Appears in *New Directions in AI Planning*, IOS Press, 1996, pp. 129-140
Also appears in *Proceedings of the Third European Workshop on Planning, 1995.*

Hybrid Learning of Search Control for Partial-Order Planning*

Tara A. Estlin and Raymond J. Mooney
Department of Computer Sciences
University of Texas at Austin
Austin, TX 78712
{estlin,mooney}@cs.utexas.edu

Abstract. This paper presents results on applying a version of the Dolphin search-control learning system to speed up a partial-order planner. Dolphin integrates explanation-based and inductive learning techniques to acquire effective clause-selection rules for Prolog programs. A version of the UCPOP partial-order planning algorithm has been implemented as a Prolog program and Dolphin used to automatically learn domain-specific search control rules that help eliminate backtracking. The resulting system is shown to produce significant speedup in several planning domains.

1 Introduction

Efficient planning often requires domain-specific search heuristics; however, constructing appropriate heuristics for a new domain is a difficult, laborious task. Research in learning and planning attempts to address this important problem by developing methods that automatically acquire search-control knowledge from experience. However, most work in learning and planning has been in the context of linear, state-based planners[14, 8, 12].

More recently, the problem of learning search control for a nonlinear planner has been presented [3, 10, 18]. Nonlinear planners have been accepted as superior to linear planners for many years, and recent experimental results support that partial-order planners are more efficient than totally-ordered planners in most domains [1, 15, 9]. Though some past work has addressed this problem [5, 7], there has been little recent work in learning control knowledge for current partial-order planning systems [10].

Zelle and Mooney [23] introduced the Dolphin speedup-learning system (Dynamic Optimization of Logic Programs through Heuristics INduction), which combines explanation-based learning with induction to learn clause-selection heuristics for Prolog programs. Standard explanation-based learning can frequently produce complex, overly-specific control rules that decrease rather than improve overall planning performance [13]. However, incorporating induction to learn simpler, approximate control rules can improve the utility of acquired knowledge [4, 12]. Dolphin combines *explanation-based generalization* (EBG) [20, 6] with techniques from *inductive logic programming* (ILP) [19, 16, 11] to learn effective, approximate search-control rules. Zelle and Mooney [23]

*This research was supported by the NASA Graduate Student Researchers Program under grant number NGT-51332 and by the National Science Foundation under grant IRI-9310819.

present results verifying that this approach outperforms standard explanation-based methods at improving the performance of a simple, linear planner written in Prolog.

This paper significantly extends these results by applying DOLPHIN to speed up a partial-order planner. A version of the UCPOP planning algorithm [17] was implemented as a Prolog program in which plan-refinement decisions are coded as clause-selection points. This allows DOLPHIN to learn heuristics for picking appropriate refinement operators during planning. Since partial-order planners examine a very different search space then linear planners, there were several new challenges to overcome in applying DOLPHIN to this problem. The original DOLPHIN system had difficulty generalizing the complex explanations generated during partial-order planning. Instead of producing useful control information, DOLPHIN would generate a large number of over-specialized rules which often resulted in slower planning. To alleviate this problem, several modifications were made to DOLPHIN's generalization algorithm so that better information could be extracted from planning traces. DOLPHIN's control rule induction algorithm was also extended so that it could learn rules which cover more complex planning situations. We present experimental results that demonstrate the resulting system can learn control knowledge which significantly improves planning efficiency.

The remainder of the paper is organized as follows. Section 2 briefly reviews both UCPOP and DOLPHIN. Section 3 explains the format used to construct control rules for a Prolog version of UCPOP. Section 4 describes how a variant of DOLPHIN was used to learn search-control rules for UCPOP. Section 5 presents experimental results on speedup learning in two planning domains. Section 6 reviews related work, Section 7 discusses limitations and future directions, and Section 8 presents our conclusions.

