Reducing Probabilistic Logic Programs

Damiano Azzolini[1] and Fabrizio Riguzzi[2]

[1] Dipartimento di Ingegneria - University of Ferrara, Via Saragat 1, I-44122, Ferrara, Italy
[2] Dipartimento di Matematica e Informatica - University of Ferrara, Via Saragat 1, I-44122, Ferrara, Italy
{damiano.azzolini,fabrizio.riguzzi}@unife.it

Abstract. The combination of the expressiveness of Probabilistic Logic Programming with the possibility of managing constraints between random variables allows users to develop simple yet powerful models to describe many real-world situations. In this paper, we propose the class of Probabilistic *Reducible* Logic Programs, in which the goal is to minimize the number of facts while preserving the validity of the constraints on the distribution induced by the program. Furthermore, we propose a practical algorithm to perform this task.

Keywords: Probabilistic Logic Programming, Statistical Relational Artificial Intelligence, Constraints

1 Introduction

In the last few years, the interest around Neural Symbolic integration shed new light on Probabilistic Logic Programming (PLP) [7,13,14]. Since the beginning of the PLP research field, that dates back to more than 25 years ago [17], several languages of increasing expressivity and flexibility have been proposed [9,18,20]. Inference in PLP usually adopts an approach called knowledge compilation [6], where the program is converted into a compact form and then operations are performed on this alternative representation.

Despite the maturity of the field, the integration between PLP and constraints on random variables' values have not yet been extensively explored: here, we propose a new class of probabilistic logic programs, namely Probabilistic *Reducible* Logic Programs, where users can mark some probabilistic facts (or even all the facts) as *reducible*, with the meaning that these facts may be removed from the program itself. The goal is to remove as many reducible facts as possible while, at the same time, preserving the validity of constraints involving random variable values.

Our work can be considered as a kind of structure learning, since we want to find the minimum subset of facts of the program. However, we do not need a background knowledge with positive and negative examples, since the search is driven by the imposed constraints.

A related idea can be found in [8], where the authors propose an algorithm to find a small (i.e., with less than a number k of clauses) ProbLog program that maximizes the likelihood of a set of positive and negative examples. It works by greedily removing one clause at the time from the whole theory, starting from the one that yields the largest likelihood when removed. In this work, we do not use a set of positive and negative examples, but a set of target probability values, and we do not restrict a priori the maximum number of clauses of the theory. Furthermore, their target is to maximize the likelihood, while our target is to minimize the number of clauses (probabilistic facts), and we also support constraints among probabilistic facts' probabilities.

The task we introduce is also different from abduction: in abduction we need to find a subset of *abducible* facts that explains a query (or, in a probabilistic scenario, that maximizes the probability of a query), but usually constraints between probability values are not considered. Similar differences can be found with the MAP (and MPE) task [4], where the goal is to find the most probable values for a subset of random variables given evidence on other variables, and with the k-best [11] and Viterbi tasks, where the goal is to find the best k explanations for a query (for Viterbi, k is set to 1).

Another related approach is the one proposed in DTProbLog [5]: given a Probabilistic Logic Program, the authors introduced the definition of *decision* facts, i.e., facts that can be selected or not, and attach utility represented by a number (that can also be negative) to a set of atoms. The goal is to find a strategy (a subset of utility facts) that maximizes the overall utility. To solve the task, the probabilistic logic program is converted into an Algebraic Decision Diagram (ADD) that is recursively traversed to compute the optimal subset of decision facts. Differently from them, we do not use ADDs and we introduce the possibility of managing constraints. Decision theory is also considered in [12], where the authors combine both Probabilistic Logic Programming and Constraint Programming, but limited to linear constraints over sum of Boolean decision variables.

The paper is structured as follows: Section 2 overviews PLP basic definitions. In Section 3 we introduce Probabilistic Reducible Logic Programs, with the Probabilistic Reducible Problem that can be solved with the algorithm proposed in Section 4 and tested in Section 5. Section 6 concludes the paper.

