Gerson Zaverucha
Vítor Santos Costa
Aline Paes (Eds.)

LNAI 8812

Inductive Logic Programming

23rd International Conference, ILP 2013
Rio de Janeiro, Brazil, August 28–30, 2013
Revised Selected Papers

Springer

Lecture Notes in Artificial Intelligence 8812

Subseries of Lecture Notes in Computer Science

More information about this series at http://www.springer.com/series/1244

Gerson Zaverucha · Vítor Santos Costa
Aline Paes (Eds.)

Inductive Logic Programming

23rd International Conference, ILP 2013
Rio de Janeiro, Brazil, August 28–30, 2013
Revised Selected Papers

Springer

Editors
Gerson Zaverucha
Federal University of Rio de Janeiro
Rio de Janeiro, Rio de Janeiro
Brazil

Aline Paes
Fluminense Federal University
Niterói, Rio de Janeiro
Brazil

Vítor Santos Costa
University of Porto
Porto
Portugal

ISSN 0302-9743 ISSN 1611-3349 (electronic)
ISBN 978-3-662-44922-6 ISBN 978-3-662-44923-3 (eBook)
DOI 10.1007/978-3-662-44923-3

Library of Congress Control Number: 2014950812

LNCS Sublibrary: SL7 – Artificial Intelligence

Springer Heidelberg New York Dordrecht London

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

Preface

This volume contains revised selected papers from ILP 2013: the 23rd International Conference on Inductive Logic Programming held during August 28–30, 2014 in Rio de Janeiro, Brazil. The ILP conference series, started in 1991, is the premier international forum on learning from structured data. Originally focusing on the induction of logic programs, it broadened its scope and attracted a lot of attention and interest in recent years. The conference now focuses on all aspects of learning in logic, multi-relational learning and data mining, statistical relational learning, graph and tree mining, relational reinforcement learning, and other forms of learning from structured data.

This edition of the conference solicited three types of submissions:

1. Long papers (12 pages) describing original mature work containing appropriate experimental evaluation and/or representing a self-contained theoretical contribution.
2. Short papers (6 pages) describing original work in progress, brief accounts of original ideas without conclusive experimental evaluation, and other relevant work of potentially high scientific interest but not yet qualifying for the above category.
3. Papers relevant to the conference topics and recently published or accepted for publication by a first-class conference such as ECML/PKDD, ICML, KDD, ICDM, etc., or journals such as MLJ, DMKD, JMLR, etc.

We received 42 submissions, 18 long, 21 short, and 3 previously published papers. The short papers were evaluated on the basis of both the submitted manuscript and the presentation at the conference. Each submission was reviewed by at least three Program Committee members. Eight long and eight short papers were accepted, the extended version of nine of these papers are included in this volume; eight further papers were accepted for inclusion in the Late-Breaking Papers volume published in the CEUR workshop proceedings series; five papers were invited for submission to the ILP 2013 open call issue of the Machine Learning journal which received a total of 20 submissions.

The subjects covered in these proceedings represent well the main topics of research in this area. The work by Muggleton et al. is a well-founded approach to statistical relation learning, with exciting practical applications. Zeng et al. address the important problem of ensuring privacy within the context of ILP. Athakravi et al. investigate learning in ASP, an area that has attracted much interest in the recent years. Binary Decision Diagrams are an effective implementation technique and Ribeiro et al. apply it to the hard problem of Interpretation Transition. Natarajan et al. discuss the difficult problem of imitation learning. Sarjant et al. contribute to reinforcement learning in the context of relational learning. Camacho et al. present innovative work in implementation of parallelism, a fundamental challenge to scaling up ILP. Statistical Relational Learning can benefit much from lifted evaluation and Taghipour et al.'s work is a step in that direction. Finally, Lisi and Straccia address an important challenge for ILP, how to learn in the Semantic Web.

The conference program included three invited talks. Prof. Jure Leskovec introduced the ongoing work on *Exploring the Structure of Online Networks and Communities*. Social interactions of hundreds of millions of people on the Web create massive digital traces, which can naturally be represented, studied, and analyzed as massive networks of interactions. By computationally analyzing such network data we can study phenomena that were once essentially invisible to us: the social interactions and collective behavior of hundreds of millions of people. In his talk he discussed how computational perspectives and mathematical models can be developed to abstract online social phenomena like: How will a community or a social network evolve in the future? What are the emerging ideas and trends in the network? How does information ow and mutate as it is passed from node to node like an epidemic?

Prof. Hendrik Blockeel discussed *Lifted Variable Elimination: Faster Correct Inference in Probabilistic Logical Models*. He started from an intriguing observation, that first-order logic allows inference on the level of variables, that is, we can reason about an object's properties without knowing the object. This boosts inference efficiency. It is not yet clear to what extent probabilistic inference can, similarly, be "lifted" to the level of logical variables. In recent years, many results have been obtained that contribute toward solving this question. A number of them were discussed in his talk, focusing on intuition rather than technical detail. He discussed how variable elimination, perhaps the simplest approach to probabilistic inference, can be lifted by identifying and exploiting particular kinds of symmetry in a probabilistic-logical model. He also discussed a number of theoretical and experimental results, both positive and negative, that provide insight into the circumstances in which lifting is (not) possible.

Prof. William W. Cohen discussed *Learning to Construct and Reason with a Large Knowledge Base of Extracted Information*. Carnegie Mellon University's "Never Ending Language Learner" (NELL) has been running for over 3 years, and has automatically extracted from the Web millions of facts concerning hundreds of thousands of entities and thousands of concepts. NELL works by coupling together many interrelated large-scale semi-supervised learning problems. In this talk, he discussed some of the technical problems the group encountered in building NELL and some of the issues involved in reasoning with this sort of large, diverse, and imperfect knowledge base. Prof. Cohen presented joint work with Tom Mitchell, Ni Lao, William Wang, and many other colleagues.

The General Chair was Gerson Zaverucha, the Program Chairs were Gerson Zaverucha and Vítor Santos Costa, and the Local Chair was Aline Paes. We thank the guest speakers for coming to ILP 2013 and for their availability during the conference. The conference was kindly sponsored by FAPERJ, the Fundação de Amparo à Pesquisa do Estado do Rio de Janeiro through grant E-26/101.541/2010. The Universidade Federal do Rio de Janeiro (UFRJ) generously supported ILP 2013 by allowing us to use the conference venue, Casa da Ciência. We thank its helpful staff: Camila Costa, Angela Monteiro, and Claudia Pereira. We also thank Maria de Fatima Cruz Marques for her valuable suggestions. Gerson Zaverucha is supported by CNPq (304399/2013-2), FAPERJ (E-26/101.541/2010) and PRONEX CNPQ - FACEPE (APQ 1188-1.03/10).

Vítor Santos Costa was supported by the grant SIBILA, NORTE-07-0124-FEDER-000059, and the FCT grants ADE, PTDC/EIA-EIA/121686/2010, and ABLe, PTDC/EEI-SII/2094/2012 (FCOMP-01-0124-FEDER-029010). Aline Paes is supported by FAPERJ (E-26/111.324/2013 APQ1) and CNPq (Universal 483448/2013-3). We acknowledge the continuous support from the Machine Learning journal through the ILP special issue, from Springer for publishing the ILP proceedings, and from CEUR for publishing the Late Breaking Papers proceedings. We thank Easychair.org for supporting submission handling. Last, but not least, we thank the Local Organizing Committee: Kate Revoredo and Fernanda Baião helped throughout in the organization, and Roosevelt Sardinha created and maintained the website.

July 2014

Gerson Zaverucha
Vítor Santos Costa
Aline Paes

Organization

General Chair

Gerson Zaverucha	COPPE – Universidade Federal do Rio de Janeiro, Brazil

Program Chairs

Gerson Zaverucha	COPPE – Universidade Federal do Rio de Janeiro, Brazil
Vítor Santos Costa	CRACS/INESC-TEC and DCC-FCUP, Portugal

Local Chair

Aline Paes	Universidade Federal Fluminense, Brazil

Local Organizing Committee

Kate Revoredo	Universidade Federal do Estado do Rio de Janeiro, Brazil
Fernanda Baião	Universidade Federal do Estado do Rio de Janeiro, Brazil
Roosevelt Sardinha	COPPE – Universidade Federal do Rio de Janeiro, Brazil

Program Committee

Erick Alphonse	LIPN – UMR CNRS 7030, France
Annalisa Appice	Università di Bari, Italy
Hendrik Blockeel	Katholieke Universiteit Leuven, Belgium
Ivan Bratko	University of Ljubljana, Slovenia
Rui Camacho	LIACC/FEUP, University of Porto, Portugal
James Cussens	University of York, UK
Luc De Raedt	Katholieke Universiteit Leuven, Belgium
Saso Dzeroski	Jozef Stefan Institute, Slovenia
Nicola Fanizzi	Università di Bari, Italy

Stefano Ferilli	Università di Bari, Italy
Peter Flach	University of Bristol, UK
Nuno Fonseca	CRACS-INESC Porto LA, Portugal and EMBL-EBI, UK
Paolo Frasconi	Università degli Studi di Firenze, Italy
Tamas Horvath	University of Bonn and Fraunhofer IAIS, Germany
Katsumi Inoue	NII, National Institute of Informatics, Japan
Nobuhiro Inuzuka	Nagoya Institute of Technology, Japan
Andreas Karwath	University of Mainz, Germany
Kristian Kersting	Technical University of Dortmund, Germany
Ross King	University of Manchester, UK
Ekaterina Komendantskaya	School of Computing, University of Dundee, UK
Stefan Kramer	University of Mainz, Germany
Nada Lavrač	Jožef Stefan Institute, Slovenia
Francesca Alessandra Lisi	Università di Bari, Italy
Donato Malerba	Università di Bari, Italy
Stephen Muggleton	Imperial College London, UK
Sriraam Natarajan	Indiana University, USA
Ramon Otero	University of A Coruña, Spain
Aline Paes	Universidade Federal Fluminense, Brazil
C. David Page	University of Wisconsin-Madison, USA
Bernhard Pfahringer	University of Waikato, New Zealand
Ganesh Ramakrishnan	IIT Bombay, India
Jan Ramon	Katholieke Universiteit Leuven, Belgium
Oliver Ray	University of Bristol, UK
Fabrizio Riguzzi	University of Ferrara, Italy
Celine Rouveirol	LIPN, Université Paris 13, France
Chiaki Sakama	Wakayama University, Japan
Claude Sammut	University of New South Wales, Australia
Jude Shavlik	University of Wisconsin-Madison, USA
Takayoshi Shoudai	Kyushu University, Japan
Ashwin Srinivasan	IBM India Research Laboratory, India
Alireza Tamaddoni-Nezhad	Imperial College London, UK
Tomoyuki Uchida	Hiroshima City University, Japan
Christel Vrain	LIFO – University of Orléans, France
Stefan Wrobel	Fraunhofer IAIS and University of Bonn, Germany
Akihiro Yamamoto	Kyoto University, Japan
Filip Zelezny	Czech Technical University in Prague, Czech Republic

Additional Reviewers

Bellodi, Elena
Heras, Jonathan
Manine, Alain-Pierre
Sato, Taisuke

Contents

MetaBayes: Bayesian Meta-Interpretative Learning Using Higher-Order Stochastic Refinement

Stephen H. Muggleton(✉), Dianhuan Lin, Jianzhong Chen, and Alireza Tamaddoni-Nezhad

Department of Computing, Imperial College London, London, UK
s.muggleton@imperial.ac.uk

Abstract. Recent papers have demonstrated that both predicate invention and the learning of recursion can be efficiently implemented by way of abduction with respect to a meta-interpreter. This paper shows how Meta-Interpretive Learning (MIL) can be extended to implement a Bayesian posterior distribution over the hypothesis space by treating the meta-interpreter as a Stochastic Logic Program. The resulting $MetaBayes$ system uses stochastic refinement to randomly sample consistent hypotheses which are used to approximate Bayes' Prediction. Most approaches to Statistical Relational Learning involve separate phases of model estimation and parameter estimation. We show how a variant of the MetaBayes approach can be used to carry out simultaneous model and parameter estimation for a new representation we refer to as a Super-imposed Logic Program (SiLPs). The implementation of this approach is referred to as $MetaBayes_{SiLP}$. SiLPs are a particular form of ProbLog program, and so the parameters can also be estimated using the more traditional EM approach employed by ProbLog. This second approach is implemented in a new system called $MilProbLog$. Experiments are conducted on learning grammars, family relations and a natural language domain. These demonstrate that $MetaBayes$ outperforms $MetaBayes_{MAP}$ in terms of predictive accuracy and also outperforms both $MilProbLog$ and $MetaBayes_{SiLP}$ on log likelihood measures. However, $MetaBayes$ incurs substantially higher running times than $MetaBayes_{MAP}$. On the other hand, $MetaBayes$ and $MetaBayes_{SiLP}$ have similar running times while both have much shorter running times than $MilProbLog$.

1 Introduction

In [19] grammars are learned using a special-purpose Prolog meta-interpreter. Hypotheses are generated along various SLD derivation paths by abduction. The approach is generalised in this paper by (1) replacing the special-purpose meta-interpreter by a general-purpose meta-interpreter which interprets user-provided meta-rules and (2) treating the meta-rules as a Stochastic Logic Program (SLP) [3,15]. In this setting we can view the hypotheses as being derived using Stochastic Refinement [23]. Figure 1 illustrates a Stochastic Refinement tree [23] for constructing a Finite State Acceptor (FSA). In this tree each path leading to

G. Zaverucha et al. (Eds.): ILP 2013, LNAI 8812, pp. 1–17, 2014.
DOI: 10.1007/978-3-662-44923-3_1

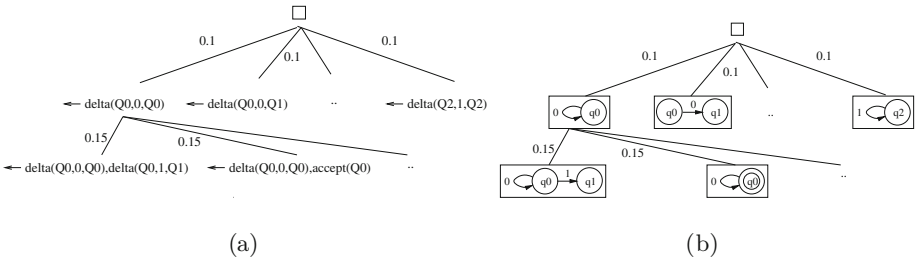


Fig. 1. Stochastic Refinement tree showing (a) clause containing arcs (delta) and acceptors, (b) corresponding Finite State Acceptors. Stochastic Refinement tree edge labels represent selection probabilities.

a hypothesis can be interpreted either (a) as a series of refinements leading to a headless Horn clause, representing the negation $\neg H$ of the ground abductive hypothesis H (see Fig. 1a) or (b) as the derivation of a finite state acceptor by a meta-interpreter applied to an SLP (Fig. 1b). Furthermore, as in [2], we can view the SLP as a structural Bayes' prior over the hypothesis space. In this case, the posterior is formed by using the positive and negative examples to prune subtrees from the prior. Following pruning, selection probabilities for each sub-tree are renormalised in the posterior.

1.1 Bayesian MIL Versus Probabilistic ILP

According to Bayesian learning theory [4,7], maximal predictive accuracy in learning is achieved by using a diversity of models, with predictions weighted according to the sum of posterior probability of the corresponding hypothesis. The implementation of such a posterior probability as a stochastic refinement graph thus provides a direct way of using hypothesis sampling approaches to approximate maximal accuracy machine learning. The relationship between Bayesian MIL (BMIL) and traditional Probabilistic ILP [22] is illustrated in Fig. 2.

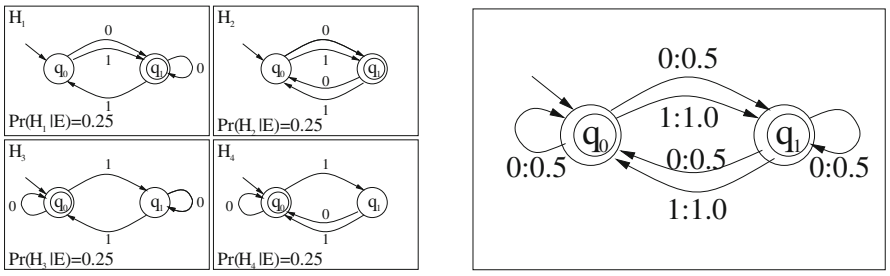


Fig. 2. (a) Finite State Acceptor hypotheses generated by BMIL from examples $e^+ = 101011011$ and $e^- = 111101$. (b) Super-imposed Logic Program formed from hypotheses in (a).

In BMIL the model consists of a set of logic programs, each with an associated probability. By contrast, in Probabilistic ILP approaches such as ProbLog [8], Bayesian Logic Programs [12] and Stochastic Logic Programs [16,17] the model consists of a single logic program with probabilistic parameters associated with individual clauses. These representations use implicit independence assumptions to support probabilistic inference. Thus, according to [22] "A ProbLog program defines a distribution over logic programs by specifying for each clause the probability that it belongs to a randomly sampled program, and these probabilities are mutually independent." Fig. 2a illustrates a uniform posterior distribution over four two-state FSA hypotheses consistent with a pair $\langle e^+, e^- \rangle$ of positive and negative examples. By contrast, Fig. 2b shows a Super-imposed Logic Program (SiLP) which can be viewed as a summary of the distribution in Fig. 2a. The SiLP is formed by labelling each arc by the sum of the posterior probabilities of hypotheses in Fig. 2a containing that arc. Note that a SiLP is a ProbLog program since the label for each clause (the arcs and acceptors) represents the probability that it belongs to a randomly sampled logic program drawn from the posterior distribution shown in Fig. 2a.

In order to show how Bayes' prediction avoids certain forms of error associated with Probabilistic ILP representations, we note that the Bayes' prediction for the negative example $e^- = 111101$ based on the distribution in Fig. 2a is zero since, by construction, no one of the FSAs accepts this string. However, the ProbLog program illustrated in Fig. 2b predicts this sequence has a probability greater than zero. The discrepancy derives from the fact that, contrary to the ProbLog assumption, the arcs in Fig. 2b are clearly not mutually independent within the FSAs in Fig. 2a.

1.2 Multiple and Single Models

By consideration of Figs. 2a and b we now compare the relative advantages of a multiple model predictor versus a single-model predictor.

Multiple models. The key advantage here is the *maximal expected predictive accuracy* offered by Bayes' prediction. It is assumed the target theory is selected randomly according to the hypothesis prior over the hypothesis space $\mathcal{H}$. Prediction that instance $x = True$ is based on the sum of posterior probabilities of all consistent hypotheses in $\mathcal{H}$ which make this prediction. This approach is infeasible in the case of $\mathcal{H}$ being a large or infinite space, though in this case it can be approximated by making predictions based on a sample of hypotheses.

Single model. This has the advantages of *increased understandability*. We can view a SiLP as providing a summarisation of the hypothesis space. In Fig. 2a we see that most (actually all) consistent hypotheses have 1-arcs from state q_0 to q_1 and q_1 to q_0. By contrast, all the 0-arcs have probability 0.5, meaning they are as likely to be true as false.

The paper is organised as follows. Section 2 describes the MetaBayes Refinement framework. The implementation of the systems MetaBayes, MetaMAP,

$\text{MetaBayes}_{\text{SiLP}}$ and MilProbLog (not to be confused with MetaProbLog [14]) are then given in Sect. 3. Experiments on binary prediction (MetaBayes vs MetaMap) and probabilistic prediction (MetaBayes vs $\text{MetaBayes}_{\text{SiLP}}$ vs MilProbLog) are conducted on various datasets, including Finite State Automata (FSAs), the ancestor relation for the Russian Royal Family and learning language semantics. In Sect. 5 we provide a comparison to related work. Lastly we conclude the paper and discuss future work in Sect. 6.

2 MetaBayes Refinement Framework

2.1 Setting

The setting for Meta-Interpretive Learning (MIL) [19] assumes as input a specialised Meta-interpreter B_M together with two sets of ground atoms representing background knowledge B_A and examples E respectively. The result of learning is a revised form of the background knowledge containing the original background knowledge B_A augmented with additional ground atoms representing a hypothesis H. We assume H is derived from $B = B_A$ and E. Applying Inverse Entailment $B, H \models E$ is equivalent to $B, \neg E \models \neg H$. In this form we see that $B, \neg E$ is given to the meta-interpreter where $\neg E$ is a goal and the resulting abduced program $\neg H$ represents a headless Horn clause such as those shown in Fig. 1a.

2.2 Generalised Meta-Interpreter

A series of specific variants of special-purpose meta-interpreters are given in [18, 19] for Regular grammars ($Metagol_R$), Context-free grammars ($Metagol_{CF}$) and a fragment of Dyadic definite clause logic ($Metagol_D$). Figure 3a shows a generalised Meta-Interpreter which can emulate each special-purpose meta-interpreter using a set of domain specific meta-rules such as those shown for finite state acceptors (Fig. 3b) [19] and the fragment of dyadic definite clauses (Fig. 3c) investigated in [18]. As discussed in [18] a meta-rule is a higher-order wff

$$\exists \mathcal{S} \forall \mathcal{T} P(s_1, .., s_m) \leftarrow .., Q_i(t_1, .., t_n), ..$$

where $\mathcal{S}, \mathcal{T}$ are disjoint sets of variables, $P, Q_i \in \mathcal{S}$ and $s_j, t_k \in \mathcal{T}$. For instance, the second finite state acceptor meta-rule in Fig. 3 indicates that with suitable higher-order ground substitution for the existentially quantified variables $\mathcal{S} = \{P, C, Q\}$ the higher-order atom $delta(P, C, Q)$ can be interpreted as the first-order clause $P([C|X], Y)$:- $Q(X, Y)$. In this way higher-order abduction of a set of atoms can be interpreted as first-order induction of a definite program.

2.3 Stochastic Refinement

According to [23] a downward *stochastic unary refinement operator* is a function $\sigma : G \rightarrow 2^{G \times [0,1]}$ defined as follows: $\sigma(C) = \{\langle D_i, p_i \rangle | D_i \in \rho(C), p_i \in [0, 1]$ and $\sum p_i = 1$ for $1 \leq i \leq |\rho(C)|\}$ and $\sigma^*(C) = \{\langle D_i, p_i \rangle | D_i \in \rho^*(C), p_i \in [0, 1]$

(a) Generalised meta-interpreter

```
prove([], Prog, Prog).
prove([Atom|As], Prog1, Prog2) :-
        metarule(RuleName, HO_Sub, (Atom :- Body), OrderTest),
        OrderTest,
        abduce(metasub(RuleName, HO_Sub), Prog1, Prog3),
        prove(Body, Prog3, Prog4),
        prove(As, Prog4, Prog2).
```

(b) Meta-rules for finite state acceptors

```
metarule(acceptor, [Q], ([Q, [], []] :- []), (term(Q))).
metarule(delta, [P, C, Q], ([P, [C|X], Y] :- [[Q, X, Y]]),
        (nonterm(Q), nonterm(P))).
```

(c) Meta-rules for dyadic fragment

```
metarule(instance, [P, X, Y], ([P, X, Y] :- []), (pred(P))).
metarule(base, [P, Q], ([P, X, Y] :- [[Q, X, Y]]),
        (pred_above(P, Q), obj_above(X, Y))).
metarule(tailrec, [P, Q], ([P, X, Y] :- [[Q, X, Z], [P, Z, Y]]),
        (pred_above(P, Q), obj_above(X, Z), obj_above(Z, Y))).
metarule(chain, [P, Q, R], ([P, X, Y] :- [[Q, X, Z], [R, Z, Y]]),
        (pred_above(P, R), obj_above(X, Z), obj_above(Z, Y))).
```

Fig. 3. Prolog representation of (a) generalised meta-interpreter, (b) Regular grammar meta-rules and (c) dyadic fragment meta-rules

and $\sum p_i = 1$ for $1 \leq i \leq |\rho^*(C)|\}$. In [23] it is shown that the n-step stochastic refinements of a clause represent a probability distribution. In the context of the meta-interpreter of Fig. 3 we can consider the refinement function ρ to consist of the selection of a consistent meta-rule followed by the related abduction of a higher-order atom[1]. Stochastic refinement with respect to a meta-interpreter involves making selections according to a probability distribution over the meta-rules.

2.4 Prior, Likelihood and Posterior

The prior of H relative to background knowledge B can now be defined as $Pr(H|B) = \sum_{\langle H,p\rangle \in \sigma^*(\neg B)} p$ and $Pr(H) = Pr(H|\emptyset)$. The likelihood of examples E with respect to the background knowledge B and hypothesis H is $Pr(E|B,H) = \begin{cases} 1 & \text{if } B, H \models E \\ 0 & \text{otherwise} \end{cases}$. Using Bayes' theorem the posterior is

$$Pr(H|B,E) = \frac{Pr(H|B)Pr(E|B,H)}{c}$$

where c is a normalisation constant. A hypothesis H is said to be MAP in the case that $H \in \text{argmax}_H Pr(H|B,E)$. A Bayes' prediction of instance x is defined by the function $\text{BayesP}(x) = \begin{cases} 1 & \text{if } \sum_H Pr(H|B,E) \geq 0.5 \\ 0 & \text{otherwise} \end{cases}$.

[1] Abduce/3 only adds a higher-order atom a to a program P to give P' when $a \notin P$.

3 Implementation

Below we describe the implementation of four systems: MetaBayes, MetaMAP, $\text{MetaBayes}_{\text{SiLP}}$ and MilProbLog. MetaBayes and MetaBayes_{MAP} are variants of the generalised meta-interpreter where stochastic refinement of the meta-rules is assumed to be conducted using a uniform distribution at each internal node of the refinement tree. $\text{MetaBayes}_{\text{SiLP}}$ is based on MetaBayes. Both $\text{MetaBayes}_{\text{SiLP}}$ and MilProbLog output probabilistic programs.

3.1 MetaBayes

This algorithm carries out an approximation to Bayes' prediction based on averaging over the posterior probabilities of a set $\mathcal{H}$ consisting of a sample of consistent hypotheses. The set is generated using a method we refer to as *Regular Sampling*, in which hypotheses are generated based on a series of fractions from the sequence $0, \frac{1}{2}, \frac{1}{4}, \frac{3}{4}, \frac{1}{8}, \frac{3}{8}, \frac{5}{8}, \frac{7}{8}, ...$ This sequence has the property of being evenly distributed in the unit interval $[0, 1]$ without repeating the same fractional value twice. Considering the consistent hypotheses to be ordered $H_1, H_2, ..$ left-to-right in SLD order within the derivation tree, the fraction p_i is used to find the rightmost H_j such that $\sum_{k=1}^{j} Pr(H_k|B, E) \leq p_i$. This is achieved efficiently by considering that the cumulative posterior probability (the sum of posterior probabilities of hypothesis preceding a given hypothesis in the derivation tree) associated with hypotheses found in the sub-trees under each node of the stochastic refinement tree is partitioned into equally sized intervals. Starting at the root of the refinement tree H_j will be found in the sub-tree whose cumulative posterior probability interval $[\min, \max]$ is such that $\min \leq p_i < \max$. Within this sub-tree we repeat by selecting the sub-tree containing the probability $(p_i - \min)(\max - \min)$. The iteration is terminated by the hypothesis returned by the base case of the meta-interpreter. By bounding the posterior probability sum of the sample and ignoring duplicates the approach can be made to achieve the effect of sampling without replacement. Although slightly more complex to program than an alternative implementation of sampling with replacement, the Regular Sampling approach achieves higher efficiency by minimising duplicate sampling due to the spread of hypotheses chosen by the sequence of fractions.

3.2 MetaBayes_{MAP}

This algorithm carries out predictions based on the leftmost consistent hypothesis at minimal depth in the stochastic refinement tree. This hypothesis can be found efficiently using iterative deepening of derivations from the generalised meta-interpreter.

3.3 MetaBayes_{SiLP}

This algorithm superimposes the set of hypothesises sampled by MetaBayes. Specifically, the posterior probabilities of sampled hypotheses are renormalised.

Then the summation is carried out for each clause C present in the set of sampled hypothesises. It follows Eq. 1, where C denotes a clause, $p(H_i|E)$ is the posterior probability of a hypothesis H_i, $p(C|H_i)$ means the probability of C being true given that H_i is true, thus $p(C|H_i)$ is either 1 or 0, depending on whether C is part of H_i. This equation is essentially the same as that of Bayesian prediction on atoms, except predicting the probabilistic labels on clauses instead of atoms.

$$p(C|E) = \sum p(C|H_i) * p(H_i|E) \tag{1}$$

3.4 MilProbLog

This algorithm is based on ProbLog but loaded with a meta-interpreter and all possible meta-substitutions, which essentially provides all possible hypothesis clauses. In this way, a learning task requiring simultaneous model and parameter estimation is reduced to only parameter estimation. If a clause is assigned with probability zero, then it implies that this clause is hypothesised as *not* part of the learned structure.

4 Experiments

In this section we describe experiments which compare MetaBayes to MetaMAP, as well as the comparison among MetaBayes, $\text{MetaBayes}_{\text{SiLP}}$ and MilProbLog.

4.1 Binary Prediction - MetaBayes vs. MetaMap

We first consider the following two Null hypothesis which compares MetaBayes to MetaMAP in terms of predictive accuracy and running time. We use datasets of learning FSAs and learning the concept of ancestor.

Null Hypothesis 1.1. MetaBayes does not have higher expected predictive accuracy than MetaMAP.

Null Hypothesis 1.2. MetaBayes does not have longer expected running time than MetaMAP.

Learning FSAs

Materials and Methods. 200 randomly chosen FSA were generated using Metagol_R [19]. Specifically, a set of sequences were randomly chosen from Σ^* for $\Sigma = \{0, 1\}$, then they are used as training examples for Metagol_R to learn FSAs. The target FSAs derived in this way are guaranteed to be minimal, since Metagol_R finds a minimal hypothesis. For each target grammar, 20 training examples were randomly chosen from Σ^* for $\Sigma = \{0, 1\}$. Another 1000 test examples were also randomly sampled without replacement. The examples are half positive and half negative, therefore the default accuracy is 50 %. Predictive accuracies and associated learning times were averaged over the 200 FSAs. We plot the learning curves at training sizes $\{2, 4, 6, 8, 10, 12, 14, 16, 18, 20\}$.

The learning systems being compared are MetaBayes and MetaMAP. The MetaMAP system makes binary prediction, while MetaBayes makes probabilistic prediction. To make them comparable, we use 0.5 as threshold to discretise the prediction by MetaBayes. Specifically, if a prediction made by MetaBayes is larger than 0.5, it is regarded as the positive, otherwise equal to 0.5 or smaller than 0.5 are considered as the negative. MetaBayes' performance varies with the size of sampled hypotheses and the given prior. Therefore we run the experiment with sample sizes as 10, 100, 500, 750 and 1000. For the prior, we considered priors which are exponential, polynomial and uniform with respect to the description length of a hypothesis. We also considered an informative prior, which is similar to the exponential prior except being given additional information about the size of a target hypothesis. The informative prior P_{inf} is defined as below, where $dl/1$ is a function which returns the description length of a hypothesis. In the case there is no sampled hypotheses having the same size as a target hypothesis, the informative prior reduces to the exponential prior.

$$P_{inf}(H) = \begin{cases} 1 & dl(H) = dl(TargetH) \\ (1/2)^{dl(H)} & dl(H) \neq dl(TargetH) \end{cases}$$

Results and Discussion. Figure 4(a) shows that MetaBayes given an informative prior has significantly higher predictive accuracies than that of MetaMAP. MetaBayes with higher sample rate 1000 also has slightly higher accuracies that of 750. However, the improvement on accuracy comes at cost of running time. As shown in Fig. 4(b), the total running time of MetaBayes with sample size 1000 is about 30 times longer than that of MetaMAP. The increase of sample rate in MetaBayes also significantly increase the running time. Therefore both Null hypothesis 1.1 and 1.2 are refuted by the experiment of learning 200 randomly chosen FSAs. Other results of MetaBayes with different sample sizes and priors which does not significantly outperform MetaMAP are not plotted due to limited space.

Learning the Concept of Ancestor

Materials and Methods. We used the family tree of Russian royal family (Romanov dynasty 1613–1917), which involves 12 generations and 119 persons. Part of the family tree is shown in Fig. 5a. This dataset has previously been used in [21]. The background knowledge contains only facts of $father/2$ and $mother/2$. The examples consist of only $ancestor/2$. No example of $parent/2$ is given, therefore to learn the target hypothesis in Fig. 5b would require not just recursion, but also predicate invention. 60 training examples with half positive and half negative are randomly chosen. They were divided into 5 folds with size 12. Results from the 5 folds were averaged. There are 1000 test examples. The default accuracy is 50 %. We plot the learning curves at training sizes $\{2, 4, 6, 8, 10, 12, 14, 16, 18, 20\}$. The rest are the same as that in learning FSAs.

Results and Discussion. Similar to the accuracy graph in Figs. 4a and 6a also show that MetaBayes given an informative prior has significantly higher predictive

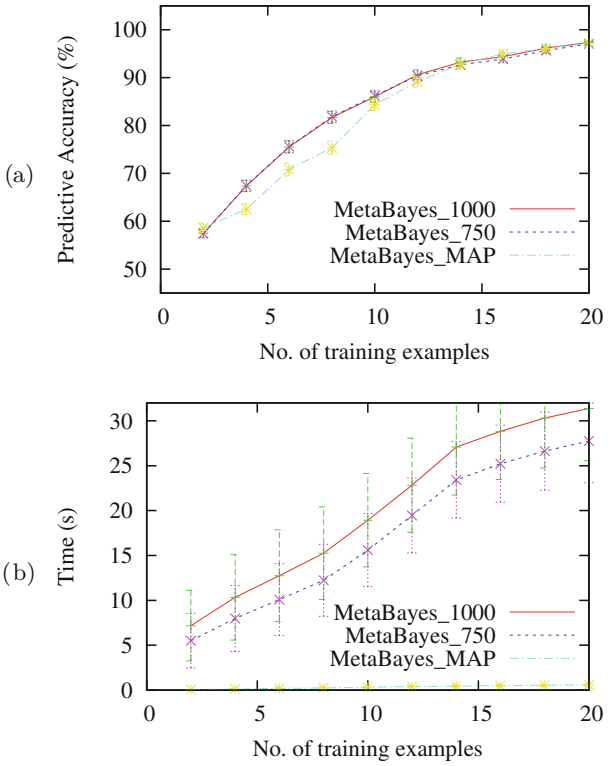
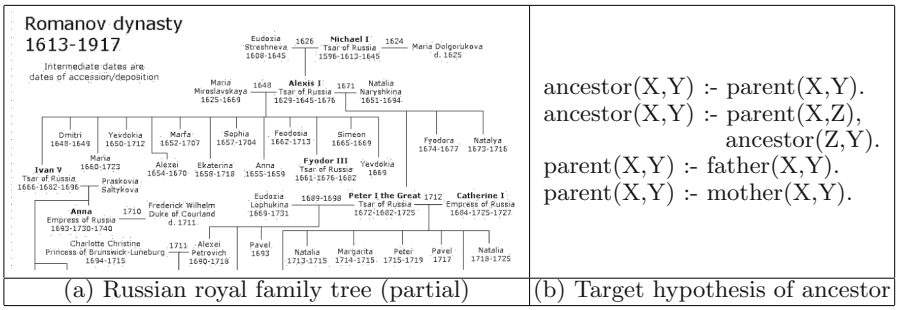


Fig. 4. Average results for learning FSAs showing (a) predictive accuracies (informative prior) and (b) running times



(a) Russian royal family tree (partial) (b) Target hypothesis of ancestor

Fig. 5. Learning the concept of ancestor

accuracies than that of MetaMAP. The running time difference between Meta Bayes and MetaMAP is even larger than that of learning FSAs, as shown in Fig. 6b. This is due to the dramatic increase in hypothesis space. In contrast, the concept of ancestor requires H_2^2 representation (Dyadic Datalog programs), which has

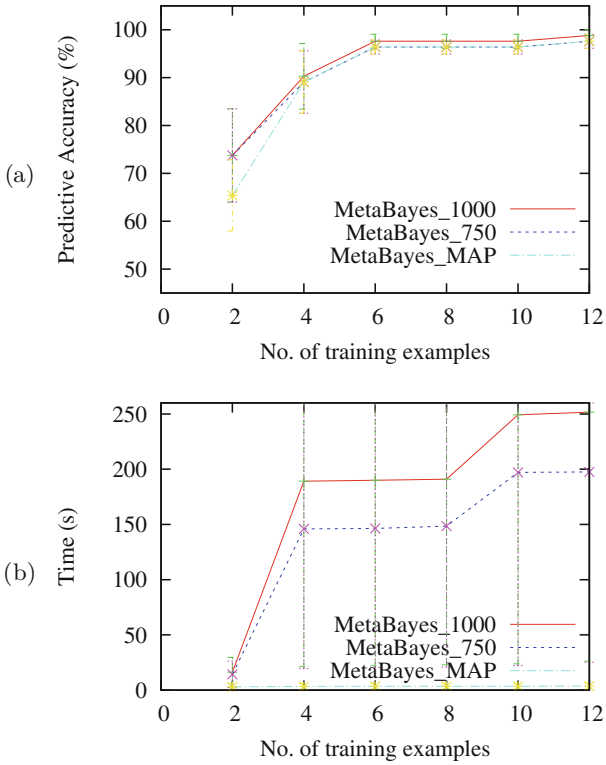


Fig. 6. Average results for learning the concept of ancestor showing (a) predictive accuracies (informative prior) and (b) running times

Universal Turing Machine expressivity [18]. Considering MetaBayes samples from the entire hypothesis space while MetaMAP only searches through part of the hypothesis space to find the shortest hypothesis, the increase of hypothesis space has larger impact on MetaBayes than MetaMAP. Therefore both Null hypothesis 1.1 and 1.2 are also refuted by the above results. The running time graph in Fig. 6b has very large deviations. Since one of the folds contains examples of ancestor involving 10 generations, which leads to significantly longer running time than the other four folds.

4.2 Probabilistic Prediction - MetaBayes vs. MetaBayes$_{\mathbf{SiLP}}$ vs. MilProbLog

In this subsection, we investigate Null hypothesis 2.1, 2.2 and 2.3 about the comparison among MetaBayes, MetaBayes$_{\mathrm{SiLP}}$ and MilProbLog. Considering all the three systems make probabilistic predictions, we use likelihood instead of accuracy as the criterion for comparison.

Null Hypothesis 2.1. MetaBayes does not have higher expected likelihood than $\text{MetaBayes}_{\text{SiLP}}$.

Null Hypothesis 2.2. MetaBayes does not have higher expected likelihood than MilProbLog.

Null Hypothesis 2.3. $\text{MetaBayes}_{\text{SiLP}}$ does not have higher likelihood than MilProbLog.

Learning FSAs

Materials and Methods. Considering MilProbLog requires the input of all possible hypothesis clauses, we have to constrain the hypothesis space to be enumerable. Therefore we considered learning FSAs with the number of maximum states N_s as 2, 3 and 4, respectively. We used regular sampling to randomly sample 100 different pairs of sequences from Σ^* for $\Sigma = \{0, 1\}$ with maximum length 10. For each pair of sequences, one is used as the positive while the other is used as the negative training examples. All possible FSAs with maximum N_s states that can be derived from the pairs are included in the set of target FSAs. For example, if $e^+ = 101011011$ and $e^- = 111101$ is one of the pairs of sampled sequences and $N_s = 2$ (only two states q_0 and q_1 are allowed), then there are four FSAs generable, as depicted in Fig. 2(a). Then all the four FSAs are part of the target FSAs. We used all the sequences with maximum length 10 as test examples, thus the size of test examples is 2047. MilProbLog was run with 10 iterations.

Results and Discussion. Figure 7 shows an example of the probabilistic programs output by $\text{MetaBayes}_{\text{SiLP}}$ and MilProbLog. Since $\text{MetaBayes}_{\text{SiLP}}$ carries out simultaneous model and parameter estimation, it does not require the provision of candidate hypothesis clauses, but only generates those candidates from examples in a data-driven fashion. That is why there are two clauses marked as *not_generated* in Fig. 7a. In contrast, MilProbLog only works when given structures. Therefore it requires all candidate clauses being provided as input, even though some of the clauses will not be used for explaining the positive examples or inconsistent with negative examples. That is why there are parameters assigned as 0 in Fig. 7b. Therefore $\text{MetaBayes}_{\text{SiLP}}$ is more efficient than MilProbLog, especially when the hypothesis space is large.

Table 1 shows the negloglikelihoods of three systems. The closer to zero the better, since *likelihood* being 1 corresponds to *loglikelihood* being 0. Therefore MetaBayes has significantly better prediction (smaller negloglikelihood) than both $\text{MetaBayes}_{\text{SiLP}}$ and MilProbLog no matter what the value N_s is. Therefore Null hypothesis 2.1 and 2.2 are refuted. This is consistent with Bayesian learning theory that Bayesian prediction being optimal.

$\text{MetaBayes}_{\text{SiLP}}$ and MilProbLog both have significantly worse prediction than MetaBayes, but $\text{MetaBayes}_{\text{SiLP}}$ has increasing better prediction than MilProbLog with the increasing of N_s. This is consistent with the fact that the increasing of N_s leads to the exponential increase of hypothesis space and enlarges the difference between the search spaces of $\text{MetaBayes}_{\text{SiLP}}$ and MilProbLog. Since $\text{MetaBayes}_{\text{SiLP}}$ is able to constrain its search space to version space, while

0.5 :: q0 →	0 :: q0 →
0.5 :: q0 → 0 q0	0 :: q0 → 0 q0
1. 0 :: q0 → 1 q1	1. 0 :: q0 → 1 q1
1.0 :: q1 → 1 q0	1.0 :: q1 → 1 q0
0.5 :: q1 → 0 q1	0.95 :: q1 → 0 q1
0.5 :: q1 →	1.0 :: q1 →
not_generated	0 :: q0 → 1 q0
not_generated	0 :: q1 → 1 q1
0.5 :: q1 → 0 q0	0.99 :: q1 → 0 q0
0.5 :: q0 → 0 q1	0.99 :: q0 → 0 q1
(a) $\text{MetaBayes}_{\text{SiLP}}$	(b) MilProbLog

Fig. 7. Probabilistic programs of learning FSA from $e^+ = 101011011$ and $e^- = 111101$.

MilProbLog has to estimate the probabilities on all candidate clauses. According to Blumer bound, the larger search space leads to higher predictive error. Therefore Null hypothesis 2.3 is refuted. In terms of running time, MilProbLog took at least 10 times longer running time than that of $\text{MetaBayes}_{\text{SiLP}}$. Such results shows that $\text{MetaBayes}_{\text{SiLP}}$ has at least competitive likelihood to that of MilProbLog while having much shorter running time.

Table 1. NegLogLikelihoods of learning FSAs

N_s	MetaBayes	$\text{MetaBayes}_{\text{SiLP}}$	MilProbLog
2	856.70±19.03	1361.43±25.84	1317.14±7.66
3	976.64±5.76	1249.72±9.65	1323.38±1.66
4	820.60±10.20	1034.30±16.70	1306.40±2.58

Learning Language Semantics. Consider a task of validating a hypothesised food web using domain literature. For example, we might want to know whether the statement 'foxes eat rabbits' is supported by a piece of text from the literature. The challenge of this task lies in the richness of natural language. Specifically, there are many different ways to convey the same meaning. For instance, the sentences 'foxes are the predator of rabbits' express the same meaning as that of 'foxes eat rabbits'. In [5] such validation task was done manually by human beings. Human beings are capable of understanding the meaning of a text and extract relevant information from the text, but it is too time consuming to read through all literature. More importantly, human beings are capable of learning the meaning of text if encounter new words or phrases.

Materials and Methods. In this experiment we consider learning semantics from texts, in particular, learning alternative phrases for expressing the same meaning. We use the representation of Definite Clause Translation Grammars (DCTG) [1] to allow both syntactic and semantic paring. Definite Clause Translation Grammars are triadic. It is similar to Definite Clause Grammars, but different in terms

```
s0(Text1,Text3,[M|Meaning]):- s1(Text1,Text2,M),s0(Text2,Text3,Meaning).
s0(Text1,Text3,[M|Meaning]):- s2(Text1,Text2,M),s0(Text2,Text3,Meaning).
s0(Text1,Text3,[M|Meaning]):- s3(Text1,Text2,M),s0(Text2,Text3,Meaning).
s0([Word|Text1],Text2,Meaning):- s0(Text1,Text2,Meaning).

s1([foxes|Text],Text,fox).
s1([fox|Text],Text,fox).
s2([predator,of|Text],Text,eats).
s2([eat|Text],Text,eats).
s3([rabbit|Text],Text,rabbit).
s3([rabbits|Text],Text,rabbit).
```

Fig. 8. Definite Clause Translation Grammars for parsing Text-Semantic pairs in Fig. 9

```
s0([foxes, are, the, predator, of, rabbits],[],[fox,eats,rabbits]).
s0([foxes, eat, rabbits],[],[fox,eats,rabbits]).
```

Fig. 9. Artificial examples of Text-Semantic pairs

of a third argument for carrying the corresponding semantic. Figure 8 gives an example of DCTG which can parse the Text-Semantic pairs given in Fig. 9. The target hypothesis of this experiment is a DCTG like the one in Fig. 8. To learn such a grammar would require learning recursion and predicate invention. An invented predicates like s_2 can be interpreted as the set of phrases for express the meaning of 'eat', while s_1 and s_3 correspond to predator and prey, respectively.

Six positive examples were gathered from real texts. Another ten negative examples were derived from positive examples by removing words like 'eat' and 'fed'. Figure 10 shows part of the examples. There are only 16 examples in total, therefore we used leave-one-out cross-validation. We constrain the candidate clauses to the phrases with maximum length 2. MilProbLog was run with 10 iterations.

```
s0([in,the,laboratory,'Pollard(1968)',found,that,agonum,dorsale,would,climb,freely,
  and,fed,on,aphids,on,the,leaves,of,brussels,sprout,plants], [], [agonum,
  dorsale,eats,aphids]).
¬s0([in,the,laboratory,'Pollard(1968)',found,that,agonum,dorsale,would,climb,freely,
  and,fed,aphids,on,the,leaves,of,brussels,sprout,plants], [], [agonum,dorsale,eats,
  aphids]).
s0(['Dicker(1951)',noted,that,the,larvae,fed,on,the,strawberry,aphid,pentatrichopus,
  fragaefolii,cocker], [], [larvae,eats,aphid]).
s0([it,therefore,seems,likely,that,although,agonum,dorsale,will,eat,a,wide,range,of,
  food,',',aphids,are,preferred], [],
  [agonum,dorsale,eats,aphids]).
```

Fig. 10. Real-world examples of Text-Semantic pairs (subset of all sixteen examples)

0.31 :: s2([fed,on|Text],Text,eats).	0.29 :: s2([fed,on|Text],Text,eats).
not_generated	0.79 :: s2([fed|Text],Text,eats).
not_generated	0.35 :: s2([on|Text],Text,eats).
0.31 :: s2([will,eat|Text],Text,eats).	0.47 :: s2([will,eat|Text],Text,eats).
(a) $\text{MetaBayes}_{\text{SiLP}}$	(b) MilProbLog

Fig. 11. Probabilistic programs of learning language semantic (partial)

Table 2. NegLogLikelihoods of learning language semantic

	MetaBayes	$\text{MetaBayes}_{\text{SiLP}}$	MilProbLog
NegLogLikelihood	0	1.58	11.71

Results and Discussion. Figure 11 shows part of the probabilistic programs generated by $\text{MetaBayes}_{\text{SiLP}}$ and MilProbLog. It compares the probabilistic labels on the same clauses. Similar to that in Fig. 7, there are clauses not considered by $\text{MetaBayes}_{\text{SiLP}}$, because they are either unnecessary for explaining the positive or inconsistent with the negative. For example, the hypothesised clause 's2([fed|Text],Text,eats)' would cover the negative example in Fig. 10.

Table 2 compares the negloglikelihood of the three systems. Similar to the previous experiment, MetaBayes has the smallest negloglikelihood among the three systems, thus MetaBayes' predictor performs best. $\text{MetaBayes}_{\text{SiLP}}$ also performs significantly better than MilProbLog, as shown in Table 2. It is worth noting that the $\text{MetaBayes}_{\text{SiLP}}$'s negloglikelihood is significantly smaller than that of MilProbLog while taking much shorter running time. Therefore all Null hypotheses 2.1, 2.2 and 2.3 are refuted.

The reason that $\text{MetaBayes}_{\text{SiLP}}$ significantly outperform MilProbLog is the same as that in the previous experiment, that is, $\text{MetaBayes}_{\text{SiLP}}$ has much smaller search space than that of MilProbLog. For example, when given example-fold1 with 15 training examples, MilProbLog has 2179 clauses to be estimated, while there are only 46 for $\text{MetaBayes}_{\text{SiLP}}$.

5 Related Work

According to Bayesian learning theory [4,7], maximal predictive accuracy in learning is achieved by using a diversity of models, with predictions weighted according to the posterior probability of the corresponding hypothesis. In [11] error bounds for various Bayesian algorithms were analysed. The paper notes that while MAP maximises probability of exact identification of the target, it may have relatively high expected error. The paper goes on to show that the Gibbs algorithm, which randomly chooses a consistent hypothesis from the posterior distribution, has an error bound which is at most twice that of a Bayes's predictor (which is known to be optimal). These theoretical results are consistent with the experiments described in Sect. 4, and are also pertinent to a number of more *ad hoc* approaches to "model averaging" which have demonstrated significant predictive accuracy increases. These approaches are usually grouped under

the title of *ensemble methods* and include *boosting* [9,13] and *bagging* [6,25]. Unlike the approach described in the present paper ensemble approaches use a more *ad hoc* approach to model-averaging, not based on an explicit Bayesian prior over the hypothesis space. However, the use of such a prior is directly comparable to the use of SLPs for sampling Bayes' nets investigated in [2]. The present paper extends this general approach to an ILP context, and demonstrates its predictive accuracy advantages in a context which supports the invention of relations and recursive programs. Within the ILP literature randomised search [20,24] has been widely investigated. However, unlike the approaches described in this paper, these searches involve heuristic step-wise optimisation, rather than sampling and averaging predictions over a posterior distribution. The related areas of Probabilistic ILP (PILP) [22] and Statistical Relation Learning (SRL) [10] involve combining Bayesian inference and ILP, though this is in the context of Probabilistic Logic representations. The treatment of a set of meta-rules as an SLP is akin to this, though the logical reasoning is necessarily in terms of higher-order clauses rather than the probabilistic first-order representations used in PILP and SRL.

