Learning Strategy Knowledge Incrementally

Manuela Veloso
Department of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213-3891
veloso@cs.cmu.edu

Daniel Borrajo
Facultad de Informática
Universidad Politécnica de Madrid
28660 Boadilla del Monte, Madrid, Spain
dborrajo@fi.upm.es

Abstract

Modern industrial processes require advanced computer tools that should adapt to the user requirements and to the tasks being solved. Strategy learning consists of automating the acquisition of patterns of actions used while solving particular tasks. Current intelligent strategy learning systems acquire operational knowledge to improve the efficiency of a particular problem solver. However, these strategy learning tools should also provide a way of achieving low-cost solutions according to user-specific criteria. In this paper, we present a learning system, HAMLET, *which is integrated in a planning architecture,* PRODIGY, *and acquires control knowledge to guide* PRODIGY *to efficiently produce cost-effective plans.* HAMLET *learns from planning episodes, by explaining why the correct decisions were made, and later refines the learned strategy knowledge to make it incrementally correct with experience.*

1 Introduction

As technological advances are applied to real-world problems the requirements for computer-based tools also change. In particular, in automated planning and problem solving, it is not only necessary to have a computational system that generates plans for achieving a set of goals, but also that those plans be achieved efficiently and that they be optimal (or satisficing) with respect to a given quality measure.

Learning strategy knowledge is, therefore, a necessary step towards the application of today's computer's technology to real world tasks. Most of the current strategy learning systems eagerly try to explain correctly the problem solving choices from a single episode or from a static analysis of the domain definition. Therefore they rely heavily on a complete axiomatic domain theory to provide the support for the generalization of the explanation of one episode to other future situations. This requirement and this explanation effort is not generally acceptable in real-world applications that are incompletely specified and subject to dynamic changes.

Instead in this paper, we present HAMLET,[1] an incremental and inductive strategy learning system that acquires control knowledge for guiding the base level nonlinear planner PRODIGY4.0 [3] to efficiently achieve optimal solutions for complex problems [1, 2].

Although this paper describes the HAMLET strategy learning tool in the context of PRODIGY, the main characteristics and contributions of this integration are general to any problem solver, in particular:

- Definition of problem solving or planning as a decision making process where choice points and available alternatives are clearly identified.
- Use of the problem solving search episodes through the space of alternatives to identify learning opportunities from the successful and failed choices made.
- Learning of meta-level strategy rules from the successful choices by explaining the situations under which the same successes are expected to occur. This (generally computationally expensive) explanation process is limited to a predefined template.
- Incremental generalization of the learned knowledge through experience by merging learned rules in similar successes and refining them in situations where they apply incorrectly.

In this paper, we introduce the underlying problem solver identifying its choice points and learning opportunities. We provide an illustrative example on the need to learn plan quality. We describes HAMLET's architecture presenting the generation of the meta-level control rules and their incremental refinement. We shows empirical results in a logistics transportation domain where HAMLET learns rules

[1] "HAMLET" stands for *H*euristics *A*cquisition *M*ethod by *L*earning from s*E*arch *T*rees.

1063-6730/94 $4.00 © 1994 IEEE

Authorized licensed use limited to: Univ Carlos III. Downloaded on February 11, 2010 at 05:19 from IEEE Xplore. Restrictions apply.

that make PRODIGY solve efficiently and with quality complex planning problems with up to 50 goals and hundreds of literals in the initial state. Finally we discuss related work and draw conclusions.

2 Planning and Control of Quality

HAMLET is integrated in the planning and learning architecture PRODIGY. The current nonlinear problem solver in PRODIGY, PRODIGY4.0, follows a means-ends analysis backward chaining search procedure reasoning about multiple goals and multiple alternative operators relevant to the goals.[2] The inputs to the basic problem solver algorithm are: the domain theory, (or for short, domain), which consists of the set of operators specifying the task knowledge, and the object hierarchy; a problem specified in terms of an initial configuration of the world, and a set of goals to be achieved; and control knowledge.

PRODIGY's planning algorithm interleaves backward-chaining planning with the simulation of plan execution, by applying operators found relevant to the goal to an internal world state. Table 1 shows an abstract view of PRODIGY's planning algorithm.