2 Probabilistic Logic Programming

A Probabilistic Logic Program integrates a Logic Program with probabilistic facts, i.e., logical facts that can be true with a certain probability. The meaning of these facts may not be straightforward, so several semantics have been proposed during the years: here we follow the Distribution Semantics (DS) [17] for programs with a finite Herbrand base (without function symbols). An *atomic* choice indicates whether a grounding of a probabilistic fact is selected. A set of atomic choices is *consistent* if no pair of atomic choices both selects and does not select the same probabilistic fact. If it is consistent, a set of atomic choices is a *composite* choice, and if it contains one atomic choice for every probabilistic fact it is a *selection*. Given a selection, we can identify a Logic Program called *world*, whose probability can be computed as the product of the probabilities of the atomic choices (since they are considered independent). A query q is a conjunction of ground atoms and its probability can be computed as the sum of the probabilities of the worlds entailed by it. In formula:

$$P(q) = \sum_{w \models q} P(w)$$

A composite choice identifies a set of worlds and, if a query is true in each of them, the composite choice is called *explanation*. A set of explanations is *covering* if every world in which the query is true belongs to the worlds identified by the set.

Following the ProbLog syntax [9], a probabilistic fact is denoted with:

$$\Pi :: fact.$$

where Π is a probability value (in the range $]0,1]$) and $fact$ is a term. For example,

```
0.4::tired.
```

states that `tired` is true with probability 0.4, and false with probability $1-0.4 = 0.6$. Similarly, Logic Programs with Annotated Disjunctions (LPADs) [20] allow the definition of disjunctive clauses of the form:

$$h_1 : \Pi_1; h_2 : \Pi_2; \ldots; h_n : \Pi_n : -b_1, \ldots, b_m.$$

where b_is are logical literals composing the body of the clause, h_is are logical atoms, and $\sum_i \Pi_i = 1$. If this is not the case, there is an implicit atom with annotation $1 - \sum_i \Pi_i = 1$. Intuitively, the meaning is that the head h_i is selected with probability Π_i if the body is true.

Consider now this illustrative example:

Example 1 (Toy).

```
0.8::good_product.
0.9::advertised.
0.6::receptive.

buy:- good_product.
buy:- advertised, receptive.
```

Here we have 3 probabilistic facts (`good_product`, `advertised`, and `receptive`, whose probabilities have been set to 0.8, 0.9, and 0.6 just to demonstrate the idea) and a ground predicate `buy/0` that is true if `good_product` is true or if both `advertised` and `receptive` are true. This simple program models a scenario where a user buys a product if the product is good or if the user has been advertised and he/she is receptive. The goal (query) may be to compute the probability that the user actually buys the product. Figure 1 left lists all the possible worlds. The query `buy` is true in the worlds 4 up to 8, and its probability is 0.908.

When the number of probabilistic facts of a program is significant, a table representation of the world is not feasible since it requires 2^n rows, where n is the number of ground probabilistic facts. The goal of knowledge compilation [6] is to provide efficient representations for problems with high memory or time requirements. Usually, probabilistic logic programs are transformed into a more compact representation, such as Binary Decision Diagrams (BDDs). A BDD is a direct acyclic graph where each node has only two edges, true and false, often indicated with 1 and 0. Terminal nodes can be false (0) or true (1) and have no edges. Some packages, such as CUDD [19] (which is used in this paper), allow the definition of a third type of edge, the 0-complemented: the function below this edge must be complemented. With this extension, only the 1-terminal node is needed, since the 0-terminal can be obtained with a complemented edge to 1. A representation of the program shown in Example 1 can be found in Fig. 1 right, where the 1 edge is indicated with a solid line (for instance, the one between *gp* and 1), the 0 edge is indicated with a dashed line (the one between *gp* and *ad*), and the 0-complemented edge with a dotted line (the one between *ad* and 1, for example). Nodes *gp*, *ad*, and *rc* represent respectively the probabilistic facts `good_product`, `advertised`, and `receptive` of Example 1. In case of PLP, each node represents a probabilistic fact: if we follow the 0 edge or the 0-complemented edge, the fact is false; if we follow the 1-edge, the fact is true, thus obtaining different probabilities.

From a BDD, we can get an equation representing a query. Consider again the BDD shown in Fig. 1 right: starting from the root, we can follow all the paths that arrive to the terminal node with a direct edge, and we obtain the equation $gp + (1 - gp) \cdot ad \cdot rc$. Each of these three variables can be substituted with its probability value (0.8 for gp, 0.9 for ad, and 0.6 for rc) to obtain the probability of the query `buy` (0.908). In other words, we sum all the equations induced by every path from source to the terminal, where every equation is given by the product of the encountered variables.

