Autonomous semi-reactive agent design based on incremental inductive learning in logic programming

Marcello Balduccini and Gaetano A. Lanzarone
Department of Computer Science, University of Milan
via Comelico, 39 20135 Milan Italy
email: marcy@morgana.usr.dsi.unimi.it, lanzarone@hermes.mc.dsi.unimi.it

ABSTRACT

In this paper we study the problems involved in realising an intelligent agent which interacts with an unknown environment and builds an inductive model of it which is incrementally updated during interaction. The project belongs to the context of logic programming and is oriented to the construction of a learner and a planner - the base components of the agent - sharing the same knowledge base (the model of the environment), which is represented in extended propositional language. This paper is particularly devoted to describing the problems related with the definition of the knowledge representation formalism and with learning, which is symbolic, inductive and incremental. This approach to building intelligent agents differs from the current literature about this subject, since for the first time we try to adopt a general-purpose inductive symbolic learner for the learning component of an agent which interacts with the environment in real-time. All the project choices will be oriented to conciling the need for reduced computational costs with the ability to reach the assigned goals in complex environments and in an acceptable time; this trade-off, as we will see, influences both the choice of the knowledge representation language and the reasoning techniques, which must not only be extremely efficient, but also very well integrated between themselves.

1.State of art and objective definition

In this paper we introduce an agent which autonomously interacts with an unknown environment and builds an incremental and inductive model of it.

The key concepts dealt with are:

- Autonomy
  The agent can't rely on an oracle guiding its inferential processes.
- Incrementality
  The agent must be able to continuously update the environment's model according to the knowledge derived from interaction, exploiting as much as possible the information it has already acquired.
- Inductivity
  The agent must infer rules on the basis of observed examples, and it is therefore necessary for it to constantly verify the validity of the rules it has induced.
- Unknown environment
  The agent must perform well even when started in a 'zero knowledge' situation.

The aim of this paper is to investigate how the characteristics of the components of an agent are influenced by the fact that the agent has to interact with an unknown environment and build at the same time an incremental and inductive model of it by means of symbolic learning.

What we want to obtain is an agent featuring sufficiently low computational costs to be able to interact with the environment in real-time, but powerful enough to manage to reach the assigned goals in complex environments and in an acceptable time.

Above all we will discuss the following four topics:

- Calibrating the knowledge representation language
  The representation language must be at the same time rich enough to be able to describe a large set of models and simple enough to allow to execute the induction and deduction processes in a short time. Moreover it is to be argued if all the components of the agent must share the same representation language, or if each may exploit a custom language and be given some translation procedures to communicate with the other components.
- Fast reasoning algorithms
  All the reasoning algorithms used must grant a fast execution to allow the interaction with the environment to really take place in real-time.
- Suitable learning techniques

Learning techniques must be developed keeping in mind that the agent will have to handle a great deal of data, which makes less important the need to perform refined inductions, but at the same time requires good backtracking capabilities and limited computational costs.

- Refined problem solving algorithms
  Care must be taken in order to let the planner wholly exploit the data structures generated by the learner, and operate without necessarily building complete plans, since this generally requires an excessive time. Moreover the planner must work well also with incomplete knowledge and handle situations in which the environment evolves differently from what had been planned.

The agent was developed in prolog, in the context of logic programming. Its performances are currently being tested on simulated bidimensional environments, and the experiments are carried on by means of a client/server multi-user environment for development and experimentation which we have created for this purpose.

This project is located at the intersection of three different context: learning, planning and problem solving, and intelligent agents construction.

From the point of view of the construction of intelligent agents literature is quite varied and in continuous evolution.

In this context two main and generally opposed currents may be identified: the rational agents one and the reactive agents one. The basic difference between them is that while in the first case the agent chooses which actions to perform according to the results of an adequate and precise planning, in the second case the agent employs a rough and short termed planning, which is very cheap from the point of view of resource requirements.

The problem of learning in intelligent agents is frequently solved by representing the environment by means of predefined and hard-wired models which the learning component adjusts by trimming suitable numeric parameters. A different approach is the one of neural network based agents which, thanks to their general purpose learning algorithms, offer a greater versatility and a computationally realisable interaction with the environment, but are penalised by a poor representation language which limits the complexity of the environments which may be tackled. Symbolic learning is rarely used in intelligent agents and, when it is, it is generally in the form of deductive learning, oriented to the optimisation of the knowledge base in order to improve its performance, without altering its informative contents.

Two important examples of this subject are in [Laird and Rosenbloom, 1990] and [Maes and Brooks, 1990]. An extensive review on reactive agents is available in [Maes, 1990]. In [Kowalski, 1995] may be found a very clear description of the overall structure of an intelligent agent, with particular attention to the reconciliation of the rational and the reactive approach.

