Learning By Explanation of Failures

Paulo Urbano[1]
Laboratório de Informática e Sistemas
Universidade de Coimbra
Quinta da Boavista, lote 1, 1ª
3000 Coimbra
Portugal

Abstract

The EBG learning technique has been mainly used in learning processes based on positive examples and successful experiences. However, several authors have demonstrated that failed proofs revealed to be quite useful as a form of avoiding future failures. The first attempts to learn from failure were based on the axiomatization of the problem-solver and on the creation of a specific meta-theory for all possible failures. Whenever there is a positive example of a failure, EBG is used to make operational the meta-theory. Siqueira & Puget designed a new technique with a different philosophy to learn from counter-examples using only the domain theory. Their method finds a sufficient generalized condition from the failed proof of a goal. EBGF is still a fragile and incomplete technique as it doesn't cover all cases. The failure of a proof has specific characteristics which are not considered when we deal with positive proofs. In this paper we show the weaknesses of EBGF and we propose an improved technique to learn from failures in the presence of a counter-example. Our method is implemented in Prolog and its efficiency is currently under analysis.

Keywords Explanation based learning, failure.

1 Introduction

Learning from unsuccessful experiences can be as important in a deduction or planning task as it is the current approaches to learning from positive examples (Minton & Carbonell, 1987; Gupta, 1987;

---

[1] Current adress: Departamento de Informática da Universidade de Lisboa, Avª 24 de Julho, 131, 7º, P-1200 Lisboa Portugal.

Hammond, 1987). Learning from failures can be very useful in presence of default knowledge and, generally, whenever we reason with incomplete knowledge. As Hirsh showed (Hirsh, 1987), if we want to apply the EBG method (Mitchell, Keller & Kedar-Kabelli, 1986) using a default rule of some goal (Goal), we have to introduce in the domain theory a particular predicate of the kind "unknown(Goal)", which will also appear in the learned rule. The problem with this procedure is that this predicate is not operational and so the EBG technique is not fully effective. Therefore, we also have to make operational the predicate which represents the incomplete information. This is an example of the large domain of applications suited for learning from failures in presence of counter-examples.

The first attempts to learn from failures were based on the definition of a meta-theory representing all possible failures. EBG was then applied to make operational the failure's theory using positives examples of a failure. This approach can be inefficient because we have to define all possible failures in the meta theory.
Siqueira & Puget developed a new technique and an algorithm called EBGF (Siqueira & Puget, 1988), in the universe of logic programming, that learns from counter-examples using only the domain theory. The method is similar to EBG but uses a counter-example rather than an example. EBGF, from a failed proof, finds a generalized condition which satisfies the counter-example, is sufficient to warrant the failure of the goal, covering similar counter-examples.
This method is still incomplete and ambiguous. There are some cases in which it cannot even make the failure operational. The different quality of literals deserves an importance that has been ignored because there are literals which demand that at least one of their variables should be instantiated before tested. EBGF is only able to deal with literals that make no demands on their variables Moreover, the order of the literals in a conjunction can be very important and cannot be changed without the risk of aborting the process.
We have developed a learning technique that is a revision and a major extension of EBGF, and is able to solve the problems found.

In section (2) we describe EBGF illustrated with an example; in (3) we analyze its insufficiencies; in section(4) we present our method and finally in section (5) we present an example showing the functioning mode of our method.

2 Explanation Based Generalization of Failures (EBGF)

Siqueira & Puget's algorithm is situated in the universe of logic programming. Its aim is to find the minimal sufficient conditions for failure during the process of classifying a counter-example. The failure is then generalized in a way to cover similar counter-examples.