1. Terminate if the goal statement is satisfied in the current state.
2. Compute the set of *pending goals* $\mathcal{G}$, and the set of *applicable operators* $\mathcal{A}$. A goal is pending if it is a precondition, not satisfied in the current state, of an operator selected to be in the plan to achieve a particular goal. An operator is applicable when all its preconditions are satisfied in the state.
3. Choose a goal G from $\mathcal{G}$ or select an operator A from $\mathcal{A}$.
4. If G has been chosen, then
   - *Expand goal* G, i.e., get the set $\mathcal{O}$ of *relevant instantiated operators* that could achieve the goal G,
   - Choose an operator O from $\mathcal{O}$,
   - Go to step 1.
5. If an operator A has been selected as directly applicable, then
   - *Apply* A,
   - Go to step 1.

Table 1: A skeleton of PRODIGY4.0*'s planning algorithm and choice points.*

The planning reasoning cycle, as shown in Table 1, involves several decision points, namely: the *goal* to select from the set of pending goals and subgoals (step 5); the *operator* to choose to achieve a particular goal (step 6); the *bindings* to choose in order to instantiate the chosen operator (step 6); *apply* an operator whose preconditions are satisfied or continue *subgoaling* on a still unachieved goal (step 5). Default decisions at all these choices can be directed by explicit control knowledge.

Although PRODIGY can use a variety of powerful domain-independent heuristics [17], it is very difficult and costly to determine in general which of these heuristics are going to succeed or fail. Therefore, learning can be used for automatically acquiring control knowledge to *override the default behavior* of a particular domain-independent search heuristic to drive the planner more efficiently to a solution. Note that this need to learn when particular domain-independent search strategies do not produce desirable results is common to any planner [19].

We show now a simple example of how control knowledge can be used to plan for solutions of better quality. The domain we use is a logistics-transportation domain [20]. In this domain, packages must be delivered to different locations in several different cities. Packages are carried within the same city in trucks and across cities in airplanes. At each city, there are several locations, such as post offices and airports. The domain consists of a set of operators to load and unload packages into and from the carriers at different locations, and to move the carriers between locations.

Consider a planning situation where there are three cities, *city1*, *city2* and *city3*, each with one post office and one airport. Initially, there is one package, *package1*, at *airport1*, and one airplane, *plane1*, at *airport2*. The goal of the problem is to bring both *package1* and *plane1* to *airport3*.

The optimal solution to the problem is the following sequence of steps:

fly-airplane(plane1, airport2, airport1)
load-airplane(package1, plane1, airport1)
fly-airplane(plane1, airport1, airport3)
unload-airplane(package1, plane1, airport3)

Notice that, although in this example the optimal plan corresponds to this unique linearization, in general the learning procedure can reason about a partially-ordered dependency network of the plan steps.

In the representation of the logistics domain, there is no knowledge in the domain description (operators), about not moving carriers if they need to be loaded; unloading a truck, instead of unloading an airplane if a package is, in the current state, in the same city as in the goal; or loading one after another two packages that are in the same place, if they need to go in the same carrier. Some of this information is not observable by doing *goal regression*, and is related to the quality of the solutions. Our approach consists on learning the needed knowledge to guide the problem solver to the optimal solutions.

Figure 1(a) shows a control rule learned by HAMLET

[2] PRODIGY4.0 is a successor of the previous linear planner, PRODIGY2.0, and PRODIGY's first nonlinear and complete planner, NOLIMIT.

Authorized licensed use limited to: Univ Carlos III. Downloaded on February 11, 2010 at 05:19 from IEEE Xplore. Restrictions apply.

from the search tree produced by PRODIGY when solving the problem.[3] A control rule consists of two parts: the *if*-part which specifies the conditions under which the rule should apply; and the *then*-part identifying the particular decision to be made.

```
(control-rule select-bind-fly-airplane-1
 (if (current-operator fly-airplane)
     (current-goal (at-airplane <plane1> <airport3>))
     (true-in-state (at-airplane <plane1> <airport2>))
     (true-in-state (at-object <package4> <airport1>))
     (other-goals ((at-object <package4> <airport3>))))
 (then select bindings ((<plane> . <plane1>)
                        (<loc-from> . <airport1>)
                        (<loc-to> . <airport3>))))
                           (a)
(Operator FLY-AIRPLANE
(preconds ((<plane> AIRPLANE) (<loc-from> AIRPORT)
           (<loc-to> AIRPORT))
 (at-airplane <plane> <loc-from>))
(effects ((add (at-airplane <plane> <loc-to>))
          (del (at-airplane <plane> <loc-from>)))))
                           (b)
```

