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
e-mail: marcy@morgana.usr.dsi.unimi.it, lanzarone@hermes.mc.dsi.unimi.it

ABSTRACT

In this paper we study the problems involved in designing an intelligent agent that 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 aimed at 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 dedicated to describing the problems related both to the definition of the knowledge representation formalism and to 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 balancing 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 among themselves.

1.State of the art and objective definition

In this paper we introduce a new agent model which autonomously interacts with an unknown environment and builds an incremental and inductive model of it by means of symbolic learning.

This is a novel approach in that it integrates in a single model both symbolic inductive incremental learning and planning / execution; in the current literature these two activities are dealt with separately and we are aware of no attempt to investigate how the characteristics of the learning and of the planning component are influenced by the fact that they have to co-operate and to feature both real-time responses and good performance in complex environments.

Our agent is semi-reactive in that its overall structure is based upon a hybrid design, unifying the rational and the reactive agent design approaches, which is similar to the one introduced in [Kowalski, 1995] (examples on the reactive and rational approaches may be found in [Laird and Rosenbloom, 1990], [Maes and Brooks, 1990] and [Maes, 1990]).

In this paper we focus the attention on the problems associated with designing the learning component, and for sake of simplicity we won't go into detail about the problems related to resource sharing between the components of the agent. With respect to the different approaches to symbolic inductive learning ([Clark and Niblett, 1989], [Ginsberg, 1990], [Muggleton, 1987], [Quinlan, 1993], [Bratko et al., 1996], [Langley and Simon, 1995], [Lavrac and Dzeroski, 1994], [De Raedt, 1996]), we have chosen to build the learner around a propositional attribute value language extended with predicative elements. This choice guarantees both good expressive power, which in turn lets the agent handle complex environments, and reduced cost of the learning algorithm, which allows real-time responses.

In the first sections of this paper we describe the language and the reasoning techniques of the learning component of the agent. Later on we try to characterize a whole class of learners, suitable as agent's learning components, by means of an abstract learner; in order to give a concrete example of the power of this abstraction, we show how it is possible to specialize the algorithm to two well-known learners, such as CN2 and AQR (which are propositional batch learners), and to the learner of our agent (which is semi-predicative and incremental).

The agent is developed in Prolog, in the context of logic programming, and its performance is currently being tested on simulated two dimensional

[*] This research is in part supported by the ex-40% MURST Project "Knowledge Representation and Automated Reasoning".

environments by means of a client-server multi-user environment for agent development and experimentation that we have created on purpose.

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 pairs.

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 decision trees; this choice is motivated by considering that it's frequently necessary to modify only some fragments of the knowledge base during the knowledge revision phase - which is cheaper than a large-scale change - but using decision trees may often bring, even in these cases, to review the entire tree structure, which worsens 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.

The class of environments that our agent model has to cope with can be characterized, with the terminology of [Russell and Norvig, 1995], as inaccessible, deterministic, nonepisodic, static and discrete. Briefly, this principally means that the next state of the environment is completely determined by the current state and the actions selected by the agents (determinism), but that the agent's sensory apparatus doesn't necessarily detect all the aspects that are relevant to the choice of actions (inaccessibility).

Since the inaccessible and deterministic environment may appear nondeterministic to the agent, in order to keep the predictive capability of the model, we describe the state, to which the premise refers, by means of the representation of a sequence of states, beginning at time t+1 and extending back into time for a suitable number of instants, which is rule-dependent and is determined by the learner at run-time.

The rule's syntax, expressed in BNF notation, augmented with ^{n} (meaning *a positive or null number n of occurrences, less than or equal to a fixed limit n_0*) is:

$$< \text{rule} > ::= < \text{selector} > \leftarrow < \text{premise} >$$
$$< \text{premise} > ::= < \text{state} > | < \text{state} > < \text{premise} >$$
$$< \text{state} > ::=$$
$$< \text{state} > ::= (< \text{sel} >)$$
$$< \text{sel} > ::= \{< \text{selector} >,\}^n \ < \text{selector} >$$
$$< \text{selector} > ::= < \text{attribute} > = < \text{value} >$$

The temporal localization of each state in the sequence is implicitly expressed by its position in it, with the left-most state immediately following the application of the rule, and the states on its right progressively extending back into time.