6 Conclusion and Further Work

This paper extends previous work on Meta-Interpretive Learning [18,19] by demonstrating that a Bayesian prior can be implemented as a meta-interpreter over a stochastic logic program consisting of higher-order meta-rules. We use this approach to suggest a method for carrying out simultaneous structure and parameter estimation for a form of ProbLog program which we refer to as SiLPs.

The approach supports sampling of hypotheses consistent with a given set of examples and background knowledge, and has been used to implement approximated Bayes' prediction in a system called MetaBayes. Similarly we implement a Bayes' MAP algorithm together with one caled $\text{MetaBayes}_{\text{SiLP}}$ which estimates structure and parameters of SiLPs. Our experiments indicate that approximated Bayes' prediction significantly outperform MAP on binary prediction tasks involving FSAs and prediction of ancestor relationships in the Russian Royal family dataset. The results are in line with theoretical predictions. Furthermore on probabilistic prediction our MetaBayes outperforms $\text{MetaBayes}_{\text{SiLP}}$ which in turn outperforms Problog on negative log likelihood prediction on both the FSA dataset and a natural language task involving ranking of probabilistic predictions.

Further work will address efficiency improvements in the algorithms as well as extensions to handle classification noise in the data and the use of Bayesian inference in active learning.

Acknowledgements. The authors would like to acknowledge the support of Syngenta in its funding of the University Innovations Centre at Imperial College. The first author would like to thank the Royal Academy of Engineering and Syngenta for funding his present 5 years Research Chair.

References

1. Abramson, H.: Definite clause translation grammars. Technical report, Vancouver, BC, Canada, Canada (1984)
2. Angelopoulos, N., Cussens, J.: Markov chain Monte Carlo using tree-based priors on model structure. In: UAI-2001. Kaufmann, Los Altos (2001)
3. Arvanitis, A., Muggleton, S.H., Chen, J., Watanabe, H.: Abduction with stochastic logic programs based on a possible worlds semantics. In: Short Paper Proceedings of the 16th International Conference on Inductive Logic Programming. University of Corunna (2006)
4. Bernardo, J.M., Smith, A.F.M.: Bayesian Theory. Wiley, New York (1994)
5. Bohan, D.A., Caron-Lormier, G., Muggleton, S.H., Raybould, A., Tamaddoni-Nezhad, A.: Automated discovery of food webs from ecological data using logic-based machine learning. PloS ONE **6**(12), e29028 (2011)
6. Breiman, L.: Bagging predictors. Mach. Learn. **24**(2), 123–140 (1996)
7. Buntine, W.: A theory of learning classification rules. Ph.D. thesis. School of Computing Science, University of Technology, Sydney (1990)
8. De Raedt, L., Kimmig, A., Toivonen, H.: Problog: a probabilistic prolog and its applications in link discovery. In: de Mantaras, R.L., Veloso, M.M. (eds.) Proceedings of the 20th International Joint Conference on Artificial Intelligence, pp. 804–809 (2007)
9. Freund, Y., Shapire, R.: A decision theoretic generalisation of on-line learning and an application to boosting. J. Comput. Syst. Sci. **55**, 119–139 (1997)
10. Getoor, L.: Tutorial on statistical relational learning. In: Kramer, S., Pfahringer, B. (eds.) ILP 2005. LNCS (LNAI), vol. 3625, pp. 415–415. Springer, Heidelberg (2005)
11. Haussler, D., Kearns, M., Shapire, R.: Bounds on the sample complexity of Bayesian learning using information theory and the VC dimension. Mach. Learn. J. **14**(1), 83–113 (1994)
12. Kersting, K., De Raedt, L.: Towards combining inductive logic programming with Bayesian networks. In: Rouveirol, C., Sebag, M. (eds.) ILP 2001. LNCS (LNAI), vol. 2157, pp. 118–131. Springer, Heidelberg (2001)
13. Lodhi, H., Muggleton, S.H.: Modelling metabolic pathways using stochastic logic programs-based ensemble methods. In: Danos, V., Schachter, V. (eds.) CMSB 2004. LNCS (LNBI), vol. 3082, pp. 119–133. Springer, Heidelberg (2005)
14. Mantadelis, T., Janssens, G.: Nesting probabilistic inference. In: Proceedings of the International Colloquium on Implementation of Constraint and LOgic Programming Systems (CICLOPS), pp. 1–16, Lexington, Kentucky. Springer-Verlag (2011)
15. Muggleton, S.H.: Stochastic logic programs. In: de Raedt, L. (ed.) Advances in Inductive Logic Programming, pp. 254–264. IOS Press, Amsterdam (1996)
16. Muggleton, S.H.: Stochastic logic programs. J. Logic Program. (2001). Accepted subject to revision
17. Muggleton, S.H.: Learning structure and parameters of stochastic logic programs. Electron. Trans. Artif.Intell. **6** (2002)
18. Muggleton, S.H., Lin, D.: Meta-interpretive learning of higher-order dyadic datalog: predicate invention revisited. In: Proceedings of the 23rd International Joint Conference Artificial Intelligence (IJCAI 2013), pp. 1551–1557 (2013)
19. Muggleton, S.H., Lin, D., Pahlavi, N., Tamaddoni-Nezhad, A.: Meta-interpretive learning: application to grammatical inference. Mach. Learn. **94**, 25–49 (2014)

20. Muggleton, S.H., Tamaddoni-Nezhad, A.: QG/GA: a stochastic search for Progol. Mach. Learn. **70**(2–3), 123–133 (2007). doi:10.1007/s10994-007-5029-3
21. Pahlavi, N., Muggleton, S.H.: Towards efficient higher-order logic learning in a first-order datalog framework. In: Latest Advances in Inductive Logic Programming. Imperial College Press (2012) (in Press)
22. De Raedt, L., Kersting, K.: Probabilistic inductive logic programming. In: De Raedt, L., Frasconi, P., Kersting, K., Muggleton, S.H. (eds.) Probabilistic Inductive Logic Programming. LNCS (LNAI), vol. 4911, pp. 1–27. Springer, Heidelberg (2008)
23. Tamaddoni-Nezhad, A., Muggleton, S.: Stochastic refinement. In: Frasconi, P., Lisi, F.A. (eds.) ILP 2010. LNCS, vol. 6489, pp. 222–237. Springer, Heidelberg (2011)
24. Železný, F., Srinivasan, A., Page, D.L.: Lattice-search runtime distributions may be heavy-tailed. In: Matwin, S., Sammut, C. (eds.) ILP 2002. LNCS (LNAI), vol. 2583, pp. 333–345. Springer, Heidelberg (2003)
25. Zhu, J., Zou, H., Rosset, S., Hastie, T.: Multi-class adaboost. Stat. Interface **2**, 349–360 (2009)

On Differentially Private Inductive Logic Programming

Chen Zeng(✉), Eric Lantz, Jeffrey F. Naughton, and David Page

Department of Computer Sciences, University of Wisconsin-Madison, Madison, USA
{zeng,lantz,naughton,page}@cs.wisc.edu

Abstract. We consider differentially private inductive logic programming. We begin by formulating the problem of guarantee differential privacy to inductive logic programming, and then prove the theoretical difficulty of simultaneously providing good utility and good privacy in this task. While our analysis proves that in general this is very difficult, it leaves a glimmer of hope in that when the size of the training data is large or the search tree for hypotheses is "short" and "narrow," we might be able to get meaningful results. To prove our intuition, we implement a differentially private version of Aleph, and our experimental results show that our algorithm is able to produce accurate results for those two cases.

1 Introduction

Recently, concomitant with the increasing ability to collect personal data, privacy has become a major concern. In this paper, we focus on privacy issues that arise in the context of inductive logic programming (ILP).

Given an encoding of a set of examples represented as a logical database of facts, an ILP algorithm will attempt to derive a hypothesized logic program which entails all the positive and none of the negative examples. Developing efficient algorithms for ILP has been widely studied by the machine learning community [1]. However, to the best of our knowledge, a differentially private approach to ILP has received little attention.

ILP induces hypotheses from examples collected from individuals and to synthesize new knowledge from the examples. This approach naturally creates a privacy concern — how can we be confident that publishing these hypotheses and knowledge does not violate the privacy of the individuals whose data are being studied? This problem is compounded by the fact that we may not even know what data the individuals would like to protect nor what side information might be possessed by an adversary. These compounding factors are exactly the ones addressed by *differential privacy* [2], which intuitively guarantees that the presence of an individual's data in a dataset does not reveal much about that individual. Differential privacy has previously been explored in the context of other machine learning algorithms [3,4]. Accordingly, in this paper we explore

G. Zaverucha et al. (Eds.): ILP 2013, LNAI 8812, pp. 18–30, 2014.
DOI: 10.1007/978-3-662-44923-3_2

the possibility of developing differentially private ILP algorithms. Our goal is to guarantee differential privacy without obliterating the utility of the algorithm.

An obvious but important observation is that privacy is just one aspect of the problem; utility also matters. In this paper, we quantify the utility of a differentially private ILP algorithm by its likelihood to produce a sound result. Intuitively speaking, "soundness" requires an algorithm to include a hypothesis that is correct in a sufficiently large subset of the database. We start by showing the trade-off between privacy and utility in ILP. Our result unfortunately indicates that the problem is very hard — that is, in general, one cannot simultaneously guarantee high utility and a high degree of privacy.

However, a closer investigation of this negative result reveals that if we can either reduce the hypotheses space or increase the size of the input data, then perhaps there is a differentially private ILP algorithm that is able to produce a high quality result while guaranteeing privacy. To verify this, we implement a differentially private ILP algorithm and run experiments on a synthetic dataset. Our results indicate that our algorithm is able to produce results of high quality while guaranteeing differential privacy when those two conditions are met.

The rest of the paper is organized as follows: Sect. 2 briefly describes the problem of ILP, and the notion of differential privacy. Section 3 formulates the problem of guaranteeing differential privacy to ILP. Section 4 explores the trade-off between privacy and utility in ILP. Section 5 proposes our differentially private ILP algorithm, and Sect. 6 evaluates our algorithm on a synthetic dataset.

2 Preliminaries

In this section we review the problem of ILP and the notion of differential privacy.

2.1 Inductive Logic Programming

Inductive logic programming investigates the inductive construction of first-order clausal theories from examples. Let $M^+(T)$ be the minimal Herbrand model of a definite clause T. The problem of inductive logic programming is formulated in Definition 1.

Definition 1 *(Inductive logic programming [1][1]): Given two languages,*

- $\mathcal{L}_1$*: the language of database.*
- $\mathcal{L}_2$*: the language of hypotheses.*

Given a consistent set of background knowledge $B \subseteq \mathcal{L}_1$, find a hypothesis $H \in \mathcal{L}_2$, such that:

1. *Validity: $\forall h \in H$, h is true in $M^+(B)$.*
2. *Completeness: if general clause g is true in $M^+(B)$, then $H \models g$.*
3. *Minimality: there is no proper subset G of H which is valid and complete.*

[1] This formulation uses non-monotonic semantics.

In the rest of the paper, we assume both $\mathcal{L}_1$ and $\mathcal{L}_2$ are fixed unless otherwise specified. Note that in the literature of differential privacy [2], the terminology of "background knowledge" is different from the context in ILP and denotes the side information an adversary possesses to attack the privacy of a specific individual in the underlying database. To prevent that confusion, we use the term "database" to represent the background knowledge shown in Definition 1. In the rest of this paper, we use $\|D\|$ to represent the number of individuals in a database D. We also refer $\mathcal{L}_2$ as the hypotheses space.

2.2 Differential Privacy

Intuitively, *differential privacy* guarantees that the presence or absence of an individual's information has little effect on the output of an algorithm, and thus, an adversary can learn limited information about any individual. More precisely, for any database $\tau \in D$, let $nbrs(\tau)$ denote the set of *neighboring databases* of τ, each of which differs from τ by at most one *row*. *Differential privacy* requires that the probability of an algorithm to output the same result on any pair of neighboring databases are bounded by a constant ratio.

Definition 2 *(ϵ-differential privacy [2]): For any input database τ, a randomized algorithm f is ϵ-differentially private iff for any $\mathcal{S} \subseteq Range(f)$, and any database $\tau' \in nbrs(\tau)$,*

$$\Pr(f(\tau) \in S) \leq e^{\epsilon} \Pr(f(\tau') \in \mathcal{S})$$

where Pr *is the probability taken over the coin tosses of the algorithm f.*

One way to guarantee differential privacy for a count query is to perturb the correct result. In particular, Ghosh et al. [5] propose the *geometric mechanism* to guarantee ϵ-differential privacy for a single count query. The geometric mechanism adds noise Δ drawn from the two-sided geometric distribution $G(\epsilon)$ with the following probability distribution: for any integer σ,

$$\Pr(\Delta = \sigma) \sim e^{-\epsilon|\sigma|} \tag{1}$$

The geometric mechanism is a discrete variant of the Laplacian mechanism [6], which adds random noise drawn from the Laplacian distribution. To ensure differential privacy for multiple count queries, we first compute the *sensitivity* of those queries, which is the largest difference between the output of those queries on any pair of neighboring databases.

Definition 3 *(Sensitivity): Given d count queries, $\mathbf{q} = \langle q_1, \ldots, q_d \rangle$, the* sensitivity *of* $\mathbf{q}$ *is:*

$$S_{\mathbf{q}} = \max_{\forall \tau, \tau' \in nbrs(\tau)} |\mathbf{q}(\tau) - \mathbf{q}(\tau')|_1$$

Notice that the output of $\mathbf{q}$ is a vector of dimension d, and we use $|\mathbf{x} - \mathbf{y}|_p$ to denote the L_p distance between two vectors $\mathbf{x}$ and $\mathbf{y}$. The following theorem is a straightforward extension of the Laplacian mechanism to the geometric mechanism.

Theorem 1. *Given d count queries $\mathbf{q} = \langle q_1, \ldots, q_d \rangle$, for any database τ, the database access mechanism: $A_{\mathbf{q}}(\tau) = \mathbf{q}(\tau) + \langle \Delta_1, \ldots, \Delta_d \rangle$ where Δ_i is drawn i.i.d from the geometric distribution $G(\epsilon/S_{\mathbf{q}})$ (1), guarantees ϵ-differential privacy for $\mathbf{q}$.*

As proved in [6], a sequence of differentially private computations also ensures differential privacy. This is called the *composition property* of differential privacy as shown in Theorem 2.

Theorem 2. *[6] Given a sequence of computations, denoted as $\mathbf{f} = f_1, \ldots, f_d$, if each computation f_i guarantees ϵ_i-differential privacy, then $\mathbf{f}$ is $(\sum_{i=1}^{i=d} \epsilon_i)$-differentially private.*

3 Problem Formulation

In analogy to Definition 2, we formulate the problem of guaranteeing differential privacy to ILP in Definition 4.

Definition 4 *(Diff. Private ILP): An ILP algorithm f is ϵ-differentially private iff for any pair of neighboring databases[2] D_1 and D_2, for any $H \in \mathcal{L}_2$.*

$$\Pr(f(D_1) = H) \leq e^{\epsilon} \Pr(f(D_2) = H)$$

By Definition 4, the output hypothesis does not necessarily satisfy the three requirements stated in Definition 1. The reason is that by Definition 2, any differentially private algorithm must be randomized in nature.

4 Trade-Off on Privacy and Utility

Although privacy is a very important problem in ILP, utility also matters; a trivial differentially private ILP algorithm can be generated by randomly outputting a hypothesis regardless of the underlying database. Though private, this algorithm is useless in practice. Therefore, we also need to quantify the utility of a hypothesis.

[2] In the differential privacy literature, databases are typically thought of as single tables. In a multi-relational setting, the proper definition of "neighboring" might change. For example, in a medical domain a neighboring database would remove one patient along with their respective prescriptions and diagnoses from other tables.

4.1 Our Utility Model

Our intuition for the utility model is to relax the requirements on hypotheses in Definition 1. In particular, we relax both the *validity* and *completeness* requirement, and introduce the notion of δ-*usefulness* ($0 \leq \delta \leq 1$).

Definition 5 *(δ-usefulness): A hypothesis H is* δ-useful *for the input database D iff $\exists D' \subseteq D$, and $\|D'\|/\|D\| \geq \delta$ such that*

1. *Approximate validity: $\forall h \in H$, $h \in M^+(D')$.*
2. *Approximate completeness: if a general clause g is true in M^+D', then $H \models g$.*
3. *Minimality: there is no subset of H which is validate and complete in D'.*

The notion of δ-usefulness quantifies the quality of a hypothesis in terms of the percentage of input database in which that hypothesis is correct. Furthermore, we define the quality of a differentially private ILP algorithm by its likelihood η to produce hypotheses of high quality. This is shown in Definition 6.

Definition 6 *((δ, η)-approximation): An ILP algorithm f is* (δ, η)-approximate *iff for any input database D,*

$$\Pr(f(D) \text{ is } \delta\text{-useful}) \geq 1 - \eta$$

Both δ and η should be within the range of $(0, 1)$ by definition. Another way to understand the notion of (δ, η)-approximation is through the idea of PAC-learning [7] and define the notion of "approximate correctness" in terms of δ-usefulness. Next, we will quantify the trade-off between privacy and utility in ILP.

4.2 A Lower Bound on Privacy Parameter

Our techniques to prove the lower bound on the privacy parameter come from differentially private itemset mining [8]. Perhaps this is no surprise since both frequent itemset mining and association rule mining have been closely connected with the context of ILP [9] in which frequent itemset mining can be encoded as a ILP problem. We prove the lower bound on the privacy parameter ϵ if an ILP algorithm must be both ϵ-differentially private and (δ, η)-useful. This is shown in Theorem 3.

Theorem 3. *For any ILP algorithm that is both ϵ-differentially private and (δ, η)-useful,*

$$\epsilon \geq \frac{\ln(\|2^n\|)(1-\eta)}{2((1-\delta)\|D\| + 1)}$$

where n is the number of atoms in the language of hypotheses $\mathcal{L}_2$.

Proof. We model the language of the input database $\mathcal{L}_1$ as follows: each atom is taken from the set $I = \{a_1, \ldots, a_n\}$, and each individual's data is represented by a conjunctive clause of the atoms. We also model the language of the hypotheses $\mathcal{L}_2$ to be all the possible conjunctive clauses over the set of atoms I.

Suppose f is an ILP algorithm that is both ϵ-differentially private and (δ, η)-useful. To better understand our proof technique, we add another atom a_{n+1} to I, and then we construct an input database D of size m by including $\delta * m$ clauses of the form $h_1 = a_1 \wedge a_2 \wedge \ldots \wedge a_n \wedge a_{n+1}$. The rest are constructed as simply $h_2 = a_{n+1}$. Since the number of all the hypotheses including a_{n+1} is 2^n, there must exist a particular hypothesis h_3 such that

$$\Pr(f(B) = h_3) \leq \frac{1}{2^n}$$

Without loss of generality, let $h_3 = a_1 \wedge a_2 \wedge \ldots a_k \wedge a_{n+1}$. Then, we construct another database D' from D by replacing one clause of h_1 by h_3, and then every clause of h_2 by h_3. Thus, there is a total number of $\delta m - 1$ clauses of h_1 in B' and the rest of them being h_3. It is not hard to show that h_3 is the only δ-useful hypothesis in B: any subset of B of cardinality δm must contain at least one h_3, and thus, the δ-valid hypotheses are those that can be entailed by h_3. Hence,

$$\Pr(f(D') = h_3) \geq 1 - \eta$$

Since D' and D differ by $2((1-\delta)m + 1)$ rows (one can think of this difference as the edit distance between two databases), by differential privacy,

$$1 - \eta \leq \frac{e^{\epsilon(2((1-\delta)m+1))}}{2^n}$$

Theorem 3 then follows.

The result of Theorem 3 is similar in flavor to [10], which proved that there is no differentially private algorithm that is able to answer $O(n^2)$ count queries in a database of size n with reasonable accuracy. That is, if an ILP algorithm can be thought of as a sequence of count queries, and if the number of count queries exceeds a certain threshold, then the ILP algorithm cannot produce a result of high quality.

This is a discouraging result, which states that in general, it is very hard to simultaneously guarantee both differential privacy and a high utility requirement since $\|\mathcal{L}_2\|$ grows exponentially with the number of atoms. Theorem 3 suggests that in order to decrease the lower bound on the privacy parameter, we must either increase the size of the database $\|D\|$, or reduce the number of atoms in the hypotheses space $\mathcal{L}_2$. If a real world problem meets those two conditions, we might be able to get results of high quality while guaranteeing differential privacy. To verify this, we propose a differentially private ILP algorithm.

5 Differentially Private ILP Algorithm

In this section, we will first briefly describe a typical non-private ILP algorithm, inverse entailment as implemented in Aleph [11], and then show our revisions

of the non-private algorithm to guarantee differential privacy. In the rest of this section, we follow the terminologies used in Aleph in which an atom is also called a "predicate."

5.1 A Non-private ILP Algorithm

The non-private ILP algorithm works as follows:

1. Select an example (selection): Select an example in the database to be generalized.
2. Build most-specific-clause (saturation [12]): Construct the most specific clause that entails the example selected, and is within language restrictions provided. This is usually a definite clause with many literals, and is called the "bottom clause."
3. Search (reduction): Find a clause more general than the bottom clause. This is done by searching for some subset of the predicates in the bottom clause that has the "best" score.
4. Remove redundant (cover removal): The clause with the best score is added to the current hypothesis, and all examples made redundant are removed.

A careful analysis of the above steps shows that the selection and reduction steps directly utilize the input data while the saturation and cover removal steps depend on the output from the previous step. Thus, as discussed in literature [2], as long as we can guarantee the output from both selection and reduction is differentially private, then it is safe to utilize those output in subsequent computation. Hence, we only need to consider the problem of guaranteeing differential privacy for those two steps.

The input to the learning algorithm consists of a *target predicate*, which appears in the head of hypothesized clauses. The input database can be divided into two parts: the set of positive examples $E^+ \subseteq D$ which satisfy the target predicate, and the set of negative examples $E^- \subseteq D$ which do not. Furthermore, the bottom clause is normally expressed as the conjunctive form of the predicates, and thus we also use a "subset of the predicates" to denote the clause that is of the conjunctive form of the predicates in that subset.

5.2 A Differentially Private Selection Algorithm

The non-private selection algorithm is a sampling algorithm that randomly selects an individual's data to generalize. However, as discussed in [6], no sampling algorithm is differentially private. In this paper, we propose to use domain knowledge to overcome this obstacle. That is, we utilize the domain knowledge to generate a "fake" example. We want to emphasize that the domain information might come from external knowledge or previous interactions with the database. This information does not weaken the definition of differential privacy as stated in Definition 2, and we only utilize these previous known information to generate a fake example. In the worst case, this example can be expressed as the conjunction of all the predicates, which is considered as the public information. In that

way, the new selection step hardly relies on the input database, and thus, it is differentially private[3].

5.3 A Differentially Private Reduction Algorithm

The non-private reduction algorithm actually consists of two steps: (1) the heuristic method to search for a clause, which is a subset of predicates in the bottom clause, and (2) the scoring function to evaluate the quality of a clause. Although there are many different methods to implement the reduction algorithm [1], in this paper we follow the standard usage in Aleph in which the heuristic method is a top-down breadth-first search and the scoring function is coverage (the number of covered positive examples minus the number of covered negative examples). The search starts from the empty set and proceeds by the increasing order of the cardinality of the clauses until a satisfying clause is found. The pseudocode of the non-private reduction algorithm is shown in Algorithm 1.

Algorithm 1. Non-Private Reduction

Input: positive evidence E^+; negative evidence E^-; bottom clause H_b
Output:the best clause

1: k = number of predicates in H_b
2: $\mathcal{L}$ = the lattice on the subset of predicates in H_b
3: **for** $i = 1$ to k **do**
4: **for** each set $S \in \mathcal{L}$, $\|S\| = i$ **do**
5: P = the number of positive examples satisfying S
6: N = the number of negative examples satisfying S
7: $H^* = S$ if S has better coverage than the previously best clause
8: **end for**
9: **end for**
10: **return** H^*

A Naïve Differentially Private Algorithm. We observe that the only part in Algorithm 1 that needs to inquire the input database is in the computation of P and N shown in line 5 and line 6, respectively. Therefore, as long as we can guarantee differential privacy to those two computations, then the reduction algorithm is differentially private. We do so by utilizing the geometric mechanism. Given a clause h, let q_h^+ and q_h^- be the queries that compute the number of positive examples satisfying h and the number of negative examples satisfying h, respectively. Then, as shown in Theorem 4, the sensitivity to evaluate a set of clauses is equal to the number of clauses in the set.

[3] An alternative is to relax the privacy definition from ϵ-differential privacy to (ϵ, δ)-differential privacy. In this context, δ (unlike the symbol's use in Sect. 4) refers to the probability that the algorithm violates the ϵ-differential privacy guarantee. We do not explore it here as it makes the already burdensome utility bounds much worse.

Theorem 4. *Given a set of clauses* $H = \{h_1, h_2, \ldots, h_n\}$*, and the corresponding evaluation queries* $\mathbf{q} = \{q^+_{h_1}, q^-_{h_1}, \ldots, q^+_{h_n}, q^-_{h_n}\}$*, the sensitivity of* $\mathbf{q}$ *is* n*.*

Proof. When we add an example, it either satisfies the clause or not, and is either positive or negative, therefore the sensitivity of $\mathbf{q}$ is at most n. Without loss of generality, suppose we add a positive example satisfying the clause. By adding an individual whose data is exactly the bottom clause, the result of every $q^+_{h_i}$ increases by one. The theorem then follows.

We show our differentially private reduction algorithm in Algorithm 2.

Algorithm 2. Diff. Private Reduction

Input: positive evidence E^+; negative evidence E^-; bottom clause H_b; privacy parameter ϵ
Output:the best clause

1: k = number of predicates in H_b
2: $\mathcal{L}$ = build a lattice on the subset of predicates in H_b
3: **for** $i = 1$ to k **do**
4: **for** each subset $h \in \mathcal{L}$, $\|S\| = i$ **do**
5: $P' = q^+_h(E^+) + G(\epsilon/\|\mathcal{L}\|)$
6: $N' = q^+_h(E^-) + G(\epsilon/\|\mathcal{L}\|)$
7: $H^* = S$ if S has better coverage than the previously best clause w.r.t. P' and N'
8: **end for**
9: **end for**
10: **return** H^*

By Theorem 4, we can prove Algorithm 2 is differentially private. This is shown in Theorem 5.

Theorem 5. *Algorithm 2 is* ϵ*-differentially private.*

The Smart Differentially Private Reduction Algorithm. We observe that Algorithm 2 has only considered the worst-case scenario in which the number of clauses to be evaluated is the whole lattice whereas in practice, the reduction algorithm seldom goes through every clause in the lattice. This occurs when criteria are set to specify unproductive clauses for pruning (preventing evaluation of supersets) or for stopping the algorithm. Thus, the number of clauses evaluated in practice is much less than that in the whole lattice, which means the scale of the noise added is larger than necessary. If the quality of a clause does not meet certain criterion, then there is no need to evaluate the subtree in the lattice rooted at that clause. This algorithm is shown in 3.

We have also introduced another parameter ℓ in Algorithm 3 to specify the maximal cardinality of the desired clause, which also helps to reduce the number of clauses to be evaluated. We prove Algorithm 3 is differentially private in Theorem 6.

Algorithm 3. SMART_DIFF_PRIVATE_REDUCTION

Input: positive evidence E^+; negative evidence E^-; bottom clause H_b; privacy parameter ϵ; levels ℓ
Output:the best clause

$\mathcal{L}$ = build a lattice on the subset of predicates in H_b
for $i = 0$ to ℓ **do**
  β = the number of clauses with k predicates existing in the lattice
  for each subset $h \in \mathcal{L}$, $\|S\| = i$ **do**
    $P' = q_h^+(E^+) + G(\frac{\epsilon}{\beta(\ell+1)})$
    $N' = q_h^+(E^-) + G(\frac{\epsilon}{\beta(\ell+1)})$
    $H^* = S$ if S has better coverage than the previously best clause
    if P' and N' does not meet the criterion **then**
      Delete the subtrees in the lattice rooted at S
    end if
  end for
end for
return H^*

Theorem 6. *Algorithm 3 is ϵ-differentially private.*

Proof. By Theorem 4, the computation for each level is $(\epsilon/(\ell+1)$-differentially private, and since there is at most $\ell + 1$ levels to compute, by the composition property of differential privacy in Theorems 2 and 6 then follows.

5.4 Our Differentially Private ILP Algorithm

By using our differentially private selection algorithm and the smart differentially private reduction algorithm, we present our differentially private ILP algorithm in Algorithm 4. Since the output might consist of multiple clauses, we add the input parameter k which specifies the maximal number of clauses in the theory. We understand that this is not a usual setting for the usage of Aleph. However, in order to guarantee differential privacy, the most convenient way is to utilize the composition property which needs to know the number of computations in advance. We leave the problem to overcome this obstacle for future work.

By the composition property of differential privacy, we can prove that Algorithm 4 is differentially private.

Theorem 7. *Algorithm 4 is ϵ-differentially private.*

6 Experiment

In our experiments, we run our differentially private algorithm on synthetic data generated by the train generator described in [12], in which the goal is to discriminate eastbound versus westbound trains based on the properties of

Algorithm 4. Diff. Private ILP Algorithm

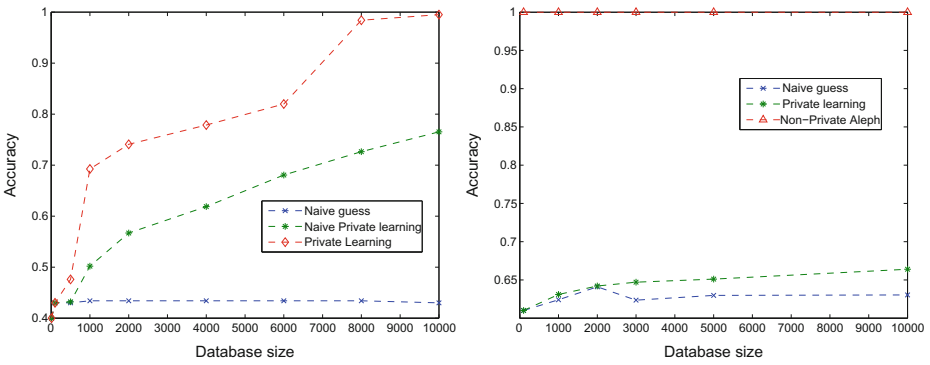
Input: positive evidence E^+; negative evidence E^-; privacy parameter ϵ; levels ℓ; rounds k

Output:the best theory

```
T = ∅
for i = 1 to k do
    H_b = Select a bottom clause in a differentially private way
    h = Smart_Diff_Private_Reduction(E+, E−, ϵ/k, ℓ)
    Add h to T
    Remove redundant examples using h
end for
return T
```

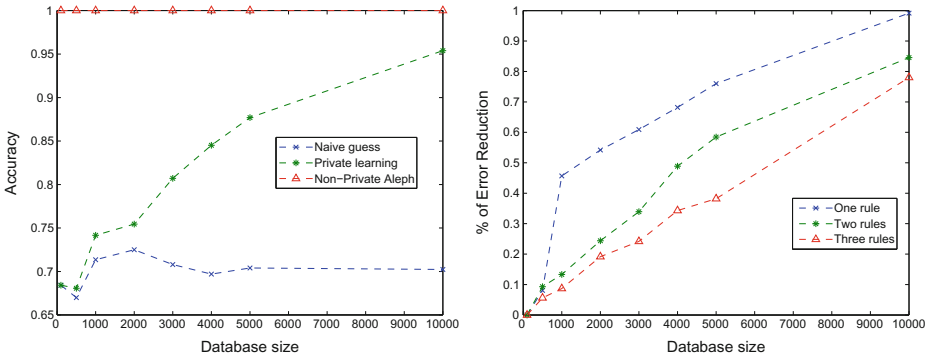
their railcars. In all the experiments, we set the privacy parameter ϵ to be 1.0, and vary both the size of the data and the desired hypothesis to see how our algorithm performs. We measure the quality of our algorithm in terms of the accuracy of the output theory on a testing set. In all of our experiments, the naïve guess is the clause that assumes every train is eastbound.

In our first experiment, we only consider a hypothesis of one clause. We vary the number of predicates in the clause, and the results are shown in Fig. 1. As we can see in Fig. 1a, when there are three predicates in the desired clause, our private learning algorithm is able to learn the clause more accurately with more training data as discussed in Sect. 4. Furthermore, we also observe that our smart reduction algorithm shown in Algorithm 3 produces better results than the naïve reduction algorithm, demonstrating that reducing added noise by pruning low scoring clauses produces more accurate results. However, when increasing the number of predicates in the desired clause, the quality of our algorithm



(a) Three Atoms (b) Six Atoms

Fig. 1. One clause with different number of predicates



(a) Two Clauses (b) Multiple Clauses

Fig. 2. Multiple clauses with the same number of predicates

decreases. This is no surprise as the hypotheses space grows exponentially with the number of the predicates.

We also investigate the effects on the number of clauses in a desired hypothesis, each of which consists of three predicates. As we can see, in Fig. 2a, our private learning algorithm is able to produce high quality hypothesis with the growth in the size of the data, which is significantly better than the case of a single clause with six predicates. This is because the addition of a clause only increases the hypotheses space multiplicatively instead of exponentially. In both Figs. 1 and 2a we see that a large performance penalty is paid by the differentially private algorithms, as the non-private algorithm achieves perfect accuracy in all cases. Figure 2b shows the percentage of error reduction as more clauses need to be learned, showing the penalty due to the privacy budget being split among multiple clauses.

7 Conclusion

In this paper, we have proposed a differentially private ILP algorithm. We have precisely quantified the trade-off between privacy and utility in ILP, and our results indicate that in order to satisfy a non-trivial utility requirement, an ILP algorithm incurs a huge risk of privacy breach. However, we find that when limiting the hypotheses space and increasing the size of the input data, our algorithm is able to output a hypothesis with high quality on synthetic data set. To the best of our knowledge, ours is the first one to attack this problem. With the availability of security-sensitive data such as electronic health records, we hope more and more people begin to pay attention to the privacy issues arising in the context of ILP.

There are many potential opportunities for future work. One such direction would be to formalize the notion of differential privacy with first-order logic, and discuss the tradeoff between privacy and utility in that context. Furthermore,

we have only considered ILP with definite clauses, and it would be interesting to expand our work to statistical relational learning [13]. Finally, since our algorithm requires one to limit the hypotheses space, it would also be interesting to investigate the feature selection problem in the context of differential privacy.

References

1. Muggleton, S., de Raedt, L.: Inductive logic programming: theory and methods. J. Logic Program. **19**, 629–679 (1994)
2. Dwork, C.: Differential privacy. In: Bugliesi, M., Preneel, B., Sassone, V., Wegener, I. (eds.) ICALP 2006. LNCS, vol. 4052, pp. 1–12. Springer, Heidelberg (2006)
3. Williams, O., McSherry, F.: Probabilistic inference and differential privacy. In: Advances in Neural Information Processing Systems, vol. 23 (2010)
4. Rubinstein, B.I.P., Bartlett, P.L., Huang, L., Taft, N.: Learning in a large function space: privacy-preserving mechanisms for SVM learning. CoRR (2009)
5. Ghosh, A., Roughgarden, T., Sundararajan, M.: Universally utility-maximizing privacy mechanisms. In: STOC (2009)
6. Dwork, C., Mcsherry, F., Nissim, K., Smith, A.: Calibrating noise to sensitivity in private data analysis. In: TCS (2006)
7. Valiant, L.G.: A theory of the learnable. Commun. ACM **27**(11), 1134–1142 (1984)
8. Zeng, C., Naughton, J.F., Cai, J.Y.: On differentially private frequent itemset mining. Proc. VLDB Endow. **6**(1), 25–36 (2012)
9. Dehaspe, L., Raedt, L.D.: Mining association rules in multiple relations. In. Proceedings of the 7th International Workshop on Inductive Logic Programming (1997)
10. Ullman, J.: Answering $n^{2+o(1)}$ counting queries with differential privacy is hard. CoRR abs/1207.6945 (2012)
11. http://www.cs.ox.ac.uk/activities/machlearn/Aleph/aleph.html
12. Muggleton, S.: Inverse entailment and progol. New Gener. Comput. **13**, 245–286 (1995)
13. De Raedt, L.: Statistical relational learning: an inductive logic programming perspective. In: Jorge, A.M., Torgo, L., Brazdil, P.B., Camacho, R., Gama, J. (eds.) PKDD 2005. LNCS (LNAI), vol. 3721, pp. 3–5. Springer, Heidelberg (2005)

Learning Through Hypothesis Refinement Using Answer Set Programming

Duangtida Athakravi(✉), Domenico Corapi, Krysia Broda, and Alessandra Russo

Imperial College London, London, UK
{duangtida.athakravi07,d.corapi,k.broda,a.russo}@ic.ac.uk

Abstract. Recent work has shown how a meta-level approach to inductive logic programming, which uses a semantic-preserving transformation of a learning task into an abductive reasoning problem, can address a large class of multi-predicate, nonmonotonic learning in a sound and complete manner. An Answer Set Programming (ASP) implementation, called ASPAL, has been proposed that uses ASP fixed point computation to solve a learning task, thus delegating the search to the ASP solver. Although this meta-level approach has been shown to be very general and flexible, the scalability of its ASP implementation is constrained by the grounding of the meta-theory. In this paper we build upon these results and propose a new meta-level learning approach that overcomes the scalability problem of ASPAL by breaking the learning process up into small manageable steps and using theory revision over the meta-level representation of the hypothesis space to improve the hypothesis computed at each step. We empirically evaluate the computational gain with respect to ASPAL using two different answer set solvers.

Keywords: Answer Set Programming · Hypothesis refinement

1 Introduction

Recent years have witnessed many novel approaches and systems for nonmonotonic Inductive Logic Programming (ILP) [4,5,11,17]. These address shortcomings of existing solutions (e.g. [7,19]) by providing theoretical semantic underpinning to the notion of nonmonotonic inductive learning [10], and proposing novel algorithms and tools capable of learning normal logic programs from a set of positive and negative examples using nonmonotonic inference [4,5,17]. The search for a hypothesis of a nonmonotonic ILP task involves the traversal of a lattice of different hypotheses, ordered by a generality principle whereby the more given examples explained by a hypothesis, the more general the hypothesis is and vice versa [10]. Systems like XHail [17] and Imparo [11] follow a specific to general search, and are called *bottom-up* approaches, whereas systems like HYPER [2], TopLog [15] and Metagol [16] are *top-down* approaches as they follow a general to specific type of search for the computation of inductive hypotheses.

G. Zaverucha et al. (Eds.): ILP 2013, LNAI 8812, pp. 31–46, 2014.
DOI: 10.1007/978-3-662-44923-3_3

Recent work [4] has demonstrated that a *meta-level* approach, called *TAL* (Top-directed Abductive Learning), can be used to compute hypotheses of a nonmonotonic learning problem by lifting a learning task into a meta-level representation that facilitates reasoning about the possible structures of the hypotheses. Specifically, the learning task is transformed into a semantically equivalent abductive reasoning problem, and meta-level encoding of the hypotheses are abduced. TAL benefits from several advantages over existing learning approaches: it is able to address a large class of non-observational and multi-predicate non-monotonic learning in a sound and complete manner; it allows expressive language bias specifications that subsume mode declarations and can be combined with integrity constraints; and makes use of constraint solving techniques [4]. To combine the theoretical advantages of this approach with computational efficiency, an Answer Set Programming (ASP) implementation of the TAL learning framework, called *ASPAL*, has been proposed in [5] with the objective of exploiting the Answer Set Programming (ASP) efficient approach to nonmonotonic logic programming [1]. The system has shown that fixpoint computation can be used for implementing meta-level learning. ASPAL uses a top-theory constructed by mapping mode declarations into partially instantiated hypotheses and makes use of an efficient ASP solver to find optimal as well as all possible inductive hypotheses. However, as noted in [5], because of the meta-level partial instantiation of possible hypotheses, the grounding of the program can often be very expensive. ASPAL's top theory grows exponentially with respect to the lengths of its clauses, which can cause a learning task to become unsolvable.

In this paper we build upon the results in [5] and propose a novel meta-level learning approach, called *RASPAL*, that overcomes the grounding problem of ASPAL. The approach breaks down a given learning task into small manageable steps and uses theory revision over the meta-level representation of the hypothesis space to improve partial hypotheses computed at each step. The basic idea is to construct hypotheses of a nonmonotonic learning task through multiple iterative theory revisions as iterative refinement steps. At each iterative step a multi-value score scheme is used to select *optimal* refinements among multiple refinement operators. These correspond to revised partial hypothesis that cover the highest (resp. least) number of positive (resp. negative) examples, and that, among such possible solutions, are the most concise revisions. In this way, a large meta-level representation of a learning task can be broken down into multiple steps of learning and/or revision of partial hypotheses that are progressively closer to a consistent and complete learning task solution (i.e. hypothesis that covers all positive examples and none of the given negative examples), so making the learning process more scalable. Empirical evaluation shows that learning tasks that require large grounding of the problem and for which ASPAL is not able to find a solution, can instead be resolved by our RASPAL approach. The paper is structured as follows. Section 2 recalls relevant material our approach builds upon. Sects. 3 and 4 present, respectively, key concepts of RASPAL, and the algorithm with completeness results. Section 5 discusses some experimental outcomes and Sect. 6 concludes with summary and future directions.

2 Background

We assume the reader is familiar with logic programming [13] and ILP [14]. As the focus of this paper is on nonmonotonic ILP [18] and meta-level computation of inductive hypotheses [3], we recall the definition of a nonmonotonic learning task and summarise the Top-directed Abductive Learning approach used by the ASPAL system [5], together with its meta-level encoding of mode declarations.

A nonmonotonic ILP task is defined as a tuple $\langle E, B, M \rangle$ where E is a set of ground literals, called *examples*, B is a normal logic program, called *background theory* and M is a set of mode declarations defining a space R_M of normal clauses. The set E of examples is defined as $\{e_1, \ldots, e_m, \text{not } e_{m+1}, \ldots, \text{not } e_n\}$, where e_i are ground atoms, $\{e_1, \ldots, e_m\}$ represents the set E^+ of positive examples and $\{e_{m+1}, \ldots, e_n\}$ the set E^- of negative examples. An inductive hypothesis, H, is a set of normal clauses in R_M such that $B \cup H \models \bigwedge_{\epsilon_i \in E} \epsilon_i$, where or equivalently:

$$B \cup H \models e_i, \text{ for every } e_i \in E^+ \tag{1}$$

$$B \cup H \not\models e_j, \text{ for every } e_j \in E^- \tag{2}$$

In (1), (2) and throughout this paper the $\models$ symbol denotes the semantic notion of brave induction [20], i.e. they refer together to the existence of a minimal model of $B \cup H$ that covers E, assuming that $B \cup H$ is consistent. The set M of mode declarations provides the schema for the literals that are allowed in a hypothesis. They are defined as $modeh(s)$ and $modeb(s)$ for head and body literals, where s is a ground literal with placemarkers of the form '$+type$', '$-type$', or '$\#type$', and $type$ is the type of the literal's argument. The symbols '+', '−', and '#' indicate, respectively, whether the argument should be an input, an output variable or a constant. An input variable in a body literal b_i is either an input variable in the head of the clause or an output variable in some literal b_j that precedes b_i in the clause (i.e. *link* constraint). An output variable is a free variable in the body of the clause. Given mode declarations M, a clause $h \leftarrow b_1, \ldots, b_n$ is said to be *compatible* with M on the declarations $(m_h, m_{b_1}, \ldots, m_{b_n})$ if h is compatible with m_h (i.e. corresponds to the schema of m_h), and for each $1 \leq i \leq n$, b_i is compatible with m_{b_i} (i.e. corresponds to the schema of m_{b_i}), and arguments of h and each b_i satisfy the constraints of the placemarkers in m_h and m_{b_i} respectively. R_M is the set of normal clauses compatible with M.

For instance, if $modeh(fly(+bird))$ and $modeb(wings(+bird, \#prop, -int))$ are two mode declarations, then a clause compatible with the given mode declarations is $fly(X) \leftarrow wings(X, has\_flight\_feathers, Y)$, where X is a bird, Y is an integer and $has\_flight\_feathers$ is a property of the bird's wings.

2.1 Top-Directed Abductive Learning in ASP

As proposed in [3,5] the computation of inductive hypotheses for a given nonmonotonic ILP task can be translated into a semantically equivalent abductive

task consisting of deriving only ground facts called *abducibles*. An abductive task computes a set Δ of ground literals, from a given set $\mathcal{A}$ of abducibles, that is consistent with a given background knowledge B and that, together with B, entails a given (possibly empty) set g of ground facts called observations (or goal)[1]. The abductive task can also allow for integrity constraints, in which case the abductive solution Δ has, together with B, satisfy the given integrity constraints. Informally, in the TAL approach the background knowledge is augmented with a social "top-theory" and the clause space R_M is "flattened" into a set of logic atoms. This meta-level encoding of the clause space defines the set $\mathcal{A}$ of a semantically equivalent abductive task whose inferred abductive solutions correspond to the inductive hypotheses of the learning task. To avoid the computation of redundant hypotheses, the meta-level encoding of R_M makes use of a *canonical* projection of the clause space R_M into a set R^r_{M} of clauses. This makes each clause in R^r_{M} represents a class of clauses in R_M that are compatible with the same mode declarations in M, but that differ only in the ordering of the body literals not subject to the link constraint over arguments. Every clause within R_M is thus represented by an equivalent clause in R^r_{M} [3,5].

ASP encoding of mode declarations. Let us assume the canonical projection R^r_{M}, compatible with mode bias M, to be the set $\{h_i \leftarrow \overline{b_i} \mid i = 1, \ldots, n\}$ of n clauses, where $\overline{b_i}$ represents the list of literals appearing in the rule of h_i, and let id be a function that associates a unique identification to each clause $r_i \in R^r_{\text{M}}$. We can then automatically construct from a given mode bias M the *top-theory* $\top = \{h_i \leftarrow \overline{b_i}, rule(id(h_i \leftarrow \overline{b_i}), \overline{C}) \mid i = 1, \ldots, n\}$, where $\overline{C}$ is a vector of new variables corresponding to the constant symbols in r_i defined by its mode declarations, and the related set of abducibles $A^\top = \{rule(id(h_i \leftarrow \overline{b_i}), \overline{C}) \mid i = 1, \ldots, n\}$. The $id(r)$ is a tuple of the form $(m_h, m_1, l_{1,1}, \ldots, l_{1,p_1}, \ldots, m_n, l_{n,1}, \ldots, l_{n,p_n})$, where m_h is the identifier of the mode declaration h_i conforms to, m_j is the identifier of the mode declaration the body literal $\overline{b_{i,j}}$ conforms to and, for each m_j, the elements $l_{j,1}, \ldots, l_{j,p_j}$ are numbers denoting the variables in the clause, counting from left to right, which the variable arguments of the literal $b_{i,j}$ are linked to. Note that the variables in a predicate head h do not link to any variable occurring before them in the clause so there is no sequence $l_{h,1}, \ldots, l_{h,p_h}$ after m_h in the representation. The structure of $rule(\cdot)$ makes this translation bijective, allowing automatic inverse translation of abduced atoms to related clause hypothesis: i.e. given a subset $\Delta \subset A^\top$, the inverse translation $h(\Delta) = \{id^{-1}(a) : rule(a) \in \Delta\}$ is the set of clauses $h_i \leftarrow \overline{b_i}, rule(id(h_i \leftarrow \overline{b_i}))$ whose $rule(id(h_i \leftarrow \overline{b_i}))$ are in Δ. The following theorem states the semantic equivalence between an inductive task and its encoding into an abductive task.

Theorem 1. *Let B be a normal logic program, $\tilde{R}$ a set of clauses r_i of the form $h \leftarrow \overline{b}, rule(a, \overline{C})$, where a is $id(h \leftarrow \overline{b})$, and let $A^\top$ be the set of ground atoms $rule(a, \overline{C})$ for which there is a clause $h \leftarrow \overline{b}, rule(a, \overline{C}) \in \tilde{R}$. For each $\Delta \subseteq A^\top$, I is an answer set of $B \cup \tilde{R} \cup \Delta$ if and only if $I \setminus \Delta$ is an answer set of $B \cup h(\Delta)$.*

[1] Note that empty goals are equivalent to $\top$.

Example 1. Consider the ILP task $\langle E, B, M\rangle$ where $B = \{bird(alex); bird(bob); penguin(bob)\}$, $E = \{fly(alex); notfly(bob)\}$, and the set $M = \{modeh(fly(+bird)); modeb(not\ penguin(+bird))\}$ with identifiers $m1$ and $m2$ respectively. The canonical projection of R_M is $R^r_{\text{M}} = R_M = \{fly(X); fly(X) \leftarrow notpenguin(X)\}$, where $id(fly(X))$ is $(m1)$ and the $id(fly(X) \leftarrow not\ penguin(X))$ is $(m1, m2, 1)$. The ASP encoding of R^r_{M} is therefore given by the set of abducibles $A^{\top} = \{rule((m1), c), rule((m1, m2, 1), c)\}$, where the constant c denotes that the clause has no constant symbol, and the top-theory:

$$\top = \left\{ \begin{array}{l} fly(X) \leftarrow bird(X), rule((m1), c)); \\ fly(X) \leftarrow bird(X), not\ penguin(X), rule((m1, m2, 1), c) \end{array} \right\}$$

If we consider the set $\Delta = \{rule((m1, m2, 1), c)\}$, the theory $B \cup \Delta \cup \top$ has the same consequence as the theory $B \cup \{fly(X) \leftarrow bird(X), not\ penguin(X)\} \cup \Delta$.

The above encoding is used by the ASPAL system [5] to transform the problem of finding candidate hypotheses of an ILP task into a problem representable in ASP [5]. Given an ILP task $\langle E, B, M\rangle$ the ASPAL system solves the equivalent ALP task where the background knowledge is given by $B \cup \top \cup \{examples \leftarrow \bigwedge \overline{E}\}$, the set of abducibles is $A^{\top}$, the set of integrity constraints includes the constraint $\{\bot \leftarrow not\ examples\}$ and the given observation is empty. The added integrity constraint ensures that each answer set solution covers all positive examples and none of the negative ones.

