in the diversity in proofs. k-Best does not take overlap into account; as it turns out, the optimum includes a small number of decision variables, for which k-best has calculated many (overlapping) proofs. Hence, the probability calculation is based on a relatively large number of proofs. k-Optimal does take overlap into account and will diversify the used proofs during the optimization. Not all these proofs involve decision variables relevant to the optimum; hence, the calculation of the probability for the final chosen optimum is based on a smaller number of proofs. This causes the wide gap between approximated and exact score in the case of k-best. Using k-optimal enlarges the search space at the cost of worse approximations close to the optimal solution. Better approximations could be obtained by doing an additional search step which favors solutions that are close to the solution found in the first step and using this to further refine the solution.

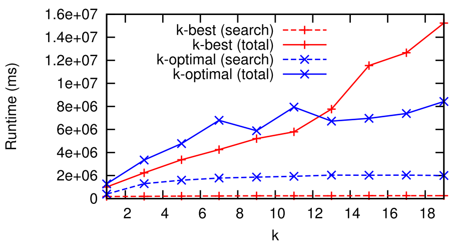
(Q6) *How does this difference in calculated utility translate to runtime performance?* Figure 6b shows the runtime as a function of the k-value for both k-optimal and k-best. The dotted lines show the search time, which is the time used to collect the proofs. The full time shows the total runtime, and includes the execution of an optimization algorithm. It is clear that k-optimal has a higher search time and for low k-values this leads to a higher total runtime for k-optimal. For higher k values the search time has a smaller effect on the total runtime and k-best needs more time.

6 Conclusions

We have introduced a new approximate inference mechanism for calculating the success probability of a query in ProbLog. This mechanism uses k proofs to approximate the ex-



(a) Score



(b) Time

Fig. 6 Results for DTProbLog using k-best and k-optimal for inference with $k = 1, 3, \ldots, 19$

act probability of a query. As k-optimal iteratively searches for proofs that increase the probability the most, it minimizes the redundancy between the selected proofs. An efficient calculation of this probability was proposed.

We compared the results of k-optimal and k-best. These experiments show that k-optimal captures a larger part of the success probability with the same number of proofs. Also relative to the size of the BDDs, BDDs that are created using k-optimal capture a bigger part of the success probability. Because of this, the run time of k-optimal is better when the probability needs to be approximated very accurately. The BDD construction time, which is proportional to the size of the BDD, is dominant in this case.

Furthermore we compared k-best and k-optimal within the context of solving decision problems in DTProbLog. These results show that although the quality of the solutions does not differ significantly, the use of k-optimal causes lower run time when using high values of k.

To complement the experimental results, a theoretical analysis has shown that the calculated probability p is $(1 - 1/e)p^* \leq p \leq p^*$, where p^* is the highest probability that can be calculated based on any set of k proofs.

We presented an extension of k-optimal which avoids adding insignificant proofs. This is easily possible in k-optimal as k-optimal computes the added probability of a proof. k-θ-Optimal produces fewer proofs, with only a small loss of probability.

In general, k-optimal can be seen as a strategy for selecting conjunctions in a formula in disjunctive normal form, such that the probability of the selected conjunctions is maximized. The use of such selection strategies in other contexts, such as other probabilistic logics, is an interesting question for future work. Furthermore, challenges remain in approximating decision problems without building data structures such as BDDs for large collections of proofs.

Acknowledgements We would like to thank Luc De Raedt for his insightful comments on this work. Joris Renkens is supported by PF-10/010 NATAR. Siegfried Nijssen and Guy Van den Broeck are supported by the Research Foundation-Flanders (FWO-Vlaanderen).

References

Bryant, R. E. (1986). Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, *35*, 677–691.

De Raedt, L., Kimmig, A., & Toivonen, H. (2007). Problog: a probabilistic prolog and its application in link discovery. In *IJCAI* (pp. 2462–2467).

De Raedt, L., Kimmig, A., Gutmann, B., Kersting, K., Costa, V. S., & Toivonen, H. (2010). *Inductive Databases and Constraint-Based Data Mining* (pp. 229–262).

Gutmann, B., Kimmig, A., De Raedt, L., & Kersting, K. (2008). Parameter learning in probabilistic databases: a least squares approach. In *ECML PKDD* (Vol. 5211/2008, pp. 473–488).

Gutmann, B., Thon, I., & De Raedt, L. (2011). Learning the parameters of probabilistic logic programs from interpretations. In *ECML PKDD* (Vol. 6911/2011, pp. 581–596).

Hazan, E., Safra, S., & Schwartz, O. (2006). On the complexity of approximating k-set packing. *Computational Complexity*, *15*, 20–39.

Ourfali, O., Shlomi, T., Ideker, T., Ruppin, E., & Sharan, R. (2007). Spine: a framework for signaling-regulatory pathway inference from cause-effect experiments. *Bioinformatics*, *23*, 359–366.

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

Van den Broeck, G., Thon, I., van Otterlo, M., & De Raedt, L. (2010). DTProbLog: a decision-theoretic probabilistic prolog. In *AAAI* (pp. 1217–1222).