2 Background

2.1 The UCPOP Planner

The base planner we chose for experimentation is UCPOP [17], a partial-order planner whose step descriptions can include conditional effects and universal quantification. In UCPOP, a partial plan can be described as a four-tuple: $\langle$S,B,O,L$\rangle$ where S is a set of actions, O is a set of ordering constraints over S, L is a set of causal links, and B is a set of codesignation constraints over variables appearing in S. Actions are described by schemata similar to STRIPS operators. These schemata contain a list of preconditions, an add list and a delete list. The set of orderings, O, specifies a partial ordering of the actions contained in S. During planning, there must always exist at least one consistent total ordering of these actions. Causal links are used to record dependencies between the effects of one action and the preconditions of another. These links are used to detect *threats* which occur when a new action interferes with a past decision. UCPOP employs several threat resolution strategies to resolve these cases.

A brief overview of the UCPOP algorithm is shown in Figure 1. The algorithm takes three inputs: a plan $\langle$S,B,O,L$\rangle$, an agenda of outstanding goals G, and a set of action schemata Λ. The initial and goal states are represented in S by initially adding two extra actions A_0 and A_∞. The effects of A_0 correspond to the initial state and the preconditions of A_∞ correspond to the desired goal state.

UCPOP begins with a null plan and an agenda containing the top-level goals. In each planning cycle, a goal is selected from the agenda and an existing or new action is chosen to assert the goal. After an action is selected, the corresponding orderings,

```
Algorithm UCPOP(⟨S,B,O,L⟩,agenda,Λ)
    1. Termination: If agenda is empty, return ⟨S,B,O,L⟩.
    2. Goal Selection: Select a goal G from the agenda where G is a precondition
       of some action $A_{need}$.
    3. Operator Selection: Choose either an existing action (from S) or a new action
       (instantiated from Λ) $A_{add}$ that adds G. Let O' = O ∪ $\{A_{add} \prec A_{need}\}$, L'
       = L ∪ $\{A_{add} \rightarrow^{G} A_{need}\}$, and let B' be the updated set of bindings. If $A_{add}$
       is a new action let S' = S ∪ $A_{add}$.
    4. Update Goal Set: Let agenda' = agenda - G. If $A_{add}$ is newly instantiated
       add each of its preconditions to agenda'.
    5. Causal Link Protection: Check for any threats to causal links in L. For each
       detected threat select a threat resolution strategy (e.g. demotion, promotion)
       to remove it. Let O" be the updated set of orderings.
    6. Recursive Invocation: UCPOP(⟨S',B',O",L'⟩,agenda',Λ)
```

Figure 1: The UCPOP Partial-Order Planning Algorithm

casual links and codesignation constraints are added to O, L,and B. Step 4 removes the selected goal from the agenda, and if a *new* action was selected to assert it, that action's preconditions are added to the agenda. Step 5 checks for possible threats and resolves them by adding an additional ordering constraint. UCPOP is called recursively until the agenda is empty. On termination, UCPOP uses the constraints found in O to determine a total ordering of the actions in S, and returns this as the final solution.

2.2 The DOLPHIN Speed-up Learning System

DOLPHIN was designed to improve the performance of a Prolog program by learning control information. This information is in the form of clause-selection heuristics which reduce or eliminate the need for backtracking. The final output of the system is a new Prolog program that includes the learned selection rules as new program clauses.

As an example, consider the naive sorting program in Figure 2, which sorts a list by generating permutations of the list until it finds one that is ordered. Permutations are generated by permuting the tail of the input list and then inserting the head somewhere in the permuted tail. This program currently performs in $O(N!)$ time. The only nondeterminism comes from the definition of the predicate `insert/3` which can either insert an item at the beginning of a list or somewhere in the tail. This nondeterminism can be eliminated by learning a control rule for the first clause which will correctly predict when the item should be placed at the head of the list.

The result of applying DOLPHIN to this program is a new insert definition. Figure 3 shows the resulting clauses after control information has been added. The first insert clause will now only be applied when a element is being inserted into an empty list or if the new element is less than the head of the current list. The result is an $O(N^2)$ insertion sort program.