On the side of inductive learning, it can be seen that in the literature there are three main approaches. On the opposite side respect to the sub-symbolic approach described earlier,. is symbolic learning with predicative languages, which are endowed with an expressiveness which allows to deal with very complex situations, but whose high computational cost makes incremental learning during interaction with the environment currently unrealisable.

Between the two extremes is the propositional approach; in this case the good expressiveness of the language allows a satisfying interaction with environments of greater complexity than with the sub-symbolic approach, while the limited computational costs make learning in real-time (that is, incremental learning during interaction with the environment) realisable.

Finally on the side of planning and problem solving the setting is very large and complex, but presents a well defined gerarchical structure.

In this field we may distinguish two main approaches, namely static and dynamic planning. In the former the planner builds a complete plan starting from complete knowledge: computation has no time constraints and it is assumed that, during execution, no unexpected events may take place.

In the latter, instead, the planner must create a plan within a limited time, and, during execution, unexpected events may happen which require a revision of the plan. In this context many reasoning techniques are known, among which let's recall the method according to which planning is interrupted as soon as it violates the constraints on resource occupation ([Kowalski, 1995]), and the method which proceeds by building first of all a high level plan, and later examines the single details according to the available resources ([Laird and Rosenbloom, 1990]).

Of course, rather than being two distinct and opposed currents, these two approaches represent the border lines of an area inside which anyone can choose the best strategy according to the application's needs.

From the point of view of the construction of intelligent agents this paper is oriented to projecting a symbolic inductive learning based agent. For what concerns the choice between the rational and the reactive approach, we believe that the need to keep computational costs low, united with our willingness to keep the planning ability, makes almost inevitable to follow the idea of a hybrid structure exposed in [Kowalski, 1995], which we will call semi-reactive agent.

Respect to the scenario of learning this paper investigates incremental inductive learning with predicative languages. In order to make learning in real-time computationally realisable, we have adopted a propositional attribute value language extended with predicative elements. This choice has the advantage that it allows us to exploit the propositional learners already described in the literature (see [Clark and Niblett, 1989], [Muggleton, 1987], [Ginsberg, 1990]), whose properties and performances are well known, united with the capability to handle much more complex environments than those propositional languages can handle.

For simplicity we will suppose that the agent is made up of only two well distinguished components, the learner and the planner, the first of which has to build the model of the environment, while the second has to exploit the model to interact in the best way - according to suitable criteria - with the environment.

The learning process is inductive and proceeds through the construction of a model, which represents the rules at the basis of the state transitions of the environment, and which is dynamically updated and corrected during the whole activity period of the agent. The learning algorithm belongs to the class of incremental learning algorithms, since updating the knowledge base which represents the model doesn't require a reelaboration of knowledge, but only a simple adaptation; the language in use is predicative, but extremely near to a propositional attribute value language.

The environment with which interaction takes place is a set of objects of which the agent is given a sensorial representation described in an attribute value language, and whose state transitions the agent may influence by means of suitable actuators. It is "unknown" in that the agent has no initial model of it; moreover the paper is focused on single-agent environments.

In this paper we will deepen the study of the learning module, by describing the representation formalism we chose for the knowledge base and by analysing the reasoning techniques in use. In order to characterise a whole class of learners suitable as agent's learning modules, we will introduce an abstract learner, that is a general algorithm, which may be specialised to both batch and incremental learners, and we will show how it is possible to derive from it two batch learners which have great importance in literature, such as AQR and CN2. We will also introduce a specialisation of the abstract learner which is used as the learning component of our agent.

The planner will be shortly discussed at the end of the paper, paying particular attention to the ways of exploiting the knowledge base. Finally we will briefly talk about the agent development and experimentation environment.

2.Knowledge representation

The model the agent builds expresses the transition rules among the states of the environment. We represent each state as a tuple of attribute value couples.

Since we believe that the developed language may easily adapt to the needs of both components, the learner and the planner share the same knowledge representation formalism.

We chose to represent the environment's model by means of rulesets, rather than with decision trees; this decision is motivated by considering that during the knowledge revision phase it's frequently necessary to modify only some fragments of the knowledge base - which is cheaper than a large-scale change - but using decision trees may often bring, even in these cases, to review the whole tree's structure, which worsens the performance. For the same reason it was decided to use unordered rulesets, rather than ordered ones.

A transition rule has a premise, which describes the state in which the environment is at time t, and a consequence, represented by an attribute value pair, indicating which value that attribute will take in the representation of the environment at time t+1.

Referring to the classification of the environments in [Russell and Norvig, 1995] we can define the class of environments the agents has to deal with as inaccessible, deterministic, nonepisodic, static and discrete**.**

As we already know, an inaccessible and deterministic environment may appear nondeterministic to the agent. For this reason, in order to keep the predictive capability of the model, it is better to use as a premise of the rules a sequence of consecutive states.

