Heuristic Based Induction of Answer Set Programs: From Default Theories to Combinatorial problems

Farhad Shakerin and Gopal Gupta

The University of Texas at Dallas, Richardson TX 75080, USA
{fxs130430,gupta}@utdallas.edu

Abstract. Significant research has been conducted in recent years to extend Inductive Logic Programming (ILP) methods to induce Answer Set Programs (ASP). These methods perform an exhaustive search for the correct hypothesis by encoding an ILP problem instance as an ASP program. Exhaustive search, however, results in loss of scalability. In addition, the language bias employed in these methods is overly restrictive too. In this paper we extend our previous work on learning stratified answer set programs that have a single stable model to learning arbitrary (i.e., non-stratified) ones with multiple stable models. Our extended algorithm is a greedy FOIL-like algorithm, capable of inducing non-monotonic logic programs, examples of which includes programs for combinatorial problems such as graph coloring and N-queens. To the best of our knowledge, this is the first heuristic-based ILP algorithm to induce answer set programs with multiple stable models.

Keywords: Inductive Logic Programming , Machine Learning , Negation as Failure , Answer Set Programming

CEUR-WS.org/Vol-2206/paper3.pdf

1 Introduction

Statistical machine learning methods produce models that are not comprehensible for humans because they are algebraic solutions to optimization problems such as risk minimization or data likelihood maximization. These methods do not produce any intuitive description of the learned model. Lack of intuitive descriptions makes it hard for users to understand and verify the underlying rules that govern the model. Also, these methods cannot produce a justification for a prediction they compute for a new data sample. Additionally, if prior knowledge (background knowledge) is extended in these methods, then the entire model needs to be re-learned. Finally, no distinction is made between exceptions and noisy data in these methods.

Inductive Logic Programming [11], however, is one technique where the learned model is in the form of logic programming rules (Horn clauses) that are comprehensible to humans. It allows the background knowledge to be incrementally extended without requiring the entire model to be re-learned. Meanwhile, the comprehensibility of symbolic rules makes it easier for users to understand and verify induced models and even edit them.

ILP learns theories in the form of Horn clause logic programs. Extending Horn clauses with negation as failure (NAF) results in more powerful applications becom-

ing possible as inferences can be made even in absence of information. This extension of Horn clauses with NAF where the meaning is computed using the stable model semantics [6]—called Answer Set Programming[1]—has many powerful applications. Generalizing ILP to learning answer set programs also makes ILP more powerful. For a complete discussion on the necessity of NAF in ILP, we refer the reader to [17].

Once NAF semantics is allowed into ILP systems, they should be able to deal with multiple stable models which arise due to presence of mutually recursive rules involving negation (called *even cycles*) [6] such as:

```
p :- not q.
q :- not p.
```

XHAIL [16], ASPAL [2], ILASP [8] are among the recently emerged systems capable of learning non-monotonic logic programs. However, they all resort to an exhaustive search for the hypothesis. The exhaustive search is not scalable on practical datasets. For instance, (all versions of) ILASP training procedure times-out after couple of hours on "Moral Reasoner" a dataset from the UCI repository. This is a small dataset containing roughly 200 examples and 50 candidate predicates in language bias.

In contrast, traditional ILP systems (that only learn Horn clauses), use heuristics to guide their search. Use of heuristics allows them to avoid an exhaustive search. These systems usually start with the most general clauses and then specialize them. They are better suited for large-scale data-sets with noise, since the search can be easily guided by heuristics. FOIL [15] is a representative of such algorithms. However, handling negation in FOIL is somewhat problematic as we discuss in [20]. Also, FOIL cannot handle background knowledge with multiple stable models, nor it can induce answer set programs.

Recently we developed an algorithm called FOLD [20] to automate inductive learning of default theories represented as stratified answer set programs. FOLD (First Order Learner of Default rules) extends the FOIL algorithm and is able to learn answer set programs that represent the underlying knowledge very succinctly. However, FOLD is only limited to dealing with stratified answer set programs, i.e., mutually recursive rules through negation are not allowed in the background knowledge or the hypothesis. Thus, FOLD is incapable of handling cases where the background knowledge or the hypothesis admits multiple stable models. In this paper, we extend the FOLD algorithm to allow both the background knowledge and the hypothesis to have multiple stable models. The extended FOLD algorithm—called the XFOLD algorithm—is much more general than previously proposed methods.

This paper makes the following novel contributions: First, it extends FOLD with non-observation learning capability (Section 3). Then it presents the XFOLD algorithm, an extension of our previous FOLD algorithm, that can handle background knowledge with multiple stable models as well as allow inducing of hypotheses that have multiple stable models (Section 4). To the best of our knowledge, XFOLD is the first heuristic based algorithm to induce such hypotheses. The XFOLD algorithm can learn ASP programs to solve combinatorial problems such as graph-coloring and N-queens. Because

[1] We use the term answer set programming in a generic sense to refer to normal logic programs, i.e., logic programs extended with NAF, whose semantics is given in terms of stable models [5].

the XFOLD algorithm is based on heuristic search, it is also scalable. Lack of scalability is a major problem in previous approaches. We assume that the reader is familiar with answer set programming and stable model semantics. The book by Gelfond and Kahl [5] is a good source of background material.