Figure 1: (a) Control rule created by HAMLET*; (b) The operator FLY-AIRPLANE.*

The rule *select-bind-fly-airplane-1* in Figure 1(a) identifies a set of bindings for the instantiation of the operator *fly-airplane* shown in Figure 1(b). Note that the backward-chaining procedure selects the operator *fly-airplane* when planning to achieve the goal *(at-airplane plane1 airport3)*. The location to which the airplane flies, i.e., *<loc-to>*, is bound by the goal, but not the location where the airplane should come from. PRODIGY needs to choose which bindings to use for the *<loc-from>* variable in the operator. If the goal is to move an airplane to an airport, and there is another goal to be achieved, namely to move a package to a different airport, then the rule *select-bind-fly-airplane-1* establishes that the airplane should reach the final airport departing from the airport where the object is currently located. Therefore the airplane should pick the object first to avoid an extra trip from its destination back to pick the object and returning to its destination.

Although this rule might seem simple, it really embodies some complexity. An explanation-based system that learns from a satisficing problem solving episode by doing goal regression, would learn a rule as shown in Figure 2.

This rule is *unoptimal*. If PRODIGY uses this rule to solve the problem introduced above, PRODIGY will produce the unoptimal solution. The airplane will fly from *airport2* to *airport3* directly, then fly to pick up the object, and then

```
(control-rule select-bind-fly-airplane-2
 (if (current-operator fly-airplane)
     (current-goal (at-airplane <plane1> <airport3>))
     (true-in-state (at-airplane <plane1> <airport2>)))
 (then select bindings ((<plane> . <plane1>)
                        (<loc-from> . <airport2>)
                        (<loc-to> . <airport3>))))
```

Figure 2: Control rule that selects flying directly an airplane from its current location to its destination. This rule produces unoptimal solutions in many situations, for the example problem introduced above.

return to its destination. This would also happen to almost all problems in this domain that require to use an airplane, at least twice.

Using the control rules learned by HAMLET, PRODIGY is able to find the optimal solutions efficiently.

3 HAMLET's architecture

The inputs to HAMLET are a task domain, a set of training problems, and a quality measure. The output is a set of control rules. HAMLET has three main modules: the Bounded-Explanation module, the Inductive module, and the Refinement module.

The Bounded-Explanation module generates control rules from a PRODIGY search tree. These rules might be over-specific or over-general. The Induction module solves the problem of over-specificity by generalizing rules when analyzing positive examples. The Refinement module replaces over-general rules with more specific ones when it finds situations in which the learned rules lead to wrong decisions. HAMLET gradually learns and refines control rules converging to a concise set of correct control rules, i.e. rules that are individually neither over-general, nor over-specific.[4]

Figure 3 shows HAMLET's modules and their connection to PRODIGY. Here ST and ST′ are search trees generated by the PRODIGY planning algorithm, L is the set of control rules, L′ is the set of new control rules learned by the Bounded Explanation module, and L″ is the set of rules induced from L and L′ by the Inductive module.

Table 2 outlines HAMLET's algorithm. If in a new training search tree, HAMLET finds new positive examples of the opportunities to learn control rules then it generates new rules and, when possible, induces more general rules. If negative examples are found of application of control rules, HAMLET refines them, i.e., generates more specific rules.

[3] For simplicity we use the same names for the instances in the problem and the variables in the rule. Note, however, that the names in the rules represent variables rather than specific packages.

[4] We have empirical evidence for this convergence phenomena, but one of our current research efforts is the formal study of the convergence of the learning mechanism.

Authorized licensed use limited to: Univ Carlos III. Downloaded on February 11, 2010 at 05:19 from IEEE Xplore. Restrictions apply.

