Machine Learning, 14, 233-247 (1994)

Explicit Representation of Concept Negation

JEAN-FRANCOIS PUGET PUGET@ILOG.FR
ILOG SA, 2 avenue Galliéni, BP 85, 94253 Gentilly, FRANCE

Abstract. We present a learning method called Negative Explanation Based Generalization (NEBG) that performs automatic changes of representation by computing the negation of an already known concept. NEBG is similar to EBG as a deductive and valid learning method using a single example. It is based on new logic programming techniques based on example-guided transformation of the completed database. We also introduce a very powerful heuristic based on functional properties of the application domain. The implemented algorithms are described and several examples are given.

Keywords. EBL, negation, logic programming

1. Introduction

1.1. Representation changes

Machine learning considered as a study of automatic representation changes has received increasing interest over the past few years. Some work aims at a better usability of knowledge, especially in problem-solving areas (see, for example, Amarel and Explanation Based Learning). We are interested in this kind of work where the goal is not to increase knowledge, but to express it in a more efficient form.

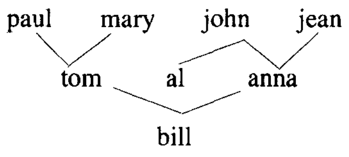
While working on a learning planning system, we found that some negative information had to be explicitly stated. This led to the design of a generic method for representing negation of already known concepts. This method, called NEBG, starts from a body of initial knowledge K represented as a logic program (a set of definite clauses) and a negative example NE for one of the concepts C defined in K. NEBG outputs a definition of not(C), which covers NE.

For instance, let us consider a simple family example. The theory defines the grandfather relationship:

```
grand-father (X, Y) ← father (X, Z) ∧ father (Z, Y)
grand-father (X, Y) ← father (X, W) ∧ mother (W, Y)
```

The problem in which we are interested is to produce an explicit representation of "not-grandfather." The example is the following family:

```
father (paul, tom) ∧
mother (mary, tom) ∧
father (tom, bill) ∧
mother (anna, bill) ∧
father (john, anna) ∧
mother (jean, anna) ∧
father (john, al)
```



The negative example is

```
grandfather (tom, bill).
```

1.2. A false solution

A simple "solution" has been proposed in the past (Hirsh, 1987a), which seems to elegantly extends EBG. The idea is to try to prove the goal concept and, if there is a failure, to use the failed proof in a similar way as in EBG and keep all the successful subgoals plus the failed one. The failed proof can be generalized as in EBG by removing all the parts that are below an operational subgoal and by performing only the unifications made in the remainder of the proof. If we use the preceding method, we would obtain the clause

```
not (grand-father (X, Y)) ← father (X, Z) ∧ not (father (Z, Y))
```

since the first subgoal succeeds, and the second one fails. However, this result is false, since we can now derive `not (grand-father (john, bill))` by substituting `X` by `john`, `Y` by `bill`, and `Z` by `anna` in the learned clause, and we can derive `grand-father (john, bill)` using the second clause of the theory.

The problem comes from the fact that we have not used *all* the possible proofs to construct the generalization of the failure. In our example, we also have to explain why the second clause of the theory cannot be used to prove `grandfather (tom, bill)`. The above method can be enhanced to take all the failed proofs into account: the result is then constructed by taking the conjunction of all the successful leaves plus the failed leaf of all the failed proofs. In the example, this gives the clause

```
not (grand-father (X, Y)) ← father (X, Z)
∧ not (father (Z, Y)) ∧ father (X, W) ∧ not (mother (W, Y))          (C)
```

However, this result is again false! Indeed, we can obtain `not (grand-father (john, bill))` by substituting `X` by `john`, `Y` by `bill`, and both `Z` and `W` by `al` in the learned clause C, which contradicts the fact that `grand-father (john, bill)` can be derived using the second clause of the theory. The problem comes from a misinterpretation of the quantifications of the variables in the clause C, as we will see in the next section.

The remainder of this article is organized as follows. Section 2 describes the logical formalism used, especially Clark's completed program. Section 3 then describes the NEBG algorithm, which transforms this completed program into a definition of the negation of a concept. We conclude with a discussion of NEBG with respect to its potential use.