2 Background

The FOLD algorithm [20] which is an extension of FOIL [15], learns a concept in terms of a default and possibly multiple exceptions (and exceptions to exceptions, and so on). FOLD tries first to learn the default by specializing a general rule of the form $\{$`target(`$V_1,...,V_n$`) :- true.`$\}$ with positive literals. As in FOIL, each specialization must rule out some already covered negative examples without decreasing the number of positive examples covered significantly. Unlike FOIL, no negative literal is used at this stage. Once the heuristic score (i.e., *information gain*) becomes zero, this process stops. At this point, if any negative example is still covered, they must be either noisy data or exceptions to the current hypothesis. Exceptions could be learned by swapping the current positive and negative examples, then calling the same algorithm recursively. As a result of this recursive process, we can learn exceptions to exceptions, and so on. The FOLD ILP problem of learning a *target* predicate is formally defined as follows:
Given

1. a background theory B, in the form of an extended logic program, i.e., clauses of the form $h \leftarrow l_1,...,l_m,$ *not* $l_{m+1},...,$ *not* l_n, where $l_1,...,l_n$ are positive literals and *not* denotes *negation-as-failure* (NAF) and B has no even cycle
2. two disjoint sets of ground target predicates E^+, E^- known as positive and negative examples respectively
3. a hypothesis language of function free predicates L, and a refinement operator ρ under $\theta - subsumption$ [14] that would disallow even cycles.

Find a set of clauses H such that:

- $\forall e \in E^+,\ B \cup H \models e$
- $\forall e \in E^-,\ B \cup H \not\models e$
- B and H are consistent.

In this paper, we extend the FOLD algorithm to relax all three preconditions stated above so that: (i) The background knowledge B can potentially have more than one stable model (section 4); (ii) Examples and target predicate could be different (Section 3); (iii) The induced hypotheses can have cycles through negation. (Section 4)

3 Non-Observation Predicate Learning in FOLD

In usual machine learning setting of "Observation Predicate Learning" (OPL), examples and hypotheses define the same predicate. In contrast, non-OPL setting allows to have examples other than the ground target predicate. Non-OPL setting is natural for many problems [13]. Therefore, a natural extension of FOLD would be to include non-OPL

setting. Intuitively, non-OPL requires to obtain how each non-target example impacts the correct hypothesis in terms of target ground atoms. The following example shows how a non-target ground predicate could be expressed in terms of positive and negative examples of the target predicate.

Example 1 *Consider the following Background knowledge. Given the positive example set* $E^+ = \{p(a), r(c)\}, E^- = \{p(d)\}$*, we want to learn the target* $r(X)$*.*

```
(1) p(X) :- s(X), not r(X).        (3) q(a,b).
(2) s(X) :- q(X,Y), r(Y).          (4) s(d).
```

Since $B \cup H$ *must imply* $p(a)$*, from rule (1) we get* $s(a)$ *must hold and* $r(a)$ *should not. For* $s(a)$ *to hold, from rule (2) we get* $q(a,Y), r(Y)$ *must hold. Such* Y *indeed exists from fact (3). Therefore,* $r(b)$ *must hold too.* $p(a)$ *requires* $r(b)$ *and not* $r(a)$*. Therefore,* $p(a)$ *can be replaced by new examples, i.e.,* $r(b)$ *a new positive example, and* $r(a)$*, a new negative example. The impact of* $p(d)$ *as a negative example is to force* $r(d)$ *not to hold, because, from (4) we get* $s(d)$ *holds, therefore,* $r(d)$ *must not. Hence,* $r(d)$ *is a new negative example and replaces* $p(d)$*.*

The computation performed in Example 1 to replace non-target examples with target examples is realized using abduction in a goal-directed answer set programming system called s(ASP) [10, 7]. The s(ASP) system takes an answer set program P and a query goal Q as inputs and enumerates all answer sets that contain the propositions/predicates in Q. This enumeration employs co-inductive SLD resolution to systematically compute elements of the greatest fixed point (GFP) of a program via backtracking. The advantage of s(ASP) over other answer set solvers is that it would lift the restriction that answer set programs must be finitely groundable. In order to process a query Q, s(ASP) would produce a set called the "partial answer set" containing the elements that are necessary to establish Q. The s(ASP) system also allows a query to run abductively, by first defining a set of predicates as abducible. By doing so, if success of a query Q depends on assuming a fact that belongs to the set of abducibles, Q abductively succeeds and the abducibles are added to the set of partial answer set associated with Q.

Algorithm 1 shows the required steps in order to solve a non-OPL ILP problem using FOLD. In case of Example 1, $p(a)$ is a non-target example. By running s(ASP) and defining `#abducible r(X)`, the following partial answer set is produced by s(ASP) on the query `?- p(a).`:

```
{p(a), q(a,b), r(b), s(a), not r(a)}
```

$r(b)$ and $r(a)$ are added to the set of positive and negative examples, respectively. It should be noted that the above set of predicates are relevant to establish the query `?- p(a)`. In practice, this is a small subset of the original stable model. The fact that s(ASP) does not ground the answer set program, makes this approach scalable comparing to SAT based answer set solvers.

4 Induction of Answer Set Programs with Multiple Stable Models

In this section we extend our FOLD algorithm to learn normal logic programs that potentially have multiple stable models. The significance of Answer Set Programming

Algorithm 1 Non-OPL Version of FOLD Algorithm

Input: $target, B, E^+, E^-$
Output: Hypothesis H

1: $abduced^+, abduced^-$ = { }
2: Let Q be the query: $? - E^+,\ not\ E^-$
3: Run Q on s(ASP) $\langle B, \#abducibles = \{target\}\rangle$ ▷ Run s(ASP) with B as input
4: Let P = partial answer set associated with Q
5: **for each** $p \in P\ s.t\ pred(p)\ == target$ **do**
6:     **if** $sign(p)\ == +$ **then**
7:         $abduced^+ \leftarrow abduced^+ \cup \{p\}$
8:     **else**
9:         $abduced^- \leftarrow abduced^- \cup \{p\}$
10:     **end if**
11: **end for**
12: $E^+ \leftarrow E^+ \cup\ abduced^+$
13: $E^- \leftarrow E^- \cup\ abduced^-$
14: Run $FOLD\langle B, E^+, E^-, target\rangle$

paradigm is that it provides a declarative semantics under which each stable model is associated with one (alternative) solution to the problem described by the program. Typical problems of this kind are combinatorial problems, e.g., graph coloring and N-queens. In graph coloring, one should find different ways of coloring nodes of a graph without coloring two nodes connected by an edge with the same color. N-queen is the problem of placing N queens in a chessboard of size $N \times N$ so that no two queens attack each other.