The reason for including, in the state sequence of the premise, the state following the application of the rule, is that it lets us represent what we call geographical knowledge, that is the kind of knowledge needed, for example, to express the fact that in every state in which X=x^* and Y=y^*, the COLOR attribute takes the $color^*$ value.

The learner employs also a more high-level knowledge representation formalism based upon the so-called metarules, whose syntax, expressed in BNF notation, is:

$$< \text{metarule} > ::= < \text{rule} >; < \text{metapremise} >$$
$$< \text{metapremise} > ::= [< \text{rule} >] |$$
$$[< \text{rule} >], < \text{metapremise} >$$

The semantics of the metarule is:

> **if** the premise is true and **if** each rule of the metapremise has been activated at least once in the past and no opposite[1] rule to it has been activated after the last activation of its, **then** the consequence if true.

Metarules are used whenever the consequence of a rule and a part of its premise have no precise temporal link: the consequence is valid apart from the time past since the verification of that part of the premise.

As an example of the use of metarules, consider the following Wumpus World problem, inspired to the homonymous environment introduced in [Russell and Norvig, 1995].

> Given a 3x3 subset of the two dimensional 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**.

A metarule for the Wumpus World problem is:

[1] For a formal definition of the terminology used see §3.1 and §3.3.

$$\begin{aligned}\mathrm{PIT} = \mathrm{true} \leftarrow (\mathrm{X} = 0, \mathrm{Y} = 0);\\ [\mathrm{BREEZE} = \mathrm{true} \leftarrow (\mathrm{X} = 1, \mathrm{Y} = 0)],\\ [\mathrm{BREEZE} = \mathrm{true} \leftarrow (\mathrm{X} = -1, \mathrm{Y} = 0)],\\ [\mathrm{BREEZE} = \mathrm{true} \leftarrow (\mathrm{X} = 0, \mathrm{Y} = 1)],\\ [\mathrm{BREEZE} = \mathrm{true} \leftarrow (\mathrm{X} = 0, \mathrm{Y} = -1)]\end{aligned}$$

The learner provides a dynamical *bias shift* mechanism for transforming rules into metarules and for reverting back to the rule form when necessary (hence the term *dynamic*). The heuristics ruling this device are still under development.

3.The learning module of the agent

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

In the area of batch propositional learners the most commonly used searching algorithm is the least-commitment search, and especially the beam search, since this method has the feature of delaying 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 approach could have been delayed until the acquisition of the whole dataset, and this involves resorting again to backtracking.

Because of this, it is no longer possible to choose to operate exclusively with generalizations or specializations, as it happens in batch learning. The search space has to be traversed both bottom-up (generalizing) and top-down (specializing).

In addition the learner's knowledge base must be immediately usable to the planner, in an application 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 information needed to extend back into time the state sequence of the rule' s premise and to build the metarule's metapremise.

Note that each rule is part of a rule structure, which is a more complex data item, holding the rule itself and other information fields, introduced later on. From now on, we will use the term rule for addressing both the rule itself and the rule structure, unless it is confusing.

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.Specialization

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 a consequence of B) if A and B have equal consequences and the premise domain of A is a subset of the premise domain of B.

Specialization 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 specialization operate on pairs of inconsistent rules, having non-empty intersection domains, in order to make the intersection set empty.

Specialization 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 of 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 specializes 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 specializes the first rule on the basis of the second, by adding to it the selectors of the

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

$$\mathrm{ATTR} = \mathrm{val} :- \mathrm{ATTR} = \mathrm{val}$$

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

first rule's extended premise having an opposite in the second rule's premise.

The third operator specializes 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 the wrong choices made in the generalization 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 generalized, taking into account the presence of the first rule, which prevents reconstructing the original second rule.

The four operators are applied in increasing order of power. Examples of the specialization operators may be found in Table 5.

From the point of view of the specialization'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.Generalization

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

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

Generalization acts by means of three operators.

