Lazy Incremental Learning of Control Knowledge for Effic ently Obtaining Quality Plans

DANIEL BORRAJO[1] and MANUELA VELOSO[2]
[1] *Departamento de Informática, Universidad Carlos III de Madrid, 28911 Leganés (Madrid), Spain*
E-mail: dborrajo@grial.uc3m.es
[2] *Computer Science Department, Carnegie Mellon University, Pittsburgh, PA 15213-3891, USA*
E-mail: veloso@cs.cmu.edu

Abstract. General-purpose generative planners use domain-independent search heuristics to generate solutions for problems in a variety of domains. However, in some situations these heuristics force the planner to perform ineff ciently or obtain solutions of poor quality. Learning from experience can help to identify the particular situations for which the domain-independent heuristics need to be overridden. Most of the past learning approaches are fully deductive and eagerly acquire correct control knowledge from a necessarily complete domain theory and a few examples to focus their scope. These learning strategies are hard to generalize in the case of nonlinear planning, where it is diff cult to capture correct explanations of the interactions among goals, multiple planning operator choices, and situational data. In this article, we present a *lazy* learning method that combines a deductive and an inductive strategy to eff ciently learn control knowledge incrementally with experience. We present HAMLET, a system we developed that learns control knowledge to improve both search *effic ency* and the *quality* of the solutions generated by a nonlinear planner, namely PRODIGY4.0. We have identif ed three lazy aspects of our approach from which we believe HAMLET greatly benef ts: lazy explanation of successes, incremental ref nement of acquired knowledge, and lazy learning to override only the default behavior of the problem solver. We show empirical results that support the effectiveness of this overall lazy learning approach, in terms of improving the eff ciency of the problem solver and the quality of the solutions produced.

Key words: speedup learning, nonlinear planning, lazy learning, multistrategy learning, learning to improve plan quality

1. Introduction

Planning uses generalized operators, describing the available actions in a task domain, to search for a solution to a problem by selecting, instantiating, and chaining appropriate operators. Control knowledge can be added to the planning procedure to guide the search, thus improving the planning performance. It has been the focus of attention of several researchers, present

authors included, *to learn* control knowledge, i.e., to automate the acquisition of knowledge that guides the problem solving search process.

One approach to learning control knowledge consists of generating explanations for the local decisions made during the search process (DeJong and Mooney 1986; Laird et al. 1986; Mitchell et al. 1986; Minton 1988; Pérez and Etzioni 1992; Katukam and Kambhampati 1994). These explanations become control rules that are used in future situations to prune the search space. These deductive approaches invest a substantial explanation effort to produce provably correct and complete control rules from a single (or few) problem solving examples and a correct underlying domain theory. They also require a complete domain theory to obtain the explanations,[1] although there has been some work on learning with incomplete, or intractable theories, such as (Tadepalli 1989). Alternatively, inductive approaches incrementally acquire correct knowledge by observing a large set of problem solving examples. These approaches strongly depend on the particular examples seen, but can also acquire simple and useful rules (Cohen 1990; Leckie and Zukerman 1991).

This article presents a method that combines a deductive and an inductive approach, integrating three aspects of lazy learning. The results show that the combination of these three lazy components has several advantages over eager deductive approaches, such as: reduced learning effort; no need for a complete domain theory as specifie by domain axioms; and reduced cost of utilizing the learned knowledge. The lazy aspects we have identifie are:

- **Bounded explanation**. The learning method explains the successes of the problem solving episodes f rst by loosely following the dependencies among choices, and by selecting a bounded set of features that will be used in the explanations. No proof of correctness or completeness for the explanations generated is attempted as in eager approaches.
- **Incremental refine ent**. Upon experiencing new problem solving episodes, the learning algorithm lazily refi es these explanations with examples of its successful or failed applications. It incrementally acquires increasingly correct control knowledge. Since the learned rules are approximately correct, there should be no need to use a large number of examples for refin ng them, as do many inductive methods. As discussed in the editorial of this special issue, there is a difference between incremental and lazy learning, in that the f rst one refers to how examples are provided to the learning system, while the second refers to how the system handles those examples. With respect to this difference, HAMLET is

[1] In addition to the set of operators and inference rules that describe the primitive problem solving actions, a complete domain theory includes a set of domain axioms that enables the proof of the universal truth of the learned knowledge in the domain.

both a lazy and an incremental system: examples are given incrementally, and each one is handled lazily.

- **Lazy learning to override only the default behavior**. With respect to identifying which are the learning opportunities, our system can operate in two learning modes: *eager* or *lazy*. In *eager* mode, it generates a positive example from every decision that leads to a solution. In *lazy* mode, it generates a positive example only if the decision leads to one of the globally best solutions and it was not the choice selected by the problem solving default heuristics.

We implemented our learning approach in a system called HAMLET, standing for *H*euristics *A*cquisition *M*ethod by *L*earning from s*E*arch *T*rees (Barrajo and Veloso 1994; Veloso and Barrajo 1994). HAMLET is integrated with PRODIGY4.0, the current nonlinear problem solver of the PRODIGY architecture for planning and learning (Carbonell et al. 1992). HAMLET learns control knowledge incrementally and inductively to improve both the search *effic ency* of the problem solver and to improve the *quality* of the plans generated.

The article is divided into seven sections. Section 2 briefl presents PRODIGY4.0, the substrate nonlinear planner, identifying its choice points and learning opportunities. It also introduces HAMLET's architecture, presenting the generation of the meta-level control rules and their incremental refine ment. Section 3 describes the Bounded Explanation component of HAMLET. Section 4 discusses the Refinemen module with the generalization and specialization algorithms. Section 5 shows empirical results in the blocksworld domain and in an elaborated logistics transportation domain, where HAMLET learns rules that improve PRODIGY's effi iency and the quality of the solutions of complex planning problems with up to 50 goals and hundreds of literals in the initial state. Section 6 relates our work with other strategy learning approaches. We discuss how HAMLET extends the explanation-based strategy learning method for applications to nonlinear planning. Finally, Section 7 summarizes our conclusions.

2. Overview of PRODIGY4.0 and HAMLET

HAMLET is integrated in the planning and learning architecture PRODIGY (Carbonell et al. 1990). 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 (Veloso et al. 1995).[2] The inputs to the problem solver algorithm are:

- Domain theory, $\mathcal{D}$ (or, for short, domain), that includes the set of operators specifying the task knowledge and the object hierarchy;
- Problem, specified in terms of an initial configuration of the world (initial state, $\mathcal{S}$) and a set of goals to be achieved ($\mathcal{G}$); and
- Control knowledge, $\mathcal{C}$, described as a set of control rules, that guides the decision-making process.

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. Figure 1 shows an abstract view of PRODIGY's planning algorithm. The function `backtrack` will return to a prior node, undoing the effects of applied operators. All `select-` functions return the best alternative according to the control knowledge.[3]

The planning/reasoning cycle, as shown in Figure 1, involves several decision points, namely:

- the *goal* to select from the set of pending goals and subgoals (step 3.1.1);
- the *operator* to choose to achieve a particular goal (step 3.1.2);
- the *bindings* to choose to instantiate the chosen operator (step 3.1.3);
- *apply* an operator whose preconditions are satisfied or continue *subgoaling* on an unsolved goal (step 3): and
- the *operator* to be applied (step 3.2.1).

Default decisions at all these choices can be directed by explicit control knowledge. Figure 2 sketches the general decision search tree considered by PRODIGY. The decision cycle first encounters steps 3.1.x (i.e., selection of goal, operator, and bindings), followed by the same set of steps 3.1.x, or by applying an operator in steps 3.2.x.

Although PRODIGY uses powerful domain-independent heuristics (Stone et al. 1994) that guide the decision making process, it is still difficult and costly to characterize when these heuristics are going to succeed or fail. Therefore, learning is used for automatically acquiring control knowledge to *override the default behavior*, so that it guides the planner more efficiently to solutions of good quality.[4]

[2] PRODIGY4.0 is a successor of the previous linear planner, PRODIGY2.0 (Minton et al. 1989), and PRODIGY's first nonlinear planner, NOLIMIT (Veloso 1989). We use the term "nonlinear" for a planner that can fully interleave subplans for different goals.

[3] We will use **boldface** for functions whose definitions appear in the article, and `typescript font` for functions whose definitions do not appear. We will not study PRODIGY4.0 in detail, and, therefore, we did not use any boldface in its definition.

[4] Independently of the base-level planning algorithm, researchers should find learning opportunities to override the planning default behavior when it leads the planner into failure, inefficient performance, or solutions of poor quality (Veloso and Blythe 1994).