The overall computation time and space of ASPAL are affected mostly by the grounding that the ASP solver has to go through to transform a program into an equivalent ground program. Despite the use of heuristics to speed up the computation, the grounding is often the bottleneck of the whole process, with the number of clauses in $\top$ being the key factor affecting this. Let us assume a mode declaration M where M_h is the number of head mode declarations and M_b is the number of body mode declarations, let max_o (resp. max_i) be the biggest number of output (resp. input) variables that appear in the body mode declarations, let max^h_i be the maximum number of input variables in the head mode declarations, and let d_{max} be the maximum number of body literals in a inductive hypothesis for a given ILP task. Then

$$|\top| \leq \sum_{d=0}^{d_{max}} |M_h| \times (|M_b| \times (max^h_i + max_o \times (d-1))^{max_i})^d$$

This is adapted from [5] and is an upper bound on $|\top|$. In practice the size of the top theory is smaller as clauses in the hypothesis space may have to satisfy type constraints on their variables. All parameters, except d_{max}, strictly depend on the given ILP task.

The main idea of our new meta-level learning approach, called RASPAL, is to put an upper bound on d_{max} and provide an ASP-based nonmonotonic ILP system that does not suffer of the grounding problem of ASPAL.

3 Learning Through Hypothesis Refinement

In this section we present the key features of *RASPAL*. The approach combines the TAL approach of computing inductive hypotheses through abductive search over a meta-level representation of the hypothesis space with the notion of theory revision. Theory revision is a type of theory refinement [22] that consists of inferring changes over a given theory in order to change its consequences. RASPAL uses theory revision during the learning process to revise partial hypotheses (i.e. hypotheses that do not yet satisfy (1) and (2)), and construct through revision steps the desired inductive solution. The revision of partial hypotheses is guided by the same examples as the initial ILP task, as the final inductive solution, together with the initially given background knowledge, has to bravely entail the examples. The revision of partial hypotheses is performed as a nonmonotonic learning task.

3.1 Hypothesis Refinement

We briefly describe our approach of theory revision through learning, and we then show how it is used in RASPAL to revise partial hypotheses during the computation of an inductive hypothesis for a given ILP task. Consider two normal logic programs H and H'. A *change transaction* C is a set of revision operations such that H' is attained by applying all revision operations in C to H. Revision operations are of four different types: addition of a clause, deletion of a clause, addition of a body literal to an existing clause, and deletion of a body literal from an existing clause. Change transactions are normally required in order to impose changes over the consequences of a given theory. A theory revision task can be defined as follows, where the clause space for possible revisions is defined in terms of mode declarations M.

Definition 1. *A theory revision task is a tuple* $\langle E, B \cup H, M\rangle$ *where* E *is a set of literals, called examples,* B *is a background theory,* H *is a* revisable theory*, and* M *is a set of mode declarations defining the clause space of revised theories. The theory* H'*, called* revised theory*, is an inductive hypothesis for the task* $\langle E, B \cup H, M\rangle$ *if and only if (i)* $H' \subseteq R_M$*; and (ii)* $B \cup H' \vDash E$*.*

The above theory revision task can be computed using the nonmonotonic meta-level refinement approach used in [6]. The revision operation of adding a new clause, from the given clause space R_M, matches directly the task of learning a new clause. To learn the other revision operations, the mode declaration M is extended by Δ_M with special mode declarations: $modeh(extension(\#rule\_id, +vars))$ and $modeh(delete(\#rule\_id, \#body\_id))$. The argument $\#rule\_id$ is a reified term that identifies an existing clause in H where a change (addition or deletion of a body literal) is learned and $vars$ is the list of variables in the existing clause that are involved in the change. The learning task $\langle E, B \cup H, M\rangle$ for theory revision is transformed into a nonmonotonic learning task $\langle E, B \cup \tilde{H}, M \cup \Delta_M\rangle$. The revisable theory $\tilde{H}$ is constructed from H as follows. For each normal clause $h_i \leftarrow b_{i,1}, \ldots, b_{i,n} \in H$, the following normal clauses are added to $\tilde{H}$:

- $h_i \leftarrow try(i,1,vars(b_{i,1})),\ldots,try(i,n,vars(b_{i,n})),\ extension(i,vars(r_i))$
- $try(i,j,vars(b_{i,j})) \leftarrow b_{i,j}, not\ delete(i,j)$, for each $try(i,j,vars(b_{i,j}))$
- $try(i,j,vars(b_{i,j})) \leftarrow delete(i,j)$, for each $try(i,j,vars(b_{i,j}))$
- $\bot \leftarrow delete(i,j), \{extension(i,vars(r_i))\}0$, for each $delete(i,j)$

The indices i and n uniquely identify the clauses and their conditions in H, $vars(r_i)$ is the list of all variables in r_i and $vars(b_{i,j})$ is the list of all variables in $b_{i,j}$. The *try* clauses test if a body literal $b_{i,j}$ in the ith clause is needed in the revised clause. If it is no longer relevant, then the corresponding $delete(i,j)$ instance is learnt, indicating that it can be removed. The constraints on each learnable $delete(i,j)$, where $\{extension(i,vars(r_i))\}0$ denotes that there are at most 0 instance of $extension(i,vars(r_i))$, ensures that deletions are only learnt if the corresponding extension clause is learnt. Revisions of H are then learned by solving the nonmonotonic learning task $\langle E, B \cup \tilde{H}, M \cup \Delta_M \rangle$ to find a set of change transactions C and generating H' by applying C to H.

The change transactions C consist of a (possibly empty) set of *delete* facts, clauses with head predicate *extension* and body literals using predicates from body declarations in M, and other clauses compatible with M. They correspond, respectively, to the revision operators of deletion, addition of body literals to existing clauses, and addition of new rules. H' is generated from the given revisable theory H and the learned change transactions C by the following steps:

- For each pair: $r_i \leftarrow b_1,\ldots,b_n \in H$ and $extension(r_i,vars(r_i)) \leftarrow b_{n+1},\ldots,b_m$ in C, the clause $r_i \leftarrow b_1,\ldots,b_n,b_{n+1},\ldots,b_m$ is added to H'.
- For each $delete(i,j)$ in C, the body literal $b_{i,j}$ is removed from clause r_i in H' (only modify clauses that are retained by C).
- Each clause $r_{new} \in C$ that are not *extension* nor *delete*, with $r_{new} \notin H$, r_{new} is added to H'.
- Each clause in H that does not have a corresponding *extension* clause in C is not added to H' (i.e. capturing the deletion of an existing clause).

Example 2. Consider the theory revision task $\langle E, B \cup H, M \rangle$ defined as follows: $B = \{\leftarrow fly(X), injured(X); pigeon(alex); pigeon(bill); ostrich(bob); seagull(clark); bird(X)\}$; $H = \{fly(X); fly(X) \leftarrow pigeon(X); injured(bill)\}$; $E = \{fly(alex); fly(clark);\ not\ fly(bob);\ not\ fly(bill)\}$; $M = \{m1{:}modeh(injured(\#bird));\ m2{:}modeb(fly(+bird));\ m3{:}modeb(pigeon(+bird));\ m4{:}modeb(not\ ostrich(+bird))\}$.

The transformed revisable theory is

$$\tilde{H} = \left\{ \begin{array}{l} fly(X) \leftarrow extension(1,vars(X)) \\ fly(X) \leftarrow try(2,1,vars(X)), extension(2,vars(X)) \\ try(2,1,vars(X)) \leftarrow pigeon(X), not\ delete(2,1) \\ try(2,1,vars(X)) \leftarrow delete(2,1) \\ \bot \leftarrow delete(2,1), \{extension(2,vars(X))\}0. \\ injured(bill) \leftarrow extension(3,vars) \end{array} \right\}$$

The set C is the learned change transactions and H' is the revised theory generated from H by applying the learned revision operation included in C

$$C = \begin{cases} extension(2, vars(X)) \\ \quad \leftarrow not\ ostrich(X) \\ delete(2,1) \\ extension(3, vars) \end{cases} \qquad H' = \begin{cases} fly(X) \leftarrow not\ ostrich(X) \\ injured(bill). \end{cases}$$

3.2 Learning a Partial Hypothesis

Given a nonmonotonic ILP task $\langle E, B, M\rangle$, a hypothesis H in the clause space R_M is said to *cover* an example e if $B \cup H \vDash e$. We denote with $cover_e$ the number of examples that are covered by a hypothesis H. A hypothesis H in R_M is said to be *complete* if, together with B, it covers all positive examples (H satisfies (1)); H is said to be *consistent* if, together with B, it does not cover any of the negative examples (H satisfies (2)). A hypothesis H in R_M is said to be a *partial hypothesis* if it is not a consistent and complete hypothesis. Our RASPAL learning approach computes consistent and complete hypotheses of an ILP task through revision steps over partial hypotheses. Each revision step generates a revised partial hypothesis that is closer to the desired consistent and complete hypothesis, i.e. a (revised and partial) hypothesis that covers the highest number of positive examples and the least number of negative examples, with it, or for change transaction its corresponding revised partial hypothesis, being the most concise. To compare hypotheses, we define a notion of *score* of a (partial) hypothesis, which induces a total ordering over partial hypotheses.

Definition 2. *Let $\langle E, B, M\rangle$ be an ILP task and let H be a partial hypothesis in R_M. The score of H is the tuple $score(H) = \langle cover_{e^+}(H), cover_{e^-}(H), len(H)\rangle$, where $cover_{e^+}(H)$ (resp. $cover_{e^-}(H)$) is the number of positive (resp. negative) examples covered by H and $len(H)$ is the total number of literals in H.*

Definition 3. *Let $\langle E, B, M\rangle$ be an ILP task and let H and H' be two (partial) hypotheses in R_M. H is* better *than H', denoted $score(H) > score(H')$ iff one of the following cases applies:*

- $cover_{e^+}(H) > cover_{e^+}(H')$,
- $cover_{e^+}(H) = cover_{e^+}(H') \wedge cover_{e^-}(H) < cover_{e^-}(H')$,
- $cover_{e^+}(H) = cover_{e^+}(H') \wedge cover_{e^-}(H) = cover_{e^-}(H') \wedge len(H) < len(H')$

Given a set $\mathcal{H}$ of (partial) hypotheses, the *optimal* (partial) hypothesis H_{opt} is a (partial) hypothesis in $\mathcal{H}$ such that $\forall H \in \mathcal{H} : score(H_{opt}) \geq score(H)$.

Example 3. Consider the ILP task in Example 1. The four (partial) hypotheses ranked from lowest to highest score are as follows, where the optimal $H4$ is the consistent and complete hypothesis, and the others are partial hypotheses:

The above scoring scheme differs from more conventional scoring mechanismes for inductive hypothesis. For instance, two other scoring schemes could be in principle assumed, namely $\langle cover_{e^+}(H) - cover_{e^-}(H), len(H)\rangle$ and the more

	Hypothesis	Score		Hypothesis	Score
$H1$	The empty hypothesis	$\langle 0,0,0\rangle$	$H2$	$fly(X)$ and $fly(X) \leftarrow not\ penguin(X)$	$\langle 1,1,3\rangle$
$H3$	$fly(X)$	$\langle 1,1,1\rangle$	$H4$	$fly(X) \leftarrow not\ penguin(X)$	$\langle 1,0,2\rangle$

traditional one $cover_{e^+}(H) - cover_{e^-}(H) - len(H)$. The latter scheme, given by the number of positive examples covered minus the number of negative examples covered and the hypothesis size would be unsuitable as it would give too much emphasis to the hypothesis size, which is not relevant for discriminating among potential revisions that have to lead to consistent and complete hypothesis. The former scoring scheme emphasises the coverage of examples (i.e. number of positive examples minus number of negative coverage) over the hypothesis size. It has the advantage of having smaller range of score values (from $-|E^-|$ to $|E^+|$ as opposed to 0 to $|E^+| \times |E^-|$), which could help reduce the search for the optimal hypothesis. However, it is too general and would result in learned revisions that are indistinguishable from each other. For instance, value 0 in $\langle 0, len(H)\rangle$ could correspond to a revised hypothesis that covers none of the positive or negative examples, or equal number of positive and negative examples. In Example 3, if this score scheme were used then the empty hypothesis would have score $\langle 0,0\rangle$ and $fly(X)$ would have score $\langle 0,1\rangle$. Therefore the empty hypothesis would have better score than $fly(X)$. This is not suitable for in our learning through hypothesis refinement as, unlike the empty hypothesis, $fly(X)$ could potentially be refined to find better hypotheses. Empirical testing of using this alternative scoring scheme in our RASPAL algorithm has shown than a larger number of iterations were needed so leading to larger programs to ground than necessary.

4 RASPAL: Iterative Learning by Refinement

Here we describe the RASPAL algorithm. It solves an ILP task through iterative hypothesis refinement using the method of theory revision presented in Sect. 3. We also show that our algorithm is complete with respect to the notion of inductive hypothesis given in Sect. 2.

4.1 Algorithms

In Sect. 2.1 we have pointed out that a drawback of the ASPAL approach is the grounding of the top-theory. Intuitively, the RASPAL algorithm overcomes this problem by finding a partial hypothesis to the ILP task, and performing iterative steps of revisions over selected partial hypothesis according to the scoring method above. At each computation step, a small size boundary is imposed on the clause length of the refinements learnt, so requiring a smaller grounding and making each step computationally more manageable. The main learning algorithm is Algorithm 1. The function Learn takes as input a learning task P, and a maximum hypothesis clause length i. It first tries to find an optimal

hypothesis within the subset of the clause space constrained by i, using the function FindOptimalHypothesis. This can, in principle, be computed using any sound and complete nonmonotonic ILP system. We assume it to be executed by the ASPAL system with parameter clause size limited to i, and using Clingo [8] as the ASP solver. To generated all partial hypotheses the goal of the ALP task is replaced with Clingo's optimisation statements are used to find answer sets with maximum number of positive examples, minimum number of negative examples, and minimal sum of the size of clauses in the hypothesis. The size of clauses in the hypothesis are given in the optimisation statement as weights for each abducibles, each assigned weight is the total size of the revised clause, and *extension* literals have the size 0 while *delete* has the size -1. The optimisation statements are each given priorities with the number of positive examples higher than the negative ones, and then prioritising both of these higher than the size of the hypothesis. This step returns a hypothesis with highest score within the limit clause size. Should the hypothesis be empty, i is then increased by 1 and Learn is executed again. Note that an empty hypothesis would be returned in case every learnable hypothesis covers no positive examples and proves every negative example.

Algorithm 1. Learn(P, i)
Require: $P = \langle E, B, M \rangle$, with E^+ and E^- being the set of positive and negative examples in E respectively
Output: $\langle Hypothesis, Score \rangle$, a complete and consistent hypothesis of P and its score

```
1: let ⟨Hypothesis, Score⟩ =FindOptimalHypothesis(P, i)
2: if Hypothesis == ∅ then                          ▷ No hypothesis found
3:     return Learn(P, i + 1)
4: loop
5:     if Score = ⟨|E+|, 0, _⟩ then      ▷ Complete and consistent hypothesis found
6:         return ⟨Hypothesis, Score⟩
7:     ⟨Hypothesis, Score_new⟩ =RefineHypothesis(Hypothesis, P, i)
8:     if Score_new ≤ Score then                    ▷ Score does not improve
9:         return Learn(P, i + 1)
10:    Score = Score_new
```

In the loop structure (lines 4–10), the current $Hypothesis$ is checked for consistency and completeness, based on its score value. If it is not, then it is refined via RefineHypothesis, which also outputs an updated score for the refinement which is checked to see whether the refinement made any improvements. If the new score is an improvement then $Score$ is updated before the next loop iteration. If the new score is not better than the previous one, the algorithm Learn is re-run with i increased by 1. The function RefineHypothesis is shown in Algorithm 2.

Given a partial $Hypothesis$, the ILP task P, and the clause length limit i, the algorithm returns a refined $Hypothesis$ and its score. This is done by creating a theory revision task P' using SetRefinement function as described

Algorithm 2. RefineHypothesis($Hypothesis, P, i$)
Output: $\langle Hypothesis_{new}, Score_{new} \rangle$, a refinement of $Hypothesis$ of and its score

1: $P' = \langle E, B \cup \Delta_B, M \cup \Delta_M \rangle$ = SetRefinement($Hypothesis$)
2: $\langle Changes, Score_{new} \rangle$ = FindOptimalHypothesis($P', i+1$)
3: $Hypothesis_{new}$ = ApplyRefinement($Hypothesis, Changes$)
4: **return** $\langle Hypothesis_{new}, Score_{new} \rangle$

in Sect. 3.1. Then using FindOptimalHypothesis to learn the change transactions $Changes$, with associated score $Score_{new}$, as hypothesis for the revision task. The changes are then applied to the current hypothesis by the ApplyRefinement function to form a revised $Hypothesis_{new}$. Note that, at line 2 in Algorithm 2, the function FindOptimalHypothesis is called with $i+1$. This is because for RefineHypothesis to extend a clause in the current hypothesis by i literals, it must be able to learn an extension clause with i body literals, thus having length $i+1$.

Consider the following task to learn the concept of even and odd numbers:

$$B = \begin{cases} even(0) \\ num(0) \\ num(s(0)) \\ num(s(s(0))) \\ num(s(s(s(0)))) \\ num(s(s(s(s(0))))) \\ num(s(s(s(s(s(0)))))) \\ succ(X, s(X)) \leftarrow \\ num(X), num(s(X)) \end{cases} \quad E^+ = \begin{cases} even(s(s(s(s(0))))) \\ odd(s(s(s(s(s(0)))))) \end{cases} \quad E^- = \begin{cases} odd(0) \\ even(s(s(0))) \\ odd(s(s(s(s(0))))) \\ even(s(s(s(0)))) \end{cases}$$

$$M = \begin{cases} modeh(odd(+num)) \\ modeh(even(+num)) \\ modeb(not\ even(+num)) \\ modeb(even(+num)) \\ modeb(succ(-num, +num)) \end{cases}$$

Figure 1 shows the hypothesis in each iteration of RASPAL, called with $i = 1$.

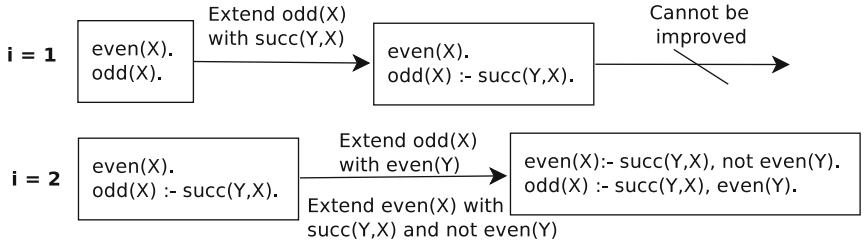


Fig. 1. Using RASPAL to learn the concept of even and odd number

In the first call of Learn with $i = 1$ there is no further improvement at the second iteration of RefineHypothesis. Therefore, learn is restarted with i increased to 2. The last hypothesis for the previous i value is re-learned, but this time it can be further revised, as $i = 2$, into $even(X) \leftarrow succ(Y, X), not\ even(Y)$ at the second refinement iteration. It can be shown that, assuming the existence

of a consistent and complete hypothesis for an ILP task P with clause length no greater than an I_{max}, the number of refinement iterations (i.e. calls to RefineHypothesis) is less than or equal to $|E^+| \times |E^-| \times I_{max}$, following from our ordering over the scores. This is due to the fact that a better hypothesis covers either one more positive example or one less negative one after refinement. Furthermore, expanded clauses would not exceed I_{max} in length because of the given assumption.

The following theorem shows the completeness of our RASPAL approach. The proof builds upon the completeness of ASPAL, which is used in FindOptimalHypothesis and subsequently in RefineHypothesis.

Theorem 2. *Let $P = \langle E, B, M\rangle$ be an inductive task and let $\mathcal{H}$ be a set of complete and consistent hypotheses of P. Then* Learn(P, i) *returns a tuple $\langle H, S\rangle$, where H is a hypothesis $H \in \mathcal{H}$, for some $i > 0$, and S is the score of H.*

Proof. Assume that for any consistent and complete hypothesis the maximum clause length is smaller than or equal to I_{max} and the smallest hypothesis size is L. Let Learn(P, i) be called for some $i \geq 1$. We consider the following cases:

Case 1: ($i \geq I_{max}$). Since a complete and consistent hypothesis exists and FindOptimalHypothesis is assumed to be complete, then FindOptimalHypothesis will find a complete and consistent hypothesis.

Case 2: ($I_{max} - i > 0$). We reason by induction on $I_{max} - i$. Assume as induction hypothesis that for all $j \geq 1$, where $I_{max} - j < I_{max} - i$, the theorem holds. In either of the two cases where Learn$(P, i + 1)$ is called (i.e. line 3 or line 9) the induction hypothesis can be applied since $I_{max} - (i + 1) < I_{max} - i$.

Case 3: ($I_{max} - i > 0$ and Learn is not called recursively). It needs to be shown that there is only a finite number of iterations with RefineHypothesis after which a consistent and complete hypothesis is generated (i.e. score can only increase a finite number of times). For RefineHypothesis to be repeatedly called, each time the score of the revised hypothesis has to be better than the current hypothesis. But the score can get better by either the number of positive example covered increases, or the number of negative examples covered decreases. This can continue until eventually a refined consistent and complete hypothesis is reached with score $\langle |E^+|, 0, K\rangle$, where $K \geq L$.

5 Experiment

In this section we compare RASPAL with ASPAL [5]. For each learning task[2] the largest ASP program produced by each system were run on two different ASP solvers, Clingo [8] and DLV [12]. This has been done to check the compatibility between the solvers and the RASPAL approach, and to show that the grounding problem of the learning program is universal for all ASP solvers.

Both the ASPAL and RASPAL encoding of a learning task as an ASP abductive task is already compatible with Clingo. For the DLV environment we have

[2] The full details of the learning tasks can be found at https://dl.dropboxusercontent.com/u/15091371/ILP2013_examples.pdf.

Table 1. Maximum size of the ground program and number of ungrounded clauses in the top theories. For RASPAL Δ_B is the number of clauses in the space of revisable hypothesis that is added to the background knowledge.

Learning Task	ASPAL			RASPAL		
	DLV	Clingo	$\|\top\|$	DLV	Clingo	$\|\top\| + \Delta_B$
odd/even	17.0 kB	11.3 kB	23	172.9 kB	159.2 kB	$65 + 5 = 70$
nonealike	-	-	251176	35.7 MB	40.5 MB	$105 + 10 = 115$
train	932.8 kB	518.0 kB	118	126.4 MB	131.0 MB	$238 + 20 = 258$
mobile	-	11.2 GB	1200	-	1.9 GB	$85 + 18 = 103$

modified this encoding by (i) replacing Clingo's choice clause by disjunctive ones; (ii) using aggregate functions instead of the limits on a choice clause, in order to limit the number of clause in the hypothesis, and defining the constraint on the abducible *delete*/2 literals; (iii) replacing optimisation statements over examples coverage by soft constraint. While both solvers return the same answer set, when solving a learning task by RASPAL on DLV, additional post-processing is required to find the most optimal partial hypothesis. To compare the sizes of the ground programs in Table 1, their sizes were found by making the solvers ground the program but not solve it.

Regarding DLV and Clingo, the results in Table 1 show that for smaller learning tasks there is not much difference in the size of the ground programs produced by the solvers. When Clingo cannot ground the program for the *nonealike* task, neither could DLV. Moreover for large problems that can be grounded by Clingo, such as the *mobile* task, DLV can neither solve nor ground the ASP program. For this reason and for the availability of optimisation statements, we have found Clingo to be more suitable for solving our RASPAL learning approach.

Regarding the number of ungrounded clauses in the top theory of each learning task, note that for RASPAL the value Δ_B denotes the number of clauses of the revisable hypothesis added to the background knowledge. For the learning tasks *odd/even* and *train* both the number of clauses in the top theory and the size of the ground programs of RASPAL are much larger than those of ASPAL. This is because RASPAL has an overhead due to the revisable theory and the additional mode declarations used to learn the revision operations, making less it efficient than ASPAL for solving smaller learning tasks. However, for the *nonealike* learning task, the top theory of ASPAL is very large as there are many permutations of the variables in the learnable clauses. Similarly for the *mobile* learning tasks, where the domain knowledge is also larger than those in the other learning tasks. For both these learning tasks RASPAL's hypothesis refinement approach can significantly reduce the size of the program's grounding. This reduction by RASPAL is due to the difference in the maximum clause size of the complete and consistent hypothesis and the minimum value of i required to learn it. In the learning tasks *odd/even* and *train*, which have maximum clause sizes of, respectively, 3 and 4, the minimum value of i required to learn these

tasks are 2 and 3 respectively. This is because their hypotheses' body literals are highly dependent on one another, thus the learning cannot be performed through independent revision iterations with smaller values for i. This makes the overhead of refining the partial hypotheses greater than the advantage of reducing the top theory's maximum clause size. On the other hand, for the *nonealike* and *mobile* learning tasks, the maximum clause sizes are 5 and 4 respectively. In these cases hypotheses have less dependencies between their body literals, so for both tasks RASPAL is ideal in finding a complete and consistent hypothesis using a very small value of i ($i = 1$), which allows each learning task to be solved by a much smaller top theory compared to that used in ASPAL.

6 Conclusion and Future Work

In this paper we have explored how iterative refinement could be used with fixpoint computation of ASP to improve a bottleneck computation of the ASPAL system. We have implemented our RASPAL approach and compared it against ASPAL. Our tests have demonstrated the impact of the additional mode declarations for learning change transactions: RASPAL performs worse than ASPAL when the learning task is small and the difference between i and the maximum size of a hypothesis clause is also small. On the other hand, when the search space for the hypothesis becomes extremely large and literals in the hypothesis are not strongly dependent on one another, our refinement approach is able to solve the task using a much smaller top theory.

In this paper we have concentrated on the comparison between ASPAL and RASPAL, but there are also other works that are similar to RASPAL and ASPAL. Past ILP algorithms have frequently used meta-level information to help with the search for the hypothesis, the language bias being the commonly used meta-level information for limiting the hypothesis search space. Systems such as ASPAL, RASPAL and Metagol [16] have taken a step further in this direction by transforming the original learning task into a meta-level learning task. This involves using transformations to abstract some or all of the inductive task into a corresponding meta-level representation. Unlike ASPAL and RASPAL, which only abstract their hypotheses and top theories, Metagol uses second order predicates to represent all of its learning task in meta-level form. By using meta-level representation of only the hypothesis space, ASPAL and RASPAL can still reason about the object level semantics and therefore allow more easily the learning of nonmonotonic hypotheses. Currently, Metagol does not extend to learning nonmonotonic hypotheses.

RASPAL also belongs to the subclass of ILP systems that use incremental learning. HYPER [2] is another example of an ILP system that learns through refinement. However, differently from our approach, it constructs the hypothesis by first finding an overly general partial hypothesis (i.e. one that covers all positive examples), which is then specialised until it covers no negative examples. In addition to being unable to learn nonmonotonic tasks, HYPER's refinement will only add at most a single body atom to each clause per each iteration.

This makes it unable to learn hypotheses where more than one body atom must be added to the same clause in the same iteration to impact the score of the refined hypothesis.

A recently proposed incremental learning system that is more related to RASPAL is ILED [9], an ILP system based on XHAIL [17], which is designed to address the scalability problem of learning from continuously collected real life temporal data. Like our work it uses hypothesis refinement and abductive reasoning for learning, and is capable of learning nonmonotonic clauses. However, unlike RASPAL, which uses refinement for learning a single learning task, ILED's incremental learning is used for processing new knowledge and incorporating it into previous learnt concepts.

There are many future directions for our work. Our RASPAL approach has potentials for making further contributions in the area of Predicate Invention [21]. Initial attempts of using ASPAL for predicate invention have shown that the search space can grow very large as there are many possible formats new predicates could take. Iterative refinement in this case could be very beneficial. The implementation could also be further optimised to eliminate re-computations.

Acknowledgment. This work is partially funded by the 7th Framework EU-FET project 600792 ALLOW Ensembles and the EPSRC project P44745.

References

1. Baral, C.: Knowledge Representation, Reasoning and Declarative Problem Solving. Cambridge University Press, Cambridge (2003)
2. Bratko, I.: Refining complete hypotheses in ILP. In: Džeroski, S., Flach, P.A. (eds.) ILP 1999. LNCS (LNAI), vol. 1634, pp. 44–55. Springer, Heidelberg (1999)
3. Corapi, D.: Nonmonotonic inductive logic programming as abductive search. Ph.D. thesis, Imperial College London (2011)
4. Corapi, D., Russo, A., Lupu, E.: Inductive logic programming as abductive search. In: Hermenegildo, M., Schaub, T. (eds.) Technical Communications of the 26th International Conference on Logic Programming (2010)
5. Corapi, D., Russo, A., Lupu, E.: Inductive logic programming in answer set programming. In: Muggleton, S.H., Tamaddoni-Nezhad, A., Lisi, F.A. (eds.) ILP 2011. LNCS, vol. 7207, pp. 91–97. Springer, Heidelberg (2012)
6. Corapi, D., Russo, A., Vos, M.D., Padget, J.A., Satoh, K.: Normative design using inductive learning. TPLP **11**(4–5), 783–799 (2011)
7. Dimopoulos, Y., Kakas, A.: Learning non-monotonic logic programs: learning exceptions. In: Lavrač, N., Wrobel, S. (eds.) ECML 1995. LNCS, vol. 912, pp. 122–137. Springer, Heidelberg (1995)
8. Gebser, M., Kaminski, R., Kaufmann, B., Ostrowski, M., Schaub, T., Schneider, M.: Potassco: the Potsdam answer set solving collection. AI Commun. **24**(2), 105–124 (2011)
9. Katzouris, N., Artikis, A., Paliouras, G.: Incremental learning of event definitions with inductive logic programming. CoRR abs/1402.5988 (2014)
10. Kimber, T.: Learning definite and normal logic programs by induction on failure. Ph.D. thesis, Imperial College London (2012)

11. Kimber, T., Broda, K., Russo, A.: Induction on failure: learning connected horn theories. In: Erdem, E., Lin, F., Schaub, T. (eds.) LPNMR 2009. LNCS, vol. 5753, pp. 169–181. Springer, Heidelberg (2009)
12. Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S., Scarcello, F.: The DLV system for knowledge representation and reasoning. ACM Trans. Comput. Logic **7**(3), 499–562 (2006)
13. Lloyd, J.: Foundations of logic programming. Springer, New York (1984)
14. Muggleton, S., De Raedt, L.: Inductive logic programming: theory and methods. J. Logic Program. **19–20**(20), 629–679 (1994)
15. Muggleton, S.H., Santos, J.C.A., Tamaddoni-Nezhad, A.: TopLog: ILP using a logic program declarative bias. In: Garcia de la Banda, M., Pontelli, E. (eds.) ICLP 2008. LNCS, vol. 5366, pp. 687–692. Springer, Heidelberg (2008)
16. Muggleton, S.H., Lin, D.: Meta-interpretive learning of higher-order dyadic datalog: predicate invention revisited. In: IJCAI (2013)
17. Ray, O.: Nonmonotonic abductive inductive learning. J. Appl. Logic **7**(3), 329–340 (2008)
18. Sakama, C.: Nonmonotonic inductive logic programming. In: Eiter, T., Faber, W., Truszczyński, M. (eds.) LPNMR 2001. LNCS (LNAI), vol. 2173, pp. 62–80. Springer, Heidelberg (2001)
19. Sakama, C.: Induction from answer sets in nonmonotonic logic programs. ACM Trans. Comput. Logic **6**(2), 203–231 (2005)
20. Sakama, C., Inoue, K.: Brave induction: a logical framework for learning from incomplete information. Mach. Learn. **67**(1), 3–35 (2009)
21. Stahl, I.: Predicate invention in inductive logic programming. In: De Raedt, L. (ed.) Advances in Inductive Logic Programming, pp. 34–47. IOS Press, Amsterdam (1996)
22. Wrobel, S.: First order theory refinement. In: De Raedt, L. (ed.) Advances in Inductive Logic Programming, pp. 14–33. IOS Press, Amsterdam (1996)

A BDD-Based Algorithm for Learning from Interpretation Transition

Tony Ribeiro[1(✉)], Katsumi Inoue[1,2], and Chiaki Sakama[3]

[1] The Graduate University for Advanced Studies (Sokendai),
2-1-2 Hitotsubashi, Chiyoda-ku, Tokyo 101-8430, Japan
{tony_ribeiro,inoue}@nii.ac.jp
[2] National Institute of Informatics,
2-1-2 Hitotsubashi, Chiyoda-ku, Tokyo 101-8430, Japan
[3] Department of Computer and Communication Sciences,
Sakaedani, Wakayama 640-8510, Japan
sakama@sys.wakayama-u.ac.jp

Abstract. In recent years, there has been an extensive interest in learning the dynamics of systems. For this purpose, a new learning method called learning from interpretation transition has been proposed recently [1]. However, both the run time and the memory space of this algorithm are exponential, so a better data structure and an efficient algorithm have been awaited. In this paper, we propose a new learning algorithm of this method utilizing an efficient data structure inspired from Ordered Binary Decision Diagrams. We show empirically that using this representation we can perform the same learning task faster with less memory space.

1 Introduction

In recent years, there has been a notable interest in the field of Inductive Logic Programming (ILP) to learn from system state transitions as part of a wider interest in learning the dynamics of systems [1,2]. Learning system dynamics has many applications in multi-agent systems, robotics and bioinformatics alike. Knowledge of system dynamics can be used by agents and robots for planning and scheduling. In bioinformatics, learning the dynamics of biological systems can correspond to the identification of the influence of genes and can help to design more efficient drugs. In some previous works, state transition systems are represented with logic programs [3,4], in which the state of the world is represented by an Herbrand interpretation and the dynamics that rule the environment changes are represented by a logic program P. The rules in P specify the

This research was supported in part by the NII research project on "Dynamic Constraint Networks" and by the "Systems Resilience" project at Research Organization of Information and Systems, Japan. We would like to thank Earl Belinger for its help to improve the english quality of the paper.

G. Zaverucha et al. (Eds.): ILP 2013, LNAI 8812, pp. 47–63, 2014.
DOI: 10.1007/978-3-662-44923-3_4

next state of the world as an Herbrand interpretation through the *immediate consequence operator* (also called the T_P *operator*) [5,6]. With such a background, Inoue *et al.* [1] have recently proposed a framework to learn logic programs from traces of interpretation transitions (LFIT). The learning setting of this framework is as follows. We are given a set of pairs of Herbrand interpretations (I, J) as positive examples such that $J = T_P(I)$, and the goal is to induce a *normal logic program* (NLP) P that realizes the given transition relations. In [1], the authors showed one of the possible usages of LFIT: **LF1T**, *learning from 1-step transitions*. In that paper, an algorithm is proposed to iteratively learn an NLP that realizes the dynamics of the system by considering step transitions one by one. The iterative character of LF1T has applications in bioinformatics, cellular automata, multi-agent systems and robotics. We can easily imagine an agent or a robot that learns the dynamics of its environment from its observations, learning the consequences of its actions according to the state of the world step-by-step. Aggregating more and more observations, the agent becomes able to predict the evolution of the world more precisely and can use this knowledge for planning and scheduling.

In this paper, we propose a new version of the LF1T algorithm based on Binary Decision Diagrams (BDDs) [7,8]. A BDD is a canonical representation of a Boolean formula which has been successfully used in many research fields such as Boolean satisfiability solvers [9], data mining [10], ILP [11] and abduction [12,13]. ProbLog [11] is a probabilistic logic programming language that computes probabilities via BDDs. A ProbLog program computes the probability of a query atom by applying sum-product computation to a BDD, but allows definite clauses only. For abduction in propositional theories, Simon and del Val [12] propose a consequence-finding procedure implemented on Zero-suppressed BDDs. Inoue *et al.* [13] run the EM algorithm over BDDs to evaluate abductive hypotheses.

The main concern of our LF1T algorithm is the size of NLPs learned. For the sake of memory usage and reasoning time, a small NLP could be preferred in multi-agent and robotics applications. In bioinformatics, it can be easier and faster to perform model checking on Boolean networks represented by a compact NLP than the set of all state transitions. In previous algorithms, LF1T uses resolution techniques to generalize rules and reduces the size of the output NLP. The novelty of our approach is the adaptation of these techniques to the BDD structure. Here, we develop a method to perform LF1T operations on a BDD that also realizes usual BDD merging operations as well as novel simplification operations. We represent an NLP by a set of BDD structures where each BDD encodes rules with the same head literal. Assuming that rules respect a variable ordering, our data structure is similar to an Ordered BDD (OBDD) [14,15]. In our approach, each BDD represents a formula in disjunctive normal form that defines whether a literal is true at the next time step. Because LF1T does not learn negative rules, our structure only represents rules that imply the head literal to be true. In that sense it can also be considered a Zero-suppressed Binary Decision Diagram (ZDD) [16].

Using a BDD representation we can also merge the common part of rules and learn the same NLP with less memory usage than in previous versions of LF1T. One weak point of the previous LF1T algorithm is that learning becomes slower and slower as the NLP learned becomes bigger because it has to check more and more rules. In practice, the compact representation of the BDD structure reduces the sensitivity of the LF1T learning time to the NLP size. Study of the computational complexity of our new method shows that it remains equivalent to the previous version of LF1T in the worst case. Using examples from the biological literature we show through experimental results that our new algorithm still outperforms the two previous versions of LF1T in practice.

The rest of this paper is organized as follows. Section 2 reviews LF1T together with two previous versions of its algorithms. Section 3 describes the new LF1T algorithm based on BDDs and discusses its computational complexity. Section 4 shows experimental results of the new algorithm compared to the two previous versions of LF1T on learning Boolean networks.

2 Learning from 1-Step Transitions

We consider a first-order language and denote the Herbrand base (the set of all ground atoms) as $\mathcal{B}$. A (*normal*) *logic program* (NLP) is a set of *rules* of the form

$$A \leftarrow A_1 \wedge \cdots \wedge A_m \wedge \neg A_{m+1} \wedge \cdots \wedge \neg A_n \tag{1}$$

where A and A_i's are atoms ($n \geq m \geq 0$). For any rule R of the form (1), the atom A is called the *head* of R and is denoted as $h(R)$, and the conjunction to the right of $\leftarrow$ is called the *body* of R. We represent the set of literals in the body of R of the form (1) as $b(R) = \{A_1, \ldots, A_m, \neg A_{m+1}, \ldots, \neg A_n\}$, and the atoms appearing in the body of R positively and negatively as $b^+(R) = \{A_1, \ldots, A_m\}$ and $b^-(R) = \{A_{m+1}, \ldots, A_n\}$, respectively. The set of ground instances of all rules in a logic program P is denoted as $ground(P)$.

An (*Herbrand*) *interpretation* I is a subset of $\mathcal{B}$. For a logic program P and an Herbrand interpretation I, the *immediate consequence operator* (or T_P *operator*) [6] is the mapping $T_P : 2^{\mathcal{B}} \rightarrow 2^{\mathcal{B}}$:

$$T_P(I) = \{\, h(R) \mid R \in ground(P),\, b^+(R) \subseteq I,\, b^-(R) \cap I = \emptyset \}. \tag{2}$$

Definition 1 (Subsumption). For two rules R_1, R_2 of the form 1 with the same head, R_1 *subsumes* R_2 if there is a substitution θ such that $b^+(R_1)\theta \subseteq b^+(R_2)$ and $b^-(R_1)\theta \subseteq b^-(R_2)$. When R_1 *subsumes* R_2 and $|b(R_1)| < |b(R_2)|$, R_1 is more general than R_2 and R_2 is more specific than R_1.

We now review the *LF1T* algorithm developed in [1]. *LF1T* is an *anytime algorithm* that takes a set of state transitions $E \subseteq 2^{\mathcal{B}} \times 2^{\mathcal{B}}$ as input. The states transitions of E can be seen as (positive) examples/observations of transition of the system. From these transitions the algorithm learns a logic program P that represents the dynamics for E. To perform this learning process we can iteratively consider one-step transitions. In *LF1T*, the Herbrand base $\mathcal{B}$ is assumed

Algorithm 1. *LF1T*(E,P)

INPUT: $E \subseteq 2^{\mathcal{B}} \times 2^{\mathcal{B}}$: (positive) examples/observations and an NLP P
OUTPUT: An NLP P such that $J = T_P(I)$ holds for any $(I, J) \in E$.

while $E \neq \emptyset$ **do**
  Pick $(I, J) \in E$; $E := E \setminus \{(I, J)\}$
  for each $A \in J$ **do**
    $R_A^I := A \leftarrow \bigwedge_{B_i \in I} B_i \wedge \bigwedge_{C_j \in (\mathcal{B} \setminus I)} \neg C_j$
    AddRule(R_A^I, P)
end while
return P

to be finite. To construct an NLP for *LF1T* we can use a bottom-up method, which generates hypotheses by *generalization* from the most specific clauses to explain positive examples that have not been covered yet. The pseudo-code of *LF1T* is given in Algorithm 1. The *LF1T* algorithm can be used with or without an initial NLP P_0. Given only the examples E, *LF1T* is initially called by $\mathbf{LF1T}(E, \emptyset)$. If an initial NLP P_0 is given, $\mathbf{LF1T}(E, P_0)$ is called. *LF1T* first constructs the most specific rule R_A^I for each positive literal A appearing in $J = T_P(I)$ for each $(I, J) \in E$. We do not construct any rule to make a literal false. The rule R_A^I is then possibly generalized when another transition from E makes A true, which is computed by several generalization methods. The two generalization methods considered in [1] are based on *resolution*. In [1], naïve and ground resolutions are defined between two ground rules as follows. Let R_1, R_2 be two ground rules and l be a literal such that $h(R_1) = h(R_2)$, $l \in b(R_1)$ and $\bar{l} \in b(R_2)$. If $(b(R_2) \setminus \{\bar{l}\}) \subseteq (b(R_1) \setminus \{l\})$ then the *ground resolution of* R_1 *and* R_2 (*upon* l) is defined as

$$res(R_1, R_2) = \left(h(R_1) \leftarrow \bigwedge_{L_i \in b(R_1) \setminus \{l\}} L_i \right). \tag{3}$$

In particular, if $(b(R_2) \setminus \{\bar{l}\}) = (b(R_1) \setminus \{l\})$ then the ground resolution is called the *naïve resolution of* R_1 *and* R_2 (*upon* l). In this particular case, the rules R_1 and R_2 are said to be *complementary* to each other *with respect to* l. Both naïve resolution and ground resolution can be used as generalization methods of ground rules. For two ground rules R_1 and R_2, the naïve resolution $res(R_1, R_2)$ subsumes both R_1 and R_2, but the non-naïve ground resolution subsumes R_1 only. For example, suppose the three rules: $R_1 = (p \leftarrow q \wedge r)$, $R_2 = (p \leftarrow \neg q \wedge r)$,

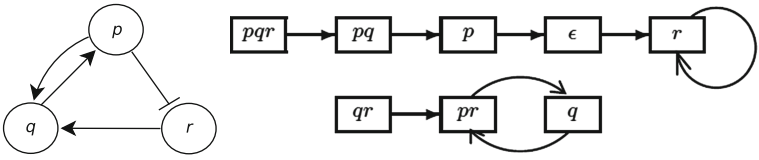


Fig. 1. A Boolean network N_1(left) and its state transition diagram (right)

$R_3 = (p \leftarrow \neg q)$, and their resolvent: $res(R_1, R_2) = res(R_1, R_3) = (p \leftarrow r)$. R_1 and R_2 are complementary with respect to q. Both R_1 and R_2 can be generalized by the naïve resolution of them because $res(R_1, R_2)$ subsumes both R_1 and R_2. On the other hand, the ground resolution $res(R_1, R_3)$ subsumes R_1 but does not subsumes R_3. In the first implementation of *LF1T* in [1], naïve resolution is used as a least generalization [17] method. This method is particularly intuitive from the ILP viewpoint, since each generalization is performed based on a least generalization operator. In [1], it is shown that for two complementary ground rules R_1 and R_2, the naïve resolution of R_1 and R_2 is the least generalization of them, that is, $lg(R_1, R_2) = res(R_1, R_2)$. When naïve resolution is used, *LF1T* needs an auxiliary set P_{old} of rules to globally store subsumed rules, which increases monotonically. Using naïve resolution, $P \cup P_{old}$ possibly contains all patterns of rules constructed from the Herbrand base $\mathcal{B}$ in their bodies. In the second implementation of *LF1T* of [1], ground resolution is used as an P_{old} alternative generalization method in **AddRule**. This replacement of resolution leads to a lot of computational gains, since the use of P_{old} is not necessary any more: all generalized rules obtained from $P \cup P_{old}$ by naïve resolution can be

Table 1. Execution of *LF1T* with ground resolution on step transitions of Fig. 1 where $pqr \rightarrow pq$ represents the state transition $(\{p, q, r\}, \{p, q\})$ [1].

Step	$I \rightarrow J$	Operation	Rule	ID	P
1	$pqr \rightarrow pq$	R_p^{pqr}	$p \leftarrow p \wedge q \wedge r$	1	1
		R_q^{pqr}	$q \leftarrow p \wedge q \wedge r$	2	1,2
2	$pq \rightarrow p$	R_p^{pq}	$p \leftarrow p \wedge q \wedge \neg r$	3	
		$res(3, 1)$	$p \leftarrow p \wedge q$	4	2,4
6	$p \rightarrow \epsilon$				
7	$\epsilon \rightarrow r$	R_r^{ϵ}	$r \leftarrow \neg p \wedge \neg q \wedge \neg r$	5	2,4,5
8	$r \rightarrow r$	R_r^{r}	$r \leftarrow \neg p \wedge \neg q \wedge r$	6	
		$res(6, 5)$	$r \leftarrow \neg p \wedge \neg q$	7	2,4,7
9	$qr \rightarrow pr$	R_p^{qr}	$p \leftarrow \neg p \wedge q \wedge r$	8	
		$res(8, 4)$	$p \leftarrow q \wedge r$	9	4,7,9
		R_r^{qr}	$r \leftarrow \neg p \wedge q \wedge r$	10	
		$res(10, 7)$	$r \leftarrow \neg p \wedge r$	11	2,4,7,9,11
10	$pr \rightarrow q$	R_q^{pr}	$q \leftarrow p \wedge \neg q \wedge r$	12	
		$res(12, 2)$	$q \leftarrow p \wedge r$	**13**	4,7,9,11,13
11	$q \rightarrow pr$	R_p^{q}	$p \leftarrow \neg p \wedge q \wedge \neg r$	14	
		$res(14, 1)$	$p \leftarrow q \wedge \neg r$	15	
		$res(15, 4)$	$p \leftarrow q$	**16**	7,11,13,16
		R_r^{q}	$r \leftarrow \neg p \wedge q \wedge \neg r$	17	
		$res(17, 7)$	$r \leftarrow \neg p \wedge \neg r$	18	
		$res(18, 11)$	$r \leftarrow \neg p$	**19**	**13,16,19**

obtained using ground resolution on P. By Theorem 3 of [1], using the naïve version, the memory use of the *LF1T* algorithm is bounded by $O(n \cdot 3^n)$, and the time complexity of learning is bounded by $O(n^2 \cdot 9^n)$, where $n = |\mathcal{B}|$. On the other hand, with ground resolution, the memory use is bounded by $O(2^n)$, which is the maximum size of P, and the time complexity is bounded by $O(4^n)$. Given the set E of complete state transitions, which has the size $O(2^n)$, the complexity of $\mathbf{LF1T}(E, \emptyset)$ with ground resolution is bounded by $O(|E|^2)$. On the other hand, the worst-case complexity of learning with naïve resolution is $O(n^2 \cdot |E|^{4.5})$.

Example 1. Consider the state transition in Fig. 1. By giving the state transitions step-by-step and using ground resolution the NLP $\{\#13, \#16, \#19\}$ is obtained in Table 1, where $\#n$ is the rule ID.

3 BDD Algorithms for *LF1T*

Now we present a new LF1T algorithm based on an efficient data structured inspired from OBDD and Zero-suppressed BDD. The novelty of our approach is the integration of LF1T operations into a BDD structure to perform ground resolution. In this approach, one BDD represents a set of rules that have the same head. Figure 2 show the evolution of the BDD that represents rules of p in Example 1: In this figure, the last schema of step 9 represents a BDD that contains two rules $p \leftarrow p \wedge q$ and $p \leftarrow q \wedge r$ which both have p as their head. The internal nodes of our data structure represent literals, and outgoing edges represent their polarity. In Fig. 2, the first BDD has one root node which represents the literal p and the edge between its child node q represents the fact that p is positive in the rule $p \leftarrow p \wedge q$. Like an OBBD, our structure respects a total variable ordering: if p, c are two nodes, c is a child of p and l_p, l_c their literals respectively, then $l_p < l_c$. If there is an edge between two nodes p, c that are not neighbors in the ordering, it means that all literals between them are absent from the rules encoded by paths including p and c. Like a ZDD, our BDD structure can have multiple root nodes, but only one leaf; it only represents positive rules. A root node always represents the first literal of one or multiple rules. The leaf node represents the end of all rules; it is the unique child of the last literal of every rule represented by the BDD. Usual BDD merging operations are not sufficient to perform the generalization operations of LF1T. In LF1T, these operations are equivalent to the use of naïve resolution without P_{old}. In Fig. 2, the generalization obtained in step 2 can be obtained by usual BDD merging operations: the node r has a positive and negative link to the same node (the leaf) and should be removed according to BDD merging operations. But the generalization obtained by ground resolution on step 9 cannot be obtained by usual BDD merging operations. To use ground resolution within a BDD structure we need to introduce specific merging operations. These operations have to ensure that the set of rules represented by a BDD is always *minimal* regarding ground resolution. In Fig. 2, the last BDD of each learning step respects

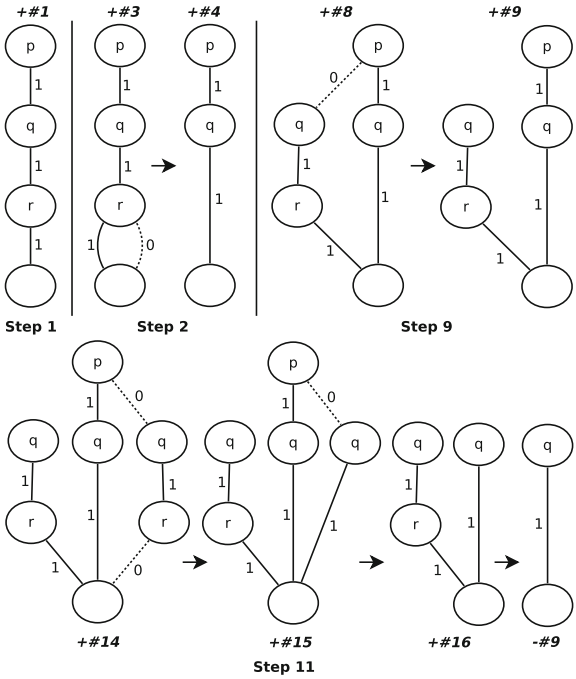


Fig. 2. Evolution of the BDD of p in Example 1, edge labelled by 0 represents negation, nodes without parent are roots and the empty node is the leaf. Last schema of each step represents the real state of the BDD; intermediate ones illustrate update operations. Step 1: from (pqr, pq) we learn $p \leftarrow p \wedge q \wedge r$. Step 2: from (pq, p) we learn $p \leftarrow p \wedge q \wedge \neg r$ and by resolution $p \leftarrow p \wedge q$. Step 9: from (qr, pr) we learn $p \leftarrow \neg p \wedge q \wedge r$ and by resolution $p \leftarrow q \wedge r$. Step 11: from (q, pr) we learn $p \leftarrow \neg p \wedge q \wedge \neg r$ which triggers two resolutions and a subsumtion to finish with $p \leftarrow q$.

this notion of minimality. Algorithm 2 describes our adaptation to BDD of the *addRule* operation of LF1T. This algorithm is an application to BDD of the previous version of LF1T based on ground resolution. Whenever a new rule is learned, the corresponding BDD is updated as follows: (1) check if the rule is subsumed, (2) generalize the rule, (3) remove subsumed rules, (4) insert the rule and (5) generalize the BDD. The details of each step is explained as follows.

