Incremental Learning of Control Knowledge for Nonlinear Problem Solving

Daniel Borrajo[1] and Manuela Veloso[2]

[1] Carnegie Mellon University, School of Computer Science, Pittsburgh, USA, on leave from Universidad Politécnica de Madrid, Facultad de Informática, Departamento de Inteligencia Artificial, Madrid, Spain
[2] Carnegie Mellon University, Department of Computer Science, Pittsburgh, PA 15213, USA

Abstract. In this paper we advocate a learning method where a deductive and an inductive strategies are combined to efficiently learn control knowledge. The approach consists of initially bounding the explanation to a predetermined set of problem solving features. Since there is no proof that the set is sufficient to capture the correct and complete explanation for the decisions, the control rules acquired are then refined, if and when applied incorrectly to new examples. The method is especially significant as it applies directly to nonlinear problem solving, where the search space is complete. We present HAMLET, a system where we implemented this learning method, within the context of the PRODIGY architecture. HAMLET learns control rules for individual decisions corresponding to new learning opportunities offered by the nonlinear problem solver that go beyond the linear one. These opportunities involve, among other issues, completeness, quality of plans, and opportunistic decision making. Finally, we show empirical results illustrating HAMLET's learning performance.

1 Introduction

Problem solving 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 improving the planning performance. It has been the focus of attention of several researchers, present authors included, to learn control knowledge, i.e., automate the acquisition process of these guiding heuristics.

One approach to learning control knowledge from a problem solving trace consists of generating explanations for the individual decisions made during the search process. These explanations become control strategies that are used in future situations to prune the search space [16]. There is also work done on doing the generation of control rules without problem solving episodes, by statically looking at the domain description [8]. However, these strong deductive approaches invest a substantial explanation effort to produce correct control strategies from a single problem solving trace. Alternatively, inductive approaches acquire correct learned knowledge by observing a large set of examples [20, 26].

In this paper, we present HAMLET,[3] a system that learns control knowledge incremental and inductively. HAMLET uses an initial deductive phase, where it generates a bounded explanation of the problem solving episode. Upon experiencing each new problem solving episode, HAMLET refines its control knowledge incrementally acquiring increasingly correct control knowledge.

The paper is organized in nine sections. Section 2 overviews the complete architecture of HAMLET, and PRODIGY as the substrate problem solver. Section 3, 4 and 5 discuss the three learning phases, namely the generation of the bounded explanation from the problem solving search tree, the generalization of the rules by induction, and the refinement strategy driven by encountered negative examples. Section 6 presents an example that illustrates the execution of the learning algorithm on a problem from a logistics transportation domain. Section 7 shows empirical results from different domains. Section 8 compares our approach with previous related work. Finally section 9 draws conclusions.

2 Overview of the Architecture

HAMLET learns effectively control knowledge from a problem solving experience. This work is developed within the nonlinear problem solver [22, 5] of the PRODIGY architecture [6]. In this section we provide a description of PRODIGY's nonlinear planner and we also present HAMLET's architectural components.

2.1 The Substrate Problem Solver

The nonlinear problem solver in PRODIGY follows a means-ends analysis backward chaining search procedure reasoning about multiple goals and multiple alternative operators relevant to the goals. Figure 1 sketches the problem solver's algorithm. The inputs to the procedure are the set of operators specifying the task knowledge and a problem specified in terms of an initial configuration of the world and a set of goals to be achieved.

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 (steps 2-4); the *operator* to choose to achieve a particular goal; the *bindings* to choose in order to instantiate the chosen operator (step 4 combines the operator and bindings selection); *apply* an operator whose preconditions are satisfied (step 5) or continue *subgoaling* on a still unachieved goal (step 3-4). Dynamic goal selection from the set of pending goals enables the planner to interleave plans, exploiting common subgoals and addressing issues of resource contention. Search control knowledge may be applied at all the above decision points: which relevant operator to select from the possible available ones, which goal or subgoal to address next, whether to reduce a new subgoal or to apply a previously selected operator whose preconditions are satisfied, or what objects in the state to use as bindings of the typed variables in the operators. Decisions at all these

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

```
1. Check if the goal statement is true in the current state, or there is a reason
   to suspend the current search path. If yes, then either return the final plan or
   backtrack.
2. Compute the set of pending goals $\mathcal{G}$, and the set of applicable operators $\mathcal{A}$.
3. Choose a goal $G$ from $\mathcal{G}$ or select an operator $A$ from $\mathcal{A}$ that is directly applicable.
4. If $G$ has been chosen, then
   – expand goal $G$, i.e., get the set $\mathcal{O}$ of relevant instantiated operators for 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.
```

Fig. 1. A skeleton of PRODIGY's nonlinear problem solving algorithm.

choices are taken based on user-given or learned control knowledge to guide the search and convert it into an intelligent commitment search strategy [22]. Control knowledge guides the search process and helps to prune the exponential search space. Previous work in the linear planner of PRODIGY uses explanation-based learning techniques [16] to extract from a problem solving trace the explanation chain responsible for a success or failure and compile search control rules therefrom. Similar efforts within the linear planner of PRODIGY were done to learn control rules from partially evaluating the domain theory [8, 19].

