A Bidirectional ILP Algorithm

Markus Wiese

Institute AIFB, University of Karlsruhe, 76128 Karlsruhe, Germany
e-mail: wiese@aifb.uni-karlsruhe.de

Abstract. The paper presents an approach for using a bidirectional search strategy for inductively learning clauses in a restricted first-order language. The learning target is to find a set of goal clauses that describes the true ground facts of the target predicate. In our example setting we further assume that the background knowledge is also given in the form of true (and false) ground facts for each background predicate. By fixing the number of variables allowed in the derived clauses we show that no explicit negative goal facts are needed in the case of the closed-world assumption since the rules are evaluated from *the premise* to *the head* rather than binding the variables of the goal literal first. As a consequence we get an efficient algorithm that tries to minimize the tuples of variable substitutions stored at each step of our covering approach.

Introduction

The paper presents an approach for using a bidirectional search strategy for inductively learning clauses in a restricted first-order language. Given goal facts and true (and false) ground facts as background knowledge, a finite set of *program clauses* [Llo87] is being derived that describes a target predicate. By excluding function symbols and fixing the number of variables allowed in one clause we restrict the hypothesis language to sets of first-order clauses that have a *finite* minimal (i.e., perfect) Herbrand model [Ull88]. This representation leads to a finite search space where it is possible to conduct a bidirectional search.

In the means of propositional logic a bidirectional search strategy is a well-known idea, originally presented in the version space algorithm by [Mit81]. This idea was adopted by the algorithms JoJo (cf. [Fen93a], [FeW93], [Wie93]) and Frog (cf. [Fen93b]) that integrate generalization and specialization refinement steps into one search procedure rather than keeping two seperate sets of most general and most specific concept descriptions. Also, JoJo and Frog perform a heuristic search contrary to the enumerated search of the version space algorithm. Since it is generally not possible to give preference to one of the both search directions ([Fen93a]), the bidirectional search might be more flexible in adapting to the unknown domain-specific characteristics of a real-world data set.

Most well-known inductive learning algorithms performing search in a first-order representation space are either "specific-to-general" or "general-to-specific" systems. Representatives of the class of "specific-to-general" include GOLEM (cf. [MuF90]), CLINT (cf. [Rae92]), MARVIN (cf. [SaB86]), DUCE (cf. [Mug87]) and CIGOL (cf. [MuB88]). On the other hand, learning algorithms like FOIL (cf. [Qui90], [Qui91], [QuC93], [CaQ93]), CLAUDIEN (cf. [RaB93]), MIS (cf.

[Sha83]), and the MOBAL system (cf. [KiW92]) start their search with the most general hypothesis and repeatedly specialize the hypothesis in order to discriminate from the negative examples, but still being consistent with the positive examples.

These "general-to-specific" algorithms all perform very well and especially fast when the induced clauses have a relative short premise. This is, for example, usually true when applying ILP systems as programming assistants. When trying to find valuable properties of a real-world domain it is generally not clear in what depth (that corresponds to the number of conditions in the premise of a rule) of the hypothesis space one should look for. We believe that starting the search with arbitrarily[1] generated rules and using a bidirectional search strategy might be very useful and efficient when dealing with huge amounts of data. By employing a very efficient method for evaluating the generated rules that also leads to avoiding the explicit representation of negative examples we believe that our bidirectional ILP algorithm is potentially applicable for data mining problems.

1 Description of our approach

To be able to perform a bidirectional search through the hypothesis space we have to make sure that the search does not get lost in an infinite branch of the search space. As our language bias we fix the number of variables that might possibly occur in our rules such that we get a finite hypothesis space. Suppose we have a k-ary goal predicate given as $G(X_1,...,X_k)$ and a number of background predicates B_i of different arity n_i given as $B_i(Y_1,...,Y_{ni})$ where the arguments of the relations have possibly different types $argtype_j$ with $1 \leq j \leq m$ and $m \geq 1$.