2. Logical formalism

We will use as a representation language a subset of first-order logic, namely, clauses (as in Prolog). We recall here the basic definitions used in Logic Programming, and refer to Lloyd (1987) for definitions of constants, functions, terms, predicates, atoms, and clauses. A definite clause is noted $P \leftarrow B_1 \wedge \ldots \wedge B_m$. P is the head of the clause, and the conjunction $B_1 \wedge \ldots \wedge B_m$ is the body of the clause. A logic program is a set of definite clauses.

The deduction procedure used is SLD resolution, as in Prolog. We note A |- B when B can be obtained from A using SLD resolution.

2.1. Generality, concepts and examples

In this formalism, a concept C is defined by a predicate p. The definition of the concept is the set of clauses having p as their head predicate. The body of the clause represents preconditions for concept membership. The generality relation used is provability: A is more general by B if B can be proved from A, which is noted A |- B.

An example for concept p(xl, . . . , xn) is a ground clause $p(tl, \ldots, tn) \leftarrow B_1 \wedge \ldots \wedge B_m$, where ti are ground terms. Given a set of clauses K, an example clause $p(tl, \ldots, tn) \leftarrow B_1 \wedge \ldots \wedge B_m$ is an instance of predicate p(xl, . . . , xn) iff

$$K, B_1 \wedge \ldots \wedge B_m \mid\!- p(t1, \ldots, tn)$$

An example clause $p(tl, \ldots, tn) \leftarrow B_1 \wedge \ldots \wedge B_m$ is a *negative* example of predicate p iff

$$K, B_1 \wedge \ldots \wedge B_m \mid\!-\!/\!- p(t1, \ldots, tn)$$

which means that the SLD resolution fails to prove the goal p(tl, . . . , tn). Two different cases can happen: either the SLD resolution loops endlessly, or all possible proofs fail in finite time. In the latter case, F is said to be a finite failure for the program T (Lloyd, 1987). We will restrict NEBG to learn only from finite failures. Thus, we start with a ground atom p(tl, . . . , tn) such that all proofs fail. NEBG is based on the completed database introduced by Clark (1978). This notion arises naturally when trying to explain failures as we now present them.

2.2. The completed database

The first step is to see that a safe generalization can be made only if *every* possible proof can be explained to fail. For instance, in a membership example, NE is the clause

```
member (1, [0, 2]) ←
```

and K is

```
member (A, [A | B]) ←
member (A, [B | C]) ← member (A, C).
```

We must explain why the first clause of K cannot be used to prove `member (1, [0, 2])`. The reason is simple: `member (X, [X | Y])` and `member (1, [0, 2])` cannot be unified. Thus we must be able to explain a unification failure. A simple way to do this is to explicate the unification that is performed at the head of the clause by introducing new variables:

```
∀ X, Y member (X, Y) ← (∃ A, B   X=A ∧ Y=[A | B])
```

This is called the homogeneous form of the clause in Clark (1978). The important property of the homogeneous form is that no unification failure can happen now, since the head of the clause is as general as possible. This transformation is carried over all the clauses:

```
∀  X, Y member (X, Y) ←
      (∃ A', B', C', X=A' ∧ Y=[B'|C'] ∧ member (A', C') )
```

The last step of the transformation is to say that *all* the possible proofs must be tested to be sure that there is a failure. In other words, there is a success only if one of the clauses can be used. This is expressed by the following formula (↔ means if and only if)

```
∀ X, Y      member (X, Y) ↔
         ( (∃A, B, X=A ∧ Y=[A|B] ) ∨
            (∃A', B', C', X=A' ∧ Y=[B'|C'] ∧ member (A', C')))          (1)
```

The process we have described is carried out for all predicates in the program. The resulting set of formulas is Clark's Completed DataBase (**CDB**). It has been proved (Jaffar et al., 1983; Clark, 1978) that all finite failures are a logical consequence of the CDB. In our example, this amounts to saying that `¬member(1, [0, 2])` follows from (1).