The DOLPHIN algorithm can be divided into three phases: example analysis, control rule induction, and program specialization. The input to the algorithm is a Prolog program, a set of training examples, and a list of program predicates for which the user wants to learn control rules. The output is a modified program which incorporates learned search-control information. The original DOLPHIN algorithm has been explained in detail in [22] and [23]. A brief outline of the algorithm is presented next.

```
naivesort(X,Y) :-  permutation(X,Y), ordered(Y).

permutation([],[]) :- true.
permutation([X|Xs],Ys) :-  permutation(Xs,Ys1), insert(X,Ys,Ys1).

insert(X,[X|Xs],Xs) :- true.
insert(X,[Y|Ys],[Y|Ys1]) :-  insert(X,Ys,Ys1).

ordered([X]) :- true.
ordered([X,Y|Ys]) :- X =< Y, ordered([Y|Ys]).
```

Figure 2: Naive Sort Program

```
insert(X,[X|Xs],Xs) :-  insert_control1(X,[X|Xs],Xs).
insert(X,[Y|Ys],[Y|Ys1]) :-  insert(X,Ys,Ys1).

insert_control1(X,[],[X]).
insert_control1(X,[Y|Z],[X,Y|Z]) :- X < Y.
```

Figure 3: Improved Insert Predicate

In the example analysis phase, the training problems are solved using the input Prolog program. During this process, two different outputs are produced: a set of correct and incorrect clause selection decisions and a set of generalized proofs of the training examples. A clause selection decision is a partially instantiated subgoal to which a clause was applied while solving a training problem. A correct decision for a clause is a subgoal that appears on the problem solution path. An incorrect decision is a subgoal that a clause attempted to reduce but failed, and the subgoal was then solved by a different clause through backtracking. Examples of correct and incorrect clause decisions are collected for all clauses specified for learning. These examples are later used as positive and negative control examples in the induction component.

In order to produce generalized proofs for the training problems, standard EBG techniques [6, 20] are applied to the original problem proofs. These proofs are tree-structured explanations that trace through the steps made during problem solving. In generalization, elements of the proof that are dependent on specific example facts are removed while the structure of the original proof is maintained.

The resulting generalized proofs are used in the control rule induction phase to guide an inductive search. Induction is used to build an operational definition of the class of subgoals to which a certain clause should be applied. The inductive search is based upon the Foil algorithm [19]; however, instead of searching through all possible literals, the search is modified to only consider the operational literals contained in the generalized proofs of the training examples. Dolphin builds a subgoal definition by finding clauses which cover some positive control examples but no negatives. New clauses are formed until all positive control examples are covered. When building a clause, literals from the generalized proofs are added as antecedents one at a time. At each step, the best literal is selected to add based on the Foil information-gain heuristic. Once a definition is complete, it is transformed into a selection rule for the original clause. The program specialization phase then incorporates the final set of clause selection rules into the original program.

3 Learning Control for Partial-Order Planning

An initial task of this research was to construct a partial-order planner in Prolog. Our planner is based upon the UCPOP algorithm,[1] described in Section 2.1. Planning decision points are represented as clause-selection problems (i.e. each choice is formulated as a separate clause). Since many planning decisions are domain-dependent, a separate Prolog program was produced for each domain. For each domain tested, an initial base planner is automatically generated from the set of domain operators.

The important decision points in UCPOP are goal selection, goal establishment (selecting an existing or new operator), threat selection and threat resolution. Currently, DOLPHIN has been focused to learn control rules for only one type of decision: selecting a new operator for goal establishment. This decision was chosen as a starting point for learning since it is frequently backtracked upon. Future research will expand the learning component to cover all types of control decisions.

The program predicate representing this decision point is `find_new_action/3`. This predicate is defined by a set of clauses each corresponding to a different effect of a particular operator. To illustrate, several `find_new_action` clauses from the blocksworld are shown below:

```
find_new_action(clear(A),S,O,L,Agenda,unstack(B,A)) :- true.

find_new_action(clear(A),S,O,L,Agenda,putdown(A)) :- true.

find_new_action(clear(A),S,O,L,Agenda,stack(A,B)) :- true.
```

The inputs to this predicate are a goal, and the current lists of actions, ordering constraints, causal links and unachieved goals.[2] The output is an action that contains the goal in its add list. The three clauses shown are the set of `find_new_action` clauses that select an action to achieve the goal `clear(A)`.