Subsumption (step 1). To check if a rule is subsumed by a BDD, we have to check whether starting from a root and following the body of the rules allow us to reach the leaf of the BDD. If we reach the leaf then the rule is subsumed. Because we use ground resolution, if a rule is subsumed by the BDD it is useless to search for generalizations of that rule. Checking for such a generalization will only lead to generating a rule that is already in the BDD. Also, it cannot generalize any rules in the BDD: every generalization which can be triggered by this rule has already been found using the rules in the BDD that subsumes it.

Algorithm 2. **addRule**(R,B)

```
1: INPUT: a rule R and a BDD B
2: g: a set of rules
   // 1) Check if R is subsumed
3: for each root node r of B do
4:     if r.subsumes(R, 0) then return
   // 2) Generalizes R
5: for each root node r of B do
6:     if r.generalizes(R, 0) then restart the for loop
   // 3) Remove rules subsumed by R
7: l := the leaf node of B
8: l.clear(R, |R|, true)
   // 4) Insert R into the BDD
9: insert(R, B)
   // 5.1) Check generalization by R
10: g ← ∅
11: for each root node r of B do
12:     r.generalizations(R, 1, g)
   // 5.2) Add the generalizations generated by R
13: for each rules R_g of g do
14:     addRule(R_g)
```

Algorithm 3. **subsumes**(R, n) member function of a LF1T-BDD node N

```
1: INPUT: a rule R and an integer n
2: OUTPUT: a Boolean value

3: literal_N: literal of the node N
4: true_children: list of child nodes linked by a true edge
5: false_children: list of child nodes linked by a false edge
6: head: the head literal of R
   // 1) Terminal node
7: if is_terminal() AND variable = head then
8:     return true
   // 2) End of the rule
9: if n > |R| then
10:     return false
11: literal_R ← n^th literal of R
   // 3) LF1T-BDD rules are more generals
12: if literal_R > literal_N then
13:     return subsumes(R, n + 1)
14: literal_R ← n^th literal of R
   // 4) The rule is more general
15: if literal_R < literal_N then
16:     return false
   // 5) Same literal
17: if literal_R is positive then
18:     children ← true_children
19: else
20:     children ← false_children
21: for each child node c of children do
22:     if c.subsumes(R, n + 1) then
23:         return true
24: return false
```

Generalization of the new rule (step 2). To search for generalizations of the rules we use a similar search. However, each time we reach a node representing the current literal l of the rule, we check if the sub-BDDs subsume the complementary rule on l. If it is the case, we generalize the rule on this literal and restart the check for generalizations with the new rule.

Algorithm 4. generalizes(R, n) member function of a LF1T-BDD node N

```
1: INPUT: a rule R and an integer n
2: OUTPUT: a Boolean value
3: literal_N: literal of the node N
4: true_children: list of child nodes linked by a true edge
5: false_children: list of child nodes linked by a false edge
   // 1) The rule is more general than all rules of the node
6: if n > |R| then return false
   // 2) Terminal node
7: if is_terminal() then return false
   // 3) Check generalization on the current node
8: literal_R ← n^th literal of R
   // 3.1) The node is more general than the rule
9: while literal_N > literal_R do
10:     if subsumes(R, n) then
11:         R ← R \ literal_R // 3.1.1) The node subsumes the complementary rule
12:         return true
13:     n ← n + 1
        // 3.1.2) No more literal to generalize
14:     if n > |R| then return false
15: end while
   // 3.2) The rule is more general
16: if literal_N < literal_R then return false
   // 3.3) The sub-bdd possibly contains the complementary
17: same ← true_children
18: oposite ← false_children
19: if literal_R is positive then
20:     same ← false_children
21:     oposite ← true_children
   // 3.3.1) Search for complementary rules
22: for each child node c of oposite do
23:     if c.subsumes(R, n + 1) then // Complementary rules is subsumed
24:         R ← R \ literal_R
25:         return true
   // 4) Search for generalizations on next literal
26: for each child node c of same do
27:     if c.generalizes(R, n + 1) then
28:         return true
29: return false
```

Removal (step 3). To delete the rules subsumed by the new rule in the BDD, this time we start from the leaf. We follow the parents according to the rule until we check all corresponding parts of the BDD. If we reach the end of the rule, it means that a rule is subsumed. If we do not encounter a node with multiple children, we just have to delete the current node and *purge* the linked nodes: we recursively delete all parent nodes that have no more children and all children who have no more parents (those poor orphans). Otherwise, we come back to the first node with multiple children we encountered, cut the child edge we followed, and purge the child node in the same way as before.

Insertion (step 4). All operations we use on our BDDs are based on the manner in which we insert a rule into the structure. First of all, when adding a rule R to a BDD B we assume that R does not subsumes and is not subsumed by any rules of B and cannot be generalized by a rule of B using ground resolution (insured by step 1–3). To add a rule in the BDD we start by searching the common part of the beginning and the end of the body. From the leaf of the BDD, we climb to its parents following the rule from the end. If a parent node

Algorithm 5. clear(R, n, can_cut) member function of a LF1T-BDD node N

```
1: INPUT: R a rule, n an integer and can_cut a Boolean
2: OUTPUT: a Boolean value

3: literal_R: the n^th literal of R
4: unlink ← false
   // 1) Choice node
5: if #child > 1 then
6:     can_cut ← false
   // 2) Check parents
7: for each parent node p do
8:     literal_p ← the literal of p
   // 2.1) Parent is more general
9:     if literal_p < literal_R then
10:        if n = 1 AND is_terminal() then
11:            CONTINUE // 2.1.1) Not subsumed
12:        if !p.clear(R, n, can_cut) then
13:            CONTINUE
           // 2.1.2) Subsumed
14:        if can_cut then
15:            remove the link with p and delete p if it do not has child
16:            unlink ← true
17:            CONTINUE
18:        return true
           // 2.2) Rule is more general
19:    if literal_p > literal_R then
20:        if !p.clear(R, n, can_cut) then
21:            delete p if it do not has any parent
22:            CONTINUE // 2.2.1) Not subsumed
           // 2.2.2) Subsumed
23:        if can_cut then
24:            remove the link with p and delete p if it do not has any child
25:            unlink ← true
26:            CONTINUE
27:        return true
           // 2.3) Same literal
28:    if n > 0 AND !p.clear(R, n − 1, can_cut) then
29:        delete p if it do not has any parent
30:        CONTINUE
           // 2.3.2) Subsumed
31:    if can_cut then
32:        remove the link with p and delete p if it do not has any child
33:        unlink ← true
34:        CONTINUE
35:    return true
36: return false
```

has multiple children we do not follow it. Adding a parent to this node will generate more rules than only the one we want to represent. We stop when there is no parent that corresponds to the literal of the rule or when we reach the beginning of the rule. Let's call the last parent reached *last* and its literal l_{last}; *last* will be connected later to the new nodes created to represent the rule. Then, we search for a root node corresponding to the first literal. If such a root node does not exist, we create a new one, and then we create and link new nodes for all literals $l < l_{last}$ of the rules. Then, *last* becomes the child of the node most recently created. If a root node corresponds to the first literal of the rule to insert, we follow its children according to the rule body. We stop the descent when no nodes correspond to the rule body, and connect the most recent one we found to *last*. This insertion policy allows us to compile common parts of the

Algorithm 6. insert(R, BDD)

```
1: INPUT: a rule R and a BDD

2: starting: the set of starting nodes of BDD
3: literal: first literal of R
4: begin, end: BDD nodes
5: n ← 0
6: push ← false
   // 1) Bottom-up search for common part
7: end ← the last ancestor node reached following R from the corresponding terminal node
   // 2) Fact rule
8: if |R| = 0 then
9:     starting ← {terminalnode}
10: begin ← NULL
   // 2.1) Search common literal within the starting nodes
11: if a node r ∈ starting correspond to literal then
12:     begin ← r
   // 2.2) New starting
13: if begin = NULL then
14:     begin ← a new node corresponding to literal
15:     starting ← starting ∪ {begin}
16:     push ← true
17: current: bdd node pointer
18: make current points on begin
   // 3) Insertion of the rest of the body
19: while n ≤ |R| do
20:     n ← n + 1
        // 3.1) Link node reached
21:     if n > |R| OR the n^th literal of R is the one of end then
22:         connect current to end according to the polarity of literal
23:         return
24:     literal ← n^th literal of R
   // 3.2) construct new nodes for the rest of the rule
25:     if push then
26:         create a new node for literal
27:         connect the node to current according to the polarity of literal
28:         make current points on the new node
29:         CONTINUE
   // 3.3) Continue to follow the rule
30:     next ← NULL
31:     for each child nodes c of current according to previous literal polarity do
32:         if c has only one parent node AND correspond to literal then
33:             next ← c
34:             BREAK
   // 3.4) No more common literal
35:     if next = NULL then
36:         push = true
37:         n ← n − 1
38:         CONTINUE
   // 3.4) // Continue to follow the LF1T-BDD
39:     Make current point on next
40: end while
41: Connect end to begin according to the polarity of literal
```

rule body to save memory space. It ensures that a node with multiple children have only one parent and cannot have an ancestor with multiple ancestors. In our implementation, this property is exploited to enhance the efficiency of the subsumption and generalization checks of LF1T.

Generalization of BDDs (step 5). To search the generalizations made by the new rule, we start from the root node. Let l be the current literal we are

checking in the rule. When we reach a node whose literal corresponds to l or before it in the ordering, we just have to retrieve all rules subsumed by the rest of the new rules. These rules can all be generalized on the current node. We continue the search for generalizations on the children until we cannot follow the rule anymore. It is necessary to clear the BDD from subsumed rules before this operation in order to avoid a cascade of useless generalizations which lead to the rule we are inserting. In fact, let R_1, R_2 be two rules such that R_1 subsumes R_2 on l. Then R_1 can generalize R_2 on l because R_1 subsumes the complementary of R_2 on l.

Algorithm 7. generalizations(R, n,G)

```
1: INPUT: R a rule, n an integer, G a list of rules
2: OUTPUT: a Boolean value

3: literal_N: node literal
4: G', rules: set of rules
   // 1) End of the rule
5: if n > |R| then return
6: literal_R ← n^th literal of R
   // 2) Node is more general
7: if literal_N > literal_R then return
   // 3) Generalizations are possible on all children
8: if literal_N < literal_R then
9:     for each child node c do
10:        rules ← all rules subsumed by R in c
11:        G ← G ∪ {rules}
   // 2.2) Retrieve deeper generalizations
12:    for each child node c do
13:        G' ← ∅
14:        c.generalizations(R, n + 1, G')
15:        literal ← literal_N
16:        if the link with c is a negation then
17:            literal ← ¬literal_N
18:        for each rule r of G' do
19:            G ← G ∪ {(h(r) ← literal ∧ ⋀_{l∈b(r)} l)}
20:    return
   // 3) Same literal
21: for each child node c do
22:    // 3.1) Search complementary rules
23:    if the link with c has the same polarity as literal_R then
24:        rules ← all rules subsumed by R in c
25:        G ← G ∪ {rules}
26:    else
27:        // 3.2) Check deeper generalizations
28:        literal ← literal_N
29:        if the link with c is a negation then
30:            literal ← ¬literal_N
31:        G' ← ∅
32:        c.generalizations(R, n + 1, G')
33:        for each rule r of G' do
34:            G ← G ∪ {(h(r) ← literal ∧ ⋀_{l∈b(r)} l)}
```

Theorem 1. *Let n be the size of the Herbrand base $|\mathcal{B}|$. Using our dedicated BDD structure the memory complexity as well as the computational complexity of LF1T remain in the same order as the previous algorithm based on ground resolution:, i.e., $O(2^n)$ and $O(4^n)$, respectively. The proof is given as appendix.*

Table 2. Memory use and learning time of **LF1T** for Boolean networks up to 15 nodes with the alphabetical variable ordering

Name	# nodes	# rules	Naïve	Ground	BDD
Arabidopsis thalania	15	28	T.O.	40.8 MB/13.8 s	31.6 MB/2.8 s
Budding yeast	12	54	11 MB/361 s	4.6 MB/0.82 s	3.6 MB/0.188 s
Fission yeast	10	23	3.3 MB/5.2 s	0.8 MB/0.68 s	0.5 MB/0.24 s
Mammalian cell	10	22	4.7 MB/5.7 s	1 MB/0.76 s	0.5 MB/0.24v s

Table 3. Experimental results of 1000 runs of **LF1T** with random variable orderings

Name	min/max # rules	Average # rules	time	std deviation rules/ time
Arabidopsis thalania	29/962	227	4.31 s	183.03/0.538 s
Budding yeast	54/310	82	0.3 s	41.91/0.019 s
Fission yeast	23/45	24	0.04 s	3.08/0.003 s
Mammalian cell	22/22	22	0.03 s	0/0.007 s

4 Experiments

In this section, we evaluate our learning methods through experiments. We apply our new LF1T algorithms to learn Boolean networks. Here we run our learning program on the same benchmarks used in [1]. These benchmarks are Boolean networks taken from Dubrova and Teslenko [18], which include those networks for control of flower morphogenesis in *Arabidopsis thaliana*, budding yeast cell cycle regulation, fission yeast cell cycle regulation and mammalian cell cycle regulation. Like in [1], we first construct an NLP $\tau(N)$ from the Boolean function of a Boolean network N where each Boolean function is transformed to a DNF formula. Then, we get all possible 1-step state transitions of N from all $2^{|\mathcal{B}|}$ possible initial states I^0's by computing all stable models of $\tau(N) \cup I^0$ using the answer set solver `clasp` [19]. Finally, we use this set of state transitions to learn an NLP using our LF1T algorithm. Because a run of **LF1T** returns an NLP which can contain redundant rules, the original NLP P_{org} and the output NLP P_{LFIT} can be different, but remain equivalent with respect to state transition, that is, $T_{P_{org}}$ and $T_{P_{LFIT}}$ are identical functions.

Table 2 shows the memory space and time of a single LF1T run in learning a Boolean network for each problem in [18] on a processor Intel Core I7 (3610 QM, 2.3 GHz) with 4 GB of RAM. In the naïve, ground and BDD versions of LF1T the variable ordering is alphabetical. The time limit is set to one hour for each experiment. The gain of memory for the BDD version is up to 50 % for the two smaller benchmarks and around 20 % for the bigger ones. The main interest of our algorithm is shown by the gain in CPU time. For the *Arabidopsis thaliana* benchmark the input size is quite big: 2^{15} state transitions. Here, naïve version

of LF1T reaches the time out (T.O.) of one hour. On this big benchmark, using BDD, we need 80 % less CPU time than the previous ground resolution method. These results show that even if the BDD structure does not have a big impact on the whole memory space use, its particular structure allows it to perform LF1T operations faster than in the previous algorithms.

Table 3 show more precise experimental results on the BDD version of LF1T. This table shows the minimum, maximum and average number of rules in the output NLP of 1000 runs of LF1T with random variable ordering. The fifth column shows the average learning time and last one is the standard deviation over the number of rules and the one of learning time.

The standard deviation shows that the impact of variable ordering does not affect learning time very much, but it has a significant influence on the rules learned by LF1T. Although those output rules are all minimal with respect to subsumption among them, some are subsumed by original rules. If we consider the original NLP as a kind of optimal NLP in terms of the number of rules, the bigger NLPs learned by our BDD version are local optima where no ground resolutions can be applied among the rules of the NLP. This is because the resolution strategy of LF1T is to perform resolution only when it produces a generalized rule, so other kinds of resolution are not allowed. For example, from $R_1 = (p \leftarrow p \wedge q)$ and $R_2 = (p \leftarrow \neg q \wedge r)$, $R = (p \leftarrow p \wedge r)$ cannot be obtained in LF1T, since R subsumes neither R_1 nor R_2. Variable ordering has the same affect on the previous versions of LF1T.

5 Conclusion and Future Work

We proposed a new algorithm for learning from interpretation transitions based on a BDD-like structure. Using this data structure, we can reduce the memory space to represent NLPs learned by LF1T. Analysis of the worst-case computational complexity demonstrated that learning with this method is equivalent to the previous method. However, experimental comparison with previous LF1T algorithms showed that our method outperforms them in practice. Just a few remarks on learning non-ground NLPs; LF1T first learns ground rules then we apply well-known generalization techniques like *anti-instantiation* and least generalization. Extension of the BDD structure in this paper to the first-order case like [20] remains as a future work. Another possible outlook is an extension of LF1T algorithm to learn the dynamics of asynchronous systems.

A Appendix

A.1 Proof of Theorem 1

Proof. Let n be the size of the Herbrand base $|B|$. This n is also the number of possible heads of rules. Furthermore, n is also the maximum size of a rule, i.e. the number of literals in the body; a literal can appear at most one time in the body of a rule. For each head there are 3^n possible bodies: each literal

can either be positive, negative or absent of the body. From these preliminaries we conclude that the size of an NLP $|P|$ learned by *LF1T* is at most $n \cdot 3^n$. But thanks to ground resolution, $|P|$ cannot exceed $n \cdot 2^n$; in the worst case, P contains only rules of size n where all literals appear and there is only $n \cdot 2^n$ such rules. If P contains a rule with m literals ($m < n$), this rule subsumes 2^{n-m} rules which cannot appear in P. Finally, ground resolution also ensures that P does not contain any pair of complementary rules, so that the complexity is further divided by n; that is, $|P|$ is bounded by $O(\frac{n \cdot 2^n}{n}) = O(2^n)$.

In our approach, a BDD represents all rules of P that have the same head, so that we have n BDD structures. When $|P| = 2^n$, each BDD represents $2^n/n$ rules of size n and are bound by $O(2^n/n)$, which is the upper bound size of a BDD for any Boolean function [21]. Because BDD merges common parts of rules, it is possible that a BDD that represents $2^n/n$ rules needs less than $2^n/n$ memory space. In the previous approach, in the worst case $|P| = 2^n$, whereas in our approach $|P| \leq 2^n$. Our new algorithm still remains in the same order of complexity regarding memory size: $O(2^n)$.

Regarding learning, each operation has its own complexity. Let k be the place of a literal in the variable ordering so that for the starting node literal of a BDD $k = 0$. In our BDD, a node has at most $2 \cdot ((n-k)-1)$ children: $(n-k)-1$ positive and negative links to all literals which are superior to k in the ordering. Insertion of a rule is done in polynomial time; in the worst case, we insert a rule where only one literal that differs from the BDD. Because we follow only the first common literals, we have to check at most $2 \cdot ((n-k)-1)$ links on $n-1$ nodes, which belongs to $O(n^2)$.

Subsumption as well as generalization checks require exponential time. In the case of subsumption, in the worst case the BDD contains $2^n/n$ rules and the rule is not subsumed by any of them.

That means that we have to check every rule, and each check belongs to $O(n^2)$ so that the whole subsumption operation belongs to $O(n^2 \cdot 2^n/n) = O(2^n)$. To clear the BDD we have to perform the inverse operation. We always have to check the whole BDD, so if the size of the BDD is 2^n then the complexity of the whole clear check also belongs to $O(2^n)$.

To generalize the new rule we have to check if the BDD subsumes one of its complementary rules. Like for subsumption, in the worst case we have to check every rule. A rule can be generalized at most n times; for each generalization we have to check at most n complementary rules, so the complexity of a complete generalization belongs to $O(n^2 \cdot 2^n/n) = O(2^n)$. For the complexity of generalization of BDD rules we consider the inverse problem. In the worst case, every rule of the BDD can be generalized by the new one. Because the new rule does not cover any rules of the BDD, it can generalize each rule of the BDD at most one time. Then, we have at most $2^n/n$ possible direct generalizations on the whole BDD. In the worst case, each of them can be generalized at most $n-1$ times, and like before, for each generalization we have to check at most n complementary rules. If a rule is generalized n times it means that its body becomes empty, i.e. the rule is a fact, and it will subsume and clear the whole

BDD. Then, the complexity of a complete generalization of the BDD belongs to $O(2^n/n \cdot (n-1) \cdot n) = O(2^n)$.

Each time we learn a rule from a step transition we have to perform these four checks which have a complexity of $O(n^2 + 2^n + 2^n + 2^n) = O(2^n)$. From 2^n state transitions, *LF1T* can directly infer $n \cdot 2^n$ rules. Learning the dynamics of the entire input implies in the worst case $2^n \cdot 2^n$ operations which belong to $O(4^n)$. Using our dedicated BDD structure the memory complexity as well as the computational complexity of *LF1T* remains the same order as the previous algorithm based on ground resolution: respectively $O(2^n)$ and $O(4^n)$.

References

1. Inoue, K., Ribeiro, T., Sakama, C.: Learning from interpretation transition. Mach. Learn. (2013). doi:10.1007/s10994-013-5353-8
2. Muggleton, S., De Raedt, L., Poole, D., Bratko, I., Flach, P., Inoue, K., Srinivasan, A.: Ilp turns 20. Mach. Learn. **86**(1), 3–23 (2012)
3. Inoue, K.: Logic programming for boolean networks. In: Proceedings of the Twenty-Second International Joint Conference on Artificial Intelligence, vol. 2, pp. 924–930. AAAI Press (2011)
4. Inoue, K., Sakama, C.: Oscillating behavior of logic programs. In: Erdem, E., Lee, J., Lierler, Y., Pearce, D. (eds.) Correct Reasoning. LNCS, vol. 7265, pp. 345–362. Springer, Heidelberg (2012)
5. Van Emden, M.H., Kowalski, R.A.: The semantics of predicate logic as a programming language. J. ACM (JACM) **23**(4), 733–742 (1976)
6. Apt, K.R., Blair, H.A., Walker, A.: Towards a theory of declarative knowledge. In: Minker, J. (ed.) Foundations of Deductive Databases and Logic Programming, pp. 89–149. Morgan Kaufmann, Los Altos (1988)
7. Akers, S.B.: Binary decision diagrams. IEEE Trans. Comput. **100**(6), 509–516 (1978)
8. Bryant, R.E.: Graph-based algorithms for boolean function manipulation. IEEE Trans. Comput. **100**(8), 677–691 (1986)
9. Aloul, F.A., Mneimneh, M.N., Sakallah, K.A.: Zbdd-based backtrack search sat solver. In: Proceedings of the International Workshop on Logic Synthesis, Lake Tahoe, California (2002)
10. Minato, S., Arimura, H.: Frequent closed item set mining based on zero-suppressed bdds. Inf. Media Technol. **2**(1), 309–316 (2007)
11. De Raedt, L., Kimmig, A., Toivonen, H.: Problog: A probabilistic prolog and its application in link discovery. In: Proceedings of the 20th International Joint Conference on Artifical Intelligence, pp. 2468–2473 (2007)
12. Simon, L., Del Val, A.: Efficient consequence finding. In: International Joint Conference on Artificial Intelligence, vol. 17, pp. 359–370. Lawrence Erlbaum Associates Ltd. (2001)
13. Inoue, K., Sato, T., Ishihata, M., Kameya, Y., Nabeshima, H.: Evaluating abductive hypotheses using an em algorithm on bdds. In: Proceedings of the 21st International Jont Conference on Artifical Intelligence, pp. 810–815. Morgan Kaufmann Publishers Inc. (2009)
14. Bryant, R.E., Meinel, C.: Ordered binary decision diagrams. In: Hassoun, S., Sasao, T. (eds.) Logic Synthesis and Verification, pp. 285–307. Springer, New York (2002)

15. Bryant, R.E.: Symbolic boolean manipulation with ordered binary-decision diagrams. ACM Comput. Surv. (CSUR) **24**(3), 293–318 (1992)
16. Minato, S.: Zero-suppressed bdds for set manipulation in combinatorial problems. In: 30th Conference on Design Automation, pp. 272–277. IEEE (1993)
17. Plotkin, G.D.: A note on inductive generalization. Mach. Intell. **5**(1), 153–163 (1970)
18. Dubrova, E., Teslenko, M.: A sat-based algorithm for finding attractors in synchronous boolean networks. IEEE/ACM Trans. Comput. Biol. Bioinform. (TCBB) **8**(5), 1393–1399 (2011)
19. Gebser, M., Kaminski, R., Kaufmann, B., Schaub, T.: Answer Set Solving in Practice. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan and Claypool Publishers, San Rafael (2012)
20. Groote, J.F., Tveretina, O.: Binary decision diagrams for first-order predicate logic. J. Logic Algebraic Program. **57**(1), 1–22 (2003)
21. Liaw, H.T., Lin, C.S.: On the obdd-representation of general boolean functions. IEEE Trans. Comput. **41**(6), 661–664 (1992)

Accelerating Imitation Learning in Relational Domains via Transfer by Initialization

Sriraam Natarajan[1], Phillip Odom[1(✉)], Saket Joshi[2], Tushar Khot[3], Kristian Kersting[4], and Prasad Tadepalli[5]

[1] Indiana University Bloomington, Bloomington, USA
podom@umail.iu.edu
[2] Cycorp Inc, Austin, USA
[3] University of Wisconsin-Madison, Madison, USA
[4] Fraunhofer IAIS, New York, Germany
[5] Oregon State University, Corvallis, USA

Abstract. The problem of learning to mimic a human expert/teacher from training trajectories is called imitation learning. To make the process of teaching easier in this setting, we propose to employ transfer learning (where one learns on a source problem and transfers the knowledge to potentially more complex target problems). We consider multi-relational environments such as real-time strategy games and use functional-gradient boosting to capture and transfer the models learned in these environments. Our experiments demonstrate that our learner learns a very good initial model from the simple scenario and effectively transfers the knowledge to the more complex scenario thus achieving a jump start, a steeper learning curve and a higher convergence in performance.

1 Introduction

It is common knowledge that both humans and animals learn new skills by observing others. This problem, which is called *imitation learning*, can be formulated as learning a representation of a policy – a mapping from states to actions – from examples of that policy. Imitation learning has a long history in machine learning and has been studied under a variety of names including learning by observation [1], learning from demonstrations [2], programming by demonstrations [3], programming by example [4], apprenticeship learning [5], behavioral cloning [6], and some others. Techniques used from supervised learning have been successful for imitation learning [7]. We follow this tradition and investigate the use of supervised learning methods to learn behavioral policies. Our focus is on relational domains where states are naturally described by relations among an indefinite number of objects. Examples include real time strategy games such as Warcraft, regulation of traffic lights, logistics, and a variety of planning domains. A supervised learning method for imitation learning was recently proposed [8]. This approach assumes an efficient hypothesis space for the policy function, and learns only policies in this space that are closest to

G. Zaverucha et al. (Eds.): ILP 2013, LNAI 8812, pp. 64–75, 2014.
DOI: 10.1007/978-3-662-44923-3_5

Fig. 1. Wargus Scenarios *(left)* The two tower scenario whereproviding examples is easier. *(right)* The three tower scenario which is significantly more complicated and requires more training trajectories.

the training trajectories [9,10]. This approach is based on functional gradient boosting [11] where a set of relational regression trees [12] are used to compactly represent a complex relational policy. This approach was demonstrated to be successful in many problems.

One of the key assumptions in the proposed approach is that the policies can be generalized across the objects in the domain. While one of the advantages of a logical representation is the generalization capability, it is also quite possible that in several large problems, the optimal policies can vary greatly as the number of the objects in the domains can increase. In such cases, the learner has to be provided with new example trajectories to learn the policies. Since the complexity of the domain has increased, the number of trajectories required for learning can also increase significantly. For instance, consider the two scenarios presented in Fig. 1 where the goal is to defend the towers from being destroyed by the enemy units. In the left figure, there are two towers and two enemy and friendly footmen and archers. In the right figure, all the numbers increase by one. As we show empirically, the optimal policies for the two scenarios can be very different. More importantly, the number of trajectories required to converge to the optimal policy is higher in the case of the more complex scenario.

In order to train on such complex scenarios, we propose to employ transfer learning [13,14] for learning in a (simpler) source problem and then transferring the learned knowledge to a (more complex) target task. More precisely, we aim to employ transfer by initialization [15] where the models learned from the source task are used to initialize the models in the learning task. Following prior work [8], we perform search through the space of policies using functional gradient boosting but initialize the gradients with the models learned in the source task. Our hypothesis is that this initialization will allow the learner to explore more complex policy spaces that might not have been accessed easily if the search started out with uniform policies. We verify this claim empirically.

In summary, we consider the problem of imitation learning in relational domains where the optimal policies can be significantly different as the number

of objects in the domain increases. Generalization of policies is still a desired property as the properties of the objects themselves can change across situations with the same number of objects. When the number of objects change, we propose to employ transfer learning by initialization to initialize the gradients in the target task. We evaluate the hypothesis in a real time strategy game and show that we are able to achieve a jump start, faster convergence to a more optimal policy.

The rest of the paper is organized as follows: we introduce the background and the prior work on relational imitation learning next. We then present our transfer algorithm for initialization and evaluate the algorithm on a complex RTS game and conclude the paper by outlining some challenges for future work.

2 Background

An MDP is described by a set of discrete states $\mathbf{S}$, a set of actions $\mathbf{A}$, a reward function $r_s(a)$ that describes the expected immediate reward of action a in state s, and a state transition function $p^a_{ss'}$ that describes the transition probability from state s to state s' under action a. A policy, π, is defined as a mapping from states to actions, and specifies what action to execute in each state. In the imitation learning, we assume that the reward function is not directly obtained from the environment. Our input consists of S, A and supervised trajectories generated by a Markov policy. We try to match it using a parameterized policy.

3 Relational Imitation Learning

Following Ratliff et al. [16], we assume that the discount factor are absorbed into the transition probabilities and policies are described by $\mu \in \mathbf{G}$ where $\mathbf{G}$ is the space of all state-action frequency counts. We assume a set of features $\mathbf{F}$ that describe the state space of the MDP and the expert chooses the action a_i at any time step i based on the set of feature values $\langle f_i \rangle$ according to some function. For simplicity, we denote the set of features at any particular time step i of the jth trajectory as $\mathbf{f_i^j}$ and we drop j whenever it is fairly clear from the context.

The goal of our algorithm is to learn a policy that suitably mimics the expert. More formally, we assume a set of training instances $\{\langle \mathbf{f_i^j}, a_i \rangle_{i=1}^{m^j}\}_{j=1}^{n}$ that is provided by the expert. Given these training instances, the goal is to learn a policy μ that is a mapping from $\mathbf{f_i^j}$ to a_i^j for each set of features $\mathbf{f_i^j}$. The key aspect of our setting is that the individual features are relational i.e., objects and relationships over these objects. The features are denoted in standard logic notation where $p(X)$ denotes the predicate p whose argument is X. The problem of imitation learning given these relational features and expert trajectories can now be posed as a regression problem or a supervised learning problem over these trajectories.

In our previous work [8], we employed Functional-Gradient Boosting for learning relational policies. The goal is to find a policy μ that is captured using the trajectories (i.e., features $\mathbf{f_i^j}$ and actions a_i^j) provided by the expert,

i.e., the goal is to determine a policy $\mu = P(a_i|\mathbf{f_i};\psi)\ \forall a, i$ where the features are relational. These features could define the objects in the domain (squares in a gridworld, players in robocup, blocks in blocksworld, archers or footmen in a real-time strategy game etc.), their relationships (type of objects, teammates in robocup etc.), or temporal relationships (between current state and previous state) or some information about the world (traffic density at a signal, distance to the goal etc.).

We assume a functional parametrization over the policy and consider the conditional distribution over actions a_i given the features to be,

$$P(a_i|\mathbf{f_i};\psi) = e^{\psi(a_i;\mathbf{f_i})}/\sum_{a_i'} e^{\psi(a_i';\mathbf{f_i})}, \forall a_i \in \mathbf{A} \tag{1}$$

where $\psi(a_i;\mathbf{f_i})$ is the potential function of a_i given the grounding $\mathbf{f_i}$ of the feature predicates at state s_i and the normalization is over all the admissible actions in the current state. Formally, functional gradient ascent starts with an initial potential ψ_0 and iteratively adds gradients Δ_i. Here, Δ_m is the functional gradient at episode m and is

$$\Delta_m = \eta_m \times E_{x,y}[\partial/\partial\psi_{m-1} log\, P(y|x;\psi_{m-1})] \tag{2}$$

where η_m is the learning rate. Note that in Eq. 2, the expectation $E_{x,y}[..]$ cannot be computed as the joint distribution $P(\mathbf{x},\mathbf{y})$ is unknown (in our case, y's are the actions while x's are the features). Instead of computing the gradients over the potential function, the gradients are computed for each training example:

$$\Delta_m(a_i^j;\mathbf{f_i^j}) = \nabla_\psi \sum_j \sum_i log(P(a_i^j|\mathbf{f_i^j};\psi))|_{\psi_{m-1}} \tag{3}$$

These are point-wise gradients for examples $\langle \mathbf{f_i^j}, a_i^j \rangle$ on each state i in each trajectory j conditioned on the potential from the previous iteration (shown as $|_{\psi_{m-1}}$). Now this set of local gradients form a set of training examples for the gradient at stage m. The main idea in the gradient-tree boosting is to fit a regression-tree on the training examples at each gradient step [17]. The idea of functional gradient boosting is presented in Fig. 2.

The functional-gradient w.r.t $\psi(a_i^j;\mathbf{f_i^j})$ of the likelihood for each example $\langle \mathbf{f_i^j}, a_i^j \rangle$ is given by:

$$\frac{\partial \log P(a_i^j|\mathbf{f_i^j};\psi)}{\partial\psi(\hat{a}_i^j;\mathbf{f_i^j})} = I(a_i^j = \hat{a}_i^j|\mathbf{f_i^j}) - P(a_i^j|\mathbf{f_i^j};\psi) \tag{4}$$

where $\hat{a}_i^j$ is the action observed from the trajectory and I is the indicator function that is 1 if $a_i^j = \hat{a}_j^i$ and 0 otherwise. The key feature of the above expression is that the functional-gradient at each state of the trajectory is dependent on the observed action $\hat{a}$. If the example is positive (i.e., it is an action executed by the expert), the gradient $(I - P)$ is positive indicating that the policy should

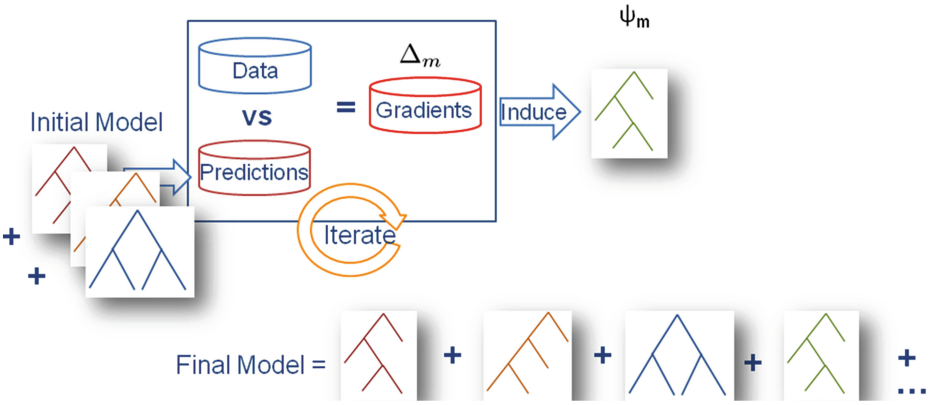


Fig. 2. Relational FGB. This is similar to the standard FGB where trees are induced in stage-wise manner; the key difference being that the trees are relational regression trees. To compute the predictions, a query, x is applied to each tree in turn, and the numerical values at the leaf reached in each tree are summed to obtain $\psi(x)$.

increase the probability of choosing the action. On the contrary if the example is a negative example (i.e., for all other actions), the gradient is negative implying that it will push the probability of choosing the action towards 0.

Following prior work [18–20], we used *Relational Regression Trees* (RRTs) [12] to fit the gradient function at every feature in the training example [8]. Hence the distribution over each action is represented as a set of RRTs on the features. These trees are learned such that at each iteration the new set of RRTs aim to maximize the likelihood. Hence, when computing $P(a(X)|f(x))$ for a particular value of state variable X (say x), each branch in each tree is considered to determine the branches that are satisfied for that particular grounding (x) and their corresponding regression values are added to the potential. For example, X could be a particular unit in Wargus. or a certain block in the blocksworld.

4 Relational Transfer

In this work, we extend the previous work in imitation learning by employing the ideas for *inductive transfer* [13]. While the previous approach was able to achieve generalization in a imitation learning setting, the generalized policies might not be sufficient in some other variations of the problems. For instance, in the scenarios considered in Fig. 1, the optimal policies for the three tower defense scenario can be significantly different from the easier task of two tower scenario. Also, since the three tower case is a harder task, as we show empirically, learning in this setting might require more example trajectories from the expert. In such cases, it is easier to transfer the knowledge gained from the two tower scenario to the three tower case for initialization and then improve upon the knowledge by learning in the three tower case. This will enable the learner to: (a) learn a better policy than the one generalized from the two tower scenario and (b) converge to

the optimal policy faster in the three tower scenario i.e., from fewer trajectories when compared to learning with no knowledge.

Table 1. Transfer Learning Algorithm

function TRANSFER(T_{source},T_{target})
  $\mathbf{\Lambda}_s$ = TIL({},T_{source}) ▷ Learn with source Trajectories
  $\mathbf{\Lambda}_t$ = TIL(Λ_s,T_{target}) ▷ Use learned models and learn on Target Trajectories
  return $\mathbf{\Lambda}_t$
end function
function TIL($\mathbf{\Lambda}$, *Trajectories* T)
  $\mathbf{\Lambda}_0 = \mathbf{\Lambda}$
  for $1 \leq k \leq |\ \mathbf{\Lambda}|$ **do** ▷ Iterate through each action
    for $1 \leq m \leq M$ **do** ▷ M gradient steps
      $S_k := GenExamples(k; T; \Lambda^k_{m-1})$
      $\Delta_m(k) := FitRRT(S_k; L)$ ▷ Gradient
      $\Lambda^k_m := \Lambda^k_{m-1} + \Delta_m(k)$ ▷ Update models
    end for
    $P(A = k|\mathbf{f}) \propto \psi^k$
  end for
return $\mathbf{\Lambda}$
end function
function GENEXAMPLES(k, T, Λ)
  $S := \emptyset$
  for $1 \leq j \leq |T|$ **do** ▷ Trajectories
    for $1 \leq i \leq |S^j|$ **do** ▷ States of trajectory
      Compute $P(\hat{a}^j_i = k|\mathbf{f}^{\mathbf{j}}_{\mathbf{i}})$ ▷ Probability of user action being the current action
      $\Delta_m(k; \mathbf{f}^{\mathbf{j}}_{\mathbf{i}}) = I(\hat{a}^j_i = k) - P(\hat{a}^j_i = k|\mathbf{f}^{\mathbf{j}}_{\mathbf{i}})$
      $S := S \cup [(\hat{a}^j_i, \mathbf{f}^{\mathbf{j}}_{\mathbf{i}}), \Delta(\hat{a}^j_i; \mathbf{f}^{\mathbf{j}}_{\mathbf{i}}))]$ ▷ Update relational regression examples
    end for
  end for
return S ▷ Return regression examples
end function

The form of the functional gradients facilitate easy transfer. Since they perform gradient descent in function space, we can initialize the models (ψ_0) for the three tower scenario with some of the trees learned in the two tower scenario. Conceptually, this is essentially the same as using the result of the first few gradient steps in the source problem while learning in the target problem. After initializing the gradient ascent with the initial set of trees, we propose to learn new set of trees in the target task that build upon the initial model. As mentioned earlier, this initial set of trees for the ψ_0.

To perform learning in the target task, the trajectories must be weighted given the initial model. Similar to Eq. 4, we compute the value of $I(a^j_i = \hat{a}^j_i|\mathbf{f}^{\mathbf{j}}_{\mathbf{i}}) - P(a^j_i|\mathbf{f}^{\mathbf{j}}_{\mathbf{i}}; \psi_0)$ for each action of each trajectory, i.e., the weight of each observed action is the difference between the indicator function of that action and the marginal probability of that action given the initial potential function. Once

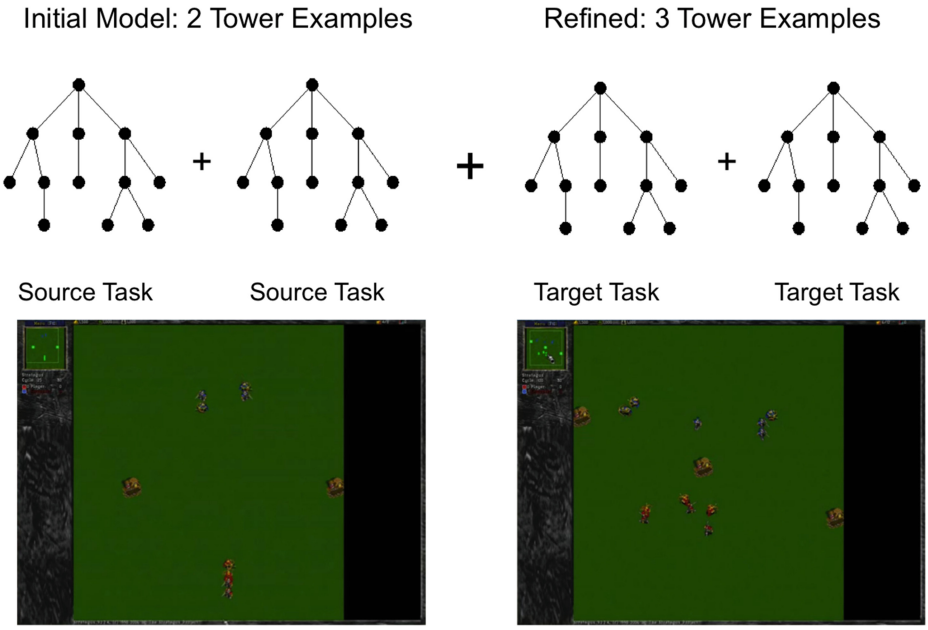


Fig. 3. Proposed transfer approach. The key idea is to learn a small set of trees from the source task and use them to initialize the RFGB algorithm for the harder task.

these weights are computed for the given trajectory, they serve as the examples for learning new set of trees. This idea is presented in Fig. 3. First we learn a few set of trees in the source task and then use them to initialize the models in the target task. We then learn a new set of trees in the target task.

Our proposed transfer learning algorithm is presented in Algorithm 1. The function *Transfer* is the main algorithm that takes as input trajectories from the source and the target tasks. The algorithm first calls the *TIL* function (which stands for $Tree - basedImitationLearning$) with an empty potential function (empty set of trees) and the source trajectories. The *TIL* function then learns a set of trees in the source task. The *TIL* function is the same one presented in [8] with the modification that it can use an initial set of trees. For each action (k), it generates the examples for our regression tree learner (called using function *FitRRT*) to get the new regression tree and updates its model (Λ_m^k). This is repeated up to a pre-set number of iterations M (typically, $M = 20$). We found empirically that increasing M has no effect on the performance as the example weights nearly become 0 and the regression values in the leaves are close to 0 as well. Note that the after m steps, the current model Λ_m^k will have m regression trees each of which approximates the corresponding gradient for the action k. These regression trees serve as the individual components ($\Delta_m(k)$) of the final potential function.

Once the set of trees have been learned in the source task, a subset of those trees (typically we use 20 trees in our experiments), is then used as the initial

model for the target task and the *TIL* function is called with this initial set and the trajectories. The function then returns a new set of trees which are then used for evaluating in the target task. It must be mentioned that when choosing to act in the target task, inference over the actions is performed using *all* the trees (the initial set of source trees plus the target trees). It is easy to see that we cannot ignore the transferred trees since they form the first step of the gradient ascent when learning the policy in the target domain.

The proposed approach is closely related to the idea of modular policies of Driessens [21]. He observed that the use of functional gradients to represent policies allows us to separate the gradient updates to different subtasks of the agent's task. Thus, we can create separate potential functions for each part of the task and the natural addition operator of functional gradients allows then to obtain the final policy which is essentially a sum of different regression trees. We extend the above to transfer learning where we consider an initial set of trees for a different task (or potentially a subtask) and a new set of trees are then learned for the new task. Hence, combining these two ideas, it is possible to learn a higher level policy in a hierarchy by transferring from the lower level subtasks.

5 Experiments

We present the empirical evaluation of our proposed algorithm on a real-time strategy game. We are particularly interested in the following questions:
Q1: How do the transferred models compare against the models that are generalized using the relational imitation learning algorithm?
Q2: How do the transferred models compare against the models that are learned directly on the target task with no prior models from source task?
Experimental Setup: Stratagus is an open-source real-time strategy (RTS) game engine written in C based off the Warcraft series of games. Like all RTS games, it allows multiple agents to be controlled simultaneously in a fully observed setting, making an ideal test bed for imitation learning. A java client was written, revised at Oregon State University[1], to connect to the Stratagus game engine via a socket connection. The client collects all of the game information from the game engine and can issue detailed commands to all units of a player in the game. This client allows for the learned policies to be executed directly in the game environment as opposed to simulation creating more realistic performance metrics.

The setting in which transfer is being tested is the tower defense scenario shown in Fig. 1. The map used for the experiments consisted of 6×6 grid world. Our scenarios consist of two opposing teams-one attacking, one defending- each with two kinds of units. Footman have more health but must be close to an enemy to attack them while ranged archers are easily killed but can attack from a distance. Towers exist on the map in set locations. The defending team must prevent the attacking team from destroying the towers while the attacking team must destroy as many towers as they can. The defending team must divide its

[1] http://beaversource.oregonstate.edu/projects/stratagusai

units among the various towers to prevent one tower from falling while another is being saved. This dynamic creates complex policies.

Predicates	Description
friendlyobject	The type of defending unit
enemyobject	The type of attacking unit
dead	Enemy unit that is dead
locationId	Location of friendly unit
strength	Strength (hit points) of friendly unit
distance	Distance of a friendly unit to a tower
enemyattower	Tower that enemy unit is attacking
attacking	Enemy unit that a friendly unit was attacking in the previous state

Fig. 4. Features that describe the state in the two scenarios. We omit the arguments of the predicates for brevity.

We used the following features to describe the state: the strength (high, medium, low) and location of all friendly units, the type (footman, archer) of all units in the game, which tower each enemy unit is currently attacking, and the enemy unit that all friendly units were attacking the previous state. Friendly units are unaware of the strength of enemy units or their exact location. The full set of information given at every state is included in Fig. 4. The actions available to the friendly units are to move to a location and attack a particular unit. The nature of the objects and the relationships between the objects in this game naturally allow for a relational representation. Each type of unit (footman or archers) shares traits such as their attacking range and their total health so certain policy rules will naturally apply to all units of that type. As mentioned earlier, the goal of this experiment is to learn to protect two towers in a source scenario and transfer the learned knowledge to a target scenario. The attacking team's strategy is as follows: At the start of each game, each member of the attacking team randomly selects a tower to attack. However, if approached by an enemy archer or footmen, they will change their target to eliminate the opposing player's offensive units. After destroying one tower, they will randomly select another tower to attack until there are no towers left and the game ends. The goal of the game is to defend the towers. The game ends if either the enemy team manages to destroy all of the towers or the friendly team kills all of the enemy units. The number of friendly and attacking units vary between the source and target scenarios. In the source scenario, there are two footmen and two archers while in the target scenario, there are three footmen and three archers. In both the scenarios, the towers cannot defend themselves.

Results: We used two performance metrics that are based on the number of towers saved. The 3-tower win percentage is the percentage of games in which all three towers were saved. This is a difficult task because it requires the friendly team to defend all three towers simultaneously; else one tower may fall while they are all defending the others. The 1-tower win percentage is the easier metric, only requiring 1 of the 3 towers to be saved. We used randomly selected samples of 30, 50, 70, 100, and 150 3-tower games from a pool of approximately 1000 expert games for training in the target scenario. For the source scenario, we used

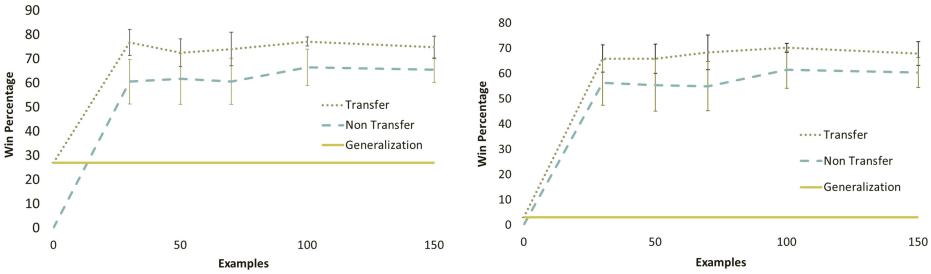


Fig. 5. Results of saving at least one tower (left) and saving all three towers (right). The transferred models dominate the one learned from two towers (generalization) and the one learned on three towers without an initial policy (non-transfer)

100 games for training. This experiment was repeated 10 times and the average percentage of winnings games were computed.