We have already done part of the work: we have obtained a logical justification of the failure. Equivalently, we have built a theory of the failures of member, which is given by the formula 1. We can have a better form of this theory by negating both sides of formula (1) (¬A means not A) and introducing a predicate `not_member`, yielding

```
∀X, Y      not_member (X, Y) ↔
           ( (∀ A, B, X≠A ∨ Y≠[A|B] ) ∧
               (∀A', B', C', X≠A' ∨ Y≠[B'|C'] ∨ not_member (A', C')))
```

And we have that `not_member(1, [0, 2]` is a logical consequence of formula (2).

This definition of `not_member` is not very satisfactory because it is not made of definite clauses, mainly because of universally quantified variables in the right side. These variables are called **parameters**, and the othe variables that appear in the left part are called **unknowns**. In formula (2), the unknown are `X, Y`, and the parameters are `A, B, A', B', C'`.

3. NEBG algorithm

3.1. Basic transformations

The basis of NEBG is to use a set of rewrite rules in order to transform a definition into a better one by removing parameters. The applications of the rules are guided by the input example. The idea is to keep a substitution σ of the unknowns called the *controlling substitution*. Initially, σ is the substitution of the unknown in the example.

For instance, in the member example, we start with formula (2) above. The example is `not_member(1, [0, 2]);` thus the controlling substitution σ is `{X/1 Y/[0, 2]}`. Formula (2) will be transformed by eliminating the parameters `A, B, A', B', C'`.

A first rule is very useful to eliminate a lot of parameters. We give it below. In the remainder of this article, the latters P, Q, and R represent arbitrary formulas when used in rule definitions.

Elimination of parameter

`∀Y, Y≠t ∨ P → P{Y/t}` (EP)

This rule is applied as soon and as much as possible.

The rule (EP) can be applied in order to eliminate the parameter `A`. In formula (2) `A` is replaced by `X`, and `A≠X` disappears, yielding the new definition of not_member:

```
∀ X, Y not_member (X, Y) ← (∀ B, Y≠[X | B]) ∧
                     (∀ A', B', C', X≠A' ∨ Y≠[B'|C'] ∨ not_member (A', C'))
```

Another application of (EP) eliminates `A'`, which is replaced by `X`:

```
∀ X, Y not_member (X, Y) ← (∀ B, Y≠[X | B]) ∧
                                   (∀ B', C', Y≠[B'|C'] ∨ not_member (X', C'))
```

Nothing more can be done with (EP), since there is no expression of the form ∀Y, $Y \neq T$ in the definition. We will use another rule that amounts to the instantiation of an unknown X.

Explosion

$X \neq t \vee R \rightarrow R\ \{X\ /\ f(X1,\ \ldots,\ Xn)\}$

X1, . . . , Xn are new unknown (EX)

Explosion is used if the following conditions hold:

- X is an unknown,
- No other rule can be applied,
- X $\in$ dom(σ), where σ is the controlling substitution. In that case, f is the principal function of Xσ were σ is the controlling substitution.

In that case the mgu of f(X1, . . . , Xn) and Xσ is added to the controlling substitution σ.

The controlling substitution σ is {X/1, Y/[0, 2]}; thus Y $\in$ dom(σ), and (EX) can be applied to replace Y by [Z|W]. Moreover, [Z|W] is unified with [0, 2], and the resulting mgu {Z/0, W/[2]} is added to the controlling substitution, which becomes {X/1, Z/0, W/[2]}. The new definition is

```
∀ X, Z, W not_member (X, [Z|W]) ← (∀ B, [Z|W]≠[X|B]) ∧
                                   (∀ B', C', [Z|W]≠[B'|C'] ∨ not_member (X, C'))
```

Then we need rules to simplify conditions of the form t1 $\neq$ t2, when both t1 and t2 are not parameters. These rules have no preconditions and are used as soon as possible (the order in which they are applied does not matter).

