Theory completion using Inverse Entailment

S.H. Muggleton and C.H. Bryant

Department of Computer Science,
University of York,
York, YO1 5DD,
United Kingdom.

Abstract. The main real-world applications of Inductive Logic Programming (ILP) to date involve the "Observation Predicate Learning" (OPL) assumption, in which both the examples and hypotheses define the same predicate. However, in both scientific discovery and language learning potential applications exist in which OPL does not hold. OPL is ingrained within the theory and performance testing of Machine Learning. A general ILP technique called "Theory Completion using Inverse Entailment" (TCIE) is introduced which is applicable to non-OPL applications. TCIE is based on inverse entailment and is closely allied to abductive inference. The implementation of TCIE within Progol5.0 is described. The implementation uses contra-positives in a similar way to Stickel's Prolog Technology Theorem Prover. Progol5.0 is tested on two different data-sets. The first dataset involves a grammar which translates numbers to their representation in English. The second dataset involves hypothesising the function of unknown genes within a network of metabolic pathways. On both datasets near complete recovery of performance is achieved after relearning when randomly chosen portions of background knowledge are removed. Progol5.0's running times for experiments in this paper were typically under 6 seconds on a standard laptop PC.

1 Introduction

Suppose that an ILP system is being used to augment an incomplete natural language grammar. The grammar so far has the productions shown as Background in Fig. 1 for the non-terminals S (Sentence) and NP (Noun Phrase). The Example sentence cannot be explained by the Background knowledge, but can be explained by the Hypothesis in Fig. 1. Note that the Example is of predicate S, while the Hypothesis is of predicate NP. This contrasts with the usual Machine Learning setting of "Observation Predicate Learning" (OPL) in which examples and hypotheses define the same predicate. The simplified example in Fig. 1 is typical of the situation in grammar learning.

A non-OPL setting is natural for many problems involved in scientific discovery. For instance, at present in functional genomics (the determination of the function of genes from their gene sequence), the metabolic pathways of cells are being progressively determined. For a particular organism, such as yeast,

	Grammar	**Functional Genomics**
Background	S → NP VP NP → DET NOUN	pheno_effect(Gene,Growth_medium) :- ... codes(Gene,Enzyme) ...
Example	S → the nasty man hit the dog	pheno_effect(gene,growth_medium).
Hypothesis	NP → DET ADJ NOUN	codes(gene,enzyme).

Fig. 1. Background knowledge, example and hypothesis involved in augmenting an incomplete **Grammar** and in a **Functional genomics** setting. "pheno_effect" is a shortening of "phenotypic_effect" (see Table 1).

it is possible to represent existing knowledge of metabolic pathways as a logic program (see Fig. 1). New experimental observation can then be explained by hypotheses such as the one shown in Fig. 1. Note once more the contrast with OPL.

The formation of hypotheses such as those shown in Fig. 1 has been intensively investigated within Abductive Logic Programming [7] (ALP). However, unlike the case in ILP, the hypotheses sought in ALP are typically not universally quantified laws. In this paper we describe an approach to non-OPL called Theory Completion using Inverse Entailment (TCIE). TCIE is based on inverse entailment [8] and is implemented within Progol5.0[1]. TCIE suffers from limitations of inverse entailment described by Yamamoto [20]. Augmentations of inverse entailment which address these limitations have been suggested in [9, 4], though these have not yet been implemented in Progol5.0. However, experiments in this paper show that good performance can be achieved using TCIE.

The paper has the following structure. Section 2 introduces TCIE as the special case of inverse entailment in which the construction of the bottom clause involves the derivation of negative ground instances from the background knowledge. In Section 2.1 it is shown how these negative instances can be derived using contra-positives introduced into the background knowledge in a fashion similar to that employed in Stickel's Prolog Technology Theorem Prover (PTTP). The multi-predicate search implemented in Progol5.0 is given in Section 2.3. Experiments are described which involve applying Progol5.0 to relearning a number grammar (Section 3) and missing enzymes in metabolic pathways (Section 4). A comparison with related work is given in Section 5. Section 6 concludes and describes future research.

[1] The C source code for Progol5.0 and the datasets used for the experiments in this paper can be downloaded from ftp://ftp.cs.york.ac.uk/pub/ML_GROUP/progol5.0/ .

2 TCIE

In the standard setting for ILP the learner is provided with logic programs describing background knowledge B and examples E and is required to find a consistent hypothesis H such that the following holds.

$$B \wedge H \models E \tag{1}$$

Inverse entailment [8] is based on the observation that (1) is equivalent for all B, H and E to the following.

$$B \wedge \overline{E} \models \overline{H} \tag{2}$$

Assuming E is a single example we can add its negation to B, deductively derive a finite conjunction of ground facts $\overline{\bot(B,E)}$, and then construct hypotheses H which subsume $\bot(B,E)$. The following non-OPL example of Inverse Entailment comes from [8].

