Logic Aided Lamarckian Evolution

Evelina Lamma
DEIS, Università di Bologna,
Viale Risorgimento 2,
40136 Bologna, Italy,
elamma@deis.unibo.it

Luís Moniz Pereira
Centro de Inteligência Artificial (CENTRIA),
Faculdade de Ciências e Tecnologia,
Universidade Nova de Lisboa,
2825-114 Caparica, Portugal
lmp@di.fct.unl.pt

Fabrizio Riguzzi
Dipartimento di Ingegneria,
Università di Ferrara, Via Saragat 1,
44100 Ferrara, Italy,
friguzzi@ing.unife.it

Abstract

We propose a multi-strategy genetic algorithm for performing belief revision. The algorithm implements a new evolutionary strategy which is a combination of the theories of Darwin and Lamarck. Therefore, the algorithm not only includes the Darwinian operators of selection, mutation and crossover but also a Lamarckian operator that changes the individuals so that they perform better in solving the given problem. This is achieved through belief revision directed mutations, oriented by tracing logical derivations.

The algorithm, with and without the Lamarckian operator, is tested on a number of belief revision problems, and the results show that the addition of the Lamarckian operator improves the efficiency of the algorithm.

We believe that the combination of Darwinian and Lamarckian operators will be useful not only for standard belief revision problems but especially for problems where the chromosomes may be exposed to different constraints and observations. In these cases, the Lamarckian and Darwinian operators would play a different role: the Lamarckian one would be used in order to bring a chromosome closer to a solution or to find an exact solution of the current belief revision problem, while Darwinian ones will have the aim of preparing chromosomes to deal with new situations by exchanging genes among them.

1 Introduction

We consider a belief revision problem that consists in removing contradiction from an extended logic program [16, 2, 3] by changing the truth value of a set of literals called *revisables*. The program contains as well clauses with false ($\perp$) in the head representing *integrity constraints*. Any model of the program must make the body of integrity constraints false for the program to be non-contradictory. Contradiction may arise as well in the extended logic program because both a literal L and its opposite $\neg L$ are derivable in the theory. Such a problem has been widely studied in the literature and various solutions have been proposed [4, 9] that are based on logic proof procedures.

In this paper, we use evolutionary computation in order to solve the problem of belief revision. Such a problem can be modeled by means of a genetic algorithm by assigning to each revisable of the theory a gene in the chromosome. In the case of a two valued revision, the gene will have the value 1 if the corresponding revisable is true and the value 0 if the revisable is false. The fitness function that is used in this case is represented by the percentage of integrity constraints that are satisfied by a chromosome.

History of science has seen two evolution theories compete: Darwin's theory and Lamarck's theory. Darwin's theory is based on the concept of natural selection: only the individuals that are better for their environment survive, and are able to generate new individuals by means of reproduction. Moreover, during their life the individuals may be subject to random mutations of their genes that they can transmit to their offspring. Lamarck's theory, instead, states that evolution is due to the process of adaptation to the environment that an individual performs in his/her life. The abilities learned during the life of an individual modify his/her genes and are therefore transmitted to their offspring.

Experimental evidence in the biological kingdom has shown Darwin's theory to be correct and Lamarck's to be wrong, even if the more recently evolved concept of "meme" supports Lamarckian evolution (cf. [5]). However, in the field of genetic computation, Lamarckian evolution has proven to be a powerful concept and various authors have investigated the combination of Darwinian and Lamarckian evolution [12, 1, 14, 11].

In this paper, we propose a genetic algorithm for solving the problem of belief revision that includes, besides Darwin's operators of selection, mutation and crossover [15], a Lamarckian operator as well. This operator consists in modifying a chromosome (i.e., a set of revisables) so that it satisfies a higher number of constraints. This is achieved through belief revision directed mutations, oriented by tracing logical derivations. We then show, experimentally, that the resulting algorithm performs better than the same algorithm does without Lamarck's operator.

We believe the combination of Darwinian and Lamarckian operators will be useful not only for standard belief revision problems but especially for problems where the chromosomes may be exposed to different constraints and observations. For example, this may happen if an agent gathers information incrementally and has imperfect memory: in this case new data may arrive over time and old data may disappear. Or, alternatively, the available data may change because the external world is changing. Moreover, different agents may explore different parts of the world where constraints and observations are different. In this case, the Darwinian and Lamarckian operators may play different roles. Lamarckian operators may be used in order to get closer to solution of the belief revision problem given a fixed problem definition. Darwinian operators, instead, may be used in order to exchange genetic material among chromosomes of different agents so that they may be prepared in advance to new situations. For example, chromosomes exposed to constraints that disappeared may provide genes to new chromosomes that will become useful if the constraints reappear. Moreover, chromosomes belonging to an agent exploring a certain part of the world contain information that is useful for agents that will explore that part in the future.

In these cases, we could consider as well Lamarckian operators that not only bring a chromosome closer to a solution but actually find an exact solution of the belief revision problem: when a new constraint is presented to an agent, it first applies a Lamarckian operator to find a chromosome satisfying the new constraint and then it applies a Darwinian operator to distribute the "knowledge" so acquired to other chromosomes in the same agent or other agents. In this way chromosomes may be prepared in advance for meeting new constraints.