#	gp	ad	rc	Probability
1	F	F	F	$0.2 \cdot 0.1 \cdot 0.4 = 0.008$
2	F	F	T	$0.2 \cdot 0.1 \cdot 0.6 = 0.012$
3	F	T	F	$0.2 \cdot 0.9 \cdot 0.4 = 0.072$
4	F	T	T	$0.2 \cdot 0.9 \cdot 0.6 = 0.108$
5	T	F	F	$0.8 \cdot 0.1 \cdot 0.4 = 0.032$
6	T	F	T	$0.8 \cdot 0.1 \cdot 0.6 = 0.048$
7	T	T	F	$0.8 \cdot 0.9 \cdot 0.4 = 0.288$
8	T	T	T	$0.8 \cdot 0.9 \cdot 0.6 = 0.432$

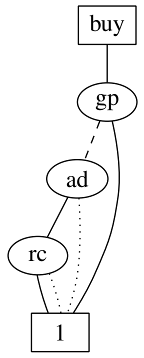

Fig. 1. Possible worlds and BDD for Example 1.

3 Probabilistic Reducible Logic Program

Here, we introduce the definition of *Probabilistic Reducible Logic Program* (PRLP) and *Probabilistic Reducible Problem*. In both, constraints are expressed by means of inequalities between terms, and we suppose, without loss of generality, that are of the form `eq > 0`, where `eq` is a possibly non-linear equation involving the probability of atoms.

Definition 1 (Probabilistic Reducible Logic Program). *Given an LPAD $\mathscr{L}$, a non-empty set of reducible facts $\mathscr{R}$, and a non-empty set of constraints $\mathscr{C}$, the tuple $(\mathscr{L},\mathscr{R},\mathscr{C})$ identifies a probabilistic reducible logic program.*

Reducible facts are denoted with the special functor `reducible` and have the following syntax:

$$\text{reducible } \Pi :: a.$$

where a is a logical term, and Π is its probability ($\Pi \in]0,1]$). To preserve the uniformity of the notation with LPADs, also the syntax reducible $a : \Pi$ is allowed.

Definition 2 (Probabilistic Reducible Problem). *Given a Probabilistic Reducible Logic Program $(\mathscr{L},\mathscr{R},\mathscr{C})$, the probabilistic reducible problem consists in finding the minimal subset of reducible facts such that the constraints in $\mathscr{C}$ are satisfied.*

The task involves discrete variables (reducible facts) and possibly non-linear functions (constraints), thus it can be classified as a mixed-integer nonlinear programming (MINLP) problem. More formally, the goal can be expressed as:

$$R^* = \underset{R \subseteq \mathscr{R}}{\arg\min}\ |R| \text{ subject to } \mathscr{C}$$

Consider the following motivating example:

Example 2 (Motivating Example - Viral Marketing, adapted from [5]). Suppose you are in a viral marketing scenario, where you target a set of people to advertise your product. The set of people can be represented as a graph, where there is uncertainty whether two people know each other or not. On a higher level, nodes can represent communities (groups of people linked by the same interests). Unfortunately, due to an economic crisis, the number of targeted people must be reduced, and you need to choose from a set of possible candidates (possibly the whole network). However, you want to maintain a lower bound on the probability that one or more people still buy the product. Overall, your final goal is to minimize the number of targeted people, to reduce as much as possible the targeting costs. A person can buy a product if she/he is directly targeted or if some trusted friend buys the product. The goal is to minimize the function that counts the number of targeted friends, not minimizing the difference between the original probability and the one obtained by removing facts. The following program may represent this scenario:

```
reducible 0.9::target(a).
reducible 0.2::target(b).
reducible 0.6::target(c).
reducible 0.7::target(d).

knows(X,Y) :- friend(X,Y).
knows(X,Y) :- friend(Y,X).
```

```
0.8::friend(a,b).
0.7::friend(b,d).
0.6::friend(a,c).
0.5::friend(c,d).

buys(X):- target(X).
buys(X):- knows(X,Y), buys(Y).
```

Probabilistic facts `friend(A,B)` represent that `A` is friend with `B` with a certain probability, and predicate `knows/2` states that `A` knows `B` if `A` is friend with `B` or `B` is friend with `A`. Reducible facts `target/1` represent the targeting of a person and they may be removed: they have an associated probability since the targeting action may fail for some reason. This value indicates the probability that the fact is true if it is not removed, not the probability that it will be removed. Predicate `buys(X)` states that `X` may buy a product if it is targeted or because it has a friend that buys the product.