Example 1. **Non-OPL example.**

$$\begin{aligned}
B &= \left\{ \begin{array}{l} \text{hasbeak}(X) \leftarrow \text{bird}(X) \\ \text{bird}(X) \leftarrow \text{vulture}(X) \end{array} \right\} \\
E &= \text{hasbeak}(\text{tweety}) \leftarrow \\
\overline{E} &= \leftarrow \text{hasbeak}(\text{tweety}) \\
\bot(B,E) &= \{\text{hasbeak}(\text{tweety}), \text{bird}(\text{tweety}), \text{vulture}(\text{tweety})\} \\
H_1 &= \text{bird}(\text{tweety}) \leftarrow \\
H_2 &= \text{bird}(X) \leftarrow \\
H_3 &= \text{vulture}(\text{tweety}) \leftarrow \\
H_4 &= \text{vulture}(X) \leftarrow
\end{aligned}$$

H_1, H_2, H_3, H_4 are potential hypotheses.

Yamamoto [20] has shown that this approach only derives clauses H for which H subsumes E relative to background knowledge B (see Plotkin [11] for the definition of relative subsumption). Relative subsumption is strictly weaker than the form of entailment shown in (1). In relative subsumption H can be used at most once in the derivation of E. This rules out, among other things, recursive applications of H.

Irrespective of its limitations, the non-OPL form of inverse entailment shown in Example 1 is not straightforward to implement using a Prolog interpreter. The problem stems from the need to derive negative ground literals (such as $\overline{\text{bird}(\text{tweety})}$) in the construction of $\bot(B,E)$. The following sections show how a Prolog implementation of non-OPL inverse entailment can be achieved based on an adaptation of ideas in Stickel's Prolog Technology Theorem Prover (PTTP).

2.1 PTTP and Contra-positives

Stickel's PTTP [17] provides a method of applying a standard Prolog interpreter to theorem proving with arbitrary non-definite clauses. This is achieved by a number of transformations including the construction of contra-positives. Thus a clause with n atoms is stored as n different rules, a technique known as *locking*. For example, $a \leftarrow b, c$ is also stored as $\overline{b} \leftarrow \overline{a}, c$ and $\overline{c} \leftarrow b, \overline{a}$. The effect of negated atomic formulae is achieved within the Prolog context by using extra predicate symbols. Thus $\overline{p(X)}$ is implemented as $non\_p(X)$. The following shows locking applied to $B \wedge \overline{E}$ from Example 1.

Example 2. **Non-OPL example revisited.**

$$B \wedge \overline{E} = \left\{ \begin{array}{l} \text{hasbeak}(X) \leftarrow \text{bird}(X) \\ \text{bird}(X) \leftarrow \text{vulture}(X) \\ \text{non\_bird}(X) \leftarrow \text{non\_hasbeak}(X) \\ \text{non\_vulture}(X) \leftarrow \text{non\_bird}(X) \\ \text{non\_hasbeak(tweety)} \leftarrow \end{array} \right\}$$

$$\overline{\bot(B,E)} = \{\text{non\_hasbeak(tweety)}, \text{non\_bird(tweety)}, \text{non\_vulture(tweety)}\}$$

$$\bot(B,E) = \{\text{hasbeak(tweety)}, \text{bird(tweety)}, \text{vulture(tweety)}\}$$

Clearly each ground atom in $\overline{\bot(B,E)}$ will succeed as a goal to a Prolog interpreter given the transformed $B \wedge \overline{E}$ in this example. The clause $\bot(B,E)$ is then formed by simply removing the prefix "non_".

2.2 Mode Declarations

Within Progol [8] the user indicates the language within which hypotheses are to be constructed using mode declarations. Mode declarations come in two forms: *modeh* statements indicate the predicates to be used in the head of hypothesised clauses and *modeb* statements indicate predicates to be allowed in the body. A mode declaration such as

```
:- modeb(1,p(+a,-b,#c))?
```

states that predicate 'p' has three arguments and will succeed with '1' answer substitution when called with the first argument ('+' represents input variable) bound with a term of type 'a'. It will return terms of type 'b' ('-' represents output variable) and 'c' ('#' represents constant) as its second and third arguments. The returned last argument will be used as a ground constant in the hypothesis.

Owing to the OPL assumption in previous versions of Progol, mode declarations not only define the hypothesis language but also define the separation between examples and background knowledge. Thus modeh statements indicate which predicates represent examples while modeb statements indicate which predicates represent background knowledge. In the non-OPL context of Progol5.0 modeh and modeb statements are still used to define the hypothesis language.

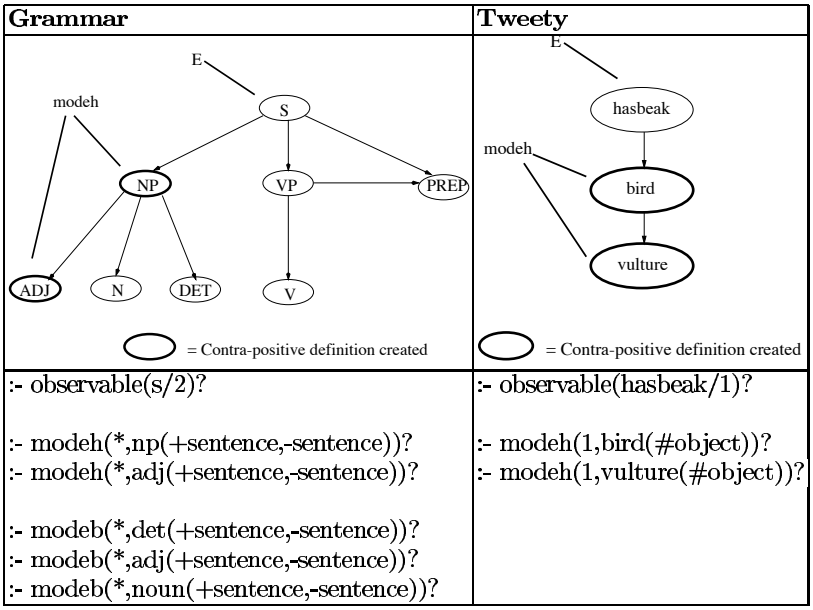

Fig. 2. Calling diagram and Progol5.0 declarations for **Grammar** from Fig. 1 and **Tweety** in Examples 1 and 2.