Decomposition

$f(t1,\ \ldots,\ tn) \neq f(u1,\ \ldots,\ un) \rightarrow t1 \neq u1 \vee \ldots \vee tn \neq un$ (D)

Clash

$f(t1,\ \ldots,\ tn) \neq g(u1,\ \ldots,\ um) \rightarrow$ true *iff* $f \neq g$ *or* $n \neq m$. (C)

Occur check

$X \neq t \rightarrow$ true *if the variable X appears in t.* (OC)

Simplification

$(\text{true} \vee R) \wedge P \rightarrow P$ (S)

(D) is applied on [Z|W] $\neq$ [X|B] and also on [Z|W] $\neq$ [B'|C'], yielding

```
∀ X, Z, W not_member (X, [Z|W]) ← (∀ B, Z≠X ∨ W≠B) ∧
                                   (∀ B', C', Z≠B' ∨ W≠C' ∨ not_member (X, C'))
```

(EP) can now be applied three times to eliminate `B, C', C'`, giving the definition

```
∀ X, Z, W not_member (X, [Z|W]) ← Z≠X ∧ not_member (X, W)
```

Then the algorithm stops with this clause, since no rule can be applied.

Besides the rules we have given, another one may be used. It is called reduction, and consist in dropping negative literals that are not necessary. We will not describe its use here and refer to Puget (1989) for a detailed discussion of it.

Reduction

$(L \vee R) \rightarrow R$ (R)

Reduction is used if the following conditions hold:

- L is a negative literal (either a literal not_A, or a difference $s \neq t$)
- L is a finite failure

The EBGF algorithm described in Siqueira et al., (1988) uses a similar rule called the rightmost heuristic. In fact, EBGF is a combination of some very simple manipulations of the completed database and the reduction rule (R).

We have proved that all these rules, as well as the ones presented below, are correct (Puget, 1989a, 1990): the result D' of the application of a rule to a definition D is a logical consequence of D. Moreover, no other rule is needed: all the generalizations of the example can be computed. Last but not least, we have also proved that the algorithm always stops: there can be no loops. The termination proof is based on the fact that each rule simplifies the formula: either the number of parameters decreases, or the height of at least one term decreases, or one literal is removed.

3.2. Computing the negation of a concept

The process we have presented is a way to obtain a clause that partially defines the negation of a known predicate. For instance, in the previous section we have learned a clause for not_member

```
not_member (X, [Z|W]) ← Z≠X ∧ not_member (X,W)          (C1)
```

Unfortunately, this clause is not sufficient to explain the example given as input: from this clause, we cannot prove `not_member (1, [0, 2])`. Indeed, if we try, we will have to prove successively `not_member (1, [2])`, then `not_member (1, [])`, and there will be a failure there. Thus we have to learn another clause. The basic idea is to apply NEGB again to the new example `not_member (1, [])`, which gives the clause

```
not_member (X, []) ←                                     (C2)
```

Then, the two clauses (C1) and (C2) define the negation of member.
The overall algorithm used in order to compute a negation consists of the following steps:

```
An example not_p(t1, ..., tn) is given
1. If no learned clause can be used to prove not_p(t1, ..., tn), use NEBG
   to produce a clause C.
2. Apply the newly learned clause C to the example not_p(t1, ..., tn), and
   repeat 1 on all the negative literals in the body of the clause C.
```

We give another example to fully illustrate the incrementality of the process. The program defines the minimum of an integer list. Integers are represented by the constant 0, and the functions s (X) . s (X) is the successor of X, i.e., X + 1.

```
min (X, [X]) ←
min (X, [X|L]) ← min (Y, L) ∧ less (X, Y)
min (X, [Y|L]) ← min (X, L) ∧ less (X, Y)
less (0, x (X)) ←
less (s (X), s (Y)) ← less (X, Y).
```

The example is not_min (s (0), [s (s (0)), 0]), that is "1 is not the minimum of the list [2, 0]." NEBG computes the clause[1]

not_min (X, [Y|Z]) ← Y≠X ∧ not_min (X, Z) (C3)

This clause is resolved with not_min (s (0), [s (s (0)), 0]), giving the new example not_min (s (0), [0]). NEBG then computes the clause

not_min (X, [Y|Z]) ← Y≠X ∧ not_less (X, Z) (C4)