As an example, with all the four reducible facts included, the probability of the query `buys(d)` is 0.920. Suppose now that you want to reduce the number of targeted people, while keeping the probability of the query above a certain threshold, say 0.9. There are four possible people to remove and 2^4 possible combinations. The optimal combination of selected people is given by

```
{target(a),target(c),target(d)}
```

(so, `target(b)` is removed from the program) and the probability of `buys(d)` is now 0.9131.

4 Algorithm

To solve the Probabilistic Reducible Problem, we extend the PITA reasoner used to perform inference in probabilistic logic programs [15] and introduce the special predicate `prob_reduce/4` with the following signature:

```
prob_reduce(TermsList,ConstraintsList,Algorithm,Result)
```

where `TermsList` is a list of terms involved in the constraints, `ConstraintsList` is a list containing one or more (possibly non-linear) constraints, `Algorithm` is the selected algorithm (exact or approximate), and `Result` is a variable that will be unified with the computed result (set of selected reducible facts). The first three variables are input variables, while the last one is an output variable.

For example, given Example 2, if we want to maintain the probability of `buys(d)` above 0.9, the query would be:

```
prob_reduce(
      [buys(d)],
      [buys(d) - 0.9 > 0],
      exact,
      Result
).
```

Here, `exact` selects the GEKKO solver [3] to compute the answer. However, other solvers can be integrated. Alternatively, we also implemented a simple greedy algorithm (that can be selected by specifying the keyword `approximate`), that iteratively removes the fact that provides the least difference in probability for all the constraints when it is removed.

Algorithm 1 illustrates the pipeline: first, we compute the equations for all the terms in `TermsList` by computing the BDDs and traversing them. All the correspondent equations obtained from the BDDs are stored in a list. Note that in program may also contain probabilistic facts (such as `friend/2` in Example 2): in this case, before traversing the BDD, we reorder the variables to have the reducible facts first in the order. This reordering is crucial, since it allows to directly call the algorithm to compute the probability from a BDD presented in [9], obtaining a more compact equation: during the traversal, once we arrive to a probabilistic node, we know for sure that there are no more nodes representing reducible facts below it, and so we can obtain a single probability value that will multiply the combinations of reducible variables represented by the nodes in the previous levels. If the BDD is not ordered with reducible facts first, this would not be possible. Furthermore, the reordering can be performed polynomially in the size of the BDD [10] and, practically, it does not affect the execution time (it is often negligible with respect to the traversing of the BDD)[1].

After that, the terms in the `ConstraintList` are replaced with the corresponding equations (line 10) and the selected solver is called. While the exact solver uses GEKKO, the approximate algorithm goes as follows: at each iteration, we compute the difference between the left part of the constraints (i.e., the part before `> 0`) with and without every reducible facts that has not been already removed (line 22, function COMPUTEDIFFERENCE). Then, we try to remove the fact that gives the least reduction (line 29, function REMOVEONE): if this is not possible, we can stop the iteration. Otherwise, once a fact has been removed by setting its probability to 0, these steps repeat until no further removal is possible. If the removal of two facts brings the same probability value, the first one according to the testing order is removed in the approximate algorithm. For the exact algorithm, this choice is directly managed by the solver.

To better understand the process, consider Example 2, represented by the BDD shown in Figure 2, where the circular nodes in the first four levels are reducible, and the remaining four are probabilistic. Overall, there are 19 paths that go to the terminal node with an even number of complemented edges. After the construction of the BDD, its equation is retrieved (omitted here for brevity). Note that, even if the program may seem small, the resulting BDD does not have a negligible size. With all reducible facts included the probability of the query `buys(d)` is 0.920. At the first iteration, the following values are the result of the function COMPUTEDIFFERENCE applied to all the four reducible facts: [0.0747, 0.0069, 0.0232, 0.1866]. The second variable gives the least reduction of the probability while maintaining the constraint true ($0.920 - 0.0069 - 0.9 > 0$), so it is removed (its probability is set to 0) by REMOVEONE. At the next iteration, the computed values are [0.0928, -, 0.0262, 0.2027] (- is a placeholder to denote a variable that has already been removed), and the probabil-

[1] Here, the BDD is reordered by iteratively swapping two adjacent variables. We do not seek the optimal BDD order, since it is an NP-hard task.

ity of the query is now 0.9131. None of the three remaining variables can be removed, since the values of the left parts of the constraints inequalities would be respectively -0.07978, -0.0131, and -0.1896, all being less than 0, so the solution shown in Example 2 is returned. In this example, the approximate algorithm also computes the optimal solution. In case there are multiple terms and multiple constraints, this description can be directly extended.

