Some Active Learning Schemes to Acquire Control Knowledge for Planning *

Raquel Fuentetaja and **Daniel Borrajo**
Departamento de Informática, Universidad Carlos III de Madrid
Avda. de la Universidad, 30. Leganés (Madrid). Spain
rfuentet@inf.uc3m.es, dborrajo@ia.uc3m.es

Abstract

Active learning consists on the incremental generation of "good" training examples for machine learning techniques. Usually, the objective is to balance between cost of generating and analyzing all the instance space, and cost of generation of "good" examples. While there has been some work on its application to inductive learning techniques, there have been very few approaches applying it to problem solving, and even less to task planning. In this paper, we present an on-going work on building some such schemes for acquiring control knowledge, in the form of control rules, for a planner. Results show that the scheme improves not only in terms of learning convergence, but also in terms of performance of the learned knowledge.

Introduction

The field of Active learning (AL) has been largely studied in the literature of inductive propositional learners (Cohn, Ghabhramani, & Jordan 1996; Liere, Tadepalli, & Hall 1997; Lin & Shaw 1997; Mitchell, Utgoff, & Banerji 1983). As defined by (Bryant *et al.* 2001), its aim is to generate an ordered set of instances (trials, experiments, experiences), such that we minimise the expected cost of eliminating all but one of the learning hypotheses. The task is NP-hard as Fedorov (Fedorov 1972) showed. Therefore, AL requires a balance between the cost of selecting the ordered set and the cost of exploring all instances.

Several ML techniques have been implemented to acquire knowledge for planning. They go from ML of control knowledge, ML of quality-based knowledge, to ML of domain knowledge. In (Zimmerman & Kambhampati 2003), the reader can find a good overview. In planning, the standard way of generating training examples has been providing a set of problems to planning-learning systems. Planners solve those problems one at a time, and learning techniques generate knowledge taking into account the search tree or the solution to the problems. One important issue to consider is that, opposite to what is common in AL for inductive learning, instances are not the problems themselves, but they are extracted from the process of solving them (like EBL techniques (Minton 1988).

Training problems are usually created by a random generator that have a set of parameters. These parameters theoretically define the problems difficulty. Usually, these parameters can be domain-independent (number of goals), or domain-dependent (number of objects, trucks, cities, robots, ...). The advantage of this scheme for generating training problems is its simplicity. As disadvantages, the user needs to adjust the parameters in such a way that learning extracts as much as possible from problems solved. Also, since problems are generated randomly, most of them will not be useful for learning. On one extreme, they are so difficult that the base planner cannot solve them. If the planner cannot solve them, no training examples will be generated. On the other extreme, if problems are so easy that the planner obtains the best solution without any wrong decisions, it is hard (for some learning techniques) to learn anything. This could be no problem for macro-operators acquisition (Fikes, Hart, & Nilsson 1972) or CBR, since they learn from solutions. However, learning techniques that are based on decisions made in the search tree can have difficulties on generating training examples when the search trees do not include failure decisions. This is the type of learning techniques that we use in this paper: they learn control knowledge from search tree decisions (Borrajo & Veloso 1997).

In order to solve those disadvantages, we propose in this paper several methods for AL that propose new training problems to the Machine Learning (ML) techniques. The first approach generates problems with an increasing difficulty. The second approach generates new training problems with an increasing difficulty, based on the previous generated problem. Finally, the third approach generates new problems, also with increasing difficulty, based on the previous problem and the learned knowledge.

The first two sections provide an overview of the planner and learning technique we use. Then we describe

*This work has been partially supported by the Spanish MCyT project TIC2002-04146-C05-05, MEC project TIN2005-08945-C06-05 and regional CAM-UC3M project UC3M-INF-05-016.

the AL methods. The next section relates to previous work. After this, we present some experiments and a discussion of the results. Finally, we draw some conclusions and propose future work.