The paper presents instead our on-going work in learning individual control rules for the nonlinear problem solver of PRODIGY [4]. We have identified several challenging problems in extending directly the previous explanation-based algorithms developed for the linear planner to the nonlinear one, since in nonlinear planning we face learning opportunities, including issues of plan quality, and opportunistic decision making. Our work applies directly to nonlinear problem solving which trivially encompasses linear problem solving. In our nonlinear problem solving framework, HAMLET learns control rules for individual decisions compiling the conditions under which the rules are to be transferred to individual decision steps in other problems. Alternative learning approaches in nonlinear planning include learning complete generalized plans as in [12], or developing a case-based learning method that provides cases as a form of global strategic knowledge [24], as discussed in the related work section.

2.2 HAMLET's Components

HAMLET has three main modules: the Bounded-Explanation learner, the Inducer and the Refiner. The Bounded-Explanation module learns control rules from the search tree. These rules are either over-specific or over-general, so they should be refined. The Induction module solves the problem of over-specificity by making them more general from more positive examples. The Refinement module attacks the over-generality by finding situations in which the learned rules were used wrongly. HAMLET gradually learns and refines so that, at the end, it converges to a concise set of correct control rules.

Figure 2 shows HAMLET's modules connected to PRODIGY. Figure 3 presents the procedure schematically, where ST and ST' are search trees, 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 learned induced from L' and L. We explain in detail each one of HAMLET's components in the next sections.

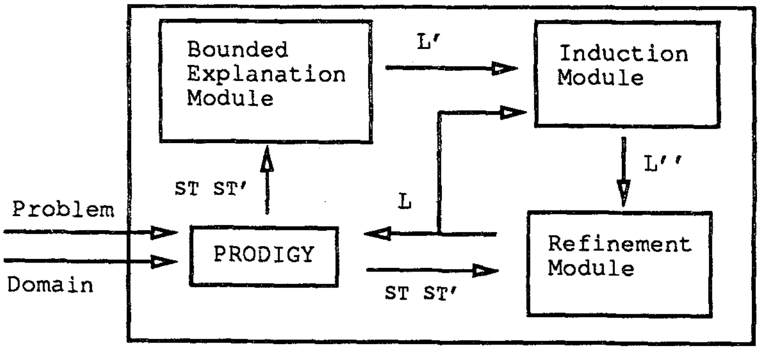

Fig. 2. HAMLET's high level architecture.

```
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.
Initially L is empty.
For all P in training problems
    ST = Result of solving P without any rules.
    ST' = Result of solving P with current set of rules L.
    L' = Bounded-Explanation(ST, ST')
    L"= Induce(L,L')
    If needs-refinement-p(ST, ST')
    Then L=Refine(ST, ST',L")
```

Fig. 3. A high level description of HAMLET's learning algorithm.

3 Bounded Explanation

In this initial phase, HAMLET learns control rules directly by loosely explaining the problem solving search tree. The algorithm relies on three main parts: labeling and credit assignment on the search tree; the actual generation of the control rules; and the generalization of the control rules.

3.1 Labeling the Search Tree and Credit Assignment

When solving a problem, the problem solver generates a search tree. The domain theory implicitly defines a subgoaling structure that links goals with the operators that achieve those goals. In a linear planner, the search tree reproduces exactly this structure, since interleaving of goals and subgoals at different search spaces is not allowed. However, in nonlinear problem solving, there is a variety of different interleaved ways to traverse the subgoaling structure which are captured in the search tree. This leads to a very large search space so that there is no tractable way to generate a correct explanation for the decisions made from a unique problem solving experience.

The labeling algorithm of HAMLET traverses the search tree top-down to label first the leaf nodes. It assigns three kinds of labels to the leaf nodes: *success*, if it was a solution path; *failure*, if it was a failed path; and *unknown*, if the planner did not expand the node. After labeling the leaf nodes, it backs up the values up to the root of the search tree. Figure 4 summarizes this labeling strategy. The credit assignment is done at the same time as the labeling, and it consists of identifying the decisions for which learning will occur.

At each decision choice to be learned, HAMLET has access to information on the problem state and meta-level planning state, which is explicitly maintained in the search tree structure. Examples of meta-level knowledge at each node, available to the learning procedures, include the goals that had not been achieved, the goal the planner is working on, and in general the alternatives known to have failed or succeeded. This information is used by the generation module to create the pre- and post-conditions of the control rules.

The parameter *eagerp* controls the situations from which control rules are generated. If *eagerp* is *true*, HAMLET will learn a select rule,[4] whenever a node has a success child. If *eagerp* is *false*, HAMLET follows a conservative learning mode. A rule is then learned only if all of its children are labeled success or failure and there is at least one child labeled failure. These two different modes correspond to different levels of learning eagerness.

The parameter *optimal-learning-p* allows to learn only from the best solution found,[5] where we can incorporate different quality criteria. If *optimal-learning-p* is *true*, HAMLET delays learning until it traverses the complete tree and finds

[4] A select rule, when applied, selects an alternative and rejects all others for which there is not a select control rule.