The rule's syntax is:

$$< \text{rule} > ::= < \text{selector} > :- \{< \text{state} >\}^{*}$$

$$< \text{state} > ::= \{< \text{selector} >\}^{n}$$

$$< \text{selector} > ::= < \text{attribute} > = < \text{value} >$$

where * and ^{n} mean, respectively, any *nonnegative number of occurrences*, and *n occurrences*.

The temporal localisation of each state in the sequence is implicitly expressed by its position in it, with the leftmost state which immediately precedes the application of the rule, and the states on its right which progressively extend back into time.

In many application areas the representation of a particular knowledge assumes great importance, which we will call geographic knowledge.

For example in [Russell and Norvig, 1995] an environment called Wumpus World is introduced, in which the following rule is valid:

> Given a 3x3 subset of the bidimensional environment, with the reference system's origin in the central square of the grid, **if in (-1,0), (1,0), (0,-1) and (0,1) a *breeze* is perceived, then in (0,0) a *pit* is present**.

In brief, the kind of geographic knowledge we're interested in is the one expressing the properties of determinate space portions. For instance we may be interested in representing the fact that in every state in which X=x^* and Y=y^*, the COLOR attribute takes the $color^*$ value. What we do in substance is to link the attribute value pair in the environment's representation at time t with an already known subset of the same.

From the syntactic point of view, it's enough to impose the premise's state sequence to begin at the instant following the rule's application, rather than the preceding one, in order to obtain suitably expressive rules.

What we have seen is not yet sufficient to solve the *Wumpus World problem*. In fact we aren't able yet to link the properties of two distinct space portions in a time independent way.

For example let's consider a simplification of the Wumpus World rule according to which in all such states that X=*0* and Y=*0*, PIT is *true* if in all states in which X=*1* and Y=*0*, BREEZE=*true* holds.

The problem in this case resides in the fact that premise and consequence have no precise temporal link: the consequence is valid apart from the time past since the verification of the premise. A rule contains, instead, a precise temporal link between premise and consequence: it would be possible to write one to say that if X=*1* and Y=*0* and BREEZE=*true* and if, at the next instant, X=*0* and Y=*0*, then, in that instant, PIT=*true* holds, but infinite rules are needed in order to remove the temporal link.

There is also a more subtle problem about this subject, which we are going to show. The above informal rule has the following formal representation:

$$\text{PIT} = \text{true} :- (\text{X} = 0\ \text{Y} = 0),$$
$$(\text{X} = 1\ \text{Y} = 0\ \text{BREEZE} = \text{true})$$

Suppose for a moment the rule doesn't imply adjacency between states; also suppose the state (X=*1* Y=*0* BREEZE=*true*) **[A]** was observed, but that the state (X=*1* Y=*0* BREEZE=*false*) was observed later. May the rule be applied yet ? It's reasonable to say it can't (probably the pit was closed).

Suppose now that instead of (X=*1* Y=*0* BREEZE=*false*) the state (X=*1* Y=*1* BREEZE=*true*) **[B]** was observed. This time one expects the rule may still be applied, since observing [B] doesn't invalidate [A]. It is clear that, at the state representation level, it is difficult to render the difference between the two cases.

To avoid the above problems we introduce the metarules. A metarule for the above example is:

$$\text{PIT} = \text{true} :- (\text{X} = 0\ \text{Y} = 0);$$
$$\{\text{BREEZE} = \text{true} :- (\text{X} = 1\ \text{Y} = 0)\}$$

whose meaning is: if, at time t, X=*0* and Y=*0*, and the $\text{BREEZE} = \text{true} :- (\text{X} = 1\ \text{Y} = 0)$ rule still holds, then at time t PIT is *true*.

Now it's possible to write a metarule for the *Wumpus World problem*. It is structured as follows:

$$\text{PIT} = \text{true} :- (\text{X} = 0\ \text{Y} = 0);$$
$$\{\text{BREEZE} = \text{true} :- (\text{X} = 1\ \text{Y} = 0)\},$$
$$\{\text{BREEZE} = \text{true} :- (\text{X} = -1\ \text{Y} = 0)\},$$
$$\{\text{BREEZE} = \text{true} :- (\text{X} = 0\ \text{Y} = 1)\},$$
$$\{\text{BREEZE} = \text{true} :- (\text{X} = 0\ \text{Y} = -1)\}$$

3.The learning module of the agent

The learner's task is to observe the environment's state transitions and consequently update the knowledge base.

In the ambit of batch propositional learners the most used searching algorithm is the least-commitment search, and especially the beam search, since this method has the feature to delay the choices until enough information is available, avoiding the need for backtracking.