The Planner. IPSS

IPSS is an integrated tool for planning and scheduling (Rodríguez-Moreno *et al.* 2004). It uses QPRODIGY as the planner component, one version of the PRODIGY planner that is able to handle different quality metrics. The planner is integrated with a constraint-based scheduler (Cesta, Oddi, & Smith 2002) that reasons about time and resource constraints. We have not yet used the learning tools in combination with the scheduler, so we will focus now only on the planning component of IPSS. The planning component is a nonlinear planning system that follows a means-ends analysis (see (Veloso *et al.* 1995) for details on the planning algorithm). It performs a kind of bidirectional depth-first search (subgoaling from the goals, and executing operators from the initial state), combined with a branch-and-bound technique when dealing with quality metrics. The inputs to IPSS are the standard ones for planners (except for the heuristics, which are not usually given as explicit input in most planners):

- domain theory: represented as a set of operators and a set of types;
- problem: represented as an initial state, a set of goals to be achieved, and a set of instances of the domain types;
- domain-dependent heuristics or control knowledge: a set of control-rules that specify how to make decisions in the search tree; and
- parameters: such as time bound, node bound, depth bound, or how many solutions to generate.

The output of the planner is a valid total-ordered plan; a sequence of instantiated operators such that, when executed, transform the initial state of the problem in a state in which the goals are true (achieved).

Suppose that we are working on the well-known (in planning literature) logistics domain (Veloso 1994). In this simplification of a real world domain, objects have to be transported from their initial locations (either post offices or airports) to their final destination, by using either trucks (cannot move among cities) or airplanes (can only fly among city airports).

Figure 2 shows a control-rule (heuristic) that determines when the `unload-airplane` operator has to be selected. The conditions (left-part) of control-rules refer to facts that can be queried to the meta-state of the search. The most usual ones are knowing whether: some literal is true in the current state (`true-in-state`), some literal is a pending goal (`current-goal`) or some instance is of a given type (`type-of-object`). But, the language for representing heuristics also admits coding user-specific functions. As an example, we have defined the `different-vars-p` meta-predicate that checks whether all different variables of the rule are bound to different values.

The planner does not incorporate powerful domain-independent heuristics, given that the main goal of PRODIGY was not building the most efficient planner, but the study of ML techniques applied to problem solving. The idea was that the architecture would incrementally learn knowledge (such as heuristics, cases, domain models, ...) through the interaction between the planner and the learning techniques, such that this knowledge would make it efficient.

The Learning Technique. HAMLET

Over the years, some ML techniques have been implemented within PRODIGY architecture. Examples range from pure EBL (Minton 1988) to CBR (Veloso 1994) or genetic programming (Aler, Borrajo, & Isasi 2002). HAMLET is one such ML methods, that is an incremental learning method based on EBL and inductive refinement of control-rules (Borrajo & Veloso 1997). The inputs to HAMLET are a task domain ($\mathcal{D}$), a set of training problems ($\mathcal{P}$), a quality measure (Q) and other learning-related parameters. Internally, HAMLET calls IPSS and also receives as input the search tree expanded by the planner, in order to decide what to learn. HAMLET output is a set of control-rules ($\mathcal{C}$) that potentially guide the planner towards *good* quality solutions. HAMLET has two main modules: the Analytical learning (EBL) module, and the Inductive learning (ILP) module. Figure 1 shows HAMLET modules and their connection to the planner.

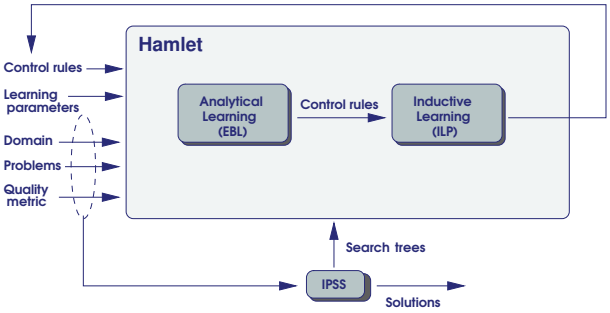

Figure 1: HAMLET high level architecture.

The Analytical learning module generates control-rules from an IPSS search tree by finding positive examples of decisions; that is, decisions that lead to the best solution, instead of a worse solution, or a failure path. Once a positive example of a decision is found, a control-rule is generated by performing an EBL step, modified to take into account the specifics of non-linear planning, such as considering the effect of the other pending goals. Therefore, HAMLET extracts the meta-state of the search, and performs a goal regression for finding which literals from the current state were needed

to be true to make this decision (the details can be found in (Borrajo & Veloso 1997)).

