Belief Revision by Multi-Agent Genetic Search

Evelina Lamma
Dipartimento di Ingegneria,
Università di Ferrara, Via Saragat 1,
44100 Ferrara, 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

The revision of beliefs is an important general purpose functionality that an agent must exhibit. The agent usually needs to perform this task in cooperation with other agents, because access to knowledge and the knowledge itself are distributed in nature.

In this work, we propose a new approach for performing belief revision in a society of logic-based agents, by means of a (distributed) genetic algorithm, where the revisable assumptions of each agent are coded into chromosomes as bit-strings. Each agent by itself locally performs a genetic search in the space of possible revisions of its knowledge, and exchanges genetic information by crossing its revisable chromosomes with those of other agents.

We have performed experiments comparing the evolution in beliefs of a single agent informed of the whole of knowledge, to that of a society of agents, each agent accessing only part of the knowledge. In spite that the distribution of knowledge increases the difficulty of the problem, experimental results show that the solutions found in the multi-agent case are comparable in terms of accuracy to those obtained in the single agent case.

The genetic algorithm we propose, besides encompassing the Darwinian operators of selection, mutation and crossover, also comprises a Lamarckian operator that mutates the genes in a chromosome as a consequence of the chromosome phenotype's individual experience obtained while solving a belief revision problem. These chromosomic mutations are directed by a logic-based belief revision procedure that relies on tracing the logical derivations leading to inconsistency of belief, so as to remove these derivations' support on the gene coded assumptions, effectively by mutating the latter. Because of the use a Lamarckian operator, and following the literature, the genes in these chromosomes that are modified by the Lamarckian operator are best dubbed "memes", since they code the memory of the experiences of an individual along its lifetime, besides being transmitted to its progeny.

We believe our method to be important for situations where classical belief revision methods hardly apply: those where environments are non-uniform and time changing. These can be explored by distributed agents that evolve genetically to accomplish cooperative belief revision, if they use our approach.

1 Introduction

Belief revision is indeed an important functionality that agents must exhibit: agents should be able to modify their beliefs in order to model the outside world. Moreover, they need to perform this task in cooperation with other agents, because access to knowledge and the knowledge itself are distributed in nature, i.e., each agent has only a partial knowledge of the world.

We consider a definition of the belief revision problem that consists in removing a contradiction from an extended logic program [15, 2, 3] by modifying the truth value of a selected set of literals called *revisables*. The program contains as well clauses with false ($\bot$) in the head, representing *integrity constraints*. Any model of the program must ensure that the body of integrity constraints be false for the program to be non-contradictory. Contradiction may also arise in an extended logic program when both a literal L and its opposite $\neg L$ are obtainable in the model of the program. Such a problem has been widely studied in the literature, and various solutions have been proposed [4, 9] that are based on

abductive logic proof procedures.

In this work, we propose a new approach for performing belief revision in a society of logic-based agents, by means of a (distributed) genetic algorithm. The problem can be modeled by means of a genetic algorithm, by assigning to each revisable of a logic program a gene in a 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 in part by the percentage of integrity constraints that are satisfied by a chromosome.

Each agent keeps a population of chromosomes and finds a solution to the revision problem by means of a genetic algorithm. We consider a formulation of the revision problem where each agent has the same set of revisables and the same program, but is subjected to possibly different observations and constraints. Observations and constraints may vary over time, and can differ from agent to agent because agents may explore different regions of the world. Each agent by itself locally performs a genetic search in the space of possible revisions of its knowledge, and exchanges genetic information by crossing its revisable chromosomes with those of other agents. In this way, we achieve distribution in belief revision since chromosomes coming from different agents, through crossover, contribute to solve the problem.

In the genetic algorithm we also exploit computational logic techniques: the algorithm, comprises a Lamarckian operator that differs from a Darwinian mutation operator because, instead of randomly modifying the genes, it modifies them in order to improve the fitness of the chromosome. Genes that are modified by this operator are also called "memes" [5]. The Lamarckian operator modifies the memes by means of a (logic-based) procedure inspired by [16]: the logical derivations leading to the inconsistency of belief are traced so as to remove these derivations' support on the meme coded assumptions, effectively by mutating the latter. In our algorithm, therefore, computational logic is used in order to find good revisions that are then distributed by means of the crossover genetic operator.