2.1 Logical framework of EBGF

The logical framework supporting this method depends on the Negation As Failure assumption (NAF). It also depends on the notion of Completed Data Base (CBD) - each concept is rewritten in its complete form. EBGF uses first order predicate logic restricted to Horn clauses and the interpreter is SLD-resolution.
The method is based on Clark's conclusion that using NAF inference rule is equivalent to make deductions from the completed domain theory. We rewrite the theory so that the right part of each clause of the target concept G is a conjunction of atomic literals CL_j (literals that cannot be defined through other literals). We can expand this notion of atomic to different operationality criterion as in EBG (Hirsh, 1988). We have then a theory in which every clause of the target concept G is compiled - each CL_j is formed by atomic or operational literals .The process is described in (Siqueira & Puget, 1988).

$$G \leftarrow CL_1 \vee CL_2 \vee \ldots \vee CL_m$$

The negation of G corresponds to the rule:

$$\neg G \leftarrow \neg CL_1 \wedge \neg CL_2 \wedge \ldots \wedge \neg CL_m$$

However, this rule is not in general an operational rule since it is too complex and may have redundant literals - literals that are not responsible for the failure. Moreover, the process of transforming any theory in its complete form can be a difficult task, especially in the presence of recursive clauses.
The idea of EBGF is to simplify each conjunction of literals CL_j so that in the end we have only sufficient literals for the failure.

2.2 General description of EBGF

Given a domain theory, a target concept, an operationality criterion and a counter-example, EBGF finds an operational definition for the negation of the target concept, increasing the problem-solver efficiency. For that purpose it builds a negative explanation of why the counter-example is not an example of the target concept. The explanation is generated from the failed proof and it is generalized covering a larger number of similar counter-examples.

EBGF is divided in three steps which we will describe:

a) Finding the generalized conjunctions CLj

The manipulation of the completed data base can be too complex and expensive, so the system tries to prove the goal using the domain theory and the counter-example. When the proof is not possible it can collect one of the generalized conjunctions CL_j mentioned above; the system backtracks and collects every generalized conjunction. The generalization is done in parallel, taking as input a generalized version of the target concept and performing SLD-resolution as done in (Kedar-Kabelli & Mc Carty, 1987).The role of the counter-example is to bias the search and to avoid recursivity problems.

b) Simplification of every conjunction CLj

The next step consists in the simplification of each generalized conjunction using the counter-example. The simplification method relies on the rightmost literal heuristics. Siqueira & Puget give an intuitive definition of the rightmost literal of a conjunction: it is the one that always fails and doesn't allow the proof to go further to the right. This method deletes, from the original conjunction, every literal which is redundant.
We will describe in detail the simplification algorithm:

Input: a conjunction of literals $CL=\Lambda_i L_i$ (i = 1..n).
Output: a conjunction of literals CS, sufficient for the failure.

0- Initially, the simplified conjunction CS has no literals.

1- Let L_d be the rightmost literal of $\Lambda_i L_i$. We remove it from $\Lambda_i L_i$ and insert it on the end of the simplified conjunction CS.

2- If CS is true for the counter-example, all the literals from CS are satisfied by an instantiation I; we apply I to the variables of $\Lambda_i L_i$ and return to step 1. Otherwise, if S is false for the counter-example then CS is the final simplified conjunction.

c) Operational rule of failure

The learned operational rule representing the negation of the target concept is:

$$\neg G \leftarrow \neg CS_1 \wedge \ldots \wedge \neg CS_m$$

where CS_j is the result of simplifying each CL_j.

2.3 Example of EBGF

The following example, representing the game of Othelo (Siqueira & Puget, 1988), will illustrate the EBGF technique.

We have a theory expressing the conditions in which a square, with a particular colour and occupying a particular position on the game board (8*8), can be flipped. "status" defines the square's situation; "pos" defines a relation of distance between two board positions. Each position is represented by a number (1...64). The surrounding of any square is made by symmetric distances and is represented by "surr1" and "surr2".

Example 1

- theory

flips_to(P,C1) ← opponent(C1,C0) ∧ surr1(P,D,C0) ∧ surr2(P,D,C0)
surr1(P,D,C1) ← pos(P,P1,D) ∧ status(P1,empty)
surr1(P,D,C1) ← pos(P,P1,D) ∧ status(P1,C1) ∧ surr1(P1,D,C1)
surr2(P,D,C1) ← pos(P2,P,D) ∧ status(P2,empty)
surr2(P,D,C1) ← pos(P2,P,D) ∧ status(P2,C1) ∧ surr2(P2,D,C1)

- target concept

flips_to(P, C1)

- Operationality criterion

operational(pos(_,_,_)) operational(status(_,_))

- Counter-example

opponent(black,white)	opponent(white,black)
pos(1,2,1)	pos(1,9,8)
pos(1,10,9)	pos(2,3,1)
pos(2,1,-1)	
status(1,white)	status(2,black)
status(3,empty)	status(9,empty)
status(10,empty)	

We cannot prove "flips_to(1,white)".