In the incremental approach at each time slice the learner must return a knowledge base consistent with the events observed till that moment. Therefore the learner is obliged to make those choices that in the batch ambit could have been delayed until the acquisition of the whole dataset, and this involves resorting again to backtracking.

Because of this, it is no more possible to choose to operate exclusively generalisations or specialisations, as it happens in batch learning. The search space has to be traversed both bottom-up (generalising) and top-down (specialising).

Besides the learner's knowledge base must be immediately usable to the planner, in an applicative area as large as possible. Therefore it's the learner's task to adequately post-process the knowledge base.

Finally the learner must record the needed information to extend back into time the state sequence of the rule's premise and to build the metarule's metapremise.

The first part of the induction process is devoted to building new rules. Given two states $state_t$ and $state_{t+1}$, images of the environment at time t and t+1 respectively, for each attribute value pair of $state_{t+1}$ a rule is built, having the pair as consequence, a void premise (meaning always true) and the ($state'_{t+1}$ $state_t$) sequence as extended premise (introduced in §3.1), where $state'_{t+1}$ is $state_{t+1}$ without the pair used as rule consequence[1].

3.1.Specialisation

Let's start with some definitions we will use later.

- Two selectors are opposite if they describe the same attribute, but have different values.
- The domain of the premise is the set of state sequences verifying the premise.
- The domain of the rule is the domain of its premise.
- Given two rules A and B, from B follows A (A is consequence of B) if A and B have equal consequences and the premise's domain of A is a subset of the premise's domain of B.

Specialisation performs a top-down traversing of the search space in order to restrict the domain of the rules covering negative examples[2]. The domain's restriction happens with the standard method of adding conditions to the rule's premise. The heuristics determining which conditions to add were refined by means of an empirical experimentation period.

We make specialisation operate on pairs of inconsistent rules, having nonempty intersection domains, in order to make the intersection set empty.

Specialisation works by means of four operators. They use two additional information fields in the rule's structure: the extended premise and the (positive) example set. The extended premise is the most specific premise covering all the rule's positive examples. The example set is just this collection, or a subset of it, according to whether the *full memory* or *partial memory* model was adopted.

The first operator specialises the second rule on the basis of the first, by adding to it the selectors of the second rule's extended premise having an opposite in the first rule's premise.

The second operator specialises the first rule on the basis of the second, by adding to it the selectors of the first rule's extended premise having an opposite in the second rule's premise.

The third operator specialises both rules by adding to each its extended premise's selectors having an opposite in the other rule's extended premise.

The fourth operator is used to remedy to wrong choices made in the generalisation phase. The second rule is removed, and from each element of its example set a new rule is generated, having as premise the one of the original rule, but as extended premise the one necessary and sufficient to cover the example it's generated from. Then the new rules are generalised, taking into account the presence of the first rule, which prevents reconstructing the original second rule.

The four operators are applied in growing order of power.

From the point of view of the specialisation's cost in time, we underline that the process isn't applied to all the rule pairs which may be extracted from the knowledge base, since this is consistent by construction, but only to the pairs having as first element each new rule to be inserted in the knowledge base and as second element each rule already present in it.

3.2.Generalisation

Generalisation performs a bottom-up traversing of the search space, in order to enlarge the rule's domain and include new examples in it.

Generalisation is applied to pairs of rules having the same consequence, and generates a third rule the first two follow from. Like specialisation, also generalisation exploits a standard method, the one of dropping conditions from the premise.

Generalisation acts by means of three operators.

The first operator controls if the second rule follows from the first, in which case removes the second rule and updates the first by adding to its example set the examples of the second, and suitably enlarging its extended premise.

The second operator controls if the first rule follows from the second, in which case removes the first rule and modifies the second by adding to its example set the examples of the first, and suitably enlarging the extended premise.

The third operator removes both rules and builds a third one which is the most specific rule both follow from. Basically its premise contains the intersection between the premises of both rules and in the same

[1] Otherwise we would allow the construction of cyclic rules like:

ATTR = val :– ATTR = val

[2] In this ambit we define examples the pairs (consequence, premise), where the premise is completely specialised, that is it has a domain of unary cardinality. Are considered negative examples (respect to a rule) the examples having a consequence referred to the same attribute of the rule, but different value.

way the extended premise is created. The example set is simply the union of starting rules' sets. Before the substitution takes place, it's verified the new rule is consistent with the knowledge base.

The third operator has the task to prevent the learner from remaining blocked in blind alleys because of some attribute value pairs wrongly introduced in a premise. Moreover this operator disengages us from a particular choice for the new rules' premise: we are currently using an initial empty premise, but equivalent performances were obtained with more specific initial premises, too.

The three operators are applied in growing order of power.

Like specialisation, also generalisation is applied only to the rule pairs obtained using as first element each new rule and as second each rule in the knowledge base.

