

Improving Control-Knowledge Acquisition for Planning by Active Learning

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. Automatically acquiring control-knowledge for planning, as it is the case for Machine Learning in general, strongly depends on the training examples. In the case of planning, examples are usually extracted from the search tree generated when solving problems. Therefore, examples depend on the problems used for training. Traditionally, these problems are randomly generated by selecting some difficulty parameters. In this paper, we discuss several active learning schemes that improve the relationship between the number of problems generated and planning results in another test set of problems. Results show that these schemes are quite useful for increasing the number of solved problems.¹

1 Introduction

The field of Active learning (AL) has been largely studied in the literature of inductive propositional learners [1,2,3,4]. As defined by [5], 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 [6] 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 [7], 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 [8]).

Training problems are usually created by a random generator that has a set of parameters. These parameters theoretically define the problems difficulty.

¹ 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.

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 the 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 [9] 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 [10].

In order to solve those disadvantages, we propose in this paper several AL methods to generate new training problems for Machine Learning (ML) techniques in the context of problem solving. 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. These AL approaches are independent of the planner and the ML technique used. In our case, we have performed some experiments using IPSS as the planner [11] and HAMLET as the ML technique [10].

Section 2 provides a brief overview of the planner and learning technique we use. Section 3 describes the AL methods. Section 4 relates to previous work. Section 5 presents some experimental results. Section 6 makes some analysis, draws some conclusions and proposes future work.

2 The Planner and the Learning Technique

The AL schemes presented in this paper follow from the experience using HAMLET as a ML technique for the IPSSplanner [11]. IPSS is an integrated tool for planning and scheduling that uses QPRODIGY as the planner component. This is a version of the PRODIGY planner that is able to handle different quality metrics. It is a nonlinear planning system that follows a means-ends analysis. 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): a domain theory, a problem, a set of planning-related parameters and, optionally, some domain-dependent heuristics expressed as control-rules. These control-rules can be manually provided or automatically learned by HAMLET. Given that the AL schemes proposed here are, somehow

independent of the planner, we will not devote more space to explain how it works (see [12] for details on the planning algorithm).

HAMLET is an incremental learning method based on EBL and inductive refinement of control-rules [10]. 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. For each training problem, HAMLET calls IPSS and receives in return the expanded search tree. Then, HAMLET extracts, by a kind of EBL, one control rule for each decision taken by IPSS in the best solution path. Thus, HAMLET output is a set of control-rules (\mathcal{C}) that can potentially guide the planner towards *good* quality solutions in future problems. Since these rules might be overly general or specific, HAMLET can generalize from two rules (merging their common parts) or specialize a rule when it failed (by adding more conditions). See [10] for more details. Figure 1 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).

```
(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)))
```

Fig. 1. Example of a control-rule learned by HAMLET for selecting the `unload-airplane` operator in the logistics domain

3 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 [13]. 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 versions two to four are really AL schemes. 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.

3.1 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.

3.2 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 tree. 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 2. The level of difficulty (Figure 2(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 2(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.

<pre>((object 1) (city 1) (plane 1) (truck 1) (goals 1)) (a)</pre>	<pre>((object 1) (no-goals 1)) ((city 1) (truck 1)) ((object 1) (no-goals 1)) ((plane 1))) (b)</pre>
---	---

Fig. 2. Example of (a) difficulty level and (b) rules for increasing the difficulty

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.

3.3 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. `object1, ... objectn`);
- 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 `objecti`, such that `(in objecti airplanej)` 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 by 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 their corresponding types. For instance, suppose that it has chosen **at_{object}**. **at_{object}** definition as predicate is (**at_{object} ?o - object ?p - place**), where **place** can be either **post-office** or **airport**. Then, it will change (**in object_i airplane_j**) for (**at object_i airport_k**), where **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 [14]. 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) [15], 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 3(a), this technique can automatically generate the new similar problem in Figure 3(b). The advantage of this approach is that if rules were learned from the previous problem, the new 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.

3.4 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 the rules learned from the last problem. Therefore, we compute the frequency of appearance of each predicate, type and instance before performing

```

(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)

```

Fig. 3. Example of (a) problem solved and (b) problem automatically generated by AL. The difference appears in bold face