[5] We consider currently best as shortest solution and less number of nodes. We will extend this criterium according to the results of [18].

```
procedure LABEL (node eagerp)
  for all successors of node do
    LABEL (successor eagerp)
  case of
    null(successors):
      case of
        solution-path: label node as success.
        failed-path: label node as failure.
        untried: label node as unknown.
    there is at least one unknown successor:
      if eagerp AND there are success children
        then if optimal-learning-p
               then store the "best" successor
               else LEARN the "best" successor
             label node as success
        else label node as unknown.
    there are only success and failure:
      if optimal-learning-p
        then store the "best" successor
        else LEARN the "best" successor
      label node as success.
    there are only failures
      label node as failure.
    there are only successes
      label node as success.
```

Fig. 4. A skeleton of the labeling and credit assignment algorithm

the best solution. In that case, after labeling, it descends only through the best solution path, learning from every decision according to the selected level of eagerness.

This algorithm builds upon some early previous work on learning and problem solving, including [14, 17]. We extend these pioneering methods in several dimensions, as discussed in section 8.

3.2 Generation of Control Rules

HAMLET proceeds to generate each control rule by acquiring its corresponding pre- and postconditions. The preconditions of the control rule need to establish the relevant conditions under which the decision was made and also define the situations under which the rule can be re-applied. The appropriate set of features that we consider in our bounded explanation technique has evolved from previous work of the first author [3]. Although there is no guarantee that this set of features is a sufficient set, there have been a number of iterations in the design of the set, to generate our confidence on it. Furthermore the empirical experiments confirm that the set is appropriate and the induction and refinement phases increase its application efficiency.

HAMLET learns four kinds of control rules: select a goal from the set of pending goals, select an operator to achieve a goal, select bindings for the chosen operators, and decide whether to apply an operator when its preconditions are met in the current state of the search, or continue subgoaling selecting a goal from the set of unachieved goals. Each rule corresponds to a generalized target concept. The target concepts are each one of the possible decisions to be made attached to some of the preconditions required to make them. For instance, for an operator decision, a target concept might be *select operator <op> to achieve the goal <goal>*. The number of target concepts of a given domain is $O + P + 2O\sum_{i=1}^{O} p(O_i)$, where O is the total number of operator schemas in the domain, P is the number of predicates of the domain, and $p(O_i)$ is the number of postconditions of the operator O_i.[6] HAMLET generates a set of rules for each target concept, each one with a conjunctive set of preconditions. This representation can be viewed as the disjunction of conjunctive rules, as we can learn several rules for the same target concept. This is equivalent, therefore, to learning a DNF description of the target concept.

Each kind of control rule has a template for describing its preconditions. The templates share a set of common features for all kinds of control rules, but each one has certain local features. Examples of common features, which become meta-predicates of the control language, are:

- True-in-state <assertion>: tests whether the <assertion> is true in the current state of the search for the solution.
- Other-goals <list of goals>: test whether any of the goals in the <list of goals> is a pending goal in the current node of the search tree.
- Prior-goal <goal>: tests whether <goal> is the first goal of the conceptual path of the node the planner is in.

Similarly, examples of the other features are:

- Current-goal <goal>: tests whether the <goal> is the one that the planner is trying to achieve.
- Candidate-applicable-op <operator>: tests whether the <operator> is applicable in the current state.

The preconditions of the control rules are created using the information on the state and the meta-level state linked to the corresponding decision node in the search tree. The postconditions are the decisions to be made, such as (*select operator unstack*), or `(select goal (on <x> <y>))`.[7] See section 6 for an

[6] This number is the sum of the number of target concepts for each kind of learned control rule. For instance, the *select operator* kind of control rule has two variables: the operator, and a goal that can be achieved by that operator. Therefore, in this case, the number of target concepts for that kind is $O\sum_{i=1}^{O} p(O_i)$.

[7] Variables are represented in brackets.

example of a learned control rule. After the rule has been created, it is parameterized, imposing the condition that two variables cannot be bound to the same value. After the induction phase, the rule is checked against the possible negative examples of the target concept. At the beginning, the set of negative examples for each target concept is empty. Section 5 explains the refinement module including how to identify negative examples. Section 6 shows an illustrative example of the learning process.

Figure 5 shows an example of a learned control rule in the blocksworld domain [10] learned after PRODIGY solves the Sussman's anomaly [21]. The control rule allows the problem solver to select the goal of holding a block, *block1*, over the goal of having another block, *block3*, on top of *block1*, given that there are three blocks on the table, and the goal of holding *block1* was created as a subgoal of the goal of having *block1* on top of another block, *block2*. This control rule allows PRODIGY later to solve similar nonlinear problems more efficiently than before the rule is learned.

```
(control-rule SELECT-ON-1
 (if ((candidate-goal (HOLDING <BLOCK1>))
      (prior-goal (ON <BLOCK1> <BLOCK2>))
      (true-in-state (ON-TABLE <BLOCK1>))
      (true-in-state (ON-TABLE <BLOCK2>))
      (true-in-state (ON-TABLE <BLOCK3>))
      (other-goals ((ON <BLOCK3> <BLOCK1>)))))
 (then SELECT GOALS (HOLDING <BLOCK1>)))
```

Fig. 5. Rule learned in the blocksworld for selecting the goal *holding* over the goal *on* for interfering block configurations.