In order to inductively learn such programs, the ILP problem definition needs to be revisited. In the new scenario, positive examples $e \in E^+$, may not hold in every model. Therefore, the ILP problem described in the background section would only allow learning of predicates that hold in all answer sets. This is too restrictive. Brave induction [18], in contrast, allows examples to hold only in some stable models of $B \cup H$. However, as stated in [8], and we will show using examples, this is not enough when it comes to learning global constraints (i.e, rules with empty head)[2]. Learning global constraints is essential because certain combinations may have to be excluded from *all* answer sets.

When $B \cup H$ has multiple stable models, there will be some instances of target predicate that would hold in all, none, or some of the stable models. Brave induction is not able to express situations in which a predicate should hold in all or none of the stable models. An example is a graph in which node 1 is colored red. In such a case, none of node 1's neighbors should be colored red. If node 1 happens to have node 2 as a neighbor, brave induction is not able to express the fact that if the atom `red(1)` appears

[2] Recall that in answer set programming, a constraint is expressed as a headless rule of the form

```
:- B.
```

which states that `B` must be false. A headless rule is really a short-form of rules of the form (called odd loops over negation [5]):

```
p :- B, not p.
```

in any stable model of $B \cup H$, `red(2)` should not. In [8], the authors propose a new paradigm called learning from partial answer sets that overcomes these limitations. We also adopt this paradigm in this work. Next, we present our XFOLD algorithm.

Definition 1. *A partial interpretation E is a pair $E = \langle E^{inc}, E^{exc} \rangle$ of sets of ground atoms called inclusions and exclusions, respectively. Let $A \in AS(B \cup H)$ denote a stable model of $B \cup H$. A extends $\langle E^{inc}, E^{exc} \rangle$ if and only if $(E^{inc} \subseteq A) \wedge (E^{exc} \cap A = \emptyset)$.*

Example 2 *Consider the following background knowledge about a group of friends some of whom are in conflict with others. The individuals in conflict will not attend a party together. Also, they cannot attend a party if they work at the time the party is held. We want our ILP algorithm to discover the rule(s) that will determine who will go to the party based on the set of partial interpretations provided.*

```
B : conflict(X,Y) :- person(X), person(Y), conflict(Y,X).
    works(X) :- person(X), not off(X).
    off(X) :- person(X), not works(X).
    person(p1). person(p2). conflict(p1,p4).
    person(p3). person(p4). person(p5). conflict(p2,p3).
```

Some of the partial interpretations are as follows:
The predicates g,w,o abbreviate goesToParty, works, off respectively:
$E_1 = \{\langle g(p1), g(p2), o(p1), o(p2), w(p3), o(p4), w(p5)\rangle, \langle g(p3), g(p4), g(p5)\rangle\}$
$E_2 = \{\langle g(p3), g(p4), g(p5), o(p1), o(p2), o(p3), o(p4), o(p5)\rangle, \langle g(p1), g(p2)\rangle\}$
$E_3 = \{\langle g(p1), g(p3), g(p5), o(p1), o(p2), o(p3), w(p4), o(p5)\rangle, \langle g(p2), g(p4)\rangle\}$
$E_4 = \{\langle g(p2), g(p5), g(p5), w(p1), o(p2), w(p3), w(p4), o(p5)\rangle, \langle g(p1), g(p3), g(p4)\rangle\}$

In the above example, each E_i for i = 1,2,3,4 is a partial interpretation and should be extended by at least one stable model of $B \cup H$ for a learned hypothesis H. For instance, let's consider the hypothesis $H_1 = \{$`goesToParty(X) :- off(X)`$\}$ for learning the target predicate `goesToParty(X)`. By plugging the background knowledge, the non-target predicates in E_1, and the hypothesis H_1 into an ASP solver (CLASP [4] in our case), the stable model returned by the solver would contain the following:

$\{$`goesToParty(p1),goesToParty(p2),goesToParty(p4)`$\}$.

It does not extend E_1. Although, $E_1^{inc} \subseteq AS(B \cup H_1)$ but $AS(B \cup H_1) \cap E_1^{exc} \neq \emptyset$. It should be noted that non-target predicates are treated as background knowledge upon calling ASP solver to compute the stable model of $B \cup H$.

Definition 2. *An XFOLD problem is defined as a tuple $P = \langle B, L, E^+, E^-, T \rangle$. B is a answer set program with potentially multiple stable models called the background knowledge. L is the language-bias such that $L = \langle M_h, M_b \rangle$, where M_h (resp. M_b) are called the head (resp. body) mode declarations [12].*

Each mode declaration $m_h \in M_h$ (resp. $m_b \in M_b$) is a literal whose abstracted arguments are either variable v or constant c. Type of a variable is a predicate defined in B. The domain of each constant should be defined separately. Hypothesis h is said to be compatible with a mode declaration m if each instance of variable in m is replaced by a variable, and every constant takes a value from the associated domain. The set of candidate predicates in the greedy search algorithm are selected from $M_b \cup M_h$.