These EBL-like rules might be overly-specific (Etzioni & Minton 1992) or overly-general. Most previous work focused on this problem as a utility problem (learning many control-rules degrades the performance of the planner when using them) (Minton 1988). However, HAMLET was the first ML technique that focused it through an inductive solution; if rules were overly-specific, HAMLET would solve more training problems and find more positive examples similar to the previously learned ones, such that HAMLET could generalize from the overly-specific examples. Similarly, if rules were overly-general, when solving new problems, the control-rules would fire when they should not, guiding IPSS towards failure paths or bad solutions. Then, HAMLET would generate negative examples of the control-rules. HAMLET would then specialize the overly-general control-rules, such that they do not longer cover the negative example. The generalization and specialization processes are performed by the Inductive learning module, that performs a kind of ILP process. HAMLET gradually learns and refines control-rules, in an attempt to converge to a concise set of correct control-rules (i.e., rules that are individually neither overly general, nor overly specific).

Figure 2 shows an example of a rule automatically learned by HAMLET in the logistics domain for selecting the `unload-airplane` operator. As it is, the control-rule says that if the goal is to have an object in a given location, `<location1>`, and the object is currently inside an airplane, then IPSS should use the `unload-airplane` instead of the `unload-truck` that also achieves the same goal (having an object in a given location). HAMLET learns this rule by first generating a rule that would require the airplane to be in another airport from `<location1>` (conditions computed from the goal regression), then learning another similar rule from a problem in which the airplane is at the same `<location1>`, and finally inducing a new more general rule from these other two (and removing the other two more specific rules).

```
(control-rule induced-select-operators-unload-airplane
  (if (and (current-goal (at <object> <location1>))
           (true-in-state (inside <object> <airplane>))
           (different-vars-p)
           (type-of-object <object> object)
           (type-of-object <airplane> airplane)
           (type-of-object <location1> airport)))
  (then select operator unload-airplane))
```

Figure 2: Example of a control-rule learned by HAMLET for selecting the `unload-airplane` operator in the logistics domain.

Active Learning for Planning

As any ML inductive tool, HAMLET needs a set of training problems to be given as input. The standard solution from the ML perspective is that the user provides as input a set of relevant planning problems. As any other ML technique, learning behaviour will depend on how similar those problems are to the ones that IPSS would need to solve in the future. However, in most real world domains, these problems are not available. So, an alternative solution consists on defining a domain-dependent generator of random problems. Then, one can assume (ML theory assumes) that if HAMLET sees enough random problems covering reasonably well the space of problems, and it learns from them, the learned knowledge will be reasonably adapted to the future. This solution requires to build one such generator for each domain, though there is some work on automating this task (McCluskey & Porteous 1997). However, a question remains: what type of problems are the most adequate ones to train a ML technique for problem solving. Here, we propose the use of active learning schemes. In this approach, HAMLET would need to select at each step what are the characteristics that the next learning problem should have in order to improve the learning process. Then, HAMLET can call a problem generator with these characteristics as input, so that the problem generator returns a problem that is expected to improve learning. Examples of these characteristics in the logistics domain might be number of objects, number of goals, or number of cities. We have implemented four ways of generating training problems for HAMLET. Only the last one is really an AL scheme in the sense that it has feedback from the learning knowledge. They are: one-step random generation; increasing difficulty generation; generation based on the last solved problem; and generation based on the rules learned. They are described in more detail in the next sections.

One-Step Random Generation

This is the standard way used in the literature of ML applied to planning. Before learning, a set of random training problems is generated at once, whose difficulty is defined in terms of some domain-independent characteristics (as number of goals) or domain-dependent characteristics (as number of objects to move). Usually, training is performed with simple problems, and generalization is tested by generating more complex test problems, also randomly.

Increasing Difficulty Generation

In this first AL scheme, at each cycle, a random generator is called to obtain a new training problem. Then, if the problem is suitable for learning, HAMLET uses it. Otherwise, the problem is discarded, and a new problem is randomly generated. The first difference of this second scheme with the previous version is that problems are incrementally generated and tested. Only the ones that pass the filter will go to the learner. As an example of a filter, we can define it as: select those problems

such that they are solved, and the first path that the planner explores does not contain the solution. This means at least one backtracking had to be performed by the planner, and, therefore, at least one failure node will appear in the search node. This simple test for validity will be valid if we try to learn knowledge that will lead the planner away from failure. However, if we want to learn knowledge on how to obtain "good" solutions, we have to use a more strict filter. Then, we can use IPSS ability to generate the whole search tree, to define another filter as: select those problems such that they are solved, the complete search tree was expanded, and the first path that the planner explores does not contain the solution. This filter allows HAMLET to learn from decisions (nodes) in which there were two children that lead to solutions: one with a better solution than the other. So, it can generate training instances from those decisions.