The new example is then not_less (S (0), 0), yielding the clause

not_less (X, 0) ← (C5)

The algorithm stops there, and the learned definition of not_min is made of the clauses (C3), (C4), and (C5). If we provide another example—not_min (s (s (0)), [s (0)]), for instance—NEBG has two choices. First, it can try to apply (C3) giving the new example not_min (s (s (0)), []). From this, NEBG computes the clause

not_min (X, []) ← (C6)

The other choice is to use (C4) instead of (C3). NEBG will eventually compute the clause

not_less (s (X), s (Y)) ← not_less (X, Y) (C7)

Thus the definition of not_min already learned can be completed.

3.3. Using functional properties

The preceding algorithm can be improved to perform more simplifications. We have extended it by using some functional properties of the already known predicates. Let us take an example for the sake of clarity. We have previously learned a clause for `not_min`,

```
not_min (X, [Y|Z]) ← Y≠X ∧ not_min (X, Z)
```

This clause can be further simplified based on the remark that `min (X, Z)` expresses a function that computes the minimum `X` of a list `Z`. In particular, for a given `Z`, `X` is unique. Thus `X` is not the minimum of the list `Z` if `X` is different from the minimum `W` of `Z`. Thus we can replace the literal `not_min (X, Z)` by the conjunction `min (W, Z) ∧ X≠W`, yielding the clause

```
not_min (X, [Y|Z]) ← Y≠X ∧ min (W, Z) ∧ X≠W
```

This is an application of Rule (EF) below. We introduce some terminology to express it.

Definition 1: *A predicate $p(X_1, \ldots, X_n, Y_1, \ldots, Y_n)$ represents a function of the variables X_i, if for all ground substitutions σ or X_i, there exist at most one ground substitution θ of the Y_i such that $p(X_1\sigma, \ldots, X_n\sigma, Y_1\theta, \ldots, Y_n\theta)$ is true. (The Xi and the Yi are distincts).*

For instance, `min (X, Y)` represents a function of `Y`.
The two rules are the following we have proved that they are correct.

Functional exclusion

not_p(t, u) $\vee$ R $\rightarrow$ p(t, Z) $\wedge$ (Z$\neq$u $\vee$ R)
Z represents a set of new unknowns (EF)

Functional exclusion is used if

- p(t, u) represents a function of t
- there are no parameters in t
- The goal $\exists$Z, p(t, Z)σ, where σ is the controlling substituion, succeeds. If there is a success with answer θ, then σ is replaced by $\sigma\theta$

Functional merge

p(t, u) $\wedge$ p(t, v) $\wedge$ R $\rightarrow$ p(t, u) $\wedge$ R.θ *if θ is the mgu of u and v.* (FF)

Functional merge is used if

- p(t, u) represents a function t
- there are no parameters in t
- in this case, θ is added to the controlling substitution

4. A complete example

The grandfather example is a clear illustration of the power of functional relations. We know that a given person has only one father and only one mother. In other words, both father (X, Y) and mother (X, Y) represents a function of Y.

The completed definition for the grandfather example is

$$\forall X, Y, \text{grand-father}(X, Y) \leftrightarrow$$
$$(\exists Z, \text{father}(X, Z) \wedge \text{father}(Z, Y)) \vee (\exists W, \text{father}(X, W) \wedge \text{mother}(W, Y))$$

The second step is to negate both sides of the equivalence. The result gives a first definition of the negation of grand-father:

$$\forall X, Y, \text{not}(\text{grand-father}(X, Y)) \leftrightarrow$$
$$\forall Z\ \text{not}(\text{father}(X, Z)) \vee \text{not}(\text{father}(Z, Y)) \wedge$$
$$\forall W\ \text{not}(\text{father}(X, W)) \vee \text{not}(\text{mother}(W, Y))$$