Control rules are learned to select between such clauses. An example of a clause which has been specialized by adding control information is shown below:

```
find_new_action(clear(A),S,O,L,Agenda,unstack(B,A)) :-
     find_existing_action(on(B,A),S,Action).
```

This rule states that `unstack(B,A)` should be selected to add `clear(A)` when there is an existing action in the current plan that adds `on(B,A)`. (The variable `Action` is an output parameter that corresponds to the action in `S` that establishes `on(B,A)`).

Learned control information is incorporated into the base program so that attempts to select a clause inappropriately will immediately fail. Control rules can consist of a single clause or a disjunction of several clauses. DOLPHIN can also make clause selection deterministic by adding a cut (!) to end of the rule to prevent backtracking. A cut is added when a learned control rule does not cover any negative examples.

[1]Neither of the domains tested for this paper actually use conditional effects or universal quantification. Experiments on more complex planning domains are planned for future work.

[2]The last four inputs are only needed for added control information and not for nondeterministic operation. The problem of adding required parameters to predicates for control purposes is discussed in future work.

4 Extending DOLPHIN

Previously, Dolphin was shown effective at speeding up the N-Queens problem, a symbolic integration program, and planning in the STRIPS robot world and the blocksworld [23]. The planning domains employed a linear, means-ends analysis planner. Dolphin has now been extended so that is can successfully speedup a partial-order planner.

The search space of a partial-order planner greatly differs from that of a standard linear planner. Items such as goal interleaving, constraint handling, and constraint representation all contribute to the greater complexity of partial-order planning. These extra complexities raise new problems when analyzing planning traces for useful control information. To handle these new issues, several important changes where made to the system. One set of modifications is directed towards increasing the space of generalized literals examined by the induction phase so that better quality rules can be constructed. Another change involves reordering program clauses during program specialization to increase program efficiency. These changes are detailed in the following sections.

4.1 Improving EBG Literals

As explained in Section 2.2, Dolphin uses literals extracted from the generalized proofs of the training problems to build control rules. The conjunction of operational literals from the leaves of a generalized proof tree is known to be a set of sufficient conditions for solving similar problems. However, there is no guarantee that the individual literals examined by Dolphin will be useful as control information. Even after EBG is performed, literals can still be too specialized to be useful. Due to the complexity of the UCPOP search space, many of the literals in the UCPOP proof traces were difficult to generalize and only decreased control rule utility. A main problem is that these literals often contain list structures that hold detailed plan information, such as a list of ordering constraints or causal links. These structures are difficult to generalize into a form that would easily apply to later planning examples.

To alleviate this problem, these complex list parameters are now generalized away to single variables. This process increases the utility of a literal by allowing it to cover more positive examples during induction while still leaving enough specialized information in other parameters to successfully rule out negatives. If a literal becomes too general, it can be specialized once Dolphin attempts to add it to a rule, as explained below.

A literal can often be further improved once it has been added to a rule. New antecedents can sometimes be specialized using information already contained in the rule. When Dolphin tests a literal for use in a control rule, the literal can contain input parameters that are not bound by the head or existing literals in the rule. These unbound inputs will cause the rule to be overly general and cover too many negative examples. However, it is possible to successfully use such literals by unifying unbound inputs with existing terms of the same type already present in the rule. For example, the rule shown below has an antecedent with an unbound input, `S1`.

```
find_new_action(clear(A),S0,O,L,Agenda,unstack(B,A)) :-
     find_existing_action(on(B,A),S1,Action).
```

This rule is too general since `S1` is unconstrained. Dolphin can now modify the rule, as shown below, so that the `S1` is matched with a term of the same type from the head of the rule.

```
find_new_action(clear(A),S,O,L,Agenda,unstack(B,A)) :-
     find_existing_action(on(B,A),S,Action).
```

This method can often construct better control rules by considering more literals as potentially useful antecedents.