The second difference with respect to the random generation, is that the user can specify some parameters that express the initial difficulty of the problems, and some simple rules to incrementally increase the difficulty (difficulty increasing rules, *DIRs*). An example of the definition of an initial difficulty level, and some rules to increase the difficulty is shown in Figure 3. The level of difficulty (Figure 3(a)) is expressed as a list of domain-dependent parameters (number of objects, initially 1, number of cities, initially 1, ...) and domain-independent parameters (number of goals, initially 1). The rules for increasing the level of difficulty (Figure 3(b)) are defined as a list of rules, each rule is a list formed by the parameters to be increased by that rule and in what quantity. The first DIR, `((object 1) (no-goals 1))`, says that in order to increase the difficulty with it, the AL method should add 1 to the current number of objects, and add 1 to the current number of goals. Therefore, the next problem to be generated will be forced to have 2 objects and 2 goals.

The AL method generates problems incrementally, and filters them. If the learning system has spent n (experimentally set as 3) problems without learning anything, the system increases the difficulty using the DIRs. We explored with other values for n, though they lead to similar behaviour. The effect of increasing n is to spend more time on problems of the same difficulty, where we could have potentially explored all types of different problems to learn from. If we decrease n, then it forces the AL method to increase the difficulty more often, leading potentially to not exploring all types of problems within the same difficulty level.

Generation Based on the Last Problem

This AL method works on top of the previous one. In each level of difficulty, the first problem is generated randomly. Then, each following problem is generated by modifying the previous one. In order to modify it, the AL method performs the following steps:

- it selects a type t from the domain (e.g. in the logistics: truck, object, airplane) randomly from the set of domain types;
- it randomly selects an instance i of that type t from the set of problem-defined instances. For instance, if the previous step has selected `object`, then it will randomly choose from the defined objects in the problem (e.g. $\text{object}_1, \ldots \text{object}_n$);
- it randomly selects whether to change the initial state or the goal of the problem;
- it retrieves the literal l in which the chosen instance appear in the state/goal. For instance, suppose that it has selected $\texttt{object}_i$, such that $\texttt{(in object}_i\ \texttt{airplane}_j\texttt{)}$ is true in the initial state;
- it randomly selects a predicate p from the predicates in which the instances of the chosen type can appear as arguments. For instance, objects can appear as arguments in the predicates: `at` and `in`;
- it changes in the state/goal of the problem the selected literal l for a new literal. This new literal is formed by selecting randomly the arguments (instances) of the chosen predicate p, except for the argument that corresponds to the chosen instance i. Those random arguments are selected according to the corresponding types of those arguments. For instance, suppose that it has chosen $\texttt{at}_{object}$. $\texttt{at}_{object}$ definition as predicate is $\texttt{(at}_{object}\ \texttt{?o - object ?p - place)}$, where `place` can be either `post-office` or `airport`. Then, it will change $\texttt{(in object}_i\ \texttt{airplane}_j\texttt{)}$ for $\texttt{(at object}_i\ \texttt{airport}_k\texttt{)}$, where $\texttt{airport}_k$ is chosen randomly.

This AL method assumes that it has knowledge on:

- types whose instances can change "easily" of state. For instance, in the logistics domain, objects can be easily changed of state by randomly choosing to be `at` another place or `in` another vehicle. However, in that domain, trucks cannot move so easily, given that they "belong" to cities. They can only move within a city. Therefore, in order to change the state of trucks, we would have to select a place of the same city where they are initially. This information could potentially be learned with domain analysis techniques as in TIM (Fox & Long 1998). Now, we are not currently considering these types.
- predicates where a given type can appear as argument. This is specified indirectly in PDDL (current standard for specifying domain models) (Fox & Long 2002), and can be easily extracted (predicates are explicitly defined in terms of types of their arguments).
- arguments types of predicates. As described before, this is explicitly defined in PDDL.