One of the generalized conjunctions obtained from the failed proof, when the resolution couldn't go further, is:

CL_1=opponent(C1,C0) ∧ pos(P,P1,D) ∧ status(P1,empty) ∧ pos(P2,P,D) ∧ status(P2,empty)

We will show, in detail, the simplification algorithm, taking CL_1 as input:

0- CS_1 initially has no literals.

cycle1

1- We apply the instantiation of the target concept, [P=1,C1=white], to CL_1. The rightmost literal of CL_1 is "pos(P2,P,D)"; we delete it from CL_1 and we insert it in CS_1.

2- CS_1=pos(P2,P,D) is true for the counter-example; we can apply the instantiation [P2=2, P=1, D=-1] to the variables of CL_1 and return to step 1.

cycle 2

1- CL_1=opponent(white,C0) ∧ pos(1,P1,-1) ∧ status(P1,empty) ∧ status(P2,empty);
the rightmost literal of CL_1 is "pos(P,P1,D)"; we delete it from CL_1 and insert it in CS_1.

2- CS_1=pos(P2,P,D) ∧ pos(P,P1,D) is false for the counter-example and it is the final simplified conjunction.

When we apply the simplification algorithm to the other conjunctions CL_j we obtain exactly the former simplified conjunction CL_1.

At the end, the system will learn the operational rule:

$$\neg\text{flips\_to}(P,C1) \leftarrow \neg[\text{pos}(P2,P,D) \wedge \text{pos}(P,P1,D)]$$

This learned rule represents the fact that square P cannot be fliped if it is a corner.

3 Insufficiencies detected in EBGF

The two limitations we found in EBGF are in the second step: the process of simplification of the original conjunctions.

The first is an error detected in the rightmost heuristics due to the dependance upon the choice of instantiations. In fact, the algorithm can output different answers depending on the chosen instantiations.

The second limitation reveals EBGF to be unprepared to deal with the particularities of the failure universe. In a conjunction there are literals that are responsible for the first instantiations of some variables, given the atomic attributes of the counter-example, and the consequent value propagation to the

right. Other literals need to receive decisive instantiations from the left and cannot be tested otherwise. The order of the literals in a conjunction can, for this reason, be very important and cannot be altered when we transform one conjunction into another. EBGF is only capable to deal with literals which make no demands on the instantiations of their variables making the method limited.

In the following two sections we are going to describe these two limitations in more detail by means of examples.

3.1 Incorrectness of EBGF

We are going to analyze how EBGF deals with the following example adapted from (Siqueira & Puget, 1988).

Example 2

- Theory

  $a(X,Y) \leftarrow b(X,Z) \wedge c(X) \wedge d(Z,Y)$

  $a(X,Y) \leftarrow e(X) \wedge f(Y)$

  $b(X,Y) \leftarrow g(X,Y)$

  $b(X,Y) \leftarrow h(X,Y)$

- Target concept

  $a(X,Y)$

- Every attribute from the counter-example is operational.

- Counter-example

  $c(2)$, $d(1,2)$, $d(3,2)$, $f(1)$, $g(2,2)$, $g(1,1)$, $h(2,2)$

We cannot prove "a(X,Y)". Let´s initiate the failure analysis:

$CL_1 = g(X,Z) \wedge c(X) \wedge d(Z,Y)$ is one of the generalized conjunctions returned after the first step.

Now, we will apply the rightmost heuristics to CL_1 to find the conjunction of literals which is the sufficient condition for CL_1's failure:

0- CS_1 is initially empty of literals.

cycle 1

1- "d(Z,Y)" is the rightmost literal of CL_1. We delete it from CL_1 and insert it in the sufficient conjunction CS_1.