The controlling substitution σ is {X/tom, Y/bill}. Since the goal $\exists$ Z1, father (Z1, Y) σ, which is $\exists$ Z1, father (Z1, tom), succeeds with Z1=tom on this example, the rule (EF) can be used on not (father (Z, Y))) to obtain the formula

$$\forall X, Y, \text{not}(\text{grand-father}(X, Y)) \leftarrow$$
$$\forall Z\ \exists Z1, (Z1 \neq Z \vee \text{not}(\text{father}(X, Z)) \wedge \text{father}(Z1, Y) \wedge$$
$$\forall W\ \text{not}(\text{father}(X, W)) \vee \text{not}(\text{mother}(W, Y))$$

The new controlling substitution σ is {X/tom, Y/bill, Z1/tom}. Then (EP) can be used to remove Z:

$$\forall X, Y, \text{not}(\text{grand-father}(X, Y)) \leftarrow$$
$$\exists Z1, \text{not}(\text{father}(X, Z1)) \wedge \text{father}(Z1, Y) \wedge$$
$$\forall W\ \text{not}(\text{father}(X, W)) \vee \text{not}(\text{mother}(W, Y))$$

Since the goal $\exists$ Z1, X1, father (X1, Z1) σ, which is $\exists$ X1, father (X1, tom), suceeds with X1=paul in this example, the rule (EF) canbe used on not (father (X, Z1)), which gives

$$\forall X, Y, \text{not}(\text{grand-father}(X, Y)) \leftarrow$$
$$\exists Z1, X1, \text{father}(X1, Z1) \wedge (\text{father}(Z1, Y) \wedge X1 \neq X \wedge$$
$$\forall W\ \text{not}(\text{father}(X, W)) \vee \text{not}(\text{mother}(W, Y))$$

The same sequence (EF) (EP) (EF) can be applied to

$$\forall W\ \text{not}(\text{rather}(X, W)) \vee \text{not}(\text{mother}(W, Y)),$$

which gives

```
∀ X, Y, not (grand-father (X, Y)) ←
        ∃ Z1, X1, father (X1, Z1) ∧ father (Z1, Y)) ∧ X1≠X ∧
        ∃ W1, X2, father (X2, W1) ∧ mother (W1, Y) ∧ X2≠X
```

with controlling substitution σ:

```
{X/tom, Y/bill, Z1/tom, X1/paul, W1/anna, X2/john}
```

This clause expresses the fact, "X is not the grand-father of Y if X is different from the two grand-fathers X1 and X2 of Y." This is interesting, since it is expressed with positive predicates instead of their negations. If we apply the controlling substitution σ to this clause, we obtain an explanation of the negative example:

```
not (grand-father (tom, bill)) ←
        father (paul, tom) ∧ father (tom, bill) ∧ paul≠tom ∧
        father (john, anna) ∧ mother (anna, bill) ∧ john≠tom
```

5. Discussion

5.1. Implementation

The formulation of NEBG is a set of rewrite rules that can be confusing since it looks quite complicated. In fact, it can be quite straightforwardly implemented in Prolog. This program uses the same trick as the program in Kedar-Cabelli et al. (1987): it maintains two copies of the formulas to be transformed, one general and one instantiated by the training negative example, In the instantiated copy, unknowns are instantiated by the controlling substitution. This implementation avoids the manipulation of explicit quantifiers. The use of rules is totally deterministic: if possible, uses (EP), (D), (C) (OC) as soon and as much as possible. Then try (EF) and (R) and finally (EX). Rule (EF) requires a test of provability. This test is performed by directly calling Prolog. The resulting program is not very big: around four pages. All the examples presented here run in about a second or less on a Sun 3 workstation. NEBG is implemented in Quintus Prolog.

5.2. Related work

NEBG shares the methodology of the EBG method. An explanation of the example is constructed and is generalized. However, the algorithm used is completely different. EBG can be described in a logic programming context as follows.

<u>EBG</u>

Given

> A theory T defining the predicate p
> A set of facts E (the example in Mitchell's terminology)
> An atom B with predicate symbol p (the goal in Mitchell's terminology)

Such that T |- (B ← E)

Find a clause H such that

> H |- (B ← E) and T |- H

The condition T |- H expresses that EBG is a *valid* method: the learned knowledge H is correct.