Since it is usually impossible to know the exact number of variables needed to produce a correct and sufficient rule set we start with the minimum number depending on the arity and the argument types of all predicates. If it is not possible to generate rules that cover enough positive goal facts (e.g. 90%) the number of variables is incremented by 1 and the rule search will restart. To calculate the minimum number of variables needed we determine for each (fore- and background) predicate the number of variables for each type that is necessary such that all argument positions have different variable names. Adding up the number of variables for each argument type gives the total number of variables needed. Having fixed the variable number we build all possible literals from the given predicates and the variables that are syntactically correct and obey semantically the argument types of each predicate. Additionally, some built-in equality literals between variables of the same type and between a variable and a theory constant of the same type may be added to the set of possible literals. Since one goal predicate may give rise to many literals containing the name of the goal predicate but different variable names we denote one of them as our *goal literal* and use the other ones as literals possibly occuring in the premises of the rules (and therefore resulting into recursive rules).

We are now able to describe our covering approach which we call JoJo-FOL since it inherits its specialization and generalization refinement operators from its propositional predecessor JoJo (cf. [Fen93a]). The core of JoJo-FOL consists of an outer and a inner loop that we will summarize in the following.

The outer loop is pretty much the same as in many other covering algorithms:

1. "Arbitrarily chosen" rather counts for our window set where some n% of the whole data is taken on which a rule search is performed as a first step. The generated rules describing the target predicate within the window set are used as starting rules for further search.

- Establish the positive facts of our goal predicate as the positive training examples.
- Until a predefined percentage of covered positive examples is not yet reached do:
  - Find a *correct* clause in the means that it covers a minimum of positive examples and potentially also some negative ones, but stays still within a predefined error threshold.
  - Remove all positive examples that satisfy the premise of the new clause.
- Try to generalize the rules by removing literals from the premise such that the error ratio stays within a specified threshold considering *all* covered positive examples (not just the remaining uncovered ones for the next rule finding iteration).
- Postprune the rule set by checking for logically equivalent rules and building a minimum cover in the sense that deleting one rule would cause at least one positive example to be no more covered by the remaining rule set.

The inner loop tries to find clauses of the following form:

$G(X_1,...,X_k) \leftarrow L_1, L_2, ..., L_n$ where each literal is an element of our set of literals.

We do not restrict our literals to be just positive. For each predicate (including the goal predicate) we might have been given negative examples either explicitly (e.g. in the case of an open-world assumption) or by applying the closed-world assumption to all unknown examples. The inner loop can now be described as follows:

- Choose a starting rule to begin the search with.
- Initalize the variable bindings of the occuring variables in the *premise*.
- **While** "preference criterion[2] can be improved"

  Do

    If the error ratio of covered positive and negative example exceeds a predefined threshold:
      For all elements of the literal set and not yet occuring in the premise
      Do
        Add the literal to the premise of the rule
        Calculate the new variable bindings (possibly with a new variable)
        Check the number of covered positive and negative examples by the new rule
        Save the best preference, literal and variable bindings
        Reset the variable bindings
      Enddo
      If "new preference is higher than old preference"
        Add best literal to the premise
        Substitute old variable bindings by new variable bindings
        continue while loop
      Else
        STOP! Discard rule and restart inner loop (with a new starting rule)!
      Endif
    Else
      For all literals occuring in the premise of the currently regarded rule
      Do
        Delete the literal from the premise of the rule
        Calculate the new variable bindings (possibly with fewer variables)

2. We use the accuracy (= #pos/(#pos+#neg)) of the rule as simple preference criterion. In case of equality we prefer the rule that covers more positive examples.

Check the number of covered positive and negative examples by the new rule
Save the best preference, literal and variable bindings
Restore the variable bindings
Enddo
If "new preference higher than old preference"
Remove best literal from the premise
Substitute old variable bindings by new variable bindings
continue while loop
Else
STOP! Save rule in the preliminary rule set!
Endif
Endif
Enddo

Choosing a good starting rule is certainly a crucial task for any heuristic bidirectional search algorithm. We propose the following iterative procedure. For a sufficient small subset (e.g. 10% or 1%) of the whole data set we test a fixed number of randomly generated starting rules[3] each time when entering the inner loop and choose the one with the highest preference. The rules generated for describing the goal facts within the *window* are the starting rules for the rule search with the complete data set.