As an example of this AL technique, if we had the problem defined in Figure 4(a), this technique can automatically generate the new similar problem in Figure 4(b). The advantage of this approach is that if rules were learned from the previous problem, the new

```
((object 1) (city 1) (plane 1)
 (truck 1) (goals 1))
```

(a)

```
(((object 1) (no-goals 1))
 ((city 1) (truck 1))
 ((object 1)  (no-goals 1))
 ((plane 1)))
```

(b)

Figure 3: Example of (a) difficulty level and (b) rules for increasing the difficulty.

problem forces HAMLET to learn from a similar problem, potentially generalizing (or specializing) the generated rules. And this can eventually lead to generating less problems (more valid problems for learning will be generated) and better learning convergence, by using gradual similar training problems.

Generation Based on the Rules Learned

In this last scheme of AL, the next training problem is generated taking into account the learned control rules of the previous problem. The idea is that the random decisions made by the previous AL scheme should also take into account the literals, predicates, instances and types that appear in the left-hand side of rules in the last problem. Therefore, we compute the frequency of appearance of each predicate, type and instance before performing the parametrization step when generating each precondition of each control rule learned. Then, instead of randomly choosing a type, instance, and new predicate for the next problem, we do it using a roulette mechanism based on the relative frequency of them.

Suppose that HAMLET has only learned the two rules in Figure 5 from a given problem (they are shown before converting the instances to variables).

```
(control-rule select-operators-unload-airplane
  (if (and (current-goal (at ob0 a0))
           (true-in-state (inside ob0 pl0))
           (different-vars-p)
           (type-of-object ob0 object)
           (type-of-object pl0 airplane)
           (type-of-object a0 airport)))
  (then select operator unload-airplane))

(control-rule select-bindings-unload-airplane
  (if (and (current-goal (at ob0 a0))
           (current-operator unload-airplane)
           (true-in-state (at ob0 a0))
           (true-in-state (at pl0 a0))
           (true-in-state (at pl1 a0))
           (different-vars-p)
           (type-of-object ob0 object)
           (type-of-object pl0 airplane)
           (type-of-object pl1 airplane)
           (type-of-object a0 airport)))
  (then select bindings ((<object> . ob0)
                         (<airplane> . pl0)
                         (<airport> . a0))))
```

Figure 5: Examples of learned rules by HAMLET.

Then, Table 1 shows the computed frequencies. The frequency of each instance, predicate and type is the number of times it appears in a true-in-state in the preconditions of the learned rules. For example, the frequency of the instance *ob*0 is two because it appears twice (in a true-in-state in the first rule and in another one in the second rule). The percentage over the same type is 100% because there are no other instance of the type `object` different than *ob*0 in the true-in-state metapredicates of these rules. Regarding the predicates, three true-in-state metapredicates contain the predicate `at` and one the predicate `inside`. Therefore, their percentages are 75% and 25% respectively. As it can be seen, there are some predicates (at), types (airport and airplane), or instances (ob0 and a0) that have more probability of being changed than others. This is based on the observation that they appear more frequently in the preconditions of the learned control rules. Since, the goal is to improve the convergence process, new problems are randomly generated from previous ones, but in such a way that they will more probably generate new training instances of control rules that will force old control rules to be generalized (from new positive examples), or specialized (from new negative examples of the application of the control rules). The difference with the previous approach is that if what is changed from one problem to the next one did not appear on the LHS of rules (it did not matter for making the decission), it would not help on generalizing or specializing control rules.

Table 1: Example of frequency table in true-in-state

Instances	Frequency	Percentage (over the same type)
ob0	2	100%
pl0	2	66%
pl1	1	33%
a0	3	100%

Predicates	Frequency	Percentage
at	3	75%
inside	1	25%

Types	Frequency	Percentage
object	2	25%
airport	3	37%
airplane	3	37%

```
(create-problem (name prob-0)
  (objects (ob0 object) (c0 city) (po0 post-office)
           (a0 airport) (tr0 truck) (pl0 airplane))
  (state (and (same-city a0 po0) (same-city po0 a0)
              (at tr0 a0) (in ob0 pl0) (at pl0 a0)))
  (goal (at ob0 po0)))
```

(a)

```
(create-problem (name prob-1)
  (objects (ob0 object) (c0 city) (po0 post-office)
           (a0 airport) (tr0 truck) (pl0 airplane))
  (state (and (same-city a0 po0) (same-city po0 a0)
              (at tr0 a0) (at ob0 a0) (at pl0 a0)))
  (goal (at ob0 po0)))
```

(b)

Figure 4: Example of (a) problem solved and (b) problem automatically generated by AL. The difference appears in bold face.