We have performed experiments comparing the evolution in beliefs of a single agent informed of the whole of knowledge, to that of a society of agents, each agent accessing only part of the knowledge. The experiments have been performed on problems of model based diagnosis, a natural domain in which belief revision techniques apply [9], and on the n-queen problem. In spite that the distribution of knowledge increases the difficulty of the problem, experimental results show that the solutions found in the multi-agent case are comparable in terms of accuracy to those obtained in the single agent case.

Moreover, we have seen that the adoption of computational logic methods in a genetic algorithm provides an improvement over purely genetic approaches.

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 [16].

2.1 Language

Given a first order language *Lang*, an extended logic program [15, 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 $\bot$ (false). 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 the 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 $\bot \leftarrow L, \neg L$.

The set of all objective literals of a program P is called its *extended Herbrand base* and is represented as $H^E(P)$.

We consider the Extended Well Founded Semantics (*WFSX*) that extends the well founded semantics (*WFS*) [17] for normal logic programs to programs extended with explicit negation, besides the implicit or default negation of normal 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 $\neg L$ belongs to the model of a program, then also not L belongs to the model*", i.e., $\neg L \rightarrow not\ L$. See [2, 11] or the Appendix for a definition of *WFSX*.

We say that a set of literals S is *contradictory* iff $\bot \in S$. The paraconsistent version of *WFSX*, that allows models to contain the atom $\bot$, is called *WFSXp* [7, 8].

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 $\bot \leftarrow L, \neg L$ that are implicitely assumed. Let us see an example of a contradictory program.

Example 2.1 Consider $P = \{a; \bot \leftarrow a, not\, b\}$[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 $\bot$ and thus have contradiction.

It is arguable that the *(CWA)* may not be held of atom b since it leads to contradiction. Revising such *CWAs* is the basis of the contradiction removal method of [16]. In order to select a particular contradiction removal process, three questions must be answered:

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

The options taken here are clarified in the discussion in section 2.4, giving two different answers to these questions. Both use the same criteria to answer 1 and 3, but differ on the second one. For example 2.1 the first way of removing contradiction gives $\{a, not\, \neg a, not\, \neg b\}$ as the intended meaning of P, where b is revised to *undefined*, achievable by adding $b \leftarrow not\, b$ to P. The second gives $\{a, b, not\, \neg a, not\, \neg b\}$, by revising b to true, achievable by adding b to P.

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 those leading to contradictions, as well as those arising from contradictions. Furthermore, the coherence principle is enforced at each step.

Example 2.2 Consider program P:

`a←not b.` (i) `¬a←not c.` (ii) `d←a.` (iii) `e←¬a.` (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. $\bot$ 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 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 the *WFSXp* model:

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

2.4 Contradiction Removal Sets

To abolish contradiction, the first issue to consider is which default literals true by *CWA* are allowed to change their truth values. We adopt the approach of [16] where the candidates for revision are all the objective literals that have no rules in the program. By CWA, their default negation is true. These literals are called *revisables*.

Definition 2.1 ***Revisables*** *The revisables of a program P are the elements of a chosen subset $Rev(P)$, of the set of all objective literals L having no rules for them in P.*

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

Definition 2.2 ***Set of assumptions supporting a literal*** *A support set (of assumptions) of a literal L of the* WFSXp *model M_P of a program P, denoted by $SS(L)$, with respect to the set of revisable $Rev(P)$ 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 $SS(L)$ formed by the union of a SS 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 for A exist in P 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 $not\, A$ which is the union of one SS for the default complement of the chosen literal in each rule.*
   (c) *if $\neg A$ belongs to M_P then there exist,* additionally, *support sets SS for $not\, A$ equal to each $SS(\neg A)$.*

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

The definitions of revisable literals and of support sets differ from those given in [16] because there the revisables are the default complement of the literals without definition and support sets there contain all the literals in the nodes of a derivation for L. We have provided these modified definitions because they simplify the introduction of the Lamarckian operator in the next section.

Example 2.3 The *WFSXp* model M_P of:

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

is $\{a, not\ \neg a, not\ b, \neg b, not\ c, not\ \neg c, not\ d, not\ \neg d, not\ e, not\ \neg e, not\ t,\ not\ \neg t, p, \neg p, not\ p, not\ \neg p, \bot\}$.

Here the revisables are $\{c, d, e\}$. There are two support sets for $not\ b$:

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

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:

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

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 $SS_1(p) = SS1(not\ b) = \{not\ c, not\ d\}$ 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 $\bot \leftarrow p, \neg p$, $SS_1(\bot) = \{not\ c, not\ d\}$ and $SS_2(\bot) = \{not\ e, not\ c\}$.