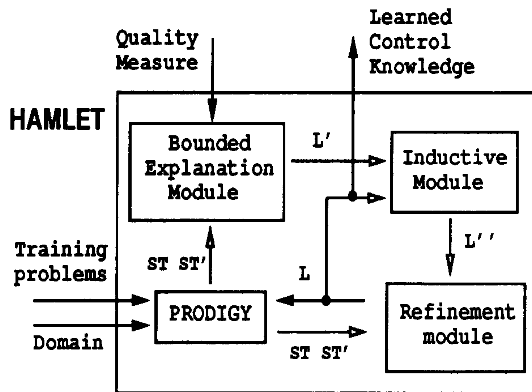

Figure 3: HAMLET's high level architecture

Let L refer to the set of learned control rules.
Let ST refer to a search tree.
Let P be a problem to be solved.
Let Q be a quality measure.
Initially L is empty.
For all P in the set of training problems
ST = Result of solving P without any rules.
ST' = Result of solving P with current set of rules L.
If positive-examples-p(ST, ST',Q)
Then L' = Bounded-Explanation(ST, ST')
L''= Induce(L,L')
If negative-examples-p(ST, ST',Q)
Then L=Refine(ST, ST',L'')
Return L

Table 2: A high-level description of HAMLET's learning algorithm

3.1 Bounded Explanation

The Bounded-Explanation module learns control rules by choosing important decisions made during the search for a solution and extracting the information that justifies these decisions from the search space. The explanation procedure consists of four phases: Labeling the decision tree; Credit assignment; Generation of control rules; and Parameterization.

Labeling the decision tree

HAMLET traverses the search tree bottom-up, starting from the leaf nodes. It assigns three kinds of labels to each node of the tree: *success*, if the node corresponds to a correct solution plan; *failure*, if the node is a dead end in the search space; and *unknown*, if the planner did not expand the node, and thus we do not know whether this node can lead to a solution

Credit Assignment

The credit assignment is the process of selecting important branching decisions, for which learning will occur. It is done concurrently with labeling. HAMLET views a branching decision as a learning opportunity if this decision (1) leads to a solution and (2) differs from the default decision made by domain-independent heuristics.

Generation of control rules

At each decision choice to be learned, HAMLET has access to information on the current state of the search and on the meta-level planning information, such as the goals that have not been achieved, the goal the planner is working on, and the possible applicable operators. This information is used by the generation module to create the applicability conditions, i.e. the *if*-parts of the control rules. The relevant features of the current state are selected using *goal regression.*

HAMLET learns four kinds of *select* control rules, corresponding to the PRODIGY's decisions of section 2, which are generalized target concepts. They are: decide whether to apply an *operator* or subgoal on an unachieved *goal*, select an unachieved *goal*, select an *operator* to achieve some specific *goal*, select bindings for an *operator* when trying to achieve a *goal*. HAMLET generates a set of rules for each target concept, where the *if*-part of each rule is described as a conjunctive set of predicates. As HAMLET can learn several rules for the same target concept, the set of all rules can be viewed as the disjunction of conjunctive rules.

Parameterization

After a rule is generated, HAMLET replaces specific constants inherited from the considered planning situation with variables of corresponding types. Distinct constants are replaced with differently named variables, and when the rule is applied, different variables must always be matched with distinct constants. The latter heuristic may be relaxed in the process of inductive generalization of the learned rules.

3.2 Inductive Generalization

The rules generated by the bounded explanation method may be over-specific as also noticed by [6]. To address this problem, we use the Inductive Learning module, which generalizes the learned rules by analyzing new examples of situations where the rules are applicable. We have devised methods for generalizing over the following aspects of the learned knowledge: state; subgoaling structure; interacting goals; and type hierarchy [2].

Example

Consider that HAMLET has previously learned the rule shown in Figure 4. The rule says that the planner should find a way of achieving *(inside-truck package1 truck1)* before moving the carriers, *plane1* and *truck1* (even though flying an empty airplane will achieve one of the two planning goals). This is a useful rule, since if PRODIGY decides to

Authorized licensed use limited to: Univ Carlos III. Downloaded on February 11, 2010 at 05:19 from IEEE Xplore. Restrictions apply.

move a carrier before loading it, the carrier will have to be moved back to pick its load, thus resulting in a non-optimal plan. This rule is over-specific, since, among other things, the prior-goal does not have to include the package in an airport, the other-goals may be ignored, and *truck1* does not have to be in the same place as *package1*.

```
(control-rule select-inside-truck-1
  (if (target-goal (inside-truck <package1> <truck1>))
      (prior-goal (at-object <package1> <post-office2>))
      (true-in-state (at-truck <truck1> <post-office1>))
      (true-in-state (at-object <package1> <post-office1>))
      (true-in-state (same-city <airport1> <post-office1>))
      (other-goals ((at-truck <truck1> <airport1>)
                    (at-airplane <plane1> <airport2>))))
  (then select goals (inside-truck <package1> <truck1>)))
```