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 4(a) 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))))
(a)

```

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%

(b)

Fig. 4. Example of (a) learned rules by HAMLET and (b) the correspondent frequency table in true-in-state

Then, Figure4(b) shows the computed frequencies. There are some predicates (at), types (object and airport), or instances (ob0 and a0) that have more probability of being changed than others. This is based on the observation that they appear 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 decision), it would not help on generalizing or specializing control rules.

4 Related Work

One of the first approaches for problem solving was the LEX architecture [4]. 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 AL approaches have also focused on classification tasks using different inductive propositional and relational techniques [5,1,2,3]. 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 training examples are extracted from their search trees. Therefore, for us it is not quite obvious how to use those techniques to generate valid instances (problem solving decisions) from previous ones. 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).

5 Experiments and Results

This section presents some results comparing the four methods of generating problems described previously. We have used the original *Logistics* and the *Zenotravel* domains. In both domains, using the *One-Step Random Generation* approach, we generated a set of 100 random training problems. In this process, the problem difficulty is defined as parameters of the random generator. In the *Logistics*, we generated problems with a random number of goals (between 1 and 5), objects (between 1 and 2), cities (between 1 and 3) and planes (between 1 and 2). In the *Zenotravel*, we generated problems with a random number of goals (between 1 and 5), persons (between 1 and 3), cities (between 1 and 3) and planes (between 1 and 4).

In the AL approaches, the “a priori” generation of random problems is not needed. We only specify the maximum number of valid problems to generate (set as 100), the initial difficulty level and the rules for increasing the difficulty. The number of problems without learning anything before increasing the difficulty was fixed to 3. The definition of the parameters used in these experiments is shown in Figures 2 and 5.

<pre>((person 1 1) (city 2 2) (aircraft 1 1))</pre> <p style="text-align: center;">(a)</p>	<pre>((person 1)) ((aircraft 1)) ((no-goals 1)) ((aircraft 1)) ((city1) (person 1)) ((no-goals 1)) ((aircraft 1) (city 1)))</pre> <p style="text-align: center;">(b)</p>
--	--

Fig. 5. (a) Difficulty level and (b) rules for increasing the difficulty in the *Zenotravel* domain

In the learning phase, we let HAMLET generate 100 valid problems, and learn control rules from them using the following 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 (only for the *Logistics*), 25, 50, 75 and 100 training problems were used.

Then, we randomly generated a set of 120 test problems, of varying difficulty from 1 to 15 goals, and from 1 to 5 objects in the *Logistics* domain and from 1 to 10 goals, 2 to 10 persons and cities and 2 to 15 planes in the *Zenotravel* domain.

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 curves for the *Logistics* (a) and *Zenotravel* domains (b). 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.

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. The improvement in the convergence of these schemes begins in training problem 75 (*Logistics*) and 50 and 75 for GBR and GBP respectively (*Zenotravel*).

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 Figure 7.

In the *Logistics* for instance, the first DIR was applied once around the problem 8 in all the AL 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 also explains 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

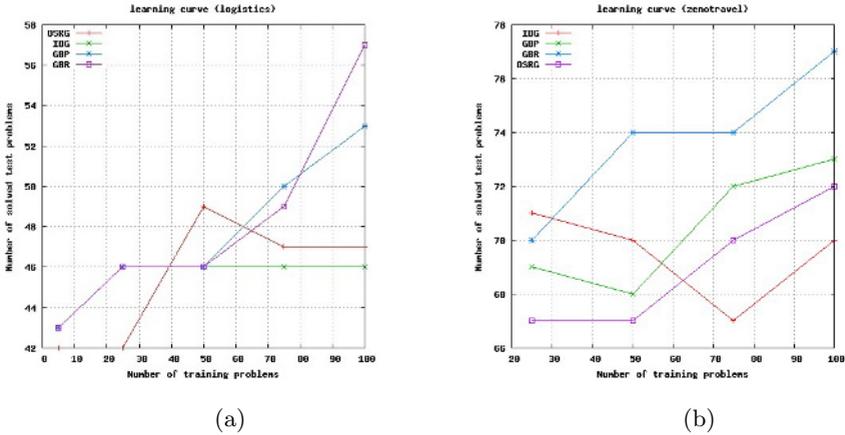


Fig. 6. Learning curves in the *Logistics* (a) and *Zenotravel* (b) domains

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

(a)

scheme	DIR 1	DIR 2	DIR 3	DIR 4	DIR 5	DIR 6	DIR 7
IDG	9	15	37	42	51	54	91
GBP	9	22	29	-	-	-	-
GBR	9	22	30	-	-	-	-

(b)

Fig. 7. Difficulty changes in the *Logistics* domain (a) and *Zenotravel* domain (b)

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. A similar analysis can be done in the case of *Zenotravel*, though, in this case, the IDG scheme changes more often of difficulty level.

6 Discussion and Future Work

One of the main constrains of ML within problem solving is that it needs that the training set contains solvable problems. If the problems are not solvable, nothing will be learned, since learning occurs when in at least one node of the search tree, there is a success child (this is true for learning from success, which is the learning scheme we are using). Without the use of an AL scheme, one must program a domain-dependent random generator for each domain, so that it generates solvable problems. This is not quite obvious in the case of planning problem-solving (there is no guarantee that from any initial state, we can arrive to a state in which goals are true). Formally proving, independently of the domain, that a problem generator generates only solvable problems is a very hard task. One of the main advantages of using AL schemes in the learning process is the fact that they avoid the need of programming a random problem generator for each domain. Thus, we transform a procedural programming task into a task of defining the right declarative knowledge that expresses: the domain types whose