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 $\{\} \in C$, then 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 the literals of some removal set of $\bot$. 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 (making L effectively undefined), 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 this (convergent) contradiction removal process must be iterated. This defines the possible revisions of a contradictory program.

We answer the second question above by considering only a two-valued revisions, i.e. where the truth value of a revisable can only be changed to true or false. We answer the third question by prefering 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 $\bot$.*

Example 2.3 (cont.) The support sets of $\bot$ are $\{not\ c, not\ d\}$ and $\{not\ c, not\ e\}$. Its removal sets are (RS_1 and RS_4 being minimal):
$RS_1(\bot, R) = \{not\ c\}$
$RS_2(\bot, R) = \{not\ c, not\ e\}$
$RS_3(\bot, R) = \{not\ c, not\ d\}$
$RS_4(\bot, R) = \{not\ d, not\ e\}$
$RS_5(\bot, R) = \{not\ c, not\ d, not\ e\}$

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 support sets of $\bot$. In [16] an algorithm for computing the $CRSs$ is presented.

3 A genetic algorithm for multi-agent belief revision

The algorithm here proposed for belief revision extends the standard genetic algorithm (described for example in [14]) in two ways:

- crossover is performed among chromosomes belonging to different agents,
- a Lamarckian operator called Learn is added in order to bring a chromosome closer to a correct revision by changing the value of the revisables.

Each agent executes the following algorithm:

GA($Fitness, max_gen, p, r, m, l$)

$Fitness$: a function that assigns an evaluation score to a hypothesis coded as a chromosome

max_gen : the maximum number of generations before termination

p: the number of individuals in the population

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

m: the fraction of the population to be mutated

l: the fraction of the population that should evolve Lamarckianly

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 be added 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:

For i=1 to rp

Probabilistically select a hypothesis h_1 from P, according to $Pr(h_1)$ given above

Obtain an hypothesis h_2 from another agent chosen at random

Crossover h_1 with h_2 obtaining h_1'

Add h_1' 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 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 a two-valued revision, 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 positive or default, for every revisable. A chromosome is obtained by associating a bit to each revisable that has value 1 if the revisable is true and 0 if it is false.

Various fitness functions can be used in belief revision. The simplest fitness function is the following

$$Fitness(h_i) = \frac{n_i}{n}$$

where n_i is the number of integrity constraints satisfied by hypothesis h_i and n is the total number of integrity constraints. We will call it an *accuracy* fitness function. Another possible fitness function is the following

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

where f_i is the number of revisables in h_i that are false, and $|h_i|$ is the total number of revisables. We will call it a *hybrid* fitness function. This function is a weighted average of the accuracy and the fraction of false literals in the solution. In this way, the fitness function prefers hypotheses with a lower number of true revisable, which is desirable in some cases.

The first extension to the standard genetic algorithm consists in a crossover operator that allows the exchange of genes among agents. The standard uniform crossover operator produces a new offspring from two parent strings by copying selected bits from each parent. The bit at position i in the offspring is copied from the bit in position i in one of the two parents. The choice of which parent provides the bit for position i is determined by an additional string called crossover mask. This string is a sequence of bits each of which has the following meaning: if bit in position i is 0, then the bit in position i in the offspring is copied from the first parent, otherwise it is copied from the second parent. In uniform crossover, the mask is generated as a bit string where each bit is chosen at random and independently of the others. The crossover operator we consider differs from the standard uniform operator because one of the parents used in crossover comes from the population of another agent.

The other extension to the standard genetic algorithm consists in the addition of the Lamarckian operator *Learn*. This operator 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 the revisables are false 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 revisable objective literals.

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 a revisable literal in M_P, 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 A 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.

To find the support sets we need to know which literals belong to the model of a program. This information is obtainable through some sound and correct procedure for *WFSXp* such as the one described in [1], or the one in [4].

In the case of the circuit diagnosis problems in section 4, the support sets procedure becomes simplified in that the occurrences of default negated literals in the program pertain only to revisables.