Another very important issue concerns the evaluation of our rules. Contrarily to FOIL we begin binding the occuring variables to constant tuples from *the premise to the head*. That is, we first look for tuples that satisfy the right-hand side of the clause before we check if the corresponding facts (by regarding only the goal literal variable substitutions) denote a positive or a negative example of the goal predicate. Binding the variables of the premise first gives rise to the following advantages:

- Since we know all literals occuring in the premise of the rule we can *efficiently bind* the variables keeping the number of stored substitution tuples as small as possible For binding the variables we first choose the literal of the premise that covers the smallest number of ground facts (in the most cases this literal will be positive, but it can also be a negative one). The next variables that will be bound are those from the literal that covers the second least ground facts and so on.
- It is possible to have *indeterminated* literals on the premise of the rule that introduce new variables without exploding the size of the substitution tuples.
- *Negated* literals can also be included in the premise without exploding the size of the substitution tuples. Since the negative literals cover in most cases more ground facts (by assuming the closed world) the variables of the positive literals will be bound first (see above). Therefore, it is very likely that all variables of the negative literals are already bound before the first negative literal will be evaluated. As a consequence the evaluation of negative literals restricts to just checking if the corresponding fact satisfies the predicate or not.
- As soon as we get an **empty** set of substitutions "satisfying" the variables of the literals we have checked so far, we can *prune* all the possible rules that contain these literals in their premise.
- If we allow only positive literals in our premise and the closed-world assumption applies

3. A starting rule has to fulfill some "useful" requirements, e. g. its premise must at least contain one variable of the goal literal!

to our goal predicate and the background predicates we *avoid representing explicit negative examples*.

When the variables of all literals occuring in the premise are bound and the set of substitutions is not empty we have to distinguish between the following three cases:

- *No* variable of the goal literal is bound: This case is actually being checked before the variables of the premise are bound since it represents the case of a stupid premise. Therefore, all literals are being removed from the premise and the rule is generalized to the most general rule from which the search will continue with a specialization step if the unit goal clause is incorrect.

- *All* variables of the goal literal are bound: In this case it is easy to check if the corresponding ground fact belongs to the goal predicate or not. In the case of the closed-world assumption we avoid explicit negative goal facts when we forbid recursive rules with a negative goal literal.

- The variables of the goal literal are *partially* bound: We have to look if the goal facts match the substitutions in the bound variables of the goal literal. In this case we are calculating the number of all possible facts that match these bound variables and get the number of covered negative examples by subtracting the number of positive matched goal facts from the total number. Again, no negative examples have to be represented explicitly when making the above mentioned assumptions.

Before we illustrate our approach on a small example we want to point out that our simply chosen preference criterion depends only on the covered and not covered goal *facts* and not on the *substitutions*. Two or more substitutions may correspond to the same goal fact. This is also a difference to the FOIL algorithm that calculates its preference (information gain) criterion from the positive and negative *substitution tuples* instead of the corresponding positive and negative goal *facts*. Though calculating the number of covered facts means an extra computational effort we believe that we get a benefit from it for our covering approach. Since the positive facts are removed after each rule finding iteration the whole search process may terminate faster when prefering rules with a higher accuracy calculated from the covered facts over rules that have a lower "fact accuracy" but may have a better accuracy calculated from the covered substitutions.

2 An example

We use the same data [QuC93] used to learn the member relationship. Given are the goal predicate **member(E,L)** and one background predicate **components(L,E,L)** that have both the two argument types E for Element and L for List. For these argument types the following constants are given:

E: {1,2,3} and
L: {[111], [112], [113], [11], [121], [122], [123], [12], [131], [132], [133], [13], [1], [211], [212], [213], [21], [221], [222], [223], [22], [231], [232], [233], [23], [2], [311], [312], [313], [31], [321], [322], [323], [32], [331], [332], [333], [33], [3], *[][4]}