4 Inductive Generalization

The rules generated by the bounded explanation method may be over-specific, as also analyzed in [9]. Particularly, the rules may be over-specific in the aspects explained below. The more over-specific the rules are the lower the transfer to other problems.[8] We follow up the deductive phase with a generalization algorithm that inductively modifies the rules based on new examples, reducing the set of preconditions. The rules may become over-general but their transfer potential increases. We have devised ways of inducing over the following aspects of the learned knowledge, that practically cover all the features in the preconditions.

- *State:* Most of the rules are over-specific because they keep many irrelevant features from the state.

[8] Note that in order to apply a control rule, we require that the rule totally matches a decision making situation, i.e., all the preconditions need to be satisfied.

- *Subgoaling structure:* By relaxing the subgoaling links, for example as captured by the *prior-goal* meta-predicate, since the same goal can be generated as a subgoal of many different goals (see section 3).
- *Interacting goals:* Identifying the correct subset of the set of pending goals that affect a particular decision extending the learning scope also to quality decisions.
- *Type hierarchy:* 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.
- *Operator types:* Further learning from an operator hierarchy to enlarge the scope of the generalization procedure.

The inductive component of HAMLET currently considers the following inductive operators relative to one or more of the above aspects:

- *Preserve main preconditions:* HAMLET is able to remove "unimportant" preconditions that are found not to affect the validity of the control rule. It keeps the *main preconditions*, i.e., the preconditions that have variables directly related to the learned decision.
- *Delete rules that subsume others:* A rule subsumes another rule of the same target concept if there is a substitution that makes its preconditions a superset of the other.
- *Intersection of preconditions:* From two rules, R_1 and R_2 of the same target concept, create a new rule with preconditions the intersection of the preconditions of R_1 and R_2 (when the intersection is not empty).
- *Refinement of subgoaling dependencies:* If there are two rules, R_1 and R_2 sharing some preconditions, but their prior goals are different, they are merged into a new rule that tests for the presence of any of the prior goals of the two rules.
- *Refinement of goal dependencies:* Similar to the previous one, but, in this case, it refers to the meta predicate *other-goals.*
- *Relaxing the subgoaling dependency:* If there is no evidence that the prior goal is needed, it gets deleted until needed.
- *Find common superclass:* When two rules can be unified by two variables that belong to subclasses of a common class (except for the root class), this operator generalizes the variables to the common class. We implemented previously a variation of this technique applied to the parameterization procedure of a single rule [2].

HAMLET tries to find an intersection of two rules using all these operators. If it finds a correct intersection that does not cover any previous negative example, a new rule is created, and the two previous ones are deleted. However HAMLET can backtrack to that learning point, and try an alternative way of intersecting the rules. This should not be considered as *plain* backtracking. When HAMLET "backtracks" to that point, it accumulated more information than when the alternative was generated, since it has found a new negative example, and it can now do better generalizations. The inductive phase significantly improves the transfer potential of the rules as it generalizes their application conditions. The inductive operators are triggered by positive examples, but also take into account the negative examples found so far as we describe in the next section.

5 Refinement

After the two previous learning phases, HAMLET may have produced over-general rules in special situations (due to the inductive operators, e.g., intersection). An over-general rule is beneficial for our inductive learning strategy as it may provide negative examples of its application. There are two main issues to be addressed: how to detect a negative example, and how to refine the learned knowledge according to it (making the rule more specific).

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.*

Once identified a negative example for a certain rule, the negative example is processed against all the current rules for the same target concept. Figure 6 shortly describes the procedures used for the refinement.

In a nutshell, the refinement module will try to relax the effect of the inductive operators by adding the tests removed in the inductive steps. The goal is to find a larger set of literals that covers the positive examples, but not the negative examples. It first checks the type of the control rule. The types are deduced, induced, and refined. Deduced are the rules generated by the Bounded-Explanation module. Induced are the ones that were generated by inducing from two rules. Refined are the ones generated by refining an existing rule, because it covered negative examples. The procedure *add-new-preconds* (not shown) does the following: for each precondition of the whole rule, having in mind even the ones that were not main preconditions, adds that precondition to the preconditions, and tests whether it covers the negative examples or not. If not, then returns the new preconditions. The procedure *find-new-intersection* (not shown) searches for other bindings of the variables of the rules from where it generated the rule, so that the new bindings substituted on the preconditions of one of those rules do not cover the negative examples.

One of the key things of any inductive method is to capture the right features in the learned description of a concept. In respect to this issue, the current version of HAMLET gets rid of irrelevant features if it learns positive examples of a target concept that do not have those features in common, or it finds negative examples of the target concept where those features are also present. In those cases, the eager inductive and refinement modules will remove these features. To speed up the convergence of the learning, we are currently introducing more informed elaborated ways of removing and adding features from the description of the target concept, such as information gain measures, similarly to [20]. These

```
procedure refine-rule (rule)
  if covers-negative-examples-p(rule)
    then
     if type(rule)=deduced
       then refine-deduced-rule(rule)
       else for all rule1 in rules(target-concept(rule))
               refine-induced-rule(rule1)
    else if deletedp(rule)
           then undelete-rule(rule)

procedure refine-deduced-rule (rule)
  preconditions=add-new-preconds(rule)
  if preconditions
    then create-rule(preconditions,postconditions(rule))
  delete-rule(rule)

procedure refine-induced-rule (rule)
  rule1=originating-rule1(rule)
  rule2=originating-rule2(rule)
  preconditions=find-new-intersection(rule1,rule2)
  if preconditions
    then create-rule(preconditions,postconditions(rule))
    else refine-rule(rule1)
         refine-rule(rule2)
  delete-rule(rule)
```

Fig. 6. High level description of the algorithm for the refinement of control rules.

methods have been tested on an analysis of a complete set of examples, and we are now exploring extending them to our incremental learning procedures.