We used three models for evaluation – the transfer model, the non-transferred model which learns only on the target task and the generalization model which learns only on the source and not the target task. We learned 20 trees on the source task and learned a further 20 on the target task. Hence, the final transfer model had 40 trees while the other two models had 20 trees each.

The results are presented in Fig. 5 corresponding to saving at least one tower and three towers respectively. As can be seen from the figures, the transfer model is superior to both the generalization and the pure imitation learner in the twin tasks. Also, the results follow the original transfer learning goals of jump start, steeper learning curve and better convergence. It is fairly clear that in defending at least one tower, the use of the initial models from the source scenario provides a bigger jump start than in defending all towers. On the other hand, when saving all towers, the jump start is not fairly high but the learning curve is steeper. It appears that the use of initial models allow for the learner in the target scenario to explore a space of policies that might not have been otherwise reached from an uniform policy. Similarly, the difference between transfer and non-transfer models is higher in the one tower case than saving all the towers though the difference in statistically significant in both the cases. Also, it must be mentioned that even when the non-transferred models were provided with a lot more trajectories, it is not able to match the performance of the transferred model. This suggests that using initial policies in some cases is more useful than obtaining more expert trajectories. It would be interesting to evaluate these results in other domains as well.

In summary, our experiments answer both the questions affirmatively in that the transfer models dominate both the original imitation learning models.

6 Discussion and Conclusion

We address the issue of sample complexity in imitation learning settings. In scenarios where the expert's time is expensive/valuable and we have access to only

a few training examples from the expert our approach is to divide the expert's time between simple (smaller domain size) and harder (larger domain size) problems. Although policies induced from the simpler problem training instances can be employed to solve the larger domain via relational generalization, in scenarios we provide (such as Wargus) this does not translate to better performance. We have presented transfer learning by using the simpler policy as our initial models and building an updatable relational model by learning from the harder examples. We observe not only a superior performance to generalization but also a drastic reduction in the sample complexity as compared with the naive method of directly inducing a model on the complex examples.

Imitation learning encounters two major problems when dealing with large state spaces. First, assuming a tabular representation of the policy to be learned is likely to exceed memory due to the large state and action space. Second, it can only make use of a limited amount of expert traces compared to the excessive amount of possible traces. The implicit feedback gained by the expert's traces on the best action to take in a state might be so sparse that a well-generalizing policy will only be discovered slowly.

The first problem can be solved using relational imitation learning (RIL) for structural domains. However, the problem of sparse feedback has not been addressed by RIL yet. For relational reinforcement learning, there is a compelling and simple solution to this problem: inject traces of execution of a reasonable policy for the task at hand [22]. Unfortunately, this does not work for imitation learning. The input consists already of traces of execution of a reasonable policy, namely the policy of the expert. Thus, we do not gain anything despite enlarging the training set as can be seen from our results. The non-transfer model seemed to have converged to a inferior policy. To overcome this problem, we intuitively propose to inject traces of a policy of a reasonably well related task. Specifically, we directly inject the complete "related" policy into a functional gradient boosting approach to RIL. This appears to be an interesting result in that sometimes prior policies have a better impact on the performance compared to more trajectories. Our immediate challenge is to validate this hypothesis on other more complex domains. Another interesting direction is the possibility of employing active learning methods for extracting the best complex examples given the initial model thereby further improving on the performance of transfer.

Acknowledgments. SN and PO thank Army Research Office grant number W911NF-13-1-0432 under the Young Investigator Program. SN and TK gratefully acknowledge the support of the DARPA DEFT Program under the Air Force Research Laboratory (AFRL) prime contract no. FA8750-13-2-0039. Any opinions, findings, and conclusion or recommendations expressed in this material are those of the authors and do not necessarily reflect the view of the DARPA, AFRL, or the US government. SJ was supported by a Computing Innovations Postdoctoral Fellowship. KK was supported by the Fraunhofer ATTRACT fellowship STREAM and by the European Commission under contract number FP7-248258-First-MM. PT acknowledges the support of ONR grant N000141110106.

References

1. Segre, A., DeJong, G.: Explanation-based manipulator learning: acquisition of planning ability through observation. In: Conference on Robotics and Automation (1985)
2. Argall, B., Chernova, S., Veloso, M., Browning, B.: A survey of robot learning from demonstration. Robot. Auton. Syst. **57**, 469–483 (2009)
3. Calinon, S.: Robot Programming By Demonstration: A Probabilistic Approach. EPFL Press, Boca Raton (2009)
4. Lieberman, H.: Programming by example (introduction). Commun. ACM **43**, 72–74 (2000)
5. Ng, A., Russell, S.: Algorithms for inverse reinforcement learning. In: ICML (2000)
6. Sammut, C., Hurst, S., Kedzier, D., Michie, D.: Learning to fly. In: ICML (1992)
7. Ratliff, N., Bagnell, A., Zinkevich, M.: Maximum margin planning. In: ICML (2006)
8. Natarajan, S., Joshi, S., Tadepalli, P., Kersting, K., Shavlik, J.: Imitation learning in relational domains: a functional-gradient boosting approach. In: IJCAI (2011)
9. Khardon, R.: Learning action strategies for planning domains. Artif. Intell. **113**, 125–148 (1999)
10. Yoon, S., Fern, A., Givan, R.: Inductive policy selection for first-order mdps. In: UAI (2002)
11. Friedman, J.H.: Greedy function approximation: a gradient boosting machine. Ann. Stat. **29**, 1189–1232 (2001)
12. Blockeel, H.: Top-down induction of first order logical decision trees. AI Commun. **12**(1–2), 119–120 (1999)
13. Pan, S., Yang, Q.: A survey on transfer learning. IEEE Trans. Knowl. Data Eng. **22**, 1345–1359 (2010)
14. Al-Zubi, S., Sommer, G.: Imitation learning and transferring of human movement and hand grasping to adapt to environment changes. In: Human Motion. Computational Imaging and Vision, vol. 36, pp. 435–452 (2008)
15. Mehta, N., Natarajan, S., Tadepalli, P., Fern, A.: Transfer in variable-reward hierarchical reinforcement learning. Mach. Learn. **73**(3), 289–312 (2008)
16. Ratliff, N., Silver, D., Bagnell, A.: Learning to search: functional gradient techniques for imitation learning. Auton. Robots **27**, 25–53 (2009)
17. Dietterich, T.G., Ashenfelter, A., Bulatov, Y.: Training conditional random fields via gradient tree boosting. In: ICML (2004)
18. Gutmann, B., Kersting, K.: TildeCRF: conditional random fields for logical sequences. In: Fürnkranz, J., Scheffer, T., Spiliopoulou, M. (eds.) ECML 2006. LNCS (LNAI), vol. 4212, pp. 174–185. Springer, Heidelberg (2006)
19. Natarajan, S., Khot, T., Kersting, K., Guttmann, B., Shavlik, J.: Gradient-based boosting for statistical relational learning: the relational dependency network case. Mach. Learn. **86**, 25–56 (2012)
20. Kersting, K., Driessens, K.: Non-parametric policy gradients: a unified treatment of propositional and relational domains. In: ICML (2008)
21. Driessens, K.: Non-disjoint modularity in reinforcement learning through boosted policies. In: Multi-disciplinary Symposium on Reinforcement Learning (2009)
22. Driessens, K., Dzeroski, S.: Integrating guidance into relational reinforcement learning. Mach. Learn. **57**(3), 271–304 (2004)

A Direct Policy-Search Algorithm for Relational Reinforcement Learning

Samuel Sarjant[1(✉)], Bernhard Pfahringer[1], Kurt Driessens[2], and Tony Smith[1]

[1] The University of Waikato, Waikato, New Zealand
{sarjant,bernhard,tcs}@waikato.ac.nz
[2] Maastricht University, Maastricht, The Netherlands
kurt.driessens@maastrichtuniversity.nl

Abstract. In the field of relational reinforcement learning — a representational generalisation of reinforcement learning — the first-order representation of environments results in a potentially infinite number of possible states, requiring learning agents to use some form of abstraction to learn effectively. Instead of forming an abstraction over the state-action space, an alternative technique is to create behaviour directly through policy-search. The algorithm named CERRLA presented in this paper uses the cross-entropy method to learn behaviour directly in the form of decision-lists of relation rules for solving problems in a range of different environments, without the need for expert guidance in the learning process. The behaviour produced by the algorithm is easy to comprehend and is biased towards compactness. The results obtained show that CERRLA is competitive in both the standard testing environment and in MS. PAC-MAN and CARCASSONNE, two large and complex game environments.

1 Introduction

Reinforcement Learning (RL) is a subfield of machine learning in which an *agent* interacts with an *environment* using *actions* and receives numerical *reward* as feedback [1]. An agent selects actions using a *policy*: a decision-making structure that produces an action when given observations for the current *state*. As the field of RL matures, the need for more advanced testing environments increases as algorithms become progressively 'smarter.' In order to represent these complex environments, the field of Relational Reinforcement Learning (RRL) came about, where environments could be represented by variable numbers of objects and relations [2–4]. This representation allows environments with any number of objects and relations to be represented with the same common formalism.

Attempting to learn the value function directly can be impossible in environments consisting of an infinite number of states so a common technique in RRL is to learn an approximate value function for estimating the utility of an action in every state [5–7]. These methods attempt to approximate the value function for every state (and action) in the environment and use the values to *extract* a policy that selects actions with the largest predicted reward. However,

G. Zaverucha et al. (Eds.): ILP 2013, LNAI 8812, pp. 76–92, 2014.
DOI: 10.1007/978-3-662-44923-3_6

in order to learn a reasonably accurate approximate value function, the agent must first discover which states are rewarding. This can be achieved by methods such as random exploration, which is ineffective in complex environments, or using some form of initial guidance such as injecting an expert trace of behaviour [8], which requires some intervention from an external agent (such as a human or pre-existing model).

Instead of approximating values for every state, then extracting a greedy policy from these values, an alternative is to attempt to learn the policy directly. *Policy-search* methods have some advantages over value-function approximation methods: policies are typically smaller than value-functions, as they only need to represent which action to take in a state; and changes in the reward received will not change the policy if the best action for a state remains constant. A disadvantage is that policy-search methods typically require a large number of episodes for training, though this value is usually unaffected by the scale of the environment. Existing policy-search RRL algorithms such as Grey and Gapi have been shown to learn optimal policies in the Blocks World, but testing has been limited to smaller environments [9,10].

Like Grey and Gapi, the algorithm presented in this paper performs direct policy-search where an agent's policy is represented as a decision-list of condition-action rules. The Cross-Entropy Method (CEM), originally developed by [11], is an optimisation algorithm already shown to be effective for learning agent behaviour [12–14], as well as a number of other domains, such as clustering, control and navigation, and continuous optimisation to name a few [15]. We use the CEM's probabilistic optimisation approach to control the policy-creation aspect of the algorithm.

This paper describes Cross-Entropy Relational Reinforcement Learning Algorithm (Cerrla), an application of the CEM for learning behaviour in a range of different relational environments. The CEM is used to identify the best combination of relational condition-action rules acting as the agent's policy. Rules are created in a top-down manner by gradually specialising useful rules in search of better policies. The policies produced by Cerrla should be effective, concise, and easily understood by a human.

To test the general applicability of the algorithm to different environments, we evaluate Cerrla on three separate environments: the standard RRL Blocks World environment, where it achieves excellent results regardless of problem scale; and two game environments, Ms. Pac-Man and Carcassonne, which provide large state spaces and complex action-interactions. Included with the results are example policies produced by Cerrla for each environment.

2 Related Work

The Cerrla algorithm was originally inspired by the algorithm presented in [12]: a Ms. Pac-Man playing agent that uses the Cross-Entropy Method (CEM) to generate and test rule-based policies. The algorithm begins with a set of 42 hand-coded candidate rules that are used to create a rule-based, deterministic

policy of maximum size 30. The algorithm learns better behaviour by randomly sampling rules for each of the 30 possible positions in the policy and then adjusting the rule sampling probabilities to produce better performing policies more frequently. This paper also looked at randomly created rules, which did not perform as well as the hand-coded rules, but still performed well. This algorithm formed the core design behind CERRLA, though CERRLA has since expanded upon this design in the following aspects: CERRLA starts without *any* rules or policy size restrictions and creates new rules as it learns; CERRLA learns relational rules/policies for a range of relational environments rather than the single Ms. Pac-Man environment; and CERRLA learns using an iterative CEM, rather than population-based, to quickly integrate newly created rules.

CERRLA uses a similar learning process as the two policy-search RRL algorithms GREY and GAPI: use an evolutionary algorithm to learn a rule-based policy. Both GREY and GAPI use a standard genetic algorithm implementation [16], treating entire policies as chromosomes to be mutated for the recombination operation. However, mutation operations also take place on the rules within each policy by randomly adding/removing literals or replacing variables with constants. Both algorithms were tested in the standard BLOCKS WORLD environment, where they each successfully created goal-achieving policies, but the policies sometimes included useless or detrimental rules. The cross-entropy method employed by CERRLA actively reduces the likelihood of including useless rules, and the rule creation process is bottom-up, resulting in fewer useless literals in rules. GAPI was also tested in a 'gold-finding' environment, which is a step towards more complicated environments. We take this further by testing CERRLA in real-world games that are challenging for humans as well as AI.

The FOXCS system creates rule-based policies by utilising the XCS system for the first-order setting [17,18]. Learning is achieved by maintaining an expected reward and accuracy value for that reward for every rule. These values are used to identify useful rules and guide rule mutation (using standard mutation operations). CERRLA also maintains a value for every rule, but the need to maintain an expected value for every rule limits the scalability of FOXCS. This can be seen in [19], where FOXCS performs worse as the size of the BLOCKS WORLD environment grows. Because CERRLA uses probabilities of *utility* for each rule, the learning rate remains roughly proportional to the number of rules, rather than size of the environment.

Two more RRL systems also deserve a mention, as they perform well on large environments. The LRW-API approach learns a policy by iteratively performing batches of *policy rollouts* as an *approximate policy iteration* algorithm [20]. At any given state, the algorithm updates the Q-value for every action by creating w policy trajectories of length h to identify the most advantageous action to perform (most difference between expected reward and actual reward). The algorithm is able to offset the cost of the rollouts by beginning learning in artificially smaller environments defined as the state reached after n random actions from the initial state. By beginning in small environments and increasing n, the algorithm can quickly scale to large and complex environments. The main

Relational State Observations:

block(a)	*thing(b)*	*clear(fl)*	*above(a, b)*	*height(b, 1)*
block(b)	*thing(c)*	*highest(a)*	*above(a, fl)*	*height(c, 1)*
block(c)	*thing(fl)*	*on(a, b)*	*above(b, fl)*	*height(fl, 0)*
floor(fl)	*clear(a)*	*on(b, fl)*	*above(c, fl)*	
thing(a)	*clear(c)*	*on(c, fl)*	*height(a, 2)*	

Valid Actions:

move(a, c) *move(a, fl)* *move(c, a)*

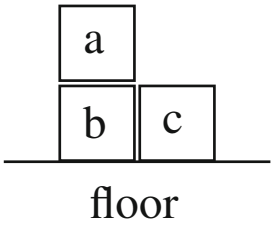


Fig. 1. A 3-block Blocks World state observation example. a is on b which is on the *floor*, and c is also on the *floor*.

disadvantage of this method is the 'controlled experiment' assumption that the world model can be accessed at any state, whereas RRL world models are typically 'black boxes' that only allow a single action per state.

The NPPG algorithm also uses bootstrapping to overcome the problem of large environments in the form of policy-gradient *boosting* [21]. The algorithm iteratively builds regression models to approximate the value function (using batches of episode traces as training data) by layering the model on top of existing models, where each regression model is created to cover the examples previous models do not adequately cover. Each model also receives a weighting to reflect the utility of its predictions. The algorithm performs very well on a 10-block Blocks World environment (thus far, the 10-block environment was typically too large for value-based algorithms to tackle directly), though it does use expert traces to seed the learning with positive examples. A downside to NPPG is the output behaviour is largely incomprehensible, as it is made up of many different weighted models.

3 Terminology

The relational representation used throughout this paper is as follows: a *constant* c is a lowercase symbol representing a uniquely named object of a given *type* (e.g. *thing*, *block*, *enemy*). A *variable* V is an uppercase symbol representing an abstract object. A *term* t may be either a constant or variable. A *predicate* p is a relation acting upon one or more objects with specifically *typed* arguments. Environments are defined by *state predicates* $P_s = \{p_{s,1}, \ldots, p_{s,n}\}$ (which include *type predicates* $P_t = \{p_{t,1}, \ldots, p_{t,n}\}$) and *action predicates* $P_a = \{p_{a,1}, \ldots, p_{a,n}\}$. An *atom* $p(t_1, \ldots, t_n)$ is a predicate with terms for arguments. A *ground atom* $p(c_1, \ldots, c_n)$ only uses constants for arguments. A *goal variable* G_i is a special indexed variable representing one of the constants in the goal and is substituted by the appropriate goal constant when the variable is evaluated. The *anonymous variable* '?' represents any object.

An environment's state observations consist of a complete description of the state $s = \{p_{s,1}(c_{1,1}, \ldots, c_{1,n}), \ldots, p_{s,m}(c_{m,1}, \ldots, c_{m,n})\}$ and the current available actions $A(s) = \{p_{a,1}(c_{1,1}, \ldots, c_{1,n}), \ldots, p_{a,m}(c_{m,1}, \ldots, c_{m,n})\}$. Any constants

$$clear(G_0), clear(G_1), block(G_0) \rightarrow move(G_0, G_1)$$
$$above(X, G_1), clear(X), floor(Y) \rightarrow move(X, Y)$$
$$above(X, G_0), clear(X), floor(Y) \rightarrow move(X, Y)$$

Fig. 2. An optimal Blocks World *onAB* policy generated by Cerrla. Note that G_0 and G_1 are parameterisable goal constants.

directly related to the environment's goal are also provided to the agent. Accompanying each state observation is a reward value. There is no guarantee that an environment will be defined by a Relational Markov Decision Process (RMDP) [3]; the learning agent must simply select an action without absolute knowledge of what state will follow.

3.1 Blocks World

The Blocks World environment is the most commonly used testing environment in the RRL and planning fields due to its simple, but fundamental dynamics. The Blocks World environment will be used for examples in the following sections. The environment consists of a number of blocks stack on top of each other, all stacked on the floor. An agent may move a block on to another block, or on to the floor. A Blocks World state is described by: $P_s = \{clear(Thing), on(Block, Thing), \ above(Block, Thing), highest(Block), height(Thing, \mathbb{N})\}$ and type predicates $P_t = \{thing, block, floor\}$. The only action predicate is $P_a = \{move(Block, Thing)\}$. Figure 1 shows an example state for a 3-block Blocks World with the listing of all state and action observations for the current state.

Commonly used goals include the *onAB* goal: place block G_0 onto block G_1 (G_0 and G_1 are randomly defined blocks at the start of every episode); and the *stack* goal: stack every block into a tower. Each episode runs for a maximum of $2n$ steps, where n is the number of blocks. The reward received is 1 if the goal is achieved in minimal steps, or some value linearly distributed between 1 and 0 inversely proportional to the number of steps over the minimum the agent took to complete the goal.

4 CERRLA Algorithm

The Cross-Entropy Relational Reinforcement Learning Algorithm (Cerrla) generates policies for a RRL agent by combining a number of randomly sampled condition-action rules into a single decision-list policy (Fig. 2).[1] When the policy is evaluated against the current state observations, it produces one or more actions depending on which rule conditions match the observations. Each rule

[1] Source code, experiment files and videos of Cerrla in action can be found at www.samsarjant.com/cerrla.

is sampled from a separate distribution of similar rules, where the probability of the rule is dynamically adjusted based on the rule's utility. This process is known as the Cross-Entropy Method (CEM) and it forms the backbone of the *probability optimisation* aspect of CERRLA, though some modifications to the core CEM are described in Sect. 4.3.

The other aspect of CERRLA is the *rule discovery* aspect (Sect. 4.2). Rules are created in CERRLA by learning general conditions for every possible action, then gradually applying specific specialisation operators to empirically useful rules to create more complex 'child' rules.

4.1 Cross-Entropy Method

The CEM is a population-based optimisation algorithm, similar to evolutionary algorithms, that uses guided random sampling from one or more distributions to produce effective combinatorial solutions to a given problem. Distributions may be discrete or continuous, but in the context of CERRLA, the distributions are sets of condition-action rules, each with a corresponding probability of being sampled. A sample is a combination of sampled rules represented in a decision-list format (the order of the rules is defined in Sect. 4.3). For a comprehensive exploration of the CEM, see [15].

The CEM is essentially composed of two repeating steps:

1. Generate and test N samples from a distribution of data.
2. Update the distribution such that the top subset of the sampled data is more likely to be generated again in the next iteration.

At every iteration, each distribution produces a randomly selected rule. These rules combine to form a deterministic policy which is then evaluated against the environment which returns the total reward received under the policy. The subset of policies that receive the greatest reward within a population of policies are used to update the rule sampling probabilities, such that the policies' rules are more likely to be sampled again. Intuitively, the algorithm works as follows: in the early stages, the algorithm does not perform any worse than random guessing, but as it gathers samples, it shapes the distribution such that guessing becomes more and more biased towards high-value samples.

Formally, using CERRLA as context, the CEM algorithm is as follows: the algorithm begins with K distributions of rules ($D_k \leftarrow \{r_{1,k}, \ldots, r_{n,k}\}$), where each rule $r_{i,k}$ has a corresponding sampling probability $p_{i,k} \in [0,1] : \sum_{j=1}^{n} p_{j,k} = 1$ (a distribution is typically uniform at the outset). N samples are generated ($\mathbf{X} \leftarrow \{\mathbf{x}_1, \ldots, \mathbf{x}_N\}$), where each sample contains a single randomly sampled rule from each distribution (arranged via some heuristic), and tested with evaluation function $f(\mathbf{x})$ (for CERRLA, this is the total reward received). The samples are then sorted into descending order according to $f(\mathbf{x})$ and all samples with $f(\mathbf{x}) \geq \gamma_{t+1}$ are extracted as 'elite samples' E_{t+1}, where γ_{t+1} is equal to the value of the N_E^{th} sorted sample. The minimum number of elite samples is defined as $N_E \leftarrow \rho \cdot N$ (typically $\rho \leftarrow 0.05$). Note that there may be more than N_E elite samples, as multiple samples could have a value equal to the threshold.

The observed distribution D'_k is then calculated for every distribution D_k as the frequency of rules within the elite samples:

$$p'_{j,k} \leftarrow \Big(\sum_{\mathbf{x}_i \in E_{t+1}} \begin{Bmatrix} 1 \text{ if } \mathbf{x}_{i,k} = r_{j,k} \\ 0 \text{ otherwise} \end{Bmatrix} \Big) \Big/ |E_{t+1}| \tag{1}$$

meaning $p'_{j,k}$ is equal to the proportion of elite samples containing rule $r_{j,k}$ from distribution D_k.

The distribution probabilities are then updated using a step-size parameter α (typically α is between 0.4 and 0.9 [15]) to smoothly modify the distribution probabilities:

$$p_{j,k,t+1} \leftarrow \alpha \cdot p'_{j,k} + (1 - \alpha) \cdot p_{j,k,t} \tag{2}$$

This sample-update loop repeats until some convergence measure is reached: (1) a predefined finite number of iterations have passed; (2) probabilities have converged to 0 or 1; or (3) the distributions sufficiently match the observed elites distributions for a given number of iterations.

4.2 Rule Discovery

CERRLA begins the rule discovery process by first calculating the Relative Least General Generalisation (RLGG) for each action in the environment [22]. Further rules are created in a top-down fashion by iteratively specialising empirically useful rules. Each rule is simplified with inferred simplification rules to remove redundant conditions and identify illegal condition combinations, removing illegal and semantically-identical redundant rules from the set of possible rules.

Learning the RLGG. The first rules created by CERRLA are the RLGG rules for every action in the environment. Because *all* state information is available, the RLGG operation only needs to perform *lgg* operations (background knowledge is considered to be part of the state). Each rule encodes the least general set of conditions that have been observed to be true whenever the rule's action is available for the current state. Contrary to the RLGG process in [22], this RLGG process uses a *lossy inverse substitution* to only record information relevant to the rule's action; other information is discarded. That is, the process only considers literals containing constants found in the action, or defined in the environment goal. This lossy inverse substitution focuses learning on the core literals involved in the rule's action, reducing the search space of rules; but has the drawback of losing potentially useful information.

Given a state s and a set of valid actions $A(s) = \{a_1, \ldots, a_n\} : a_i = p_{a,i}(c_{i,1}, c_{i,2}, \ldots)$, the RLGG conditions for each action are defined as:

$$r^a_{RLGG,t} = lgg\big(r^a_{RLGG,t-1}, \theta^{-1} r(s, a_i)\big)$$

where $r^a_{RLGG,t-1}$ is the existing RLGG rule for action predicate p_a and $r(s, a_i)$ is a rule composed of atomic action a_i and every state observation containing

one or more of the constants in a_i. The RLGG of the two rules uses a lossy inverse substitution defined by the current arguments of the atomic action a_i, such that $\theta_{a_i}^{-1} = \{c_{i,1}/X, c_{i,2}/Y, \ldots\}$. Any non-numerical constants not included in $\theta_{a_i}^{-1}$ are replaced by the anonymous variable '?' which can be substituted for any constant when evaluated. Numerical constants are replaced by free variables representing the number. The resulting rule encodes a near-least general set of conditions (due to lossy inverse substitution) required for taking action p_a.

Example 1. Referring to Fig. 1, the RLGG calculation process for the three valid actions *move(a, c)*, *move(a, fl)*, *move(c, a)* is described in the following example, processing one rule at a time (beginning with $t = 1$):

$r_{move(a,\ c)}$ = *block(a)*, *block(c)*, *thing(a)*, *thing(c)*, *clear(a)*, *clear(c)*, *on(a, b)*, *on(c, fl)*, *above(a, b)*, *above(a, fl)*, *above(c, fl)* → *move(a, c)*
$\theta^{-1}_{move(a,\ c)} = \{a/X, c/Y\}$
$r^{move}_{RLGG,1}$ = *block(X)*, *block(Y)*, *thing(X)*, *thing(Y)*, *clear(X)*, *clear(Y)*, *on(X, ?)*, *on(Y, ?)*, *above(X, ?)*, *above(Y, ?)* → *move(X, Y)*

This is already very close to the actual RLGG; only the conditions *block(Y)*, *on(Y, ?)*, and *above(Y, ?)* are not always true, as evidenced in the following example:

$r_{move(a,\ fl)}$ = *block(a)*, *floor(fl)*, *thing(a)*, *thing(fl)*, *clear(a)*, *clear(fl)*, *on(a, b)*, *on(b, fl)*, *on(c, fl)*, *above(a, b)*, *above(a, fl)*, *above(b, fl)*, *above(c, fl)* → *move(a, fl)*
$\theta^{-1}_{move(a,\ fl)} = \{a/X, fl/Y\}$
$r^{move}_{RLGG,2}$ = *block(X)*, *thing(X)*, *thing(Y)*, *clear(X)*, *clear(Y)*, *on(X, ?)*, *above(X, ?)* → *move(X, Y)*

This rule is in fact the RLGG for the Blocks World *move* action, so there is no need to describe the process for the final action of the state (as the rule cannot generalise any further). Many of the conditions in this rule are redundant with respect to other facts though (e.g. *on(X, ?)* is always true if *above(X, ?)* is true) and can be removed using the simplification rules described in the rule simplification section. The simplified rule is:

$$r^{move}_{\text{RLGG},\ 2} = \textit{clear(X)}, \textit{clear(Y)}, \textit{block(X)} \rightarrow \textit{move(X, Y)} \tag{3}$$

Rule Specialisation. CERRLA uses three specialisation operators: (1) *additive*, (2) *goal-replacement*, and (3) *range-splitting*. Each specialisation operator creates a new rule with more specialised conditions.

(1) *Additive specialisation* specialises a rule by adding a condition to it. Instead of adding an arbitrary condition to a rule r^a with any possible argument bindings, CERRLA restricts the set of specialisation conditions to those that include action-related conditions and have been *observed* to be true (but not in the RLGG conditions) when action a is available. Each specialisation

condition is recorded with inversely-substituted arguments (replace all action-related constants with variables *and* replace all goal-related constants with goal variables). Negated specialisation conditions are also used to specialise a rule.

(2) For every constant in the environment goal, *goal-replacement* replaces all occurrences of one of the action's arguments with an indexed 'goal variable' G_i representing one of the constants in the environment's current goal. After replacing the variable, if all conditions containing the goal variable in the rule have previously been observed to be possible (i.e. ensure the rule's conditions can feasibly be met), the specialised rule is valid, otherwise it is invalid and discarded.

(3) *Range splitting* creates specialised rules by splitting an existing range (or a variable representing a number) into up to five overlapping sub-ranges: the *lower half*, the *upper half*, a *central half*, and if applicable, a negative sub-range (*lower bound* to 0), and a positive sub-range (0 to *upper bound*). Except when 0 is a bound, the range bounds are expressed as variable fractions of the observed minimum and maximum bounds so the rule does not need to change when the bounds change. For instance, a range from $[-4, 8]$ would be split into the following subranges: $[-4, 2]$, $[2, 8]$, $[-1, 5]$, $[-4, 0]$, and $[0, 8]$ (all represented as variable fractions of the original range).

Example 2. The RLGG rule from the prior example (Eq. 3) can be specialised into the following example rules:

$$r_1^{move} = clear(X), clear(Y), block(X), floor(Y) \rightarrow move(X,\ Y)$$
$$r_2^{move} = clear(X), clear(Y), block(X), not(highest(X)) \rightarrow move(X,\ Y)$$
$$r_3^{move} = clear(G_0), clear(Y), block(G_0) \rightarrow move(G_0, Y)$$
$$r_4^{move} = clear(X), clear(Y), block(X), on(X, G_0) \rightarrow move(X,\ Y)$$

As there are no variables representing numerical arguments, the range splitting specialisation does not produce any rules. Note that some rules contain redundant conditions and can be simplified (see the following section).

Rule Simplification. To avoid creating illegal rules or rules containing redundant conditions, CERRLA also infers *simplification rules* for the environment.

Simplification rules are created using the RLGG method in Sect. 4.2 but instead of calculating the RLGG relative to each *action predicate* p_a, it is calculated relative to each *state predicate* p_s. The resulting set of RLGG conditions encode the relationship between each state predicate p_s and all other predicates, producing implication rules in the form $A \Rightarrow B$, such that condition-action rules containing both A and B remove condition B. Furthermore, rules containing A and $\neg B$ are marked as illegal rules and are deleted from CERRLA's distributions. If $B \Rightarrow A$ as well, the rule is instead recorded as an equivalence rule $A \Leftrightarrow B$, such that condition-action rules containing B replace it with A.

The RLGG can also be calculated for the set of atoms that are *never true* when p_s is present as well. The variable representation of the lossy inverse substitution $\theta^{-1}_{p_s(t_1,\ldots,t_n)}$ results in a finite set of possible atoms given the set of

state predicates (as all constants are replaced by variable $X, Y, \ldots$ or '?'). By removing the inversely substituted true atoms from this set, the RLGG of *never true* atoms relative to p_s can also be calculated to produce simplification rules involving negated conditions.

Example 3. Some of the simplification rules created for BLOCKS WORLD include:

$$on(X,\ Y) \Rightarrow above(X,\ Y), \qquad floor(Y) \Leftrightarrow clear(Y), on(X,\ Y),$$
$$highest(X) \Rightarrow not(on(?,\ X)), \qquad block(X) \Leftrightarrow on(X,\ ?)$$
$$block(X) \Rightarrow not(floor(X))$$

Whenever the right-hand side of a simplification rule subsumes a rule's literal(s), they are removed (or replaced by the substituted left-hand side literals for equivalence rules). Note that '?' explicitly represents the anonymous variable when performing rule simplification (i.e. all '?' are treated as constants).

4.3 Policy-Search Process

The CERRLA algorithm (Algorithm 1) uses a modified version of the CEM to produce policies. Each distribution of candidate rules is sampled to produce a single rule, which is included into the policy in a specific order. The policy is then deterministically evaluated throughout the episode to produce the agent's actions. The policies that received the largest reward form an 'elite distribution' which the rule sampling probabilities are adjusted toward, such that rules in an elite policy are more likely to be sampled again.

Algorithm 1. Pseudocode summary of CERRLA.

```
Initialise the distribution set 𝔻                    ▷ Initially empty. Learns RLGG rules to start
repeat
    Generate a policy π from 𝔻                       ▷ Sample ≤ 1 rule from each D in 𝔻 and order into policy
    Evaluate π, receiving average reward R           ▷ Run three times and average
    Update elite samples E with sample π and value R ▷ If π is good, add to E
    Update 𝔻 using E   ▷ Adjust probabilities for each D in 𝔻 to be closer to distribution in E
    Specialise rules (if 𝔻 is ready)                 ▷ If a rule is highly probable, branch it to a new D
until 𝔻 has converged                                ▷ Until no more branching is possible
```

Instead of a population-based approach, CERRLA uses an *online variation* of the CEM, similar to the CEM variant in [23], which updates the distributions after every sample. Instead of sampling batches of N samples, the algorithm maintains a sliding window of N samples, such that the elites E consist of the best samples from the last N samples (instead of the best samples in a batch). The online variation is able to adapt to a changing number of rules and distributions as CERRLA creates new specialisations.

Initialisation. CERRLA begins with no rules or distributions ($\mathbb{D} \leftarrow \{\}$) but quickly creates distributions by firstly observing the RLGG for every available action, then creating all immediate specialisations of the RLGG (as described in Sect. 4.2). Each of these rules acts as a *seed* for a new distribution, such that a distribution consists of a uniform distribution of the seed rule and all immediate specialisations of the seed rule. Each D also has two properties: the probability that a rule from D is present within a policy, $p(D) \in [0, 1]$ (initially $p(D) \leftarrow 0.5$); and the average relative positions of sampled rules within generated policies, $q(D) \in [0, 1]$, where 0 represents the first position and 1 represents the last (initially $q(D) \leftarrow 0.5$).

The number of rules within D is written as $|D|$ and $KL(D)$ represents the Kullback-Leibler (KL) size, or distance from the uniform distribution, of D such that:

$$KL(D) \leftarrow \max \left[|D| \cdot \left(1 - \sum_{r \in D} p_r \log_{|D|}(|D| \cdot p_r) \right), 1 \right] \tag{4}$$

A uniform distribution has $KL(D) = |D|$, but a distribution with a single high probability rule (e.g. $p_j \geq 0.95$) has $KL(D) = 1$.

Policy Generation. A policy π is generated by sampling a rule from every distribution D in $\mathbb{D}$. For each D, a rule will only be sampled from D with probability $p(D)$. The position of the sampled rule is determined by the relative ranking to all other rules in the policy. This relative value $relQ(D)$ is sampled from a normal distribution with parameters $q(D)$ for the mean, and $1-|1-2p(D)|$ as the standard deviation. Rules are ordered in the policy in ascending order according to their respective $relQ(D)$. When D is initialised with $p(D) = 0.5$, the relative position of each sampled rules varies wildly, but as the $p(D)$ converges towards 0 or 1, the relative position fluctuates less.

The policy π is evaluated at every decision step using the current state observations. Starting with the first rule in the policy, each rule is evaluated as a query on the state. If a substitution(s) for the rule's conditions is found, the rule's action is returned with the substituted values applied (there may be more than one substitution). If multiple actions are returned by the agent, an environment-specific selection mechanism (e.g. sort by distance) selects and resolves one of the actions and advances to the next state. The value of a policy ($f(\pi)$) is equal to the average total-episodic-reward received for a given number of episodes (we use three episodes in experiments).

Updating. After a policy π has been evaluated, it is added to the floating set of elite samples E if $f(\pi) \geq \gamma$ (the worst elite sample). Before this occurs, any elite samples that have existed for greater than N iterations are removed to ensure the elites represent recent samples. Rather than using a fixed elite sample size, CERRLA dynamically changes the number of elites N_E based on the state of the current distributions:

$$N_E \leftarrow \max \left\lceil \underbrace{\arg\max_{D \in \mathbb{D}} \left(KL(D) \cdot p(D) \right)}_{\text{largest distribution}}, \quad \underbrace{\sum_{D \in \mathbb{D}} p(D)}_{\text{sum distribution means}} \right\rceil \tag{5}$$

where $N \leftarrow N_E/\rho$, as with the regular CEM. This equation results in a large N_E when CERRLA contains new, undertested distributions, and a smaller N_E when rule probabilities and distribution means are close to 0 or 1.

After potentially adding the sample to the elite samples, the distributions are updated. The increased frequency of updates requires a reduced update step-size to 'smooth' the learning rate. Instead of α, the single-step update parameter $\alpha_1 \leftarrow \alpha/N$ is used, which is approximately equal to the standard α update rate.

A restriction applies to updates: a distribution D is only updated if it has produced $C \cdot |D|$ samples. This reduces update bias towards early samples, and provides enough samples for a distribution to be fairly represented in the elite samples. $C = 3$ is used in experiments as it provides 95 % coverage of all rules in a uniform distribution [24].

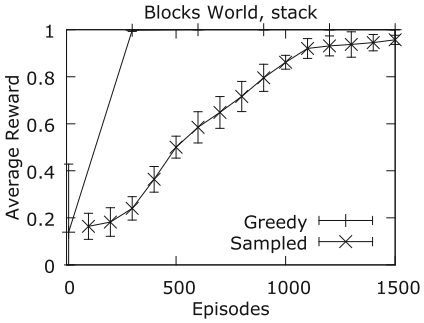
The update process consists of updating every candidate rule distribution in $\mathbb{D}$ (as per Sect. 4.1), as well as their properties $p(D)$ and $q(D)$. Only the rules that fired throughout the sample's testing episodes matter, therefore unused rules (and their distributions) are not included in the update and, implicitly, negatively updated. Each update operation uses Eq. 2 to adjust the values in a step-wise manner. The observed value for $p(D)$ is simply the proportion of policies containing D within the elites. The observed value for $q(D)$ is the average relative position of rules from D within the elite policies, where 0 represents the first position and 1 represents the last.

Rule Exploration. When a rule has a sufficiently high probability, it 'branches,' creating new rules with more specialised conditions in hopes of finding better rules. This process is triggered when a distribution's $KL(D) \leq \delta \cdot |D|$, where $\delta = \min\left[(d+1)^{-1}, p(D)\right]$, representing the splitting point with respect to the *depth* d of the distribution or number of branches from the RLGG distribution. The highest probability rule r from D is removed from D, and is used to seed a new distribution, populating the distribution with r and all immediate specialisations of r. The only restriction to this exploration process is if r originally seeded D, in which case no branching occurs.

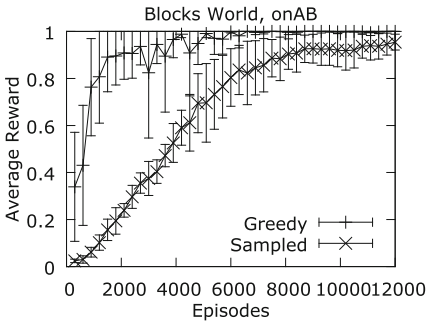
Convergence. CERRLA is considered converged when each distribution is considered converged. A distribution is converged when the sum divergence of the rule probabilities between updates is less than $\alpha_1 \cdot \beta$ (a convergence threshold). Alternatively, experiments can specify a fixed number of training episodes. Upon convergence, the current best elite sample is also output as the best solution.

(a) Performances comparison in BLOCKS WORLD with 3–10 blocks for various RRL algorithms. Note that some figures are approximate readings from a graph.

Algorithm	Average Reward		# of Training Episodes ($\times 10^3$)	
	stack	*onAB*	*stack*	*onAB*
CERRLA	**1.00**	**0.99**	**1.60**	**10.30**
P-RRL [2]	1.00	0.65	0.05	0.05
RRL-TG [5]	0.88	0.92	0.50	12.50
RRL-TG (P) [5]	1.00	0.92	30.00	30.00
RRL-RIB [5]	0.98	0.90	0.50	2.50
RRL-KBR [5]	1.00	0.98	0.50	2.50
TRENDI [6]	1.00	0.99	0.50	2.50
TREENPPG [21]	—	0.99	—	2.00
MARLIE [7]	1.00	0.98	2.00	2.00
FOXCS [18]	1.00	0.98	20.00	50.00



(b) *Stack* goal in 100-block BLOCKS WORLD.



(c) *OnAB* goal in 100-block BLOCKS WORLD.

Fig. 3. CERRLA's performance in the BLOCKS WORLD environment.

5 Evaluation

CERRLA is evaluated in three separate environments: BLOCKS WORLD, MS. PAC-MAN, and CARCASSONNE.[2] Each environment presents a different problem for the agent to solve, though all share a common RRL representational format. All reported results are averaged across 10 separate experiments. Each figure contains two performances: *sampled performance*, the reward received during training; and *greedy performance*, the average reward received for 100 testing policies (not included in training time/number of episodes) using the current best elite sample.

5.1 Blocks World

Figure 3a compares CERRLA's BLOCKS WORLD performance against other RRL algorithms for both *stack* and *onAB* goals. A summary of current RRL algorithms can be found in [4]. Like most RRL algorithms it is able to learn optimal or near-optimal behaviour in both goals, though it requires more episodes

[2] An additional environment detailed in [25] was omitted for space reasons.

than the value-based algorithms to do so (training time for each goal was 6 and 34 s respectively). Figure 3b and c demonstrates a powerful property of CERRLA: even in a BLOCKS WORLD of 100 blocks, the learning rate remains roughly constant (training time is only ∼50 times longer compared to exponential state increase). In most other compared approaches, learning rate deteriorates as the number of blocks increase.

The effects of the simplification rules are tested by comparing two 12,000 fixed-episode experiments for the *onAB* goal: one using rule simplification, the other not using rule simplification (denoted in brackets). CERRLA initially creates 244 (1,150) rules spread over 17 (32) distributions. At 12,000 training episodes, the algorithm has 330 (1,405) rules spread over 27 (40) distributions, with an average greedy performance of 0.99 (0.86), and an average training time of 43 (165) seconds. It is clear that rule simplification is highly effective in all aspects of CERRLA's learning.

5.2 Ms. Pac-Man

MS. PAC-MAN is a famous arcade video-game in which the player (CERRLA) controls a character that eats dots for points inside a finite maze while avoiding four hostile ghosts. When one of four 'power dots' is eaten, the ghosts become non-hostile for a short time and can be eaten for an increasingly larger number of points. CERRLA's control of the character is defined by high-level actions that resolve into low-level directional movement. If multiple actions are predicted by the same rule, the action with the closest object to the agent is acted upon. If multiple objects are equidistant in different directions, the next rule in the policy breaks the tie (otherwise choose randomly).

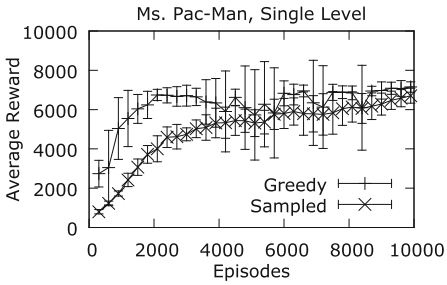
A MS. PAC-MAN state is described by similar predicates seen in [12]: $P_s = \{distance(Thing, \mathbb{N}), junctionSafety(Junction, \mathbb{N}), blinking(Ghost), edible(Ghost)\}$ and type predicates $P_t = \{thing, dot, ghost, powerDot, ghostCentre, junction\}$. The action predicates are $P_a = \{moveTo(Thing, \mathbb{N}), moveFrom(Thing, \mathbb{N}), toJunction(Junction, \mathbb{N})\}$ (the numeric value is meta-data for resolving the action).

Within a single level, CERRLA achieves an average greedy performance of 7196 points per episode. Compared to the conceptually equivalent CE-RANDOMRB agent from [12] (6382 points), CERRLA learns a slightly better policy. CERRLA performs slightly worse than the reported hand-coded and human average scores of 8186 and 8064 points respectively of [12]. An agent can achieve a theoretical maximum of 15,600 points in a single level, so CERRLA's performance could be improved. Figure 4a shows an example policy produced by CERRLA that focuses primarily on eating *edible ghosts*, *powerDots*, and *dots* in that order.

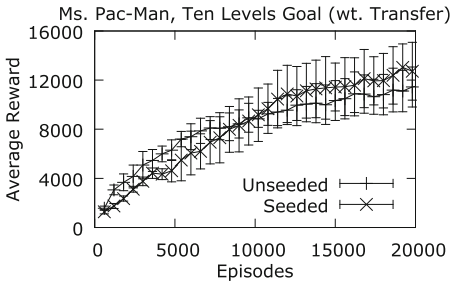
CERRLA's rule-based representation can also facilitate *transfer learning* (transfer learned behaviour for a source goal into behaviour for a target goal). During initialisation for the target goal, each rule in the greedy policy for the source goal seeds a new distribution, providing a headstart in the specialisation process. When the behaviour learned in the *Single-Level* goal is used to initialise the algorithm for a *Ten-Levels* goal, an improvement can be seen in the resulting behaviour (Fig. 4c).

edible(X), distance(X, Y) → moveTo(X, Y)
powerDot(X), distance(X, Y) → moveTo(X, Y)
thing(X), distance(X, Y), not(ghost(X)), not(ghostCentre(X)) → moveTo(X, Y)
dot(X), distance(X, (26 ≤ Y ≤ 52)) → moveFrom(X, Y)

(a) Example *Single Level* Ms. Pac-Man policy generated by Cerrla. Achieves an average reward of 7534.



(b) Single level of Ms. Pac-Man, limited to 10,000 episodes.



(c) Ten levels of Ms. Pac-Man, with *Single Level* seeded policy and unseeded learning, limited to 20,000 episodes.

Fig. 4. Cerrla's performance in the Ms. Pac-Man environment.

5.3 Carcassonne

Carcassonne is a turn-based, medieval-themed board game in which players attempt to control terrain via tokens called 'meeples' to score points. Each player has two actions per turn: place a randomly drawn tile adjacent to existing tiles such that all edges match up, then optionally place a meeple on any terrain on the placed tile. An episode ends when all tiles have been placed, at which point any unfinished terrain is scored.

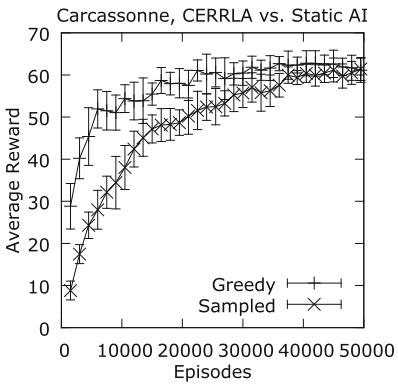
A Carcassonne state is described using a combination of the 22 state predicates and 10 type predicates (a full specification can be found in [25]), with the valid actions described as one of two actions: $P_a = \{placeTile(Player, Tile, Location, Orientation), placeMeeple(Player, Tile, Terrain)\}$.

In a Carcassonne game against a static AI using a min-max strategy for making decisions (the default AI for JCloisterZone[3]), Cerrla achieves an average score of 63 per game. In comparison, the min-max AI scores 92 and a related Monte-Carlo Tree Search approach achieves approximately 85 [26]. Carcassonne's complex dynamics prove to be challenging for Cerrla, but it is able to learn an 'easy opponent' strategy. Figure 5a shows a policy produced by Cerrla. The policy places tiles in close groups or near *cloisters*, or anywhere by default. Meeples are typically placed on high *worth* (>3) terrain, or any *open* terrain if Cerrla has ≥4 meeples left.

[3] http://jcloisterzone.com/en/

currentPlayer(X), controls(X, ?), validLoc(Y, Z, W), numSurroundingTiles(Z, 4.5 ≤ V ≤ 8.0) → placeTile(X, Y, Z, W)
currentPlayer(X), meepleLoc(Y, Z), worth(Z, 3.0 ≤ V ≤ 6.0), not(nextTo(?, ?, Z)) → placeMeeple(X, Y, Z)
currentPlayer(X), controls(X, ?), meepleLoc(Y, Z), worth(Z, 3.0 ≤ V ≤ 6.0) → placeMeeple(X, Y, Z)
currentPlayer(X), meeplesLeft(X, 4.0 ≤ U ≤ 7.0), meepleLoc(Y, Z), worth(Z, 1.5 ≤ V ≤ 4.5), not(completed(Z)) → placeMeeple(X, Y, Z)
currentPlayer(X), controls(X, ?), validLoc(Y, Z, W), numSurroundingTiles(Z, 3.625 ≤ V ≤ 5.375) → placeTile(X, Y, Z, W)
currentPlayer(X), meeplesLeft(X, 4.0 ≤ U ≤ 7.0), meepleLoc(Y, Z), tileEdge(Y, ?, Z), open(Z, V) → placeMeeple(X, Y, Z)
currentPlayer(X), validLoc(Y, Z, W), numSurroundingTiles(Z, 2.75 ≤ V ≤ 6.25), cloisterZone(Z, ?) → placeTile(X, Y, Z, W)
currentPlayer(X), controls(X, ?), validLoc(Y, Z, W), numSurroundingTiles(Z, 2.75 ≤ V ≤ 6.25) → placeTile(X, Y, Z, W)
currentPlayer(X), validLoc(Y, Z, W) → placeTile(X, Y, Z, W)

(a) Example Carcassonne policy generated by Cerrla. Achieves an average reward of 65.



(b) Cerrla vs. Min-max AI in Carcassonne, limited to 50,000 episodes.

Fig. 5. Cerrla's performance in the Carcassonne environment.

6 Conclusions

The application of the CEM to RRL — Cerrla — has been shown to be capable of creating and combining sets of relational condition-action rules into effective policies in a range of different environments. Although the number of training episodes exceed value-based methods, the learning rate remains constant with increased scale of the problem and the simplified rules minimise rule evaluation time. It should be noted that the representation of Ms. Pac-Man and Carcassonne may not be ideal, and may even limit Cerrla's behaviour, but it is clear that Cerrla can create effective behaviour with it. In general, Cerrla exhibits good scalability and, given only a problem's high-level specification and state observations, produces human-readable policies that are competitive with more specialised single-domain approaches.