Additional statements indicate which predicates are "observable". Observable predicates are those which represent examples. Fig. 2 shows the calling diagram and Progol5.0 declarations for the Grammar example of Fig. 1 and the Tweety example of Examples 1 and 2. Note that in the case of the Grammar, it is not necessary to construct contra-positive definitions for each predicate in the calling diagram. It is only necessary to do so for those on the paths in the calling diagram between "modeh" (hypothesis) predicates and "E" (observable) predicates.

2.3 Multi-predicate Search Algorithm

As indicated in the Grammar example shown in Fig. 2 Progol5.0 can be given multiple modeh declarations. Progol5.0 uses a standard covering algorithm where each example is generalised using a multi-predicate search. This search is carried out, over all the predicates associated with modeh declarations, to find the hypothesis which covers the given example with maximal information compression. Compression is calculated using the Minimal Description Length function

$$f = p - (c + n)$$

where p is the number of positive examples covered among the observable predicates, c is the number of atoms in the body of the hypothesised clause and n is the number of negative examples covered by the hypothesised clause. The

```
MultiPredicateSearch(ModehPredicates,Example)
    BestHypothesis=NULL              % Best hypothesised clause so far
    Max=0                            % Maximum Compression achieved
    For each P in ModehPredicates    % Maximise over all modeh predicates
        SinglePredicateSearch(P,Example,BestHypothesis,Max)
    End for each
    If Max>0 then return BestHypothesis

SinglePredicateSearch(P,Example,BestHypothesis,Max)
    Ms is Modehs(P)                  % Find all modeh declarations for P
    MakeContrapositives(P)           % Contra-positives of the form 'non_P'
    As are NegAtomsOf Ms             % Goals of the form "non_P"
    Derive ground atomic Start-set from calling As
    Obs are the observable predicates
    For each s in Start-set
        Let s' be s with 'non_' prefix removed.
        Let e be (s' ← Body(Example))
                                     % Example may be non-unit clause
        Construct ⊥_i from e         % See Section B.1
        Retract Example
        Find C, EmptyClause ⪯ C ⪯ ⊥_i with
            maximum compression Comp wrt Obs
                                     % See Section B.2
        If Comp > Max then
            Max = Comp
            BestHypothesis = C
        Assert Example
    End for each
```

Fig. 3. Multi-predicate search algorithm

search related to each example to be generalised is carried out by the set of related procedures given in Fig. 3.

The underlying refinement graph search is based on the Progol procedure described in [8] (see Appendix A) modified to calculate compression over the observable predicates.

3 Number Grammar Experiment

3.1 Materials

The experiment in this section involves learning a well-defined fragment of natural language, that is the translation of number phrases into their numerical form. For instance, the grammar translates the English phrase "three hundred and twenty-five" to the expression 3*100+2*10+5, or simply 325. Note that this is a special case of the general problem of translation of syntax to semantics. A

```
wordnum(A,[],T) :- tenu(A,[],T).
wordnum(A,[],T) :- word100(A,[],T).
wordnum(A,[],T) :- word1000(A,[],T).

word1000(A,[],T) :- thou(A,[],T).
word1000(A,B,T+N) :- thou(A,[and|C],T), tenu(C,B,N).
word1000(A,B,T+H) :- thou(A,C,T), word100(C,B,H).

thou([D,thousand|R],R,T*1000) :- digit(D,T).

word100(A,[],H) :- hun(A,[],H).
word100(A,B,H+T) :- hun(A,[and|C],H), tenu(C,B,T).

hun([D,hundred|R],R,H*100) :- digit(D,H).

tenu([D],[],N) :- digit(D,N).
tenu([ten],[],10).  tenu([eleven],[],11).  tenu([twelve],[],12).
tenu([thirteen],[],13).  tenu([fourteen],[],14).  tenu([fifteen],[],15).
tenu([sixteen],[],16).  tenu([seventeen],[],17).  tenu([eighteen],[],18).
tenu([nineteen],[],19).

tenu([T],[],N) :- tenmult(T,N).
tenu([T,D],[],N+M) :- tenmult(T,N), digit(D,M).

digit(one,1).  digit(two,2).  digit(three,3).  digit(four,4).
digit(five,5).  digit(six,6).  digit(seven,7).  digit(eight,8).
digit(nine,9).

tenmult(twenty,20).  tenmult(thirty,30).  tenmult(forty,40).
tenmult(fifty,50).  tenmult(sixty,60).  tenmult(seventy,70).
tenmult(eighty,80).  tenmult(ninety,90).
```

Fig. 4. Grammar for translating English number phrases to numbers.

complete Definite Clause Grammar (DCG) for the numbers 1-9999 is shown in Fig. 4.