Function **Prodigy4.0** ($\mathcal{S}, \mathcal{G}, \mathcal{D}, \mathcal{C}$)

$\mathcal{S}$ is the state of the problem
$\mathcal{G}$ is the set of goals to be achieved, *pending goals*
$\mathcal{D}$ is the domain description: operators and objects hierarchy
$\mathcal{C}$ is the set of control rules (control knowledge)
$\mathcal{P}$ is the plan, initially $\emptyset$
O is a variablized operator (from the set of operators in $\mathcal{D}$)
B is a substitution (bindings) of the variables of an operator
O_B is the operator O instantiated with bindings B
$\mathcal{O}$ is the set of chosen instantiated operators not yet in $\mathcal{P}$, initially $\emptyset$
$\mathcal{A}$ is the set of applicable operators (a subset of $\mathcal{O}$)
$\mathcal{G}_\mathcal{O}$ is the set of preconditions of all operators in $\mathcal{O}$, $\mathcal{G}_\mathcal{O} = \cup_{O_B \in \mathcal{O}}$ preconditions(O_B)

While $\mathcal{G} \not\subseteq \mathcal{S}$ AND search tree not exhausted
1. $\mathcal{G} = \mathcal{G}_\mathcal{O} - \mathcal{S}$
2. $\mathcal{A} \leftarrow \emptyset$
Forall $O_B \in \mathcal{O} \mid$ preconditions(O_B)$\subseteq \mathcal{S}$ do $\mathcal{A} \leftarrow \mathcal{A} \cup \{O_B\}$
3. If `select-subgoal-or-apply`($\mathcal{G}, \mathcal{A}, \mathcal{S}, \mathcal{C}$)=*subgoal*
Then **3.1.1.** $G \leftarrow$`select-goal`($\mathcal{G}, \mathcal{S}, \mathcal{C}$)
3.1.2. $O \leftarrow$`select-relevant-operator`($G, \mathcal{D}, \mathcal{S}, \mathcal{C}$)
3.1.3. $B \leftarrow$`select-bindings`($O, \mathcal{S}, \mathcal{C}$)
3.1.4. $\mathcal{O} \leftarrow \mathcal{O} \cup O_B$
Else **3.2.1.** $O_B \leftarrow$`select-applicable-operator`($\mathcal{A}, \mathcal{S}, \mathcal{C}$)
3.2.2. $\mathcal{S} \leftarrow$`apply`($O_B, \mathcal{S}$)
3.2.3. $\mathcal{O} \leftarrow \mathcal{O} - O_B$
3.2.4. $\mathcal{P} \leftarrow$ `enqueue-at-end`($\mathcal{P}, \mathcal{O}_\mathcal{B}$)
4. If there is a reason to suspend the current search path
Then `backtrack`.
$ST \leftarrow$ the planning search tree.
Return plan $\mathcal{P}$ and ST

Figure 1. PRODIGY4.0's planning algorithm.

HAMLET is integrated with the PRODIGY planner. The inputs to HAMLET are a task domain ($\mathcal{D}$), a set of training problems ($\mathcal{P}$), a quality measure (Q), a learning mode (L), and an optimality parameter (O). Q, L, and O will be explained shortly. The output is a set of control rules ($\mathcal{C}$). HAMLET has two main modules: the Bounded Explanation module, and the Refinemen module. Figure 3 shows HAMLET's modules and their connection to PRODIGY.

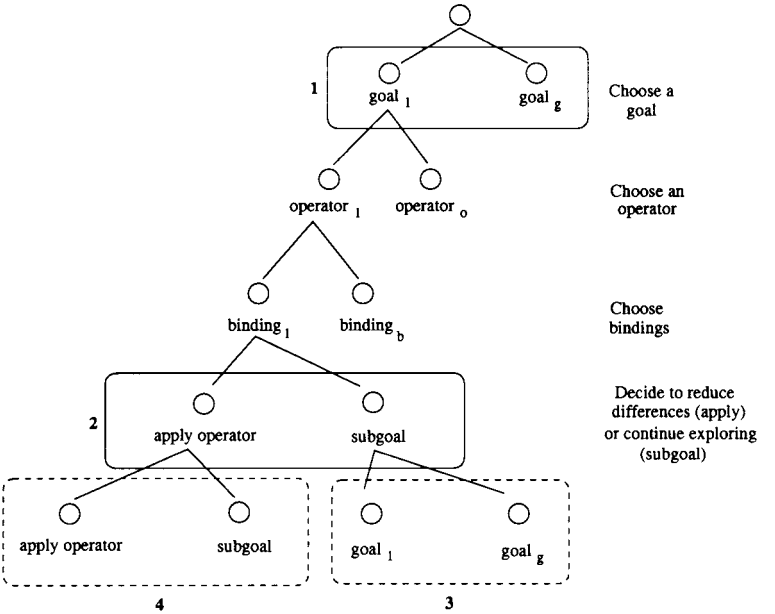

Figure 2. Tree of decisions generated by PRODIGY when searching for a solution to a problem. Decisions 3 and 4 enclosed in dashed rectangles are of the same type as 1 and 2, respectively.

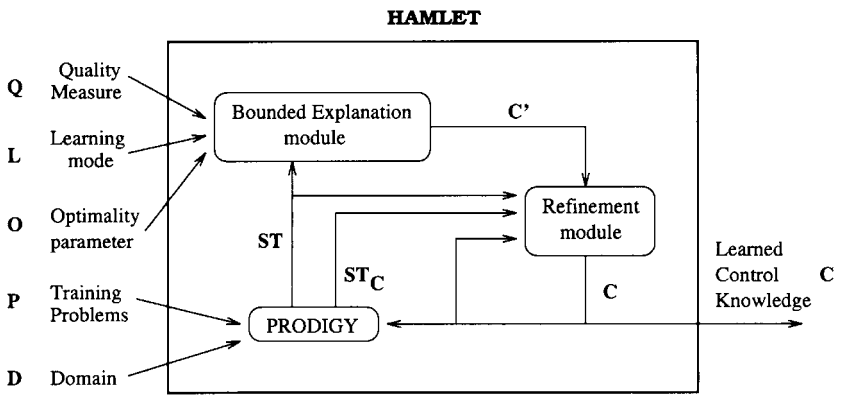

Figure 3. HAMLET's high level architecture.

The Bounded Explanation module generates control rules from a PRODIGY search tree. These rules might be overly specifi or overly general. The Refinemen module solves the problem of being overly specifi by generalizing rules when analyzing positive examples. It also replaces overly general rules with more specifi ones when it f nds situations in which the learned rules lead to wrong decisions. HAMLET gradually learns and refine 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 . ST and $ST_{\mathcal{C}}$ are planning search trees generated by two calls to PRODIGY's planning algorithm, $\mathcal{C}$ is the set of control rules, and $\mathcal{C}'$ is the new set of control rules learned by the Bounded Explanation module.

Figure 4 outlines HAMLET's learning algorithm.

Function **Hamlet** ($\mathcal{P}$, $\mathcal{D}$,Q, L, O)

$\mathcal{P}$ is the set of training problems
$\mathcal{D}$ is the domain description
Q is a quality measure
L is the learning-mode, which can be either *lazy* or *eager*
O is the boolean *optimality parameter*
$\mathcal{C}$ is the set of learned control rules, initially $\emptyset$
P is a problem
ST is the search tree for solving a problem P without control knowledge, $\mathcal{C} = \emptyset$
$ST_{\mathcal{C}}$ is the search tree for solving a problem P with control knowledge $\mathcal{C}$
$\mathcal{C}'$ is the new set of control rules learned from solving P

Forall $P \in \mathcal{P}$ do
  $ST \leftarrow$`search-tree`(**Prodigy4.0**(initial-state(P),goal(P),$\mathcal{D}$, $\emptyset$))
  $ST_{\mathcal{C}} \leftarrow$`search-tree`(**Prodigy4.0**(initial-state(P),goal(P),$\mathcal{D}$, $\mathcal{C}$))
  $\mathcal{C}' \leftarrow$**Bounded-Explanation**(ST, Q, L, O)
  $\mathcal{C} \leftarrow$**Refinemen** (ST, $ST_{\mathcal{C}}$, $\mathcal{C}$, $\mathcal{C}'$)
Return $\mathcal{C}$

Figure 4. A high-level description of HAMLET's learning algorithm.

For each problem P in the set of training problems $\mathcal{P}$, HAMLET calls PRODIGY4.0 twice. In the first call, PRODIGY generates a search tree ST to identify the optimal solutions. This is done by solving P without any control rules, exhausting the search space.[5] HAMLET generates new positive examples from ST, as ST was not pruned by any control rules. In the second call, PRODIGY uses the current set of learned control rules $\mathcal{C}$ and produces a search tree $ST_{\mathcal{C}}$. HAMLET identifies possible negative examples by the pruned search tree, $ST_{\mathcal{C}}$, with the complete search tree ST. Positive and negative examples are used to refine the learned rules to produce the new set of control rules $\mathcal{C}$.

[5] PRODIGY also generates a plan that HAMLET does not use.

3. Bounded Explanation

The Bounded Explanation module learns control rules by lazily choosing "key" decisions (as will be characterized later) made during the search for a solution and extracting the information that justifi s these decisions from the search space. The explanation procedure consists of three phases: Labeling the search tree; Credit assignment; and Generation of control rules. The Bounded Explanation module behaves lazily in two aspects:

- In contrast with *eager* approaches that learn control knowledge for planning (e.g., (Minton 1988; Etzioni 1993)), HAMLET does not require learning initially correct or complete knowledge. Incremental refinemen will be responsible for the correctness of the control knowledge at the end of the learning process. Therefore, there is no need for additional domain axioms.
- It does not require to learn from all search paths, as opposed to an *eager* learner that would learn from all decision nodes. In our experiments with multiple domains, we found that there is no need to learn from all decision nodes. Instead, one could only learn from the ones that: lead to successful solutions; were not the best alternative considered; and belonged to the path of the best solution (according to a quality measure, Q).

Function **Bounded-Explanation** (ST, Q, L, O)

ST is the search tree of solving a problem P without control knowledge, $\mathcal{C} = \emptyset$

Q is a quality measure

L is the learning-mode, which can be either *lazy* or *eager*

O is the boolean *optimality parameter*

$\mathcal{C}'$ is the set of new learned control rules

$\mathcal{C}' \leftarrow$**Label**(`root`(ST), Q, L, O)

If $O =$*True*

Then Return `follow-best-path`(root(ST))

Else Return $\mathcal{C}'$

Figure 5. Bounded Explanation high level algorithm.

Figure 5 shows a high level description of the Bounded Explanation algorithm. The algorithm takes as input a search tree ST generated without using the learned control knowledge, the *quality measure* Q, the *learning-mode* L, and the *optimality parameter* O. It returns a new set of control rules learned from the search tree decisions. The function `Label`, explained in subsection 3.1, assigns labels to each node of the search tree and possibly learns control rules. If the *optimality parameter* is true, then it will delay learning until it fin shes labeling the whole tree, so that it learns only from the path to the best solution (function `follow-best-path`). If false, it will learn at the same time that it labels the nodes of the search tree.