It is not our purpose to propose here a competitor to extant classical belief revision methods [4, 9], in particular as they apply to diagnosis. More ambitiously, we do propose a new approach for allowing belief revision – any assumption based belief revision – to deal with time/space distributed, and possibly intermittent or noisy observations about an albeit varying artifact or environment, by a multiplicity of agents which exchange genetically encoded diversified experience. About which more in the final section.

At the same time, we are employing belief revision, and accepted logic programming techniques, simply to illustrate an approach. The latter can be generally applied to accomplished Lamarckian learning, in settings where the trace of logical derivations can be used in order to backpropagate constraint violations into gene coded assumptions.

The paper is organized as follows. We first review some logic programming fundamentals and give a definition of a belief revision problem in section 2. Then, we describe the algorithm together with the Lamarckian operator in section 3. The results of experiments with the algorithm are shown in section 4. We examine related works in section 5, and we conclude in section 6.

2 Logic Programming Basis

In this section we first provide some logic programming fundamentals, and then we give a definition of the belief revision problem adapted from [19].

2.1 Language

Given a first order language *Lang*, an extended logic program [16, 2, 3] is a set of rules and integrity constraints of the form

$$H \leftarrow B_1, \ldots, B_n, not\ C_1, \ldots, not\ C_m \quad (m \geq 0, n \geq 0)$$

where $H, B_1, \ldots, B_n, C_1, \ldots, C_m$ are objective literals, and in integrity constraints H is $\perp$ (contradiction). An objective literal is either an atom A or its explicit negation $\neg A$, where $\neg\neg A = A$. $not\ L$ is called a default or negative literal. Literals are either objective or default ones. The default complement of objective literal L is $not\ L$, and of default literal $not\ L$ is L. A rule stands for all its ground instances with respect to *Lang*. The notation $H \leftarrow \mathcal{B}$ is also used to represent a rule, where set $\mathcal{B}$ contains the literals in its body. For every pair of objective literals $\{L, \neg L\}$ in *Lang*, we implicitly assume the constraint $\perp \leftarrow L, \neg L$.

The set of all objective literals of a program P is called its *extended Herbrand base* and is represented as $H^E(P)$. An *interpretation* I of an extended program P is denoted by $T \cup not\ F$, where T and F are disjoint subsets of $H^E(P)$. Objective literals in T are said to be *true* in I, objective literals in F are said to be *false* in I and those in $H^E(P) - I$ are said to be *undefined* in I. We introduce in the language the proposition $\mathbf{u}$ that is undefined in every interpretation I.

For Extended Logic Programs we consider the Extended Well Founded Semantics (*WFSX*) that extends the well founded semantics (*WFS*) [20] for normal logic programs to the case of extended logic programs. *WFSX* is obtained from *WFS* by adding the Coherence Principle (CP) relating the two forms of negation: "*if L is an objective literal and ¬L belongs to the model of a program, then also not L belongs to the model*", i.e., $\neg L \rightarrow not\ L$. See [2, 10] for a definition of *WFSX*.

We say that a set of literals S is *contradictory* iff $\perp \in S$.

2.2 Revising Contradictory Extended Logic Programs

Extended logic programs are liable to be contradictory because of integrity constraints, either those that are user-defined or those of the form $\perp \leftarrow L, \neg L$ that are implicitly assumed. Let us see an example of a contradictory program.

Example 2.1 Consider $P = \{a; \perp \leftarrow a, not\, b\}$[1]. Since we have no rules for b, by the Closed World Assumption *CWA*, it is natural to accept $not\, b$ as true. However, because of the integrity constraint, we can conclude $\perp$ and thus have contradiction.

It is arguable that the *CWA* may not be held of atom b since it leads to contradiction. Revising such *CWA* is the basis of the contradiction removal method of [19]. In order to select a particular contradiction removal process, three questions must be answered:

1. For which literals is the revision of their truth-value allowed ?
2. To what truth values do we change the revisable literals ?
3. How to choose among possible revisions ?

In section 2.4, two different contradiction removal processes are presented. They both use the same criteria to answer 1 and 3, but differ on the second one. The first way of removing contradiction in example 2.1 gives $\{a, not\, \neg a, not\, \neg b\}$ as the intended meaning of P, where b is revised to *undefined*. The second gives $\{a, b, not\, \neg a, not\, \neg b\}$, by revising b to true.

2.3 Contradictory Well Founded Model

To revise contradictions, we need to identify the contradictory sets of consequences implied by the applications of *CWA*. The main idea is to compute all consequences of the program, even if contradiction is found at some step. Furthermore, the Coherence Principle is enforced at each step. The following example provides an intuitive preview of what we intend:

Example 2.2 Consider program P:

$$\texttt{a}\leftarrow\texttt{not b.}\ \text{(i)} \qquad \neg\texttt{a}\leftarrow\texttt{not c.}\ \text{(ii)} \qquad \texttt{d}\leftarrow\texttt{a.}\ \text{(iii)} \qquad \texttt{e}\leftarrow\neg\texttt{a.}\ \text{(iv)}$$

1. $not\, b$ and $not\, c$ hold since there are no rules for either b or c.
2. $\neg a$ and a hold from 1 and rules (i) and (ii).
3. $\perp$ holds from 2 and implicit constraint $\leftarrow a, \neg a$.
4. $not\, a$ and $not\, \neg a$ hold from 2 and inference rule (CP).
5. d and e hold from 2 and rules (iii) and (iv).
6. $not\, d$ and $not\, e$ hold from 4, the CWA and rules (iii) and (iv), as they are the only rules for d and e.
7. $not\, \neg d$ and $not\, \neg e$ hold from 5 and inference rule (CP).