Related Work

One of the first approaches for AL in problem solving was the LEX architecture (Mitchell, Utgoff, & Banerji 1983). It consisted of four interrelated steps of problem solving, AL, and learning: generation of a new suitable learning episode (e.g. a new integral to be solved), problem solving (generation of the solution to the integral), critic (extraction of training examples from search tree), and learning (generation from examples of version space of when to apply a given operator to a particular type of integral). In this case, AL was guided by the version space of each operator. Our approach is based on that work. However, since we do not have a version space for each type of control rule (target concept, equivalent to the operators in LEX), we have defined other approaches for guiding the AL process.

Other related work has been the one reported in (Bryant *et al.* 2001) where they use of active learning for ILP for genomics applications. It is similar to our approach in that learned knowledge also has a relational representation. However, their task was a classification task, instead of acquisition of control knowledge for planning. Other AL approaches have also focused on classification tasks using different inductive propositional techniques (Cohn, Ghabhramani, & Jordan 1996; Liere, Tadepalli, & Hall 1997; Lin & Shaw 1997). Usually, AL selects instances based on their similarity to other previous instances, or to their distance to frontiers among classes. In our case, instances are extracted from a search tree, while AL generates problems to be solved, and from whose training examples are extracted. Therefore, for us it is not quite obvious how to use those techniques to generate valid instances (problem solving decisions) from previous ones. Another alternative could have been to generate new potential hypotheses (control rules) directly from current hypothesis by modifying them. However, we have prefered to work on the generation of problems, given that from problem solving we can extract valid instances more easily, rather than on the generation of hypotheses directly.

Finally, work on reinforcement learning also has dealt with the problem of how to obtain new training instances. This is related to the exploitation vs. exploration balance, in that, at each decision-action step, these techniques have to decide whether to re-use past experience by selecting the best option from previous experience (exploitation) or select an unknown state-action pair and allow learning (exploration).

Experiments and Results

This section presents some preliminary results comparing the four ways of generating random problems described previously. We have performed the experiments using the original *logistics* domain (Veloso 1994).

In the *One-step Random Generation* approach, we generated a set of 100 random training problems. In this process, the problem's difficulty is defined as parameters of the random generator. In the logistics random generator, these parameters are the number of goals, the number of objects, the number of cities and the number of planes. Thus, we set all as numbers randomly chosen between: 1 and 5 for goals, 1 and 2 for objects, 1 and 3 for cities and 1 and 2 for planes. In the AL approaches, the "a priori" generation of random problems is not needed. We only specify the initial difficulty level and the rules for increasing the difficulty. The definition of the parameters used in these experiments is shown in Figure 3. The number of problems without learning anything before increasing the difficulty was fixed to 3. The maximum number of valid training problems for all the other schemes was also fixed to 100.

Then, we randomly generated a set of 120 test problems, composed of:

- test-1: 20 problems with 1 goal, 5 objects, 5 cities and 1 plane.
- test-2: 20 problems with 2 goals, 6 objects, 4 cities and 1 plane.
- test-3: 20 problems with 3 goals, 7 objects, 4 cities and 1 plane.
- test-5: 20 problems with 5 goals, 10 objects, 10 cities and 1 plane.
- test-10: 20 problems with 10 goals, 15 objects, 15 cities and 1 plane.
- test-15: 20 problems with 15 goals, 20 objects, 20 cities and 1 plane.

In the learning phase, we let HAMLET learn control rules using the schemes: *One-step Random Generation* (OSRG), *Increasing Difficulty Generation* (IDG), *Generation based on the last Problem* (GBP), and *Generation based on the Rules learned* (GBR). In each case,

we saved the control rules learned when 5, 25, 50, 75 and 100 training problems were used.

The IPSS planner was run on the test set for all the sets of control rules saved and for all the schemes. Figure 6 shows the obtained learning curve. In this figure, the y-axis represents the number of test problems (over 120) solved by the planner in each case, while the x-axis represents the number of training problems used for learning.