4. The asterix before the empty list denotes a theory constant that can appear in the premise of the rule (see [QuC93]).

For predicate member the negative tuples are given explicitely whereas for predicate components only positive facts are presented and we therefore apply the closed-world assumption:

member^{+}: {1,[1]; 3,[3]; 1,[11]; 1,[13]; 3,[13]; 1,[31]; 3,[31]; 3,[33]; 1,[111]; 1,[113]; 3,[113]; 1,[131]; 3,[131]; 1,[133]; 3,[133]; 1,[311]; 3,[311]; 1,[313]; 3,[313]; 1,[331]; 3,[331]; 3,[333]}

member^{-}: {1,[]; 1,[3]; 1,[33]; 1,[333]; 3,[]; 3,[1]; 3,[11]; 3,[111]; 1,[2]; 1,[22]; 1,[23]; 1,[32]; 1,[222]; 1,[223]; 1,[232]; 1,[233]; 1,[322]; 1,[323]; 1,[332]}

components^{+}: {[1],1,[]; [2],2,[]; [3],3,[]; [11],1,[1]; [12],1,[2]; [13],1,[3]; [21],2,[1]; [22],2,[2]; [23],2,[3]; [31],3,[1]; [32],3,[2]; [33],3,[3]; [111],1,[11]; [112],1,[12]; [113],1,[13]; [121],1,[21]; [122],1,[22]; [123],1,[23]; [131],1,[31]; [132],1,[32]; [133],1,[33]; [211],2,[11]; [212],2,[12]; [213],2,[13]; [221],2,[21]; [222],2,[22]; [223],2,[23]; [231],2,[31]; [232],2,[32]; [233],2,[33]; [311],3,[11]; [312],3,[12]; [313],3,[13]; [321],3,[21]; [322],3,[22]; [323],3,[23]; [331],3,[31]; [332],3,[32]; [333],3,[33]}

Since member needs a least one variable for both types E and L, but components needs two variables of type L and one of type E the minimum number of variables needed is calculated to be three. We denote the two variables of type L with X and Y and the single variable of type E with Z.

Therefore, we get our set S of possible literals: S={components(X,Z,Y); components(Y,Z,X); member(Z,X); member(Z,Y)}. Arbitrarily, we choose the literal *member(Z,X)* to be our goal literal.

Suppose now that the following rule was generated randomly:

member(Z,X) ← components(X,Z,Y) & member(Z,Y)

Since components contains 40 positive tuples and member just 22, we start our evaluation of the premise with literal member(Z,Y) and get the following tuples <Z,Y>:

{<1,[1]>; <3,[3]>; <1,[11]>; <1,[13]>; <3,[13]>; <1,[31]>; <3,[31]>; <3,[33]>; <1,[111]>; <1,[113]>; <3,[113]>; <1,[131]>; <3,[131]>; <1,[133]>; <3,[133]>; <1,[311]>; <3,[311]>; <1,[313]>; <3,[313]>; <1,[331]>; <3,[331]>; <3,[333]>}

When binding the variables of the next literal components(X,Z,Y) we just have to check which of the already bound <Z,Y> tuples correspond to positive facts of the predicate components. We get the following 8 tuples given in <X,Z,Y> order:

{<[11],1,[1]>; <[33],3,[3]>; <[111],1,[1]>; <[113],1,[13]>; <[313],3,[13]>; <[131],1,[31]>; <[331],3,[31]>; <[333],3[33]>}

Evaluating the corresponding <Z,X> facts we see that they are all different and belong to the positive facts of predicate member. Therefore, the rule covers 8 positive examples and no negative examples and the next step will be trying to generalize the rule.

There exist two possibilities for generalizing:

a) dropping literal components(X,Z,Y):

   This leads to rule: *member(Z,X) ← member(Z,Y)* and the same 22 <Z,Y> tuples as after evaluating the first literal of our starting rule. But in this case, only variable Z is bound of the goal literal. So, when determing the number of covered goal facts, we have to match all positive and negative[5] goal facts that have the bound <Z> tuples = {<1>; <3>} as first argument value. As result, the rule covers 22 positive and 19 negative facts. Since the preference given to this more general rule is less than the preference of the more special rule it is quashed.

b) dropping literal member(Z,Y):

We get rule: *member(Z,X) ← components(X,Z,Y)*. In this case we get the 40 <X,Z,Y> tuples given as positive facts for predicate components. Checking the goal facts of predicate member the number divides into 14 positive examples and 26 unknown cases. Therefore, this rule (14+/0-) is of higher preference than the starting rule (8+/0-) and the generalization will be performed.