3.3.Rule and metarule processing

As we said it may happen that, *from the point of view of the agent*, the environment behaves nondeterministically. From this follows that in some situations specialisation isn't enough to make the knowledge base consistent.

The typical situation in which it's not possible to restore consistency is when two rules have opposite consequences, identical premises, and it is impossible to specialise their premises any more.

The approach we follow to solve this problem consists in extending back into time the premises of the inconsistent rules, hoping to obtain some information to supply to the environment's inaccessibility.

What we do is marking the inconsistent rules in order to temporarily exclude them from the knowledge base, which therefore returns consistent. From now on the inconsistent rules are kept under observation and, when the condition regarding the less recent state of the premise gets true, the representation of the environment at the previous instant is recorded in a suitable rule's field. When the rule is completely verified, in that also the consequence holds, then we have found an example for the rule with a more extended premise than the rule itself. This information is then added to the rule which may be put back into the knowledge base.

We have also found that, if the rules are continuously kept under observation, independently from whether they're marked, this makes it is possible to know at each time slice, without other computations, which rules are going to be activated at the next instant, which turns out to be very useful during planning.

The specialisation and generalisation algorithms shown above may be applied also to metarules, provided that the following definitions are given and all references to rules in the algorithms are substituted with references to metarules.

- Two elements A and B of a metapremise (so also two rules) are opposite if they have opposite consequences and the intersection of the premises' domains is not void.
- An element A of a metapremise is verified in a state sequence if the rule A is activated in the sequence and no opposite rule to A is activated in the sequence after A.
- The domain of the metapremise is the set of all state sequences verifying each element of the metapremise.
- The domain of the metarule is the set of all state sequences verifying both the premise and the metapremise of the metarule.

4.A more general learner

The abstract algorithm we introduce, named Alpha, allows to describe a large class of top-down, bottom-up and hybrid learners, either **batch** or **incremental**, working on rulesets which may be either **ordered** or **unordered**, with **noise handling** capabilities and marked by an attribute value knowledge representation language, extended with predicative elements if necessary. As we said above (§1), we are interested in such an algorithm because it allows us to characterise a whole class of learners suitable for working as the learning module of the agents we are studying.

The learner is based upon two predicates implementing respectively top-down and bottom-up searching in the hypothesis space. As you may see in the listing the structure of the two predicates is the same: in fact both execute a beam search. What changes is the definition of the predicates characterising the various learners. For this reason in the following paragraphs we will describe in general the searching algorithm, distinguishing between the two versions only when necessary.

It may be reasonably supposed that any beam seach based batch learner may be derived from Alpha, (we'll show the AQR and CN2 derivation later on), like also many incremental learners and in particular the one we have developed, named Gamma, on which the agents described above is based.

4.1.Algorithm description

The core of Alpha is based on a beam search in the hypothesis space which at each recursion specialises or generalises the current star on the basis of the learner defined criteria (*select_negex*, *specialise_star*, *select_noncov*, *generalise_star*), prunes it by both removing those elements subsumed by others (*subsumption_pruning*), and discarding the less significant elements if the star's size exceeds a predefined threshold (*size_pruning*), and selects the

best element by choosing between the best in the star and the best computed in the previous recursion (*select_best_complex*).

The top level predicate of Alpha's top-down searching algorithm [top_down(KB, To_be_classified, Rules, New_KB)] receives two example sets (KB and To_be_classified) and returns a ruleset (Rules) covering the second input example and a copy of KB made consistent with Rules (New_KB). When the predicate is called, no difference is made between positive and negative examples, which are mixed in the two input sets and distinguished only by a label representing the class they belong to, but only between examples to be used for generating new rules (To_be_classified) and examples to be used only to compute the statistical significance of the rules and to test consistency (KB).

By suitably handling KB and To_be_classified it is possible to determine the way the learner works. The different cases are reported in Table 1.

The **noise handling** capability is introduced by the implementation of the predicates *select_seed*, *select_negex* and *specialise_star*, as shown in [Clark and Niblett, 1989].

The bottom-up search predicate [bottom_up(KB, To_be_generalised, Rules, New_KB)] receives two rulesets (KB and To_be_generalised) and returns a ruleset (Rules) which generalises the second input set and a copy of KB updated according to the contents of Rules (New_KB).

In order to support **incremental** learning the structure of the *complexes* (the star's elements, which in Alpha are not distinguished from the rules) was extended respect to the one presented in [Clark and Niblett, 1989] in order to contain, besides the rule under construction, also a copy of the original KB set, suitably modified by *specialise_star* (or by *generalise_star*). Thanks to this device Alpha can specialise (or generalise) both the complexes under construction and parts of the knowledge base already present, reaching in this way incrementality.

It must be underlined that in the incremental use of the KB set it is implicitly assumed it can contain at the same time examples and rules. It is the derived learner's task to allow this to really happen.