Function **Label** (N, Q, L, O)

N is the node HAMLET is going to label
Q is a quality measure
L can be either *eager* or *lazy*
O is the optimality parameter
$\mathcal{C}$ is the set of new control rules without ref nement, initially $\emptyset$
$\mathcal{S}$ is the set of successors of node N

Forall $s \in \mathcal{S}$ do $\mathcal{C} \leftarrow \mathcal{C} \cup$ **Label**(s, Q, L, O)
Case of
- If $\mathcal{S} = \emptyset$
  Then Case of
    If `solution-path`(N) Then `node-label`(N)←*success*
    If `failed-path`(N) Then `node-label`(N)←*failure*
    If `untried`(N) Then `node-label`(N)←*unknown*
- If $\exists s \in \mathcal{S}$ | `node-label`(s)=*unknown*
  Then If L =*eager* AND $\exists s' \in \mathcal{S}$ | `node-label`(s')=*success*
    Then $\mathcal{C} \leftarrow \mathcal{C} \cup$ **Store-or-learn-rule**(N, Q, L, O)
    Else `node-label`(N)←*unknown*
- If $\exists s \in \mathcal{S}$ | `node-label`(s)=*success* AND $\nexists s' \in S$ | `node-label`(s')=*unknown*
  Then $\mathcal{C} \leftarrow \mathcal{C} \cup$ **Store-or-learn-rule**(N, Q, L, O)
- If $\forall s \in \mathcal{S}$ `node-label`(s)=*failure*:
  Then `node-label`(N)←*failure*

Return $\mathcal{C}$

Figure 6. High level description of HAMLET's labeling procedure.

3.1. *Labeling the Search Tree*

HAMLET traverses the search tree bottom-up, starting from the leaf nodes. After labeling the leaf nodes, the algorithm propagates the labels up to the root of the search tree, using the algorithm described in Figures 6 and 7.

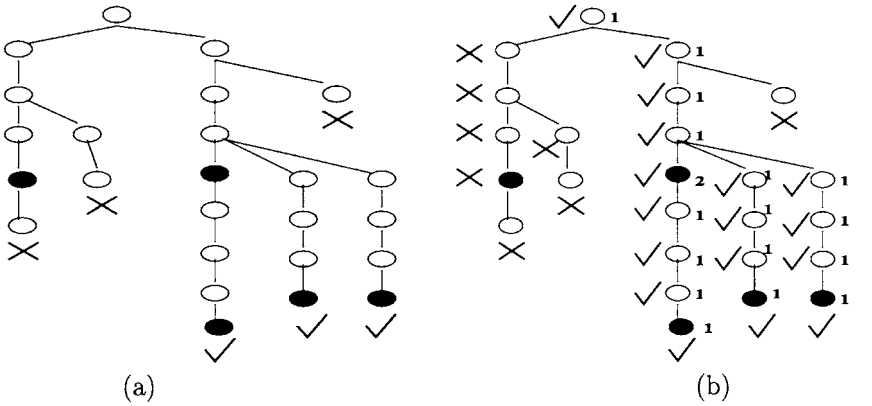

Figure 7. (a) A sketch of a PRODIGY search tree, where each leaf node is labeled as success (√) or failure (X); (b) The same tree after HAMLET labels it and attaches the optimal solution length from each node. The black nodes correspond to the operators applied to the state. A solution is the sequence of applied operators in a path from the root node to a success leaf node.

HAMLET 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.

The parameter *learning-mode* controls the way to label a node, as well as the way in which it assigns credit (discussed in next subsection). If its value is *eager*, HAMLET will eagerly label as *success* every node that has at least one success child, even if it did not explore all its subtrees. If its value is *lazy*, then HAMLET will label as *success* only the nodes that explored all their subtrees, and had at least one *success* child. The function `Store-or-learn-rule` (Figure 9) decides f rst whether to learn from the node's decision. If it decides learning, it also considers whether directly learning a control rule from the node's decision (function `learn`), or delaying the learning process until the end of the tree labeling: it stores now the decision with function `store` and later learns it with function `learn`. In the second case, HAMLET would only learn from nodes in the optimal path (see subsection 3.2). The function `learn` creates and returns a new control rule according to the target concept corresponding to the decision made in the node, the state at that node, and the meta-level information from the search tree (see Subsection 3.3). Figure 7 shows an illustration of the labelling procedure.[6]

[6] In this example, there are no unknown labeled nodes.

Figure 7(a) shows an example of a typical search tree, in which each leaf node is labeled by the PRODIGY planner as success ($\surd$) or failure (X). Figure 7(b) shows how HAMLET propagates labels to the root of this tree. In general, there might be several solutions to a problem as shown by the different solution paths. The nodes in each solution path are also labeled with the length of the optimal solution that can be reached from this node.

3.2. *Credit Assignment*

Credit assignment is the process of selecting important branching decisions for which learning will occur. It is done concurrently with labeling. Two parameters, *optimality parameter* O and *learning-mode* L, control the way in which HAMLET assigns credit. If O is *true*, the system learns only from the paths that lead to optimal solutions of the problem. It waits until the whole tree is labeled to generate the control rules, since this is the only way to know when a node is in an optimal solution path. If *optimality parameter* is *false*, HAMLET learns from every path to a solution. In this case, it does not have to wait to finish the credit assignment and labeling to generate the control rules. Instead, it interleaves credit assignment and generation of the control rules.

When we first designed HAMLET, it would learn from every node in a solution path. If all the possible solutions to a given problem are of the same quality (according to a given criteria), or one is only concerned with learning to produce the solutions more efficiently, then this eager approach is correct. However, we wanted to create a learning system capable of also improving the quality of the solutions provided. Therefore, we created a parameter, *learning-mode*, that would control the way in which HAMLET assigns credit. If it is *eager*, HAMLET views a branching decision as a learning opportunity only if the decision leads to any solution. If *lazy*, a decision is a learning opportunity if it leads to an optimal solution and differs from the default decision made by the domain-independent heuristics. As discussed in Section 5, *eager* mode learns many more rules than *lazy* mode. We found that *lazy* mode was almost always more efficient in solving problems, and also the solutions obtained were better according to the quality measure Q.

Figure 8 shows the learning opportunities that HAMLET finds in the example search tree of Figure 7(b). In Figure 8(a), the thick solid lines show the branching decisions that would be learned in *lazy* mode. In Figure 8(b), the dashed lines indicate the additional decisions learned in *eager* mode.

3.3. *Generation of Control Rules*

At each decision choice to be learned, HAMLET has access to information on the current state $\mathcal{S}$, and on the meta-level planning information, such as the

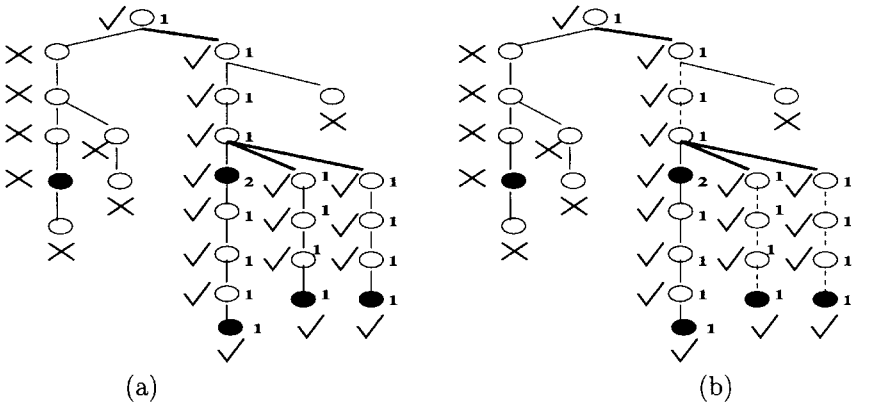

Figure 8. The learning opportunities corresponding to the example labeled tree of Figure 7(b). The default decisions are the left-most successors of every node.

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 (the function `learn` in Figure 9) to create the applicability conditions (i.e., the antecedents of the control rules). The relevant predicates of the current state are selected using *goal regression* (Waldinger 1981).

Function **Store-or-learn-rule** (N, Q, L, O)

N is the node HAMLET is going to label
Q is a quality measure
L can be either *eager* or *lazy*
O is the optimality parameter
$\mathcal{B}$ is the set of best successors of N according to Q

`node-label`$(N) \leftarrow$*success*
If $L =$*eager* OR `default-alternative`$(N) \notin \mathcal{B}$
Then If O=True
  Then `store`$(N, \mathcal{B})$
    Return $\emptyset$
  Else Return $\{$`learn`$(N, \mathcal{B})\}$
Else Return $\emptyset$

Figure 9. Auxiliary function for HAMLET's labeling procedure.

HAMLET learns five kinds of *select* control rules, corresponding to PRODIGY's decisions as discussed in Section 2. These are the generalized target concepts:[7]

- decide to apply an *operator* for achieving a *goal*;
- decide to subgoal on an unachieved *goal*;
- select an unachieved *goal*;
- select an *operator* to achieve some *goal*; and
- select bindings for an *operator* when trying to achieve a *goal*.

They are instantiated on each domain by replacing the above variables (words in italics) by specific names of operators and goals. For instance, there will be one target concept of the type select-operator for each possible <*operator, goal*> pair, where *goal* refers to the goals that can be achieved by *operator*. As an example in the blocksworld, two target concepts will be *select unstack for achieving holding-object* and *select pick-up for achieving holding-object*. HAMLET generates a set of rules for each target concept, where the antecedent of each rule is described as a conjunctive set of tests.[8] 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.