3.2 Method

Note that the grammar in Fig. 4 has a hierarchically description similar to that of Fig. 1. The aim of the experiment is to determine whether Progol5.0 can recover performance of the complete theory in Fig. 4 after a randomly chosen subset of clauses from throughout the theory is removed.

The complete theory shown in Fig. 4 has 40 clauses. Progol5.0 was applied to learning from 100 randomly chosen examples with background knowledge consisting of partial grammars in which randomly chosen subsets of size 5, 10, 15 and 20 clauses were left out. For each size 10 randomly chosen left-out subsets

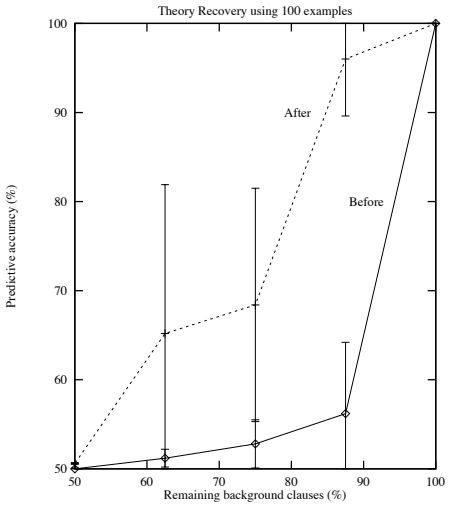

Fig. 5. Results of experiments on number grammar

were chosen and the results were averaged. Performance was measured on the complete set of 9999 examples.

3.3 Results

Fig. 5 shows the results, indicating the predictive accuracy measured both before and after relearning using Progol5.0. Error bars indicate the standard deviation in the predictive accuracy. Each experiment took around 0.1 seconds to run on a Dell 7000 Intel 686 laptop.

3.4 Discussion

The results in Fig. 5 indicate that substantial recovery of performance of a relatively complex grammar can be achieved when up to half of the grammar productions are deleted. However, when half or more of the clauses are deleted recovery fails. The reason for this appears to be related to the incompleteness of the approach discussed in [20]. That is, when multiple clauses are needed to complete a proof Progol5.0 fails to be able to reconstruct them all.

4 Functional Genomics experiment

4.1 Context of Experiment

Genomic data is now being obtained on an industrial scale. The complete genomes of around a dozen micro-organisms have been sequenced. The genomes of about another 50 organisms are in the process of being sequenced and the completion

of the sequencing of the human genome is imminent. The analysis of this data needs to become as industrialised as the methods for obtaining it.

The focus of genome research is moving to the problem of identifying the biological functions of genes. This is known as *functional genomics*. The problem is important because nothing is known about the function of between 30-60% of all new genes identified from sequencing [3, 5, 10]. Functional genomics is recognised as central to a deeper understanding of biology, and the future exploitation of biology in medicine, agriculture, and biotechnology in general. Functional genomics is an appropriate application to test TCIE because: a) relational representations are appropriate for the problem; b) it is only possible to make experimental observations about concepts which are related to the target concept, as opposed to the target concept itself.

Metabolism and Growth Experiments One approach to functional genomics involves growth experiments. These involve feeding a micro-organism different mixtures of nutrients known as *growth media* and measuring the resulting growth. Cells of the micro-organism import molecules from a growth medium and convert them to molecules which are essential for growth via pathways of chemical reactions known as *metabolic pathways*. Each chemical reaction is catalysed by an enzyme. Some of these enzymes are known and others are not. The genes which code for the latter are genes whose function is not known.

To find out what the function of a gene is, mutants of a particular micro-organism are grown which do not contain the gene in question. This process is referred to as *gene deletion*. The effect of not having the gene can then be compared with a control i.e. the wild form of the organism which does not have the mutation. To make the effect of the mutation observable it is necessary to feed samples of the mutant with different growth media. The observable characteristics of an organism are collectively referred to as its *phenotype*. A phenotypic effect is a difference between the phenotype of the wild strain of an organism and the phenotype of one of its mutants.

4.2 Experimental Materials

The Functional Genomics Model In this experiment we simulate the effect of single-gene-deletion growth experiments using the abstract, highly simplified model of a cell shown in Fig. 6. In Computer Science terms, Fig. 6 may be viewed as a graph in which nodes represent molecules, arcs represent chemical reactions, labels of arcs represent enzymes which catatyse particular reactions and paths correspond to metabolic pathways. Although the model is highly simplified it is worthy of study because it has some of the characteristics of the functional genomics domain.