The first operator controls if the second rule follows from the first, in which case it 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 it 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 that both follow from. Basically its premise contains the intersection between the premises of both rules and the extended premise is created in the same way. The example set is simply the union of the starting rules' sets. Before the substitution takes place, the new rule is verified to be consistent with the knowledge base.

The third operator has the task of preventing the learner from remaining blocked in blind alleys because of some attribute value pairs wrongly introduced into 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 performance was obtained with more specific initial premises, too.

The three operators are applied in increasing order of power. Examples of the generalization operators may be found in Table 6.

Like specialization, generalization is also 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 non-deterministically. From this it follows that in some situations specialization 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 specialize their premises any more.

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

What we do is mark 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 becomes 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 the consequence also 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, 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 specialization and generalization 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 the description of 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 the searching algorithm in general, distinguishing between the two versions only when necessary.

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

4.1.Algorithm description

The core of Alpha is based on a beam search in the hypothesis space which at each recursion specializes or generalizes the current star on the basis of the learner defined criteria (*select_negex*, *specialize_star*, *select_noncov*, *generalize_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 *specialize_star*, as shown in [Clark and Niblett, 1989].

The bottom-up search predicate [bottom_up(KB, To_be_generalized, Rules, New_KB)] receives two rulesets (KB and To_be_generalized) and returns a ruleset (Rules) which generalizes 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 with 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 *specialize_star* (or by *generalize_star*). Thanks to this device Alpha can specialize (or generalize) both the complexes under construction and parts of the knowledge base already present, thus reaching incrementality.

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

Another subtlety we want to point out is that the implementation of Alpha is founded on the assumption that examples can't be further specialized; this allows the direct resolution of the need to operate on the union of KB and To_be_classified, while specializing only the elements of KB. Until now we have found no learner violating this assumption, nor do we see a reason why this should happen in the 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 the terminology used, such as star, complexes, etc. and for a more detailed 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 the condition that the covers predicate has the following property (which seems reasonable):

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

The use of the KB and To_be_classified sets depends upon the kind of ruleset adopted (**ordered** or **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 7 shows the derivation of the incremental learner Gamma. The specialization and generalization operators implementing *specialize_star* and *generalize_star* are those described in §3. As you may note Gamma operates with a fixed value of MaxStar, equal to one.

An advantage of Alpha is that it lets us <u>compare</u> the learners with each other, even when apparently they have nothing in common, and obtain useful indications on the possible changes 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 by 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 area of **incremental** learning.

5.Implementation status and final observations

The planning component of the agent is still under development. A great advantage by which our planner benefits is the availability of the information the learner records in order to extend back into time the state sequence of the rule's premise. This make it 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; an experimental server which users can connect to by means of a graphical client written in Java is currently available. Authorized users can connect to the server using any network browser and create their own experiments or observe other users' experiments (a more restricted anonymous access is also available). An experiment is made up of a two dimensional world, whose topology and object behaviour may be defined at will, and an agent, which may itself be defined as desired.

A small 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 the Internet address:

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

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

6.References

[Bratko et al., 1996] I. Bratko, B. Cestnik and I. Kononenko, *Attribute-based learning,* AI Communications 9, 27-32 (IOS Press, 1996).

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

[De Raedt, 1996] L. De Raedt, *Advances in Inductive Logic Programming* (IOS Press, Amsterdam, Netherlands, 1996).

[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.

[Langley and Simon, 1995] P. Langley and H.A. Simon, *Applications of Machine Learning and Rule Induction*, Communications of the ACM. Volume 38/11, November 1995.

[Lavrac and Dzeroski, 1994] N. Lavrac and S. Dzeroski, *Inductive Logic Programming. Techniques and Applications* (Ellis Horwood, New York, USA, 1994).

[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.

[Quinlan, 1993] J.R. Quinlan, *C4.5: Programs for Machine Learning* (Morgan Kaufmann Publishers, Los Altos, California, 1993).

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

7.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.