The hill climbing performance of our global learning algorithm will approach the ultimately correct control knowledge by converging gradually closer from both over-specific and over-general rule sets. Our learning algorithm reasons about and converges from points in the generalization space as it is prohibitively costly to maintain both the specific and general sets as in the version space method [17].

6 Illustrative Example

We show now an example of the learning method applied to a logistics domain where we illustrate the phases of the generation of the control rules and their inductive refinement.[9] In this domain, packages are to be moved among different cities. Packages are carried within the same city in trucks and across cities in

[9] This domain was first introduced in [23].

airplanes. At each city, there are several locations, e.g. post offices and airports. This transportation domain represents a considerable scale up in length of the solution, size of the search space, and other difficult learning issues, such as non-linearity, un-optimality of solutions, and a large number of planning alternatives.

Consider the following problem solving situation illustrated in Figure 7. There are two cities, *city1* and *city2*, with one post-office each, respectively *post-office1* and *post-office2*, and with one airport each, namely *airport1* and *airport2*. Initially, at *post-office1*, there are two objects, *object1* and *object3*, and two trucks, *truck1* and *truck3*. At *airport1* there is an airplane, *airplane1*, and another object, *object2*. At *city2*, there is only one truck, *truck2*, at the city airport, *airport2*. There are two goals: *object1* must be at *post-office2*, and *airplane1* at *airport2*. This problem is interesting because both the object and the airplane need to be moved to a different city. HAMLET will learn, among other things, that the object should be loaded into the airplane (or any other needed carrier) before the airplane moves.

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

- *(load-truck object1 truck1 post-office1)*,
- *(drive-truck truck1 post-office1 airport1)*,
- *(unload-truck object1 truck1 airport1)*,
- *(load-airplane object1 airplane1 airport1)*,
- *(fly-airplane airplane1 airport1 airport2)*,
- *(unload-airplane object1 airplane1 airport2)*,
- *(load-truck object1 truck2 airport2)*,
- *(drive-truck truck2 airport2 post-office2)*,
- *(unload-truck object1 truck2 post-office2)*.

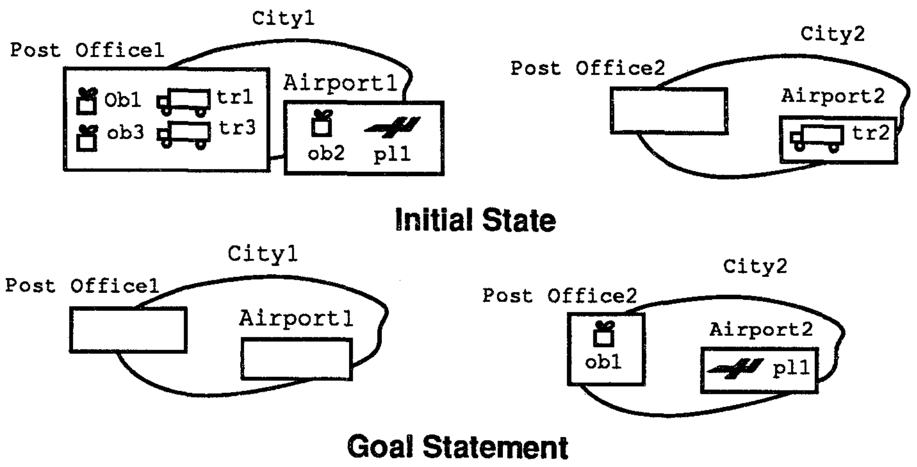

Fig. 7. An illustrative example - initial state and goal statement

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.

HAMLET labels and assigns credit to decisions made in the search tree generated by PRODIGY. The rule in Figure 8 is learned at one of the decisions made, namely when PRODIGY finds that it should delay moving the carriers until the object is loaded.[10] The rule says that the planner should plan first to achieve *(inside-truck object1 truck1)* before moving the carriers, including *truck1* and *airplane1*. This is a very effective control rule, since if the problem solver works first on any of the other two goals, it will arrive to an un-optimal solution, where carriers would have to do trips not needed.

```
(control-rule SELECT-INSIDE-TRUCK-1
 (if ((target-goal (INSIDE-TRUCK <OBJECT1> <TRUCK1>))
      (prior-goal (AT-OBJ <OBJECT1> <POST-OFFICE2>)) **
      (true-in-state (AT-TRUCK <TRUCK1> <POST-OFFICE1>))
      (true-in-state (AT-TRUCK <TRUCK2> <AIRPORT2>))
      (true-in-state (AT-TRUCK <TRUCK3> <POST-OFFICE1>)) *
      (true-in-state (AT-AIRPLANE <AIRPLANE1> <AIRPORT1>))
      (true-in-state (AT-OBJ <OBJECT1> <POST-OFFICE1>))
      (true-in-state (AT-OBJ <OBJECT2> <AIRPORT1>))
      (true-in-state (AT-OBJ <OBJECT3> <POST-OFFICE1>))*
      (true-in-state (SAME-CITY <AIRPORT1> <POST-OFFICE1>))
      (true-in-state (SAME-CITY <AIRPORT2> <POST-OFFICE2>))
      (other-goals ((AT-TRUCK <TRUCK1> <AIRPORT1>)
                    (AT-AIRPLANE <AIRPLANE1> <AIRPORT2>)))))
 (then SELECT GOALS (INSIDE-TRUCK <OBJECT1> <TRUCK1>)))
```