4.2 Considering Additional Literals

DOLPHIN's original induction algorithm only considered literals exactly as they appear in the generalized proof trees. However, negated versions of proof-tree literals are frequently useful as control conditions. A good reason for not selecting a clause is that a subsequent clause is preferable. Thus a good control-rule antecedent for one clause can be successfully used as a negated antecedent in a rule for an earlier competing clause. Clauses are considered competing if they can both reduce the same subgoal (i.e. competing clauses have matching predicate names and inputs but usually produce different outputs). The three `find_new_action` clauses shown in Section 3 illustrate a set of competing clauses.

Potential negated antecedents for a particular clause's control rules are determined by combining the sets of possible antecedents for the control rules of all following competing clauses. These antecedents can often be easily incorporated into a rule by using the previously described technique of unifying unbound input terms with existing terms in the rule.

One other technique of adding antecedents is also introduced. The original algorithm followed the FOIL approach of only adding antecedents to a rule until all negatives are removed. If there are any positives left to cover, a new rule is begun. DOLPHIN now considers adding negated antecedents in order to cover more positives. Instead of only appending negated antecedents to the end of a control rule, the new induction algorithm considers conjunctively grouping them with existing negated antecedents. This procedure can increase the number of positive examples covered by a rule. For example, assume the induction algorithm is currently considering adding the antecedent: not(ant4) to the following clause.

```
rulehead :- ant1, not(ant2), ant3.
```

The new algorithm can form either of the two clauses shown below, where previously only the first clause would have been considered. The second clause could possibly cover more positive examples than the original clause.

```
rulehead :- ant, not(ant2), ant3, not(ant4).
rulehead :- ant, not(ant2, ant4), ant3.
```

The induction algorithm has been changed so that if all negatives have been ruled out but there are still positives left to cover, additional negated antecedents are considered that could help cover more positives. New antecedents are considered until they no longer produce positive gain.

DOLPHIN can now learn complex rules such as the one shown below learned for the blocksworld domain:

```
find_new_action(holding(A),S,O,L,Agenda,unstack(A,B)) :-
       not(find_existing_action(on_table(A),S,Action1),
           can_add_ordering(Action1,unstack(A,B),O)),
       find_existing_action(on(A,B),S,Action2).
```

This rule states that unstack(A,B) should be chosen to add holding(A) when *there is not* an existing action in the current plan that adds on_table(A) and that can be ordered before unstack(A,B), and *there is* an existing action that adds on(A,B).

These added extensions for rule construction are very beneficial when learning rules for partial-order planners. Unlike linear planning, there is no "current state" that can be easily reasoned about. Instead, control rules must be able to analyze more complex partial-plan structures. By increasing the number of methods used to construct control rules, DOLPHIN can formulate better rules.

4.3 Clause Reordering

One final change was added to further increase planner efficiency. The original DOLPHIN algorithm preserved clause ordering during program optimization. New clauses incorporating control rules were output in the same order found in the original program. Now, clause reordering is used to further increase program efficiency. Before starting induction, competing clauses are grouped together and ordered by increasing number of positive examples. This reordering puts more frequently used clauses later in the program where they are selected by default if no earlier clause is chosen. Control rules are only learned for the preceding clauses which are used less frequently. It is much easier for induction to form simple controls rules for these clauses since there are fewer positive control examples to cover.

Together these changes enable DOLPHIN to significantly improve a Prolog version of the UCPOP partial-order planner. These changes were primarily motivated by observing DOLPHIN's actions when applied to such a planner. However, we feel many of them would also be beneficial in learning search control for other types of problem solving.

5 Experimental Evaluation

5.1 Experimental Design

The blocksworld and logistics transportation domains were used to test the final system. In the logistics domain [21], packages must be delivered to different locations in several cities. Packages are transported between cities by airplane and within a city by truck. In both domains, a test set of 100 independently generated problems was used to evaluate performance. DOLPHIN was trained on separate example sets of increasing size. Ten trials were run for each training set size, after which results were averaged. Training and test problems were generated using the following methods. In the blocksworld, problems were generated by creating a random initial state and applying a random sequence of operators. Problems contained five to ten blocks and one to five goals. In the logistics domain, problems were generated by placing packages in random initial and final locations. Problems in this domain contained up to two packages, three cities, and one or two goals. No time limit was imposed on either domain, but a uniform depth bound on the plan length was used during testing that ensured testing in a reasonable amount of time.