Each of the five kinds of generalized target concepts has a template for describing its antecedents as shown in Figure 10. The templates share a set of common predicates for all kinds of control rules, but each kind has certain local predicates. Below are the predicates used in the antecedents of all five types of control rules (i.e., the common predicates):

- True-in-state *assertion*: tests whether *assertion* is true in the current state of the search.
- Other-goals *list-of-goals*: tests whether any of the goals in *list-of-goals* is an unachieved goal.
- Prior-goals *list-of-goals*: tests whether any goal in *list-of-goals* is the top-level goal that created the subgoal which the planner is currently trying to achieve.
- Type-of-object *object type*: tests whether *object* is of type *type*.

The local predicates that are used in some, but not all, kinds of control rules, are:

- Target-goal *goal*: tests whether *goal* is one of the unachieved goals.
- Current-goal *goal*: tests whether *goal* is the current goal the planner is trying to achieve.

[7] HAMLET does not yet learn rules to control the choice of which operator to apply (see step 3.2.1 in Figure 1), as this decision has only been recently added to PRODIGY4.0 as a control choice point.

[8] Some of the tests are, in fact, disjunctive, such as *other-goals* and *prior-goals*.

(control-rule *name*
(if (and (current-goal *goal-name*)
[(prior-goals (*literal**))]
(true-in-state *literal*)*
(other-goals (*literal**))
(type-of-object *object type*)*))
(then select **operators** *operator-name*))

(a)

(control-rule *name*
(if (and (current-operator *operator-name*)
(current-goal *goal-name*)
[(prior-goals (*literal**))]
(true-in-state *literal*)*
(other-goals (*literal**))
(type-of-object *object type*)*))
(then select **bindings** *bindings*))

(b)

(control-rule *name*
(if (and (applicable-op *operator*)
[(prior-goals (*literal**))]
(true-in-state *literal*)*
(other-goals (*literal**))
(type-of-object *object type*)*))
(then decide {**apply**|**sub-goal**}))

(c)

(control-rule *name*
(if (and (target-goal *literal*)
[(prior-goals (*literal**))]
(true-in-state *literal*)*
(other-goals (*literal**))
(type-of-object *object type*)*))
(then select **goals** *literal*))

(d)

Figure 10. Templates (regular expressions) of the five kinds of target concepts. They correspond to the decisions: (a) operator decision; (b) bindings decision; (c) decide to apply or subgoal, (both have the same antecedent); and (d) goal decision.

- Current-operator *operator*: tests whether *operator* is the operator PRODIGY is considering to achieve a goal.
- Applicable-op *instantiated operator*: tests whether *instantiated operator* is applicable in the current state.

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. When the rule is applied, different variables must always be matched with distinct constants. The latter heuristic can be relaxed when generalizing rules.

3.4. *Example*

We show now a simple example of how control knowledge can be generated. The domain we use is a logistics-transportation domain (Veloso 1994b). In this domain, packages must be delivered to different locations in several 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 the problem in Figure 11, where there are three cities, each with 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*.

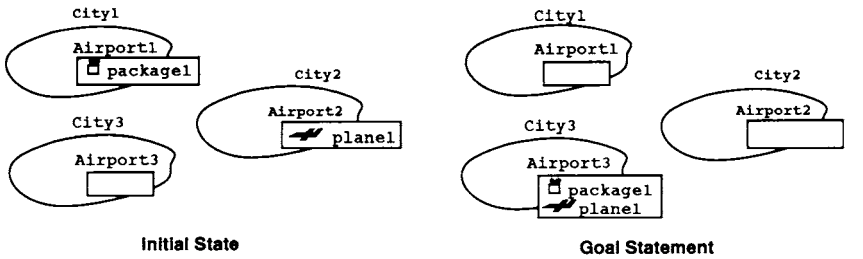

Figure 11. An illustrative example – initial state and goal statement.

```
(control-rule select-bind-fl  -airplane-1
  (if (and (current-operator f y-airplane)
           (current-goal
               (at-airplane <plane1> <airport3>))
           (true-in-state
               (at-airplane <plane1> <airport2>))
           (other-goals
               ((at-object <package1> <airport3>)))))
  (then select bindings ((<plane> . <plane1>)
                         (<from> . <airport1>)
                         (<to> . <airport3>))))
```

(a)

```
(operator FLY-AIRPLANE
  (preconds
     ((<plane> AIRPLANE)
      (<from> AIRPORT)
      (<to> AIRPORT))
     (at-airplane <plane> <from>))
  (effects
     ((add (at-airplane <plane> <to>))
      (del (at-airplane <plane> <from>)))))
```

(b)

Figure 12. (a) Control rule created by HAMLET; (b) The operator FLY-AIRPLANE. The rule controls the selection of the bindings for instantiating the operator's variables.

After PRODIGY solves the problem, HAMLET learns the rule shown in Figure 12, which would be the one that a standard EBL system would learn. The rule says that the planner should f y an airplane, *plane1*, from *airport1* to *airport3* if the goal is to have the airplane at *airport3*, it is initially in *airport2*, and it has another goal of having a package, *package1*, at the same destination airport, *airport3*. This rule is erroneous, since it does not capture why it should f y from *airport1*, instead of fly ng from *airport2*, which is where *plane1* is in the current state. What is missing is the location of *package1* in the state. If *package1* is at *airport1*, then everything is explained correctly, as it is captured by the rule in Figure 13. However, if *package1* is at *airport2* in the state of another similar problem, then this rule would f re incorrectly, leading

to a nonoptimal solution, as shown in Subsection 4.3. While traditional EBL systems could not recover from this erroneous rule, HAMLET's refine ent approach specializes the rule, correcting it, as explained in Section 4.

```
(control-rule select-bind-f y-airplane-2
   (if (and (current-operator f y-airplane)
            (current-goal (at-airplane <plane1> <airport3>))
            (true-in-state (at-airplane <plane1> <airport2>))
            (true-in-state (at-object <package1> <airport1>))
            (other-goals ((at-object <package1> <airport3>)))))
   (then select bindings ((<plane> . <plane1>)
                          (<from> . <airport1>)
                          (<to> . <airport3>))))
```

Figure 13. Rule learned by HAMLET after specializing the overly general rule in Figure 12.

4. Refin ment. Generalization and Specialization

The rules generated by the Bounded Explanation module may be overly specific as also noticed by (Etzioni and Minton 1992), or overly general, as we showed in Section 3. To address this problem, HAMLET uses the Ref nement module, which generalizes the learned rules by analyzing new examples of situations where the rules are applicable, and specializes the overly general rules when it f nds new negative examples. We have devised methods for generalizing and specializing four aspects of the learned knowledge:

- *Current state:* The predicates from the state are the f rst ones to be removed, since their presence in the antecedents of the control rules is the reason why most of the rules are overly specific
- *Subgoaling structure:* We may relax the subgoaling links, for example as captured by the *prior-goal* predicate, since the same goal can be generated as a subgoal of many different goals.
- *Interacting goals:* Another way of relaxing the preconditions of rules consists of identifying the correct subset of the set of pending goals that affect a particular decision.
- *Object hierarchy:* Finally, it is also important to fi d the generalization level to which the variables in the control rules belong, considering the ontological type hierarchy that is available in the nonlinear version of PRODIGY.

HAMLET's refinemen component behaves lazily in that it delays generalizing until new examples are found, and when these examples are found, the

Function **Refin ment** ($ST, ST_{\mathcal{C}}, \mathcal{C}, \mathcal{C}'$)

ST is the search tree of solving a problem P with $\mathcal{C} = \emptyset$
$ST_{\mathcal{C}}$ is the search tree of solving a problem P with control knowledge $\mathcal{C}$
$\mathcal{C}$ is the set of learned control rules
$\mathcal{C}'$ is the set of new control rules without ref nement
$\mathcal{T}$ is the set of target concepts for which there is at least one new negative example
R is a new learned control rule of the set $\mathcal{C}'$

Forall $R \in \mathcal{C}'$ do $\mathcal{C} \leftarrow$**Generalize**($R, \mathcal{C}$)
$\mathcal{T} \leftarrow$`find-negative-examples`($ST, ST_{\mathcal{C}}$)
Forall $T \in \mathcal{T}$ do $\mathcal{C} \leftarrow$**Specialize**($T, \mathcal{C}$)
Return $\mathcal{C}$

Figure 14. A high-level description of the Ref nement algorithm.

generalization does not consider all previously seen examples, but only the ones that are being used at that time for their corresponding target concepts. This can be considered similar to other lazy learning systems that only keep prototypes of the different classes (Porter et al. 1990). However, a major difference with these approaches is that HAMLET still keeps all examples seen that are not subsumed by others.[9] This is needed for refinemen purposes, as explained later in this section. A future research direction would study the possibility of only keeping some of them by computing the set of predicates that most probably will correctly refin an overly general control rule.

Figure 14 shows the top-level description of the Refinemen module algorithm. It fir t calls the generalization phase (in case new positive examples were found, that is, new rules), followed by the specialization phase (in case any negative example was found). The function `find-negative-examples` returns the list of target concepts for which there is at least one new negative example. Negative examples are found by analyzing the differences between the search trees generated using the learned control rules ($ST_{\mathcal{C}}$) and when not using them (ST). The following subsections describe the refine ment algorithms. Subsection 4.1 discusses the generalization process, Subsection 4.2 presents the specialization process, and Subsection 4.3 presents an example of specialization.

[9] We use a similar concept to ILP's θ-subsumption (Muggleton 1992).

4.1. *Inductive Generalization of Control Knowledge*

Figure 15 presents the generalization algorithm. Upon generating a new control rule for a target concept, HAMLET tries to generalize it with previously learned control rules of the same target concept. If there were none, then the rule is stored as the only one of its target concept. If there were rules, the function `generalize-rule` will try to apply the generalization operators to combine all the rules of the same target concept. If it succeeds, it will create a new rule and delete the old ones. Also, it will recursively call itself with the new induced rule. If it fails to generalize the rule with any other rule, the rule is simply added to the set of rules of the target concept.