NEBG proceeds completely differently from EBG. First of all, the inputs are very different: the theory cannot be used to prove F, and thus EBG cannot be applied. We can summarize NEBG as follows:

<u>NEBG</u>

Given

> A theory T defining the predicate p
> An atom F with predicate symbol p

Such that T |-/- F

Find a set of clauses H such that

> T, H |- not_F and comp(T) |- H

The condition comp(T) |- H expresses that NEBG is also a *valid* method: the learned knowledge H is a logical consequence of the completed database of the theory T.

This clearly shows the difference between NEBG and all the work based on the application of EBG to learn from failures, such as PRODIGY (Minton et al., 1987), FAILSAFE (Mostow et al., 1987), SOAR (Newell et al., 1987; Gupta, 1987). In all these systems, a theory TF of the possible failures must be provided along with the theory T. Then the failure F is explained building a positive explanation of F from TF, and then apply EBG on this explanation. In fact NEBG may be seen as a way to automatically build TF from T, and thus it is complementary to all these works.

This parallel between NEBG and EBG raises the question, "What about the operationality criteria used in EBG?" Such criteria can be given, and NEBG will avoid learning definitions of the negation of operational predicates. This achieves the intended effect of ignoring the details of the definitions of operational predicates.

Besides these works in machine learning, some work in the theoretic fields of computer science is related to ours. The basic transformation we gave in section 3 is similar to work on disunification (Comon, 1988). Disunification has been applied in Logic Programming to generate the negation of a predicate (Lugiez, 1988). However, our work has two advantages over Lugiez's: we use an example that enables us to have a very efficient algorithm, and we use functional properties.

5.3. Use of NEBG

Often, in artificial intelligence, the negation of the already known concept C is accomplished through a form of negation as failure: A is false if A cannot be proved. Unfortunately, this is not always sufficient; an explicit definition of not(C) is required. NEBG can be used in such cases.

For instances, we have used NEBG is planning domains as a basic mechanism to construct a definition of what does not change from what changes. This is used by the LIFE system to learn knowledge about invariant features of the application domains. This work is fully described by Puget (1986b), and we just show here how NEBG is used.

The problem of LIFE is to explain the failures of a planner. In particular, this requires us to be able to explain why something cannot be changed by the planner, whatever he does. Unfortunately, in planning work, actions are often represented with so-called STRIPS axioms (Fikes et al., 1971) of the following form:

$$P(Op) \colon D \rightarrow A.$$

The intended meaning of such an axiom is that if the formula P(Op) is true and the action Op is performed, then the literal D is deleted, and the literal A is added to the state description.

Using this, it is difficult to explain why something does not change, whatever action is performed. This equation is related to the well-known "frame problem" (Brown, 1987). We give here our solution. A literal L can be changed by an action if and only if there is a STRIPS axiom $P(Op){:}D \rightarrow A$ such that P(Op) is true and D matches L. This is expressed by defining a predicate "change" by transforming every STRIPS axiom $P(Op){:}D \rightarrow A$ into the clause

$$change(D) \leftarrow P(Op)$$

Thus a literal L cannot be changed in a step if and only if the predicate change (L) fails. Then, in order to explain why L cannot change, NEBG is applied to the definition of change.

Another potential use of NEBG is for nonmonotonic learning systems such as MOBAL (Emde, 1987; Wrobel, 1992). In these systems, negative examples to an already learned concept C are stored. When sufficiently many such negative examples are available, a generalization NC is computed, and every clause $C \leftarrow B$ defining C is replaced by

$$C \leftarrow B \wedge not(NC)$$

thus introducing negations. NEBG could be used to remove these transformations.

5.4. *Summary of main results*

We have introduced NEBG, a completely new machine learning tool that learns the negation of already known concepts. NEBG resembles the EBG method as a valid and deductive method that uses a single example. However, the principles used are totally different, and they are described in detail. We have proved nice properties, such as the correctness of the results, and also the termination of the algorithm used (there can be no loops). Moreover, our method is fully automated and does not rely on the user for control purpose. It is even used as a subpart in LIFE, another learning system. Improvements are still possible because NEBG cannot always remove parameters in the definitions it produces. This is a theoretical issue in logic programming, and the solutions are not always satisfactory for the time being.