XFOLD is extended with mode declaration to make sure that every clause generated is safe for the ASP solver CLASP as it needs to ground the program. To obtain a finite grounded program, CLASP must ensure that every variable is safe. A variable in *head* is *safe* if it occurs in a positive literal of body. XFOLD adds predicates required to ensure safety, but to keep our examples simple, we omit safety predicates in the paper. E^+ and E^- are sets of partial interpretations called positive and negative examples, respectively. $T \in M_h$ is the target predicate's name. Each XFOLD run learns a single target predicate. A hypothesis $h \in L$ is an inductive solution of T if and only if:

1. $\forall e^+ \in E^+ \exists A \in AS(B \cup H)$ such that A extends e^+
2. $\forall e^- \in E^- \nexists A \in AS(B \cup H)$ such that A extends e^-

The above definition adopted from [8] subsumes brave and cautious induction semantics [18]. Positive examples should be extended by at least one stable model of $B \cup H$ (brave induction). In contrast, no stable model of $B \cup H$ extends negative examples (cautious induction). The *generate and test* problems such as N-queen and graph coloring could be induced using our XFOLD algorithm. It suffices to use positive examples for learning the *generate* part and negative examples for learning the *test* part.

Figure 1 represents the input to the XFOLD algorithm for learning an answer set program for graph coloring. Every positive example states if a node is colored red, then that node cannot be painted blue or green. Likewise for blue and green. However, this is not enough to learn the constraint that two nodes connected by an edge cannot have the same color. To learn this constraint, negative examples are needed. For instance, E_1^-, states that if any stable model of $B \cup H$ contains $\{$`red(1)`$\}$, in order not to extend E_1^-, it should contain $\{$`not red(2)`$\}$ or equivalently, it should not contain $\{$`red(2)`$\}$. Intuitively, XFOLD is similar to FOLD and FOIL: To specialize a clause *cl*, for every

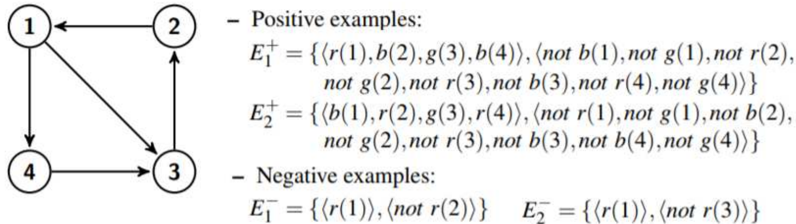

Fig. 1: Partial interpretations as examples in graph coloring problem

positive example $e \in E^+$, the background knowledge B, all non-target predicates in e^{inc} and *cl* are passed to the ASP solver as inputs. The resulting answer set is compared with the target predicates in e^{inc} and e^{exc} to compute a partial score. Next, by summing up all partial scores, total score of that clause is computed. Among all candidate clauses, the one with highest total score is selected. Once for all $e \in E^+$ no target predicate in e^{exc} is covered, the internal loop finishes and the discovered rule(s) are added to the learned theory. Just like FOLD, if no literal with positive score exists, swapping occurs on each remaining partial interpretation and the XFOLD algorithm is recursively called. In this case, instead of introducing abnormality predicates, the negation symbol, "-",

Algorithm 2 The XFOLD Algorithm

Input: $target$, B, $\{e = (e^{inc}, e^{exc}) | e \in E^+\}$
Output: Hypothesis H

```
function SPECIALIZE(cl, B, E+)                    ▷ Other functions remain unchanged as in FOLD
    while ∃e ∈ E+ such that e^exc != ∅ do
        for each c ∈ ρ(cl) do                                ▷ FOIL inner loop (refinement)
            for each e_i ∈ E+ do
                compute partial_score[i][c]                  ▷ partial score for each clause
            end for
            total_score[c] = Σ_{e_i∈E+} partial_score[i][c]
        end for
        Let c_best, max_score, be the clause with the highest score and its associated score
        if max_score > 0 then
            cl ← c_best
            H ← H ∪ {cl}
        else
            E_swapped+ = Swap(E+)
            XFold(B, E_swapped+, −target)
        end if
        update E+
    end while
end function
```

is prefixed to the current target predicate to indicate that the algorithm is now trying to learn the negation of concept being learned. It should also be noted that swapping examples is performed slightly differently due to the existence of partial interpretations. For each $e \in E^+$ the following operations are performed upon swapping:

1. $\forall t \in e^{inc}$, where t is an old target atom already covered and removed, t is restored
2. $\forall t \in e^{inc}$, where t is an old target atom, $-t$ is added to e^{exc}
3. $\forall t \in e^{exc}$, where t is an old target atom, $-t$ is added to e^{inc}
4. $T \leftarrow -T$. (Target predicate T now becomes its negation, -T)

Figure 2 shows execution of XFOLD on Example 2. At the end of first iteration, the predicate `off(X)` gets the highest score. E_4 will be removed as it is already covered by the current hypothesis. In the second iteration, all candidate literals fail to get a positive score. Therefore, swapping of positive and negative examples occurs and algorithm tries to learn the predicate `-goesToParty(X)`. Since the new target predicate is `-goesToParty(X)`, all ground atoms of `goesToParty` in E^{inc} are restored back. The old target atoms in E^{exc} are transformed to negated version and become members of E^{inc}. In Figure 2, after one iteration E_4 is removed because all target atoms in E^{inc} are already covered and targets atoms in E^{exc} are already excluded. After swapping, XFOLD is recursively called to learn `-goesToParty`. After 2 iterations, all examples are covered and the algorithm terminates.