2- CS_1 is true for the counter-example: there are two instantiations that satisfy CS_1: I_1=[Z=1,Y=2] and I_2=[Z=3,Y=2]. We can apply I_1 or I_2 to CL_1.

cycle 2

1- the rightmost literal of CL_1 after I_1 is "c(X)". the rightmost literal of CL_1 after I_2 is "g(X,Z)", which is not a correct one.

This case demonstrates that the simplification algorithm is ambiguous and can output wrong answers. The simplification process cannot depend on a casual choice of instantiations of the current sufficient condition.

3.2 Incompleteness of EBGF

Let us consider the following example taken from (Mitchell & Kedar-Kabelli, 1986):

Example 3

- Theory

safe_to_stack(X,Y) ← lighter(X,Y)
lighter(X,Y) ← weight(X,Px) ∧ weight(Y,Py) ∧ <(Px,Py)
weight(X,Px) ← volume(X,Vx) ∧ density(X,Dx) ∧ ×(Vx,Dx,Px)
weight(X,5) ← isa(X,table) (default rule)

- Target concept

safe_to_stack(X,Y)

- Operationality criterion

operational(volume(_,_)) operational(density(_,_))
operational(isa(_,_)) operational(<(_,_))
operational(×(_,_,_))

- Counter-example

volume(box1,2) volume(table1,3)
density(box1,4) density(table1,2)
isa(box1,box) isa(table1,table)

After the first step of the algorithm, one of the operational conjunctions that causes the failure of "safe_to_stack(box1,table1)" is:

CL_1=volume(X,Vx) ∧ density(X,Dx) ∧ ×(Vx,Dx,Px) ∧ isa(Y,table) ∧ <(Px,5)

Now, we will apply the rightmost heuristics to CL_1 to find the conjunction of literals which is the sufficient condition for CL_1's failure:

0- CS_1 is initially empty of literals.

1- The rightmost literal of CL_1 is "<(Px,5)"; we delete it from CL_1 and insert it in the sufficient conjunction CL_1.
2- When we try to test if CS_1=<(Px,5) is satisfied for the counter-example we detect an error: literal "</2" demands its first variable instantiated before tested.

In this way, we conclude that it is not enough to find the rightmost literal of a conjunction. The simplification algorithm has to be changed to cover the case in which there are literals that demand at least one of its variables previously instantiated. We have to put other literals in the sufficient condition to warrant the necessary instantiations. Note that when we put literals in the sufficient conditions CS_j we have to maintain their original order in CL_j because if not there is the risk of breaking the original link of variables that guaranteed the propagation of instantiations.

4 Alternative method

Our method has also three steps as EBGF. We have reviewed and expanded EBGF on two points: we have a different and improved process of finding the initial generalized conjunctions from the failed proof, and we have reformulated the rightmost literal heuristics in a way to solve both limitations analyzed above.

4.1 Finding the initial generalized conjunctions

In EBGF, we initially obtain the generalized conjunctions responsible for the failure. However, these conjunctions might be too complex and redundant, which made us think in a way of simplifying them.

The advantage is that we can directly obtain a conjunction with the first rightmost literal in the end. To accomplish this goal we built a special interpreter adapted to failures.

The interpreter is quite simple: the explanation of the failure of a literal which is not operational and is the head of a clause, is the explanation of the failure of the clause's body. The explanation of the failure of an operational literal which fails is just the literal. The explanation of the failure of a conjunction of literals $\Lambda_i N_i$ (i=1...x...n), is based on the first conjunction $\Lambda_i N_i$ (i=1...x) which does not satisfy the counter-example because of its last literal N_x. That is, $\Lambda_i N_i$ (i=1...x) is false but $\Lambda_i N_i$ (i=1...x-1) is true. We operationalize the conjunction $\Lambda_i N_i$ (i=1...x-1) using EBG and concatenate it with the explanation of the failure of the literal N_x.
That way, when we fail to prove a goal we always obtain an explanation represented by a conjunction of operational literals in which the last one is the first rightmost literal. In conclusion, our initial generalized conjunctions represent a first direct simplification to the initial generalized conjunctions of EBGF. We do the generalization process in parallel, maintaining a generalized version of the goal. If there is more than one clause for some literal then the system backtracks, collecting every explanation. This interpreter has to ignore all control symbols he finds traversing every possible path, otherwise it could ignore other clauses that could fail.