The algorithm implementing the *Learn* operator is given below.

procedure $Learn(C, C')$
  inputs : A chromosome C translated into a set of revisables
  outputs : 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

procedure $Support_sets(GL, C, S, SSin, SSout)$:
  inputs :
    GL a list of goals
    A chromosome H translated into a set of revisables
    The current support set S
    The current set of support sets $SSin$
  outputs :
    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 ($GL = [G|RGL]$ 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 $Support_sets(RGL, H, \{\}, SSin \cup \{S\}, SSout)$
    (2) if G is not empty ($G = [L|RG]$) then:
      (2a) if L is a revisable and is in H, then add it to S, and call the algorithm recursively on the rest of G $Support_sets([RG|RGL], H, S \cup \{L\}, SSin, SSout)$
      (2b) if L is a revisable and it is not in H, or its opposite is in H, discard S and call the algorithm recursively on the rest of GL $Support_sets(RGL, H, S \cup \{L\}, SSin, SSout)$
      (2c) if it is not 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 algorihtm recursively $Support_sets([[G_1|RG], ..., [G_n|RG]|RGL], H, S, SSin, SSout)$
      (2d) if it is not a revisable and there are no rules, then return without adding S to SS ($SSout = SSin$)

procedure $hitting_set(SS, HS)$:
  Pick a literal from every support set in SS
  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.

Simplified versions of this algorithm have also been considered in order to separately test the effectiveness of each of the features added to the standard genetic algorithm. In particular, four algorithms have been considered named in the sequel algorithms 1, 2, 3 and 4. Algorithm 1 is a standard single agent genetic algorithm: crossover is performed only among chromosomes of the same agent and the Lamarckian operator is not used. Algorithm 2 adds to algorithm 1 the use of the Lamarckian operator, with a parameter l (percentage of the population to be mutated Lamarckianly) equal to 0.6. Algorithm 3 is a multi-agent algorithm without the Lamarckian operator, i.e., crossover is performed between chromosomes of different agents but the operator Learn is not applied to them. Algorithm 4 extends algorithm 3 by adding the Lamarckian operator, with a parameter l equal to 0.6. For all the algorithms, the mutation rate (parameter m) and the crossover rate (parameter r) have been set to 0.2.

In algorithms 3 and 4 the agents share the same set of observations and program clauses but have different sets of constraints. At the end of the computation, in order to find a single solution for the revision problem, the best chromosome in each agent is considered and is scored with a fitness function that considers all the constraints (global fitness function). Then the chromosome with the highest global fitness is returned as the solution. In this way the multi-agent system finds a solution for the global belief revision problem.

These algorithms have been used in order to experimentally prove the following theses:

1. the distributed algorithm (with or without the Lamarckian operator) has a performance that is comparable (and, in particular, not significantly inferior) to that of the non-distributed one, in the same number of generations and the same overall number of individuals, despite the distrbution of knowledge;

2. Lamarckism is never worse than Darwinism and may outperform it both in the single and in the multi agent case;

In order to test thesis 1, the results obtained by algorithm 1 is compared to the one obtained by algorithm 3 and the same is done for algorithms 2 and 4. In order to test thesis 2, the results obtained by algorithm 1 is compared to the one obtained by algorithm 2 and the same is done for algorithms 3 and 4.

4 Experiments

The algorithms have been tested on a number of belief revision problems in order to prove the above theses. In particular, we have considered problems of digital circuit diagnosis, as per [9], and the n-queen problem.

4.1 Experiment Methodology

In order to evaluate if the accuracy differences between algorithms are significant, we have computed a 10-fold cross-validated paired t test for every pair of algorithms (see [10] for an overview of statistical tests for the comparison of machine learning algorithms). This test is computed as follows. Given two algorithms A and B, let $p_A(i)$ (respectively $p_B(i)$) be the maximum fitness achieved by algorithm A (respectively B) in trial i. If we assume that the 10 differences $p(i) = p_A(i) - p_B(i)$ are drawn independently from a normal distribution, then we can apply the Student t-test by computing the statistic

$$t = \frac{\bar{p}\sqrt{n}}{\sqrt{\frac{\sum_{i=1}^{n}(p^{(i)}-\bar{p})^2}{n-1}}}$$

where n is the number of folds (10) and $\bar{p}$ is

$$\bar{p} = \frac{1}{n}\sum_{i=1}^{n} p^{(i)}$$

In the null hypothesis, i.e. that A and B obtain the same fitness, this statistic has a t distribution with $n-1$ (9) degrees of freedom. If we consider a probability of 90%, then the null hypothesis can be rejected if

$$|t| > t_{9,0.90} = 1.383$$

4.2 Digital Circuit Diagnosis

In problems of digital circuit diagnosis there is a difference between the observed and the predicted outputs. Figure 1 shows a sample circuit together with the observed inputs and outputs of the circuit and 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 cannot be different from the observed value. The representation formalism we use is the one of [9]. As regards the integrity constraints, we have two constraints for each output of the circuit,