In Example 2, we haven't introduced any explicit negative example. Nevertheless, the algorithm was able to successfully find the cases in which the original target predicate does not hold (via learning `-goesToParty(X)` predicate). In general, it is not

After iteration #1: {goesToParty(X) :- off(X)}

E_1 = {⟨o(p_1), o(p_2), w(p_3), o(p_4), w(p_5)⟩, ⟨g(p_4)⟩}

E_2 = {⟨o(p_1), o(p_2), o(p_3), o(p_4), o(p_5)⟩ , ⟨g(p_1), g(p_2)⟩}

E_3 = {⟨o(p_1), o(p_2), o(p_3), w(p_4), o(p_5)⟩, ⟨g(p_2)⟩}

~~E_4 = {⟨w(p_1), o(p_2), w(p_3), w(p_4), o(p_5)⟩, ⟨⟩}~~

After swapping E_{inc}, E_{exc}

E_1 = {⟨-g(p_4), g(p_1), g(p_2), o(p_1), o(p_2), w(p_3), o(p_4), w(p_5)⟩, ⟨-g(p_1), -g(p_2)⟩}

E_2 = {⟨-g(p_1), -g(p_2), g(p_3), g(p_4), g(p_5), o(p_1), o(p_2), o(p_3), o(p_4), o(p_5)⟩, ⟨-g(p_3), -g(p_4), -g(p_5)⟩}

E_3 = {⟨-g(p_2), g(p_1), g(p_3), g(p_5), o(p_1), o(p_2), o(p_3), w(p_4), o(p_5)⟩, ⟨-g(p_1), -g(p_3), -g(p_5)⟩}

After iteration #1: { -goesToParty(X) :- conflict(X,Y) }

E_1 = {⟨g(p_1), g(p_2), o(p_1), o(p_2), w(p_3), o(p_4), w(p_5)⟩, ⟨-g(p_1), -g(p_2)⟩}

E_2 = {⟨g(p_3), g(p_4), g(p_5), o(p_1), o(p_2), o(p_3), o(p_4), o(p_5)⟩, ⟨-g(p_3), -g(p_4)⟩}

E_3 = {⟨g(p_1), g(p_3), g(p_5), o(p_1), o(p_2), o(p_3), w(p_4), o(p_5)⟩, ⟨-g(p_1), -g(p_3)⟩}

Iteration #2: { -goesToParty(X) :- conflict(X,Y), goesToParty(Y) }

~~E_1 = {⟨g(p_1), g(p_2), o(p_1), o(p_2), w(p_3), o(p_4), w(p_5)⟩, ⟨⟩}~~

~~E_2 = {⟨g(p_3), g(p_4), g(p_5), o(p_1), o(p_2), o(p_3), o(p_4), o(p_5)⟩, ⟨⟩}~~

~~E_3 = {⟨g(p_1), g(p_3), g(p_5), o(p_1), o(p_2), o(p_3), w(p_4), o(p_5)⟩, ⟨⟩}~~

Hypothesis = { {goesToParty(X) :- off(X), not -goesToParty(X).}, {-goesToParty(X) :- conflict(X,Y), goesToParty(Y).} }

Fig. 2: Trace of XFOLD execution on the Party Example

always feasible for the algorithm to figure out prohibited patterns without getting to see a very large number of positive examples.

5 Application: Combinatorial Problems

A well-known methodology for declarative problem solving is the *generate and test* methodology, whereby possible solutions to a problem are generated first, and then non-solutions are eliminated by testing. In Answer Set Programming, the *generate* part is encoded by enumerating the possibilities by introducing even cycles. The *test* part is realized by having constraints that would eliminate answer sets that violate the test conditions. ASP syntax allows rules of the form $l\{h_1,...,h_k\}u$ such that $0 \leq l \leq u \leq k$ and $\forall i \in [1,k]$, $h_i \in L$, where L is the language bias. This syntactic sugar for combination of even cycles and constraints is called *choice rule* in the literature [5].

ILASP [8] directly searches for choice rules by including them in the search space. XFOLD, on the other hand, performs the search based on θ-subsumption [14] and hence disallows search for choice rule hypotheses. Instead, it directly learns even cycles as well as constraints. This is advantageous as it allows for more sophisticated and flexible language bias.

It turns out that inducing the *generate* part in a combinatorial problem such as graph-coloring requires an extra step compared to the FOLD algorithm. For instance, `red(X)` predicate has the following clause:

```
red(X):- not blue(X), not green(X).
```

To enable XFOLD to induce such a rule, we adopted the "Mathews Correlation Coefficient" (MCC) [22] measure to perform the task of feature selection. MCC is calculated as: $MCC = \frac{TP \times TN - FP \times FN}{\sqrt{(TP+FP)(TP+FN)(TN+FP)(TN+FN)}}$

This measure takes into account all the four terms TP (true positive), TN (true negative), FP (false positive) and FN (false negative) in the confusion matrix and is able to fairly assess the quality of classification even when the ratio of positive tuples to the negative tuples is not close to 1. The MCC values range from -1 to +1. A coefficient of +1 represents a perfect classification, 0 represents a classification that is no better than a random classifier, and -1 indicates total disagreement between the predicted and the actual labels. MCC cannot replace XFOLD heuristic score, i.e, *information gain*, because the latter tries to maximize the coverage of positive examples, while the former only maximally discriminates between the positives and negatives. Nevertheless, for the purpose of feature extraction among the negated literals which are disallowed in XFOLD algorithm, MCC can be applied quite effectively. For that matter, before running XFOLD algorithm, the MCC score of all candidate literals are computed. If a predicate scores "close" to +1, the predicate itself is added to the language bias. If it scores "close" to -1, its negation is added to the language bias. For example, in case of learning `red(X)`, after running the feature extraction on the graph given in Figure 1, XFOLD computes the scores -0.7, -0.5 for `green(X)` and `blue(X)`, respectively. Therefore, {`not green(X),not blue(X)`} are appended to the list of candidate predicates. Now, after running the XFOLD algorithm, after two iterations of the inner loop, it would produce the following rule:

```
red(X) :- not green(X), not blue(X).
```