Table 1 lists part of the logic program which represents the functional genomics model. A growth medium is a combination of growth nutrients. Thus there are just seven growth media because there are just three nutrients. The program also contains:

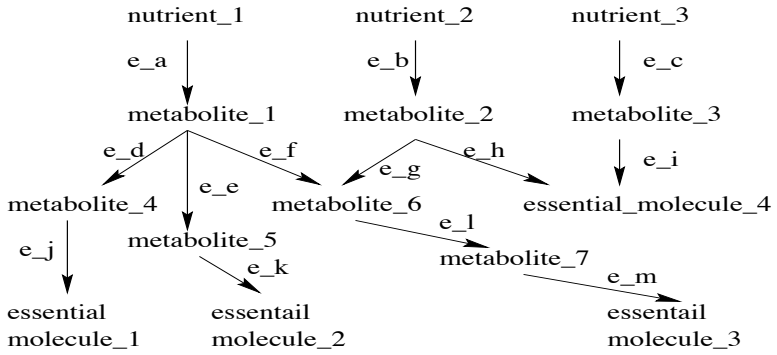

Fig. 6. Abstract, highly simplified model of a cell.

```
phenotypic_effect(Gene, Growth_medium):-
     nutrient_in(Nutrient, Growth_medium),
     metabolic_path(Nutrient, Mi),
     enzyme(E, Mi, Mj),
     codes(Gene, E),
     metabolic_path(Mj, Mn),
     essential_molecule(Mn),
     not(path_without_E(Growth_medium, Mn, E)).

nutrient_in(Nutrient, Growth_medium):- element(Nutrient, Growth_medium).

metabolic_path(A, A).
metabolic_path(A, B):- enzyme(_, A, B).
metabolic_path(A, B):- enzyme(_, A, X), metabolic_path(X, B).

path_without_E(Growth_medium, Mn, E):-
     nutrient_in(Nutrient, Growth_medium),
     path_without_E(Nutrient, Mn, E).
path_without_E(A,A,_).
path_without_E(A,B,E):- enzyme(E2,A,B),not(E=E2).
path_without_E(A,B,E):- enzyme(E2,A,X),not(E=E2),path_without_E(X,B,E).

essential_molecule(ess_mol_1).       essential_molecule(ess_mol_2).
essential_molecule(ess_mol_3).       essential_molecule(ess_mol_4).
```

Table 1. Part of the logic program which represents the functional genomics model.

Table 2. Experimental Method

---

for j **in** (23, 45, 68, 91)
  a training set was created by selecting j examples at random;
  for k **in** (0, 1, 4, 7, 10, 12, 13)
    for i **in** 1 **to** 10 **do**
      – k `codes/2` facts were selected at random;
      – the other $13 - k$ `codes/2` facts were removed from the model;
      – the performance of the resulting incomplete model was measured;
      – Logical Back Propagation was applied to the training set and the incomplete model;
      – the performance of the updated model was measured.
    end
  end
end

– one `enzyme(enzyme, molecule_in, molecule_out)` fact for each enzyme in the model i.e. for each arc in the graph shown in Fig.6;
– `codes(gene, enzyme)` facts representing a one-to-one mapping of genes to enzymes.

The Example Set Examples were represented by positive or negative instances of the predicate `phenotypic_effect(gene, growth_medium)`. For example `phenotypic_effect(g1, [nutrient_1, nutrient_2])` represents the fact that a phenotypic effect is observed if a mutant strain of an organism is created by removing the gene `g1` and this mutant is fed the growth medium which contains nutrients 1 and 2. There are 91 such examples since there are seven examples for each of the 13 genes in the model: there are only seven possible growth media. The class of each example (positive or negative) was deduced from the complete model. There are 45 positives and 46 negatives.

4.3 Experimental Method

Table 2 summarises the experimental method. The performance measure used was predictive accuracy on the complete distribution of the observable predicate i.e. `phenotypic_effect(gene, growth_medium)`. The effect of the inner-most loop is to measure pre-learning and post-learning performance ten times for every possible pair of j and k values shown in Table 2; the purpose of this repetition was to allow the subsequent calculation of the mean accuracy and standard deviation for each pair of j and k values.

It was not possible to use a purely inductive approach to regenerate the `codes/2` facts from the training data because `codes/2` is below `phenotypic_effect/2` in the calling diagram. CProgol was instructed to abduce `codes/2` facts as follows.

```
:- modeh(1,codes(#gene,#enzyme))?
:- observable(phenotypic_effect/2)?
```

During training, Progol was instructed to assume that `codes(gene, enzyme)` is a one-to-one mapping by placing the following constraints in the learning file.

```
:- codes(Gene, Enzyme1), codes(Gene, Enzyme2), not (Enzyme1 = Enzyme2).
:- codes(Gene1, Enzyme), codes(Gene2, Enzyme), not (Gene1 = Gene2).
```

4.4 Results and Analysis

Fig. 7 shows a plot of the results in the form of four learning curves. The results show that applying TCIE to a version of the model which has been made incomplete by removing some of `codes/2` facts leads to a recovery in performance, regardless of how many facts were removed or the size of the training set. The more `codes/2` facts that are removed, the greater the recovery. The larger the training set, the greater the recovery.

Running times for the experiments in Fig. 7 were typically under 6 seconds on a Silicon Graphics O2 workstation.

5 Comparison with related work

TCIE fits within the general scheme of theory refinement described in [19]. Within this scheme TCIE is a form of theory revision based on generalising (completing) revisions. Earlier systems with comparable aims include MIS [16] CLINT [12, 13], RUTH [1], AUDREY [18] and BORDA [6] (see also [15] and [14]). We would argue that though similar in spirit to many of these earlier systems, TCIE as implemented in Progol5.0, has advantages related to the inheritance of efficient pruning, logical constraints and the general Progol framework from earlier versions of Progol. Progol5.0's efficiency is indicated by the fact that running times for experiments in this paper were typically under 6 second on a standard workstation.