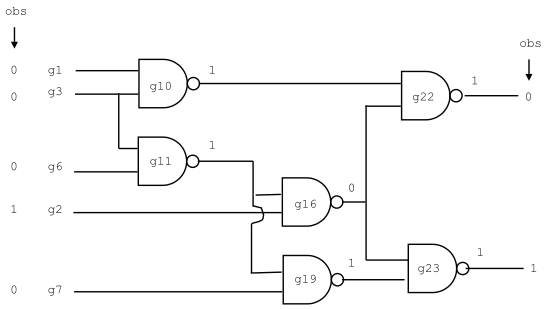

Figure 1: `c17` circuit from ISCAS85's set of benchmark circuits.

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.

The revisable in this case are of the form `ab(Name)` and their meaning is that component `Name` is abnormal.

The system has been tested on some real world problems taken from the ISCAS85 benchmark circuits [6] that have been used as well for testing the belief revision system REVISE [9].[2]

We have considered the `voter` circuit that has 59 gates and 4 outputs, corresponding respectively to 59 revisables and 8 constraints.

Algorithms 1, 2, 3 and 4 have been tested on the `voter` circuit. Each algorithm was run 10 times. The parameters that have been used for the runs are: 10 maximum generations, 40 individuals for algorithms 1 and 2 (single agent), 10 individuals per agent and 4 agents for algorithms 3 and 4. In algorithms 3 and 4 each agent has the same set of observations and program clauses, while the integrity constraints are distributed among the agents so that each agent knows only the constraints that are related to one same output. The hybrid fitness function was adopted.

In table 1 we show, for each algorithm, the value of the fitness function and of its standard deviation for the best hypothesis after ten generations averaged over the 10 runs, while table 2 shows the value of the t statistics for the various couples of algorithms.

4.3 n-queen Problem

The n-queen problem consists in positioning n queens on a $n \times n$ checkboard so that no queen attacks each other. This problem can be seen as Constraint Satisfaction Problem (CSP) where the constraints are: the total number of queens must be n; for each row, the total number of queens must not be bigger than one; for each column, the total number of queens must not be bigger than one and, for each diagonal, the total number of queens must not be bigger than one. This problem can be seen as a belief revision problem by assigning a revisable of the form `queen(Row,Column)` to each position `(Row,Column)` in the checkboard. Then, each constraint of the CSP can be written as an integrity constraint.

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

Algorithm	Fitness	Standard Deviation
1	0.9537	0.035494
2	0.9582	0.015414
3	0.9776	0.007866
4	0.9805	0.007209

Table 1: Voter experiments with algorithms 1, 2, 3 and 4

Comparison	$\|t\|$ value
1-2	0.394
3-4	0.775
1-3	**1.749**
2-4	**2.647**

Table 2: Result of the t-test for different couples of algorithms on the voter dataset.

Algorithms 1, 2, 3 and 4 have been tested also on the n-queen problem with the same parameter as for the `voter` experiment: each algorithm was run 10 times, each run had 10 maximum generations, 40 individuals for algorithms 1 and 2 (single agent), 10 individuals per agent and 4 agents for algorithms 3 and 4. The accuracy fitness function was adopted.

We have considered a problem with $n = 8$. In this case there is a total of 43 constraints: 1 constraint for the total number of queens, 8 constraints for the rows, 8 for the columns and 26 for the diagonals. For multi-agent experiments each agent has the same set of observations and program clauses, while the constraints were divided amongst them: 2 constraints on the rows and 2 on the columns have been assigned to each agent, while the constraints on diagonals have been divided in groups of 6, 6, 7 and 7 and correspondingly assigned to the agents. The constraint on the total number of queens has been assigned to one of the agents with only 6 constraints on the diagonals. Therefore, three agents have 9 constraints and one agent has only 8.

Table 3 shows, for each algorithm, the value of the fitness function for the best hypothesis averaged over the 10 runs while table 4 shows the value of the t statistics for the various couples of algorithms.

Algorithm	Fitness	Standard Deviation
1	0.7581	0.06686
2	0.8232	0.01200
3	0.7930	0.02992
4	0.8069	0.03110

Table 3: n-queen experiments with algorithms 1, 2, 3 and 4

Comparison	$\|t\|$ value
1-2	**2.590**
3-4	0.949
1-3	1.158
2-4	1.328

Table 4: Result of the t-test for different couples of algorithms on the n-queen dataset.