For each trial, DOLPHIN learned control rules from the given training set and produced a modified program. Since DOLPHIN only specializes clauses in the original program, the new program is guaranteed to be sound with respect to the original program. Unfortunately, the output program is not guaranteed to be complete. Some clauses could be too specialized and thus the final program may not solve all problems solvable by the original program. In order to guarantee the completeness of the new program, a strategy from [4] is adopted. If the final program fails to find a solution to a test problem, the initial program is used to solve the problem. When this situation occurs in testing, both the failure time for the new program and the solution time for the original program are included in the total solution time for that problem.

5.2 Results

Figures 4 and 5 present the experimental results. The times shown represent the number of seconds required to solve the problems in the test set. In both domains, DOLPHIN consistently produced a more efficient program and significantly decreased solution times on the test sets. In the blocksworld, DOLPHIN produced new programs that were an average of 3.6 times faster than the original program. For the logistics domain, DOLPHIN produced programs that were an average of 1.82 times faster. These results show that DOLPHIN can successfully improve the efficiency of a partial-order planner.

Unfortunately, we are not able to reports results from running the original DOLPHIN system on these domains. The original DOLPHIN had difficulty running efficiently on these problems and could not be tested consistently on large training set sizes. The trials we were able to perform showed DOLPHIN only *increased* planning times by learning too many overly-specialized control rules. For example, when trained on 10 examples in the blocksworld domain, the original DOLPHIN produced programs that were an average of 1.8 times *slower* than the original base planner.

6 Related Work

It is difficult to directly compare DOLPHIN's performance to other systems since there has been little research on learning search control for current partial-order planners. Most related learning systems have been tested on different planning algorithms. For example, SNLP+EBL [10] learns search-control rules for SNLP, a predecessor of UCPOP. SNLP does not handle universal quantification or conditional effects, however, a comparison can still be made between the two learning systems on a domain which does not employ these extra features. For these tests, we followed a similar experimental method to that used in [10]. Problems were randomly generated from the blocksworld domain that contained between three to eight blocks and two to six goals.[3] DOLPHIN was trained on a set of 50 problems. The test set contained 100 problems and a CPU time limit of 120 seconds was imposed during testing. Data was collected on planner speedup and on the number of test problems that could be solved under the time limit. The results are shown in the following table.

System	*Orig. Time*	*Final Time*	*Speedup*	*Orig. %Sol*	*Final %Sol*
SNLP+EBL	6063	2503	2.42X	51%	81%
DOLPHIN	7365	1895	3.89X	63%	96%

DOLPHIN achieved a higher speedup ratio, producing a more efficient planner for this domain. Both systems were able to increase the number of test problems solved. The

[3]It should be noted that the blocksworld domain theory used for DOLPHIN slightly differed from the one used for SNLP+EBL. Both domains employed similar predicates however DOLPHIN's domain consisted of four operators while the SNLP+EBL domain used only two.