The whole set of consequences is

$$\{\perp, \neg a, a, not\, a, not\, \neg a, not\, b, not\, c, d, not\, d, not\, \neg d, e, not\, e, not\, \neg e\}$$

There exists a paraconsistent version of *WFSX*, called *WFSXp* [7, 8], that allows models to contain the atom $\perp$.

[1] $\perp \leftarrow a, \neg a$ and $\perp \leftarrow b, \neg b$ are implicitly assumed.

2.4 Contradiction Removal Sets

To revise contradiction, the first issue to consider is which default literals true by *CWA* are allowed to change their truth values. We simplify the approach of [18] along the lines of [17] taking as candidates for change default literals true by *CWA* in the *WFSXp* model. By making this simplification we can give a syntactic condition for electing the revisable literals, in contradistinction to the semantic one of [18].

Definition 2.1 (Revisables) *The revisables of a program P are all the default literals not L having no rules for L in P, and so true by CWA. The set of all revisables is called $Rev(P)$.*

Thus, all the objective literals that do not appear in rule heads but only in rule bodies, either in a positive or default form, are revisable. By the *CWA*, every revisable $R = not\ A$ is true, i.e., A is false. Now we identify the revisables that have to be revised to false or undefined in order to restore consistency. These are the ones that support contradiction. Intuitively, a support of a literal consists of the literals in nodes of a derivation for it:

Definition 2.2 (Support set of a literal) *A support set of a literal L of the* WFSXp *model M_P of a program P, denoted by $SS(L)$, is obtained as follows:*

1. *If L is an objective literal in M_P then for each rule $L \leftarrow \mathcal{B}$ in P, such that $\mathcal{B} \subseteq M_P$ there is one $SS(L)$ for each $B_i \in \mathcal{B}$. If $\mathcal{B}$ is empty then $SS(L) = \{\}$.*
2. *If L is a default literal not $A \in M_P$:*
   - *(a) if no rules exist for A in P (i.e., L is a revisable) then a support set of L is $\{not\ A\}$.*
   - *(b) if rules for A exist in P that have a non-empty body, then choose from each such rule a single literal such that its default complement belongs to M_P. There exists one SS for not A for every SS of each default complement of the chosen literals.*
   - *(c) if $\neg A$ belongs to M_P then there exist,* additionally*, support sets SS of not A equal to each $SS(\neg A)$.*

Example 2.3 The *WFSXp* model M_P of:

```
¬p←not c.   p←t.          b←c, a.   ¬b←not e.   a.
            p←a,not b.    b←d.
```

is $\{a, not\ \neg a, not\ b, \neg b, not\ c, not\ \neg c, not\ d, not\ \neg d, not\ e, not\ \neg e, not\ t, not\ \neg t, p, \neg p, not\ p, not\ \neg p, \bot\}$. The revisables are $\{not\ c, not\ d, not\ e\}$. There are two support sets for *not b*:

$$SS_1(not\ b) = SS(not\ c) \cup SS(not\ d) \qquad by\ rule\ 2b$$
$$SS_1(not\ b) = \{not\ c\} \cup \{not\ d\} = \{not\ c, not\ d\} \qquad by\ rule\ 2a$$

Notice that the other possibility of choosing literals for $SS(not\ b)$, i.e. $SS_1(not\ b) = SS(not\ a) \cup SS(not\ d)$, can't be considered because *not a* doesn't belong to M_P. The other support set for *not b* is obtained using rule 2c:

$$SS_2(not\ b) = SS(\neg b) \qquad by\ rule\ 2c$$
$$SS_2(not\ b) = SS(not\ e) \qquad by\ rule\ 1$$
$$SS_2(not\ b) = \{not\ e\} \qquad by\ rule\ 2a$$

Now the support sets for the objective literal p are easily computed:

$$SS(p) = SS(a) \cup SS(not\ b) \quad by\ rule\ 1$$
$$SS(p) = \{\} \cup SS(not\ b) \quad by\ rule\ 1\ (the\ only\ rule\ for\ a\ is\ fact\ a)$$

So $SS_1(p) = SS_1(not\, b) = \{not\, c, not\, d\}$ and $SS_2(p) = SS_2(not\, b) = \{not\, e\}$. $\neg p$ has the unique support set $\{not\, c\}$. Consequently, because contradiction is obtained only via $\perp \leftarrow p, \neg p$, $SS_1(\perp) = \{not\, c, not\, d\}$ and $SS_2(\perp) = \{not\, e, not\, c\}$.