4.4 Discussion of Experimental Results

As can be seen from tables 2 and 4, in the voter case algorithm 3 performs significantly better than algorithm 1 as well as algorithm 4 with regards to algorithm 2. In the n-queen case, instead, the difference for the pairs of algorithms (1,3) and (2,4) is not statistically significant for the voter problem but not for the n-queen problem. These two cases prove thesis 1, i.e., adopting a distributed algorithm does not penalize and may actually improve the fitness. So our algorithm allows to benefit from parallel execution. The difference in results between the two cases may occur because in the n-queen problem the problem is more constrained and the constraints are more interdependent and therefore distributedly finding local solutions and then exchanging them is less effective.

As regards thesis 2, the difference for the pairs of algorithms (1,2) and (3,4) is statistically significant only for the couple (1,2) in the n-queen case. Therefore, thesis 2 is confirmed by the data. Moreover, in another follow-up work [12] we have shown that, by allowing memes to be crossed over only if they have been accessed by the Lamarckian algorithm, we obtain a statistically significant difference for the couple (3,4) in both cases.

5 Related Work

In [13] the authors propose an approach for performing belief revision in a multi agent context. In their approach, each agent exploits an Assumptions Based Truth Maintenance (ATMS) system in order to perform the revision of beliefs. As in our approach, each agent has a different repository for knowledge and its beliefs may not be consistent with those of other agents, consistency is enforced only locally inside each agent. Differently from us, in [13] the authors consider an exchange of beliefs by means of a number of communications primitives. Communication happens in three cases. The first is when an agent can not establish by itself the truth value of an assumption or a goal: in this case, it asks it to its acquaintances. The second case is when an agent finds a conclusion or an assumption that it knows being of interest for another agent: in this case it communicates the results. The third case is when an agent has revised the truth value of a belief that it had previously communicated to other agents: in this case the agent communicates the other agents the new truth value for the belief. Therefore, in [13] the cooperation among agents is explicit, while in our work the cooperation emerges as the result of the continuous exchange of chromosomes among agents.

In [13] the system is able to answer uniquely to queries posed to the system by means of a meta-level algorithm that works in the following way: a fact is false if it is considered false by at least one agent, a fact is true if no agent considers it false and there is at least one agent that considers it true. In our system, instead, the global result of the belief revision process is given by the chromosome with the best global accuracy, also computed by means of a meta-level algorithm that considers all the constraints.

6 Conclusions and Future Work

We have presented a genetic algorithm for performing belief revision in a multi-agent environment. In this setting, individuals belonging to different agents are exposed to different experiences. This happens because the agents explore different parts of the world or because the world surrounding an agent changes over time. The algorithm permits the exchange of chromosomes from different agents and this allows a distributed belief revision process to be performed.

The algorithm combines two different evolution strategies, one based on Darwin's and the other on Lamarck's evolutionary theory. The algorithm therefore includes also a Lamarckian operator that changes the memes of an agent in order to improve its fitness. The operator is implemented by means of a logic-based belief revision procedure that, by tracing logical derivations, identifies the memes leading to contradiction.

Experiments on problems of digital circuit design and on the n-queen problem show that, in spite of the fact that each agent has only a partial knowledge of the world, the multi-agent system finds revisions that are comparable in terms of accuracy with those obtained by a single-agent possessing all the knowledge.

In the future, we plan to explore also the case in which the chromosomes do not have all the relevant revisables to start with (three-valued revision). In this case, when a chromosome acquires new revisables from another chromosome, it is obtaining specialized knowledge. 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 others' revisables.

7 Acknowledgements

L. M. Pereira acknowledges the support of PRAXIS project MENTAL "An Architecture for Mental Agents".

References

[1] J. J. Alferes, C. V. Damásio, and L. M. Pereira. A logic programming system for non-monotonic reasoning. *Journal of Automated Reasoning*, 14:93–147, 1995.

[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] T. Dietterich. Approximate satistical tests for comparing supervised classification learning algorithms. Neural Computation, in press (draft version available at `http://www.cs.orst.edu/ tgd/projects /supervised.html`), 2000.

[11] 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.

[12] E. Lamma, F. Riguzzi, and L. M. Pereira. Belief revision via lamarckian evolution. Technical Report DEIS-LIA-00-004, University of Bologna (Italy), 2000. LIA Series no. 44.

[13] B. Malheiro, N. R. Jennings, and E. Oliveira. Belief revision in multiagent systems. In *Proceedings of the 11th European Conference on Artificial Intelligence*, 1994.

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

[15] 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.

[16] 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.

[17] 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.