Function **Generalize** $(R, \mathcal{C})$

R is a new learned control rule
$\mathcal{C}$ is the set of learned control rules
T is the target concept of rule R
$\mathcal{R}$ is the old set of rules for target concept T
$\mathcal{R}'$ is the new set of rules for target concept T

If $\mathcal{R} = \emptyset$
Then `target-concept-rules`$(T) \leftarrow \{R\}$
Else $\mathcal{C} \leftarrow \mathcal{C} - \mathcal{R}$
  $\mathcal{R}' \leftarrow$`generalize-rule`$(R, \mathcal{R}, T)$
  If $\mathcal{R}' = \emptyset$
  Then `target-concept-rules`$(T) \leftarrow$`target-concept-rules`$(T) \cup \{R\}$
  Else `target-concept-rules`$(T) \leftarrow \mathcal{R}'$
Return $\mathcal{C} \cup$ `target-concept-rules`(T)

Figure 15. Algorithm for the generalization of control rules.

A graphical example of the generalization process within a target concept is shown in Figure 16. "WSP_i" stands for *Whole Set of Preconditions* for $rule_i$ and refers to the complete description of the current state and all available information about the meta-level decision on a certain search node.

4.2. *Specialization of Overly General Control Rules*

HAMLET may generate overly general rules, either by *goal regression* when generating the rules, or by applying the generalization step. The overly general rules need to be specialized. There are two main issues to be addressed: how to detect a negative example, and how to refin the learned knowledge according to it, shown in Figure 17.

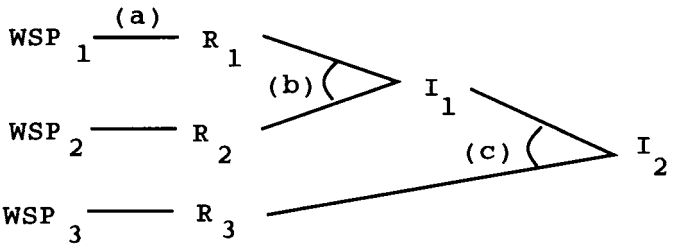

Figure 16. Three learning episodes of HAMLET within the same target concept. (a) shows the initial step of bounding the explanation from the *Whole Set of Preconditions* to R_1. (b) shows a generalization step from another rule R_2, generating induced rule I_1. (c) shows a second generalization step, generating I_2 from I_1 and R_3.

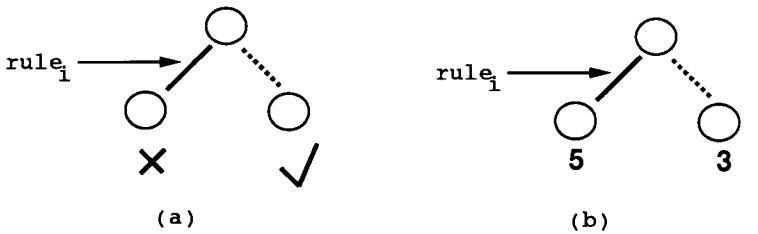

Figure 17. Two cases of negative examples (i.e., situations in which a rule applies incorrectly): (a) The rule selects a node that leads to a failure path; (b) The rule selects a node that leads to a node with a longer solution (f ve operators) than the best one (only three).

Defini ion: A negative example for HAMLET is a situation in which a control rule was applied, and the resulting decision led to either a failure (i.e., instead of the expected success), or a nonoptimal solution for that decision.

Negative examples are also represented as rules with their whole set of preconditions, WSP. They are stored in their corresponding target concept, and serve two purposes: refin an overly general rule, and establish an upper limit of generalization for future applications of the generalization module. Every time a rule is generated by either the Bounded Explanation or when applying the generalization module, it is checked against the negative examples of its target concept to determine whether it covers any of them. If so, the rule is refi ed. This is a *lazy* aspect in the sense that it does not try to obtain complete descriptions from all examples of the target concepts. Instead, it refine on demand the representative examples of each target concept, when new positive or negative examples are found.

4.2.1. *The specialization algorithm*

Since there are two kinds of rules (bounded and induced), there are two kinds of recovering methods. The bounded rules are refine by adding literals from their corresponding WSP set. The induced rules come from two generating

rules, so HAMLET tests whether each one of its corresponding generating rules also covers the negative example. If so, then HAMLET recursively refine that rule. If not, the induced rule is refine using a set of refinemen operators. The top-level specialization algorithm is described in Figure 18.

Function **Specialize** $(T, \mathcal{C})$

T is an erroneous target concept (a negative example was found)
$\mathcal{C}$ is the set of learned control rules
$\mathcal{R}$ is the old set of rules for target concept T
$\mathcal{R}'$ is the new set of rules for target concept T, initially $\emptyset$
$\mathcal{N}$ is the set of negative examples for T

$\mathcal{C} \leftarrow \mathcal{C} - \mathcal{R}$
Forall $R \in \mathcal{R}$ do
  If `describes-negative-examples`$(R, \mathcal{N})$
  Then $\mathcal{R}' \leftarrow \mathcal{R}' \cup$ `refine-rule`$(R, \mathcal{N}, \mathcal{R}')$
  Else $\mathcal{R}' \leftarrow \mathcal{R}' \cup \{R\}$
`target-concept-rules`$(T) \leftarrow \mathcal{R}'$
Return $\mathcal{C} \cup \mathcal{R}'$

Figure 18. HAMLET's specialization algorithm.

Figure 19 shows the case in which, while refin ng an overly general rule I_2,[10] one of its generating rules, I_1, was also overly general. In this case, it backtracks, and refine I_1 and I_2 with the rules that generated them, R_1, R_2, and R_3. R_1 was also overly general, so it had to backtrack to consider WSP_1, generating a refine version of R_1, RF_1. It generates RF_1 by find ng one or two preconditions of the set *preconds*(WSP_1) – *preconds*(R_1) that added to the set *preconds*(R_1) do not cover the negative examples. Then, HAMLET deletes R_1. R_2 was not overly general, so it created a rule RF_2 from R_2 and I_1, then deleted I_1. Finally, R_3 was not overly general, so it generated a new rule RF_3, and deleted I_2. The dotted lines represent deleted links, the dotted boxes deleted rules, and the solid boxes the active rules after refinemen .

4.2.2. *Choosing the right precondition to add*

HAMLET fi st tries to add preconditions from a set called *preferred-preconds*. We have found that the reason why most control rules were overly general was that they did not consider the needed preconditions (goal regression) for the other goals present in the decision. Instead of *eagerly* adding those

[10] R_i means a directly learned rule, I_i a generalized rule, and RF_i a specialized rule.

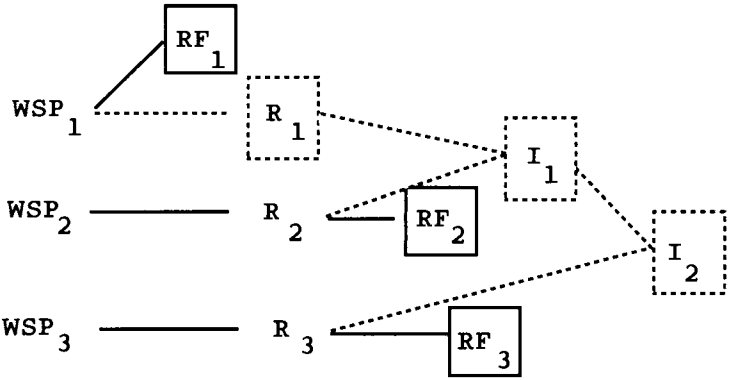

Figure 19. Graphical representation of the ref nement of an overly general control rule I_2.

preconditions when the rule is created, the refin ment algorithm incrementally adds the ones that it f nds necessary for solving the over generality. When trying to add a precondition, HAMLET creates a pool of preconditions that can be added to the antecedent of the control rule,[11] it sorts the pool, and then it tests each one's coverage of negative examples. If a rule was directly learned by the Bounded Explanation module (i.e., it has no previous rules), then the tests are taken from its corresponding WSP. If it was an induced or refine rule, the tests are taken from the preconditions of its corresponding generating rules. The preconditions are always sorted according to an ordering function that prefers the preconditions that:

- refer to more variables that already appear in preconditions of the control rule – in this way, we keep the locality heuristic that was proposed in earlier work (Muggleton 1992; Borrajo et al. 1992a, b; Borrajo and Veloso 1994), and that empirical results show was effective-;
- refer to literals that have appeared more times in the preconditions of control rules of the same target concept; and
- refer to tests over the state rather than to prior goals, since computing a precondition referring to the prior goals is frequently more expensive than computing a precondition referring to the state.

4.3. *Example*

Suppose that PRODIGY solves a new problem in which *package1* and *plane1* also have to go to *airport3*, but this time they are initially in the same airport, *airport2*. When solving this problem using the overly general rule in Figure

[11] HAMLET creates this pool from the rest of preconditions that were not introduced directly into the rule when it was created.

12, PRODIGY yields a nonoptimal solution in the path that applies the rule, such as:

Plan	Achieved goals
fly airplane(plane1,airport2,airport1)	
fly airplane(plane1,airport1,airport3)	(at plane1 airport3)
fly airplane(plane1,airport3,airport2)	
load-airplane(object1,plane1,airport2)	
fly airplane(plane1,airport2,airport3)	
unload-airplane(object1,plane1,airport3)	(at-object object1 airport3)

In order to solve this problem, the specialization algorithm selects a test on the location of the object, since it is part of the goal regression of the other pending goals. Then, it creates the new control rule that was shown in Figure 13.