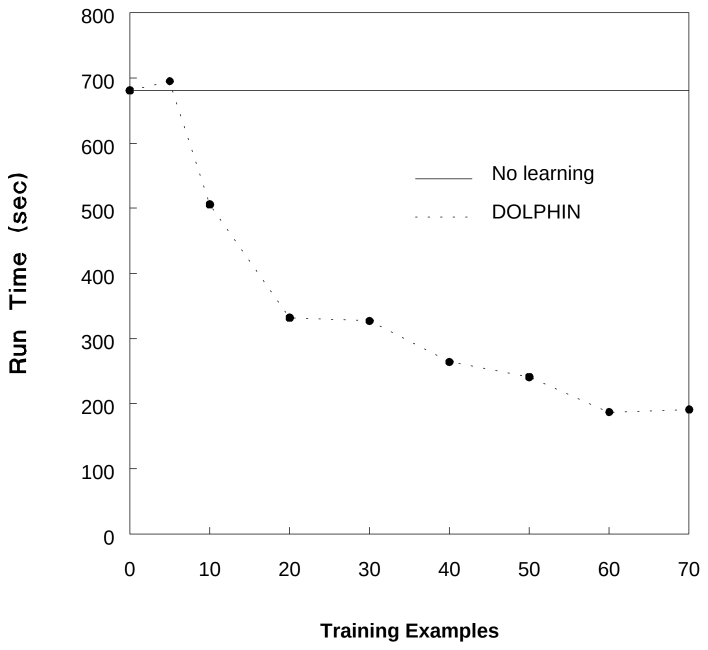

Figure 4: Dolphin Performance in the Blocks World Domain

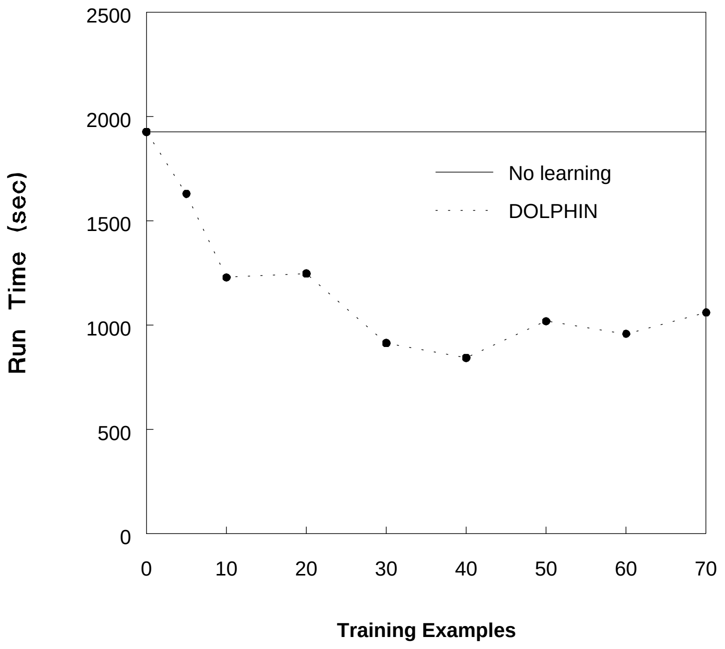

Figure 5: Dolphin Performance in the Logistics Domain

results are particularly significant since SNLP+EBL was used to learn control rules for several types of planning decisions while DOLPHIN was constrained to learn rules for only new action selection. SNLP+EBL also requires additional domain knowledge which was not provided to DOLPHIN.

HAMLET [3] is another related system which learns control knowledge for the nonlinear planner underlying PRODIGY4.0. Hamlet also combines induction with EBL but uses a very different approach than DOLPHIN. It is diffcult to directly compare HAMLET and DOLPHIN for several reasons. First, HAMLET is directly integrated with the PRODIGY4.0 system which does not use a partial-order planner. PRODIGY4.0 also employs some initial built-in control knowledge not given to our planner. HAMLET has also successfully improved planner performance in the blocksworld and logistics planning domains. When just comparing speedup factors, it appears DOLPHIN achieves greater speedup in blocksworld, and similar speedup in logistics.

7 Future Work

There are several issues that should be addressed in future research. The learning algorithm needs to be expanded to cover additional planning decisions. Decision points such as simple goal establishment (selecting an existing action), goal selection and threat resolution could also be made more efficient by learned control information. DOLPHIN should also be tested on domains which contain conditional effects, universal quantification, and other more-expressive planning constructs.

We also plan to examine ways of using DOLPHIN to improve plan quality as well as planner efficiency. Borrajo and Veloso [2] and Pérez and Carbonell [18] have used learned control information to guide a planner towards better solutions. DOLPHIN could be modified to collect positive control examples in a different method so that control rules are focused on quality issues as well as speedup.

One shortcoming of the current approach is its dependence on the initial program structure. Predicates for which control rules are being learned may leave out important control information. Some parameters that are not necessary for nondeterministic operation may be needed to make control decisions. Such parameters could be automatically added to program predicates during control rule learning.