Proposition 2.1 (Existence of support sets) *Every literal L belonging to the* WFSXp *model of a program P has at least one support set $SS(L)$.*

We define a spectrum of possible revisions using the notion of hitting set:

Definition 2.3 (Hitting set) *A hitting set of a collection C of sets is formed by the union of one non-empty subset from each $S \in C$. A hitting set is minimal iff no proper subset is a hitting set. If $\{\} \in C$, C has no hitting sets.*

Definition 2.4 (Removal set) *A removal set of a literal L of a program P is a hitting set of all support sets $SS(L)$.*

We can revise contradictory programs by changing the truth value of all the literals of a removal set of $\perp$. The truth value can be changed either to undefined or false. It can be changed to undefined by adding, for each literal *not* L in the removal set, the inhibition rule $L \leftarrow not\, L$ to P, while it can be changed to false by adding L to P. In case the literals are revised to undefined, then the contradiction is removed and no new contradiction can arise. In case they are revised to false, a new contradiction may arise and therefore the contradiction removal process must be iterated. This defines the possible revisions of a contradictory program.

We answer the third question by preferring to revise minimal sets of revisables:

Definition 2.5 (Contradiction removal set) *A contradiction removal set (CRS) of P is a minimal removal set of $\perp$.*

Example 2.3 (cont.) The assumption support sets of $\perp$ are $\{not\, c, not\, d\}$ and $\{not\, c, not\, e\}$. The removal sets are (RS_1 and RS_4 being minimal):

$$\begin{array}{ll} RS_1 = \{not\, c\} & RS_4 = \{not\, d, not\, e\} \\ RS_2 = \{not\, c, not\, e\} & RS_5 = \{not\, c, not\, d, not\, e\} \\ RS_3 = \{not\, c, not\, d\} & \end{array}$$

A program is not revisable if $\perp$ has a support set without revisable literals.

Definition 2.6 (Revisable program) *A program is revisable iff it has a contradiction removal set.*

The $CRSs$ are minimal hitting sets of the collection of assumptions sets of $\perp$. In [19] an algorithm for solving this problem is presented.

In this paper, we consider a simplified belief revision problem where the program does not contain explicit negation and the only default literals that are allowed are revisables, i.e., default literals of predicates that have no definition. In this case, the *WFSXp* model of the program is two valued, i.e., it assigns to every literal the truth value true or false. Therefore, we change the truth value of revisables from true to false but not to undefined. In the future, we will consider three-valued revisions of full extended logic programs.

3 A genetic algorithm for belief revision

The algorithm we propose extends the standard genetic algorithm (described for example in [15]) by adding a Lamarckian operator called *Learn*. This operator modifies a fraction of the chromosomes of the population in order to improve their fitness. The resulting algorithm[2] is given as follows:

GA($Fitness, max\_gen, p, r, m, l$)

input :

$Fitness$: a function that assigns an evaluation score, given a hypothesis

max_gen : the maximum number of generations before termination

p: number of individuals in the population

r: the fraction of population to be replaced by Crossover at each step

m: the fraction of population to be mutated

l: the fraction of population that should evolve Lamarckianly

output :

h_{max} : the hypothesis with the highest fitness

Initialize population: $P \leftarrow$ generate p hypotheses at random

Evaluate: for each h in P, compute $Fitness(h)$

$gen \leftarrow 0$

While $gen \leq max\_gen$

Create a new population P_s:

Select: Probabilistically select $(1-r)p$ members of P to add to P_s.

The probability $Pr(h_i)$ of selecting hypothesis h_i from P is given by

$Pr(h_i) = \frac{Fitness(h_i)}{\Sigma_{j=1}^{P} Fitness(h_j)}$

Crossover: Probabilistically select $rp/2$ pairs of hypotheses from P, according to $Pr(h_i)$ given above.

For each pair $\langle h_1, h_2 \rangle$, produce two offspring by applying the Crossover operator. Add all offspring to P_s

Mutate: Choose m percent of the members of P_s with uniform probability. For each, invert one randomly selected bit in its representation

Learn: Choose lp hypotheses from P_s with uniform probability and substitute each of them with the modified hypotheses returned by the procedure *Learn*

Update: $P \leftarrow P_s$

Return the hypothesis h_{max} from P with the highest fitness

In belief revision, each individual hypothesis is described by the truth value of all the revisables. Since we consider two valued models, each hypothesis gives the truth value true or false to every revisable and therefore it can be considered as a set containing one literal, either objective or default, for every revisable. A chromosome is obtained by associating a bit to each revisable that has value 1 if the revisable must be true and 0 if it must be false.

[2]The algorithm has been implemented in Sicstus Prolog 3#5.

The fitness function that has been used takes the following form:

$$Fitness(h_i) = \frac{n_i}{n} + \frac{f_i}{|h_i|} \times 0.5$$

where n_i is the number of integrity constraints satisfied by hypothesis h_i, n is the total number of integrity constraints, f_i is the number of revisables in h_i that are false, and $|h_i|$ is the total number of revisables. In this way, the fitness function takes into account both the fraction of constraints that are satisfied and the number of revisables whose truth value must be changed to true, preferring hypotheses with a lower number of these, that is minimal revisions are encouraged. The factor 0.5 was chosen in order to give more importance to the accuracy, rather than to the number of unchanged revisables.

The Lamarckian operator $Learn$ changes the values of the revisables in a chromosome C so that a bigger number of constraints is satisfied, thus bringing C closer to a solution. $Learn$ differs from a normal belief revision operator because it does not assume that all revisables are true by CWA before the revision but it starts from the truth values that are given by the chromosome C. Therefore, it has to revise some revisables from true to false and others from false to true. As a consequence, the support set does not contain only default literals but also objective literals whose default complement is a revisable.