Trying to further generalize the rule *member(Z,X) ← components(X,Z,Y)* is not successful since this would lead to the most general rule that covers all given positive (22) and negative (19) goal facts having a lower preference than the currently found rule. At this point, the inner loop terminates and returns the first correct rule ***member(Z,X) ← components(X,Z,Y)***.

After removing the 14 examples from the list of positive examples still 8 remain uncovered such that the outer loop calls the inner loop again. But this time, no other correct rule can be found since allowing just 3 variables was to restrictive. Therefore, we start our search for rules again by allowing this time 4 variables to occur in the rules. When incrementing the variable number the algorithm first has to decide which argument type to assign to the variable. Preference is given to the type that least increases the hypothesis space. In our example we have to decide between type E and type L:

a) a new variable W of type E gives rise to 4 more literals: {components(X,W,Y), components(Y,W,X), member(W,X), member(W,Y)}
b) a new variable W of type L results into 5 new literals: {components(X,Z,W), components(W,Z,X), components(Y,Z,W), components(W,Z,Y), member(Z,W)}

The new variable W is therefore assigned to type E. Performing the search again with 4 variables we find the standard definition for member

member(Z,X) ← components(X,Z,Y)

member(Z,X) ← components(X,W,Y) & member(Z,Y)

that covers all given positive goal facts.

3 Some first evaluations of Our Approach

We compared our approach with FOIL version 6.3 using 3 small example sets from [Qui90] and [QuC93] and the larger Finite Element Mesh Design data provided by Bojan Dolsak. All evaluations were run on a Sparc 10 Unix workstation.

3.1 A Small Network

Characterisation:

5. If no explicit negative facts were given we would apply the closed-world assumption and calculate the number of covered negative facts by assigning to it the difference between the maximum number and the covered positive facts. (In this case 2*40-22=58.)

- argument types: Node (9 constants)
- goal predicate: can-reach(Node,Node)
- background predicates: linked-to(Node,Node)
- positive examples: 19 negative examples: closed-world (62)

Learned Rules:

FOIL: can-reach(A,B) :– linked-to(A,B) and
can-reach(A,B) :– linked-to(A,C), can-reach(C,B)

JOJO-FOL: The same rules when fixing the variable number to 3.

Runtime:

Both algorithms took 0.0 seconds to derive the rules.

3.2 Learning the Definition for Member

Characterisation:

- argument types: Element (3 constants)
  List (40 elements)
- goal predicate: member(Element,List)
- background predicates: components(List,Element,List)
- positive examples: 22 negative examples: explicit 19

Learned Rules:

FOIL: member(A,B) :– components(B,A,C) and
member(A,B) :– components(B,C,D), member(A,D)

JOJO-FOL came up with the same rules when fixing the variable number to 4.

Runtime:

Both algorithms took 0.1 seconds to derive the rules.

3.3 Learning Definitions For Arches

Characterisation:

- argument types: Thing (12 constants)
- goal predicate: arch(Thing,Thing,Thing)
- background predicates: supports(Thing,Thing); left-of(Thing,Thing);
  touches(Thing,Thing); brick(Thing); wedge(Thing)
  parallelpiped(Thing)
- positive examples: 2 negative examples: closed-world (1726)

Learned Rules:

FOIL: arch(A,B,C)) :– left-of(B,C), supports(B,A), not(touches(B,C))

JOJO-FOL: arch(X,Y,Z)) <– left-of(Y,Z) & supports(Z,X) & not(touches(Y,Z))

Both rules are correct and only differ in the supporting thing.

Runtime:
Both algorithms took 0.4 seconds to derive the rules.