Many of the basic mechanisms used throughout theory revision systems are based on abductive operators from logic programming. In [2] Dimopoulos and Kakas compared abductive and inductive forms of reasoning. They note that ILP and abduction share the following relationship among background theory T, hypothesis H and observation O.

$$T, H \models O$$

They summarise the main difference between abduction and induction as follows.

> Whilst in abduction the addition of H to T just adds missing facts related to the particular observations in induction H introduces new general relations in T.

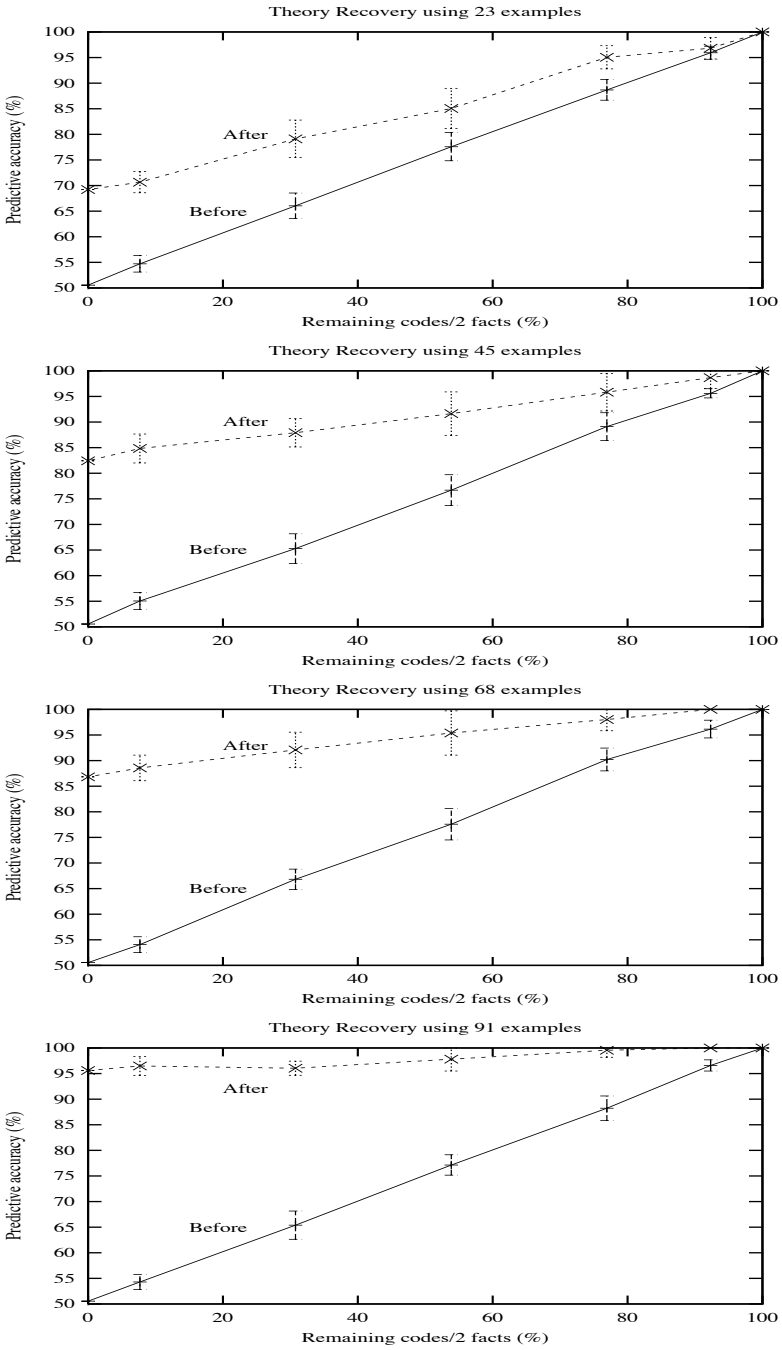

Fig. 7. Learning curves for the model of functional genomics.

TCIE incorporates a form of abduction based on the use of contra-positives. This is used to derive the set of ground atoms referred to as the Start-set in the algorithm in Fig. 3. However, TCIE then goes on to generalise these facts to find new general relations. We would therefore argue that TCIE is a form of induction which incorporates abductive reasoning.

6 Conclusions and further work

In this paper we describe the non-OPL setting for ILP. Here the predicates involved in observations are distinct from those in the head of hypothesised clauses. It is shown how Inverse Entailment has been used to implement TCIE within Progol5.0. Progol5.0 is then applied in two experiments to datasets in which random subsets of the complete theory are deleted. The results indicate that Progol5.0 is capable of recovering predictive accuracy to a substantial degree, even when large sections of the background knowledge have been deleted.

However, it was noted in the discussion in Section 3.4 that the incompleteness described by Yamamoto [20] limits the performance of Progol5.0. This incompleteness stems from multiple uses of the hypothesised clause within the derivation of an example. The performance limitation might come as some surprise in that this incompleteness limitation has often been assumed to be related to learning of recursive theories which necessarily require recursive hypotheses to be used more than once. The grammar shown in Fig. 3 is non-recursive. However, consider parsing the phrase "two hundred and two". This parse uses the ground unit clause digit(two,2) twice. Therefore if this ground unit clause is missing from the background theory then the incompleteness noted by Yamamoto will prevent its being relearned. Thus methods to circumvent this form of incompleteness are important for future research.