$Learn$ works in the following way: given a chromosome C, it finds all the support sets for $\perp$ such that they contain literals in C. Therefore, it does not find all support sets for $\perp$ but only those that are subsets of C.

The definition of support set that is used by the Lamarckian operator is therefore different from definition 2.2 and is given as follows:

Definition 3.1 (Lamarckian support set of a literal) *A support set of a literal L of the* WFSXp *model M_P of a program P according to a given set of literals H is denoted by $SS(L,H)$ and is obtained as follows:*

1. *If L is an objective literal in M_P then for each rule $L \leftarrow \mathcal{B}$ in P, such that $\mathcal{B} \subseteq M_P$ there is one $SS(L,H)$ for each $B_i \in \mathcal{B}$. If $\mathcal{B}$ is empty then $SS(L,H) = \{\}$.*
2. *If L is an objective literal in M_P for which there are no rules $L \leftarrow \mathcal{B}$ in P and L is the default complement of a revisable, then*
   (a) *if L belongs to H, then a support set of L is $\{L\}$.*
   (b) *if the default complement of L belongs to H, then there is no support set for L.*
3. *If L is a default literal $not\ A \in M_P$:*
   (a) *if no rules exist for A in P (i.e., L is a revisable) then:*
      i. *if L belongs to H, then a support set of L is $\{not\ A\}$.*
      ii. *if A belongs to H, then there is no support set for L.*
   (b) *if rules for A exist in P that have a non-empty body, then choose from each such rule a single literal such that its default complement belongs to M_P. There exists one SS for $not\ A$ for every SS of each default complement of the chosen literals.*
   (c) *if $\neg A$ belongs to M_P then there exist,* additionally, *support sets SS of $not\ A$ equal to each $SS(\neg A)$.*

Since the Lamarckian support sets for $\perp$ represent only a subset of all the support sets for $\perp$, a hitting set generated from them is not necessarily a contradiction removal set and therefore it does not represent a solution to the belief revision problem. However, it eliminates some of the derivation paths to $\perp$ and, therefore, may increase the number of satisfied constraints, thus improving the fitness, as required by the notion of Lamarckian operator.

The following algorithm implements the Lamarckian operator. For simplicity, we present a restricted version of the algorithm that deals with programs that contain neither explicitly negated literals nor the default negation of non revisable literals. This restricted version will however be sufficient for dealing with the experiments presented in section 4.

procedure $Learn(C, C')$

input :

a chromosome C translated into a set of revisables

output :

a revised chromosome C'

Find the support sets for $\perp$: $Support\_sets([\perp], C, \{\}, \{\}, SS)$

Find a hitting set HS: $Hitting\_set(SS, HS)$

Change the value of the literals in the chromosome C

that appear as well in HS obtaining C'

procedure $Support\_sets(GL, C, S, SSin, SSout)$:

input :

a list of goals GL

a chromosome C translated into a set of revisables

the current support set S

the current set of support sets $SSin$

output :

a set $SSout$ containing the support sets

for the first goal in the list

If GL is empty, then return $SSout = SSin$

Consider the first literal L of the first goal G of GL

($GL = [[L|RG]|RGL]$ using Prolog notation for lists)

(1) if G is empty then add the current support set to $SSin$ and call

recursively the algorithm on the rest of GL

$Support\_sets(RGL, C, \{\}, SSin \cup \{S\}, SSout)$

(2) if L is a revisable or is the default complement of a revisable and is in C, then add it to S and call the

algorithm recursively on the rest of G

$Support\_sets([RG|RGL], C, S \cup \{L\}, SSin, SSout)$

(3) if L is a revisable or is the default complement of a revisable and it is not in C, or its opposite is in C, then

discard S and call the algorithm recursively on the rest of GL

$Support\_sets(RGL, C, \{\}, SSin, SSout)$

(4) if L is not a revisable nor the default complement of a revisable then reduce it with all the rules obtaining

the new goals $G_1, ..., G_n$, one for each matching rule,

add the goals to GL and call the algorithm recursively

$Support\_sets([[G_1|RG], ..., [G_n|RG]|RGL], C, S, SSin, SSout)$

(5) if L is not a revisable nor the default complement of a revisable and there are no rules for it, then discard S and call the algorithm recursively on the rest of GL
$Support\_sets(RGL, C, \{\}, SSin, SSout)$

procedure *hitting_set*(SS, HS):

input :

a set of support sets SS

output :

a hitting set HS

pick a literal from every support set in SS
and add it to HS if it does not lead to contradiction
(i.e. the literal must not be already present in its complemented form).
If it leads to contradiction pick another literal.

Thus the Lamarckian operator differs both from a full belief revision operator and from a purely Darwinian operator. It differs from a full belief revision operator because it does not directly find a solution of the given belief revision problem but rather finds a hypothesis that is closer to a solution than the starting one. It differs from a purely Darwinian operator because it does not change the current hypothesis blindly but changes it in order to improve its fitness. In the next section, after having introduced the application of belief revision to digital circuit diagnosis, the difference between a full belief revision operator and the Lamarckian operator will be further explained by means of an example.