References

1. Sutton, R.S., Barto, A.G.: Reinforcement Learning: An Introduction (Adaptive Computation and Machine Learning). The MIT Press, Cambridge (1998)
2. Džeroski, S., De Raedt, L., Driessens, K.: Relational reinforcement learning. Mach. Learn. **43**, 7–52 (2001)
3. van Otterlo, M.: The Logic of Adaptive Behaviour: Knowledge Representation and Algorithms for the Markov Decision Process Framework in First-Order Domains. IOS Press, Amsterdam (2009)
4. Wiering, M., van Otterlo, M. (eds.): Reinforcement Learning: State-Of-The-Art, vol. 12. Springer-Verlag New York Incorporated, New York (2012)
5. Driessens, K.: Relational reinforcement learning. Ph.D. thesis, Department of Computer Science, Katholieke Universiteit Leuven, Belgium (2004)

6. Driessens, K., Džeroski, S.: Combining model-based and instance-based learning for first order regression. In: Proceedings of the 22nd International Conference on Machine Learning, pp. 193–200. ACM (2005)
7. Croonenborghs, T., Ramon, J., Blockeel, H., Bruynooghe, M.: Online learning and exploiting relational models in reinforcement learning. In: Proceeding of the International Conference on Artificial Intelligence (IJCAI), pp. 726–731 (2007)
8. Driessens, K., Džeroski, S.: Integrating guidance into relational reinforcement learning. Mach. Learn. **57**(3), 271–304 (2004)
9. Muller, T., van Otterlo, M.: Evolutionary reinforcement learning in relational domains. In: Proceedings of the 7th European Workshop on Reinforcement Learning, Citeseer (2005)
10. van Otterlo, M., De Vuyst, T.: Evolving and transferring probabilistic policies for relational reinforcement learning. In: BNAIC 2009: Benelux Conference on Artificial Intelligence, October 2009
11. Rubinstein, R.Y.: Optimization of computer simulation models with rare events. Eur. J. Oper. Res. **99**(1), 89–112 (1997)
12. Szita, I., Lörincz, A.: Learning to play using low-complexity rule-based policies: illustrations through Ms. Pac-Man. J. Artif. Int. Res. **30**(1), 659–684 (2007)
13. Kistemaker, S., Oliehoek, F., Whiteso, S.: Cross-entropy method for reinforcement learning. Bachelor thesis, University of Amsterdam, Amsterdam, The Netherlands, June 2008
14. Tak, M.: The cross-entropy method applied to SameGame. Bachelor thesis, Maastricht University, Maastricht, The Netherlands (2010)
15. De Boer, P., Kroese, D., Mannor, S., Rubinstein, R.: A tutorial on the cross-entropy method. Ann. Oper. Res. **134**(1), 19–67 (2004)
16. Goldberg, D.E.: Genetic Algorithms in Search, Optimization and Machine Learning, 1st edn. Addison-Wesley Longman Publishing Co., Inc., Boston (1989)
17. Wilson, S.W.: Classifier fitness based on accuracy. Evol. Comput. **3**(2), 149–175 (1995)
18. Mellor, D., Mellor, D.: A learning classifier system approach to relational reinforcement learning. In: Takadama, K., et al. (eds.) IWLCS 2006 and IWLCS 2007. LNCS (LNAI), vol. 4998, pp. 169–188. Springer, Heidelberg (2008)
19. Mellor, D.: A learning classifier system approach to relational reinforcement learning. Ph.D. thesis, School of Electrical Engineering and Computer Science, The University of Newcastle, Australia (2008)
20. Fern, A., Yoon, S., Givan, R.: Approximate policy iteration with a policy language bias: solving relational markov decision processes. J. Artif. Int. Res. **25**(1), 75–118 (2006)
21. Kersting, K., Driessens, K.: Non-parametric policy gradients: a unified treatment of propositional and relational domains. In: Proceedings of the 25th International Conference on Machine Learning, ICML '08, pp. 456–463. ACM, New York (2008)
22. Plotkin, G.D.: A note on inductive generalization. Mach. Intell. **5**, 153–163 (1970)
23. Szita, I., Lörincz, A.: Online variants of the cross-entropy method. Technical report, arXiv:0801.1988 (2008)
24. Aslam, J.A., Popa, R.A., Rivest, R.L.: On estimating the size and confidence of a statistical audit. In: Proceedings of the USENIX Workshop on Accurate Electronic Voting Technology, EVT'07, pp. 8–8. USENIX Association, Berkeley (2007)
25. Sarjant, S.: Policy search based relational reinforcement learning using the cross-entropy method. Ph.D. thesis, The University of Waikato (2013)
26. Heyden, C.: Implementing a computer player for Carcassonne. Master's thesis, Maastricht University (2009)

AND Parallelism for ILP: The APIS System

Rui Camacho[1(✉)], Ruy Ramos[1], and Nuno A. Fonseca[2]

[1] DEI and Faculdade de Engenharia and LIAAD-INESCTEC,
Universidade do Porto, Porto, Portugal
rcamacho@fe.up.pt
[2] EMBL Outstation, European Bioinformatics Institute (EBI) and
CRACS-INESCTEC, Cambridge, UK

Abstract. Inductive Logic Programming (ILP) is a well known approach to Multi-Relational Data Mining. ILP systems may take a long time for analyzing the data mainly because the search (hypotheses) spaces are often very large and the evaluation of each hypothesis, which involves theorem proving, may be quite time consuming in some domains. To address these efficiency issues of ILP systems we propose the APIS (**A**nd **P**arallel**IS**m for ILP) system that uses results from Logic Programming AND-parallelism. The approach enables the partition of the search space into sub-spaces of two kinds: sub-spaces where clause evaluation requires theorem proving; and sub-spaces where clause evaluation is performed quite efficiently without resorting to a theorem prover. We have also defined a new type of redundancy (Coverage-equivalent redundancy) that enables the prune of significant parts of the search space. The new type of pruning together with the partition of the hypothesis space considerably improved the performance of the APIS system. An empirical evaluation of the APIS system in standard ILP data sets shows considerable speedups without a lost of accuracy of the models constructed.

1 Introduction

Multi-Relational Data Mining (MRDM) addresses the important challenge of how to learn or mine the large multi-relational databases that are being developed by individuals and organizations. Inductive Logic Programming (ILP) is a well known approach to MRDM. It starts from a logic-based representation in order to induce theories that can describe common patterns in the data, or that discriminate between classes of examples. ILP benefits from the expressiveness and conciseness of logic and has been shown to be effective over a large range of applications.

As most other Multi-Relational Data Mining (MRDM) systems, ILP systems must *search* over a very large space. Controlling the running time is thus a key consideration and has become even more important as data-base size increase. Indeed, often ILP practitioners have to reduce the search space by using techniques such as sampling or strong language bias in order to actually obtain results.

G. Zaverucha et al. (Eds.): ILP 2013, LNAI 8812, pp. 93–106, 2014.
DOI: 10.1007/978-3-662-44923-3_7

There is therefore a strong motivation to making ILP systems *faster* [8]. Of the several approaches being considered, *parallelism* is a natural fit, given the widespread availability and the low-cost of modern parallel platforms. Indeed, one can argue that parallelism is nowadays fundamental in large-scale data mining. Therefore, it is unsurprising there has been much interest on parallel ILP [10].

We propose a novel algorithm for parallel ILP data-mining, APIS (**A**nd **P**aralel**IS**m). APIS takes advantage of previous work in logic programming for AND-parallelism, and takes it to the context to ILP. Results for our initial implementation look very promising.

The remainder of the paper is organized as follows. Section 2 provides a short introduction to ILP necessary to understand our proposal. It also explains the Logic Programming AND-parallelism foundation of the proposal. Section 3 describes the APIS systems. Section 4 presents an empirical evaluation of the proposed technique. We survey the parallel execution of ILP systems in Sect. 5. Finally, in Sect. 6, we draw some conclusions and describe future work.

2 Background

The fundamental goal of a predictive ILP system is to construct a model H given background knowledge B and observations E, usually called examples in the machine learning literature. The problem that a *predictive ILP* system must solve is to find a consistent and complete model H, i.e., find a set of hypotheses that *explain* all given positive examples, while being consistent with the given negative examples. More formally, given:

- B: background knowledge encoded as statements of a Logic Program.
- $\mathcal{L}$: a pre-defined language for acceptable hypotheses.
- E: a finite set of examples $= E^+ \cup E^-$ where the E^+ are named positive examples; E^- is an optional set of negative examples; and $B \not\models E^+$

the goal is to find a set of logical statements H from the set $\mathcal{L}$ of clauses that are sufficient and consistent with the examples. *Sufficiency* is defined as $B \cup H \models E^+$ and $B \cup \{H_i\} \models e_1 \vee e_2 \vee \cdots \vee e_p$ $(1 \leq i \leq k)$. *Consistency* is defined as $B \cup H \not\models \square$ and $B \cup H \not\models \cup E^-$. The sufficiency requirements are designed to ensure that the theory H *predicts* the positive examples and that every clause h_i predicts at least one positive example. The consistency requirements try to ensure that the theory is consistent with the background knowledge, and that it is a good classifier. One particularly popular framework to constraint the clauses considered is *mode declarations*, where one assigns types to clause's arguments and says that some arguments must have been bind by a previous literal in the same clause.

Mode-Directed Inverse Entailment (MDIE) [15] takes advantage of mode declarations to constrain the ILP search space. The key idea in MDIE is to find all literals that could be used in rules that explain the example. This is achieved by selecting a seed example and then constructing the *saturated clause* from the

set of all literals that could be used to prove (directly or indirectly) the example. Several ILP systems [2,9,15,22] use the saturated clause in order to anchor the search space lattice.

MDIE implementations such as Progol [15] or Aleph [22] start from a most general clause, and then enumerate the clauses that subsume the bottom-clause until finding a good clause that can be included in the theory. The search space is therefore bound by the combinations of literals in the bottom-clause and thus can grow very quickly, severely restricting the scalability of ILP systems. Several approaches have been proposed in order to address this important problem. Work has included faster evaluation of nodes in the search space [8,20], and reducing redundancy in the search space through more intelligent search or refining bias. A promising approach is to divide the search space and to use parallelism in order to improve running times, as discussed next. During the search each clause has to be evaluated by counting how many examples can be derived when the hypothesis is added to the background knowledge. This evaluation procedure requires a theorem prover and is most often the major time consuming step in the search procedure. For efficiency sake it is usual to keep track of the examples derivable by each clause (coverage lists). To avoid evaluating the refined clauses in the complete list of examples the coverage lists of the "parent clause" are usually used.

2.1 Parallel Execution of Logic Programs

There is a strong connection between parallelism in the context of ILP and parallelism in the context of logic programming (LP). Parallelism has been widely studied in LP [12], where it can be exploited implicitly, by parallelising the LP inference mechanism, or explicitly, by extending logic programs with primitives that create and manage tasks and allow for task communication.

Explicit parallelism is often implemented by interfacing to existing low-level primitives, such as Posix Threads [23], or MPI [11]. In contrast, implicit parallelism provides independency from the underlying low-level primitives. Two major sources of implicit parallelism have been recognized. In *or-parallelism*, the search in the LP system is run in parallel. Or-parallelism is known to achieve scalable speedups on current hardware [6] but it works better when we want to perform complete search, which may be expensive in the context of ILP.

And-Parallelism corresponds to running conjunctions of goals, or and-tasks, in parallel. If the goals communicate during the parallel computation, it is called *dependent and-parallelism*. Dependent and-parallelism may be used for concurrent languages or to implement pipelines [1]. On the other hand, *independent and-parallelism* (IAP) is useful in divide-and-conquer applications and often corresponds to coarse-grained tasks.

Modern IAP implementations support both shared-memory, such as thread-based systems [14], and distributed platforms [4]. Our approach is based on *independent and-parallelism* (IAP).

3 The APIS System

The APIS system is based on a new approach to the parallel execution of ILP systems. This approach establishes a partition on the hypothesis space enabling each sub-space to be executed in parallel. There are two types of sub-spaces: sub-spaces requiring theorem proving for clause evaluation; and sub-spaces that efficiently compute clause evaluation without the need of theorem proving. Not only the partition enables the parallel search but also achieves additional speedups resulting from the fact that some of the sub-spaces do not use theorem proving to evaluate the hypotheses. Although a partition is established on the hypothesis space the resulting sub-spaces are not completely independent as we explain later.

The theoretical foundation of our proposal is based on results from Logic Programming (LP) AND-parallelism.

And-Parallelism corresponds to running conjunctions of goals, or and-tasks, in parallel. *Independent and-parallelism* (IAP) is useful in divide-and-conquer applications and often corresponds to coarse-grained tasks.

It is well known in LP that if a clause has subsets of literals with literals in each subset not sharing variables with any literal of the other subsets, then each subset can be executed in parallel. When traversing the hypothesis space an MDIE-based ILP system constructs and evaluates clauses. Traditionally clause evaluation is done using a theorem prover[1]. Among the clauses constructed during the search, there are clauses that satisfy the LP IAP constraint: clauses with sets of literals that do not share variables. We can then think of a search procedure that generates in parallel each subset of literals in the "traditional" way (using theorem proving for evaluation) and then combines each sub-set to form a new clause and make the evaluation of the combined clause in a more efficient way. The coverage of the combined clause is computed by the intersection of the coverage lists of the clauses being combined. This result cannot, however, be efficiently applied in a traditional ILP system since it is computationally expensive to determine if the partition of the clause's literals into sub-sets that do not share variables exists. The key point of the APIS system approach is to analyze the mode declarations and establish the partition of the hypothesis space based on the mode declarations, thus avoiding the analysis of each clause for independent sets of literals at induction-time. Such partition can be computed as a pre-processing step in an efficient way. The overall process is therefore divided in two steps: a pre-processing step where mode declarations are used to establish the partition of the hypothesis space; and the execution in parallel of the sub-spaces resulting from the previous step. We now explain each step in detail.

Definition 1. *Island. An island is a set of mode declarations satisfying the following two conditions. Each mode declaration shares at least one type with other modes in the same island. Each mode declaration does not share any type*

[1] Counting the number of examples derivable from the hypothesis and the background knowledge.

with any other mode declaration outside the island. Types of the head literal are excluded from the above mentioned "type checking".

The core of the APIS system is the identification of the *islands* since they will be used in the partition of the hypothesis space. The algorithm for the automatic identification of the *islands* is described by Algorithm 1. The use of the islands in the the parallel search of the hypothesis space is described by Algorithm 2.

Algorithm 1. *Islands* computation from the mode declarations

```
1: function ComputeIslands(AllModes)
2:     IslandsSet ← ∅
3:     Modes ← removeHeadInputArguments(AllModes)          ▷ pre-processing step
4:     while Modes ≠ ∅ do                                  ▷ process all modes
5:         Mode = withoutInputArguments(Modes)
6:         Modes = Modes \ { Mode }
7:         Island = ExtendIsland({Mode}, Modes)
8:         IslandsSet ← IslandsSet ∪ { Island }
9:     end while
10:    return IslandsSet
11: end function
12:
13: function ExtendIsland(Island, Modes)
14:    repeat
15:        Mode = LinkedToTheIsland(Modes)          ▷ returns ∅ if no mode was found
16:        Modes = Modes \ { Mode }
17:        Island ← Island ∪ { Mode }
18:    until Mode = ∅
19:    return Island                                ▷ Island as a set of modes
20: end function
```

The algorithm to compute the islands accepts as input a set of mode declarations and returns a set of *islands*. First, a pre-processing is done to remove the types appearing in the head mode declaration and the mode arguments that are constants. After the pre-processing the algorithm enters a cycle where each island is determined and terminates whenever there are no more mode declarations to process. In the main cycle a seed mode is chosen to start a new *island* and then the island is "expanded". Expanding an island consists in adding any mode declaration not yet in the island sharing a type with any mode already in the island. The expansion stops as soon as there is no mode outside the island sharing a type with the modes inside the island.

APIS execution algorithm is schematized in Algorithm 2. Algorithm 2 starts by computing the *islands* and each client node is instructed to upload the data set without the mode declarations. In the line of MDIE greedy cover ILP algorithms the main cycle generates hypotheses, adds the best discovered hypothesis to the final theory and removes the examples covered by the added hypothesis. The cycle is repeated until no uncovered positive examples are left. The specificity of APIS is evident in (steps 8 through 19). In this part of the algorithm APIS uses a pool of client nodes and a pool of sub-spaces of the hypothesis space to search (determined by the partition made on the mode declarations). Each node searches a sub-space. There are two kinds of sub-spaces: "saturation-based" sub-spaces; and "combination-based" sub-spaces. A saturation-based sub-space

Algorithm 2. The APIS parallel execution algorithm

```
1: function InduceTheory(DataSet, Clients)
2:     Islands ← ComputeIslands(GetModes(DataSet))
3:     Theory ← ∅
4:     Examples ← PositiveExamples(DataSet)                    ▷ initial positive examples
5:     broadCast(Clients, loadIslandsDataSets)
6:     while Examples ≠ ∅ do                                   ▷ while not covering all positives
7:         Samples = getSample(Examples)
8:         Jobs ← getJobs(Islands, Samples)
9:         while Jobs ≠ ∅ do                                   ▷ all islands processed in the cycle
10:            if Clients ≠ ∅ then
11:                W ← client(Clients)                         ▷ get next available client
12:                Clients ← Clients \ { W }
13:                J ← nextJob(Jobs)                           ▷ select a non-processed job
14:                Jobs ← Jobs \ { J }
15:                sendMsg(W, J)                               ▷ client W processes job J
16:            end if
17:            if FinishedClient(C) ≠ ∅ then Clients ← Clients ∪ { C }
18:            end if
19:        end while
20:        h = IslandsResults()                                ▷ returns the best hupothesis
21:        Covered = Cover(h, Examples)                        ▷ compute h coverage
22:        Examples = Examples \ Covered
23:        if Examples ≠ ∅ then broadcast(Clients, removeExamples(Covered))
24:        end if
25:        Theory ← Theory ∪ { h }
26:    end while
27:    return Theory
28: end function
```

is generated as in a typical saturation followed by reduction steps that characterize MDIE systems. The difference is that to generate the sub-space a sub-set of the mode declarations (an *island*) is used. All clauses constructed in this kind of subspace are evaluated by proving the examples from background knowledge and the hypothesis under evaluation. On the other hand in "combination-based" sub-spaces theorem proving is not required. Each clause constructed in a combination-based sub-space merges pairs of clauses each one coming from previously searched spaces that do not share islands. This restriction allows the evaluation of the new clauses by intersection of the parent's coverage lists. We can see that there is a dependency among combination-based sub-spaces. The saturation-based sub-spaces are the only ones completely independent. Let us further remark that in the main cycle of the algorithm we search several hypothesis spaces at the same time[2]. We have an hypothesis space for each example of the seed. All of the jobs to execute (sub-spaces to be searched) are in a common pool but only sub-spaces belonging to the same example are combined. The number of jobs associated with each example is equal to the number of all possible combinations of the islands up to the clause length. First the saturation-based sub-spaces are generated, then these sub-spaces are combined in pairs them in groups of three and so on up to the "clause length" value. The combinations are all computed once before execution of the algorithm and each sub-space is schedule to run as soon as the two "parents" finish.

[2] As many as the size of the sample.

3.1 Redundancy Avoidance

It is well known that there is a lot of redundancy among the hypotheses in an ILP search space. Several types and remedies have been identified and proposed, see [19]. With the APIS approach there is a another redundancy situation that can be avoided and therefore improving the search.

As explained previously, if a clause is "constructed" by combining two clauses from different *islands*, its coverage is computed by the intersection of the two coverage lists of the clauses being combined. The coverage result depends only on the coverage lists of the combining clauses and not on the clauses *per se*[3]. If we have clause C_1 and clause C_2 with the same positives and negatives coverage lists originated from the same *island* and we try to combine each of them with clause C_3, from a different island, we will necessarily obtain two clauses (C_1 "+" C_3 and C_2 "+" C_3) with the same coverage lists (positives and negatives). Combining each of C_1 or C_2 with clauses from other *islands* will always result in clauses with equal coverage lists. We call such clauses (C_1 and C_2) **coverage equivalent clauses**.

Definition 2. Coverage-equivalent clauses. *Two clauses C_1 and C_2 are coverage equivalent if both cover exactly the same positive and negative examples.*

Although coverage equivalent clauses may not be equivalent in the logic sense, a coverage-based ILP system will always report only one exemplar of the coverage equivalent class. In the APIS system we keep only one exemplar of each coverage equivalent classes (the shortest clause).

Coverage equivalence is used in APIS for pruning in the following way. During the search of a sub-space the inconsistent clauses are stored in a file. The purpose is to combine them with other inconsistent clauses from other sub-spaces. Pruning takes place at saving time. From each coverage equivalence class only a single clause is saved.

4 Experiments and Results

4.1 Experimental Settings

We have used four data sets to evaluate the APIS system. DBPCAN is part of the water disinfection by-products database and contains predicted estimates of carcinogenic potential for 178 chemicals. The goal is to provide informed estimates of carcinogenic potential to be used as one factor in ranking and prioritizing future monitoring, testing, and research needs in the drinking water area [24]. The second data set is CPDBAS, the Carcinogenic Potency Data Base that contains detailed results and analyzes of 6540 chronic, long term carcinogenesis bio assays.

[3] Opposite from what happens when literals share variables.

A description of the background knowledge for these two data sets[4] can be found in [3]. Other two data sets used in this study are the carcinogenesis and mutagenesis well known in ILP and can be found in the Oxford University Machine Learning repository[5] along with an explanation of the domain that produced the data.

The data sets are characterized in Table 1 together with the associated Aleph's parameters used in the experiments. The nodes limit parameter indicated in the table concern the sequential execution value. When running APIS we have divided the nodes limit among the saturation-based sub-spaces. For each saturation-based sub-space the nodes limit is a weighted proportion of the nodes limit of the sequential execution. The weight used is based on the number of mode declaration of the corresponding island. For instance, let us consider the carcinogenesis data set. The nodes limit is set to 1 million (1M) clauses in the sequential execution. Four islands where identified hence the nodes limits in the saturation-based sub-spaces were the following ones: 3/34 * 1M for island 1; 24/34 * 1M for island 2; 4/34 * 1M for island 3 and 3/34 * 1M for island 4. In carcinogenesis there are 34 mode declarations, 3 in island 1, 24 in island 2, 4 in island 3 and 3 in island 4. The nodes limit used in the sequential execution is the overall nodes limit used by APIS for each example. In the current experiments the overall nodes limit is split among the saturation-based sub-spaces according to the number of mode declaration in their island. If the saturation-based sub-spaces did not reach their nodes limit (what happens frequently for some of them) the combination-based sub-spaces can run and use the number of nodes not used by the saturation-based sub-spaces. As said before, for each example the global limit, used in the sequential execution, is never surpassed by the complete set of sub-spaces searched.

Table 1. Characterization of the data sets used in the study. In the cells of the second column P/N represents the number of positive examples (P) and negative examples (N). The 5 right most columns are the values for Aleph's parameters.

data set name	number of examples	number of *islands*	clause length	nodes (Millions)	noise	minimum positives	sample size
carcinogenesis	162/136	4	5	0.5	10	12	30
mutagenesis	125/63	5	6	1	4	9	25
dbpcan	80/98	37	7	1	2	5	30
cpdbas	843/966	37	6	0.1	150	150	5

All the experiments were carried out on a cluster of 8 nodes having two quad-core Xeon 2.4 GHz and 32 GB of RAM per node and running Linux Ubuntu 8.10.

[4] Source data for both data sets is available from the Distributed Structure-Searchable Toxicity (DSSTox) Public Data Base Network from the U.S. Environmental Protection Agency http://www.epa.gov/ncct/dsstox/index.html,accessedDec2008.

[5] http://www.cs.ox.ac.uk/activities/machlearn/applications.html

Table 2. Speedups (a) and accuracy (b) obtained in the experiments numbers in each cell correspond to average and standard deviation (in parenthesis). There is no statistical difference ($\alpha \leq 0.05$) between the sequential execution accuracy values and the parallel execution for each data set.

data set	**number of worker nodes**			
	2	4	6	7
carcinogenesis	4.8(2.1)	5.6(2.7)	6.7(2.6)	6.1(2.6)
mutagenesis	76.5(32.9)	138.9(82.7)	188.4(119.3)	231.3(148.6)
dbpcan	13.8(2.5)	26.7(4.3)	36.5(5.5)	41.1 (5.9)
cpdbas	18.3(7.0)	31.4(16.1)	36.2(26.9)	28.5(11.8)

(a)

data set	**sequential execution**	**number of worker nodes**			
		2	4	6	7
carcinogenesis	53.7(3.8)	58.9(5.5)	57.8(3.8)	57.8(4.8)	58.0(7.6)
mutagenesis	84.1(6.9)	80.7(5.4)	82.0(4.8)	80.9(5.2)	81.3(4.7)
dbpcan	87.9(5.0)	89.8(4.1)	89.3(5.1)	89.3(5.1)	89.3(5.1)
cpdbas	54.0(1.8)	51.2(1.4)	53.6(1.2)	53.5(1.2)	53.4(1.0)

(b)

To estimate the predictive quality of the classification models we compute the average values (speed-up and accuracy) of 10 (70 %/30 %) train/test splits. The ILP system used was Aleph 5.0 [22].

4.2 Results and Discussion

Overall, the results show that significant speedups were achieved by APIS, well beyond the number of processors (Table 2(a))[6] without affecting accuracy (no statistical significant difference for $\alpha \leq 0.05$), Table 2(b). To understand the results a second set of experiments were performed with several sorts of countings on all parts of the APIS system. In these second set of experiments we have measured the execution times of all sub-spaces, we have counted the number of constructed clauses and the number of pruned clauses (shown in Table 3).

We have focus our initial attention on the saturation-based sub-spaces since their running times and number of nodes searched are much larger than the intersection-based sub-spaces. Results in Table 3 concern the saturation-based sub-spaces only.

We can observe that the number of clauses constructed by APIS (in the saturation-based sub-spaces) is smaller than in the sequential execution. For example, in the mutagenesis data set the whole number of clauses constructed in saturation-based sub-spaces are 20 % of the number in sequential runs. This is due to the lower limit imposed in each sub-space and because some of those sub-spaces do not reach the nodes limit. The accuracy values are similar (Table 2(b)) despite the reduction in the total number of nodes searched.

[6] Except for the carcinogenesis data set.

The major contribution for the speedups is, however, from the parallel search of the sub-spaces. We identified two sources of the parallel execution on the speedups. With enough CPUs (number of workers larger than the number of the islands) the execution time would be broadly determined by the slower subspace search. For example, in mutagenesis data set, if we have more than 5 CPU workers we can search the five saturation-based sub-spaces in parallel. The overall time is determined by the slower search. With this effect alone we would expect the speedups to be close to the speedup of the search in the slower subspace. In the first result's column of Table 3 we can see, for example, that the slowest sub-space in mutagenesis has a speedup of 10.8 when compared with the sequential run.

Looking at the global data sets speedup results we see that the speedup of the slowest sub-space search alone does not explain the global speedups obtained. Again, looking at the results columns 4th and 5th in Table 3, we can see in column 4 the number of "slow" sub-spaces (1 in mutagenesis and 3 in dbpcan, for example) and can also see in column 5 that the other sub-saves use less than 10 % of the time of the slower ones. The is there are a one or few "slow" subspaces and their run time is much larger than the others. This means that we can start processing the next example much earlier than the finish time of the slower sub-space. In practice we can run several examples in parallel. This is also a significant contribution for the global speedup.

Another contribution, although weaker, for the speedup results is the use of intersection of coverage lists instead of theorem-proving. The number of clauses evaluated using intersection of coverage lists is rather small (when compared with the theorem-proving case) but represent also a faster method to evaluate clauses.

Table 3. Execution statistics. Column two shows the average (and standard deviation) of the quotient between the sequential run time of an example and the slowest sub-space (speedup). Column three sown the percentage of nodes constructed by APIS in the saturation-subspaces and the nodes constructed in the sequential run. Column four shows the number of "slow" subs-paces (left) and total number of saturation-based subspaces (right). Column 5 shows the average run time of all sub spaces (except the slowest ones) as a percentage of the slowest run time. The last column shows the coverage equivalence pruned nodes as a percentage of the total number of nodes constructed. Results concern the saturation-based sub-spaces only. Execution times and nodes constructed are negligible when compared with saturation-based subspace's values.

data set name	Slowest sbsp. speedup	nodes constructed in the sbspcs. (%)	Number of "slow" sbsps.	Av. other sub-spaces (%)	Coverage Equiv. pruning
carcinogenesis	2.7(1.8)	29	1/4	2(5)	12(4)
mutagenesis	10.8(8.7)	20	1/5	7(0)	16(10)
dbpcan	15.3(11.1)	80	3/37	1(1)	27(8)
cpdbas	5.3(5.8)	16	1/37	1(1)	18(5)

Table 4 show the island's membership of predicates that appear in the sequential execution theories.

Table 4. Island's membership of the predicates found in the clauses of the theories of sequential execution. N means a clause with all predicates in island N, N-M means a clause with predicates belonging to islands N and M, and N-M-L means a clause with predicates belonging to islands N, M and L. A list of the island's predicates can be found in Table 5 of the Appendix.

data set name	**islands ids**
carcinogenesis	1 , 1-3, 1-4, 2-3, 3-4, 1-2-3
mutagenesis	3, 4, 2-3, 2-4, 3-4, 1-3-4
dbpcan	1, 1-2
cpdbas	1, 2, 1-2

5 Parallel Execution of ILP Systems

Based on the principal performance bottlenecks for ILP systems identified in Sect. 1, we classify three main sources of parallelism in ILP systems [10].

Search parallelism arises from the need to enumerate clauses. We can further distinguish between parallel execution of multiple searches, and the parallel execution within a search. The granularity of the latter is substantially finer than the former. This strategy was the first to be exploited, as an extension of Dehaspe and De Raedt's Claudien system [7]. It is also exploited by Ohwada *et al.* [18] and by Wielemaker and Srinivasan in the context of randomised search [23].

Evaluation parallelism arises from the need to compute the utility of a clause. This usually requires determining the subset of E entailed by the D_i given B and H_{i-1}. A coarse-grained strategy involves partitioning E into blocks. The blocks are then provided to individual processors, which compute the examples covered in the block. Ohwada and Mizoguchi [17] implement evaluation and search parallelism in the context of inverse entailment.

Data parallelism arises when individual processors are provided with subsets of the examples prior to invoking the search procedure in Figure. Wang and Skillicorn [21] use this technique to parallelise the Progol algorithm [16]. They also use search and evaluation parallelism. Matsui *et al.* [13] compared search and evaluation parallelism, with initial promise for data-parallelism.

Notice that other classification criteria can be used. For example, as for LP systems, we can divide strategies into those that expect to use shared memory and those that expect to use distributed memory. Clare and King's Polyfarm [5] is an example of a system designed for distributed environments. Fonseca*et al.*'s survey of parallel ILP systems [10], reports that most of the best results for parallel ILP were obtained on shared-memory architecture, but argues that there is scope for experimenting with distributed-memory "clusters".

6 Conclusions

A new ILP system based on the partition of the hypothesis space and parallel search of the generated sub-spaces was presented. The partition of the hypothesis space results in two types of sub-spaces: "saturation-based and "combination-based" sub-spaces. Saturation-based sub-spaces are searched as in a "traditional" MDIE-based system. Combination-based sub-spaces combine clauses from two previously searched sub-spaces and evaluate them efficiently by intersection of the coverage lists of the clauses being combined. Using the process of combination of clauses a new type of redundancy was identified and implemented. Results of the APIS system, on well known data sets, show very good speed-ups without lost in accuracy. We are currently performing further runs in order to achieve good speedups without any decrease in accuracy. The procedure taken consists in finding a reasonable way of determining the "nodes" limit for the sub-spaces. This limit is specially critical for the saturation-based sub-spaces since they produce the initial set of clauses that are being combined in other sub-spaces. If node limit is too small we may loose crucial (sub-)clauses important for the combination process.

Acknowledgments. This work has been partially supported by Fundação para a Ciência e Tecnologia (FCT) through the project ADE (PTDC/EIA-EIA/121686/2010 (FCOMP-01-0124-FEDER-020575)). The work was also partial supported by project NORTE-07-0124-FEDER-000059, financed by the North Portugal Regional Operational Programme (ON.2 O Novo Norte), under the National Strategic Reference Framework (NSRF), through the European Regional Development Fund (ERDF), and by national funds, through the Portuguese funding agency, FCT.

A Composition of the Dataset's Islands

Table 5 shows the partial composition of the islands that where used to define the hypothesis sub-spaces. In the table we show only the predicates that appear in the models constructed in the sequential execution runs.

Table 5. Island's membership of the predicates that appear in the final theories induced by the APIS system.

data set name	island 1	2	3	4
carcinogenesis	ames/1 has_property/3 mutagenic/1	ashby_alert/3 ether/2 ar_halide/2 non_ar_6c_ring/2 non_ar_hetero_5_ring/2	atm/5 lteq/2 gteq/2	ind/3 lteq/2
mutagenesis	ring_size_5/2	logp/2 gteq/2	lumo/2 lteq/2	atm/5 bond/4 gteq/2 lteq/2
dbpcan	chemical_fingerprint/2 rotatable_bondcount/2 primary_carbon/2 atLeastOneOfFuncGroups/2 resonant_count/2 tertiary_carbon/2 primary_carbon/2 secondary_carbon	pharmacophore_fingerprint/4 ltPharmacophoreArg3/2 ltPharmacophoreArg2/2 gtPharmacophoreArg2/2		
cpdbas	atLeastOneOfFuncGroups/2 heteroaromatic_ringcount/2 fusedaliphatic_ringcount/2 tertiary_carbon/2 tautomer_count/2 ringcount/2 tertiary_carbon/2	pharmacophore_fingerprint/4 ltPharmacophoreArg2/2 gtPharmacophoreArg/2		

References

1. Bone, P., Somogyi, Z., Schachte, P.: Estimating the overlap between dependent computations for automatic parallelization. TPLP **11**(4–5), 575–591 (2011)
2. Camacho, R.: IndLog — induction in logic. In: Alferes, J.J., Leite, J. (eds.) JELIA 2004. LNCS (LNAI), vol. 3229, pp. 718–721. Springer, Heidelberg (2004)
3. Camacho, R., Pereira, M., Costa, V.S., Fonseca, N.A., Adriano, C., Simoes, C.J.V., Brito, R.M.M.: A relational learning approach to structure-activity relationships in drug design toxicity studies. J. Integr. Bioinform. **8**(3), 182 (2011)
4. Casas, A., Carro, M., Hermenegildo, M.V.: A high-level implementation of non-deterministic, unrestricted, independent and-parallelism. In: Garcia de la Banda, M., Pontelli, E. (eds.) ICLP 2008. LNCS, vol. 5366, pp. 651–666. Springer, Heidelberg (2008)
5. Clare, A.J., King, R.D.: Data mining the yeast genome in a lazy functional language. In: Dahl, V. (ed.) PADL 2003. LNCS, vol. 2562, pp. 19–36. Springer, Heidelberg (2002)
6. Costa, V.S., de Castro Dutra, I., Rocha, R.: Threads and or-parallelism unified. TPLP **10**(4–6), 417–432 (2010)
7. Dehaspe, L., De Raedt, L.: Parallel inductive logic programming. In: Proceedings of the MLnet Familiarization Workshop on Statistics, Machine Learning and Knowledge Discovery in Databases (1995)
8. Fonseca, N.A., Costa, V.S., Rocha, R., Camacho, R., Silva, F.: Improving the efficiency of inductive logic programming systems. Softw. Pract. Exper. **39**(2), 189–219 (2009)

9. Fonseca, N.A., Silva, F., Camacho, R.: April – an inductive logic programming system. In: Fisher, M., van der Hoek, W., Konev, B., Lisitsa, A. (eds.) JELIA 2006. LNCS (LNAI), vol. 4160, pp. 481–484. Springer, Heidelberg (2006)
10. Fonseca, N.A., Srinivasan, A., Silva, F.M.A., Camacho, R.: Parallel ilp for distributed-memory architectures. Mach. Learn. **74**(3), 257–279 (2009)
11. The MPI Forum: Mpi: a message passing interface (1993)
12. Gupta, G., Pontelli, E., Ali, K.A.M., Carlsson, M., Hermenegildo, M.V.: Parallel execution of prolog programs: a survey. ACM Trans. Program. Lang. Syst. **23**(4), 472–602 (2001)
13. Matsui, T., Inuzuka, N., Seki, H., Itoh, H.: Comparison of three parallel implementations of an induction algorithm. In: 8th International Parallel Computing Workshop, Singapore, pp. 181–188 (1998)
14. Moura, P., Crocker, P., Nunes, P.: High-level multi-threading programming in logtalk. In: Hudak, P., Warren, D.S. (eds.) PADL 2008. LNCS, vol. 4902, pp. 265–281. Springer, Heidelberg (2008)
15. Muggleton, S.: Inverse entailment and Progol. New Gener. Comput., Spec. Issue Induct. Log. Program. **13**(3–4), 245–286 (1995)
16. Muggleton, S., Firth, J.: Relational rule induction with CProgol4.4: a tutorial introduction. In: Džeroski, S., Lavrač, N. (eds.) Relational Data Mining, pp. 160–188. Springer, Heidelberg (2001)
17. Ohwada, H., Mizoguchi, F.: Parallel execution for speeding up inductive logic programming systems. In: Arikawa, S., Nakata, I. (eds.) DS 1999. LNCS (LNAI), vol. 1721, pp. 277–286. Springer, Heidelberg (1999)
18. Ohwada, H., Nishiyama, H., Mizoguchi, F.: Concurrent execution of optimal hypothesis search for inverse entailment. In: Cussens, J., Frisch, A.M. (eds.) ILP 2000. LNCS (LNAI), vol. 1866, pp. 165–173. Springer, Heidelberg (2000)
19. Costa, V.S., Srinivasan, A., Camacho, R., Blockeel, H., Demoen, B., Janssens, G., Struyf, J., Vandecasteele, H., Van Laer, W.: Query transformations for improving the efficiency of ILP systems. J. Mach. Learn. Res. **4**, 465–491 (2003)
20. Camacho, R., Blockeel, H., Demoen, B., Janssens, G., Struyf, J., Vandecasteele, H., van Laer, W.: Query Transformations for Improving the Efficiency of ILP Systems. J. Mach. Learning Res. Ashwin Srinivasan **4**, 465–491 (2003)
21. Skillicorn, D.B., Wang, Y.: Parallel and sequential algorithms for data mining using inductive logic. Knowl. Inf. Syst. **3**(4), 405–421 (2001)
22. Srinivasan, A.: The Aleph Manual (2003). http://web.comlab.ox.ac.uk/oucl/research/areas/machlearn/Aleph
23. Wielemaker, J.: Native preemptive threads in SWI-prolog. In: Palamidessi, C. (ed.) ICLP 2003. LNCS, vol. 2916, pp. 331–345. Springer, Heidelberg (2003)
24. Woo, Y.T., Lai, D., McLain, J.L., Manibusan, M.K., Dellarco, V.: Use of mechanism-based structure-activity relationships analysis in carcinogenic potential ranking for drinking water disinfection by-products. Environ. Health Perspect. **110**, 75–87 (2002)

Generalized Counting for Lifted Variable Elimination

Nima Taghipour, Jesse Davis, and Hendrik Blockeel(✉)

Department of Computer Science, KU Leuven, Leuven, Belgium
hendrik.blockeel@cs.kuleuven.be

Abstract. Lifted probabilistic inference methods exploit symmetries in the structure of probabilistic models to perform inference more efficiently. In lifted variable elimination, the symmetry among a group of interchangeable random variables is captured by *counting formulas*, and exploited by operations that handle such formulas. In this paper, we generalize the structure of counting formulas and present a set of inference operators that introduce and eliminate these formulas from the model. This generalization expands the range of problems that can be solved in a lifted way. Our work is closely related to the recently introduced method of joint conversion. Due to its more fine grained formulation, however, our approach can provide more efficient solutions than joint conversion.

1 Introduction

Probabilistic logical languages combine elements of first-order logic with graphical models to succinctly model complex, uncertain, structured domains [4]. These domains often involve a large number of objects, making efficient inference a major challenge. To address this problem, Poole [7] introduced the concept of *lifted probabilistic inference*, i.e., inference that exploits the symmetries in the structure of the model to gain efficiency (for an overview, see [5]). Lifted inference methods use two main techniques or tools for *lifting*:

1. **Isomorphic decomposition**: *decompose* the problem into *isomorphic* subproblems, solve one instance, and aggregate the result
2. **Counting**: *Count* the number of isomorphic configurations for a group of *interchangeable* variables instead of enumerating all possible configurations.

Our focus in this paper is on the second tool, counting, in the context of lifted variable elimination (LVE) [3,6,7].

LVE uses *counting formulas* to capture the interchangeability among objects [6]. A counting formula aggregates the joint state of a group of random variables into *histograms* that show only the *number* of variables with each state, without distinguishing between the individuals. For instance, the formula $\#_X[Attends(X)]$, captures the number of people who attend a workshop, without distinguishing between their identity. As the number of possible aggregate states (histograms) is much smaller than the number of joint states of the group, lifted operations achieve

G. Zaverucha et al. (Eds.): ILP 2013, LNAI 8812, pp. 107–122, 2014.
DOI: 10.1007/978-3-662-44923-3_8

large efficiency gains by directly manipulating counting formulas, instead of the individual variables.

Counting formulas, as used so far in LVE [6], have specific syntactic restrictions. Some of these are not fundamental and can be removed, yielding more opportunities for lifting. One such restriction is that a counting formula contains only a single atom, i.e., it aggregates the states of a group of *individual* random variables. In this paper, we generalize the definition of counting formulas, allowing them to aggregate the states of a group of *tuples* of random variables; e.g., $\#_X[Attends(X), Presents(X)]$ counts the number of people that (do not) attend *and* (do not) present a paper at the workshop. We present a set of inference operators that introduce and manipulate these generalized formulas, and show that these expand the opportunities for lifting, and hence for more efficient inference, compared to the original formulation of counting operations.

Our work is closely related to that of *joint conversion* and *just-different* counting conversion [1]. However, our method uses a more fine grained formulation, and can offer more efficient solutions than joint conversion.

2 Representation

Many representation languages for probabilistic logical models have been proposed [4]. Like earlier work on LVE [2,6,7], we use a representation based on *parametrized random variables* and *parametric factors* [7]. This representation combines random variables and factors (as used in factor graphs) with concepts from logic. The goal is to compactly define complex probability distributions over many variables. We now introduce the necessary terminology.

We use the term 'variable' in both the logical and probabilistic sense. We use *logvar* for logical variables and *randvar* for random variables. We write variables in uppercase and values in lowercase.

Preliminaries. A *factor* $f = \phi_f(\mathcal{A}_f)$, where $\mathcal{A}_f = (A_1, \ldots, A_n)$ are randvars and ϕ_f is a *potential* function, maps each configuration of $\mathcal{A}_f$ to a real. A *factor graph* is a set of factors F over randvars $\mathcal{A} = \bigcup_{f \in F} \mathcal{A}_f$ and defines a probability distribution $\mathcal{P}_F(\mathcal{A}) = \frac{1}{Z} \prod_{f \in F} \phi_f(\mathcal{A}_f)$, with Z a normalization constant.

The vocabulary consists of *predicates* (representing properties and relations), *constants* (representing objects) and *logvars*. A *constant* represents an object in our universe. A *term* is a constant or a logvar. An *atom* is of the form $P(t_1, t_2, \ldots, t_n)$, where P is a predicate of arity n and each argument t_i is a term. An atom is *ground* if all its arguments are constants. Each logvar X has a finite *domain* $\mathcal{D}(X)$, which is a set of constants $\{x_1, \ldots, x_n\}$. A *constraint* C on a set of logvars $\mathbf{X} = \{X_1, \ldots, X_n\}$ is a conjunction of inequalities of the form $X_i \neq t$ where t is a constant in $\mathcal{D}(X_i)$ or a logvar in $\mathbf{X}$. A *substitution* $\theta = \{X_1 \rightarrow t_1, \ldots, X_n \rightarrow t_n\}$ maps logvars to terms. Applying a substitution θ to an atom (or term) a, replaces each occurrence of X_i in a with t_i; the result is denoted $a\theta$. When all t_i's are constants, θ is called a *grounding substitution*. Given a constraint C, we use $gr(\mathbf{X}|C)$ to denote the set of grounding substitutions to $\mathbf{X}$ that are consistent with C.

Parametrized randvars. The representation associates atoms with randvars. For this, every predicate is assigned a *range*, i.e., a set of possible values, e.g., $range(BloodType) = \{a, b, ab, o\}$. A ground atom then represents a randvar, e.g., $BloodType(joe)$. The randvar/atom has the same range as the involved predicate. Unlike in first-order logic, a range is not limited to $\{true, false\}$ but can be any finite set. To compactly encode distributions over many randvars, *Parametrized randvars (PRV)* were introduced [7]. A PRV is of the form $P(\mathbf{X})|C$, where $P(\mathbf{X})$ is an atom and C is a constraint on $\mathbf{X}$. A PRV $\mathcal{V} = P(\mathbf{X})|C$ represents (or *covers*) the set of randvars $RV(\mathcal{V}) = \{P(\mathbf{X})\theta | \theta \in gr(\mathbf{X}|C)\}$.

Example 1. Suppose $\mathcal{D}(X) = \mathcal{D}(Y) = \{a, b, c, d\}$, where a stands for the person *ann*, b for *bob*, etc. The PRV $Friends(X, Y)|X \neq Y$ represents a set of 12 randvars, namely $\{Friends(a, b), Friends(a, c), \ldots, Friends(d, c)\}$. Similarly, the (unconstrained) PRVs $Smokes(X)$ and $Drinks(X)$ each represent a set of 4 randvars.

Parametric factors (parfactors). Like PRVs compactly encode sets of randvars, *parfactors* compactly encode sets of factors. A parfactor is of the form $\forall \mathbf{L} : \phi(\mathcal{A})|C$, with $\mathbf{L}$ a set of logvars, C a constraint on $\mathbf{L}$, $\mathcal{A} = (A_i)_{i=1}^n$ a sequence of atoms parametrized with $\mathbf{L}$, and ϕ a potential function on $\mathcal{A}$. The set of logvars occurring in $\mathcal{A}$ is denoted $logvar(\mathcal{A})$, and we have $logvar(\mathcal{A}) \subseteq \mathbf{L}$. When $logvar(\mathcal{A}) = \mathbf{L}$, we omit $\mathbf{L}$ and write the parfactor as $\phi(\mathcal{A})|C$. A factor $\phi(\mathcal{A}')$ is called a *grounding* of a parfactor $\phi(\mathcal{A})|C$ if $\mathcal{A}'$ can be obtained by instantiating $\mathbf{L}$ according to a grounding substitution $\theta \in gr(\mathbf{L}|C)$. The set of all groundings of a parfactor g is denoted $gr(g)$.

Example 2. We abbreviate *Drinks* to D, *Friends* to F and *Smokes* to S. The parfactor $g_1 = \phi_1(S(X), F(X, Y), D(Y))|X \neq Y$ represents a set of 12 factors, namely $gr(g_1) = \{\phi_1(S(a), F(a, b), D(b)), \ldots, \phi_1(S(d), F(d, c), D(c))\}$. This parfactor can encode, for instance, that if X is a smoker and is friends with Y, then Y is likely to be a drinker.

Parfactor models. When talking about a *model* below, we mean a set of parfactors. In essence, a set of parfactors G is a compact way of defining a set of factors $F = \{f | f \in gr(g) \wedge g \in G\}$. The corresponding probability distribution is $\mathcal{P}_G(\mathcal{A}) = \frac{1}{Z} \prod_{f \in F} \phi_f(\mathcal{A}_f)$. By $G_1 \equiv G_2$, we denote that models G_1 and G_2 define the same probability distribution.

3 Lifted Variable Elimination

The state of art in lifted variable elimination (LVE) is the result of various complementary efforts [1,2,6,7,11–13]. In this section we briefly review the C-FOVE [6] algorithm, which forms the basis of our work. Another extension to C-FOVE, namely *joint formulas* [1], is discussed and compared with in Sect. 7.

Variable elimination calculates a marginal distribution by *eliminating* randvars in a specific order from the model until reaching the desired marginal [8].

To eliminate a single randvar V, it first *multiplies* all the factors containing V into a single factor and then *sums out* V from that single factor. LVE does this on a lifted level by eliminating parametrized randvars (i.e., whole groups of randvars) from parfactors (i.e., group of factors), using a set of *lifted* operators. **Lifted sum-out** sums-out a PRV, and hence all the randvars represented by that PRV, from the model. It is applicable only when each randvar represented by the PRV appears in exactly one grounding of exactly one parfactor in the model. The goal of all other operators is to manipulate the parfactors into a form that satisfies this precondition. In this sense, all operators except lifted sum-out are *enabling operators*. **Lifted multiplication** prepares the model for sum-out by replacing all the parfactors that share a particular PRV by a single equivalent product parfactor. It performs the equivalent of many factor multiplications in one lifted operation. **Splitting** and **shattering** rewrite the model into a *normal* form in which, e.g., each pair of PRVs represent either identical or disjoint randvars. This can enable subsequent lifted multiplication. **Counting conversion** introduces counting formulas in the model, to exploit *interchangeability* among a group of randvars. By replacing an atom such as $A(X)$ with a counting formula $\#_X[A(X)]$, we compactly represent and manipulate a single potential on randvars $A(x_1), \ldots, A(x_n)$. In C-FOVE, this operator is applicable on logvars that only appear in a single atom. We formally introduce and generalize counting formulas, along with operations that handle them, in the following sections.

4 Generalized Counting Formulas

Counting formulas aggregate the state of a group of interchangeable randvars into histograms that show the number of randvars with each value. They thus *lift* the computations to the level of the aggregate state of the group, without considering the individuals. This speeds up inference as the number of possible aggregate states (histograms) is polynomial in the group size, whereas the number of joint states is exponential. In this section, we generalize counting formulas, such that they aggregate the state of a group of *tuples* of atoms, instead of individual atoms. This permits lifting in cases where not all individual randvars are interchangeable, but specific tuples of randvars are.

4.1 A Motivating Example

The following example is representative for useful models for which C-FOVE does not have a lifted solution. We use it as a running example.

Example 3. Consider the model consisting of the two following parfactors (S stands for *Smokes*, F for *Friends*, D for *Drinks*).

$$g_1 = \phi_1(S(X), F(X,Y), S(Y)) \qquad g_2 = \phi_2(D(X), F(X,Y), D(Y))$$

This model is representative for models that express (anti-)homophily between linked entities w.r.t. *multiple* properties (in this case, *Smokes* and *Drinks*).