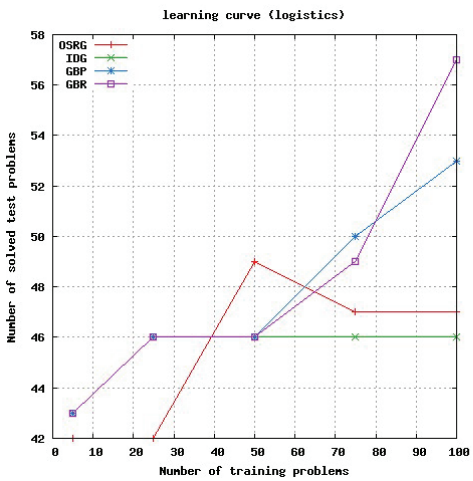

Figure 6: Learning curve in logistics domain

As one would expect, the scheme with the best convergence is GBR, followed by GBP. These results confirm that the more informative the generation of each next training problem is, the better the convergence of the learning process. It can be observed that the improvement in the convergence of these schemes begins in training problem 75. When 50 training problems were used, OSRG solved more test problems than the other schemes. But, from problem 75 to the last training problem, the schemes that solved more problems were GBR and GBP. These results can be well explained analyzing when the difficulty was increased (in the same level or using the next difficulty rule). The changes in the difficulty are shown in Table 2. The numbers in this table indicate in which training problem each difficulty rule was applied. As it can be observed, the first DIR was applied once around the problem 8 in all the schemes. However, the second DIR was applied first by GBR in training problem 70, second by GBP in training problem 88, while it was never applied by IDG. On one hand, the fact that GBR applies the second DIR before than GBP means that HAMLET learns from more training problems in this level using GBR than using GBP. For this reason, the final number of solved test problems is bigger using GBR. When the second difficulty rule was applied explains also why the improvement in the convergence appears in problem 75 for these two schemes. On the other hand, the low performance of IDG is due to the fact that using this scheme the learning engine does not have the opportunity of learning from training problems with more than one city and one truck, because the second DIR was never applied.

Table 2: Difficulty changes

scheme	DIR 1	DIR 2	DIR 3	DIR 4
IDG	7	-	-	-
GBP	9	88	-	-
GBR	9	70	-	-

In this domain the last two rules for increasing the difficulty were not used in the 100 training problems. However, for example in *GBR*, from the application of the first DIR in problem 9 there were no more increases in the difficulty until problem 70. We could increase the explored difficulty levels by fixing a maximum number of generated problems to be in the same difficulty.

Table 3: Useful problems to learn (over 100).

IDG	GBP	GBR
77	75	83

The aim of using an AL scheme here is to learn as quickly as possible correct knowledge, minimizing the number of training problems. This implies implicitly to generate each training problem such that the probability of modifying previous learned knowledge is maximized. Table 3 shows the number of problems (over 100) that were useful to learn because the previous learned knowledge was modified. The scheme that generates more problems that turned our to be useful for learning was *GBR*.

As the first training problems in each difficulty level are randomly generated in all the schemes, it would be appropriate to perform an analysis of statistical significance. We do not include this kind of analysis in this paper, but it will be the next step in the near term.

The maximum number of test problems solved by IPSS using control rules learned with the GBR scheme is 57, without solving any problem in test-15 and only one problem in test-10. As we explained before IPSS is not very competitive with modern planners given that its goal was the study of ML techniques. However the approaches to generate the next training problem presented in this paper are independent of the planner. They do not depend on the way the planner solves the problem, but in the representation language of the learned knowledge and the control-rule matching algorithm. This language could be very similar for different planners and the learned knowledge can be transferred between them, as it is shown in (Fernández, Aler, & Borrajo 2006). Therefore, using an AL scheme in other planning-ML settings with a modern planner would require almost no work (except for programming their

matching algorithm).

Conclusions and Future Work

In this paper, we described our on-going work on building AL schemes for acquiring control knowledge for planning. Three approaches have been presented to generate the next training problem. The simplest one is to use a set of difficulty rules previously defined to generate training problems with increasing difficulty. The next scheme consists on using not only the difficulty rules but the last problem. The last scheme exploits the learned knowledge. The use of these schemes directs the generation of training problems towards the refinement of previous learned knowledge.

The results of the experiments we perfomed suggest that using AL schemes in the learning process improves its convergence, and the more informed the AL scheme the better the convergence and the performance of the learned knowledge.