5. Empirical Results

We conducted extensive empirical tests to study the performance of HAMLET's learning algorithms in several experimental domains and problems within domains. We report in this article the results we obtained in two domains, the blocksworld and the logistics transportation domain.[12] The experiments are mainly of three kinds to test the behavior of the system along the following dimensions:

- **Lazy versus eager learning.** We compare the performance of HAMLET when it learns form the training problems with different eagerness, namely lazily and eagerly. Following a "lazy" learning bias, HAMLET learns control knowledge only from choices that contradict the default PRODIGY's heuristic behavior. Using an "eager" learning bias, HAMLET learns from every successful choice. The experiments interestingly show that HAMLET performs better with lazy than with eager learning.
- **Improvement of plan-generation efficienc and plan quality.** We show how HAMLET's *lazy* mode improves the search effic ency of the Base-Level PRODIGY planner.[13] More importantly, the experiments show that HAMLET learns knowledge capable of improving the quality of the generated plans.

[12] We have gathered results from other domains that will not be presented in this paper.

[13] All future references to Base-Level PRODIGY will refer to the planner using only its default heuristics and no control rule.

- **Convergence towards the correct learned knowledge.** These experiments illustrate the effect from incrementally refinin the learned knowledge. The results show that HAMLET in *lazy* mode converges to the correct set of control rules, as a function of the number of training problems seen.

The following subsections describe the experiments and discuss the results. In all experiments, all problems were randomly generated, different sets of problems were used for learning and testing, and the distributions used for both sets of problems were the same. Also, the *optimality parameter* was always set to `true`, quality is measured as the number of operators in the resulting plan, and every training phase began with no control rules, $\mathcal{C} = \phi$.

5.1. *Lazy vs. Eager Learning*

This experiment compares HAMLET's performance in *lazy* and *eager* learning modes. To show the difference between the two learning modes, we carried out the following experiment in the logistics domain. We trained HAMLET with the same 400 randomly-generated problems in *eager* and in *lazy* modes.[14] The learning problems were 200 one-goal problems involving up to four packages, and 200 two-goal problems involving up to fiv packages. In *lazy* mode, 19 rules were learned, while in *eager* mode, it learned 98 rules.

Then we tested the performance of the learned rules on 475 randomly generated problems from the same distribution as the learning set. The testing problems were of increasing complexity, generated by varying the number of goals in the problems from 1 to 10, and the maximum number of packages from 5 up to 20. The time bound given for all experiments reported in this article is $150 \times (1+\text{mod}(\text{number-goals},10))$ seconds.[15] The results are shown in Tables 1, 2 and 3. In Table 1 the f rst column shows the number of problems solved by Base-Level PRODIGY (281), and using the learned control rules (239). The second column shows the total time spent, where the time of the unsolved problems is the time bound. The rest of the columns show the comparison using only the problems that were solved by both configu ations. For instance, the third column shows the number of problems in which the solutions provided by Base-Level PRODIGY were better (i.e., according the the quality measure used) than using the learned control rules (eight cases) compared to the number of problems in which using the rules achieved better solutions than Base-Level PRODIGY (31 cases). The same applies for the other two columns. Table 2 shows the results using the rules learned in *lazy* mode

[14] *Eager* mode means calling HAMLET'algorithm (Figure 4) with the parameter $L = eager$, while *lazy* mode means calling HAMLET with $L = lazy$.

[15] We also tested using several time bounds, and the results were similar to the ones presented here.

using the same set of problems, and Table 3 shows the comparison between the two learning modes.

Table 1. Comparison of performance between Base-Level PRODIGY4.0 and PRODIGY4.0 using HAMLET's learned rules in *eager* mode for the logistics domain.

Rules Used	Solved Problems	Time (seconds)	Solved by both (197 problems)		
			Better solutions	Solution length	Nodes explored
no rules	281	58900	8	1208	7699
eager (98 rules)	239	72699	31	1164	5560

Table 2. Comparison of performance between Base-Level PRODIGY4.0 and PRODIGY4.0 using HAMLET's learned rules in *lazy* mode for the logistics domain.

Rules Used	Solved problems	Time (seconds)	Solved by both (278 problems)		
			Better solutions	Solution length	Nodes explored
no rules	281	58900	2	2995	16366
lazy (19 rules)	334	44091	89	2763	13030

Table 3. Comparison of performance between PRODIGY4.0 using HAMLET's learned rules in *lazy* and *eager* modes for the logistics domain.

Rules Used	Solved problems	Time (seconds)	Solved by both (228 problems)		
			Better solutions	Solution length	Nodes explored
eager (98 rules)	239	72699	12	1593	7810
lazy (19 rules)	334	44091	44	1474	6802

Discussion. In *lazy* learning mode, HAMLET performs more efficiently and yields higher quality solutions. With respect to efficiency, the number of solved problems increases from 50% up to 70%, while the time spent on solving the problems also decreases. In addition, the solutions provided are much better. More specifically, in *eager* mode, it performs worse for eight problems and better for 31. In *lazy* mode, it performs worse in only two problems, and better in 89! We believe HAMLET's lazy behavior is responsible for such good results, given that it learns only when it seems clear that a decision was the best one in any node in the search tree, and was not chosen by the default problem solver. That is, the goal of HAMLET is to provide the problem solver with a learned way of controlling when not to follow the

default procedure, thus letting the problem solver decide when the default decision is correct.

There is also an important difference in the behavior of both modes: the number of learned rules. Since the *eager* mode learns from every decision, it learns many more rules, and many more types of rules. For instance, suppose that operator O_1 appears in the domain description before operator O_2, and both are relevant to the same goal g. Then the default behavior of PRODIGY4.0 is that, when it does not have control knowledge to know which operator to use, it uses the first operator that appears in the domain description. Thus, it will always try O_1 before O_2. If O_1 is always the best operator to achieve the goal g, then the lazy mode of HAMLET will never learn a control rule for that decision, while the eager mode will learn a rule each time that it uses O_1 and succeeds in achieving a solution. Therefore, eager mode suffers more from the utility problem than the lazy mode as can be observed in these tables in the time columns.

With respect to learning time, learning in *lazy* mode is a relatively fast process, since it generates fewer rules, and, therefore, spends a short time learning. The time spent on learning after every PRODIGY4.0 run on a problem is between two and ten times less than PRODIGY's search time. In contrast, learning eagerly can be very slow, since the refinement procedure needs to match many more rules against each other. It might take up to 20 times PRODIGY's search time.

We have been working on many experiments to find additional evidence of these two slow down phenomena (i.e., number of rules and learning time), and we have found that the eager mode does not work for most domains. It overloads the system with control rules, not allowing HAMLET to finish the learning phase due to memory problems. A potential advantage of *eager* mode over *lazy* mode might be that the *lazy* mode requires the whole search tree to be expanded for learning control rules, while *eager* mode can learn as soon as any solution has been found. Therefore, *lazy* mode should be used for domains for which learning in easy problems scales well to complex problems. We have found empirically that this is true for most of our planning domains.

Finally, if we compare the solution length and the number of nodes of both modes against not using the rules, we see that in *eager* mode it saves 3.6% in solution length and 27.8% in number of nodes, while in *lazy* mode it saves 7.7% in solution length and 20.4% in number of nodes. The performance is almost doubled for *lazy* mode in solution length, and better for *eager* mode in number of nodes. The reason why the savings in number of nodes is bigger for *eager* mode is due to the number of control rules it has, which prune more of the search space.

5.2. *Improving Efficiency and Solution Quality*

To show that HAMLET improves not only the efficiency of the base-level problem solver, PRODIGY4.0, but also improves the quality of the solutions provided by PRODIGY4.0 with the learned knowledge, we have carried out experiments in the logistics transportation domain, described in Subsection 3.4, and in blocksworld. We use the blocksworld to show a domain in which HAMLET improves mostly the efficiency of the problem solver, and logistics as a domain in which it also greatly improves the quality of the solutions provided by the problem solver. In this last domain the quality of the solutions is an important aspect, due to the possible unnecessary movements of the carriers. HAMLET learns knowledge that allows the planner to generate effective solutions for transporting packages.

Blocksworld results. We trained HAMLET with 200 simple randomly generated problems of one and two goals, and up to ten blocks. HAMLET generated 11 control rules. Then, we randomly generated 375 testing problems of increasing complexity. Table 4 shows the results of those tests. We varied the number of goals in the problems from one up to ten, and the maximum number of blocks from five up to 50.

Table 4. Results on increasingly complex problems in the blocksworld domain.

Test sets		Solved problems		Solved by both (174 problems, 46.4%)					
				Better solutions		Solution length		Nodes explored	
Goals	Problems	without rules	with rules	without rules	with rules	without rules	with rules	without rules	with rules
1	100	68	100	0	0	245	245	9227	1187
2	100	56	92	1	2	196	193	3924	1258
5	100	45	80	1	3	188	180	4085	1895
10	75	13	34	0	1	76	72	11048	313
Totals	375	182	306	2	6	705	690	28284	4653
%		48.5%	81.6%	1.1%	3.4%			Ratio	6.1

Discussion. The results show a remarkable improvement on the Base-Level PRODIGY solver performance when using the learned control rules. As an example, it increases the number of solved problems from 48.5% up to

81.6%, and expands six times fewer number of nodes.[16] It also generates better solutions using the rules in six occasions, while Base-Level PRODIGY only generates better solutions in two occasions. These results on solution quality are not as impressive as in the logistics domain (detailed below) due to the fact that, in the blocksworld task, most problems have the same number of operators in all possible solutions to the same problem.

Logistics results. We trained HAMLET in *lazy* mode with 400 simple randomly generated problems of one and two goals, and up to three cities and fi e packages. HAMLET generated 26 control rules. Then, we randomly generated 525 testing problems of increasing complexity. Table 5 shows the results of those tests. We varied the number of goals in the problems from one up to 50, and the maximum number of packages from f ve up to 50. Problems with 20 and 50 goals pose very complex problems to f nd good or optimal solutions, even for humans.