Acknowledgments

This work has been done while the author was a Ph.D. student in Yves Kodratoff's group at LRI, Orsay, France.

I have really appreciated the comments made by Celine Rouveirol, Yves Kodratoff, Katharina Morik, and Jacques Nicolas on an earlier version of this article. The author had a CNRS scholarship and has been partially supported by PRC-IA of CNRS.

Notes

1. This is fully detailed in Puget (1989a).

References

Brown (1987). The frame problem in artificial intelligence. *Proceedings of the 87 Workshop.* Morgan Kaufmann.

Clark, K.L. (1978). Negation as failure. In H. Gallaire & J. Minker (Eds.), *Logic and data bases.* New York: Plenum Press, pp. 293–322.

Comon, H. (1988). *Unification et disunification: théorie et applications.* Doctoral thesis, INPG, Grenoble.

Dawson, Siklossy. (1977). The role of preprocessing in problem solving. *Proceedings of IJCAI 5* (pp. 465–471). Morgan Kaufmann.

Dejong, J., & Mooney, R. (1986). Explanation-based learning: An alternative view. *Machine Learning, 2.*

de Siqueira, J. & Puget, J.-F. (1988). Explanation based generalization of failures. Proceedings of ECAI 88, Pisa, Italy. New York: Pitman, pp. 339–344.

Emde, W. (1987). Non cumulative learning in Metaxa. *Proceedings of the Tenth International Conference on Artificial Intelligence.* Milan, Italy. Morgan Kaufmann.

Fikes, Nilsson. (1971). Acquiring and executing generalized robot plans. In Weber & Nilsson (Eds.) (1980). *Readings in A.I.* Morgan Kaufmann.

Gupta, A. (1987). Explanation based failure recovery. *Proceedings of AAAI 87.* Seattle, WA: Morgan Kaufmann.

Hirsh (1987). In P. Brazdil (Ed.), *Proceedings of the Workshop on Metalevel Reasoning and Learning.*

Jaffar, J., Lassez, J.L., & Lloyd, J.W. (1983). Completeness of the negation as failure rule. *Proceedings of 8th IJCAI.* Karlsruhe, Germany. Morgan Kaufmann.

Lloyd, J.W. (1987). *Foundations of logic programming,* 2nd edition. Berlin: Springer Verlag.

Lugiez (1989). A deduction procedure for first order programs. *Proceedings of International Conference on Logic Programming, ICLP89.*

Minton, S., & Carbonell, J.G. (1987). Strategies for learning search control rules: An explanation based approach. *Proceedings of the Tenth International Conference on Artificial Intelligence* (pp. 228–235) Milan Italy: Morgan Kaufmann.

Mitchell, T., Keller, R., & Kedar-Cabelli, S. (1986). Explanation-based generalization: A unifying view. *Machine Learning, 1.*

Mostow, Bhatnagar. (1987). Failsafe. In *Proceedings of IJCAI87* (pp. 249–256). Morgan Kaufmann.

Puget, J.-F. (1989a). (In French). Evaluation partielle d'echecs en prolog. In *Actes du séminaire de programmation en logique de Trégastel, SPLT'89* Lannion, 24–26 mai. Université Paris Sud, 91405 Orsay, France. Also available as report n°475, LRI, April.

Puget, J.-F. (1989b). Learning invariant from explanations. In *Proceedings of the Sixth International Machine Learning Workshop.* Ithaca, NY. Morgan Kaufmann.

Puget, J.-F. (1990). *Apprentissage par explications d'echecs.* Ph.D. dissertation, Université Paris Sud, 91405 Orsay, France.

Shepherdson, J.C. (1988). Negation in logic programming. In J. Minker (Ed.), *Foundation of deductive databases and logic programming.* Morgan Kaufmann, pp. 19–87.

Wrobel, Stefan. (1993). Concept formation during interactive theory revision. *Machine Learning* (this issue).

Received February 10, 1992
Accepted June 4, 1992
Final Manuscript September 15, 1992