Figure 4: Rule learned by HAMLET *after applying the Bounded-Explanation.*

Suppose that PRODIGY solves a new problem in which *package1* and *plane1* have to go to *airport2*, and *truck1* is initially at *airport1*. HAMLET learns a new rule shown in Figure 5.

```
(control-rule select-inside-truck-2
  (if (target-goal (inside-truck <package1> <truck1>))
      (prior-goal (at-object <package1> <airport2>))
      (true-in-state (at-truck <truck1> <airport1>))
      (true-in-state (at-object <package1> <post-office1>))
      (true-in-state (same-city <airport1> <post-office1>))
      (other-goals ((at-airplane <plane1> <airport2>))))
  (then select goals (inside-truck <package1> <truck1>)))
```

Figure 5: Rule learned by HAMLET *from a second example.*

From these two rules, namely in Figure 4 and in Figure 5, the Induction module generates the rule presented in Figure 6, which is the result of applying three inductive operators: *Relaxing the subgoaling dependency*, which removes the the prior-goal from the *if*-part; *Refinement of goal dependencies*, which requires the other-goals to be the union of the other-goals of each rule; and *Finding common superclass*, which finds a common general type in the hierarchy of PRODIGY types, *location*, that is a common ancestor of the types *airport* and *post-office*.

```
(control-rule select-inside-truck-3
  (if (target-goal (inside-truck <package1> <truck1>))
      (true-in-state (at-truck <truck1> <location1>))
      (true-in-state (at-object <package1> <post-office1>))
      (true-in-state (same-city <location1> <post-office1>))
      (other-goals ((at-truck <truck1> <location1>)
                    (at-airplane <plane1> <airport2>))))
  (then select goals (inside-truck <package1> <truck1>)))
```

Figure 6: Rule induced by HAMLET *from the rules shown in Figures 4 and 5.*

3.3 Refinement

HAMLET may also generate over-general rules, either by inducing or by doing *goal regression* when generating the rules. Therefore, the over-general rules need to be refined. There are two main issues to be addressed: how to detect a negative example, and how to refine the learned knowledge according to it. A negative example for HAMLET is a situation in which a control rule was applied, and the resulting decision led to either a failure (instead of the expected success), or a worse solution than the best one for that decision.

When a negative example is found, HAMLET tries to recover from the over-generalization by refining the wrong rule. The goal is to find, for each over-general rule, a larger set of predicates that covers the positive examples, but not the negative examples.

Since there are two kinds of rules (bounded, and induced), there are two kinds of recovering methods. The bounded rules are refined by adding literals from the whole set of features present when the rule was learned. The induced rules come from two rules, so for each one of the generating rules, HAMLET tests whether the generating rule also covers the negative example. If so, HAMLET steps back and refines recursively that rule. If not, the induced rule is refined by adding predicates from the set difference of the *if*-part of the generating rule and the *if*-part of the induced rule.

4 Empirical Results

We have carried out experiments on several domains, and the results we report in this section were done on the logistics transportation domain already mentioned. We trained HAMLET with 400 randomly generated problems of one and two goals, and up to three cities and five packages. HAMLET generated 26 control rules. Then, we randomly generated 500 testing problems of increasing complexity. Table 3 shows the results of those tests. We varied the number of goals in the problems from 1 up to 50, and the maximum number of packages from 5 up to 50. Notice that the problems with 20 and 50 goals are really very complex especially in terms of finding good or optimal solutions, involving solutions of over 100 steps.

In the experiments we compare two configurations, namely PRODIGY not using and using the rules learned by HAMLET. In both situations, PRODIGY was given a CPU running time limit that varied with the number of goals

Authorized licensed use limited to: Univ Carlos III. Downloaded on February 11, 2010 at 05:19 from IEEE Xplore. Restrictions apply.

