Inference with Logic Programs with Annotated Disjunctions under the Well Founded Semantics

Fabrizio Riguzzi

ENDIF, Università di Ferrara, Via Saragat, 1, 44100 Ferrara, Italy.
`fabrizio.riguzzi@unife.it`

Abstract. Logic Programs with Annotated Disjunctions (LPADs) allow to express probabilistic information in logic programming. The semantics of an LPAD is given in terms of well founded models of the normal logic programs obtained by selecting one disjunct from each ground LPAD clause. The paper presents SLGAD resolution that computes the (conditional) probability of a ground query from an LPAD and is based on SLG resolution for normal logic programs. SLGAD is evaluated on classical benchmarks for well founded semantics inference algorithms, namely the stalemate game and the ancestor relation. SLGAD is compared with Cilog2 and SLDNFAD, an algorithm based on SLDNF, on the programs that are modularly acyclic. The results show that SLGAD deals correctly with cyclic programs and, even if it is more expensive than SLDNFAD on problems where SLDNFAD succeeds, is faster than Cilog2 when the query is true in an exponential number of instances.

Topics: Probabilistic Logic Programming, Well Founded Semantics, Logic Programs with Annotated Disjunctions, SLG resolution.

1 Introduction

The combination of logic and probability is a long standing problem in philosophy and artificial intelligence. Recently, the work on this topic has thrived leading to the proposal of novel languages that combine relational and statistical aspects. Each of these languages has a different semantics that makes it suitable for different domains.

When we are reasoning about actions and effects and we have causal independence among different causes for the same effect, Logic Programs with Annotated Disjunctions (LPADs) [1] seem particularly suitable. They extend logic programs by allowing program clauses to be disjunctive and by annotating each atom in the head with a probability. A clause can be causally interpreted in the following way: the truth of the body causes the truth of one of the atoms in the head non-deterministically chosen on the basis of the annotations. The semantics of LPADs is given in terms of the well founded model of the normal logic programs obtained by selecting one head for each disjunctive clause.

[2] showed that acyclic LPADs can be converted to Independent Choice Logic (ICL) [3] programs. Thus inference can be performed by using the Cilog2 system

[4]. An algorithm for performing inference directly with LPADs was proposed in [5]. The algorithm, that will be called SLDNFAD in the following, is an extension of SLDNF derivation and uses Binary Decision Diagrams. Both Cilog2 and SLDNFAD are complete and correct for programs for which the Clark's completion semantics and the well founded semantics coincide, as for acyclic and modularly acyclic programs [6], but can go into a loop for cyclic programs.

In this paper we present the SLGAD top-down procedure for performing inference with possibly (modularly) cyclic LPADs. SLGAD is based on the SLG procedure [7] for normal logic programs and extends it in a minimal way.

SLGAD is evaluated on classical benchmarks for well founded semantics inference algorithms, namely the stalemate game and the ancestor relation. In both cases, extensional databases encoding linear, cyclic or tree-shaped relations are considered. SLGAD is compared with Cilog2 and SLDNFAD on the modularly acyclic programs. The results show that SLGAD is able to deal with cyclic programs and, while being more expensive than SLDNFAD on problems where SLDNFAD succeeds, is faster than Cilog2 when the query is true in an exponential number of instances.

2 Preliminaries

A Logic Program with Annotated Disjunctions [1] T consists of a finite set of formulas of the form $(H_1 : \alpha_1) \vee (H_2 : \alpha_2) \vee \ldots \vee (H_n : \alpha_n) : -B_1, B_2, \ldots B_m$ called *annotated disjunctive clauses*. In such a clause the H_i are logical atoms, the B_i are logical literals and the α_i are real numbers in the interval $[0, 1]$ such that $\sum_{i=1}^{n} \alpha_i \leq 1$. The head of LPAD clauses implicitly contains an extra atom *null* that does not appear in the body of any clause and whose annotation is $1 - \sum_{i=1}^{n} \alpha_i$.

In order to define the semantics of a non-ground T, we must generate the grounding T' of T. By choosing a head atom for each ground clause of an LPAD we get a normal logic program called an *instance* of the LPAD. A probability distribution is defined over the space of instances by assuming independence among the choices made for each clause.