Table 5. Results on increasingly complex problems in the logistics domain.

Test sets		Solved problems		Solved by both (279 problems, 53.14%)					
				Better solutions		Solution length		Nodes explored	
Goals	Problems	without rules	with rules	without rules	with rules	without rules	with rules	without rules	with rules
1	100	95	100	0	11	327	307	2097	1569
2	100	85	94	0	25	528	479	3401	2308
5	100	56	82	1	33	865	777	5170	3463
10	100	32	68	1	24	770	668	3482	2941
20	75	13	39	0	10	505	455	2216	1924
50	50	1	10	0	0	34	34	143	141
Totals	525	282	393	2	103	3029	2720	16509	12346
%		53.7%	74.9%	0.7%	36.9%			Ratio	1.3

Discussion. The results show again a remarkable increase in the number of solved problems using the learned rules, as well as a large number of problems (103) in which the solution generated using the rules is of better quality, compared to the two problems in which Base-Level PRODIGY produced a

[16] In 10 goal problems using the learned rules, it expands 35 times fewer number of nodes in the search tree.

solution of better quality. The running times decreased using the rules, but not significan ly.

Learning control knowledge to improve problem solving performance is a potential source of inneff ciency due to the tradeoff between the savings obtained and the cost of using the learned knowledge. This is generally known as the *utility problem* in speedup learning (Minton 1988). Interestingly, HAMLET does not suffer substantially from the utility problem even with no special organization of the learned knowledge.[17] Due to its inductive step over the control rules of the same type, HAMLET keeps a small number of control rules, instead of retaining many variations of each rule type. This is one of the sources of power of the induction step applied to the bounded explained control rules. Nevertheless, we are currently developing effic ent methods for organizing and matching the learned control rules. We consider this organization essential to the overall learning process to avoid a potential utility problem due to inefficien matching (Doorenbos and Veloso 1993).

After analyzing why some problems were not solved, we concluded that some rules were not correct after the training phase. This fact led us to carry on the next set of experiments towards testing the convergence of the learning approach.

5.3. *Convergence to the Correct Control Knowledge*

The previous results are important in the sense that they show that the learned rules after 400 training problems perform well. But someone could ask if the learned knowledge improves over time when more training problems are given, or it begins to oscillate in a local maximum, adding and removing the preconditions of the control rules. This is an important issue for any machine learning system, especially for a *lazy* system as HAMLET, since HAMLET relies on incremental refine ent to correctly characterize the learned control rules. Therefore, it needs to converge as it sees more training examples.

To show that HAMLET improves with the number of training problems, we performed the following experiment: we trained HAMLET in *lazy* mode with 75 problems, 150 problems, and 400 problems in the logistics domain. Then, we tested the respective learned control rules on the same test set as before (525 problems of increasing complexity). Table 6 shows results that clearly illustrate the rules converge towards the correct behavior.

Discussion. After analyzing these results and the previous ones, we noted that there were problems in which HAMLET did not learn anything. These

[17] This is true even for *eager* mode that learns many more rules than *lazy* mode.

Table 6. Results on convergence in the logistics domain.

Training problems	Solved problems		Solved by both				
			Better solutions		Ratio Solution Length	Ratio Time	Ratio Nodes
	without rules	with rules	without rules	with rules	without/ with rules	without/ with rules	without/ with rules
75	53.71%	63.62%	0.35%	25.89%	1.11	0.49	1
150	53.71%	65.71%	0.72%	31.9%	1.06	0.33	1.25
400	53.71%	74.86%	0.72%	36.92%	1.08	0.32	1.34

problems belong to two different groups. The first group are problems that were too easy. They were solved by PRODIGY with no search, or the default strategy made the right decisions in the first place. The second group of problems were too complicated, so PRODIGY could not expand the whole search tree in the given time bound (usually 100 seconds), and, therefore, HAMLET could not learn any control rule. Here, there is some underlying assumption (bias) that influence HAMLET's behavior: HAMLET **will learn better from medium difficulty problems**. Having this bias in mind, the training problems were randomly generated so that: they were not too easy, such as problems in which the goals are true in the initial state; and they were not too complicated, such as problems with more than two goals. We believe that this is not an over simplifying assumption: control knowledge learned from them will still be applicable to easier or more difficult problems. As we have shown, the learned knowledge, even coming from a "biased" training phase, transfers well to more complicated problems, and does not interfere in the default PRODIGY behavior with simpler problems.

Since training can benefit from well-suited training problems, we are currently developing a better training schema in which problems are not generated randomly, but are biased towards the kinds of problems from which HAMLET learns better. This domain-independent biased generation of training problems should improve the convergence of HAMLET, and also reduce the training effort. We also plan to analyze, for each domain, the number of training problems that will be needed to obtain a certain degree of accuracy, based on research from computational learning theory (Valiant 1984).

6. Related Work

Most previous lazy learning approaches have been inductive approaches, such as the work in instance-based (Aha et al. 1991), memory-based (Stanfill and

Waltz 1986), or exemplar-based learning (Porter et al. 1990). Only some of the work has been applied to planning, usually in the context of analogy or case-based reasoning (Hammond 1989; Hanks and Weld 1995; Kambhampati 1989; Kettler et al. 1994; Veloso 1994a, b). Most of this work concerns domain-specifi algorithms. Also, although these approaches demonstrated some useful lazy learning behavior, they did not, as we have, compare lazy and eager learning modes. The two main differences with these approaches are: control rules represent knowledge to control individual decisions, while cases chain multiple decisions together allowing therefore a global control of the planning process; and control rules fi e only if their antecedents fully match, while cases allow partial matching.

Many of the inductive systems require many examples for learning complex defin tions, since they do not use prior knowledge that can guide the search of generalized hypothesis. Some new techniques have been developed that use prior knowledge, but they are still mainly used for learning domain theories (e.g., (Quinlan 1990; Muggleton 1992)), instead of learning control knowledge.

Similar work on lazy learning includes Lazy Explanation-Based Learning, LEBL (Tadepalli 1989) and Lazy Partial Evaluation, LPE (Clark and Holte 1992). While LEBL refine the knowledge by introducing exceptions, HAMLET modifie the control rules themselves by adding or removing their applicability conditions. Also, LEBL applies to games, while we use HAMLET for general task planning. Finally, LEBL does not consider plan quality. In turn, LPE learns from all search paths, following a more *eager* approach than HAMLET's *lazy* mode, and has been used in a linear problem solver (Prolog) to solve contraint satisfaction problems, instead of applying it to nonlinear planning.

Most of the planners that learn follow an eager deductive approach. They try to eagerly and correctly explain the problem solving choices from a single episode or from a static analysis of the domain defin tion. These speedup learning systems are usually applied to problem solvers with the linearity assumption, such as the ones applied to logic programming problem solvers (Quinlan 1990; Zelle and Mooney 1993; Muggleton 1992), special-purpose problem solvers (Langley 1983; Mitchell et al. 1983), or other general-purpose linear problem solvers (Etzioni 1993; Fikes et al. 1972; Leckie and Zukerman 1991; Minton 1988; Pérez and Etzioni 1992). These problem solvers are known to be incomplete and incapable of find ng optimal solutions (Rich 1983; Veloso 1989).

If we remove the linearity assumption, we are dealing with nonlinear problem solvers. In this article we show that nonlinear problem solving offers new learning opportunities where domain-dependent control knowledge can be used to further improve not only the problem solver's performance but also

the quality of the solutions produced. Moreover, eagerly constructing correct explanations of the nonlinear problem solver's successes and failures *from a single example* is computationally expensive, since it is difficu t to defin the right language for describing the relations among goals when making decisions. Also, even if those needed predicates are kept, goal regression leads in nonlinear planning to control knowledge that is either overly general or overly specifi . Some approaches to learning in nonlinear planning are (Estlin and Mooney 1995; Bhatnagar 1992; Laird et al. 1986; Leckie and Zukerman 1991; Kambhampati and Kedar 1991; Pérez and Carbonell 1994; Ruby and Kibler 1992; Veloso 1994b; Veloso et al. 1995). The main difference with our approach is the lazy aspects of HAMLET, and the way in which learned knowledge is represented. While improving problem solving eff ciency has been studied frequently, learning to improve solution quality has only been recently pursued by some researchers, including (Pérez and Carbonell 1994; Ruby and Kibler 1992). We differ from Pérez and Carbonell's work in that HAMLET performs inductive refinemen of the control rules, and in the way positive examples are generated. Ruby and Kibler's approach differs in the knowledge representation of the learned control knowledge, since it is a case-based learner.

7. Conclusions

We have described a learning approach, HAMLET, that lazily acquires successful and failure control patterns for a nonlinear problem solver. Within HAMLET, *lazily* means the combination of three lazy aspects:

- It learns rules that are not provenly correct by bounding the explanation of the problem solving successes to a reduced set of features that explain why a certain decision is the best one. The explanation does not consider the whole search tree, nor does it try to prove that it is correct. Hence, this is a *lazy* learning approach.
- It incrementally refine learned control rules. Since the rules might not be correct, they might fail to (optimally) solve future problems. HAMLET does not eagerly try to f nd those erroneous applications of the control rules, but lazily waits until it fi ds a negative example. Also, it refi es on demand the descriptions of the target concepts, using only a subset of the examples found.
- These rules are learned only at the decision points that override the default behavior of the problem solver, instead of at all the decision points (i.e., as is done by other learning mechanisms). This is again a *lazy* learning approach for determining which rules should be learned.