4 Experiments

The system has been tested on a number of belief revision problems in order to show the effectiveness of the Lamarckian operator. In particular, we have considered problems of digital circuit diagnosis, as it is done in [9]. Figure 1 shows a sample circuit. In problems of digital circuit diagnosis, we have a difference between the

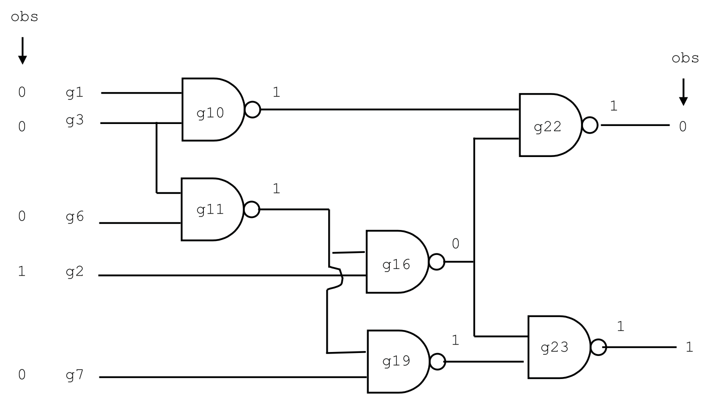

Figure 1: `c17` benchmark circuit from ISCAS85 [6].

observed and the predicted outputs. Figure 1 shows the observed inputs and outputs of the circuit together with the predicted outputs of each gate. The aim of the diagnosis is to find which components are faulty.

A problem of digital circuit diagnosis can be modelled as a belief revision problem by describing it with a logic program consisting of four groups of clauses: one that allows to compute the predicted output of each component, one that describes the topology of the circuit, one that describes the observed inputs and outputs, and one that consists of integrity constraints stating that the predicted value for an output of the system can not be different from the observed value. The representation formalism we use is the one of [9]. As regards the clauses for computing the predicted output of a gate, let us consider the case of a two-input NAND:

```
val( out(nand2,Name), V ) :-
  not ab(Name),
  val( in(nand2,Name,1), W1),
  val( in(nand2,Name,2), W2),
  nand2_table(W1,W2,V).
val( out(nand2,Name), V ) :-
  ab(Name),
  val( in(nand2,Name,1), W1),
  val( in(nand2,Name,2), W2),
  and2_table(W1,W2,V).
```

In these clauses `Name` is the name of the component and the definition of `nand2_table, and2_table` consist of facts describing the input/output relation of, respectively, a two-input NAND gate and a two-input AND gate. `ab(Name)` is a revisable that can be assumed true or false. If it is assumed true, it expresses a faulty behaviour of a component of the circuit, described in this case by the second clause above. If it is assumed false, it expresses a correct behaviour of a component of the circuit, being described by the first clause above.

The topology of the circuit is described by a set of facts for the predicate `conn/2`. For example, consider the fact `conn(in(nand2, g10, 1), out(inpt0, g1))` describing a part of the circuit shown in figure 1. This fact states that the input 1 of gate `g10` of type `nand2` is connected to the output of gate `g1` of type `inpt0`. The gates of type `inpt0` are the input pins of the circuit.

The clauses that describe the observed values for the input and for the output of the circuit are facts for the `obs/2` predicate. `obs(out(inpt0, g1), 0)` states that the input `g1` has value 0.

As regards the integrity constraints, we have two constraints for each output of the circuit, one stating that the output can not be 0 if it was observed to be 1 and the other stating that the output can not be 1 if it was observed to be 0. For example, the constraint `ic([obs(out(nand2, g22), 0), val(out(nand2, g22), 1)]).` states that the value of the output of `g22` cannot be 1 if it was observed to be 0.

In case the circuit is faulty, one or more of the constraints will be violated. By means of belief revision, the values of the revisables are changed in order to restore consistency. The literals of the form `ab(Name)` that are assigned the value true identify the faulty components.

In order to show the difference between a belief revision operator and the Lamarckian operator, let us show their behaviour on the `c17` circuit supposing the following hypothesis is given:

```
C={ab(g10), not ab(g11), ab(g16), not ab(g19), not ab(g22), not ab(g23)}
```

In this case, two out of four constraints are violated because the predicted output of `g22` is 1 and of `g23` is 0.

A belief revision operator, as for example REVISE, finds a solution where the only abnormality literal that is true is `ab(g22)` while all the others are false. This solution is found independently of the initial starting hypothesis

Circuit	l	Fitness	Standard Deviation	Correct solution
voter	0	1.295	0.00634	100 %
	0.6	1.312	0.01728	100 %
alu4_flat	0	1.193	0.03939	20 %
	0.6	1.213	0.01765	33 %

Table 1: Experiments on digital circuits debugging

because the support sets for $\perp$ are found independently of the initial values of the revisables. This new hypothesis eliminates both constraint violations.

The Lamarckian operator, instead, will modify `C` into the following hypothesis:

```
C'={ab(g10), ab(g11), ab(g16), not ab(g19), not ab(g22), not ab(g23)}
```

that differs from `C` only in the values of `ab(g11)`. This new hypothesis eliminates only one constraint violation because the output of `g22` is still different from the observed value.

The system has been tested on some real world problems taken from the ISCAS85 benchmark circuits [6] that has been used as well for testing the belief revision system REVISE [9].[3] Our aim was to prove that the Lamarckian operator provides an improvement over a purely Darwinian algorithm. Moreover, we wanted to investigate how the initial population influences the computation. In fact, since the Lamarckian operator has an effect that greatly depends on the initial hypothesis on which it is applied, our algorithm may run the risk of being highly dependent on the initial population. In the worst case, it could happen that the algorithm, guided by the greediness of the Lamarckian operator, finds a local optimum and does not move from there.