Predicate	Description
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.
specialize_star	Applies the specialization operators to the star.
generalize_star	Applies the generalization operators to the star.
subsumption_pruning	Removes from the star the complexes subsumed by other complexes in the star.
size_pruning	Removes the less significant complexes from the star until the star's size is below the imposed threshold.
select_best_complex	Chooses the best complex between the best in the 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.
specialize_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 the star.
size_pruning	Removes the less significant complexes until the star's size is less than or equal to 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).
specialize_star	Given Star, the new StarOUT is built in this way:

Predicate	Definition
	$\begin{cases} Extension = \{sel \mid \text{covers}(sel, Seed) \wedge \neg \text{covers}(sel, NegE)\} \\ StarOUT = \{x \wedge y \mid x \in Star,\ y \in Extension\} \end{cases}$
subsumption_pruning	Removes all complexes subsumed by other complexes in the star and all unsatisfiable complexes.
size_pruning	Removes the less significant complexes until the star's size is less than or equal to the user defined threshold.
select_best_complex	Returns the best complex between the best in the 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.
specialize_star	Applies Gamma's specialization operators to the star.
generalize_star	Applies Gamma's generalization operators to the star.
subsumption_pruning	Removes all the complexes subsumed by other complexes in the star and all unsatisfiable complexes.
size_pruning	Removes the less significant complexes until the star's size is less than or equal to **one**.
select_best_complex	Returns the only complex contained in the star.

Table 5 - Examples of the specialization operators

S1 (first specialization operator)	S2 (second specialization operator)
$\text{A:}\begin{cases} \text{COLOR} = \text{RED} \leftarrow (\text{X} = 0,\ \text{Y} = 1) \\ \text{extended premise: } (\text{X} = 0,\ \text{Y} = 1) \end{cases}$ $\text{B:}\begin{cases} \text{COLOR} = \text{BLUE} \leftarrow (\text{X} = 0) \\ \text{extended premise: } (\text{X} = 0,\ \text{Y} = 0) \end{cases}$ $\Rightarrow$ $\text{B}':\begin{cases} \text{COLOR} = \text{BLUE} \leftarrow (\text{X} = 0,\ \text{Y} = 0) \\ \text{extended premise: } (\text{X} = 0,\ \text{Y} = 0) \end{cases}$	$\text{A:}\begin{cases} \text{COLOR} = \text{RED} \leftarrow (\text{X} = 0) \\ \text{extended premise: } (\text{X} = 0,\ \text{Y} = 1) \end{cases}$ $\text{B:}\begin{cases} \text{COLOR} = \text{BLUE} \leftarrow (\text{X} = 0,\ \text{Y} = 0) \\ \text{extended premise: } (\text{X} = 0,\ \text{Y} = 0) \end{cases}$ $\Rightarrow$ $\text{A}':\begin{cases} \text{COLOR} = \text{RED} \leftarrow (\text{X} = 0,\ \text{Y} = 1) \\ \text{extended premise: } (\text{X} = 0,\ \text{Y} = 1) \end{cases}$

S3 (third specialization operator)	S4 (fourth specialization operator)
$A: \begin{cases} \text{COLOR} = \text{RED} \leftarrow (X = 0) \\ \text{extended premise: } (X = 0, Y = 1) \end{cases}$ $B: \begin{cases} \text{COLOR} = \text{BLUE} \leftarrow (X = 0) \\ \text{extended premise: } (X = 0, Y = 0) \end{cases}$ $\Rightarrow$ $A': \begin{cases} \text{COLOR} = \text{RED} \leftarrow (X = 0, Y = 1) \\ \text{extended premise: } (X = 0, Y = 1) \end{cases}$ $B': \begin{cases} \text{COLOR} = \text{BLUE} \leftarrow (X = 0, Y = 0) \\ \text{extended premise: } (X = 0, Y = 0) \end{cases}$	$A: \begin{cases} \text{COLOR} = \text{RED} \leftarrow (X = 0, Y = 1) \\ \text{extended premise: } (X = 0, Y = 1) \end{cases}$ $B: \begin{cases} \text{COLOR} = \text{BLUE} \leftarrow (X = 0) \\ \text{extended premise: } (X = 0) \\ \text{examples: } \left\langle \begin{matrix} (X = 0, Y = 0) \\ (X = 0, Y = 2) \end{matrix} \right. \end{cases}$ $\Rightarrow$ B: erased $B'_1: \begin{cases} \text{COLOR} = \text{BLUE} \leftarrow (X = 0, Y = 0) \\ \text{extended premise: } (X = 0, Y = 0) \end{cases}$ $B'_2: \begin{cases} \text{COLOR} = \text{BLUE} \leftarrow (X = 0, Y = 2) \\ \text{extended premise: } (X = 0, Y = 2) \end{cases}$