The combination of these first two *lazy* aspects results in a system that can solve problems more efficiently, achieving better solutions than the heuristic-based problem solver.[18] Also, this lazy aspect allows HAMLET to learn useful control rules in nonlinear planning, which is a complex task to perform by *eager* approaches: it is difficult to describe the complete extra domain theory that *eager* approaches require in nonlinear planning (Minton 1988; Katukam and Kambhampati 1994). Furthermore, the results show that the third *lazy* aspect has several advantages over an *eager* learner since it achieves *better* solutions *more efficiently*, requires fewer learning resources (i.e., learning time and memory), and it suffers less from the utility problem.

Finally, HAMLET has been tested in a variety of experiments involving complex planning problems. The empirical results support the effectiveness of HAMLET's learning approach, in terms of improvement in planning efficiency, in the quality of plans generated, and in its incremental convergence towards the correct knowledge. In summary, HAMLET's learning power comes most directly from its overall lazy learning approach.

Acknowledgements

This research for the first author was sponsored by Ministerio de Educación y Ciencia and Comunidad de Madrid. This research for the second author is sponsored by the Wright Laboratory, Aeronautical Systems Center, Air Force Materiel Command, USAF, and the Advanced Research Projects Agency (ARPA) under grant number F33615-93-1-1330. The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of Wright Laboratory or the U.S. Government. The authors would like to thank David Aha and the anonymous reviewers for their many useful comments.

References

Aha, D. W., Kibler, D. & Albert, M. K. (1991). Instance-based learning algorithms. *Machine Learning* **6**(1): 37–66.

Bhatnagar, N. (1992). Learning by incomplete explanations of failures in recursive domains. In *Proceedings of the Ninth International Conference on Machine Learning*, pp. 30–36, Aberdeen, Scotland: Morgan Kaufmann.

[18] As stated in Section 2, the base problem solver already incorporates many useful domain-independent heuristics. It would be easier to outperform a system that performs completely uninformed search.

Borrajo, D., Caraç a-Valente, J. P. & Morant, J. L. (1992a). Learning heuristics in planning. In *Proceedings of the Sixth International Conference on Systems Research, Informatics and Cybernetics*, pp. 43–49, Baden-Baden, Germany: The International Institute for Advanced Studies in Systems Research and Cybernetics.

Borrajo, D., Caraç a-Valente, J. P. & Pazos, J. (1992b). A knowledge compilation model for learning heuristics. In *Proceedings of the First Workshop on Knowledge Compilation*, Aberdeen, Scotland.

Borrajo, D. & Veloso, M. (1994). Incremental learning of control knowledge for nonlinear problem solving. In *Proceedings of the European Conference on Machine Learning*, pp. 64–82. Catania, Italy: Springer Verlag.

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

Carbonell, J. G., Knoblock, C. A. & Minton, S. (1990). Prodigy: An integrated architecture for planning and learning. In VanLehn, K. (ed.), *Architectures for Intelligence*, Erlbaum, Hillsdale, NJ. Also Technical Report CMU-CS-89-189.

Clark, P. & Holte, R. (1992). Lazy partial evaluation: An integration of explanation-based generalisation and partial evaluation. In *Proceedings of the Ninth International Conference on Machine Learning*, pp. 82–91, Aberdeen, Scotland: Morgan Kaufmann.

Cohen, W. W. (1990). Learning approximate control rules of high utility. In *Proceedings of the Seventh International Conference on Machine Learning*, pp. 268–276, Austin, TX: Morgan Kaufmann.

DeJong, G. F. & Mooney, R. (1986). Explanation-based learning: An alternative view. *Machine Learning* **1**(2): 145–176.

Doorenbos, R. B. & Veloso, M. M. (1993). Knowledge organization and the utility problem. In *Proceedings of the Third International Workshop on Knowledge Compilation and Speedup Learning*, pp. 28–34, Amherst, MA.

Estlin, T. A. & Mooney, R. (1995). Hybrid learning of search control for partial order planning. In *New Directions in AI Planning*, pp. 115–128. IOS Press.

Etzioni, O. (1993). Acquiring search-control knowledge via static analysis. *Artif cial Intelligence* **62**(2): 255–301.

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

Fikes, R. E., Hart, P. E. & Nilsson, N. J. (1972). Learning and executing generalized robot plans. *Artif cial Intelligence* **3**: 251–288.

Hammond, K. J. (1989). *Case-based Planning: Viewing Planning as a Memory Task*. New York, NY: Academic Press.

Hanks, S. & Weld, D. (1995). A domain-independent algorithm for plan adaptation. *Journal of Artif cial Intelligence Research* **2**: 319–360.

Kambhampati, S. (1989). *Flexible Reuse and Modifica ion in Hierarchical Planning: A Validation Structure Based Approach*. PhD thesis, Computer Vision Laboratory, Center for Automation Research, College Park, MD: University of Maryland.

Kambhampati, S. & Kedar, S. (1991). Explanation based generalization of partially ordered plans. In *Proceedings of the Ninth National Conference on Artificia Intelligence*, pp. 679–685. Anaheim, CA: AAAI Press.

Katukam, S. & Kambhampati, S. (1994). Learning explanation-based search control rules for partial order planning. In *Proceedings of the Twelfth National Conference on Artificia Intelligence*, pp. 582–587. Seattle, WA: AAAI Press.

Kettler, B. P., Hendler, J. A., Andersen, A. W. & Evett, M. P. (1994). Massively parallel support for case-based planning. *IEEE Expert* **2**: 8–14.

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

Langley, P. (1983). Learning effective search heuristics. In *Proceedings of the Eighth International Joint Conference on Artificia Intelligence*, pp. 419–421, Los Altos, CA: Morgan Kaufmann.

Leckie, C. & Zukerman, I. (1991). Learning search control rules for planning: An inductive approach. In *Proceedings of the Eighth International Workshop on Machine Learning*, pp. 422–426, Evanston, IL: Morgan Kaufmann.

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

Minton, S., Knoblock, C. A., Kuokka, D. R., Gil, Y., Joseph, R. L. & Carbonell, J. G. (1989). PRODIGY 2.0: The manual and tutorial. Technical Report CMU-CS-89-146, School of Computer Science, Carnegie Mellon University.

Mitchell, T. M., Keller, R. M. & Kedar-Cabelli, S. T. (1986). Explanation-based generalization: A unifying view. *Machine Learning* **1**: 47–80.

Mitchell, T. M., Utgoff, P. E. & Banerji, R. B. (1983). Learning by experimentation: Acquiring and ref ning problem-solving heuristics. In R. S. Michalski, J. G. Carbonell & T. Mitchell (eds.), *Machine Learning, An Artif cial Intelligence Approach*. Palo Alto, CA: Tioga Press.

Muggleton, S. (1992). *Inductive Logic Programming*. London: Academic Press Limited.

Pérez, M. A. & Carbonell, J. G. (1994). Control knowledge to improve plan quality. In *Proceedings of the Second International Conference on AI Planning Systems*, pp. 323–328, Chicago, IL: AAAI Press.

Pérez, M. A. & Etzioni, O. (1992). DYNAMIC: A new role for training problems in EBL. In *Proceedings of the Ninth International Conference on Machine Learning*, pp. 367–372, Aberdeen, Scotland: Morgan Kaufmann.

Porter, B. W., Bareiss, R. & Holte, R. (1990). Knowledge acquisition and heuristic classif cation in weak-theory domains. *Artif cial Intelligence* **45**: 229–263.

Quinlan, J. R. (1990). Learning logic def nitions from relations. *Machine Learning* **5**: 239–266.

Rich, E. (1983). *Artif cial Intelligence*. McGraw-Hill, Inc.

Ruby, D. & Kibler, D. (1992). Learning episodes for optimization. In *Proceedings of the Ninth International Conference on Machine Learning*, pp. 379–384, Aberdeen, Scotland: Morgan Kaufmann.

Stanf ll, C. & Waltz, D. (1986). Toward memory-based reasoning. *Communications of the Association for Computing Machinery* **29**: 1213–1228.

Stone, P., Veloso, M. & Blythe, J. (1994). The need for different domain-independent heuristics. In *Proceedings of the Second International Conference on AI Planning Systems*, pp. 164–169, Chicago, IL: AAAI Press.

Tadepalli, P. (1989). Lazy explanation-based learning: A solution to the intractable theory problem. In *Proceedings of the Eleventh International Joint Conference on Artif cial Intelligence*, pp. 694–700, San Mateo, CA: Morgan Kaufmann.

Valiant, L. (1984). A theory of the learnable. *Communications of the ACM* **27**(11): 1134–1142.

Veloso, M. & Blythe, J. (1994). Linkability: Examining causal link commitments in partial-order planning. In *Proceedings of the Second International Conference on AI Planning Systems*, pp. 170–175, Chicago, IL: AAAI Press.

Veloso, M. & Borrajo, D. (1994). Learning strategy knowledge incrementally. In *Proceedings of the Sixth IEEE International Conference on Tools with Artificia Intelligence*, pp. 484–490, New Orleans, LO: IEEE Computer Society Press.

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

Veloso, M. M. (1989). Nonlinear problem solving using intelligent causal-commitment. Technical Report CMU-CS-89-210, School of Computer Science, Carnegie Mellon University.

Veloso, M. M. (1994a). Flexible strategy learning: Analogical replay of problem solving episodes. In *Proceedings of the Twelfth National Conference on Artificia Intelligence*, Seattle, WA: AAAI Press.

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

Waldinger, R. (1981). Achieving several goals simultaneously. In Nilsson, N. J. & Webber, B. (eds.), *Readings in Artif cial Intelligence*, pp. 250–271. Los Altos, CA: Morgan Kaufmann.

Zelle, J. & Mooney, R. (1993). Combining FOIL and EBG to speed-up logic programs. In *Proceedings of the Thirteenth International Joint Conference on Artificia Intelligence*, pp. 1106–1113, Chambery, France: Morgan Kaufmann.