Corresponding rules for `green(X)` and `blue(X)` are learned in a similar manner. This essentially takes care of the *generate* part of the combinatorial algorithm. In order to learn the *test* part for graph coloring, we need the negative examples shown in Figure 1. It should be noted that in order to learn a constraint, we first learn a new target predicate which is the negation of the original one. Then we shift the negated predicate from the head to the body inverting its sign in the process. That is, we first learn a clause of the form {`-T :-` b_1, b_2 ... b_n.} which is then transformed into the following constraint: {`:-` b_1, b_2 ... b_n, `T`.} Thus, the following steps should be taken to learn constraints from negative examples:

1. Add rule(s) induced for *generate* part to B.
2. $\forall e^+ \in E^+, e^- \in E^-$, if $e^-_{inc} \subseteq e^+_{inc}$:
   - **if** e^-_{exc} is of the form (`not` $p(V_1, ...V_m)$) **then** $e^+_{inc} \leftarrow e^+_{inc} \cup \{-p(V_1, ...V_m)\}$
   - **else** $e^+_{exc} \leftarrow e^+_{exc} \cup \{-p(V_1, ...V_m)\}$
3. compute the contrapositive form of the rule(s) learned in *generate* part and remove the body predicates from the list of candidate predicates
4. run XFOLD to learn `p`
5. shift `-p` from the head to the body for each rule returned by XFOLD

The contrapositive form of a clause is computed by negating the head and applying the De Morgan's law to the body. The resulting disjunctions are resolved by separating them

into new clauses. For instance, the contrapositive of {`red(X) :- not green(X), not blue(X)`} is obtained as follows:
{`-red(X) :- green(X)`},{`-red(X) :- blue(X)`}. Without step 3, XFOLD would learn these trivial clauses. However, as soon as those trivial choices are removed from search space, XFOLD algorithm comes up with the next best hypothesis which is as follows: {`-red(X) :- edge(X,Y), red(Y).`} Shifting `-red(X)` to the body yields

Algorithm 3 The generate and Test Version of XFOLD algorithm

Input: $L = \langle M_h, M_b \rangle, B, E^+, E^-$
Output: Hypothesis H
1: % - **Induction of "generate" part** - %
2: Initialize $H \leftarrow \emptyset$
3: **for each** $t \in M_h$ **do**
4:     **for each** $l \in M_b$ **do**
5:         $L \leftarrow L \cup \{not\ l\}$ if MCC of the clause {`t :- not l`} is close to +1
6:     **end for**
7:     $h_t \leftarrow$ XFOLD $\langle B, L, E^+_{inc}, E^+_{exc}, t \rangle$
8:     $H \leftarrow H \cup h_t$
9: **end for**
10: $B \leftarrow B \cup H$
11: % - **Induction of "test" part** - %
12: **for each** $t \in M_h$ **do**
13:     **for each** $e^+ \in E^+, e^- \in E^-$ **do**
14:         **if** $e^-_{inc} \subseteq e^+_{inc}$ **then**
15:             **if** e^-_{exc} is of the form `not t`$(V_1, ..., V_m)$ **then**
16:                 $e^+_{inc} \leftarrow e^+_{inc} \cup \{$`-t`$(V_1, ..., V_m)\}$
17:             **else**
18:                 $e^+_{exc} \leftarrow e^+_{exc} \cup \{$`-t`$(V_1, ..., V_m)\}$
19:             **end if**
20:         **end if**
21:     **end for**
22: **end for**
23: compute the contrapositive form for each $h \in H$ in *generate* part and remove the body predicates from the list of candidate predicates L
24: **for each** $t \in M_h$ **do**
25:     $h_t \leftarrow$ XFOLD $\langle B, L, E^+_{inc}, E^+_{exc}, -t \rangle$
26:     shift -t from the head to the body to get a constraint $\hat{h}_t$
27:     $H \leftarrow H \cup \{\hat{h}_t\}$
28: **end for**

the following constraint: `:- red(X),edge(X,Y),red(Y).` In graph coloring problem, M_h = {`red(X), green(X), blue(X)`}. Once similar examples for `green(X)` and `blue(X)` are provided, XFOLD is able to learn the complete solution as shown below:

```
red(X) :- not green(X), not blue(X).
green(X) :- not blue(X), not red(X).
blue(X) :- not green(X), not red(X).
:- red(X), edge(X,Y), red(Y).
:- blue(X), edge(X,Y), blue(Y).
:- green(X), edge(X,Y), green(Y).
```

Algorithm 3 shows how XFOLD induces a *generate and test* hypothesis.