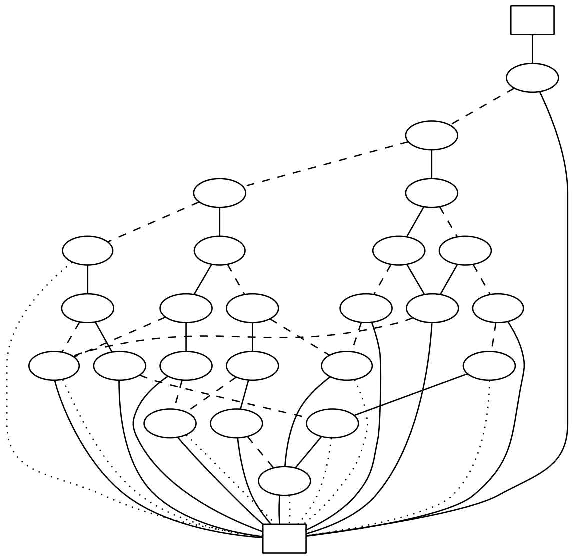

Fig. 2. BDD for queries `buys(b)` (left) and `buys(d)` (right) of Example 2.

5 Experiments

To test our algorithm, we conducted multiple experiments on datasets having a graph structure. The construction of BDDs is written in C, the implementation of the approximate solver is written in Python, and all the remaining parts are implemented using SWI-Prolog [21] version 8.3.15[2]. Experiments were conducted on a cluster[3] with Intel® Xeon® E5-2630v3 running at 2.40 GHz. The maximum execution time is 8 hours, and the maximum memory usage is set to 8GB. For GEKKO, we selected the APOPT solver and set the following options: `minlp_maximum_iterations = 100000`, `minlp_branch_method = 2`, `minlp_as_nlp = 0`, `minlp_integer_tol = 0.00005`, `minlp_gap_tol = 0.00001`, `nlp_maximum_iterations = 5000`, and `minlp_max-`

[2] Source code and datasets available at: `https://bitbucket.org/machinelearningunife/prlp_experiments`

[3] `https://www.fe.infn.it/coka/doku.php`

Algorithm 1 Function MinimizeReducibles: minimizing the number of reducible facts.

```
function MINIMIZEREDUCIBLES(termsList,constraintsList,algorithm)
    equationsList ← []
    for all term ∈ termsList do
        bdd ← COMPUTEBDD(term)
        reorderedBdd ← REORDER(bdd)
        pathsList ← COMPUTEALLPATHS(reorderedBdd)
        symbolicEquation ← CONVERTINTOSYMBOLICEQUATION(pathsList)
        equationsList ← equationsList ∪ [symbolicEquation]
    end for
    list ← REPLACEWITHSYMBOLICEQUATION(ConstraintsList,equationsList)
    if algorithm is exact then
        assignments ← SOLVEEXACT(constraintList)                ▷ Compute exact solution
    else                                                        ▷ Approximate algorithm is used
        factsList ← [term,true] for all term in termsList
        endOpt ← false
        while endOpt is false do
            gl ← []
            for all constraint in constraintsList do
                g ← []
                for all [fact,selected] in factsList do
                    if selected is true then
                        g ← g ∪ COMPUTEDIFFERENCE(fact,factsList,constraint)
                    else
                        g ← g ∪ {-1}
                    end if
                end for
                gl ← gl ∪ g                    ▷ Add the current list of facts to the total list
            end for
            index ← REMOVEONE(gl,constraintsList)
            if index == -1 then
                endOpt ← true                            ▷ No more variables can be removed
            else
                factsList[index].selected = false
            end if
        end while
        assignments ← factsList
    end if
    return assignments
end function
```

_iter_with_int_sol = 5000. Execution times are `real time` values obtained using bash command `time`.