This research is in its first stages. In the near term, we plan to expand it with experiments in other domains, including an analysis of statistical significance. We are interested also in improving the AL scheme, for example including other parameters as the maximum number of problems in the same difficulty level. Our longer term plans include implementing a learning mechanism similar to HAMLET in another planner, probably a heuristic one. Heuristic planners have shown to be competitive in the last planning competitions. This will provide a new platform for experimentation with AL schemes.

References

Aler, R.; Borrajo, D.; and Isasi, P. 2002. Using genetic programming to learn and improve control knowledge. *Artificial Intelligence* 141(1-2):29–56.

Borrajo, D., and Veloso, M. 1997. Lazy incremental learning of control knowledge for efficiently obtaining quality plans. *AI Review Journal. Special Issue on Lazy Learning* 11(1-5):371–405. Also in the book "Lazy Learning", David Aha (ed.), Kluwer Academic Publishers, May 1997, ISBN 0-7923-4584-3.

Bryant, C.; Muggleton, S.; Oliver, S.; Kell, D.; Reiser, P.; and King, R. 2001. Combining inductive logic programming, active learning and robotics to discover the function of genes. *Linkoping Electronic Articles in Computer and Information Science* 6(12).

Cesta, A.; Oddi, A.; and Smith, S. F. 2002. A Constrained-Based Method for Project Scheduling with Time Windows. *Journal of Heuristics* 8:109–136.

Cohn, D.; Ghabhramani, Z.; and Jordan, M. 1996. Active learning with statistical models. *Journal of Artificial Intelligence Research* 4:129–145.

Etzioni, O., and Minton, S. 1992. Why EBL produces overly-specific knowledge: A critique of the Prodigy approaches. In *Proceedings of the Ninth International Conference on Machine Learning*, 137–143. Aberdeen, Scotland: Morgan Kaufmann.

Fedorov, V. 1972. *Theory of Optimal Experiments*. London: Academic Press.

Fernández, S.; Aler, R.; and Borrajo, D. 2006. Transfer of learned heuristics among planners. In *Workshop on Learning for Search in the AAAI'06*.

Fikes, R. E.; Hart, P. E.; and Nilsson, N. J. 1972. Learning and executing generalized robot plans. *Artificial Intelligence* 3:251–288.

Fox, M., and Long, D. 1998. The automatic inference of state invariants in TIM. *Journal of Artificial Intelligence Research* 9:317–371.

Fox, M., and Long, D. 2002. *PDDL2.1: An Extension to PDDL for Expressing Temporal Planning Domains*. University of Durham, Durham (UK).

Liere, R.; Tadepalli, P.; and Hall, D. 1997. Active learning with committees for text categorization. In *Proceedings of the Fourteenth Conference on Artificial Intelligence*, 591–596.

Lin, F.-R., and Shaw, M. 1997. Active training of backpropagation neural networks using the learning by experimentation methodology. *Annals of Operations Research* 75:129–145.

McCluskey, T. L., and Porteous, J. M. 1997. Engineering and compiling planning domain models to promote validity and efficiency. *Artificial Intelligence* 95(1):1–65.

Minton, S. 1988. *Learning Effective Search Control Knowledge: An Explanation-Based Approach*. Boston, MA: Kluwer Academic Publishers.

Mitchell, T. M.; Utgoff, P. E.; and Banerji, R. B. 1983. Learning by experimentation: Acquiring and refining problem-solving heuristics. In *Machine Learning, An Artificial Intelligence Approach*. Palo Alto, CA: Tioga Press.

Rodríguez-Moreno, M. D.; Oddi, A.; Borrajo, D.; Cesta, A.; and Meziat, D. 2004. IPSS: A hybrid reasoner for planning and scheduling. In de Mántaras, R. L., and Saitta, L., eds., *Proceedings of the 16th European Conference on Artificial Intelligence (ECAI 2004)*, 1065–1066. Valencia (Spain): IOS Press.

Veloso, M.; Carbonell, J.; Pérez, A.; Borrajo, D.; Fink, E.; and Blythe, J. 1995. Integrating planning and learning: The PRODIGY architecture. *Journal of Experimental and Theoretical AI* 7:81–120.

Veloso, M. 1994. *Planning and Learning by Analogical Reasoning*. Springer Verlag.

Zimmerman, T., and Kambhampati, S. 2003. Learning-assisted automated planning: Looking back, taking stock, going forward. *AI Magazine* 24(2):73–96.