Another subtlety we want to underline is that the implementation of Alpha is founded on the assumption that examples can't be further specialised; this allows to solve in a direct way the need to operate on the union of KB and To_be_classified, while specialising only the elements of KB. Until now we have found no learner violating this assumption, nor we see a reason why this should happen in future, therefore we won't dwell upon this subject any longer.

Finally note that the learner works in bottom-up fashion only on unordered rulesets and, therefore, incremental hybrid learning may be done only on knowledge bases of this kind.

4.2.Deriving the learners

The overall behaviour of the algorithm depends upon the predicates which must be defined by each learner and which are described in Table 2.

We are now going to show how it is possible to characterise Alpha's predicates in order to derive from it AQR, CN2 and Gamma. Of course, since AQR and CN2 are based only upon top-down searching, the predicates related to bottom-up searching won't be reported in their derivations.

For a more deep description of AQR and CN2 refer to [Clark and Niblett, 1989].

Table 3 shows how Alpha's predicates must be defined in order to derive AQR. Note that the KB set coincides with Neg and To_be_classified coincides with Pos.

Table 4 summarises the derivation of CN2. Note that the derivation is correct only on condition that the covers predicate has the following property (which seems reasonable):

$$\begin{cases} \forall sel \ \text{covers}(sel, \textbf{true}) \\ \forall sel \ \neg \text{covers}(sel, \textbf{false}) \end{cases}$$

The use of the KB and To_be_classified sets depends upon the adopted kind of ruleset (**ordered** o **unordered**).

It is particularly interesting to note that CN2 and AQR are substantially identical, except for the definition of *select_seed* and *select_negex*.

Table 5 shows the derivation of the incremental learner Gamma. The specialisation and generalisation operators implementing *specialise_star* and *generalise_star* are those described in §3. As you may note Gamma operates with a value of MaxStar fixed, equal to one.

An advantage of Alpha is that it allows to compare the learners each other, even when apparently they have nothing in common, and obtain useful indications on the possible modifies to improve the learner's behaviour. From the observation of the derivations of the three learners shown above we had the idea, for example, of an **incremental** learner, similar to Gamma, but having a MaxStar value greater than one, and with termination and selection criteria for the examples inspired to those of CN2. Even if the learning capability of such a learner has still to be evaluated, it would undoubtedly offer the chance to have **noise handling** and a large star in the ambit of **incremental** learning.

5.Realisation status and final observations

The planning component of the agent is still under development. As we said in §1, we are currently following the approach in [Kowalski, 1995], which takes into account resource occupation while computing, in order to reduce the computational cost. A great advantage by which our planner benefits is the availability of the information derived from what exposed in §3. It is in fact possible for our planner to foresee the state following the current one by taking into account only a small number of rules, and without recording the whole sequence of states which has been observed.

Agent tests are executed by means of a client/server multi-user development environment, designed on purpose, inspired to EDEN/POPBEAST ([Paine, 1993]).

The system is still in beta version, and a server is currently available one can connect to by means of a graphics client written in Java language. Authorised people can connect to the server using any network browser and create their own experiments, or observe someone else's experiments. An experiment is made up of a bidimensional world, whose topology and object behaviour may be defined at will, and an agent, which may itself be defined at will.

A little library of worlds and agents is currently available; in order to create one's own it is not enough to connect to the server, but a login on the computer where the server resides is needed.

Worlds are implemented in C++ and a class library is available implementing the basic objects' functions. Agents may be written either in C++ or in Prolog (the server has an embedded prolog interpreter). In both cases an interface is available which simplifies the interaction with the server.

The development environment may be reached at Internet address:

http://artu.usr.dsi.unimi.it/marcy/Eden

where it is available some documentation and an access page to the server. We believe the source code for client and server will be also soon distributed.

6.Appendix

KB	To_be_classified	Working manner
empty	Pos+Neg	**Ordered** rulesets, **batch** learning, covering for both Pos and Neg
Pos+Neg	Pos+Neg	**Unordered** rulesets, **batch** learning, covering for both Pos and Neg
Neg	Pos	**Unordered** rulesets, **batch** learning, covering for Pos
KB	Pos+Neg	**Ordered** rulesets, **incremental** learning, covering for both Pos and Neg
KB+Pos+Neg	Pos+Neg	**Unordered** rulesets, **incremental** learning, covering for both Pos and Neg
KB+Neg	Pos	**Unordered** rulesets, **incremental** learning, covering for Pos

Table 1 - Working manners of Alpha

Predicate	Description
select_seed	Returns an example for which a covering complex must be generated.
select_noncov	Returns a positive example not covered by the specified complex.
add_complex_to_cover	Adds a complex to the covering set.
termination	Implements the termination criteria.
select_negex	Returns a negative example the current complex must not cover.
specialise_star	Applies the specialisation operators to the star.
generalise_star	Applies the generalisation operators to the star.
subsumption_pruning	Removes from the star the complexes subsumed by other complexes in the star.