Fig. 8. Rule learned by HAMLET after applying the Bounded-Explanation.

This example, though simple,[11] shows a learning opportunity that is needed in order to deal with new problems effectively and with quality. This learning possibility is not encountered by a linear problem solver who handles multiple goals independently. It would also not be compiled as a local choice by other learning methods applied to nonlinear planning [12, 23].

The preconditions are the initial bounded explanation of the problem solving decision. This rule is over-specific, since, among other things, it records the positions of other trucks available and another object present at the post office, which turn out to be irrelevant features for this instantiated target concept. The operator *Preserve Main Preconditions* removes the preconditions that we marked with

[10] 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 refer to general variables.

[11] In our extensive empirical tests we ran problems of much greater complexity. Section 7 shows the results of some of these tests.

a star (*) in the description above. Note that there are still irrelevant features kept, such as *truck2* and *object2*, because they are at a location directly related to the goals rejected in the decision, namely *airport2*. The operator *Relaxing the Subgoaling Dependency* removes the precondition on the prior-goal that we marked with a double star (**).

Now suppose that we encounter a new problem where the goal is the same but there are no additional trucks or objects, and there is an additional airplane. Figure 9 shows a rule that HAMLET would learn in this case, after applying the same operators as before.

```
(control-rule SELECT-INSIDE-TRUCK-2
 (if ((target-goal (INSIDE-TRUCK <OBJECT1> <TRUCK1>))
      (true-in-state (AT-TRUCK <TRUCK1> <POST-OFFICE1>))
      (true-in-state (AT-AIRPLANE <AIRPLANE1> <AIRPORT1>))
      (true-in-state (AT-AIRPLANE <AIRPLANE2> <AIRPORT1>))*
      (true-in-state (AT-OBJ <OBJECT1> <POST-OFFICE1>))
      (true-in-state (SAME-CITY <AIRPORT1> <POST-OFFICE1>))
      (other-goals ((AT-TRUCK <TRUCK1> <AIRPORT1>)))))
 (then SELECT GOALS (INSIDE-TRUCK <OBJECT1> <TRUCK1>)))
```

Fig. 9. Rule learned by HAMLET in a second problem. It will help HAMLET generalize the rule in Figure 8.

The inductive method combines these rules with the operator *Intersection of Preconditions*, as they refer to the same decision. It also merges the goals in the *other-goals* meta-predicate. HAMLET generates a new rule, which consists of the intersection of the rules, being, therefore, more general than the initial two rules. Figure 10 shows the resulting rule.

```
(control-rule SELECT-INSIDE-TRUCK-3
 (if ((target-goal (INSIDE-TRUCK <OBJECT1> <TRUCK1>))
      (true-in-state (AT-TRUCK <TRUCK1> <POST-OFFICE1>))
      (true-in-state (AT-AIRPLANE <AIRPLANE1> <AIRPORT1>))
      (true-in-state (AT-OBJ <OBJECT1> <POST-OFFICE1>))
      (true-in-state (SAME-CITY <AIRPORT1> <POST-OFFICE1>))
      (other-goals ((AT-TRUCK <TRUCK1> <AIRPORT1>)
                    (AT-AIRPLANE <AIRPLANE1> <AIRPORT2>)))))
 (then SELECT GOALS (INSIDE-TRUCK <OBJECT1> <TRUCK2>)))
```

Fig. 10. Rule induced from the rules in Figures 8 and 9.

The induced rule is "almost" fully correct and it becomes "completely" correct after a couple of more training situations. Irrelevant features are removed with more positive examples, while important features are captured by the refinement with negative examples. HAMLET proceeds in this hill-climbing way searching the hypotheses space converging to the set of correct rules.[12] It induces and generalizes upon experiencing positive examples and it refines the learned control rules, when negative examples are found.

7 Empirical Results

We have been performing extensive empirical experiments in several domains. We report here results from the logistics transportation domain [23] and the blocksworld domain (where we experience nonlinear learning situations, similar to the Sussman's anomaly [21]). The results illustrate our main claims about the effectiveness of the combined deductive and inductive methods. The nonlinear version of PRODIGY we are using has embedded several domain-independent search heuristics [1]. Therefore, the substrate problem solver has some underlying "intelligence" that makes it difficult for a learning mechanism to outperform it by large. However, as we will show, HAMLET is able to perform considerably better.