Table 6 - Examples of the generalization operators

G1 (first generalization operator)	G2 (second generalization operator)
$A: \begin{cases} \text{COLOR} = \text{BLUE} \leftarrow (X = 0) \\ \text{extended premise: } (X = 0, Y = 1) \end{cases}$ $B: \begin{cases} \text{COLOR} = \text{BLUE} \leftarrow (X = 0, Y = 0) \\ \text{extended premise: } (X = 0, Y = 0) \end{cases}$ $\Rightarrow$ $A': \begin{cases} \text{COLOR} = \text{BLUE} \leftarrow (X = 0) \\ \text{extended premise: } (X = 0) \\ \text{examples: } \left\langle \begin{matrix} (X = 0, Y = 1) \\ (X = 0, Y = 0) \end{matrix} \right. \end{cases}$ B erased	$A: \begin{cases} \text{COLOR} = \text{BLUE} \leftarrow (X = 0, Y = 1) \\ \text{extended premise: } (X = 0, Y = 1) \end{cases}$ $B: \begin{cases} \text{COLOR} = \text{BLUE} \leftarrow (X = 0) \\ \text{extended premise: } (X = 0, Y = 0) \end{cases}$ $\Rightarrow$ $B': \begin{cases} \text{COLOR} = \text{BLUE} \leftarrow (X = 0) \\ \text{extended premise: } (X = 0) \\ \text{examples: } \left\langle \begin{matrix} (X = 0, Y = 1) \\ (X = 0, Y = 0) \end{matrix} \right. \end{cases}$ A erased

G3 (third generalization operator) $\text{A:}\begin{cases}\text{COLOR} = \text{BLUE} \leftarrow (\text{X} = 0,\ \text{Y} = 1) \\ \text{extended premise: } (\text{X} = 0,\ \text{Y} = 1)\end{cases}$ $\text{B:}\begin{cases}\text{COLOR} = \text{BLUE} \leftarrow (\text{X} = 0,\ \text{Y} = 0) \\ \text{extended premise: } (\text{X} = 0,\ \text{Y} = 0)\end{cases}$ $\Rightarrow$ $\text{C}'\text{:}\begin{cases}\text{COLOR} = \text{BLUE} \leftarrow (\text{X} = 0) \\ \text{extended premise: } (\text{X} = 0) \\ \text{examples:}\left\langle\begin{matrix}(\text{X} = 0,\ \text{Y} = 1) \\ (\text{X} = 0,\ \text{Y} = 0)\end{matrix}\right.\end{cases}$ A, B erased	

Table 7 - 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),
       specialize_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_generalized,RulesOUT,New_KB) :-
       bottom_up_1(KB,To_be_generalized,[],RulesOUT,New_KB).

bottom_up_2(New_KB,To_be_generalized,_,Rules,Rules,New_KB) :-
       empty(To_be_generalized), !.
bottom_up_2(KB,To_be_generalized,_,RulesIN,RulesOUT,New_KB) :-
       bottom_up_1(KB,To_be_generalized,RulesIN,RulesOUT,New_KB).
bottom_up_1(KB,To_be_generalized,RulesIN,RulesOUT,New_KB) :-
       select_seed(RulesIN,To_be_generalized,Seed),
       %% selects a rule in RulesIN to be generalized
```

```
        append(KB,To_be_generalized,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_generalized,Covered),
        set_difference(To_be_generalized,Covered,New_To_be_generalized),
        new_examples(Best,KB,To_be_generalized,New_KB1),
        bottom_up_2(New_KB1,New_To_be_generalized,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
        generalize_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