As a first experiment, we generated *complete* graphs (KN) of increasing size to test the performance when the program has an exponentially increasing number of groundings. In these types of graphs, each node is connected with all the other nodes by an edge. The programs have the following structure:

```
buys(X):- target(X).
buys(X):- knows(X,Y), buys(Y).
```

There are two versions of this dataset: an easy one (directed graph), and a hard one (undirected graph). For the easy one, the `knows/2` predicate has the following structure (directed graph):

```
knows(X,Y):- friend(X,Y).
```

where each `target/1` is a reducible fact with an associated probability of 0.5. For the hard one, the predicate `knows/2` has an additional clause

```
knows(X,Y):- friend(Y,X).
```

allowing the relation to be bidirectional. In other words, all the programs have the structure of Example 2 but with a different number of reducible facts and different numbers of probabilistic facts. By default, we used the exact solver, and fallback to the approximate algorithm when a solution cannot be computed (this is often the case, since equations are long and complex except for smallest datasets). Figures 3 and 4 show the results. In particular, Figure 3 shows the execution time for complete graphs, both directed and undirected. The query was `buys(1) - 0.5 > 0` for all the sizes. The exact solver was used only for graphs of size less than 8 for both directed and undirected (this since, for bigger graphs, it fails to find a solution and returns an error caused by the excessive length of the equation). Figure 4 shows the gap between the lowest acceptable value for the probability of the query and the actual computed probability. For example, suppose that the constraint is `buys(1) - 0.5 > 0` and the computed value for `buys(1)` is 0.6. This gap is then 0.6 - 0.5 = 0.1. We selected values from 0.5 (as in this example) up to 0.9 with step 0.1. Note that, for the graph on the left (KN directed) with value 0.9, the gap is below 0, meaning that a solution cannot be computed, i.e., even with all the reducible facts included, the probability of the query is less than 0.9. As expected, after a certain size (9 for undirected and 15 for directed), the execution time explodes, due to the exponentially increasing number of groundings. The gap is in the order of 0.05 except for the experiment with 0.6 as value, where it is about 0.15 for both directed and undirected. This may happen due to the structure of the graph itself, which does not allow a combination of facts such that the probability can be so low.

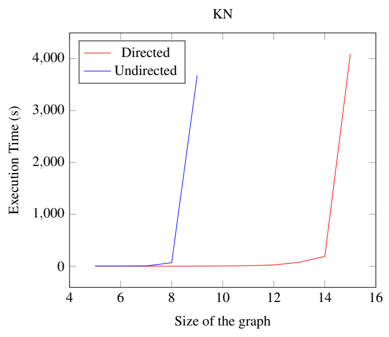

Fig. 3. Execution time for directed and undirected complete graphs.

As a second experiment, we used the dataset *probabilistic graph* taken from [4]: it consists of 10 sets of graphs, with a number of edges ranging from 50 to 500 with step 50. For each number of edges (10), there are 10 different graphs, since the generation process of this dataset is not deterministic (each graph follows a Barabási-Albert model where the number of edges to attach from a new node to existing node was set to 2). All the edges are reducible facts and have probability 0.9. However, some of them may not be involved in the computation of the probability of the query. The goal is

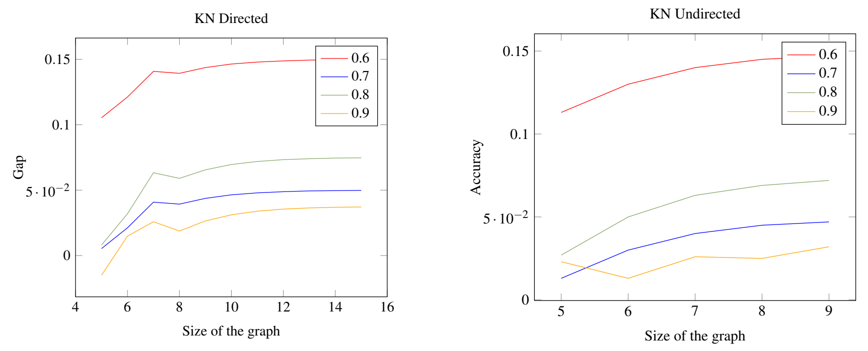

Fig. 4. Computed gaps of the approximate algorithm on both directed (left) and undirected (right) complete graphs.

to constrain the probability of the path between node index 0 and node of index size of the graph - 1, to be greater than 0.5. Table 1 shows the result of the approximate algorithm, where a dash indicates that a solution cannot be computed given time and memory constraints described above. In general, the execution time increases as the size of the graph increases, but, at the same time, the variance increases as well: since the generation of these graphs is probabilistic, nodes may be more or less connected, with possibly very complex structures.