We have considered the `voter` and `alu4_flat` circuits: `voter` has 59 gates and 4 outputs, corresponding respectively to 59 revisables and 8 constraints, while `alu4_flat` has 100 gates and 8 outputs, corresponding respectively to 100 revisables and 16 constraints. The system was first run without the Lamarckian operator ($l = 0$), and then using it ($l = 0.6$). The other parameters of the genetic algorithms were 30 for the population and 10 for the number of generations. For each case, the system was run five times and the resulting maximum fitness averaged. In table 1 the Fitness column shows the value of the fitness function for the best hypothesis after ten generations averaged over the five runs together with its standard deviation, while the Correct solution column shown the percentage of times in which a correct solution was found.

From these results we can state that the use of a Lamarckian operator improves the fitness of the best hypothesis. Moreover, the algorithm does not heavily depend on the initial population, as it is shown by the low values for the standard deviation. Finally, the Lamarckian operator does not greatly influence the dependency on the initial population, as can be seen from the fact that in one case (`voter`) the use of the Lamarckian operator has increased the standard deviation but in the other case (`alu4_flat`) it has decreased it.

5 Related Work

Various authors have investigated the integration of Darwinian and Lamarckian evolution in genetic algorithm [1, 11, 12, 14]. A Lamarckian operator first translates a genotype into its corresponding phenotype and performs

[3] These examples can be found at `http://www.soi.city.ac.uk/~msch/revise/`.

a local search in the phenotype's space. The local optimum that is obtained is then translated back into its corresponding genotype and added to the population for further evolution. [12] has shown that the traditional genetic algorithm performs well for searching widely separated portions of the search space caused by a scattered population, while Lamarckism is more proficient for exploring localized areas of the population that would otherwise be missed by the global search of the genetic algorithm. Therefore, Lamarckism can play an important role when the population has converged to areas of local maxima that would not be thoroughly explored by the standard genetic algorithm. The adoption of a Lamarckian operator provides a significant speedup in the performance of the genetic algorithm.

Similarly to the approaches in [1, 11, 12, 14], we adopt a procedure for Lamarckian evolution that first translate the chromosome into its phenotype and then modifies it in order to improve its fitness. In our case as well the Lamarckian operator improves the performance of the genetic algorithm. Differently from [1, 11, 12, 14], the procedure does not perform a local search but finds an improvement by tracing logical derivations.

6 Conclusions and Future Work

We have presented a multistrategy genetic algorithm for performing belief revision. The algorithm combines two different evolution strategies, one based on Darwin's theory and the other based on Lamarck's theory. The algorithm therefore includes, besides Darwin's evolutionary operators of selection, mutation and crossover, also a Lamarckian operator that changes the genes of an individual in order to improve his/her fitness.

While Darwin's operators are independent of the task at hand, Lamarck's operator necessarily depends on the task. In this case, we have considered the problem of revising the beliefs of a contradictory theory in order to restore consistency. The beliefs to be revised are those generated by means of the CWA. In this case the process of belief revision consists in assigning a truth value to the default literals $notL$ having no rule for L in the program.

In this paper the operator is implemented by means of a belief revision procedure that, by tracing logical derivations, identifies the genes leading to contradiction. The overall algorithm has been tested on a number of problems of circuit diagnosis. The results of the tests show that the Lamarckian operator improves the fitness of the hypothesis that is found by the algorithm after a fixed number of generations.

In this paper, we have considered only two valued belief revisions of a restricted set of extended logic programs, i.e. programs that do not contain explicit negation and that allow only default literals that are revisable. In the future, we will consider three valued revisions of full extended logic programs.

Moreover, we will investigate the case in which different individuals are exposed to different experiences. This may happen because the world surrounding an agent changes over time or because we consider agents exploring different parts of the world. In this case, Lamarckian and Darwinian operators will have complementary functions: Lamarckian operators will be used to get closer to a solution of a given belief revision problem, while Darwinian operators will be used in order to distribute the acquired knowledge among various individuals. We could consider as well Lamarckian operators that not only bring a chromosome closer to a solution but actually turn it into a solution. In this case, when a new constraint is presented to an agent, it first applies a Lamarckian operator to find a chromosome satisfying the new constraint and then it applies a Darwinian operator to distribute the "knowledge" so acquired to other chromosomes in the same agent or other agents. In this way chromosomes may be prepared in advance for meeting new constraints.

An example of this would be that of a group of scientists that decides to classify living forms in nature by travelling each to a different part of the world. Each of them has a partial view of nature, with a limited set of (noisy) observations, and he/she can come up with a consistent life taxonomy by exchanging chromosomes with the other scientists.

The exchange of genetic material is useful also in the case in which the chromosomes do not have all the relevant revisables to start with (three valued revision). When they acquire new revisables from other chromosomes, they are obtaining specialized knowledge from others. This, is for example, the case of the diagnosis of a car fault performed by different experts: the expert mechanic, the expert electrician, the expert car designer, etc. Each of them makes a diagnosis about the part of the car that concerns their speciality. Next they all have to come to a joint diagnosis by exchanging information about each other's revisables.