Consider summing out all the randvars. C-FOVE can eliminate the F randvars by multiplying g_1 and g_2 into $g_{12} = \phi_{12}(S(X), D(X), F(X,Y), S(Y), D(Y))$, then summing out the F atoms: $g'_{12} = \phi'_{12}(S(X), D(X), S(Y), D(Y))$. After this, no other lifted operations are applicable on the model: lifted sum-out is not applicable for any of the atoms (none of them contains all the logvars in the parfactor), and counting conversion is not applicable on any of the logvars (there is no logvar that appears in only one atom). The only option left for C-FOVE is to ground one of the logvars and work with the (much larger) resulting model.

The reason counting conversion is not applicable here is that each logvar appears in two atoms, while C-FOVE's formulation of counting formulas (and conversion) are only applicable on a single atom. Intuitively, lifting is not possible with the existing counting tools since we cannot rewrite the model equivalently in terms of two separate formulas $\#_X[S(X)]$ and $\#_X[D(X)]$. These formulas tell us how many people smoke and how many people drink; to evaluate the model we need to know how many people (do not) smoke *and* (do not) drink. In the following, we show how this problem can be solved in a lifted way, by rewriting the model in terms of counting formulas of the form $\#_X[S(X), D(X)]$, which count *tuples* of atoms and provide sufficient information to evaluate the model.

4.2 Definition

We define a counting formula (CF) to be of the form $\gamma = \#_{X:C}[P_1(\mathbf{X}_1), \ldots, P_k(\mathbf{X}_k)]$, with C a constraint on the *counted logvar* X, and $X \in \mathbf{X}_i (i = 1, \ldots, k)$. The logvar X is bound by the formula and excluded from $logvar(\gamma)$. In a *grounded CF* all terms except the counted logvar are constants. Such a formula represents a *counting randvar (CRV)* whose range is the set of possible histograms that distribute n elements into $r = \prod_{i=1}^{k} |range(P_i)|$ buckets. Each histogram $h = \{(r_i, n_i)\}_{i=1}^{r}$ shows for each $r_i \in \times_{i=1}^{k} range(P_i)$ the number n_i of tuples $(P_1(\ldots, x, \ldots), \ldots, P_k(\ldots, x, \ldots))$ whose state is r_i. The state of the CRV is thus determined by the state of the randvars $\cup_{i=1}^{k} RV(P_i(\ldots, X, \ldots)|C)$.

We write a histogram $h = \{(r_i, n_i)\}_{i=1}^{r}$ as a list of counts $(n_1, \ldots, n_r)$, when r_i are apparent from the context. Further, we abbreviate $\{true, false\}$ to $\{t, f\}$.

Example 4. Having $D(X) = \{ann, bob, carl, dave\}$, the counting randvar γ of the form $\#_X[Smokes(X), Asthma(X)]$ covers tuples of randvars

$$RV(\gamma) = \{(Smokes(x_i), Asthma(x_i))|x_i \in D(X)\}.$$

Assume $RV(\gamma)$ are assigned the following values:

The value of the CRV is then the histogram $h = \{(tt, 1), (tf, 0), (ft, 1), (ff, 2)\}$. □

Our definition of a CF is a simple generalization of Milch et al.'s definition [6], which only allows a single atom in a CF. We show how these more general CFs create more opportunities for lifting through a suite of model conversion operations, in Sect. 5, and present a lifted sum-out operation for them in Sect. 6.

X	$Smokes(X)$	$Asthma(X)$
ann	f	t
bob	f	f
carl	f	f
dave	t	t

But first, we recall a *normal form* for the model, which guarantees correct semantics, and facilitates handling of counting formulas [6].

Normal parfactors. As mentioned, the range of a counting formula depends on the number of randvars that it counts, which in turn depends on its associated constraint. For instance, consider the formula $\#_{Y:\{Y\neq X,Y\neq x_1\}}[F(X,Y)]$. For $X = x_1$, it represents a CRV that counts $n-1$ randvars (Y can take on all n values except x_1), but for $X = x' \neq x_1$ it represents a CRV that counts $n-2$ randvars (Y can take on all n values except x_1 and x'). To ensure that all CRVs in the groundings of a parfactor have the same range, following Milch et al. [6], we require the constraints and parfactors to be in a *normal form*. A constraint C is in normal form if for each pair of logvars X_1 and X_2 with an inequality constraint $X_1 \neq X_2$, $E_{X_1} \setminus \{X_2\} = E_{X_2} \setminus \{X_1\}$, where the *excluded* set $E_X = \{t \,|(X \neq t) \in C\}$ is the set of terms in an inequality constraint with X in constraint C. This guarantees that a logvar X_i has the same number of values in C, given any value to the rest of the logvars. This number is denoted $|X : C|$ and equals $|\mathcal{D}(X)| - |E_X|$. A parfactor is in normal form if the conjunct of its constraint with the constraints of its counting formulas is in normal form. Any model can be rewritten into an equivalent one in normal form with splitting [6].

Example 5. Consider the parfactor $g = \phi(\,\#_{Y:\{Y\neq X,Y\neq x_1\}}[F(X,Y)]\,)$, which is not in normal form. In order to get normal form parfactors, we split g into two parfactors g_1 and g_2, defined as follows.

$$g_1 = \phi\big(\#_{Y:\{Y\neq x_1\}}[F(x_1,Y)]\big)$$
$$g_2 = \phi\big(\#_{Y:\{Y\neq X,Y\neq x_1\}}[F(X,Y)]\big)\,|X \neq x_1$$

Note that both g_1 and g_2 are in normal form. □

5 Conversion Operations

In this section, we present conversion operations that rewrite the model in terms of counting formulas. The first operation is a generalization of C-FOVE's counting conversion [6], while the rest are new operations. Throughout this paper, we illustrate the application of the new operators on examples for which C-FOVE [6] has no lifted solution. As such, we show how the new operations perform inference with complexity polynomial in the domain size, where C-FOVE cannot avoid the exponential complexity of propositional inference.

5.1 Counting Conversion

In C-FOVE, counting formulas are introduced into the model by *counting conversion*. By rewriting the model (replacing an *atom*) with a counting formula, this operation allows us to compactly represent and manipulate a high dimensional factor on a set of interchangeable randvars. Intuitively, this conversion achieves the equivalent of multiplying groundings of a single parfactor with each other, and thus functions as an enabling operation for lifted sum-out. We generalize this operation to a rewrite rule that replaces a *tuple* of atoms with a counting formula. By removing the restriction on the number of counted atoms, this generalization provides more opportunities for lifting.

Example 6. Consider the model consisting of the single parfactor

$$\phi(A(X), B(X), C(Y), D(Y)).$$

Lifted sum-out is not applicable here, as no atom contains all the logvars. Counting conversion removes a logvar from the set of free logvars, hence preparing the model for lifted sum-out. Here we perform counting conversion on logvar X by introducing a counting formula on the tuple of atoms $A(X), B(X)$, and rewrite the parfactor as $\phi'(\#_X[A(X), B(X)], C(Y), D(Y))$, where ϕ' is such that for each histogram $h(.)$ in the range of $\#_X[A(X), B(X)]$, $\phi'(h(.), c, d)$ is equal to

$$\phi(t,t,c,d)^{h(tt)} \cdot \phi(t,f,c,d)^{h(tf)} \cdot \phi(f,t,c,d)^{h(ft)} \cdot \phi(f,f,c,d)^{h(ff)}$$

Note that after this conversion, logvar Y is the only free logvar in the parfactor, which enables lifted sum-out of both $C(Y)$ and $D(Y)$ from the model.[1] □

Example 7. The same technique for conversion can be applied on the model introduced in Example 3. With two applications of counting conversion, on logvars X and Y, we can rewrite the problematic parfactor $g'_{12} = \phi'_{12}(S(X), D(X), S(Y), D(Y))$ into the form

$$g''_{12} = \phi''_{12}(\#_X[S(X), D(X)], \#_Y[S(Y), D(Y)]),$$

consisting of two counting formulas. □

Counting conversion is formally defined in Operator 1. The preconditions for counting conversion of a logvar X require that it does not appear inside an existing counting formula. This means that X cannot appear (1) in an atom inside a counting formula, or (2) in the constraint associated with a counting formula. Excluding the first case ensures that the result of conversion can be represented by our counting formulas, and does not require more complicated structures like nested or overlapping counting formulas, which are not well defined in our formulation. In the second case, where X is in an inequality constraint with a counted logvar, counting conversion is still possible, but by the operation of *merge-counting*, which will be introduced in Sect. 5.3. Note that the precondition for counting conversion as defined above, is weaker than the original one of C-FOVE, which requires X to appear in exactly one atom in the parfactor [6].

[1] In Sect. 6.1, we introduce a sum-out operator that can eliminate $A(X)$ and $B(X)$ in a lifted way.

Operator: COUNT-CONVERT
Input:
(1) a parfactor $g = \forall \mathbf{L} : \phi(\mathcal{A})|C$
(2) a logvar $X \in logvar(\mathcal{A})$
Preconditions:
(1) there is no counting formula in the set $\mathcal{A}_X = \{A \in \mathcal{A} | X \in logvar(A)\}$
(2) there is no counting formula $\gamma = \#_{X_i:C_i}[\dots]$ in $\mathcal{A}$, such that $(X_i \neq X) \in C_i$
Output: $g' = \forall \mathbf{L}' : \phi'(\mathcal{A}')|C'$, such that:
(1) $\mathbf{L}' = \mathbf{L} \setminus \{X\}$
(2) C' is the projection of C on $\mathbf{L}'$
(3) $\mathcal{A}' = \mathcal{A} \setminus \mathcal{A}_X \cup \{\#_X[\mathcal{A}_X]\}$, and
(4) for each valuation $(h(.), \mathbf{a})$ to $(\#_X[\mathcal{A}_X], \mathcal{A} \setminus \mathcal{A}_X)$:

$$\phi'(h(.), \mathbf{a}) = \prod_{\mathbf{a}' \in range(\mathcal{A}_X)} \phi(\mathbf{a}'; \mathbf{a})^{h(\mathbf{a}')}$$

Postcondition: $G \equiv G \setminus \{g\} \cup \{\text{COUNT-CONVERT}(g, X)\}$

Operator 1: The generalized counting conversion operator.

5.2 Merging Counting Formulas

Two counting formulas can count over tuples of randvars with overlapping randvars between them (see for instance, the parfactor in Example 7). When such formulas appear in a parfactor together, to sum-out the common randvars, we need to first merge these counting formulas into one.

Example 8. Consider the two counting formulas in the parfactor

$$\phi(\#_X[S(X)], \#_Y[S(Y), D(Y)]).$$

The first formula, $\gamma_1 = \#_X[S(X)]$, aggregates the state of randvars $RV(S(X))$, and the second formula, $\gamma_2 = \#_Y[S(Y), D(Y)]$, the state of $RV(S(X), D(X))$. As the second group of randvars is a superset of the first set, given a histogram $h_2(.)$ for γ_2, we can infer the value h_1 of γ_1. We can therefore merge the two counting formulas into one $\#_X[S(X), D(X)]$ and rewrite the parfactor as $\phi'(\#_X[S(X), D(X)])$ where $\phi'(h_2) = \phi(h_1, h_2)$ and h_1 is the histogram that results from projecting h_2 on the assignments to the $S()$ randvars. Concretely, having the counts $(n_{tt}, n_{tf}, n_{ft}, n_{ff})$ for h_2, the value of h_1 is the histogram with counts $(n_{tt} + n_{tf}, n_{ft} + n_{ff})$. □

Example 9. Consider again our running example (Example 7). We can use the above merging technique to rewrite the parfactor

$$g''_{12} = \phi''_{12}(\#_X[S(X), D(X)], \#_Y[S(Y), D(Y)])$$

into the form

$$g'''_{12} = \phi'''_{12}(\#_X[S(X), D(X)]),$$

Operator: MERGE
Input:
(1) a parfactor $g = \forall \mathbf{L} : \phi(\mathcal{A})|C$
(2) a pair of counting formulas $(\gamma_1, \gamma_2) = (\#_{X_1:C_1}[\mathcal{A}_1], \#_{X_2:C_2}[\mathcal{A}_2])$ in $\mathcal{A}$
Precondition: $gr(X_1|C \wedge C_1) = gr(X_2|C \wedge C_2)$
Output: $g' = \forall \mathbf{L} : \phi'(\mathcal{A}')|C$, such that:
(1) $\mathcal{A}' = \mathcal{A} \setminus \{\gamma_1, \gamma_2\} \cup \{\#_{X_1}[\mathcal{A}_{12}]\}$, with $\mathcal{A}_{12} = \mathcal{A}_1 \cup \mathcal{A}_2\theta$ and $\theta = \{X_2 \to X_1\}$
(2) for each valuation $(h(.), \mathbf{a})$ to $(\#_{X_1}[\mathcal{A}_{12}], \mathcal{A} \setminus \{\gamma_1, \gamma_2\})$:

$$\phi'(h, \mathbf{a}) = \phi(h_{[\mathcal{A}_1]}, h_{[\mathcal{A}_2\theta]}; \mathbf{a})$$

Postcondition: $G \equiv G \setminus \{g\} \cup \{\text{MERGE}(g, \gamma_1, \gamma_2)\}$

Operator 2: The merging operation for two counting formulas.

where for each histogram h in the range of the counting formula $\#_X[S(X), D(X)]$:

$$\phi'''_{12}(h) = \phi''_{12}(h, h).$$

Although in the above examples one CRV is a superset of the other, merging can be applied to any pair of formulas with overlapping randvars. For instance, by merging, the parfactor $\phi(\#_X[A(X), B(X)], \#_Y[B(Y), C(Y))])$ is transformed into $\phi(\#_X[A(X), B(X), C(X)])$. To formalize this operator we introduce the notion of *compatible* valuations to atoms, and then the *projection* of histograms.

Compatible valuations. A pair of valuations $(\mathbf{a}_1, \mathbf{a}_2)$ to $(\mathcal{A}_1, \mathcal{A}_2)$ are *compatible*, denoted by $\mathbf{a}_1 \sim \mathbf{a}_2$, if each atom $A_i \in \mathcal{A}_1 \cap \mathcal{A}_2$ is assigned with the same value a_i in both $\mathbf{a}_1$ and $\mathbf{a}_2$.

Projection of histograms. Given a counting formula $\gamma = \#_X[\mathcal{A}]$, the *projection* of a histogram $h \in range(\gamma)$ on $\mathcal{A}' \subseteq \mathcal{A}$, is a histogram $h' \in range(\#_X[\mathcal{A}'])$ such that for each $\mathbf{a}' \in range(\mathcal{A}')$: $h'(\mathbf{a}') = \sum_{\mathbf{a} \sim \mathbf{a}'} h(\mathbf{a})$. We denote the projection of h on $\mathcal{A}'$ by $h_{[\mathcal{A}']}$.

Using these definitions, Operator 2 defines the merge operation.

5.3 Merge-Counting

This operation is applicable when counting conversion cannot replace a tuple of atoms by a new counting formula, but needs to merge them into an existing counting formula. Concretely, this happens during counting conversion on a logvar X, in a parfactor with a counting formula $\#_{Y:Y \neq X}[\mathcal{A}_Y]$, that is, when an existing counted logvar is in an inequality constraint with X. In such cases, we cannot convert the parfactor to an equivalent one with two separate counting formulas $\#_Y[\mathcal{A}_Y]$ and $\#_X[\mathcal{A}_X]$. Intuitively, this is because the histograms of these two counting formulas do not determine the value of the original parfactor. Instead, we can apply *merge-counting*, which incorporates the atoms $\mathcal{A}_X$ inside the existing counting formula. Consider the following example.

Example 10. Consider the parfactor g of the form $\phi(\#_{X:X\neq Y}[S(X)], D(Y))$. We show how merge-counting on logvar Y rewrites g as an equivalent parfactor $g' = \phi'(\#_X[S(X), D(X)])$. For this we need to properly define the potential ϕ' based on ϕ. Note that g' represents a single factor, while g represents n factors, one for each $y \in \mathcal{D}(Y)$. The potential ϕ' should thus be defined such that g' evaluates the product of these n factors, for any valuation of randvars $S(X)$ and $D(X)$. Consider a valuation that yields histogram $h(.)$ for $\#_X[S(X), D(X)]$. To compute $\phi'(h)$, we compute the value of g at this valuation, based on the following observations: in $gr(g)$, each factor $g_y = \phi(\gamma_y, D(y))$, has a *distinct* counting formula $\gamma_y = \#_{X:X\neq y}[S(X)]$, which covers all the randvars $RV(S(X))$ *except* $S(y)$, due to the inequality constraint. Since the histogram of $\#_X[S(X)]$ is $h_{[S]} = (n_t, n_f)$, each CRV γ_y, which excludes the value of one randvar $S(y)$, takes on one of the two histograms

- $h^{-t}_{[S]} = (n_t - 1, n_f)$, when $S(y) = t$, or
- $h^{-f}_{[S]} = (n_t, n_f - 1)$, when $S(y) = f$

Each factor g_y in $gr(g)$ thus evaluates to one of the four values $\phi(h^-, d)$, for $(h^-, d) \in \{h^{-t}_{[S]}, h^{-f}_{[S]}\} \times \{t, f\}$. Knowing the number $\#(h^-, d)$ of factors with each value $\phi(h^-, d)$, we can compute the desired potential ϕ' as:

$$\phi'(h(.)) = \prod_{(h^-,d)} \phi(h^-, d)^{\#(h^-,d)} \tag{1}$$

The numbers $\#(h^-, d)$ are inferred directly from the histogram $h(.)$. For instance, $\#(h^{-t}_{[S]}, t)$, the number of ys with $\gamma_y = h^{-t}_{[S]}$ and $D(y) = t$, by definition equals $h(tt)$, that is, the number of ys with $S(y) = D(y) = t$. With similar reasoning, we determine all the numbers $\#(h^-, d)$ from $h(.)$ and define the desired potential ϕ' as:

$$\phi'(h(.)) \;=\; \phi(h^{-t}_{[S]}, t)^{h(tt)} \;\cdot\; \phi(h^{-f}_{[S]}, t)^{h(ft)} \;\cdot\; \phi(h^{-t}_{[S]}, f)^{h(tf)} \;\cdot\; \phi(h^{-f}_{[S]}, f)^{h(ff)}$$

As such, merge-counting replaces the parfactor g with the equivalent parfactor $g' = \phi'(\#_X[S(X), D(X)])$, by directly computing ϕ' from ϕ. □

We can now also see why a counting conversion like $\phi'(\#_X[S(X)], \#_Y[D(Y)])$ does not work on this example. For this conversion, we need to be able to compute the right hand side of Eq. 1 based on the pair of histograms (h_S, h_D) of the two counting formulas $(\#_X[S(X)], \#_Y[D(Y)])$. However, we cannot determine the numbers $\#(h^-_S, d)$ uniquely based on the pair of histograms (h_S, h_D). For instance, assume $|\mathcal{D}(X)| = |\mathcal{D}(Y)| = 10$, and that $h_S = (10, 0)$ and $h_D = (5, 5)$. Then the number $\#(h^{-t}_S, t)$ of factors with value $\phi(h^{-t}_S, t)$, which is the number of y's for which $S(y) = D(y) = true$, can be any number between 0 and 5.

Merge-counting is formally defined in Operator 3. Note that the second precondition can be established by *merging* counting formulas. Merging is thus an enabling operator for this conversion operator.

Operator: MERGE-COUNT
Input:
(1) a parfactor $g = \forall \mathbf{L} : \phi(\mathcal{A})|C$
(2) a counting formula $\gamma = \#_{X_1:C_1}[\mathcal{A}_1]$ in $\mathcal{A}$
(3) a logvar X_2 in $logvar(\mathcal{A})$
Preconditions:
(1) there is no counting formula in $\mathcal{A}_2 = \{A \in \mathcal{A} | X_2 \in logvar(A)\}$
(2) γ is the only counting formula whose counted logvar X_1 is in an inequality constraint with X_2
Output: $g' = \forall \mathbf{L}' : \phi'(\mathcal{A}')|C'$, such that:
(1) $\mathbf{L}' = \mathbf{L} \setminus \{X_2\}$
(2) C' is the projection of C on $\mathbf{L}'$
(3) $\mathcal{A}' = \mathcal{A} \setminus \{\gamma, \mathcal{A}_2\} \cup \{\#_{X_1:C_{12}}[\mathcal{A}_{12}]\}$, with $\mathcal{A}_{12} = \mathcal{A}_1 \cup \mathcal{A}_2\{X_2 \to X_1\}$, and $C_{12} = C_1 \setminus (X_1 \neq X_2)$
(4) for each valuation $(h(.), \mathbf{a})$ to $(\#_{X_1:C_{12}}[\mathcal{A}_{12}], \mathcal{A}' \setminus \#_{X_1:C_{12}}[\mathcal{A}_{12}])$:

$$\phi'(h(.), \mathbf{a}) = \prod_{(\mathbf{a}_{12}) \in range(\mathcal{A}_{12})} \phi(h^{-\mathbf{a}_1}_{[\mathcal{A}_1]}, \mathbf{a}_2; \mathbf{a})^{h(\mathbf{a}_{12})}$$

where $\mathbf{a}_i$ denotes the projection of the valuation $\mathbf{a}_{12}$ on $\mathcal{A}_i\{X_2 \to X_1\}$, and the histogram $h^{-\mathbf{r}}$ is such that $h^{-\mathbf{r}}(\mathbf{r}) = h(\mathbf{r}) - 1$, and $h^{-\mathbf{r}}(\mathbf{r}') = h(\mathbf{r}')$, for $\mathbf{r}' \neq \mathbf{r}$.
Postcondition: $G \equiv G \setminus \{g\} \cup \{\text{MERGE-COUNT}(g, \#_{X_1:C_1}[\mathcal{A}_1], X_2)\}$

Operator 3: The merge-counting operation.

6 Elimination Operations

In the previous section, we presented the operators that introduce or merge counting formulas. In this section, we present a lifted *sum-out* operator that eliminates these formulas, as well as an operator that aggregates the results after sum-out.

6.1 Sum-out by Counting

We present an operation for summing out an atom inside counting formulas. This lifted operation sums out all the randvars represented by the atom from the model. It functions as a rewrite rule that removes the atom from a counting formula, and has the sum-out operation of C-FOVE as a special case.

Example 11. Consider summing-out the PRV $S(X)$ from the model defined by the parfactor $\phi(\#_X[S(X), D(X)])$, from Example 9. By lifted sum-out we derive the parfactor $\phi'(\#_X[D(X)])$, for which we define the potential function ϕ' such that for each histogram $h' \in range(\#_X[D(X)])$:

$$\phi'(h') = \sum_{h \sim h'} \text{MUL}(h|h')\phi(h)$$

where $h \sim h'$ denotes a histogram $h \in range(\#_X[S(X), D(X)])$ whose projection on $D(X)$ equals h', that is, $h_{[D]} = h'$. The coefficient $\text{MUL}(h|h')$ equals the

Operator: SUM-OUT
Input:
(1) a parfactor $g = \forall \mathbf{L} : \phi(\mathcal{A})|C$ in model G
(2) a counting formula $\gamma = \#_{X_1:C_1}[\mathcal{A}_1]$ in $\mathcal{A}$
(3) an atom $A \in \mathcal{A}_1$
Precondition:
(1) $logvar(A) = \mathbf{L}$
(2) for all PRVs $(A'|C')$ in the model, other than $(A|C \wedge C_1)$:
$RV(A|C \wedge C_1) \cap RV(A'|C') = \emptyset$
Output: $g' = \forall \mathbf{L} : \phi'(\mathcal{A}')|C$, such that:
(1) $\mathcal{A}' = \mathcal{A} \setminus \{\gamma\} \cup \{\gamma'\}$, with $\gamma' = \#_{X_1:C_1}[\mathcal{A}_1 \setminus \{A\}]$
(2) for each valuation $(h'(.), \mathbf{a})$ to $(\gamma', \mathcal{A}' \setminus \gamma')$:

$$\phi'(h'(.), \mathbf{a}) = \sum_{h \sim h'} \textsc{Mul}(h|h') \cdot \phi(h(.), \mathbf{a})$$

where $h \sim h'$ denotes a histogram $h \in range(\gamma)$ such that $h_{[\mathcal{A}_1 \setminus \{A\}]} = h'$.
Postcondition: $\mathcal{P}_{G \setminus \{g\} \cup \{\text{SUM-OUT}(g,\gamma,A)\}} = \sum_{RV(A|C \wedge C_1)} \mathcal{P}_G$

Operator 4: The generalized counting sum-out operation.

number of possible ways that a valuation to $RV(D(X))$ with the counts h', can be extended to a valuation to $RV(S(X), D(X))$ with the counts h. □

The coefficient $\textsc{Mul}(h|h')$ is defined based on the number $\textsc{Mul}(h)$ of possible valuations to the randvars that result in a histogram h. For each histogram $h = \{(r_i, n_i)\}_{i=1}^{r}$, with $\sum_i n_i = n$, we define $\textsc{Mul}(h) = \frac{n!}{(n_1!)\ldots(n_r!)}$ and $\textsc{Mul}(h|h') = \frac{\textsc{Mul}(h)}{\textsc{Mul}(h')}$.

Note that sum-out removes an atom from the counting formula. Summing-out the last atom, such as $D(X)$ in the above example, results in an empty counting formula, for which we define the range as $\{(Null, 0)\}$, $\textsc{Mul}((Null, 0)) = 1$, and which we trivially remove from the list of arguments after sum-out. The operation is formally defined is Operator 4.

This operation has the sum-out operation of C-FOVE as a special case, namely when $\mathcal{A}_1 = \{A\}$. It can also be further generalized to sum-out a group of atoms $\{A_1, \ldots, A_n\} \subseteq \mathcal{A}$ in one operation, if the two preconditions are satisfied for all atoms.

6.2 Aggregation

In lifted inference, after dividing a problem into isomorphic subproblems, first the result of one prototypical instance of this problem is computed and then the result is *aggregated*, usually with the trivial operation of exponentiation. This operator in fact multiplies a group of identical factors and is applied when a logvar disappears from the parfactor after a sum-out operation. Aggregation can, however, be extended to cases where a simple exponentiation does not work. In

this section, we extend this operator and show how this allows for more efficient lifted computations.

Example 12. Consider the parfactor g of the form $\forall Y : \phi(\#_{X:X\neq Y}[S(X)], D(Y))$. Summing-out $D(Y)$ results in the parfactor $g' = \forall Y : \phi'(\#_{X:X\neq Y}[S(X)])$, on which sum-out is no longer applicable. Note that the logvar Y is still part of the parfactor, although it does not appear in any atom. We show how, by *aggregation*, we can rewrite g' as an equivalent parfactor $g'' = \phi''(\#_X[S(X)])$, which is free of logvar Y. Assume $\mathcal{D}(X) = \mathcal{D}(Y) = \{ann, bob, carl, dave\}$. Then g' represents four ground factors, one for each person in $\mathcal{D}(Y)$, e.g., for $Y = ann$ there is a factor $\phi'(\#_{X:X\neq ann}[S(X)])$ in $gr(g')$. g'' should be defined such that $\phi''(\#_X[S(X)])$ equals the product of these four factors. Note that these factors all have the same potential, but each on a CRV that excludes one distinct randvar from the group $R_S = \{S(ann), S(bob), S(carl), S(dave)\}$. Given any assignment to R_S with n_t true and n_f false randvars, each of these CRVs has either one less *true* than n_t or one less $false$ than n_f. Specifically, there are n_t histograms with counts $(n_t - 1, n_f)$ and n_f histograms with counts $(n_t, n_f - 1)$. Since the value $\phi'(h)$ is the same for all factors with the same histogram h, to compute the product it suffices to know (n_t, n_f). Aggregation rewrites g' as $\phi''(\#_X[S(X)])$, where the potential ϕ'' is such that:

$$\phi''((n_t, n_f)) = \phi'((n_t - 1, n_f))^{n_t} \cdot \phi'((n_t, n_f - 1))^{n_f}$$

With Y removed from g', we can eliminate $S(X)$ by lifted sum-out. □

The more expensive alternative to summing-out D and using aggregation is to apply counting on both atoms, and work with the parfactor $\phi^*(\#_X[S(X), D(X)])$. This alternative solution eliminates both S and D atoms with counting sum-out, in poly time, while the above solution uses counting only for the S atoms, and runs in linear time. We formalize aggregation in Operator 5.

7 Relation to Joint Conversion

Our contributions are closely related to, and target similar problems as, *joint conversion* and just-different counting conversion [1]. Our approach, however, can provide more efficient solutions than those based on the mentioned methods.

Joint conversion enables counting the states of a group of tuples of randvars, without modifying Milch et al.'s definition of counting formulas [6]. For instance, to enable counting *tuples* of randvars $(A(x), B(x))$, joint conversion replaces each occurrence of atoms $A(X)$ and $B(X)$ in the model with a joint atom $J_{AB}(X)$, whose state is the Cartesian product of the two atoms. Counting conversion can then derive a counting formula like $\#_X[J_{AB}(X)]$, which corresponds to a formula $\#_X[A(X), B(X)]$ in our formulation. When combined with just-different counting [1], joint conversion may also enable counting on logvars that are constrained to be unequal, similar to our approach.

However, there are differences between the two methods. Joint conversion is a global operation on the model, which introduces more dependencies by coupling

Operator: AGGREGATE
Input:
(1) a parfactor $g = \forall \mathbf{L} : \phi(\mathcal{A})|C$ in model G
(2) a counting formula $\gamma = \#_{X_1:C_1}[\mathcal{A}_1]$ in $\mathcal{A}$
(3) a logvar $X_2 \in \mathbf{L} \setminus logvar(\mathcal{A})$
Precondition:
$\mathcal{A}$ has no counting formula $\#_{X_i:C_i}[.]$ other than γ, such that $(X_i \neq X_2) \in C_i$
Output: $g' = \forall \mathbf{L}' : \phi'(\mathcal{A}')|C'$, such that:
(1) $\mathbf{L}' = \mathbf{L} \setminus \{X_2\}$
(2) C' is the projection of C on $\mathbf{L}'$
(3) $\mathcal{A}' = \mathcal{A} \setminus \{\gamma\} \cup \{\#_{X_1:C_1'}[\mathcal{A}_1]\}$, with $C_1' = C_1 \setminus \{X_1 \neq X_2\}$
(4) for each valuation $(h(.), \mathbf{a})$ to $(\#_{X_1:C_1'}[\mathcal{A}_1], \mathcal{A}' \setminus \{\#_{X_1:C_1'}[\mathcal{A}_1]\})$:

$$\phi'(h(.), \mathbf{a}) = \prod_{\mathbf{a}_1 \in range(\mathcal{A}_1)} \phi(h^{-\mathbf{a}_1}(.), \mathbf{a})^{h(\mathbf{a}_1)}$$

Postcondition: $G \equiv G \setminus \{g\} \cup \{\text{AGGREGATE}(g, \#_{X_1:C_1}[\mathcal{A}_1], X_2)\}$

Operator 5: The aggregation operation

two randvars into a joint randvar. After this operation, inference deals solely with the joint atom, and never directly with its constituents. Our method, however, uses a more fine grained formulation, by which it not only can simulate joint conversion, but also provide more efficient solutions than those possible by joint conversion. This primarily happens when the operations can divide the problem into independent parts, by eliminating a subset of the atoms that joint conversion couples in a joint atom. This allows for more efficient computations by avoiding the dependencies induced by unnecessary joint conversions. We illustrate this advantage in the following example.

Example 13. Consider a market domain involving a group of competing vendors $v_1, v_2, \ldots, v_n$. Let $P_i(v_j)$ be a binary randvar expressing whether at the ith time step (e.g., ith month) vendor v_j sets a high price for the product. We can model this domain for m time steps using the following set of parfactors.

$$\begin{aligned} g_1 &= \phi_1(P_1(X), P_2(Y))|X \neq Y \\ g_2 &= \phi_2(P_2(Y), P_3(X))|X \neq Y \\ &\ldots \\ g_m &= \phi_m(P_m(X), P_{m+1}(Y))|X \neq Y \end{aligned}$$

where $\mathcal{D}(X) = \mathcal{D}(Y) = \{v_1, \ldots, v_n\}$. The task is to compute the partition function in this model, that is, to sum-out all the random variables. □

Our approach. To sum out all the randvars, i.e., to eliminate all the atoms, our approach proceeds as follows. It first performs counting conversion on Y in g_1 to derive $g_1' = \phi'(P_1(X), \#_{Y:Y \neq X}[P_2(Y)])$. Next, it eliminates $P_1(X)$ from the model by lifted sum-out and aggregation, resulting in $g_1^* = \phi_1^*(\#_Y[P_2(Y)])$. To prepare the model for summing-out $P_2(Y)$, we perform the following operations:

1. COUNT-CONVERT on logvar Y in g_2 to derive $g_2' = \phi_2'(\#_{Y:Y\neq X}[P_2(Y)], P_3(X))$
2. MERGE-COUNT on logvar X in g_2' to derive $g_2'' = \phi_2''(\#_Y[P_2(Y), P_3(Y)])$
3. MULTIPLY g_1^* and g_2'' to derive $g_{12} = \phi_{12}(\#_Y[P_2(Y), P_3(Y)]), \#_Y[P_2(Y)])$
4. MERGE the counting formulas to derive $g_{12}' = \phi_{12}'(\#_Y[P_2(Y), P_3(Y)])$

Now from this parfactor, we sum-out $P_2(Y)$ and get a parfactor g_2^* of the form $\phi_{12}^*(\#_Y[P_3(Y)])$. Inference continues by eliminating P_3 from parfactors g_2^* and g_3, in a similar way as it eliminated P_2 from g_1^* and g_2. We repeat this procedure for all the remaining atoms P_i, until we eliminate the last atom P_{m+1}, which concludes the inference. The complexity of the procedure is proportional to the size of the largest potential it handles. For elimination of each atom P_i, the size of the largest potential we handle is $O(n^3)$, proportional to the number of histograms in the range of a counting formula $\#_X[P_i(X), P_{i+1}(X)]$. As there are m atoms P_i, the whole procedure is in time $O(mn^3)$.

Joint conversion. Any solution based on *joint conversion* and *just different* counting conversion is less efficient than the above method. Here we present one such typical solution. Joint conversion first replaces the atoms P_1 and P_2 with a *joint* atom J_{12}, which represents the joint state of the atoms. This changes g_1 and g_2 respectively into $\phi_1'(J_{12}(X), J_{12}(Y))|X \neq Y$ and $\phi_2'(J_{12}(Y), P_3(X))|X \neq Y$. Still J_{12} cannot be summed-out from the model, due to the free logvar X in g_2. Just-different conversion, to derive a $\phi_1''(\#_X[J_{12}(X)])$ is not helpful either. The only option is to continue applying joint conversions between atoms such as $J_{12\ldots k}$ and P_{k+1}, to finally have only one joint atom $J_{1\ldots m+1}$ in the model. Note that $range(J_{1\ldots m+1}) = \{true, false\}^{m+1}$. The model would then consist of parfactors of the form $\phi_i(J_{1\ldots m+1}(X), J_{1\ldots m+1}(Y))|X \neq Y$. By multiplying these m parfactors into one, and then just-different conversion, we derive a parfactor $g^* = \phi^*(\#_X[J_{1\ldots m+1}(X)])$. Finally we can sum-out the counting formula $\gamma^* = \#_X[J_{1\ldots m+1}]$ from g^*. The complexity of these operations is dominated by the manipulation of the counting formula γ^* and is proportional to $|range(\gamma^*)|$ which is $O(n^{2^m-1})$. Comparing this complexity to the complexity of our approach, $O(mn^3)$, we see that our method can be much more efficient. For example, for $m = 10$, our approach has complexity $O(10n^3)$, as opposed to $O(n^{1023})$.

8 Conclusion

Counting formulas and their manipulation play a crucial role in lifted inference. We showed how generalizing the structure of counting formulas and the operators that manipulate them, provides more opportunities for lifting. Our approach is closely related to joint conversion [1], but can offer more efficient solutions.

Further generalizations of counting formulas are conceivable. E.g., one can allow multiple counted logvars in a formula, instead of one. The semantics of such formulas is straightforward (e.g., $\#_{XY}[P(X,Y), Q(X,Y)]$ aggregates the joint state of the tuples $(P(x,y), Q(x,y))$), but manipulation of such formulas, especially counting the number of isomorphic states, can easily lead to non-trivial combinatorial problems. Future research on such problems can bring valuable

insights for lifted inference. A very interesting direction is Pu et al.'s formulation of the counting problem in the domain of exponential random graphs [9].

Taghipour [10] presents more results on generalized counting, excluded here for lack of space; these include a completeness result showing that C-FOVE with generalized counting is complete [14] for the class of monadic models.

Acknowledgments. Thanks to Daan Fierens for numerous discussions and insightful comments, Research Fund KU Leuven (GOA 08/008, CREA/11/015, OT/11/051), EU FP7 Marie Curie Career Integration Grant (#294068), FWO-Vlaanderen (G.0356.12).

References

1. Apsel, U., Brafman, R.I.: Extended lifted inference with joint formulas. In: Proceedings of the 27th Conference on Uncertainty in Artificial Intelligence (UAI), pp. 11–18 (2011)
2. de Salvo Braz, R.: Lifted First-order Probabilistic Inference. Ph.D. thesis, Department of Computer Science, University of Illinois at Urbana-Champaign (2007)
3. de Salvo Braz, R., Amir, E., Roth, D.: Lifted first-order probabilistic inference. In: Proceedings of the 19th International Joint Conference on Artificial Intelligence (IJCAI), pp. 1319–1325 (2005)
4. Getoor, L., Taskar, B. (eds.): An Introduction to Statistical Relational Learning. MIT Press, Cambridge (2007)
5. Kersting, K.: Lifted probabilistic inference. In: Proceedings of the 20th European Conference on Artificial Intelligence (ECAI), pp. 27–31 (2012)
6. Milch, B., Zettlemoyer, L.S., Kersting, K., Haimes, M., Kaelbling, L.P.: Lifted probabilistic inference with counting formulas. In: Proceedings of the 23rd AAAI Conference on Artificial Intelligence (AAAI), pp. 1062–1608 (2008)
7. Poole, D.: First-order probabilistic inference. In: Proceedings of the 18th International Joint Conference on Artificial Intelligence (IJCAI), pp. 985–991 (2003)
8. Poole, D., Zhang, N.L.: Exploiting contextual independence in probabilistic inference. J. Artif. Intell. Res. (JAIR) **18**, 263–313 (2003)
9. Pu, W., Choi, J., Amir, E.: Lifted inference on transitive relations. In: Proceedings of the 3rd International Workshop on Statistical Relational AI (StaRAI) (2013)
10. Taghipour, N.: Lifted Probabilistic Inference by Variable Elimination. Ph.D. thesis, Department of Computer Science, KU Leuven (2013)
11. Taghipour, N., Fierens, D., Davis, J., Blockeel, H.: Lifted variable elimination with arbitrary constraints. In: Proceedings of the 15th International Conference on Artificial Intelligence and Statistics (AISTATS), pp. 1194–1202 (2012)
12. Taghipour, N., Fierens, D., Davis, J., Blockeel, H.: Lifted variable elimination: decoupling the operators from the constraint language. J. Artif. Intell. Res. **47**, 393–439 (2013)
13. Taghipour, N., Fierens, D., Van den Broeck, G., Davis, J., Blockeel, H.: Completeness results for lifted variable elimination. In: Proceedings of the 16th International Conference on Artificial Intelligence and Statistics (AISTATS), pp. 572–580 (2013)
14. Van den Broeck, G.: On the completeness of first-order knowledge compilation for lifted probabilistic inference. In: Proceedings of the 24th Annual Conference on Advances in Neural Information Processing Systems (NIPS), pp. 1386–1394 (2011)

A FOIL-Like Method for Learning under Incompleteness and Vagueness

Francesca A. Lisi[1(✉)] and Umberto Straccia[2]

[1] Dipartimento di Informatica,
Università degli Studi di Bari "Aldo Moro", Bari, Italy
francesca.lisi@uniba.it
[2] ISTI - CNR, Pisa, Italy

Abstract. Incompleteness and vagueness are inherent properties of knowledge in several real world domains and are particularly pervading in those domains where entities could be better described in natural language. In order to deal with incomplete and vague structured knowledge, several fuzzy extensions of Description Logics (DLs) have been proposed in the literature. In this paper, we present a novel FOIL-like method for inducing fuzzy DL inclusion axioms from crisp DL knowledge bases and discuss the results obtained on a real-world case study in the tourism application domain also in comparison with related works.

1 Introduction

Motivation of the paper. Incompleteness and vagueness are inherent properties of knowledge in several real world domains and are particularly pervading in those domains where entities could be better described in natural language. The issues raised by incomplete and vague knowledge have been traditionally addressed in the field of Knowledge Representation (KR).

The *Open World Assumption* (OWA) is used in KR to codify the informal notion that in general no single agent or observer has complete knowledge. The OWA limits the kinds of inference and deductions an agent can make to those that follow from statements that are known to the agent to be true. In contrast, the *Closed World Assumption* (CWA) allows an agent to infer, from its lack of knowledge of a statement being true, anything that follows from that statement being false. Heuristically, the OWA applies when we represent knowledge within a system as we discover it, and where we cannot guarantee that we have discovered or will discover complete information. In the OWA, statements about knowledge that are not included in or inferred from the knowledge explicitly recorded in the system may be considered unknown, rather than wrong or false. Description Logics (DLs) are KR formalisms compliant with the OWA, thus turning out to be particularly suitable for representing *incomplete* knowledge [2]. Thanks to the OWA-compliancy, DLs have been considered the ideal starting point for the

G. Zaverucha et al. (Eds.): ILP 2013, LNAI 8812, pp. 123–139, 2014.
DOI: 10.1007/978-3-662-44923-3_9

definition of ontology languages for the Web (an inherently open world), giving raise to the OWL 2 standard.[1]

In many applications, it is important to equip DLs with expressive means that allow to describe "concrete qualities" of real-world objects such as the length of a car. The standard approach is to augment DLs with so-called *concrete domains*, which consist of a set (say, the set of real numbers in double precision) and a set of n-ary predicates (typically, $n = 1$) with a fixed extension over this set [3]. Starting from numerical properties such as the length one may want to deduce whether, *e.g.*, a car is long or not. However, it is well known that "classical" DLs are not appropriate to deal with *vague* knowledge [24]. We recall for the inexpert reader that there has been a long-lasting misunderstanding in the literature of artificial intelligence and uncertainty modelling, regarding the role of probability/possibility theory and vague/fuzzy theory. A clarifying paper is [8]. Specifically, under *uncertainty theory* fall all those approaches in which statements are true or false to some *probability* or *possibility* (for example, "it will rain tomorrow"). That is, a statement is true or false in any world/interpretation, but we are "uncertain" about which world to consider as the right one, and thus we speak about, *e.g.*, a probability distribution or a possibility distribution over the worlds. On the other hand, under *fuzzy theory* fall all those approaches in which statements (for example, "the car is long") are true to some *degree*, which is taken from a truth space (usually $[0, 1]$). That is, an interpretation maps a statement to a truth degree, since we are unable to establish whether a statement is entirely true or false due to the involvement of vague concepts, such as "long car" (the degree to which the sentence is true depends on the length of the car). Here, we shall focus on fuzzy logic only.

Contribution of the paper. Although a relatively important amount of work has been carried out in the last years concerning the use of fuzzy DLs as ontology languages [26], the problem of automatically managing the evolution of fuzzy ontologies by applying machine learning algorithms still remains relatively unaddressed [11,13,17]. In this paper, we present a novel method, named FOIL-$\mathcal{DL}$, for learning fuzzy DL inclusion axioms from any crisp DL knowledge base. The popular rule induction method FOIL [19] has been chosen as a starting point in our proposal for its simplicity and efficiency. The distinguishing feature of FOIL-$\mathcal{DL}$ w.r.t. previous work in DL learning (see, *e.g.*, [9,15,16]) is the treatment of numerical concrete domains with fuzzification techniques so that the induced axioms may contain fuzzy concepts.

Structure of the paper. For the sake of self-containment, Sect. 2 introduces some basic definitions we rely on. Section 3 provides a formal statement of the learning problem solved by FOIL-$\mathcal{DL}$ and details of the distinguishing features of FOIL-$\mathcal{DL}$ w.r.t. FOIL. Section 4 discusses relevant literature. Section 5 illustrates some experimental results obtained on a real-world case study in the tourism application domain. Section 6 concludes the paper with final remarks on the current work and possible directions of future work.

[1] http://www.w3.org/TR/2009/REC-owl2-overview-20091027/

2 Preliminaries

Description Logics. For the sake of illustrative purposes, we present here a salient representative of the DL family, namely $\mathcal{ALC}$ [21], which is often considered to illustrate some new notions related to DLs. The set of constructors for $\mathcal{ALC}$ is reported in Table 1. A DL *Knowledge Base* (KB) $\mathcal{K} = \langle \mathcal{T}, \mathcal{A} \rangle$ is a pair where $\mathcal{T}$ is the so-called *Terminological Box* (TBox) and $\mathcal{A}$ is the so-called *Assertional Box* (ABox). The TBox is a finite set of *General Concept Inclusion* (GCI) axioms which represent is-a relations between concepts, whereas the ABox is a finite set of *assertions* (or *facts*) that represent instance-of relations between individuals (resp. couples of individuals) and concepts (resp. roles). Thus, when a DL-based ontology language is adopted, an ontology is nothing else than a TBox (*i.e.*, the intensional level of knowledge), and a populated ontology corresponds to a whole KB (*i.e.*, encompassing also an ABox, that is, the extensional level of knowledge). We also introduce two well-known DL macros, namely (i) *domain restriction*, denoted $domain(R, A)$, which is a macro for the GCI $\exists R.\top \sqsubseteq A$, and states that the domain of the abstract role R is the atomic concept A; and (ii) *range restriction*, denoted $range(R, A)$, which is a macro for the GCI $\top \sqsubseteq \forall R.A$, and states that the range of R is A. Finally, in $\mathcal{ALC}(\mathbf{D})$ (obtained by enriching $\mathcal{ALC}$ with concrete domains $\mathbf{D}$), each role is either *abstract* (denoted with R) or *concrete* (denoted with T). A new concept constructor is then introduced, which allows to describe constraints on concrete values using predicates from the concrete domain. We shall make further clarifications about the notion of concrete domains later on in this Section while presenting fuzzy $\mathcal{ALC}(\mathbf{D})$.

Table 1. Syntax and semantics of constructs for $\mathcal{ALC}$.

bottom (resp. top) concept	$\bot$ (resp. $\top$)	$\emptyset$ (resp. $\Delta^{\mathcal{I}}$)
atomic concept	A	$A^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}$
(abstract) role	R	$R^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$
individual	a	$a^{\mathcal{I}} \in \Delta^{\mathcal{I}}$
concept intersection	$C \sqcap D$	$C^{\mathcal{I}} \cap D^{\mathcal{I}}$
concept union	$C \sqcup D$	$C^{\mathcal{I}} \cup D^{\mathcal{I}}$
concept negation	$\neg C$	$\Delta^{\mathcal{I}} \setminus C^{\mathcal{I}}$
universal role restriction	$\forall R.C$	$\{x \in \Delta^{\mathcal{I}} \mid \forall y\ (x, y) \in R^{\mathcal{I}} \rightarrow y \in C^{\mathcal{I}}\}$
existential role restriction	$\exists R.C$	$\{x \in \Delta^{\mathcal{I}} \mid \exists y\ (x, y) \in R^{\mathcal{I}} \wedge y \in C^{\mathcal{I}}\}$
general concept inclusion	$C \sqsubseteq D$	$C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$
concept assertion	$a : C$	$a^{\mathcal{I}} \in C^{\mathcal{I}}$
role assertion	$(a, b) : R$	$(a^{\mathcal{I}}, b^{\mathcal{I}}) \in R^{\mathcal{I}}$

The semantics of DLs can be defined directly with set-theoretic formalizations (as shown in Table 1 for the case of $\mathcal{ALC}$) or through a mapping to FOL (as shown in [5]). Specifically, an *interpretation* $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$ for a DL KB consists of a domain $\Delta^{\mathcal{I}}$ and a mapping function $\cdot^{\mathcal{I}}$. For instance, $\mathcal{I}$ maps a concept C into a set of individuals $C^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}$, *i.e.* $\mathcal{I}$ maps C into a function $C^{\mathcal{I}} : \Delta^{\mathcal{I}} \rightarrow \{0, 1\}$ (either an individual belongs to the extension of C or does not belong to it).

Under the *Unique Names Assumption* (UNA) [20], individuals are mapped to elements of $\Delta^{\mathcal{I}}$ such that $a^{\mathcal{I}} \neq b^{\mathcal{I}}$ if $a \neq b$. However UNA does not hold by default in DLs. An interpretation $\mathcal{I}$ is a *model* of a KB $\mathcal{K}$ iff it satisfies all axioms and assertions in $\mathcal{T}$ and $\mathcal{A}$. In DLs a KB represents many different interpretations, *i.e.* all its models. This is coherent with the OWA that holds in FOL semantics. A DL KB is *satisfiable* if it has at least one model. We also write $C \sqsubseteq_{\mathcal{K}} D$ if in any model $\mathcal{I}$ of $\mathcal{K}$, $C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$ (concept C is subsumed by concept D). Moreover we write $C \sqsubset_{\mathcal{K}} D$ if $C \sqsubseteq_{\mathcal{K}} D$ and $D \not\sqsubseteq_{\mathcal{K}} C$.

The main reasoning task for a DL KB $\mathcal{K}$ is the *consistency check* which tries to prove the satisfiability of $\mathcal{K}$. Another well known reasoning service is *instance checking*, i.e., to check whether an ABox assertion is a logical consequence of $\mathcal{K}$. A more sophisticated version of instance checking, called *instance retrieval*, retrieves all (ABox) individuals that are instances of the given (possibly complex) concept expression C, i.e., all those individuals a such that $\mathcal{K}$ entails that a is an instance of C, denoted $\{a \mid \mathcal{K} \models a{:}C\}$.