Predicate	Description
size_pruning	Removes the less significant complexes from star until star's size is under the imposed threshold.
select_best_complex	Chooses the best complex between the best in star and the best computed by the previous recursion.

Table 2 - Description of the customisable predicates

Predicate	Definition
select_seed	Selects a positive example not covered by the current cover. Selection follows AQR's heuristics.
add_complex_to_cover	Adds the complex to the cover.
termination	not covers(Star, negative_KB_examples_respect_to_Seed)
select_negex	Selects a negative example covered by the star.
specialise_star	Given Star, the new StarOUT is built in this way: $\begin{cases} Extension = \{sel \mid \text{covers}(sel, Seed) \wedge \neg \text{covers}(sel, NegE)\} \\ StarOUT = \{x \wedge y \mid x \in Star,\ y \in Extension\} \end{cases}$
subsumption_pruning	Removes all complexes subsumed by other complexes in star.
size_pruning	Removes the less significant complexes until star's size is less or equal the user defined threshold.
select_best_complex	Returns the best complex among the ones in the current star, ignoring the best complex of the previous recursion. It may be easily shown that this is equivalent to the selection used in the implementation of AQR.

Table 3 - AQR's derivation

Predicate	Definition
select_seed	Returns true, an example always verified (in [Clark and Niblett, 1989] it is indicated as an empty complex).
add_complex_to_cover	Adds the complex to the cover.
termination	not empty(Star)
select_negex	Returns false, an example never verified (in [Clark and Niblett, 1989] it is indicated as a null complex).
specialise_star	Given Star, the new StarOUT is built in this way: $\begin{cases} Extension = \{sel \mid \text{covers}(sel, Seed) \wedge \neg \text{covers}(sel, NegE)\} \\ StarOUT = \{x \wedge y \mid x \in Star,\ y \in Extension\} \end{cases}$
subsumption_pruning	Removes all complexes subsumed by other complexes in star and all unsatisfiable complexes.
size_pruning	Removes the less significant complexes until star's size is less or equal to the user defined threshold.
select_best_complex	Returns the best complex between the best in star and the best computed by the previous recursion.

Table 4 - CN2's derivation

Predicate	Definition
select_seed	Selects a positive example not covered by the current cover. Selection follows Gamma's heuristics.
add_complex_to_cover	Adds the complex to the cover.
termination	not covers(Star, negative_KB_examples_respect_to_Seed)
select_negex	Returns a negative example the current complex must not cover.
select_noncov	Returns a positive example the current complex doesn't cover.
specialise_star	Applies Gamma's specialisation operators to the star.
generalise_star	Applies Gamma's generalisation operators to the star.
subsumption_pruning	Removes all the complexes subsumed by other complexes in star and all unsatisfiable complexes.
size_pruning	Removes the less significant complexes until star's size is less or equal to **one**.
select_best_complex	Returns the only complex contained in star.

Table 5 - Gamma's derivation

```
gamma_top_level(KB_in,New_examples,KB_out) :-
          append(KB_in,New_examples,Temp_KB),       %% build unordered rulesets
          top_down(Temp_KB,New_examples,S_rules,S_KB),
          bottom_up(S_KB,S_rules,G_rules,G_KB),
          append(G_KB,G_Rules,KB_out).
```