A *choice* κ is a set of triples (C, θ, i) where $C \in T$, θ is a substitution that grounds C and $i \in \{1, \ldots, |head(C)|\}$. (C, θ, i) means that, for ground clause $C\theta$, the head $H_i : \alpha_i$ was chosen. A choice κ is *consistent* if $(C, \theta, i) \in \kappa, (C, \theta, j) \in \kappa \Rightarrow i = j$, i.e. only one head is selected for a ground clause. A consistent choice is a *selection* σ if for each clause $C\theta$ in the grounding T' of T there is a triple (C, θ, i) in σ. We denote the set of all selections of a program T by $\mathcal{S}_T$. A consistent choice κ identifies a normal logic program $T_\kappa = \{(H_i(C) : -body(C))\theta | (C, \theta, i) \in \kappa\}$ that is called a *sub-instance* of T. If σ is a selection, T_σ is called an *instance.*

The *probability of a consistent choice* κ is the product of the probabilities of the individual choices made, i.e. $P_\kappa = \prod_{(C,\theta,i)\in\kappa} \alpha_i(C)$. The *probability of instance* T_σ is P_σ. The semantics of the instances of an LPAD is given by the well founded semantics (WFS). Given a normal program T, we call $WFM(T)$ its *well founded partial model.* For each instance T_σ, we require that $WFM(T_\sigma)$ is two-valued, since we want to model uncertainty solely by means of disjunctions. We call *sound* such a program.

The probability of a formula χ is given by the sum of the probabilities of the instances where the formula is true according to the WFS: $P_T(\chi) = \sum_{T_\sigma \models_{WFS} \chi} P_\sigma$

3 SLGAD Resolution Algorithm

In this section we present *Linear resolution with Selection function for General logic programs with Annotated Disjunctions* (SLGAD) that extends SLG resolution [8, 7] for dealing with LPADs.

SLG uses X-clauses to represent resolvents with delayed literals: an *X-clause* X is a clause of the form $A : -D|B$ where A is an atom, D is a sequence of ground negative literals and (possibly unground) atoms and B is a sequence of literals. Literals in D are called *delayed literals*. If B is empty, an X-clause is called an *X-answer* clause. An ordinary program clause is seen as a X-clause with an empty set of delayed literals.

SLG is based on the operation of SLG resolution and SLG factoring on X-clauses. In particular, SLG resolution is performed between an X-clause $A : -|A$ and a program clause or between an X-clause and an X-answer.

In SLGAD, X-clauses are replaced by XD-clauses: an *XD-clause* G is a quadruple (X, C, θ, i) where X is an X-clause, C is a clause of T, θ is a substitution for the variables of C and $i \in \{1, \ldots, |head(C)|\}$. Let X be $A : -D|B$: if B is empty, the XD-clause is called an *XD-answer* clause. With XD-clauses we keep track not only of the current resolvent but also of the clauses and head that originated it.

In SLGAD, SLG resolution between an X-clause $A : -|A$ and a program clause is replaced by SLGAD goal resolution and SLG resolution between an X-clause and an X-answer is replaced by SLGAD answer resolution. SLG factoring is replaced by SLGAD factoring.

We report here the definition for SLGAD goal resolution. Let A be a subgoal and let C be a clause of T such that A is unifiable with an atom H_i in the head of C. Let C' be a variant of C with variables renamed so that A and C' have no variables in common. We say that A is *SLGAD goal resolvable* with C and the XD-clause $((A : -|body(C'))\theta, C', \theta, i)$ is the *SLGAD goal resolvent* of A with C on head H_i, where θ is the most general unifier of A and H'_i.

SLGAD answer resolution and SLGAD factoring differ from SLG answer resolution and SLG factoring because they produce an XD-clause that contains the clause and head index of the starting XD-clause while the substitution is updated. We refer to [9] for the details of these operators.

With respect to SLG, SLGAD keeps an extra global variable that is a choice κ to record all the clauses used in the SLGAD derivation together with the head selected. This extra global variable is updated by ADD_CLAUSE that is the only procedure of SLGAD not present in SLG. ADD_CLAUSE is called when an answer for a subgoal has been found and generates different derivation branches for different choices of atoms in the head of the ground clause $C\theta$ that contains the answer in the head. ADD_CLAUSE first checks whether the clause $C\theta$ already appears in the current choice κ with a head index different from i: if so, it fails the derivation. Otherwise, it non-deterministically selects a head index j from $\{1, \ldots, |head(C)|\}$: if $j = i$ this means that the subgoal in the head is derivable in the sub-instance represented by κ, so the calling procedure can add the answer to the table. If $j \neq i$, then the table is not altered. In backtracking, all elements of $\{1, \ldots, |head(C)|\}$ are selected. Since an answer ia added to the table only when an XD-clause is reduced to an answer and eventually all XD-clauses for successful derivations will reduce to answers, it is sufficient to consider the available choices only at this point.

With this approach, SLGAD is able to exploit all the techniques used by SLG to avoid loops: the delaying of literals, the use of a global stack of subgoals, the recording of the "depth" of each subgoal and the tracking, for each subgoal A, of the deepest subgoal in the stack that may depend on A positively or negatively. For the full details of the algorithm, we refer the reader to [9].

SLGAD is sound and complete with respect to the LPAD semantics and the proof is is based on the theorem of partial correctness of SLG [8, 10]: SLG is sound and complete given an arbitrary but fixed computation rule when it does not flounder.