3.4 Finite Element Mesh Design

Characterisation:

- argument types: Edge (506 constants: a1,...,a55; b1,...,b42; c1,...,c28; d1,...,d57;e1,...,e96; f1,...,f41; g1,...,g60; h1,...,h71; i1,...i26; j1,...j30)
  Number (12 constants: 1, ..., 12)
- goal predicate: mesh(Edge,Number)
- background predicates: long(E); usual(E); short(E); circuit(E); half-circuit(E); quarter-circuit(E); short_for_hole(E); long_for_hole(E); circuit_hole(E); half_circuit_hole(E); not_important(E); free(E); one_side_fixed(E); two_side_fixed(E); fixed(E); not_loaded(E); one_side_loaded(E); two_side_loaded(E); cont_loaded(E); neighbour(E,E); opposite(E,E)

- positive examples: 629 negative examples: closed-world (5443)

Learned Rules:

FOIL:mesh(A,1) :- not_important(A), free(A).
mesh(A,1) :- not_important(A), one_side_loaded(A)
mesh(A,1) :- short(A), fixed(A), one_side_loaded(A).
mesh(A,1) :- short(A), cont_loaded(A), free(A), neighbour(A,C), usual(C).
mesh(A,1) :- short(A), not_loaded(A), neighbour(A,C), usual(C), one_side_loaded(C).
mesh(A,1) :- not_important(A), not_loaded(A), neighbour(A,C), opposite(C,D), not_loaded(D).
mesh(A,1) :- short(A), neighbour(A,C), usual(C), one_side_fixed(C), neighbour(A,D), opposite(D,E), fixed(D).
mesh(A,2) :- usual(A), two_side_loaded(A).
mesh(A,2) :- usual(A), fixed(A), one_side_loaded(A).
mesh(A,2) :- short(A), opposite(A,C), not_important(C).
mesh(A,2) :- usual(A), opposite(A,C), opposite(C,D), half_circuit(D).
mesh(A,2) :- short(A), not_loaded(A), neighbour(A,C), cont_loaded(C).
mesh(A,2) :- free(A), opposite(A,C), usual(C), neighbour(A,D), short(D).
mesh(A,2) :- short(A), free(A), not_loaded(A), neighbour(A,C), one_side_fixed(C).
mesh(A,2) :- neighbour(A,C), not_important(C), opposite(C,D), not_important(D), neighbour(A,D).
mesh(A,2) :- fixed(A), short(A), cont_loaded(A), neighbour(A,C), opposite(C,D), usual(C).
mesh(A,2) :- usual(A), one_side_fixed(A), neighbour(A,C), opposite(C,D), free(C), not_loaded(D).
mesh(A,4) :- one_side_loaded(A), one_side_fixed(A), usual(A).
mesh(A,8) :- not_loaded(A), half_circuit(A), neighbour(A,C), opposite(C,D), long(D).
mesh(A,9) :- circuit(A), opposite(A,C).

```
mesh(A,9) :-   two_side_fixed(A), quarter_circuit(A).
mesh(A,9) :-   not_loaded(A), half_circuit(A), neighbour(A,C), opposite(C,D),
               long(D).
mesh(A,10) :- one_side_loaded(A), long(A).
mesh(A,11) :- circuit(A), not_loaded(A), free(A), neighbour(A,C),not_important(C).
mesh(A,12) :- circuit(A), not_loaded(A), free(A), neighbour(A,C),not_important(C).
```

```
JOJO-FOL:   mesh(X,1) <- not_important(X)
            mesh(X,2) <- cont_loaded(X) & long_for_hole(X)
            mesh(X,2) <- free(Y) & opposite(X,Y) & short(X)
            mesh(X,2) <- long_for_hole(X) & one_side_fixed(X)
            mesh(X,9) <- quarter_circuit(X) & two_side_fixed(X)
            mesh(X,10) <- one_side_fixed(X) & quarter_circuit(X)
            mesh(X,11) <- circuit(X) & one_side_loaded(X)
            mesh(X,12) <- circuit(X) & one_side_loaded(X)
```

Runtime:

FOIL took 54.9 seconds to come up with 25 rules.

JOJO-FOL needed 43.0 seconds to produce 8 rules.