Test sets		Unsolved problems		Solved by both configurations (271 problems)					
				Better solutions		Solution length		Nodes explored	
Number of Goals	Number of Problems	without rules	with rules	without rules	with rules	without rules	with rules	without rules	with rules
1	100	5	0	0	11	327	307	2097	1569
2	100	15	6	0	25	528	479	3401	2308
5	100	44	23	0	29	789	708	4822	3472
10	100	68	35	1	22	731	640	3309	2846
20	75	62	45	0	8	348	322	1516	1360
50	25	24	18	0	0	34	34	143	141
Totals	500	218	125	1	95	2757	2490	15288	11696

Table 3: Empirical results on increasingly complex problems in the logistics domain.

of the problem according to the formula Time_bound = $150 * (1 + \text{mod}(\text{number_of_goals}, 10))$ in seconds. The results show a very significant decrease in the number of unsolved problems when PRODIGY used the learned rules. To compare the solution quality, we can only account for the problems solved by both configurations. Table 3 shows that, for a large number of those problems, the solution generated using the rules is of better quality than the one generated without the rules. This shows that the learned rules drive the planner to find solutions of quality in addition to doing it efficiently, which supports our research goals underlying HAMLET's learning algorithm. The overall running times also decreased using the rules, but not significantly. We did not find empirically with our learned rules that the time spent solving the problem degraded so much to consider it a utility problem [11]. However, we are currently developing efficient methods for organizing and matching the learned control rules. We consider this organization essential and part of the overall learning process [4].

5 Related work

Most speedup learning systems have been applied to problem solvers with the linearity assumption, such as the ones applied to Prolog or logic programming problem solvers [15, 21], special-purpose problem solvers [12, 9, 18], or other general-purpose linear problem solvers [5, 10, 11, 14]. These problem solvers are known to be incomplete and and incapable of finding optimal solutions. If we remove the linearity assumption, we are dealing with nonlinear problem solvers. This kind of problem solvers are needed to address real world complex problems. In general, there have not been as many learning systems applied to nonlinear problem solving. Some approaches include [7, 8, 13, 16, 20].

While improving problem solving performance has been largely studied, learning to improve solution quality has only been recently pursued by some researchers including [13, 16]. The difference with Pérez's work relies on the inductive refinement HAMLET performs, and in the way positive examples are generated. The second approach differs in the knowledge representation of the learned control knowledge, since it is a case-based learner. HAMLET combines the two kinds of optimization, by learning control rules, that allow not only to do more effective search, but also to achieve better solutions.

In terms of knowledge representation of the control knowledge, there have been two major approaches: control knowledge integrated into the domain operators, either learning new operators, or modifying the existing ones [9, 21]; and control knowledge separated from the domain knowledge [5, 11]. HAMLET explicitly learns procedural control knowledge separated from the domain operators, according to PRODIGY's architecture, which allows the transparent incorporation of meta-level problem solving knowledge.

We have extended the learning methods used with the linear planning algorithm of PRODIGY2.0 [5, 11, 14] in the following directions.

- **Extending the work on linear problem solving** requires to address new problems raised by the decisions on multiple goal choices of nonlinear problem solving together with the previous multiple operator and binding choices. We have identified the following extensions to the previous work: extension of the language for describing the learned control rules; definition of new learning opportunities; and consideration of the quality of the solutions.
- **Reducing the explanation effort** is achieved by generating partial ("bounded") explanations of branching decisions made during the search for a solution. Our inductive learning technique does not require insuring the correctness of every deduced control rule.

Authorized licensed use limited to: Univ Carlos III. Downloaded on February 11, 2010 at 05:19 from IEEE Xplore. Restrictions apply.

The learned rules may be incorrect, over-general or over-specific, but they converge to the correct ones as the system analyze more examples. The use of inductive learning eliminated the need for the extra domain knowledge, such as domain axioms used in EBL [5, 11, 14].

- **Incremental refinement of control rules** allows the system to improve the quality of the learned rules by analyzing new examples. The rules generated by the bounded-explanation module may be over-general or too specific. However, the system is capable of testing the control rules against new examples and modifying them as necessary until it produces correct rules of appropriate generality.

6 Conclusions

We have discussed a learning tool, HAMLET, that captures action patterns of success for a problem solver. The learned knowledge enables the problem solver to find solutions to problems efficiently and of good quality according to some user defined criteria. We believe that it is necessary to build this kind of strategy learning tools within existing problem solvers, in order to apply them to real-world applications with dynamic changing environments.