In the blocksworld, we used a set of 100 problems randomly generated from which HAMLET learns the control rules. It learned 14 control rules. These rules were applied to a test set of 200 randomly generated problems. We varied the time bound given to PRODIGY to solve the problems from 100 to 300 seconds obtaining similar results for the different time bounds. Figure 11 shows the results obtained with 100 seconds of time bound. The unsolved problems are accounted for in the running time by a term equal to the running time bound.

In the logistics domain, we performed a training phase of 350 problems, 300 one-goal problems and 50 two-goal problems. We used the resulting learned 22 control rules in testing with three test sets of different complexity in terms of initial state and number of goals. We again varied the time bounds, from 150 to 300 seconds, obtaining similar results to the ones shown in Figure 11 for a time of bound of 150 seconds.

Domain	Number of Problems per Test Set	Unsolved problems		Running time (sec)		Nodes explored	
		without rules	with rules	without rules	with rules	without rules	with rules
Blocksworld	200	43	19	5402	2872	34065	9513
Logistics	300	57	25	9890	5615	13443	7798

Fig. 11. Table that reflects the performance of HAMLET in two different domains: blocksworld and a logistics transportation domain.

[12] Since it is an incremental inductive system, the performance of HAMLET depends on the order of the examples given.

The main conclusion from the table is that the number of unsolved problems drops from 21.5% to 9.5% in the blocksworld, and from 19% to 8.3% in the logistics. This corresponds to a considerable increase in the solvability horizon of the problem solver when using the rules. Also, since the matcher for the control rules is not using any optimum retrieving and organization algorithm [7], the time spent matching the rules represents the usual utility problem. The results shown in the table are especially relevant as the use of the learned set of rules outperformed the base-level problem solver even with the rudimentary matcher.

8 Related Work

There are several dimensions along which we can compare this work with previous approaches, including the representation, the granularity, the correctness of the control knowledge, and the essence of the substrate problem solving algorithm.

With respect to the representation, control guidance to prune the problem solving search space has been introduced as additional preconditions of the operators of the domain in simple one-goal planning situations, such as in [14, 17]. In our framework, control knowledge is explicitly distinct from the set of operators describing the domain knowledge because it introduces knowledge about the various problem solving decisions, such as selecting which goal/subgoal to address next, which operator to apply, what bindings to select for the operator or where to backtrack in case of failure. This clear division between the declarative domain knowledge, i.e., the operators, and the more procedural control knowledge simplifies both the initial specification of a domain and the automated acquisition, i.e., the learning of the control knowledge.

Learned control knowledge can be local or global. Local control knowledge is used on each decision of a problem solver, while global knowledge guides the problem solver strategy as a whole. [23] develops a case-based learning method for PRODIGY that consists of storing individual complete problems solved to guide the planner when solving similar new problems. The guiding similar plans provide global control knowledge in the sense that they consist of a chain of decisions. Other examples of learning global knowledge although with different granularities are [10, 11, 12, 13, 25]. These systems learn macro-operators or complete generalized plans HAMLET, however, learns local control rules that apply independently to individual decision steps with greater potential for transfer. We are already studying the integration of the two kinds of strategic knowledge, since we believe they have complementary benefits to the problem solver.

Previous work has usually learned control knowledge on simple problem solvers such as linear planners with no more than one goal [8, 14, 15, 16, 17, 19, 26]. In this kind of problem solver the underlying complexity of interleaving goals at different levels of the search does not exist, so the learning methods lack these learning opportunities. Other new learning opportunities not present in linear problem solving, consist of opportunistic operator choices driven by other planning goals. In our case, the nonlinear planner introduces this factor of complexity and the learning task becomes more challenging, due to having to find the right language for describing the hypotheses among other things. Also, the kinds of domains/problems we use are more complex than the previously studied ones, except in [23]. There is no immediately clear way to directly compare our results to other systems that learn local control rules, because none applies to nonlinear problem solving.

Another difference can be found in the way positive and negative instances are presented to the system. Systems like [15, 20, 26] are one-step learning algorithms in that they work on all examples at one time. However, HAMLET learns incrementally the control knowledge.

Usually, rules produced by learning methods are worse than the ones produced by the experts. However, learning methods produce those control rules much faster than the experts, since the acquisition of control knowledge is much harder than the acquisition of domain knowledge. We believe that one could get better results if one could present the learned rules to the expert and he or she could refine them to make them more accurate. Then, the issue of clarity of the rules is a major point and HAMLET's rules are very easy to read and require very little effort to debug or refine.

9 Conclusions

The approach we have presented addresses a new speedup learning strategy to efficiently acquire control knowledge for improving the performance of a nonlinear problem solver. We proposed a solution where we combine a bounded deductive explanation method with an inductive technique. In this case, the tradeoff between the accuracy of the learned control knowledge, and the learning effort required is addressed by bounding the learning step with a fixed set of tests that have been manually adapted from the experience of the authors. Upon experiencing new problem solving episodes, HAMLET refines its control knowledge incrementally acquiring increasingly correct control knowledge. Therefore, HAMLET combines analytical-based learning, using the problem solving search tree, and induction-based learning, refining and generalizing the control rules from examples. We showed empirical results that demonstrate the improvement that this learning strategy provides to PRODIGY's nonlinear problem.