Figure 1 - Gamma's top level predicate

```
top_down(KB,To_be_classified,Rules,New_KB) :-
        top_down_1(KB,To_be_classified,[],Rules,New_KB).

top_down_2(New_KB,To_be_classified,_,Rules,Rules,New_KB) :-
        empty(To_be_classified), !.
top_down_2(New_KB,_,Best,Rules,New_KB) :-
        Best==[_,false], !.
top_down_2(KB,To_be_classified,_,RulesIN,RulesOUT,New_KB) :-
        top_down_1(KB,To_be_classified,RulesIN,RulesOUT,New_KB).
top_down_1(KB,To_be_classified,RulesIN,RulesOUT,New_KB) :-
        select_seed(RulesIN,To_be_classified,Seed),
        append(KB,To_be_classified,Temp_examples),
        generate_complex(Seed,[[Temp_examples,true]],[Temp_examples,false],Best),
        add_complex_to_cover(RulesIN,Best,RulesIN1),
        covered_by(Best,To_be_classified,Covered),
        set_difference(To_be_classified,Covered,New_to_be_classified),
        new_examples(Best,KB,To_be_classified,New_KB1),
        top_down_2(New_KB1,New_to_be_classified,Best,RulesIN1,RulesOUT,New_KB).

%% generate_complex(Seed,StarIN,Best_Cpx,Complex)
%% top-down module

generate_complex(Seed,Star,Complex,Complex) :-
        termination(Star,Seed), !.

generate_complex(Seed, StarIN,Best_Cpx,Complex) :-
        select_negex(Seed,StarIN,NegE),
        specialise_star(Seed,NegE,StarIN,StarOUT1),
        subsumption_pruning(StarOUT1,StarOUT2),
        size_pruning(StarOUT2,StarOUT),
        select_best_complex(StarOUT,Best_Cpx,New_Cpx),
        generate_complex(Seed,StarOUT,New_Cpx,Complex).

bottom_up(KB,To_be_generalised,RulesOUT,New_KB) :-
        bottom_up_1(KB,To_be_generalised,[],RulesOUT,New_KB).

bottom_up_2(New_KB,To_be_generalised,_,Rules,Rules,New_KB) :-
        empty(To_be_generalised), !.
bottom_up_2(KB,To_be_generalised,_,RulesIN,RulesOUT,New_KB) :-
        bottom_up_1(KB,To_be_generalised,RulesIN,RulesOUT,New_KB).
```

```
bottom_up_1(KB,To_be_generalised,RulesIN,RulesOUT,New_KB) :-
        select_seed(RulesIN,To_be_generalised,Seed),
        %% selects a rule in RulesIN to be generalised
        append(KB,To_be_generalised,Temp_examples),
        Temp_Best=[Temp_examples,Seed],
        generate_complex(Seed,[Temp_Best],Temp_Best,Best),
        add_complex_to_cover(RulesIN,Seed,Best,RulesIN1),
        %% substitute Seed in RulesIN with Best; result in RulesIN1
        covered_by(Best,To_be_generalised,Covered),
        set_difference(To_be_generalised,Covered,New_To_be_generalised),
        new_examples(Best,KB,To_be_generalised,New_KB1),
        bottom_up_2(New_KB1,New_To_be_generalised,Best,RulesIN1,RulesOUT,New_KB).

%% generate_complex(Seed,StarIN,Best_Cpx,Complex)
%% bottom-up module

generate_complex(Seed,Star,Complex,Complex) :-
        termination(Star,Seed), !.

generate_complex(Seed,StarIN,Best_Cpx,Complex) :-
        select_noncov(Seed,StarIN,NonE),
        %% selects a (positive) non covered example
        generalise_star(Seed,NonE,StarIN,StarOUT1),
        subsumption_pruning(StarOUT1,StarOUT2),
        size_pruning(StarOUT2,StarOUT),
        select_best_complex(StarOUT,Best_Cpx,New_Cpx),
        generate_complex(Seed,StarOUT,New_Cpx,Complex).
```

System predicates

```
covered_by(A,B,C)          unifies C with the subset of B covered by the ruleset A
set_difference(A,B,C)      unifies C with the difference between A and B (A\B)
empty(L)                   succeeds if L is the empty list
append(A,B,C)              unifies C with the union of the two lists A and B
new_examples([A,_],B,C,D)  unifies C with the elements of A which are not in B∩C
```

Figure 2 - Alpha's top-down and bottom-up predicates

7.Bibliography

[Clark and Niblett, 1989] P. Clark and T. Niblett, *The CN2 Induction Algorithm*, Machine Learning Journal (Netherlands, 1989).

[Ginsberg, 1990] A. Ginsberg, *Theory Reduction, Theory Revision, and Retranslation*, Proceedings of the 8th National Conference on AI, 1990.

[Kowalski, 1995] R.A. Kowalski, *Using meta-logic to reconcile reactive with rational agents* (Dept. Computing, Imperial College, 1995).

[Laird and Rosenbloom, 1990] J.E. Laird and P.S. Rosenbloom, *Integrating Execution, Planning and Learning in Soar for External Environments*, Proceedings of the 8th National Conference on AI, 1990.

[Maes and Brooks, 1990] P. Maes and R.A. Brooks, *Learning to Co-ordinate Behaviours*, Proceedings of the 8th National Conference on AI, 1990.

[Maes, 1990] P. Maes, *Designing Autonomous Agents: Theory and Practice from Biology to Engineering and Back* (MIT Press, London, England, 1990).

[Muggleton, 1987] S. Muggleton, *Duce, an oracle based approach to constructive induction*, Proceedings to the 10th International Joint Conference on AI, 1987.

[Paine, 1993] J. Paine, *The EDEN/POPBEAST AI competition and teaching kit*, Logic Programming - The newsletter of the Association of Logic Programming. Volume 6/1, February 1993.

[Russell and Norvig, 1995] S.J. Russell and P. Norvig, *Artificial Intelligence: A modern approach* (Prentice Hall, Upper Saddle River, New Jersey, 1995).