Dataset	50	100	150	200	250	300	350	400	450	500
1	1.88	1.979	802.418	3523.055	3.121	1111.115	171.09	-	11.668	115.06
2	1.77	30.293	2.459	1249.843	7.834	4.052	6.264	7.179	301.37	-
3	4.78	115.845	3.58	14533.699	540.74	9.714	98.1	3.939	5.995	8265.052
4	1.8	2	3.823	190.697	-	103.688	30.582	192.336	145.644	16.393
5	26.33	1.756	1422.327	-	92.451	1692.265	-	9241.692	-	-
6	1.669	2.027	1.981	4308.51	174.596	423.971	28.414	250.672	-	-
7	2.7	16.654	1456.541	4.068	3.959	4.897	4.098	100.503	61.22	11.634
8	1.768	492.654	251.157	149.31	73.821	43.264	4689.642	21.82	18	17.5
9	1.705	1.772	2.131	2.431	459.622	57.278	11.003	78.667	10.95	136.639
10	2.852	2.624	88.738	3.39	6.841	330.987	419.668	92.207	-	627.594
Mean	4.72	66.76	403.51	2662.78	151.44	378.12	606.54	1109.89	79.26	1312.83
Variance	58.55	23642.07	359016.56	22501362.06	42731.49	330434.32	2362332.53	9305915.23	12070.10	9445459.51

Table 1. Execution time for probabilistic graph dataset.

To better test the exact algorithm, we selected some datasets taken from [16]: ca-netscience, power-494-bus, rt-retweet, and webkb-wisc. These programs consist of (as before) directed graphs, and the goal is to constrain the probability to reach a random destination to be greater than 0.1. We selected 10 random pairs source-destination and set the number of reducible facts to half of the total probabilistic facts. Both probabilistic and reducible facts have probability 0.9. Results are shown in Table 2. Since the datasets have a small ratio Edge / Nodes, the resulting BDD for the query is usually small (this is evident since the execution time is of the order of seconds).

Dataset	Nodes	Edges (total)	Execution Time (s)
ca-netscience	379	914	3.80
power-494-bus	494	586	1.94
rt-retweet	97	117	1.83
webkb-wisc	265	530	7.17

Table 2. Results for exact algorithm.

Overall, the exact algorithm is practical only for BDD with few nodes, since the complexity of the equation quickly increases. In general, the approximate algorithm offers a good trade-off between execution time and the gap of the computed solution. However, to better test the accuracy, the knowledge of the optimal solution (if it can be computed) is required, but it is often very difficult to have, given the complexity of the task.

6 Conclusions

In this paper, we proposed the class of Probabilistic Reducible Logic Programs with the related Probabilistic Reducible Problem. The goal is to select the smallest subset of probabilistic facts such that constraints on random variables' probabilities are not violated. Several experiments, both on synthetic and real-world datasets, show the effectiveness of our proposal. In the future, we plan to test other exact solvers, and to integrate the possibility to perform also approximate inference [1,2] to reduce the execution time.

References

1. Azzolini, D., Riguzzi, F., Lamma, E.: A semantics for hybrid probabilistic logic programs with function symbols. Artificial Intelligence **294**, 103452 (2021). https://doi.org/10.1016/j.artint.2021.103452
2. Azzolini, D., Riguzzi, F., Lamma, E., Masotti, F.: A comparison of MCMC sampling for probabilistic logic programming. In: Alviano, M., Greco, G., Scarcello, F. (eds.) Proceedings of the 18th Conference of the Italian Association for Artificial Intelligence (AI*IA2019), Rende, Italy 19-22 November 2019. Lecture Notes in Computer Science, Springer, Heidelberg, Germany (2019). https://doi.org/10.1007/978-3-030-35166-3_2
3. Beal, L., Hill, D., Martin, R., Hedengren, J.: Gekko optimization suite. Processes **6**(8), 106 (2018). https://doi.org/10.3390/pr6080106
4. Bellodi, E., Alberti, M., Riguzzi, F., Zese, R.: MAP inference for probabilistic logic programming. Theory and Practice of Logic Programming **20**(5), 641655 (2020). https://doi.org/10.1017/S1471068420000174
5. Van den Broeck, G., Thon, I., van Otterlo, M., De Raedt, L.: DTProbLog: A decision-theoretic probabilistic Prolog. In: Fox, M., Poole, D. (eds.) Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence. pp. 1217–1222. AAAI Press (2010)
6. Darwiche, A., Marquis, P.: A knowledge compilation map. Journal of Artificial Intelligence Research **17**, 229–264 (2002). https://doi.org/10.1613/jair.989