Now we will show the difference between our method and EBGF, using example 3 with the following counter-example:

Example 4

- Counter-example

isa(box3,box)	density(box4,4)
volume(box3,2)	isa(box4,box)

In this case EBGF outputs CL_1 as one of the initial generalized conjunctions for the failure of "safe_to_stack(box3,box4)":

CL_1=volume(X,Vx) Λ density(X,Dx) Λ $\times$(Vx,Dx,Px) Λ weight(Y,Py) Λ <(Px,Py)

Let´s apply our method:

- safe_to_stack(box3,box4) fails because weight(box3,Px) Λ weight(box4,Py) Λ <(Px,Py) fails.
- weight(box3,Px) Λ weight(box4,Py) Λ <(Px,Py)] fails because weight(box3,Px) fails.
- weight(box3,Px)] fails because volume(box3,Vx) Λ density(box3,Dx) Λ $\times$(Vx,Dx,Px) fails.
- volume(box3, Vx) Λ density(box3, Dx) Λ $\times$(Vx, Dx, Px) fails because volume(box3,Vx) Λ density(box3,Dx) fails; volume(box3,Vx) is true.

We conclude that the failure of volume(box3,Vx) ∧ density(box3,Dx), which is already composed by operational literals, explains the failure of the target concept. Therefore, the generalized conjunction is:

CL_1=volume(X,Vx) ∧ density(X,Dx)

When we try to prove the goal we simplify directly the conjunction wich is obtained by EBGF. If the rightmost literal is the last to be responsible for the failure then the literals which are positioned on its right are redundant for the failure.

4.2 Reformulation of the simplification algorithm

We begin by formally defining the concept of the rightmost literal of a conjunction.

Definition 1
The rightmost literal of a conjunction $CL = \Lambda_i L_i$ (i=1...n) is:
- L_1 if L_1 is always false.
- L_d if $\Lambda_i L_i$ (i=1...d-1) is true and $\Lambda_i L_i$ (i=1...d) is always false.

Next, we will present two algorithms that solve the two insufficiencies detected in EBGF.
The former algorithm solves the first problem: the next rightmost literal of a conjunction CL during the simplification process cannot depend on the different instantiations that satisfy the conjunction formed by the past rightmost literals CS.
The final algorithm expands the first as it is capable to deal with literals that demand at least one of their variables previously instantiated to be correctly applied.

4.2.1 First simplification algorithm

When we want to output the rightmost literal of a conjunction in any moment of the simplification algorithm, we must take into account every literal of the original conjunction. EBGF separates the sufficient literals and looks for the next rightmost literal in the remaining conjunction, making the process dependent on the instantiations of the separated literals, as we have shown above. To solve this problem, what we do is to change the order of the several rightmost literals, putting them on the left of the

remaining conjunction. Then we look for the next rightmost literal in the conjunction formed by every original literal where the past rightmost literals have changed order.
This way the choice of the rightmost literal is not ambiguous, giving as output the only rightmost literal of a conjunction of literals.

The algorithm

We consider that the initial conjunction CL is composed by two kinds of literals: those which are decisive for failure and belong to the sufficient conjunction CS and those which are redundant.and belong to the redundant conjunction CR.

Input: a conjunction of literals $CL=\Lambda_i L_i$ (i = 1..n).
Output: a sufficient conjunction of literals CS.
a conjunction of redundant literals CR.

0- initially CR = CL; CS has no literals.

1- Find the rightmost literal of the ordered concatenation of conjunctions CS and CR (CS∧CR); delete it from CR and put it in the end of CS.

2- If CS is false for the counter-example then CS is the final simplified conjunction, else return to step 1.