We conjecture that in this new problem setting, where there is dynamicity in the data, the integration of the Lamarckian and Darwinian operators will fully exhibit, and be extolled, in its potential. A first study of this approach can be found in [13].

7 Acknowledgements

L. M. Pereira acknowledges the support of PRAXIS project MENTAL "An Architecture for Mental Agents. E. Lamma and F. Riguzzi acknowledge the support of the MURST project "Intelligent Agents: Interaction and Knowledge Acquisition".

References

[1] D. H. Ackely and M. L. Littman. A case for lamarckian evolution. In C. G. Langton, editor, *Artificial Life III*. Addison Wesley, 1994.

[2] J. J. Alferes and L. M. Pereira. *Reasoning with Logic Programming*, volume 1111 of *LNAI*. Springer-Verlag, 1996.

[3] J. J. Alferes, L. M. Pereira, and T. C. Przymusinski. "Classical" negation in non-monotonic reasoning and logic programming. *Journal of Automated Reasoning*, 20:107–142, 1998.

[4] J. J. Alferes, L. M. Pereira, and T. Swift. Well-founded abduction via tabled dual programs. In D. De Schreye, editor, *Procs. of the 16th International Conference on Logic Programming*, pages 426–440, Las Cruces, New Mexico, 1999. MIT Press.

[5] Susan Blackmore. *The Meme Machine*. Oxford U.P., Oxford, UK, 1999.

[6] F. Brglez, P. Pownall, and R. Hum. Accelerated ATPG and fault grading via testability analysis. In *Proceedings of IEEE Int. Symposium on Circuits and Systems*, pages 695–698, 1985. The ISCAS85 benchmark netlist are available via ftp `mcnc.mcnc.org`.

[7] C. V. Damásio and L. M. Pereira. Abduction on 3-valued extended logic programs. In V. W. Marek, A. Nerode, and M. Trusczynski, editors, *Logic Programming and Non-Monotonic Reasoning - Proc. of 3rd International Conference LPNMR'95*, volume 925 of *LNAI*, pages 29–42, Germany, 1997. Springer-Verlag.

[8] C. V. Damásio and L. M. Pereira. A survey on paraconsistent semantics for extended logic programs. In D.M. Gabbay and Ph. Smets, editors, *Handbook of Defeasible Reasoning and Uncertainty Management Systems*, volume 2, pages 241–320. Kluwer Academic Publishers, 1998.

[9] C. V. Damásio, L. M. Pereira, and M. Schroeder. REVISE: Logic programming and diagnosis. In *Proceedings of Logic-Programming and Non-Monotonic Reasoning, LPNMR'97*, volume 1265 of *LNAI*, Germany, 1997. Springer-Verlag.

[10] J. Dix, L. M. Pereira, and T. Przymusinski. Prolegomena to logic programming and non-monotonic reasoning. In J. Dix, L. M. Pereira, and T. Przymusinski, editors, *Non-Monotonic Extensions of Logic Programming - Selected papers from NMELP'96*, number 1216 in LNAI, pages 1–36, Germany, 1997. Springer-Verlag.

[11] J. J. Grefenstette. Lamarckian learning in multi-agent environment. In *Proc. 4th Intl. Conference on Genetic Algorithms*. Morgan Kauffman, 1991.

[12] W. E. Hart and R. K. Belew. Optimization with genetic algorithms hybrids that use local search. In R. K. Belew and M. Mitchell, editors, *Adaptive Individuals in Evolving Populations*. Addison Wesley, 1996.

[13] E. Lamma, F. Riguzzi, and L. M. Pereira. Multi-agent logic aided lamarckian learning. Technical report, Dipartimento di Ingegneria - University of Ferrara, 2000.

[14] Y. Li, K. C. Tan, and M. Gong. Model reduction in control systems by means of global structure evolution and local parameter learning. In D. Dasgupta and Z. Michalewicz, editors, *Evolutionary Algorithms in Engineering Applications*. Springer Verlag, 1996.

[15] T. M. Mitchell. *Machine Learning*. McGraw Hill, 1997.

[16] L. M. Pereira and J. J. Alferes. Well founded semantics for logic programs with explicit negation. In *Proceedings of the European Conference on Artificial Intelligenece ECAI92*, pages 102–106. John Wiley and Sons, 1992.

[17] L. M. Pereira, J. J. Alferes, and J. N. Aparício. Contradiction removal within well founded semantics. In A. Nerode, editor, *Proc. Logic Programming and Non-Monotonic Reasoning '91*, pages 105–119. MIT Press, 1991.

[18] L. M. Pereira, J. J. Alferes, and J. N. Aparício. Contradiction removal semantics with explicit negation. In M. Masuch and L. Polos, editors, *Knowledge Representation and Reasoning Under Uncertainty*, number 808 in LNAI, pages 91–106. Springer-Verlag, 1994.

[19] L. M. Pereira, C. V. Damásio, and J. J. Alferes. Diagnosis and debugging as contradiction removal. In L. M. Pereira and A. Nerode, editors, *Proceedings of the 2nd International Workshop on Logic Programming and Non-monotonic Reasoning*, pages 316–330. MIT Press, 1993.

[20] A. Van Gelder, K. A. Ross, and J. S. Schlipf. The well-founded semantics for general logic programs. *Journal of the ACM*, 38(3):620–650, 1991.