The datasets used in this paper differ substantially from those used in the more familiar OPL setting, in which there is only one predicate to be learned. We believe that the novel testing methodology used in the experimental sections of this paper are important contribution. Moreover, we believe that the non-OPL setting should be of increasing interest in applications of ILP.

Acknowledgements

The authors would like to thank Akihiro Yamamoto, Koichi Furukawa, Antonis Kakas, Stefan Wrobal and Luc De Raedt for discussions about Abduction and inverse entailment. The first author would like to thank his wife wife Thirza and daughter Clare for their cheerful support during the writing of this paper. This work was supported partly by the Esprit RTD project "ALADIN' (project 28623), EPSRC grant "Closed Loop Machine Learning", BBSRC/EPSRC grant "Protein structure prediction - development and benchmarking of machine learning algorithms" and EPSRC ROPA grant "Machine Learning of Natural Language in a Computational Logic Framework".

References

1. H. Ade, L. De Raedt, and M. Bruynooghe. Theory revision. In S. Muggleton, editor, *Proceedings of the 3rd International Workshop on Inductive Logic Programming*, pages 179–192, 1993.
2. Y. Dimopoulos and A. Kakas. Abduction and inductive learning. In L. De Raedt, editor, *Proceedings of the Fifth Inductive Logic Programming Workshop (ILP95)*, pages 25–28, Leuven, Belgium, 1995. KU Leuven.
3. B. Dujon. The yeast genome project - what did we learn? *Trends in Genetics*, 12:263–270, 1996.
4. K. Furukawa. On the completion of the most specific hypothesis computation in inverse entailment for mutual recursion. In *Proceedings of Discovery Science '98*, LNAI 1532, pages 315–325, Berlin, 1998. Springer-Verlag.
5. Goffeau, A. *et multi al.* Life with 6000 genes. *Science*, 274:546–567, 1996.
6. K. Ito and A. Yamamoto. Finding hypotheses from examples by computing the least generlisation of bottom clauses. In S. Arikawa and H. Motoda, editors, *Proceedings of Discovery Science '98*, pages 303–314. Springer, Berlin, 1998. LNAI 1532.
7. A.C. Kakas, R.A. Kowalski, and F. Toni. Abductive logic programming. *Journal of Logic and Computation*, 2, 1992.
8. S. Muggleton. Inverse entailment and Progol. *New Generation Computing*, 13:245–286, 1995.
9. S. Muggleton. Completing inverse entailment. In C.D. Page, editor, *Proceedings of the Eighth International Workshop on Inductive Logic Programming (ILP-98)*, LNAI 1446, pages 245–249. Springer-Verlag, Berlin, 1998.
10. S.G. Oliver. From DNA sequence to biological function. *Nature*, 379:597–600, 1996.
11. G. Plotkin. A further note on inductive generalization. In *Machine Intelligence*, volume 6. Edinburgh University Press, 1971.
12. L. De Raedt. *Interactive Theory Revision: an Inductive Logic Programming Approach.* Academic Press, 1992.
13. L. De Raedt and M. Bruynooghe. Interactive concept-learning and constructive induction by analogy. *Machine Learning*, 8:107–150, 1992.
14. L. De Raedt and N. Lavrac. Multiple predicate learning in two inductive logic programming settings. *Journal on Pure and Applied Logic*, 4(2):227–254, 1996.
15. B. L. Richards and R. J. Mooney. Automated refinement of first-order Horn-clause domain theories. *Machine Learning*, 19(2):95–131, 1995.
16. E.Y. Shapiro. *Algorithmic program debugging.* MIT Press, 1983.
17. M. Stickel. A Prolog technology theorem prover: implementation by an extended Prolog compiler. *Journal of Automated Reasoning*, 4(4):353–380, 1988.
18. J. Wogulis. Revising relational theories. In *Proceedings of the 8th International Workshop on Machine Learning*, pages 462–466. Morgan Kaufmann, 1991.
19. S. Wrobel. First-order theory refinement. In L. De Raedt, editor, *Advances in Inductive Logic Programming*, pages 14–33. IOS Press, Ohmsha, Amsterdam, 1995.
20. A. Yamamoto. Which hypotheses can be found with inverse entailment? In N. Lavrač and S. Džeroski, editors, *Proceedings of the Seventh International Workshop on Inductive Logic Programming*, pages 296–308. Springer-Verlag, Berlin, 1997. LNAI 1297.

A Progol algorithm

B Definition of most-specific clause

Definition 1. Most-specific clause $\bot_i$. *Let h, i be natural numbers B be a set of Horn clauses, $e = a \leftarrow b_1, .., b_n$ be a definite clause, M be a set of mode declarations containing exactly one modeh m such that $a(m) \preceq a$ and $\bot$ be the most-specific (potentially infinite) definite clause such that $B \wedge \bot \wedge \overline{e} \vdash_h EmptyClause$. $\bot_i$ is the most-specific clause in $\mathcal{L}_i(M)$ such that $\bot_i \preceq \bot$.*

B.1 Construction of most-specific clause

Algorithm 1 Algorithm for constructing $\bot_i$.