Both algorithms only produced rules of 80% accuracy or higher. This might be the reason why both algorithms could not find rules for every number of finite elements. In the case of JOJO-FOL the additional requirement was made that each rule must cover at least 3 positive examples. So even no useful rules were found for 4 and 8 finite elements. Also, we allowed JOJO-FOL to use 5 variables (3 of type Edge and 2 of type Number[6]), but actually only three are occuring in the rule set (the variable of type Number has been replaced by the constant it is assigned to). Negative literals were excluded from the rule premises. Comparing the two generated rule sets it is surprising that JOJO-FOL produces in many cases more general rules than FOIL though it performs a bidirectional search and not a general-to-specific one like FOIL. The reason for that result can be assumed to come from the hill-climbing strategy of FOIL that always prefers the literal with the highest information gain (except determinate literals) to be added to the premise. Since the resulting rules (in this case) are always correct no other alternative literals are being checked in a backtracking procedure that might lead to shorter but also correct rules. Except of the first rule, JOJO-FOL also generated only rules that are correct in the sense that they do not cover any negative example! The first rule has been accepted by JOJO-FOL because it covers 54 positive facts and just 3 negative ones achieving an accuracy of 94.7%. The reason for the comparable small number of rules produced by JOJO-FOL may also be another stop-criterion we implemented in our system. The outer loop terminates if in a number of consecutive tries no acceptable rule has been found. (This parameter is set to 10 by default, but may be changed by the user.) We know that we have to judge this first result on the mesh design data very carefully since we have spend no effort so far on trying to optimize the parameter settings neither of FOIL nor of JOJO-FOL. We just used the default settings except excluding the negative literals from the premises. But it was our intention to show that JOJO-FOL can handle larger data sets and is capable to generate useful rules with its bidirectional search strategy. Further evaluation on the MESH data as well as on other domains like the protein secondary structure, drug design or mutagenicity is necessary to allow a more

6. Of course, only one variable of type Number is needed, but our approach adds first a variable of type Number before it adds another variable of type Edge. A third variable of type Edge was thought to be important for deriving useful rules.

significant predication on the comparison between both approaches.

4 Conclusion and Future Work

We presented an algorithm that learns single predicate definitions by using a bidirectional search strategy. To be able to conduct a bidirectional search through a first-order hypothesis space we restrict the space to be finite by excluding function symbols and fixing the number of variables to syntactically limit the set of possible literals. Evaluating the rules from the premise to the head allows a very efficient way to bind variables to constant tuples keeping the number of stored substitutions as small as possible. Further on, if we do not allow negative goal literals in our premise no explicit negative examples are needed under the closed-world assumption. Avoiding negative examples gives our approach the possibility to handle larger amount of data more efficiently. Some first small evaluations of our approach seem to support our intention to have build an ILP algorithm that is potentially useful for data mining because of its efficient rule evaluation and its sparingly used storage space by avoiding the explicit representation of negative examples.

Future work mainly concentrates on a thorough and extensive evaluation of our approach, but there are still some other directions that need to be mentioned. Most urgent, we have to adopt some methods to prevent infinite recursive clauses. In some test runs when allowing equality build-in predicates we got such useless clauses as mesh(X,Y) <- mesh (X,Z) & =(Y,Z). We are considering methods similar to the ordering of recursive literals by [CaQ93]. Another challenging task will be to find better heuristics for estimating the number of variables needed and how they distribute to possibly more than one type. The system should automate the process of inventing a new variable of the most promising type when it sees no more chance to derive useful rules with the given set of literals. Of interest is also to extend and/or improve the refinement operators of our bidirectional heuristic search strategy since it is very myopia to consider always just one literal to be added or deleted. A simulated annealing strategy like already employed to Frog [Fen93b] may also be of use to overcome unimportant local optima. Finally, we can imagine to extend the window method to an iterative "data mining method". After dividing the huge amount of real-world data into some n disjunctive data sets we start with one of these sets to produce the first clauses. In a second iteration we use another data segment together with the rules found in the first iteration to start a new search with. This iterative process continues until all data has been input to the system.

Acknowledgements

I want to thank Dieter Fensel, Robert Engels and Guido Lindner for giving me valuable input for the ideas presented in this paper and Bojan Dolsak from the Mechanical Engineering Department of the Faculty of Technical Sciences in Maribor, Slovenia, for providing the Finite Element Mesh Design data.