7. De Raedt, L., Frasconi, P., Kersting, K., Muggleton, S. (eds.): Probabilistic Inductive Logic Programming, LNCS, vol. 4911. Springer (2008)
8. De Raedt, L., Kersting, K., Kimmig, A., Revoredo, K., Toivonen, H.: Compressing probabilistic Prolog programs. Machine Learning **70**(2-3), 151–168 (2008). https://doi.org/10.1007/s10994-007-5030-x
9. De Raedt, L., Kimmig, A., Toivonen, H.: Problog: A probabilistic prolog and its application in link discovery. In: Veloso, M.M. (ed.) IJCAI. pp. 2462–2467 (2007)
10. Jiang, C., Babar, J., Ciardo, G., Miner, A.S., Smith, B.: Variable reordering in binary decision diagrams. In: 26th International Workshop on Logic and Synthesis. pp. 1–8 (2017)
11. Kimmig, A., Santos Costa, V., Rocha, R., Demoen, B., De Raedt, L.: On the efficient execution of ProbLog programs. In: 24th International Conference on Logic Programming (ICLP 2008). LNCS, vol. 5366, pp. 175–189. Springer (December 2008). https://doi.org/10.1007/978-3-540-89982-2_22
12. Latour, A.L.D., Babaki, B., Dries, A., Kimmig, A., Van den Broeck, G., Nijssen, S.: Combining stochastic constraint optimization and probabilistic programming. In: Beck, J.C. (ed.) Principles and Practice of Constraint Programming. pp. 495–511. Springer International Publishing, Cham (2017). https://doi.org/10.1007/978-3-319-66158-2_32
13. Manhaeve, R., Dumancic, S., Kimmig, A., Demeester, T., De Raedt, L.: Deepproblog: Neural probabilistic logic programming. In: Advances in Neural Information Processing Systems. pp. 3749–3759 (2018)
14. Riguzzi, F.: Foundations of Probabilistic Logic Programming. River Publishers, Gistrup, Denmark (2018)
15. Riguzzi, F., Swift, T.: Tabling and answer subsumption for reasoning on logic programs with annotated disjunctions. In: Technical Communications of the 26th International Conference on Logic Programming (ICLP 2010). LIPIcs, vol. 7, pp. 162–171. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2010). https://doi.org/10.4230/LIPIcs.ICLP.2010.162
16. Rossi, R.A., Ahmed, N.K.: The network data repository with interactive graph analytics and visualization. In: Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence. p. 42924293. AAAI Press (2015)
17. Sato, T.: A statistical learning method for logic programs with distribution semantics. In: Sterling, L. (ed.) Logic Programming, Proceedings of the Twelfth International Conference on Logic Programming, Tokyo, Japan, June 13-16, 1995. pp. 715–729. MIT Press (1995)
18. Sato, T., Kameya, Y.: PRISM: a language for symbolic-statistical modeling. In: 15th International Joint Conference on Artificial Intelligence (IJCAI 1997). vol. 97, pp. 1330–1339 (1997)
19. Somenzi, F.: CUDD: CU Decision Diagram Package Release 3.0.0. University of Colorado (2015)
20. Vennekens, J., Verbaeten, S., Bruynooghe, M.: Logic programs with annotated disjunctions. In: Demoen, B., Lifschitz, V. (eds.) 20th International Conference on Logic Programming (ICLP 2004). LNCS, vol. 3131, pp. 431–445. Springer (2004). https://doi.org/10.1007/978-3-540-27775-0_30
21. Wielemaker, J., Schrijvers, T., Triska, M., Lager, T.: SWI-Prolog. Theory and Practice of Logic Programming **12**(1-2), 67–96 (2012). https://doi.org/10.1017/S1471068411000494