4 Experiments

We tested SLGAD on some synthetic problems that were used as benchmarks for SLG [7, 11]: `win`, `ranc` and `lanc`. `win` is an implementation of the stalemate game and contains the clause $win(X) : 0.8 : -move(X, Y), \neg win(Y)$. `ranc` and `lanc` model the ancestor relation with right and left recursion respectively. Various definitions of `move` are considered: a linear and acyclic relation, containing the tuples $(1, 2), \ldots, (N - 1, N)$, a linear and cyclic relation, containing the tuples $(1, 2), \ldots, (N - 1, N), (N, 1)$, and a tree relation, that represents a complete binary tree of height N, containing $2^{N+1}+1$ tuples. For `win`, all the `move` relations are used, while for `ranc` and `lanc` only the linear ones.

SLDAG was compared with Cilog2 and SLDNFAD. Cilog2 [4] computes probabilities by identifying consistent choices on which the query is true, then it makes them mutually incompatible with an iterative algorithm. SLDNFAD [5] extends SLDNF in order to store choices and computes the probability with an algorithm based on Binary Decision Diagrams. For SLGAD and SLDNFAD we used the implementations in Yap Prolog available in the `cplint` suite[1]. SLGAD code is based on the SLG system. For Cilog2 we ported the code available on the web to Yap.

The computation time of the queries `win(1)` and `ancestor(1,N)` were recorded as a function of N for `win`, `ranc` and `lanc` respectively. `win` has an exponential number of instances where the query is true and the experimental results show

[1] `http://www.ing.unife.it/software/cplint/`

the combinatorial explosion. On the ancestor datasets, the proof tree has only one branch with a number of nodes proportional to N. However, the execution time of SLGAD increases roughly as $O(N \log N)$ because each derivation step requires a lookup and an insert in the table $\mathcal{T}$ that take logarithmic time.

Cilog2 and SLDNFAD are applied only to the problems that are modularly acyclic and right recursive, i.e. `win` with linear and tree `move` and `ranc` with linear `move`, because on the other problems they would go into a loop. In `win` all the algorithms show the combinatorial explosion, with SLGAD performing better than Cilog2 and worse than SLDNFAD. On `ranc` with linear `move`, SLGAD takes longer than Cilog2 and SLDNFAD: the execution times for $N = 20,000$ are 4726.8, 8.3 and 1165.4 seconds respectively. Thus the added complexity of avoiding cycles has a computational cost. However, this cost is unavoidable when we are not sure whether the program under analysis is (modularly) acyclic or not.

Acknowledgments: This work has been partially supported by the FIRB project TOCAI.IT: *Tecnologie orientate alla conoscenza per aggregazioni di imprese in Internet.*

References

1. Vennekens, J., Verbaeten, S., Bruynooghe, M.: Logic programs with annotated disjunctions. In: International Conference on Logic Programming. Volume 3131 of LNCS., Springer (2004)
2. Vennekens, J., Verbaeten, S.: Logic programs with annotated disjunctions. Technical Report CW386, K. U. Leuven (2003) http://www.cs.kuleuven.ac.be/∼joost/techrep.ps.
3. Poole, D.: The Independent Choice Logic for modelling multiple agents under uncertainty. Artif. Intell. **94**(1–2) (1997) 7–56
4. Poole, D.: Abducing through negation as failure: stable models within the independent choice logic. J. Log. Program. **44**(1-3) (2000) 5–35
5. Riguzzi, F.: A top down interpreter for LPAD and CP-logic. In: Congress of the Italian Association for Artificial Intelligence. Volume 4733 of LNAI., Springer (2007) 109–120
6. Ross, K.A.: Modular acyclicity and tail recursion in logic programs. In: Symposium on Principles of Database Systems, ACM Press (1991) 92–101
7. Chen, W., Swift, T., Warren, D.S.: Efficient top-down computation of queries under the well-founded semantics. J. Log. Program. **24**(3) (1995) 161–199
8. Chen, W., Warren, D.S.: Query evaluation under the well founded semantics. In: Symposium on Principles of Database Systems, ACM Press (1993) 168–179
9. Riguzzi, F.: The SLGAD procedure for inference opn logic programs with annotated disjunctions. Technical Report CS-2008-01, University of Ferrara (2008) http://www.unife.it/dipartimento/ingegneria/informazione/informatica/rapporti-tecnici-1/cs-2008-01.pdf/view.
10. Chen, W., Warren, D.: Towards effective evaluation of general logic programs. Technical report, State University of New York at Stony Brook (1993)
11. Castro, L.F., Swift, T., Warren, D.S.: Suspending and resuming computations in engines for SLG evaluation. In: Practical Aspects of Declarative Languages. Volume 2257 of LNCS., Springer (2002) 332–350