Example 3 *Learning an answer set program for the 4-queen problem. The following items are assumed: Background knowledge B including predicates describing a* 4×4 *board, rules describing different ways through which two queens attack each other and examples of the following form:*

B: $attack_r(R_1,C_1,R_2,C_2)$:- $q(R_1,C_1), q(R_2,C_2), C_1 != C_2,\ R_1 = R_2.$
$attack_c(R_1,C_1,R_2,C_2)$:- $q(R_1,C_1), q(R_2,C_2), R_1 != R_2,\ C_1 = C_2.$
$attack_d(R_1,C_1,R_2,C_2)$:- $q(R_1,C_1), q(R_2,C_2), R_1 != R_2, R_1 - C_1 = R_2 - C_2.$
$attack_d(R_1,C_1,R_2,C_2)$:- $q(R_1,C_1), q(R_2,C_2), R_1 != R_2, R_1 + C_1 = R_2 + C_2.$

E: $E_1^+ = \{\langle q(2,1), q(4,2), q(1,3), q(3,4)\rangle, \langle q(1,1), q(1,2), ..., q(4,4)\rangle\}$
...
$E_1^- = \{\langle q(2,1)\rangle, \langle not\ q(2,2)\rangle\}$
$E_2^- = \{\langle q(2,1)\rangle, \langle not\ q(2,3)\rangle\}$
$E_3^- = \{\langle q(4,2)\rangle, \langle not\ q(1,2)\rangle\}$
$E_4^- = \{\langle q(4,2)\rangle, \langle not\ q(2,3)\rangle\}$

As far as the *generate* part is concerned, XFOLD algorithm learns the following program:

```
q(X,Y) :- not -q(X,Y).
-q(X,Y) :- not q(X,Y).
```

The predicate -q(X,Y) is introduced by XFOLD algorithm as a result of swapping the examples and calling itself recursively. After computing the contrapositive form, `q(X,Y), -q(X,Y)` are removed from the list of candidate predicates. Then based on the examples provided in Example 3, XFOLD would learn the following rules:

```
-q(V1,V2) :- attack_r(V1,V2,V3,V4).
-q(V1,V2) :- attack_c(V1,V2,V3,V4).
-q(V1,V2) :- attack_d(V1,V2,V3,V4).
```

After shifting the predicate -q(V_1,V_2) to the body, we get the following constraint:

```
:- q(V1,V2), attack_r(V1,V2,V3,V4).
:- q(V1,V2), attack_c(V1,V2,V3,V4).
:- q(V1,V2), attack_d(V1,V2,V3,V4).
```

It should be noted that, since XFOLD is a sequential covering algorithm like FOIL, it takes three iterations before it can cover all examples which in turn becomes three constraints as shown above.

6 Experiments and Results

Table 1 reports the classification accuracy using 10-fold cross-validation and runtime measurements of Aleph, FOLD and XFOLD on a number of UCI datasets [9] and combinatorial problems discussed in this paper. In [20] we compare our FOLD algorithm with Aleph which is a state-of-the-art ILP system. However, Aleph [21] does not support multiple stable model ILP. Therefore, we can only compare our results with that of ILASP. In case of UCI datasets, the “Size” column denotes the number of data samples,

whereas, in graph-coloring (N-queen) it denotes the number of nodes(board size) respectively. We have also examined the application of statistical feature selection on the performance of our XFOLD algorithm. We report a significant improvement due to the application of a scalable feature-selection method, i.e., xgboost, prior to invoking the learning algorithm. Exclusion of low ranked features and the use of negation-as-failure results in a significant improvement over the accuracy of learned hypotheses.

"Extreme Gradient Boosting" (xgboost) [1] is a scalable and powerful ensemble classifier based on decision trees that provides a feature importance score. Since, in ILP we deal with propositions, it makes sense to discretize numeric features first using MDL method [3]. In this method, for each numeric feature categories are defined such that the overall information gain is maximized.

Next, a dataset that now contains only categorical features is propositionalized. That is, every value belonging to the domain of a categorical feature turns into a new binary feature. This is called *one hot encoding*. One hot encoding makes the feature selection more fine grained. This is because, in this technique instead of measuring the contribution of a feature as a whole, the importance of every value from the domain of that feature is measured. Then the data set is fed into xgboost which ranks each binary feature based on its importance in the classification. From the xgboost's output, the M lower ranked features are filtered out of the XFOLD language bias. The optimal M should be computed via cross-validation.

In small problems such as graph coloring, ILASP slightly outperforms our XFOLD algorithm due to embedding the learning algorithm in the ASP solver engine. In a larger data set such as Moral reasoner with 202 examples and 50 predicates, there are potentially 3^{50} different hypotheses to choose from. This is because, for each predicate it can either be included positively, included negatively or excluded. In this case, ILASP times out after couple of hours.

		Accuracy (%)				Running Time (s)	
Dataset	Size	Aleph	Fold	XFold	ILASP	XFold	ILASP
breast-cancer	286	70	82	88	—	4.1	timed-out
moral	202	96	96	100	—	4.8	timed-out
diabetes	768	73	86	89	—	27.2	timed-out
graph-coloring	4	—	—	100	100	8	4.5
graph-coloring	8	—	—	100	100	8.9	3.5
N-queen	4×4	—	—	100	100	9.5	5
N-queen	8×8	—	—	100	100	9.9	6.2

Table 1: XFold Evaluation on UCI benchmarks and Combinatorial Problems

7 Related Work

A survey of extending Horn clause based ILP to non-monotonic logics can be found in [17]. "Stable ILP" [19] was the first effort to explore the expressiveness of background knowledge with multiple stable models. In [17], Sakama introduces algorithms to induce a *categorical* logic program[3] given the answer set of the background knowledge and *either* positive *or* negative examples. Essentially, given a single answer set,

[3] A categorical logic program is an answer set program with at most one stable model.

Sakama tries to induce a program that has that answer set as a stable model. In [18], Sakama and Inoue extend their work to learn from multiple answer sets. They introduce *brave* induction, where the learned hypothesis H is such that *some* of the answer sets of $B \cup H$ cover the positive examples. The limitation of this work is that it accepts only one positive example as a conjunction of atoms. It does not take into account negative examples at all. Cautious induction, the counterpart of brave induction, is also too restricted as it can only induce atoms in the intersection of all stable models. Thus, neither brave induction nor cautious induction are able to express situations where something should hold in all or none of the stable models. An example of this limitation arises in the graph coloring problem where the following should hold in all answer sets: no two neighboring nodes in a graph should be painted the same color.