Mathematical Fuzzy Logic. *Fuzzy Logic* is the logic of fuzzy sets [27]. A *fuzzy set* A over a countable crisp set X is characterised by a function $A\colon\ X \to [0,1]$. Unlike crisp sets that can be characterised by a function $A\colon\ X \to \{0,1\}$, that is, for any $x \in X$ either $x \in A$ (*i.e.*, $A(x) = 1$) or $x \notin A$ (*i.e.*, $A(x) = 0$), for a fuzzy set A, $A(x)$ dictates that $x \in X$ belongs to the set A to a degree in $[0,1]$. The classical set operations of intersection, union and complementation naturally extend to fuzzy sets as follows. Let A and B be two fuzzy sets. The standard fuzzy set operations are $(A \cap B)(x) = \min(A(x), B(x))$, $(A \cup B)(x) = \max(A(x), B(x))$ and $\bar{A}(x) = 1 - A(x)$, while the *inclusion degree* between A and B is typically defined as

$$(A \subseteq B)(x) = \frac{\sum_{x \in X}(A \cap B)(x)}{\sum_{x \in X} A(x)} \ . \tag{1}$$

The trapezoidal (Fig. 1(a)), the triangular (Fig. 1(b)), the left-shoulder function (Fig. 1(c)), and the right-shoulder function (Fig. 1(d)) are frequently used functions to specify *membership functions* of fuzzy sets. Although fuzzy sets have a greater expressive power than classical crisp sets, their usefulness depends critically on the capability to construct appropriate membership functions for various given concepts in different contexts. The problem of constructing meaningful membership functions is a difficult one (see, *e.g.*, [12, Chap. 10]). However, one easy and typically satisfactory method to define the membership functions is to uniformly partition the range of values into 5 or 7 fuzzy sets using either trapezoidal functions, or triangular functions. The latter is the more used one, as it has less parameters and is also the approach we adopt.

While classical logic is based on crisp set theory, *Mathematical Fuzzy Logic* (MFL) [10] is based on generalised fuzzy set theory. Specifically, in MFL the convention prescribing that a statement is either true or false is changed and is a matter of degree measured on an ordered scale that is no longer $\{0,1\}$, but *e.g.* $[0,1]$. This degree is called *degree of truth* of the logical statement ϕ in the interpretation $\mathcal{I}$. For us, *fuzzy statements* have the form $\langle \phi, \alpha \rangle$, where $\alpha \in (0,1]$ and ϕ is a statement, encoding that the degree of truth of ϕ is *greater or*

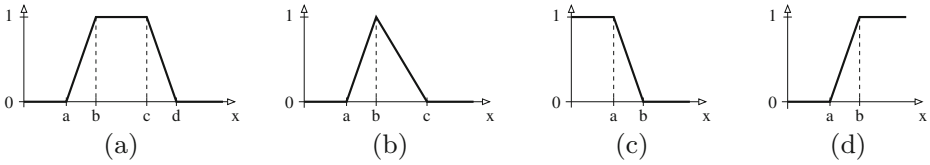


Fig. 1. (a) Trapezoidal function $trz(a, b, c, d)$, (b) triangular function $tri(a, b, c)$, (c) left-shoulder function $ls(a, b)$, and (d) right-shoulder function $rs(a, b)$.

equal α. A *fuzzy interpretation* $\mathcal{I}$ maps each atomic statement p_i into $[0, 1]$ and is then extended inductively to all statements as follows:

$$\begin{array}{l} \mathcal{I}(\phi \wedge \psi) = \mathcal{I}(\phi) \otimes \mathcal{I}(\psi) \\ \mathcal{I}(\phi \vee \psi) = \mathcal{I}(\phi) \oplus \mathcal{I}(\psi) \\ \mathcal{I}(\neg\phi) = \ominus \mathcal{I}(\phi) \\ \mathcal{I}(\phi \rightarrow \psi) = \mathcal{I}(\phi) \Rightarrow \mathcal{I}(\psi) \\ \mathcal{I}(\exists x.\phi(x)) = \sup_{y \in \Delta^{\mathcal{I}}} \mathcal{I}(\phi(y)) \\ \mathcal{I}(\forall x.\phi(x)) = \inf_{y \in \Delta^{\mathcal{I}}} \mathcal{I}(\phi(y)) \end{array} \tag{2}$$

where $\Delta^{\mathcal{I}}$ is the domain of $\mathcal{I}$, and $\otimes$, $\oplus$, $\Rightarrow$, and $\ominus$ are so-called *t-norms*, *t-conorms*, *implication functions*, and *negation functions*, respectively, which extend the Boolean conjunction, disjunction, implication, and negation, respectively, to the fuzzy case. One usually distinguishes three different logics, namely Łukasiewicz, Gödel, and Product logics [10], whose combination functions are reported in Table 2. Note that any other continuous t-norm can be obtained from them (see, *e.g.* [10]).

Satisfiability and *logical consequence* are defined in the standard way, where a fuzzy interpretation $\mathcal{I}$ *satisfies* a fuzzy statement $\langle \phi, \alpha \rangle$ or $\mathcal{I}$ is a *model* of $\langle \phi, \alpha \rangle$, denoted as $\mathcal{I} \models \langle \phi, \alpha \rangle$, iff $\mathcal{I}(\phi) \geq \alpha$.

Description Logics with Fuzzy Concrete Domains. We recap here the syntactic features of the fuzzy DL obtained by extending $\mathcal{ALC}$ with fuzzy concrete domains [25]. A *fuzzy concrete domain* or *fuzzy datatype theory* $\mathbf{D} = \langle \Delta^{\mathbf{D}}, \cdot^{\mathbf{D}} \rangle$ consists of a datatype domain $\Delta^{\mathbf{D}}$ and a mapping $\cdot^{\mathbf{D}}$ that assigns to each data value an element of $\Delta^{\mathbf{D}}$, and to every n-ary datatype predicate $\mathbf{d}$ an n-ary fuzzy

Table 2. Combination functions of various fuzzy logics.

	Łukasiewicz logic	Gödel logic	Product logic	Zadeh logic
$a \otimes b$	$\max(a + b - 1, 0)$	$\min(a, b)$	$a \cdot b$	$\min(a, b)$
$a \oplus b$	$\min(a + b, 1)$	$\max(a, b)$	$a + b - a \cdot b$	$\max(a, b)$
$a \Rightarrow b$	$\min(1 - a + b, 1)$	$\begin{cases} 1 & \text{if } a \leq b \\ b & \text{otherwise} \end{cases}$	$\min(1, b/a)$	$\max(1 - a, b)$
$\ominus a$	$1 - a$	$\begin{cases} 1 & \text{if } a = 0 \\ 0 & \text{otherwise} \end{cases}$	$\begin{cases} 1 & \text{if } a = 0 \\ 0 & \text{otherwise} \end{cases}$	$1 - a$

Table 3. Syntax and semantics of constructs for fuzzy $\mathcal{ALC}(\mathbf{D})$.

bottom (resp. top) concept	$\bot^{\mathcal{I}}(x) = 0$ (resp. $\top^{\mathcal{I}}(x) = 1$)
atomic concept	$A^{\mathcal{I}}(x) \in [0,1]$
abstract role	$R^{\mathcal{I}}(x,y) \in [0,1]$
concrete role	$T^{\mathcal{I}}(x,z) \in [0,1]$
individual	$a^{\mathcal{I}} \in \Delta^{\mathcal{I}}$
concrete value	$v^{\mathcal{I}} \in \Delta^{\mathbf{D}}$
concept intersection	$(C \sqcap D)^{\mathcal{I}}(x) = C^{\mathcal{I}}(x) \otimes D^{\mathcal{I}}(x)$
concept union	$(C \sqcup D)^{\mathcal{I}}(x) = C^{\mathcal{I}}(x) \oplus D^{\mathcal{I}}(x)$
concept negation	$(\neg C)^{\mathcal{I}}(x) = \ominus C^{\mathcal{I}}(x)$
concept implication	$(C \rightarrow D)^{\mathcal{I}}(x) = C^{\mathcal{I}}(x) \Rightarrow D^{\mathcal{I}}(x)$
universal abstract role restriction	$(\forall R.C)^{\mathcal{I}}(x) = \inf_{y \in \Delta^{\mathcal{I}}} \{R^{\mathcal{I}}(x,y) \Rightarrow C^{\mathcal{I}}(y)\}$
existential abstract role restriction	$(\exists R.C)^{\mathcal{I}}(x) = \sup_{y \in \Delta^{\mathcal{I}}} \{R^{\mathcal{I}}(x,y) \otimes C^{\mathcal{I}}(y)\}$
universal concrete role restriction	$(\forall T.\mathbf{d})^{\mathcal{I}}(x) = \inf_{z \in \Delta^{\mathbf{D}}} \{T^{\mathcal{I}}(x,z) \Rightarrow \mathbf{d}^{\mathbf{D}}(z)\}$
existential concrete role restriction	$(\exists T.\mathbf{d})^{\mathcal{I}}(x) = \sup_{z \in \Delta^{\mathbf{D}}} \{T^{\mathcal{I}}(x,z) \otimes \mathbf{d}^{\mathbf{D}}(z)\}$
general concept inclusion	$(C \sqsubseteq D)^{\mathcal{I}} = \inf_{x \in \Delta^{\mathcal{I}}} \{C^{\mathcal{I}}(x) \Rightarrow D^{\mathcal{I}}(x)\}$
concept assertion	$a^{\mathcal{I}} \in C^{\mathcal{I}}$
abstract role assertion	$(a^{\mathcal{I}}, b^{\mathcal{I}}) \in R^{\mathcal{I}}$
concrete role assertion	$(a^{\mathcal{I}}, v^{\mathcal{I}}) \in T^{\mathcal{I}}$

relation over $\Delta^{\mathbf{D}}$. We will restrict to unary datatypes as usual in fuzzy DLs. Therefore, $\cdot^{\mathbf{D}}$ maps indeed each datatype predicate into a function from $\Delta^{\mathbf{D}}$ to $[0,1]$. Typical examples of datatype predicates are

$$\mathbf{d} := ls(a,b) \mid rs(a,b) \mid tri(a,b,c) \mid trz(a,b,c,d) \mid \geq_v \mid \leq_v \mid =_v , \qquad (3)$$

where *e.g.* $\geq_v$ corresponds to the crisp set of data values that are greater or equal than the value v.

In fuzzy DLs, an *interpretation* $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$ consist of a nonempty (crisp) set $\Delta^{\mathcal{I}}$ (the *domain*) and of a *fuzzy interpretation function* $\cdot^{\mathcal{I}}$ that, *e.g.*, maps a concept C into a function $C^{\mathcal{I}} : \Delta^{\mathcal{I}} \rightarrow [0,1]$ and, thus, an individual belongs to the extension of C to some degree in $[0,1]$, *i.e.* $C^{\mathcal{I}}$ is a fuzzy set. The definition of $\cdot^{\mathcal{I}}$ for $\mathcal{ALC}(\mathbf{D})$ with fuzzy concrete domains is reported in Table 3 (where $x, y \in \Delta^{\mathcal{I}}$ and $z \in \Delta^{\mathbf{D}}$). Note that the truth degrees vary according to the chosen fuzzy logic, *i.e.* to its set of combination functions.

Axioms in a fuzzy $\mathcal{ALC}(\mathbf{D})$ KB $\mathcal{K} = \langle \mathcal{T}, \mathcal{A} \rangle$ are graded, *e.g.* a GCI is of the form $\langle C_1 \sqsubseteq C_2, \alpha \rangle$ (*i.e.* C_1 is a sub-concept of C_2 to degree at least α). We may omit the truth degree α of an axiom; in this case $\alpha = 1$ is assumed. An interpretation $\mathcal{I}$ *satisfies* an axiom $\langle \tau, \alpha \rangle$ if $(\tau)^{\mathcal{I}} \geq \alpha$. $\mathcal{I}$ is a *model* of $\mathcal{K}$ iff $\mathcal{I}$ satisfies each axiom in $\mathcal{K}$. We say that $\mathcal{K}$ *entails* an axiom $\langle \tau, \alpha \rangle$, denoted $\mathcal{K} \models \langle \tau, \alpha \rangle$, if any model of $\mathcal{K}$ satisfies $\langle \tau, \alpha \rangle$. The *best entailment degree* of τ w.r.t. $\mathcal{K}$, denoted $bed(\mathcal{K}, \tau)$, is defined as

$$bed(\mathcal{K}, \tau) = \sup\{\alpha \mid \mathcal{K} \models \langle \tau, \alpha \rangle\} . \qquad (4)$$

For a crisp axiom τ, we also write $\mathcal{K} \models_+ \tau$ iff $bed(\mathcal{K}, \tau) > 0$, *i.e.* τ is entailed to some degree $\alpha > 0$.

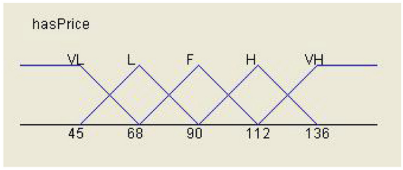


Fig. 2. Fuzzy sets derived from the concrete domain used as range of the role `hasPrice` in Example 1: `VeryLow` (VL), `Low` (L), `Fair` (F), `High` (H), and `VeryHigh` (VH).

Example 1. Let us consider the fuzzy GCI axiom $\langle \exists \texttt{hasPrice}.\texttt{High} \sqsubseteq \texttt{GoodHotel}, 0.569 \rangle$, *where* `hasPrice` *is a concrete role whose range has values measured in euros and the price concrete domain has been automatically fuzzified as follows. The partition into 5 fuzzy sets (*`VeryLow`, `Low`, `Fair`, `High`, *and* `VeryHigh`*) is obtained by considering the interval defined by minimal and maximal hotel prices (resp. 45 and 136), and then by splitting it into four equal subintervals on which three triangular functions, a left-shoulder and a right-shoulder function are built as illustrated in Fig. 2. In particular, the membership function underlying the fuzzy set* `High` *is* $tri(90, 112, 136)$.

Now, let us suppose that the room price for hotel `verdi` *is* 105 *euro,* i.e. *the KB contains the assertion* $\texttt{verdi}{:}\exists \texttt{hasPrice}.{=_{105}}$. *It can be verified under Product logic that* $\mathcal{K} \models \langle \texttt{verdi}{:}\texttt{GoodHotel}, 0.18 \rangle$, i.e. *hotel* `verdi` *is an instance of the class* `GoodHotel` *with a truth degree* 0.18.[2]

3 Learning Fuzzy $\mathcal{EL}(\mathbf{D})$ Axioms

3.1 The Problem Statement

The problem considered in this paper and solved by FOIL-$\mathcal{DL}$ is the automated induction of fuzzy $\mathcal{EL}(\mathbf{D})$[3] GCI axioms from a crisp $\mathcal{DL}$[4] KB and crisp examples, which provide a sufficient condition for a given atomic target concept A_t. It can be cast as a rule learning problem, provided that positive and negative examples of A_t are available. This problem can be formalized as follows. Given: (i) a consistent crisp $\mathcal{DL}$ KB $\mathcal{K} = \langle \mathcal{T}, \mathcal{A} \rangle$ (the *background theory*); (ii) an atomic concept A_t (the *target concept*); (iii) a set $\mathcal{E} = \mathcal{E}^+ \cup \mathcal{E}^-$ of crisp concept assertions e labelled as either positive or negative examples for A_t (the *training set*); and (iv) a set $\mathcal{L}_\mathcal{H}$ of fuzzy $\mathcal{EL}(\mathbf{D})$ GCI axioms (the *language of hypotheses*), the goal is to find a set $\mathcal{H} \subset \mathcal{L}_\mathcal{H}$ (a *hypothesis*) such that $\mathcal{H}$ satisfies the following conditions: $\forall e \in \mathcal{E}^+, \mathcal{K} \cup \mathcal{H} \models_+ e$ (completeness), and $\forall e \in \mathcal{E}^-, \mathcal{K} \cup \mathcal{H} \not\models_+ e$ (consistency).

[2] Note that $0.18 = 0.318 \cdot 0.569$, where $0.318 = tri(90, 112, 136)(105)$.
[3] $\mathcal{EL}(\mathbf{D})$ is a fragment of $\mathcal{ALC}(\mathbf{D})$ [26].
[4] $\mathcal{DL}$ stands for any DL.

Remark 1. In the above problem statement we assume that $\mathcal{K} \cap \mathcal{E} = \emptyset$. Please observe that two further restrictions hold naturally. One is that $\mathcal{K} \not\models_+ \mathcal{E}^+$ since, in such a case, $\mathcal{H}$ would not be necessary to explain $\mathcal{E}^+$. The other is that $\mathcal{K} \cup \mathcal{H} \not\models_+ a{:}\bot$, which means that $\mathcal{K} \cup \mathcal{H}$ is a consistent theory, *i.e.* has a model, that is, adding the learned axioms to the KB keeps the KB consistent.

The background theory. A DL KB allows for the specification of very rich background knowledge in the form of axioms, *e.g.* defining the range of roles or the concept subsumption hierarchy. We do not make any specific assumption about the DL which the language $\mathcal{L}_\mathcal{K}$ of the background theory is based on, except that $\mathcal{K}$ is a crisp KB. However, since $\mathcal{H}$ is a set of fuzzy GCI axioms, $\mathcal{K} \cup \mathcal{H}$ is fuzzy as well.

The language of hypotheses. The language $\mathcal{L}_\mathcal{H}$ is given implicitly by means of syntactic restrictions over a given alphabet, as usual in ILP. In particular, the alphabet underlying $\mathcal{L}_\mathcal{H}$ is a subset of the alphabet for $\mathcal{L}_\mathcal{K}$. However, $\mathcal{L}_\mathcal{H}$ differs from $\mathcal{L}_\mathcal{K}$ as for the form of axioms. More precisely, given the target concept A_t, the hypotheses to be induced are fuzzy GCIs of the form

$$C \sqsubseteq A_t \,, \tag{5}$$

where the left-hand side is defined according to the following syntax

$$C \longrightarrow \top \mid A \mid C_1 \sqcap C_2 \mid \exists R.C \mid \exists T.\mathbf{d} \,. \tag{6}$$

and the concrete domain predicates are the following ones

$$\mathbf{d} := ls(a,b) \mid rs(a,b) \mid tri(a,b,c) \,. \tag{7}$$

Note that the syntactic restrictions of Eq. (6) w.r.t. fuzzy $\mathcal{ALC}(\mathbf{D})$ (see Table 3) allow for a straightforward translation of the inducible axioms into rules of the kind "if x is a C_1 and ... and x is a C_n then x is an A_t", which corresponds to the usual pattern in fuzzy rule induction (in our case, $C \sqsubseteq A_t$ is seen as a rule "if C then A_t"). Also, the restriction of Eq. (7) w.r.t. Eq. (3) is due to the fact that we build fuzzy concrete domain predicates out of numerical data as illustrated in Example 1.

The language $\mathcal{L}_\mathcal{H}$ generated by this syntax is potentially infinite due, *e.g.*, to the nesting of existential restrictions yielding to complex concept expressions such as $\exists R_1.(\exists R_2 \ldots.(\exists R_n.(C))\ldots)$. $\mathcal{L}_\mathcal{H}$ is made finite by imposing further restrictions on the generation process such as the maximal number of conjuncts and the depth of existential nesting allowed in the left-hand side. Also, note that the learnable GCIs do not have an explicit truth degree. However, as we shall see later on, once we have learned a fuzzy GCI of the form (5), we attach to it a confidence degree cf that is obtained by means of the function in Eq. (12).

The training examples. Given the target concept A_t, the training set $\mathcal{E}$ consists of concept assertions of the form $a{:}A_t$, where a is an individual occurring in $\mathcal{K}$. Note that $\mathcal{E}$ is crisp. Also, $\mathcal{E}$ is split into $\mathcal{E}^+$ and

```
function FOIL-DL(K, A_t, E+, E-, L_H): H
begin
1. H := ∅;
2. D = INITIALISEFUZZYCONCRETEDOMAIN(K);
3. while E+ ≠ ∅ do
4.     C := ⊤;
5.     φ := C ⊑ A_t;
6.     E-_φ := E-;
7.     while cf(φ) < θ or E-_φ ≠ ∅ do
8.         C_best := C;
9.         maxgain := 0;
10.        Φ := SPECIALIZE(φ, L_H, K)
11.        foreach φ' ∈ Φ do
12.            gain := GAIN(φ', φ);
13.            if gain ≥ maxgain then
14.                maxgain := gain;
15.                C_best := C';
16.            endif
17.        endforeach
18.        φ := C_best ⊑ A_t;
19.        E-_φ := {e ∈ E- | K ∪ {φ} |=+ e};
20.    endwhile
21.    H := H ∪ {φ};
22.    E+_φ := {e ∈ E+ | K ∪ {φ} |=+ e};
23.    E+ := E+ \ E+_φ;
24. endwhile
end
```

Fig. 3. FOIL-$\mathcal{DL}$: Learning a set of GCI axioms.

$\mathcal{E}^-$. Note that, under OWA, $\mathcal{E}^-$ consists of all those individuals which can be proved to be instance of $\neg A_t$. On the other hand, under CWA, $\mathcal{E}^-$ is the collection of individuals, which cannot be proved to be instance of A_t. We say that an axiom $\phi \in \mathcal{L}_\mathcal{H}$ *covers* an example $e \in \mathcal{E}$ iff $\mathcal{K} \cup \{\phi\} \models_+ e$.

3.2 The Solution Strategy

In FOIL-$\mathcal{DL}$, the *sequential covering* approach of FOIL is kept as shown in Fig. 3. However, due to the peculiarities of the language of hypotheses in FOIL-$\mathcal{DL}$, necessary changes are made to FOIL as concerns both candidate generation and evaluation. A pre-processing phase is also required in order to generate the fuzzy datatypes to be used during the candidate generation phase. These novel features impact the definition of the functions INITIALISEFUZZYCONCRETEDOMAIN, SPECIALIZE and GAIN as detailed in the remainder of this section.

Pre-processing. Given a crisp $\mathcal{DL}$ KB $\mathcal{K}$, the function INITIALISEFUZZYCONCRETEDOMAIN behaves as follows: For each concrete role T occurring in $\mathcal{K}$,

1. determine the minimal and maximal value that T entails according to $\mathcal{K}$, that is $min_T = \min\{v \mid \mathcal{K} \models a{:}\exists T. \leq_v\}$ and $max_T = \max\{v \mid \mathcal{K} \models a{:}\exists T. \geq_v\}$;
2. partition the interval $[min_T, max_T]$ into four uniform subintervals and, for $k = (max_T - min_T)/4$, build the fuzzy concrete domain predicates: $VeryLow_T = ls(min_T, min_T + k)$, $Low_T = tri(min_T, min_T + k, min_T + 2k)$, $Fair_T = tri(min_T + k, min_T + 2k, min_T + 3k)$, $High_T = tri(min_T + 2k, min_T + 3k, max_T)$ and $VeryHigh_T = rs(min_T + 3k, max_T)$.

Eventually, the function returns the set of all built fuzzy datatype predicates

$$\mathbf{D} = \bigcup_{T \text{ concrete role occurring in } \mathcal{K}} \{VeryLow_T, Low_T, Fair_T, High_T, VeryHigh_T\}$$

The method has been illustrated in Example 1.

Candidate generation. In line with the tradition in ILP and in conformance with the search direction in FOIL-$\mathcal{DL}$, the function SPECIALIZE implements a *downward refinement* operator $\rho_{\mathcal{K}}$ which actually exploits the background theory $\mathcal{K}$ in order to avoid the generation of redundant or useless hypotheses:

$$\text{SPECIALIZE}(\phi, \mathcal{L}_{\mathcal{H}}, \mathcal{K}) = \{\phi' \in \mathcal{L}_{\mathcal{H}} \mid \phi' \in \rho_{\mathcal{K}}(\phi)\} \ . \tag{8}$$

The refinement operator $\rho_{\mathcal{K}}$ acts only upon the left-hand-side of a GCI:

$$\rho_{\mathcal{K}}(\phi) = \rho_{\mathcal{K}}(C \sqsubseteq A_t) = \{C' \sqsubseteq A_t \mid C' \in \rho^{\mathcal{C}}_{\mathcal{K}}(C)\} \tag{9}$$

by either adding a new conjunct or replacing an already existing conjunct with a more specific one. More formally, the refinement rules for $\mathcal{EL}(\mathbf{D})$ concepts are defined as follows (here $\mathbf{d}_T$ is one of the fuzzy datatypes for concrete role T build by means of the INITIALISEFUZZYCONCRETEDOMAIN function, while A, B, D and E are atomic concepts, R is an abstract role):

$$\rho^{\mathcal{C}}_{\mathcal{K}}(C) = \begin{cases} \{A\} \cup \{\exists R.\top\} \cup \{\exists R.B \mid \mathit{range}(R, B) \in \mathcal{T}\} \cup \{\exists T.\mathbf{d}_T\} & \text{if } C = \top \\ \{A \sqcap D \mid D \in \rho^{\mathcal{C}}_{\mathcal{K}}(\top)\} \cup \{B \mid B \sqsubset_{\mathcal{K}} A\} & \text{if } C = A \\ \{\exists R.E \mid E \in \rho^{\mathcal{C}}_{\mathcal{K}}(D)\} \cup \{\exists R.(D \sqcap E) \mid E \in \rho^{\mathcal{C}}_{\mathcal{K}}(\top)\} & \text{if } C = \exists R.D \\ \{\exists T.\mathbf{d} \sqcap D \mid D \in \rho^{\mathcal{C}}_{\mathcal{K}}(\top)\} & \text{if } C = \exists T.\mathbf{d}_T \\ \{C_1 \sqcap \cdots \sqcap C_{i-1} \sqcap D \sqcap C_{i+1} \sqcap \cdots \sqcap C_n \mid D \in \rho^{\mathcal{C}}_{\mathcal{K}}(C_i), 1 \leq i \leq n\} & \text{if } C = \prod_{i=1}^{n} C_i \end{cases} \tag{10}$$

Note that the use of relevant knowledge from $\mathcal{K}$ such as range axioms and concept subsumption axioms makes $\rho^{\mathcal{C}}_{\mathcal{K}}$ an "informed" refinement operator. Indeed, its refinement rules combine the syntactic manipulation with the semantic one. Also, this allows the operator to perform "cautious" big steps in the search space. More precisely, the less blind the rules are, the bigger the steps. $\rho^{\mathcal{C}}_{\mathcal{K}}$ also incorporates, in our implementation, a series of simplifications of the concepts built such as

$$\begin{aligned} C \sqcap C &\mapsto C \\ C \sqcap D \text{ and } D \sqsubseteq_{\mathcal{K}} C &\mapsto D \\ C \sqcap D \text{ and } C \sqcap D \sqsubseteq_{\mathcal{K}} \bot &\mapsto \bot \text{ (in this case we drop the refinement)} \end{aligned}$$

to reduce the search space. We are not going to detail them here.

Example 2. Let us consider that A_t is the target concept, A, A', B, R, R', T are concepts and properties occurring in $\mathcal{K}$, and $A' \sqsubseteq_{\mathcal{K}} A$. Under these assumptions, the axiom $\exists R.B \sqsubseteq A_t$ is specialised into the following axioms:

- $A \sqcap \exists R.B \sqsubseteq A_t$, $B \sqcap \exists R.B \sqsubseteq A_t$, $A' \sqcap \exists R.B \sqsubseteq A_t$;
- $\exists R'.\top \sqcap \exists R.B \sqsubseteq A_t$, $\exists T.\mathbf{d}_T \sqcap \exists R.B \sqsubseteq A_t$;
- $\exists R.(B \sqcap A) \sqsubseteq A_t$, $\exists R.(B \sqcap A') \sqsubseteq A_t$;
- $\exists R.(B \sqcap \exists R.\top) \sqsubseteq A_t$, $\exists R.(B \sqcap \exists R'.\top) \sqsubseteq A_t$, $\exists R.(B \sqcap \exists T.\mathbf{d}_T) \sqsubseteq A_t$.

Note that in the above list, $\mathbf{d}_T$ *has to be instantiated for any of the five candidates for concrete role* T *(*i.e.*,* $VeryLow_T, Low_T, Fair_T, High_T, VeryHigh_T$*).*

It can be verified that $\rho^{\mathcal{C}}_{\mathcal{K}}$ is correct, *i.e.* it drives the search towards more specific concepts according to $\sqsubseteq$. Please note that $\rho_{\mathcal{K}}$ reduces the number of examples covered by a GCI. More precisely, the aim of a refinement step is to reduce the number of covered negative examples, while still keeping some covered positive examples. Since learned GCIs cover only positive examples, $\mathcal{K}$ will remain consistent after the addition of a learned GCIs.

Candidate evaluation. The function Gain implements an information-theoretic criterion for selecting the best candidate at each refinement step according to the following formula:

$$\textsc{Gain}(\phi', \phi) = p * (log_2(cf(\phi')) - log_2(cf(\phi))) \ , \tag{11}$$

where p is the number of positive examples covered by the axiom ϕ that are still covered by ϕ'. Thus, the gain is positive iff ϕ' is more informative in the sense of Shannon's information theory, *i.e.* iff the confidence degree (cf) increases. If there are some refinements, which increase the confidence degree, the function Gain tends to favour those that offer the best compromise between the confidence degree and the number of examples covered. Here, cf for an axiom ϕ of the form (5) is computed as a sort of fuzzy set inclusion degree (see Eq. (1)) between the fuzzy set represented by concept C and the (crisp) set represented by concept A_t. More formally:

$$cf(\phi) = cf(C \sqsubseteq D) = \frac{\sum_{a \in \mathsf{Ind}^+_D(\mathcal{A})} bed(\mathcal{K}, a{:}C)}{|\mathsf{Ind}_D(\mathcal{A})|} \tag{12}$$

where $\mathsf{Ind}^+_D(\mathcal{A})$ (resp., $\mathsf{Ind}_D(\mathcal{A})$) is the set of individuals occurring in $\mathcal{A}$ and involved in $\mathcal{E}^+_\phi$ (resp., $\mathcal{E}^+_\phi \cup \mathcal{E}^-_\phi$) such that $bed(\mathcal{K}, a{:}C) > 0$. We remind the reader that $bed(\mathcal{K}, a{:}C)$ denotes the best entailment degree of the concept assertion $a{:}C$ w.r.t. $\mathcal{K}$ as defined in Eq. (4). Note that $\mathcal{K} \models a{:}A_t$ holds for individuals $a \in \mathsf{Ind}^+_D(\mathcal{A})$ and, thus, $bed(\mathcal{K}, a{:}C \sqcap A_t) = bed(\mathcal{K}, a{:}C)$. Also, note that, even if $\mathcal{K}$ is crisp, the possible occurrence of fuzzy concrete domains in expressions of the form $\exists T.\mathbf{d}_T$ in C may imply that both $bed(\mathcal{K}, C \sqsubseteq A_t) \notin \{0, 1\}$ and $bed(\mathcal{K}, a{:}C) \notin \{0, 1\}$.

4 Related Work

Several extensions of Foil to the management of vague knowledge are reported in the literature [7,22,23] but they are not conceived for DL ontologies. In DL

learning, DL-Foil [9] adapts Foil to learn OWL DL equivalence axioms. DL-Learner [14] is a state-of-the-art system which features several algorithms, none of which however is based on Foil. Yet, among them, the closest to Foil-$\mathcal{DL}$ is ELTL since it implements a refinement operator for concept learning in $\mathcal{EL}$ [16]. Conversely, CELOE learns class expressions in the more expressive OWL DL [15]. All these algorithms work only under OWA and deal only with crisp DLs.

Learning in fuzzy DLs has been little investigated. Konstantopoulos and Charalambidis [13] propose an ad-hoc translation of fuzzy Łukasiewicz $\mathcal{ALC}$ DL constructs into LP in order to apply a conventional ILP method for rule learning. However, the method is not sound as it has been recently shown that the mapping from fuzzy DLs to LP is incomplete [18] and entailment in Łukasiewicz $\mathcal{ALC}$ is undecidable [6]. Iglesias and Lehmann [11] propose an extension of DL-Learner with some of the most up-to-date fuzzy ontology tools, e.g. the *fuzzyDL* reasoner [4]. Notably, the resulting system can learn fuzzy OWL DL equivalence axioms from FuzzyOWL 2 ontologies.[5] However, it has been tested only on a toy problem with crisp training examples and does not build automatically fuzzy concrete domains. Lisi and Straccia [17] present *Soft*Foil, a Foil-like method for learning fuzzy $\mathcal{EL}$ inclusion axioms from fuzzy DL-Lite$_{core}$ ontologies (a fuzzy variant of the classical DL, DL-Lite$_{core}$ [1]). We would like to stress the fact that Foil-$\mathcal{DL}$ provides a different solution from *Soft*Foil not only as for the KR framework but also as for the refinement operator and the heuristic. Also, unlike *Soft*Foil, Foil-$\mathcal{DL}$ has been implemented and tested.

5 Towards an Application in Tourism

A variant of Foil-$\mathcal{DL}$ has been implemented in the *fuzzyDL-Learner*[6] system. Notably, fuzzy GCIs in $\mathcal{L}_{\mathcal{H}}$ are interpreted under Gödel semantics (see Table 2). However, since $\mathcal{K}$ and $\mathcal{E}$ are represented in crisp DLs, we have used a classical DL reasoner, together with a specialised code, to compute the confidence degree of fuzzy GCIs. Therefore, the system relies on the services of DL reasoners such as Pellet[7] to solve all the deductive inference problems necessary to Foil-$\mathcal{DL}$ to work, namely instance retrieval, instance check and subclasses retrieval. The system can be configured to work under both CWA and OWA.

In order to demonstrate the potential usefulness of Foil-$\mathcal{DL}$ on a real-world application, we have considered a case study in the tourism application domain. More precisely, we have focused on the task of hotel finding because it can be reformulated as a classification problem solvable with Foil-like algorithms. Our goal is to use Foil-$\mathcal{DL}$ to find axioms of the form (5) for discriminating good hotels from bad ones. To the purpose, we have built an ontology, named *Hotel*,[8] which models the meaningful entities of the domain in hand.

[5] http://straccia.info/software/FuzzyOWL
[6] http://straccia.info/software/FuzzyDL-Learner/
[7] http://clarkparsia.com/pellet/
[8] http://straccia.info/software/FuzzyDL-Learner/download/FOIL-DL/examples/Hotel/Hotel.owl/

The ontology. The ontology *Hotel* consists of 8000 axioms, 74 classes, 4 object properties, 2 data properties, and 1504 individuals.

The main concepts forming the terminology of *Hotel* model the sites of interest (class `Site`), and the distances between sites (class `Distance`). Sites include accommodations (class `Accommodation`) such as hotels, attractions (class `Attraction`) such as parks, stations (class `Station`) such as airports, and civic facilities (class `Civic`) such as hospitals. The terminology encompasses also the amenities (class `Amenity`) offered by hotels (class `Hotel`) and the official 5-star classification system for hotel ranking (class `Rank`). The object properties `hasDistance` and `isDistanceFor` model the relationship between a site and a distance, and between a distance and the two sites, respectively. The data properties `hasPrice` and `hasValue` represent the average price of a room and the numerical value of a distance, respectively. Note that the latter would be better modeled as attribute of a ternary relation. However, since only binary relations can be represented in OWL, one such ternary relation is simulated with the class `Distance` and the properties `hasDistance`, `isDistanceFor` and `hasValue`.

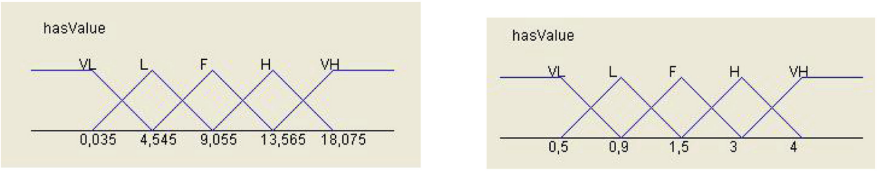
The 1504 individuals occurring in *Hotel* refer to the case of Pisa, Italy. In particular, 59 instances of the class `Hotel` have been automatically extracted from the web site of TripAdvisor.[9] Information about the rank, the amenities and the average room price has been added in the ontology for each of these instances. Further 24 instances have been created for the class `Site` and distributed among the classes under `Attraction`, `Civic` and `Station`. Finally, 1416 distances (instances of `Distance`) between the accommodations and the sites of interest have been measured in km and computed by means of Google Maps[10] API.

Anectodal experiments with Foil-$\mathcal{DL}$. As an illustration of the potential of FOIL-$\mathcal{DL}$, we discuss here the results obtained in two trials concerning the aforementioned learning problem with the class `Good_Hotel` as target concept. Graded hotel ratings from TripAdvisor users have been exploited for distinguishing good hotels from bad ones. Out of the 59 hotels, 12 with a higher percentage of positive feedback have been classified as instances of `Good_Hotel` (*i.e.*, as positive examples). In both trials, FOIL-$\mathcal{DL}$ has been configured to work under OWA by using Pellet as a DL reasoner. Syntactic restrictions are imposed on the form of the learnable GCI axioms. More precisely, conjunctions can have at most 5 conjuncts and at most 2 levels of nesting are allowed in existential role restrictions. The two trials differ as for the alphabet underlying the language of hypotheses.

First trial. In the first experiment, all the classes and the properties occurring in *Hotel* are considered to be part of the alphabet of the language of hypotheses. The membership functions for fuzzy concepts derived from the data properties `hasPrice` and `hasValue` in this trial are shown in Figs. 2 and 4(a), respectively. The results obtained for this configuration of FOIL-$\mathcal{DL}$ are:

```
Confidence          Axiom
   1,000       Hostel subclass of Good_Hotel
   1,000       hasPrice_veryhigh subclass of Good_Hotel
   0,569       hasPrice_high subclass of Good_Hotel
```

[9] http://www.tripadvisor.com/
[10] http://maps.google.com/



(a) (b)

Fig. 4. Membership functions for the fuzzy concepts `hasValue_verylow`, `hasValue_low`, `hasValue_fair`, `hasValue_high`, and `hasValue_veryhigh`, derived by FOIL-$\mathcal{DL}$ from the data property `hasValue` of *Hotel* in (a) the first trial and (b) the second trial.

```
0,286      hasAmenity some (24h_Reception) and hasAmenity some (Disabled_Facilities)
               and hasPrice_low subclass of Good_Hotel
0,282      hasAmenity some (Babysitting) and hasRank some (Rank)
               and hasPrice_fair subclass of Good_Hotel
0,200      Hotel_1_Star subclass of Good_Hotel
```

These results suggest the existence of user profiles, *e.g.* families and disabled people. Note that no axiom has been induced which encompasses knowledge about the distance of the accommodation from the sites of interest.

Second trial. In the second experiment, the configuration of FOIL-$\mathcal{DL}$ remains unchanged except for the alphabet underlying the language of hypotheses and the definition of membership functions for the fuzzy concepts. More precisely, the use of the object property `hasAmenity` is forbidden. Also, the fuzzification of the data property `hasValue` is more reasonable for a foot distance. Here, a very low distance does not exceed 900 meters, an average distance is about 1500 meters, and so on, as illustrated in Fig. 4(b). The axioms learned by FOIL-$\mathcal{DL}$ are:

```
Confidence     Axiom
  1,000     Hostel subclass of Good_Hotel
  1,000     hasPrice_veryhigh subclass of Good_Hotel
  0,739     hasDistance some (isDistanceFor some (Bus_Station) and hasValue_low)
              and hasDistance some (isDistanceFor some (Town_Hall) and hasValue_fair)
              and hasRank some (Rank) and hasPrice_verylow subclass of Good_Hotel
  0,569     hasPrice_high subclass of Good_Hotel
  0,289     Hotel_3_Stars and hasDistance some (isDistanceFor some (Train_Station)
              and hasValue_verylow) and hasPrice_fair subclass of Good_Hotel
  0,198     Hotel_4_Stars and hasDistance some (isDistanceFor some (Square) and hasValue_high)
              and hasRank some (Rank) and hasPrice_fair subclass of Good_Hotel
```

Note that the knowledge concerning the distance of the hotels from the sites of interest has now emerged during the learning process. In particular, the induced axioms suggest that closeness to stations of transportation means is a desirable feature when choosing a hotel.

A comparison with DL-Learner. In order to better illustrate the differences between FOIL-$\mathcal{DL}$ and state-of-the-art DL learning algorithms, we report here the results obtained for the same learning problem with two of the algorithms in

the suite of DL-Learner: ELTL and CELOE.[11] For the purpose of this comparison we have adapted *Hotel* to make it compatible with DL-Learner. Notably, it has been necessary to provide explicitly negative examples for `Good_Hotel`. They have been generated by exploiting once again the graded hotel ratings of TripAdvisor users. More precisely, of the 59 hotels, 11 with a lower percentage of positive feedback have been classified as negative examples for `Good_Hotel`. The two algorithms have been run with default configuration. ELTL has returned 100 class expressions, out of which the first ten are reported below:

```
1: Thing (pred. acc.: 52,17%, F-measure: 68,57%)
2: Site (pred. acc.: 52,17%, F-measure: 68,57%)
3: Hotel (pred. acc.: 52,17%, F-measure: 68,57%)
4: Accomodation (pred. acc.: 52,17%, F-measure: 68,57%)
5: hasDistance some Distance (pred. acc.: 52,17%, F-measure: 68,57%)
6: (Site and hasDistance some Distance) (pred. acc.: 52,17%, F-measure: 68,57%)
7: (Hotel and hasDistance some Distance) (pred. acc.: 52,17%, F-measure: 68,57%)
8: (Accomodation and hasDistance some Distance) (pred. acc.: 52,17%, F-measure: 68,57%)
9: hasDistance some isDistanceFor some University (pred. acc.: 52,17%, F-measure: 68,57%)
10: hasDistance some isDistanceFor some Train_Station (pred. acc.: 52,17%, F-measure: 68,57%)
```

Note that they are rather weak as for predictive accuracy and quite trivial as for the significance except for the last two ones which involve the notion of distance from a site of interest. CELOE has returned the following ten solutions, with a little improvement of the effectiveness with respect to ELTL due to the augmented expressivity of the DL employed for the language of hypotheses:

```
1: ((not Camping) and (not Hotel_2_Stars)) (pred. acc.: 60,87%, F-measure: 72,73%)
2: (not Hotel_2_Stars) (pred. acc.: 56,52%, F-measure: 70,59%)
3: (not Camping) (pred. acc.: 56,52%, F-measure: 70,59%)
4: (Rank or (not Hotel_2_Stars)) (pred. acc.: 56,52%, F-measure: 70,59%)
5: (Rank or (not Camping)) (pred. acc.: 56,52%, F-measure: 70,59%)
6: (Place or (not Hotel_2_Stars)) (pred. acc.: 56,52%, F-measure: 70,59%)
7: (Place or (not Camping)) (pred. acc.: 56,52%, F-measure: 70,59%)
8: (Distance or (not Hotel_2_Stars)) (pred. acc.: 56,52%, F-measure: 70,59%)
9: (Distance or (not Camping)) (pred. acc.: 56,52%, F-measure: 70,59%)
10: (Amenity or (not Hotel_2_Stars)) (pred. acc.: 56,52%, F-measure: 70,59%)
```

Interestingly, the most accurate defines a good hotel by saying what it can not be considered as such. However, none of the learned class expressions provides a definition on the basis of the hotel features or the closeness with sites of interest.

6 Conclusions

We have described a novel method, named Foil-$\mathcal{DL}$, which addresses the problem of learning fuzzy $\mathcal{EL}(\mathbf{D})$ GCI axioms from any crisp $\mathcal{DL}$ KB. The method extends Foil in a twofold direction: from crisp to fuzzy and from rules to GCIs. Notably, vagueness is captured by the definition of confidence degree reported

[11] One such comparison could not be made with DL-Foil since the implemented algorithm was not made available by the authors.

in (12) and incompleteness is dealt with the OWA. Also, thanks to the variable-free syntax of DLs, the learnable GCIs are highly understandable by humans and translate easily into natural language sentences. In particular, FOIL-$\mathcal{DL}$ adopts the user-friendly presentation style of the Manchester OWL syntax.[12]

The experimental results are quite promising and encourage the application of FOIL-$\mathcal{DL}$ to more challenging real-world problems. Notably, in spite of the low expressivity of $\mathcal{EL}$, FOIL-$\mathcal{DL}$ has turned out to be robust mainly due to the refinement operator and to the fuzzification facilities. A distinguishing feature of $\rho_{\mathcal{K}}$ is that it exploits the TBox, *e.g.* for concepts $A_2 \sqsubset A_1$, we reach A_2 via $\top \rightsquigarrow A_1 \rightsquigarrow A_2$. This way, we can stop the search if A_1 is already too weak. The operator also uses the range of roles to reduce the search space. This is similar to mode declarations widely used in ILP. However, in DL KBs, domain and range are usually explicitly given, so there is no need to define them manually. Overall, $\rho_{\mathcal{K}}$ supports more structures, *i.e.* concrete domains, than *e.g.* [16] and tries to smartly incorporate background knowledge. Additionally, unlike CELOE, the fuzzification of concrete domains enables the invention of new concepts during the learning process, which can be considered as a special case of predicate invention.

For the future, we intend to conduct a more extensive empirical evaluation of FOIL-$\mathcal{DL}$, which could suggest directions of improvement of the method towards more effective formulations of, *e.g.*, the information gain function and the refinement operator. Also, it can be interesting to analyse the impact of the different fuzzy logics on the learning process. Eventually, we shall investigate about learning fuzzy GCI axioms from FuzzyOWL 2 ontologies, by coupling the learning algorithm to the *fuzzyDL* reasoner, instead of learning from crisp OWL 2 data by using a classical DL reasoner.

References

1. Artale, A., Calvanese, D., Kontchakov, R., Zakharyaschev, M.: The DL-Lite family and relations. J. Artif. Intell. Res. **36**, 1–69 (2009)
2. Baader, F., Calvanese, D., McGuinness, D., Nardi, D., Patel-Schneider, P. (eds.): The Description Logic Handbook: Theory Implementation and Applications, 2nd edn. Cambridge University Press, Cambridge (2007)
3. Baader, F., Hanschke, P.: A scheme for integrating concrete domains into concept languages. In: Mylopoulos, J., Reiter, R. (eds.) Proceedings of the 12th International Joint Conference on Artificial Intelligence, pp. 452–457. Morgan Kaufmann (1991)
4. Bobillo, F., Straccia, U.: fuzzyDL: An expressive fuzzy description logic reasoner. In: IEEE International Conference on Fuzzy Systems, pp. 923–930. IEEE (2008)
5. Borgida, A.: On the relative expressiveness of description logics and predicate logics. Artif. Intell. **82**(1–2), 353–367 (1996)
6. Cerami, M., Straccia, U.: On the (un)decidability of fuzzy description logics under Łukasiewicz t-norm. Inf. Sci. **227**, 1–21 (2013)

[12] http://www.w3.org/TR/owl2-manchester-syntax/

7. Drobics, M., Bodenhofer, U., Klement, E.P.: FS-FOIL: an inductive learning method for extracting interpretable fuzzy descriptions. Int. J. Approximate Reasoning **32**(2–3), 131–152 (2003)
8. Dubois, D., Prade, H.: Possibility theory, probability theory and multiple-valued logics: a clarification. Ann. Math. Artif. Intell. **32**(1–4), 35–66 (2001)
9. Fanizzi, N., d'Amato, C., Esposito, F.: DL-FOIL concept learning in description logics. In: Železný, F., Lavrač, N. (eds.) ILP 2008. LNCS (LNAI), vol. 5194, pp. 107–121. Springer, Heidelberg (2008)
10. Hájek, P.: Metamathematics of Fuzzy Logic. Kluwer, Dordrecht (1998)
11. Iglesias, J., Lehmann, J.: Towards integrating fuzzy logic capabilities into an ontology-based inductive logic programming framework. In: Proceedings of the 11th International Conference on Intelligent Systems Design and Applications. IEEE Press (2011)
12. Klir, G.J., Yuan, B.: Fuzzy Sets and Fuzzy Logic: Theory and Applications. Prentice-Hall Inc., Englewood Cliffs (1995)
13. Konstantopoulos, S., Charalambidis, A.: Formulating description logic learning as an inductive logic programming task. In: Proceedings of the 19th IEEE International Conference on Fuzzy Systems, pp. 1–7. IEEE Press (2010)
14. Lehmann, J.: DL-Learner: learning concepts in description logics. J. Mach. Learn. Res. **10**, 2639–2642 (2009)
15. Lehmann, J., Auer, S., Bühmann, L., Tramp, S.: Class expression learning for ontology engineering. J. Web Seman. **9**(1), 71–81 (2011)
16. Lehmann, J., Haase, C.: Ideal Downward Refinement in the $\mathcal{EL}$ Description Logic. In: De Raedt, L. (ed.) ILP 2009. LNCS, vol. 5989, pp. 73–87. Springer, Heidelberg (2010)
17. Lisi, F.A., Straccia, U.: A logic-based computational method for the automated induction of fuzzy ontology axioms. Fundamenta Informaticae **124**(4), 503–519 (2013)
18. Motik, B., Rosati, R.: A faithful integration of description logics with logic programming. In: Veloso, M. (ed.) Proceedings of the 20th International Joint Conference on Artificial Intelligence, pp. 477–482 (2007)
19. Quinlan, J.R.: Learning logical definitions from relations. Mach. Learn. **5**, 239–266 (1990)
20. Reiter, R.: Equality and domain closure in first order databases. J. ACM **27**, 235–249 (1980)
21. Schmidt-Schauss, M., Smolka, G.: Attributive concept descriptions with complements. Artif. Intell. **48**(1), 1–26 (1991)
22. Serrurier, M., Prade, H.: Improving expressivity of inductive logic programming by learning different kinds of fuzzy rules. Soft. Comput. **11**(5), 459–466 (2007)
23. Shibata, D., Inuzuka, N., Kato, S., Matsui, T., Itoh, H.: An induction algorithm based on fuzzy logic programming. In: Zhong, N., Zhou, L. (eds.) PAKDD 1999. LNCS (LNAI), vol. 1574, pp. 268–274. Springer, Heidelberg (1999)
24. Straccia, U.: Reasoning within fuzzy description logics. J. Artif. Intell. Res. **14**, 137–166 (2001)
25. Straccia, U.: Description logics with fuzzy concrete domains. In: Proceedings of the 21st Conference in Uncertainty in Artificial Intelligence, pp. 559–567. AUAI Press (2005)
26. Straccia, U.: Foundations of Fuzzy Logic and Semantic Web Languages. CRC Studies in Informatics Series. Chapman & Hall, London (2013)
27. Zadeh, L.A.: Fuzzy sets. Inf. Control **8**(3), 338–353 (1965)

Author Index