Example

Now we are going to apply the simplification algorithm to the conjunction of example 2 that created problems to EBGF.

CL=g(X,Z) ∧ c(X) ∧ d(Z,Y)

0- CR=CL; CS is a conjunction with no literals.

cycle 1
1- The rightmost literal of CS∧CR is "d(Z,Y)". CR=g(X,Z) ∧ c(X); CS=d(Z,Y).
2- d(Z,Y) is true for the instantiation [Z=1,Y=2]; we return to step 1.

cycle 2
1- CS∧CR=d(Z,Y) ∧ g(X,Z) ∧ c(X). By definition 1, the rightmost literal of CS∧CR is "c(X)".

The literals from CS always have to take part in the conjunction where every rightmost literal is sought - this conjunction has always the same literals, their order only being changed during the aplication of the algorithm.

4.2.2 The final simplification algorithm

Type of literals

We consider two types of literals: those which do not need any of its variables previously instantiated and those which demand that at least one of their variables should be previously instantiated.
The first is the general case. The second type of literals have to be declared in the domain theory.

For instance, if we want to assert that the literal with symbol "×" and arity 3 needs its first two variables instantiated before it is applied, we declare:

instantiated(×\3,[1,2])

The former version of the rightmost literal algorithm we designed is only able to deal with the first type literals. To deal with all kinds of literals, we have to be sure that when we test the successive rightmost literals conjunction CS, there are always literals that supply them with the necessary instances. We have thus to formalize the notion that a literal receives its necessary instances from a conjunction of other literals. The following section is entirely dedicated to this formalization.

Definitions

Definition 2
A literal L_x is a basic literal if
- it is a literal that does not need any of its variables previously instantiated
or if
- it is a literal in which every variable that needs instantiation is instantiated.

Definition 3
A literal L_x is supported in a conjunction of literals $\Lambda_i L_i$ (i =1...x...n) if
-it is a basic literal
or if
- x>1 and each variable in L_x that needs instantiation and it is not instantiated, belongs to some literal in $\Lambda_i L_i$ (i=1...x-1) wich is supported in $\Lambda_i L_i$ (i=1...x...n).

The algorithm

Now, we can expand the first version of the rightmost literal algorithm:

Input: A conjunction of literals CL.
Output: CS - sufficient conjunction of literals for the failure of CL.
CR - conjunction of redundant literals.

0- Initially CR=CL and CS is empty of literals.

1- Find L_d,the rightmost literal of the ordered concatenation of conjunctions CS and CR (CS$\wedge$CR). Delete L_d from CR and insert it in CL.

2- Case1: if Ld is supported in CS goto 3, else
Case 2: Look for a minimum set of literals of CR that once inserted in CS result in Ld being now supported in CS. Delete the set of literals found from CR and goto to 3.

3- If CS is false then $\neg$CS is the sufficient condition; else goto step 1.

Note that, whenever we insert any literal in CS we have to maintain its original position in LC. If we change the order we might break the link between variables that allows the propagation of decisive values.

Example

We are going to show how this algorithm deals with the following generalized conjunction taken from example 3.

CL_1=volume(X,Vx) $\wedge$ density(X,Dx) $\wedge$ $\times$(Vx,Dx,Px) $\wedge$ isa(Y,table) $\wedge$ <(Px,5)

We have to add the following literals to the original theory.

instantiated(<\2[1,2])
instantiated(×\3,[1,2])

Let's apply our method to simplify CL_1.

0- CR_1=CL_1; CS_1 has no literals.

1- The rightmost literal in $CS_1 \wedge CR_1$ is L_d="<(Px,5)". We inserte it in CS_1.

2- "<(Px,5)" is not a basic literal because, by definition 2, it needs Px previously instantiated. L_d is the only literal in CS_1 and so, by definition 3, it is not supported in CS_1. We have to find a minimum set of literals from CR_1 so that when inserted in CS_1, L_d will be supported in CS_1:

[volume(X,Vx),density(X,Dx),×(Vx,Dx,Px)] is the minimum set of literals from CR_1 so that when inserted in CS_1, maintaining their original positions in CL_1, L_d will be supported in CS_1. Next, we are going to prove it:

The sufficient condition will be:

CS_1=volume(X,Vx) $\wedge$ density(X,Dx) $\wedge$ ×(Vx,Dx,Px) $\wedge$ <(Px,5)

"<(Px,5)" is supported in CS_1: it is not a basic literal but Px belongs to the literal "×(Vx,Dx,Px)" which is supported in CS_1.