instances can “easily” change of state, together with the different possible states of the objects of this type; the initial difficulty level; and the rules to increase the difficulty. Thus, each training problem is generated using the previous one, and if the domain has some features, explained below, each generated problem from a previous solvable problem is solvable also. Furthermore, the AL schemes we presented here only need just one solvable problem for each difficulty level; the first one of each level. Although the learning process has the bias imposed by this first problem, it is compensated with a random exploration of problems in each level.

The main disadvantage of the approaches presented in this paper is that they are limited to some types of domains. These domains should have the following two features: (1) They must include types whose instances can change “easily” of state (though not all types are required to have that feature). This means that the application of operators which have as parameters objects of these types should produce one-literal changes in the state. The reason is that it is not possible to control complex changes (more than one literal involved) in the states just using separately the information of the different possible states of each type of object. This limitation could be solved using a forward planner to generate the next problem by applying just one applicable operator in the initial state of the problem at hand. But, in this case, in order to obtain a solvable problem, the domain has to be symmetrical (next feature). (2) It is preferable to use symmetrical domains (for each instantiated operator o_i , there must be a sequence of instantiated operators, such that, if applied, they return to the planning state before applying o_i). Otherwise, the AL mechanism could generate a big number of unsolvable problems, decreasing thus the number of useful problems to learn. Useful problems to learn are defined by features of their planning process. For example, it can be considered useful problems whose planning process supposed at least a backtrack, because the first path followed by the planner did not end in the best solution and this produces opportunities to learn. Therefore, not all solvable problems are useful for learning, but a problem only can be useful for learning if it is solvable.

Examples of domains with both of previous features are the *Logistics*, the *Zenotrail* and the *Satellite* domains used in the International Planning Competitions. The *Blocksworld* domain, does not include instances that change “easily” of state. It has only the type *object*. When an object o_1 is changed from a state where it is *on* another object o_2 , to a state where it is *holding*, the change implies changes in the state of o_2 too (it should be *clear*). Changes like this are considered complex. The same happens with the objects in the *Depots* domain. Regarding the second feature, the *Rockets* domain is an example of non-symmetrical domain, because the rockets only have enough fuel to just one change of location.

Currently, we are already performing experiments in other domains with the goal of understanding in what type of domains, it is experimentally useful (apart from the discussion we presented previously on symmetrical domains and “easy” changes).

References

1. Cohn, D., Ghabhrmani, Z., Jordan, M.: Active learning with statistical models. *Journal of Artificial Intelligence Research* **4** (1996) 129–145
2. Liere, R., Tadepalli, P., Hall, D.: Active learning with committees for text categorization. In: *Proceedings of the Fourteenth Conference on Artificial Intelligence*, Providence, RI (USA) (1997) 591–596
3. Lin, F.R., Shaw, M.: Active training of backpropagation neural networks using the learning by experimentation methodology. *Annals of Operations Research* **75** (1997) 129–145
4. Mitchell, T.M., Utgoff, P.E., Banerji, R.B.: Learning by experimentation: Acquiring and refining problem-solving heuristics. In: *Machine Learning, An Artificial Intelligence Approach*. Tioga Press, Palo Alto, CA (1983)
5. Bryant, C., Muggleton, S., Oliver, S., Kell, D., Reiser, P., King, R.: Combining inductive logic programming, active learning and robotics to discover the function of genes. *Linkoping Electronic Articles in Computer and Information Science* **6**(12) (2001)
6. Fedorov, V.: *Theory of Optimal Experiments*. Academic Press, London (1972)
7. Zimmerman, T., Kambhampati, S.: Learning-assisted automated planning: Looking back, taking stock, going forward. *AI Magazine* **24**(2) (2003) 73–96
8. Minton, S.: *Learning Effective Search Control Knowledge: An Explanation-Based Approach*. Kluwer Academic Publishers, Boston, MA (1988)
9. Fikes, R.E., Hart, P.E., Nilsson, N.J.: Learning and executing generalized robot plans. *Artificial Intelligence* **3** (1972) 251–288
10. Borrajo, D., Veloso, M.: Lazy incremental learning of control knowledge for efficiently obtaining quality plans. *AI Review Journal. Special Issue on Lazy Learning* **11**(1-5) (1997) 371–405 Also in the book "Lazy Learning", David Aha (ed.), Kluwer Academic Publishers, May 1997, ISBN 0-7923-4584-3.
11. Rodríguez-Moreno, M.D., Borrajo, D., Cesta, A., Oddi, A.: Integrating planning and scheduling in workflow domains. *Expert System with Applications* **33**(2) (2007)
12. Veloso, M., Carbonell, J., Pérez, A., Borrajo, D., Fink, E., Blythe, J.: Integrating planning and learning: The PRODIGY architecture. *Journal of Experimental and Theoretical AI* **7** (1995) 81–120
13. McCluskey, T.L., Porteous, J.M.: Engineering and compiling planning domain models to promote validity and efficiency. *Artificial Intelligence* **95**(1) (1997) 1–65
14. Fox, M., Long, D.: The automatic inference of state invariants in TIM. *Journal of Artificial Intelligence Research* **9** (1998) 317–371
15. Fox, M., Long, D.: PDDL2.1: An Extension to PDDL for Expressing Temporal Planning Domains. University of Durham, Durham (UK). (2002)