References

[CaQ93] R. M. Cameron-Jones and J. R. Quinlan: When Learning Recursive Theories. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence,* Morgan Kaufmann, 1993.

[Fen93a] D. Fensel: JoJo: Integration of Generalization and Specialization. In *Proceedings of the Workshop Knowledge and Data Engineering*, Atelier d´Ingenierie des Connaissances et des Donees, A.I.C.D., Strasbourg, France, January 25-27, 1993.

[Fen93b] D. Fensel: RELAX, JoJo, and Frog: Step by Step Generalization of Search Strategies in Applied Machine Learning. In research report, no 279, Institut AIFB, University of Karlsruhe, 1993.

[FeW93] D. Fensel und M. Wiese: Incremental Refinement of Rule Sets with JoJo. In *Proceedings of the European Conference on Machine Learing ECML-93*, Vienna, Austria, April 5-8, 1993, Lecture Notes in Artificial Intelligence, no 667, Springer-Verlag, Berlin, 1993.

[KiW92] J.-U. Kietz and S. Wrobel: Controlling the Complexity of Learning in Logic Through Syntactic and Task-Oriented Models. In *S. Muggelton (ed.), Inductive Logic Programming*, Academic Press, 1992.

[Llo87] J. W. Lloyd: *Foundations of Logic Programming*, Springer-Verlag, 2nd edition, Berlin, 1987.

[Mit81] T.M. Mitchell: Generalization as Search, B. Webber et al. (eds). In *Readings in Artificial Intelligence*, Tioga Publishinh Co., Palo Alto, 1981.

[MuB88] S. Muggleton and W. Buntine: Machine Invention of First-Order Predicate by Inverting Resolution. In *Proceedings of the 5th International Conference on Machine Learning (ICML´88)*, 1988.

[MuF90] S. Muggleton and C. Feng: Efficient Induction of Logic Programs. In *Proceedings of the Workshop on Algorithmic Learning Theory (ALT´90)*, Tokyo, October 8-10, 1990.

[Mug87] S. Muggelton: Duce, An Oracle Based Approach to Constructive Induction. In *Proceedings of the 10th International Joint Conference on Artificial Intelligence*, Morgan Kaufmann, 1987, pp. 287 - 292.

[QuC93] J. R. Quinlan and R. M. Cameron-Jones: FOIL: A Midterm Report. In *Proceedings of the European Conference on Machine Learning, Machine Learning: ECML-93*, Vienna, Austria, April 5-7, 1993, P. B. Brazdil (ed.), Springer-Verlag, Lecture Notes in Artificial Intelligence, no 667, 1993.

[Qui91] J. R. Quinlan: Determinate Literals in Inductive Logic Programming. In *Proceedings of the 12th International Joint Conference on Artificial Intelligence*, Sydney, 1991, pp. 746 - 750.

[Qui90] J. R. Quinlan: Learning Logical Definitions from Relations. In *Machine Learning*, vol 5, no 3, 1990, pp. 239-266.

[RaB93] L. De Raedt and M. Bruynooghe: A Theory of Clausal Discovery. In *Proceedings of the 13th International Joint Conference on Aritificial Intelligence (IJCAI´93)*, Chambery, France, 28 August - 3 September, 1993.

[Rae92] L. De Raedt: *Interactive Theory Revision: An Inductive Logic Programming Approach*, Academic Press, 1992.

[SaB86] C. Sammut and R. B. Banerji: Learning Concepts by Asking Questions. In *Machine Learning: An Artificial Approach, vol 2*, Morgan Kaufmann, Los Altos, 1986, pp. 167 - 192.

[Sha83] E. Y. Shapiro: *Algorithmic Program Debugging*, MIT Press, 1983.

[Ull88] J.D. Ullman: *Database and Knowledge-Base Systems, vol. 1*, Computer Science Press, New York, 1988.

[Wie93] M. Wiese: JoJo: *Integration von Generalisierung und Spezialisierung in ein heuristisches Verfahren zum maschinellen Lernen von Regeln aus Beispielen*, master thesis, Institut AIFB, University of Karlsruhe, 1993.