"×(Vx,Dx,Px)" is supported in CS_1: It is not a basic literal (it needs Vx and Dx instantiated); however, Vx and Dx belong respectively to the literals "volume(X,Vx)" and "density(X,Dx)" which are both supported in CS_1, because they are basic literals.

3- Finally, CS_1 is false for the counter-example and the redundant conjunction is CR_1=isa(Y,table). At the end we have $\neg$[volume(X,Vx) $\wedge$ density(X,Dx) $\wedge$ ×(Vx,Dx,Px) $\wedge$ <(Px,5)] as the sufficient condition for the failure of CL_1.

5 Example of our technique

Let us illustrate our method solving example 3 in order to obtain the operational rule for the failure of "safe_to_stack(box1,table1)".

After the first step we have as the initials conjunctions for failure:

CL_1=volume(X,Vx) ∧ density(X,Dx) ∧ ×(Vx,Dx,Px) ∧ volume(Y,Vy) ∧ density(Y,Dy) ∧ ×(Vy,Dy,Py) ∧ <(Px,Py)

CL_2=volume(X,Vx) ∧ density(X,Dx) ∧ ×(Vx,Dx,Px) ∧ isa(Y,table) ∧ <(Px,5)

CL_3=isa(X,table)

The final operational rule the system learns is:

¬safe_to_stack(X,Y) ←
¬[volume(X,Vx) ∧ density(X,Dx) ∧×(Vx,Dx,Px) ∧ volume(Y,Vy) ∧ density(Y,Dy) ∧ ×(Vy,Dy,Py) ∧ <(Px,Py)]
∧
¬[volume(X,Vx) ∧ density(X,Dx) ∧ ×(Vx,Dx,Px) ∧ <(Px,5)]
∧
¬[isa(X,table)]

6 Conclusion

We have developed a revision and a major extension of the method EBGF which learns from failure. EBGF is a method that on the contrary of the current approaches doesn't need a meta-theory specific for failures. We found insufficiencies in EBGF in which the most serious was the fact that it was not capable to deal with literals which demand at least one of their variables previously instantiated. These problems limited EBGF making learning inefficient. We have developed and implemented a technique which solves the insufficiencies limiting EBGF.

Acknowledgments

I would like to thank Prof. Ernesto Costa for all his support.

References

Siqueira, J. CL. e Puget, J. F., Explanation-Based Generalization of Failures, *Proceedings of the* ECAI-88, pp 339-344, 1988.

Mitchell T. M., Keller R. M.,& Kedar-Kabelli S. T., Explanation-based Generalization: a unifying view, *Machine Learning* 1:1, pp 47-80, 1986.

Minton, S. and Carbonell, J. G., Strategies for Learning Search Control Rules: An Explanation-based Approach. *Proceedings of the* 10th. IJCAI, Milan, pp 228-235, 1987.

Gupta, A., Explanation-Based Failure Recovery, *Proceedings* AAAI- 87, pp 606-610, 1987.

Hirsh, H., Explanation-based generalization in a logic-programming environment. *In Proceedings of the* 10th. IJCAI, Milan, pp 221-227, 1987.

Hirsh H., Reasoning about operationality for Explanation-based Learning. *In Proceedings of the Fourth International Workshop on Machine Learning* , pp 214-220, 1988.

Kedar-Kabelli, S. T.& Mc Carty, CL. T., Explanation-Based Generalization as Resolution Theorem Proving, *Proceedings of the Fourth International Workshop on Machine Learning*, pp 383-389, 1987.

Hammond, J. K., Explanation and Repairing Plans that Fail, *Proceedings of the* 10th. IJCAI, Milan, pp 109-114, 1987.