DOLPHIN's generalization process could also be improved to extract better literals directly from the training-example proofs. In addition, replacing FOIL's information-gain metric for picking literals with a metric that more directly measures resulting control-rule utility could improve ultimate planning performance.

8 Conclusions

Learning search control for current planning systems is an important under-studied problem. This paper has presented a new approach for automatically improving the efficiency of a partial-order planner. The DOLPHIN learning system combines explanation-based and inductive methods to learn search control rules which improve program performance. The original DOLPHIN system was modified in several important ways to allow it to learn more effective control rules for a planner directly based on the UCPOP algorithm. The resulting system was shown to significantly improve planning performance in several domains.

References

[1] A. Barrett and D. Weld. Partial order planning: Evaluating possible efficiency gains. *Artificial Intelligence*, 67:71–112, 1994.

[2] D. Borrajo and M. Veloso. Incremental learning of control knowledge for improvment of planning efficieny and plan quality. In *AAAI-94 Fall Symposium*, pages 5–9, New Orleans, LA, October 1994.

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

[4] W. W. Cohen. Learning approximate control rules of high utility. In *ML-90*, pages 268–276, Austin, TX, June 1990.

[5] D. Croft. Choices made by a planner: Identifying them, and improving the way in which they are made. Master's thesis, University of Edinburgh, 1984.

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

[7] R. Desimone. Learning control knowledge within an explanation-based learning framework. In *Proceedings of 2nd European Working Session on Learning, EWSL-87*, Bled,Yugoslavia, May 1987.

[8] J. Gratch and G. DeJong. COMPOSER: A probabilistic solution to the utility problem in speed-up learning. In *AAAI-92*, pages 235–240, San Jose, CA, July 1992.

[9] S. Kambhampati and J. Chen. Relative utility of EBG based plan reuse in partial ordering vs. total ordering. In *AAAI-93*, pages 514–519, Washington, D.C., 1993.

[10] S. Katukam and S. Kambhampati. Learning explanation-based search control for partial order planning. In *AAAI-94*, pages 582–587, Seattle,WA, August 1994.

[11] N. Lavrač and S. Džeroski, editors. *Inductive Logic Programming: Techniques and Applications.* Ellis Horwood, 1994.

[12] C. Leckie and I. Zuckerman. An inductive approach to learning search control rules for planning. In *IJCAI-93*, pages 1100–1105, Chamberry,France, August 1993.

[13] S. Minton. Quantitative results concerning the utility of explanation-based learning. In *AAAI-88*, pages 564–569, St. Paul, MN, August 1988.

[14] S. Minton. Explanation-based learning: A problem solving perspective. *Artificial Intelligence*, 40:63–118, 1989.

[15] S. Minton, M. Drummond, J. L. Bresina, and A. B. Phillips. Total order vs. partial order planning: Factors influencing performance. In *KR-92*, pages 83–92, Cambridge,CA, October 1992.

[16] S. Muggleton, editor. *Inductive Logic Programming.* Academic Press, NY, NY, 1992.

[17] J.S. Penberthy and D. S. Weld. UCPOP: A sound, complete, partial order planner for ADL. In *KR-92*, pages 113–114, Cambridge,MA, October 1992.

[18] M. A. Pérez and J. Carbonell. Control knowledge to improve the plan quality. In *AIPS-94*, Chicago, IL, June 1994.

[19] J.R. Quinlan. Learning logical definitions from relations. *Machine Learning*, 5(3):239–266, 1990.

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

[21] M. Veloso. *Learning by Analogical Reasoning in General Problem Solving.* PhD thesis, Carnegie Mellon University, August 1992.

[22] J. M. Zelle. Learning search-control heuristics for logic programs: Applications to speed-up learning and language acquisition. Technical Report AI93-200, University of Texas, Austin, TX, 1993.

[23] J. M. Zelle and R. J. Mooney. Combining FOIL and EBG to speed-up logic programs. In *IJCAI-93*, pages 1106–1111, Chambery, France, 1993.