Acknowledgements

We greatly appreciate the help from Juan Pedro Caraça-Valente in previous steps of the research, and insightful comments from Jaime Carbonell, Alicia Pérez, and the anonymous reviewers.

This work was sponsored for the first author by a grant of the Ministerio de Educación y Ciencia and the project CO44-91 from Comunidad de Madrid.

For the second author, this research was 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 U. S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation thereon. This manuscript is submitted for publication with the understanding that the U. S. Government is authorized to reproduce and distribute reprints for Governmental purposes.

References

1. Jim Blythe and Manuela M. Veloso. An analysis of search techniques for a totally-ordered nonlinear planner. In *Proceedings of the First International Conference on AI Planning Systems*, College Park, MD, June 1992.
2. Daniel Borrajo, Juan P. Caraça-Valente, and José Luis Morant. Learning heuristics in planning. In *Sixth International Conference on Systems Research, Informatics and Cybernetics*, Baden-Baden, Germany, 1992.
3. Daniel Borrajo, Juan P. Caraça-Valente, and Juan Pazos. A knowledge compilation model for learning heuristics. In *Proceedings of the Workshop on Knowledge Compilation of the 9th International Conference on Machine Learning*, Scotland, 1992.
4. Daniel Borrajo and Manuela 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.
5. Jaime G. Carbonell, and the PRODIGY Research Group. PRODIGY4.0: The manual and tutorial. Technical Report CMU-CS-92-150, School of Computer Science, Carnegie Mellon University, June 1992.
6. Jaime G. Carbonell, Craig A. Knoblock, and Steven Minton. Prodigy: An integrated architecture for planning and learning. In K. VanLehn, editor, *Architectures for Intelligence*. Erlbaum, Hillsdale, NJ, 1990. Also Technical Report CMU-CS-89-189.
7. Robert B. Doorenbos and Manuela 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.
8. Oren Etzioni. *A Structural Theory of Explanation-Based Learning*. PhD thesis, School of Computer Science, Carnegie Mellon University, 1990. Available as technical report CMU-CS-90-185.
9. Oren Etzioni and Steven 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.
10. Richard E. Fikes and Nils J. Nilsson. Strips: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2:189–208, 1971.
11. G. A. Iba. A heuristic approach to the discovery of macro-operators. *Machine Learning*, 3(4):285–317, 1989.
12. Subbarao Kambhampati and Smadar Kedar. Explanation based generalization of partially ordered plans. In *Proceedings of AAAI-91*, pages 679–685, 1991.

13. Richard E. Korf. Macro-operators: A weak method for learning. *Artificial Intelligence*, 26:35–77, 1985.
14. Pat Langley. Learning effective search heuristics. In *Proceedings of IJCAI-83*, pages 419–421, 1983.
15. C. Leckie and I. Zukerman. Learning search control rules for planning: An inductive approach. In *Proceedings of Machine Learning Workshop*, 1991.
16. Steven Minton. *Learning Effective Search Control Knowledge: An Explanation-Based Approach*. PhD thesis, Computer Science Department, Carnegie Mellon University, 1988. Available as technical report CMU-CS-88-133.
17. Tom M. Mitchell, Paul E. Utgoff, and R. B. Banerji. Learning by experimentation: Acquiring and refining problem-solving heuristics. In R. S. Michalski, J. G. Carbonell, and T. M. Mitchell, editors, *Machine Learning, An Artificial Intelligence Approach*, pages 163–190. Tioga Press, Palo Alto, CA, 1983.
18. M. Alicia Pérez and Jaime G. Carbonell. Automated acquisition of control knowledge to improve the quality of plans. Technical Report CMU-CS-93-142, School of Computer Science, Carnegie Mellon University, April 1993.
19. M. Alicia Pérez and Oren 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*. Morgan Kaufmann, San Mateo, CA, 1992.
20. J. Ross Quinlan. Learning logical definitions from relations. *Machine Learning*, 5:239–266, 1990.
21. Gerald J. Sussman. *A Computer Model of Skill Acquisition*. American Elsevier, New York, 1975. Also available as technical report AI-TR-297, Artificial Intelligence Laboratory, MIT, 1975.
22. Manuela M. Veloso. Nonlinear problem solving using intelligent casual-commitment. Technical Report CMU-CS-89-210, School of Computer Science, Carnegie Mellon University, 1989.
23. Manuela M. Veloso. *Learning by Analogical Reasoning in General Problem Solving*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, August 1992. Available as technical report CMU-CS-92-174.
24. Manuela M. Veloso and Jaime G. Carbonell. Derivational analogy in PRODIGY: Automating case acquisition, storage, and utilization. *Machine Learning*, 10:249–278, 1993.
25. Hua Yang and Douglas Fisher. Similarity-based retrieval and partial reuse of macro-operators. Technical Report CS-92-13, Department of Computer Science, Vanderbilt University, 1992.
26. J. Zelle and R. Mooney. Combining FOIL and EBG to speed-up logic programs. In *Proceedings of IJCAI-93*, 1993.