ASPAL [2] is the first ILP system to learn answer set programs by encoding ILP problems as ASP programs and having an ASP solver find the hypothesis. Its successor ILASP [8], is a pioneering ILP system capable of inducing hypotheses expressed as answer set programs too. ILASP defines a framework that subsumes brave/cautious induction and allows much broader class of problems relating to learning answer set programs to be handled by ILP. However, the algorithm exhaustively searches the space of possible clauses to find one that is consistent with all examples and background knowledge. The Exhaustive search is a weaknesses that limits the applicability of ILASP to many useful situations. Our research presented in this paper does not suffer from this issue.

XHAIL [16] is another ILP system capable of learning non-monotonic logic programs. It heavily incorporates abductive logic programming to search for hypotheses. It uses a similar language-bias as ILASP does, and thus suffers from the limitations similar to ILASP. It also does not support the notion of inducing answer set programs from partial answer sets.

8 Conclusion and Future Work

In this paper we presented the first heuristic-based algorithm to inductively learn normal logic programs with multiple stable models. The advantage of this work over similar ILP systems such as ILASP [8] is that unlike these systems, XFOLD does not perform an exhaustive search to discover the “best” hypothesis. XFOLD adopts a greedy approach, guided by heuristics, that is scalable and noise resilient. We also showed how our algorithm could be applied to induce declarative logic programs that follow the *generate and test* paradigm for finding solutions to combinatorial problems such as graph-coloring and N-queens.

There are two main avenues for future work: (i) handling large datasets using methods similar to QuickFoil [22]. In QuickFoil, all the operations of FOIL are performed in a database engine. Such an implementation, along with pruning and query optimization tricks can make the XFOLD training much faster; (ii) XFOLD learns function-free answer set programs. We plan to investigate extending the language bias towards accommodating functions.

Acknowledgements

Authors are partially supported by NSF Grant IIS 1718945.

Bibliography

[1] Chen, T., Guestrin, C.: Xgboost: A scalable tree boosting system. In: SIGKDD 22. pp. 785–794. KDD '16, ACM, New York, NY, USA (2016)

[2] Corapi, D., Russo, A., Lupu, E.: Ilp in answer set programming. In: ILP 21 - 2011, UK. pp. 91–97 (2011)

[3] Fayyad, U.M., Irani, K.B.: Multi-interval discretization of continuous-valued attributes for classification learning. In: IJCAI. pp. 1022–1029 (1993)

[4] Gebser, M., Kaufmann, B., Schaub, T.: Conflict-driven answer set solving: From theory to practice. Artificial Intelligence 187-188, 52–89 (2012)

[5] Gelfond, M., Kahl, Y.: Knowledge Representation, Reasoning, and the Design of Intelligent Agents: The ASP Approach. Cambridge University Press (2014)

[6] Gelfond, M., Lifschitz, V.: The stable model semantics for logic programming. In: Logic Programming,Proceedings ICLP,1988 (2 Volumes). pp. 1070–1080 (1988)

[7] Gupta, G.: A case for query-driven predicate asp. In: ARCADE'17,1st Int. Workshop on Automated Reasoning,Sweden,2017. pp. 64–68 (2017)

[8] Law, M., Russo, A., Broda, K.: Inductive learning of answer set programs. In: Logics in Artificial Intelligence - 14th European Conference, JELIA (2014)

[9] Lichman, M.: UCI,ml repository, http://archive.ics.uci.edu/ml (2013)

[10] Marple, K., Salazar, E., Gupta, G.: Computing stable models of normal logic programs without grounding (2017), http://arxiv.org/abs/1709.00501

[11] Muggleton, S.: Inductive logic programming. New Generation Comput. 8(4), 295–318 (1991)

[12] Muggleton, S.: Inverse entailment and progol. New Generation Computing 13(3), 245–286 (Dec 1995)

[13] Muggleton, S.H., Bryant, C.H.: Theory completion using inverse entailment. In: Cussens, J., Frisch, A. (eds.) ILP. pp. 130–146. Springer Berlin Heidelberg (2000)

[14] Plotkin, G.D.: A further note on inductive generalization, in machine intelligence, volume 6, pages 101-124 (1971)

[15] Quinlan, J.R.: Learning logical definitions from relations. Machine Learning 5, 239–266 (1990)

[16] Ray, O.: Nonmonotonic abductive inductive learning. Journal of Applied Logic 7(3), 329 – 340 (2009), special Issue: Abduction and Induction in AI

[17] Sakama, C.: Induction from answer sets in nonmonotonic logic programs. ACM Trans. Comput. Log. 6(2), 203–231 (2005)

[18] Sakama, C., Inoue, K.: Brave induction: a logical framework for learning from incomplete information. Machine Learning 76(1), 3–35 (2009)

[19] Seitzer, J.: Stable ilp : Exploring the added expressivity of negation in the background knowledge. In: IJCAI-97 Workshop on Frontiers of ILP (1997)

[20] Shakerin, F., Salazar, E., Gupta, G.: A new algorithm to automate inductive learning of default theories. TPLP 17(5-6), 1010–1026 (2017)

[21] Srinivasan, A.: The Aleph Manual (2001), http://web.comlab.ox.ac.uk/oucl/research/areas/machlearn/Aleph/

[22] Zeng, Q., Patel, J.M., Page, D.: Quickfoil: Scalable inductive logic programming. Proc. VLDB Endow. 8(3), 197–208 (Nov 2014)