1. *Given natural numbers h, i, Horn clauses B, definite clause e and set of mode declarations M.*
2. *Let $k = 0$, $hash : Terms \rightarrow N$ be a hash function which uniquely maps terms to natural numbers, $\overline{e}$ be the clause normal form logic program $\overline{a} \wedge b_1 \wedge .. \wedge b_n$, $\bot_i = \langle\rangle$ and InTerms$= \emptyset$.*
3. *If there is no modeh in M such that $a(m) \preceq a$ then return EmptyClause. Otherwise let m be the first modeh declaration in M such that $a(m) \preceq a$ with substitution θ_h. Let a_h be a copy of $a(m)$ and for each v/t in θ_h if v corresponds to a #type in m then replace v in a_h by t otherwise replace v in a_h by v_k where $k = hash(t)$ and add v to InTerms if v corresponds to +type. Add a_h to $\bot_i$.*
4. *If $k = i$ return $\bot_i$ else $k = k + 1$.*
5. *For each modeb m in M let $\{v_1, .., v_n\}$ be the variables of +type in $a(m)$ and $T(m) = T_1 \times .. \times T_n$ be a set of n-tuples of terms such that each T_i corresponds to the set of all terms of the type associated with v_i in m (term t is tested to be of a particular type by calling Prolog with type(t) as goal). For each $\langle t_1, .., t_n \rangle$ in $T(m)$ let a_b be a copy of $a(m)$ and $\theta = \{v_1/t_1, .., v_n/t_n\}$. If Prolog with depth-bound h succeeds on goal $a_b\theta$ with the set of answer substitutions Θ_b then for each θ_b in Θ_b and for each v/t in θ_b if v corresponds to a #type in m then replace v in a_b by t otherwise replace v in a_b by v_k where $k = hash(t)$ and add v to InTerms if v corresponds to -type. Add $\overline{a_b}$ to $\bot_i$.*
6. *Goto step 4.*

B.2 *A**-like algorithm for finding clause with maximal compression

Firstly we define some auxiliary functions used in Algorithm 2.

Definition 2. Auxiliary functions. *Let the examples E be a set of Horn clauses. Let $h, i, B, e, M, \bot_i$ be as in Definition 1. Let C be a clause, k be a natural number and θ be a substitution.*

$$d'(v) = \begin{cases} 0 & \text{if there is no -type variable in the head of } \bot_i \\ 0 & \text{if } v \text{ is -type in the head of } \bot_i \\ \infty & \text{if } v \text{ is not in } \bot_i \\ (min_{u \in U_v} d'(u)) + 1 & \text{otherwise} \end{cases}$$

where U_v are the -type variables in atoms in the body of C which contain +type occurrences of v. Below state s has the form $\langle C, \theta, k \rangle$. c is a user-defined parameter for the maximal clause body length. $|S|$ denotes the cardinality of any set S.

$$
\begin{aligned}
p_s &= |\{e : e \in E \text{ and } B \wedge C \wedge \overline{e} \vdash_h \mathit{EmptyClause}\}| \\
n_s &= |\{e : e \in E \text{ and } B \wedge C \wedge e \vdash_h \mathit{EmptyClause}\}| \\
c_s &= |C| - 1 \\
V_s &= \{v : u/v \in \theta \text{ and } u \text{ in body of } C\} \\
h_s &= min_{v \in V_s} d'(v) \\
g_s &= p_s - (c_s + h_s) \\
f_s &= g_s - n_s
\end{aligned}
$$

best(S) is a state $s \in S$ which has $c_s \leq c$ and for which there does not exist $s' \in S$ for which $f_{s'} > f_s$.

$$
\mathit{prune}(s) = \begin{cases} \mathit{true} & \text{if } n_s = 0 \text{ and } f_s > 0 \\ \mathit{true} & \text{if } g_s \leq 0 \\ \mathit{true} & \text{if } c_s \geq c \\ \mathit{false} & \text{otherwise} \end{cases}
$$

$$
\mathit{terminated}(S, S') = \begin{cases} \mathit{true} & \text{if } s = \mathit{best}(S), n_s = 0, f_s > 0 \text{ and} \\ & \quad \text{for each } s' \text{ in } S' \text{ it is the case that } f_s \geq g_{s'} \\ \mathit{false} & \text{otherwise} \end{cases}
$$

Algorithm 2 Algorithm for searching $\mathit{EmptyClause} \preceq C \preceq \bot_i$**.**

1. *Given $h, B, e, \bot_i$ as in Definition 1.*
2. *Let $\mathit{Open} = \{\langle \mathit{EmptyClause}, \emptyset, 1 \rangle\}$ and $\mathit{Closed} = \emptyset$.*
3. *Let $s = \mathit{best}(\mathit{Open})$ and $\mathit{Open} = \mathit{Open} - \{s\}$.*
4. *Let $\mathit{Closed} = \mathit{Closed} \cup \{s\}$.*
5. *If $\mathit{prune}(s)$ goto 7.*
6. *Let $\mathit{Open} = (\mathit{Open} \cup \rho(s)) - \mathit{Closed}$.*
7. *If $\mathit{terminated}(\mathit{Closed}, \mathit{Open})$ then return $\mathit{best}(\mathit{Closed})$.*
8. *If $\mathit{Open} = \emptyset$ then print 'no compression' and return $\langle e, \emptyset, 1 \rangle$.*
9. *Goto 3.*