References

[1] D. Borrajo and M. Veloso. Bounded explanation and inductive refinement for acquiring control knowledge. In *Proceedings of the Third International Workshop on Knowledge Compilation and Speedup Learning*, pages 21–27, Amherst, MA, June 1993.

[2] D. Borrajo and M. Veloso. Incremental learning of control knowledge for nonlinear problem solving. In *Proceedings of the European Conference on Machine Learning, ECML-94*, pages 64–82. Springer Verlag, April 1994.

[3] J. G. Carbonell, J. Blythe, O. Etzioni, Y. Gil, R. Joseph, D. Kahn, C. Knoblock, S. Minton, A. Pérez, S. Reilly, M. Veloso, and X. Wang. PRODIGY4.0: The manual and tutorial. Technical Report CMU-CS-92-150, SCS, Carnegie Mellon University, June 1992.

[4] R. B. Doorenbos and M. M. Veloso. Knowledge organization and the utility problem. In *Proceedings of the Third International Workshop on Knowledge Compilation and Speedup Learning*, pages 28–34, Amherst, MA, June 1993.

[5] O. Etzioni. Acquiring search-control knowledge via static analysis. *Artificial Intelligence*, 62(2):255–301, 1993.

[6] O. Etzioni and S. Minton. Why EBL produces overly-specific knowledge: A critique of the prodigy approaches. In *Proceedings of the Ninth International Conference on Machine Learning*, pages 137–143, 1992.

[7] S. Kambhampati and S. Kedar. Explanation based generalization of partially ordered plans. In *Proceedings of AAAI-91*, pages 679–685, 1991.

[8] J. E. Laird, P. S. Rosenbloom, and A. Newell. Chunking in SOAR: The anatomy of a general learning mechanism. *Machine Learning*, 1:11–46, 1986.

[9] P. Langley. Learning effective search heuristics. In *Proceedings of the Ninth International Joint Conference on Artificial Intelligence*, pages 419–421, 1983.

[10] C. Leckie and I. Zukerman. Learning search control rules for planning: An inductive approach. In *Proceedings of Machine Learning Workshop*, pages 422–426, 1991.

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

[12] T. M. Mitchell, P. E. Utgoff, and R. B. Banerji. Learning by experimentation: Acquiring and refining problem-solving heuristics. In *Machine Learning, An Artificial Intelligence Approach*, pages 163–190. Tioga Press, Palo Alto, CA, 1983.

[13] M. A. Pérez and J. G. Carbonell. Control knowledge to improve plan quality. In *Proceedings of the Second International Conference on AI Planning Systems*, Chicago, IL, 1994.

[14] M. A. Pérez and O. Etzioni. DYNAMIC: A new role for training problems in EBL. In D. Sleeman and P. Edwards, editors, *Proceedings of the Ninth International Conference on Machine Learning*, pages 367–372. Morgan Kaufmann, San Mateo, CA, 1992.

[15] J. R. Quinlan. Learning logical definitions from relations. *Machine Learning*, 5:239–266, 1990.

[16] D. Ruby and D. Kibler. Learning episodes for optimization. In *Proceedings of the Machine Learning Conference 1992*, pages 379–384, San Mateo, CA, 1992. Morgan Kaufmann.

[17] P. Stone, M. Veloso, and J. Blythe. The need for different domain-independent heuristics. In *Proceedings of the Second International Conference on AI Planning Systems*, pages 164–169, June 1994.

[18] P. Tadepalli. Lazy explanation-based learning: A solution to the intractable theory problem. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, pages 694–700, San Mateo, CA, 1989. Morgan Kaufmann.

[19] M. Veloso and J. Blythe. Linkability: Examining causal link commitments in partial-order planning. In *Proceedings of the Second International Conference on AI Planning Systems*, pages 170–175, June 1994.

[20] M. M. Veloso. Flexible strategy learning: Analogical replay of problem solving episodes. In *Proceedings of Twelfth National Conference on Artificial Intelligence*, Seattle, WA, August 1994. AAAI Press.

[21] J. Zelle and R. Mooney. Combining FOIL and EBG to speed-up logic programs. In *Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence*, 1993.

Authorized licensed use limited to: Univ Carlos III. Downloaded on February 11, 2010 at 05:19 from IEEE Xplore. Restrictions apply.