Learning from Goal Interactions in Planning: Goal Stack Analysis and Generalization

Kwang Ryel Ryu and **Keki B. Irani**
Artificial Intelligence Laboratory
Department of Electrical Engineering and Computer Science
The University of Michigan, Ann Arbor, MI 48109-2122
nbge@caen.engin.umich.edu & irani@caen.engin.umich.edu

Abstract

This paper presents a methodology which enables the derivation of goal ordering rules from the analysis of problem failures. We examine all the planning actions that lead to failures. If there are restrictions imposed by a problem state on taking possible actions, the restrictions manifest themselves in the form of a restricted set of possible bindings. Our method makes use of this observation to derive general control rules which are guaranteed to be correct. The overhead involved in learning is low because our method examines only the goal stacks retrieved from the leaf nodes of a failure search tree rather than the whole tree. Empirical tests show that the rules derived by our system PAL, after sufficient training, performs as well as or better than those derived by systems such as PRODIGY/EBL and STATIC.

Introduction

When a problem solver is given a problem with a conjunctive goal condition, perhaps the most difficult task to perform is the ordering of the individual component goals to avoid harmful goal interactions. Efforts have been made in the past to cope with this problem either by learning [Sussman 1975; Minton 1988] or through reasoning [Cheng and Irani 1989; Etzioni 1991]. The learning approach taken by Minton's PRODIGY [Minton 1988] shows comprehensive coverage on learning from various problem solving phenomena by using EBL (Explanation-Based Learning). The goal ordering rules learned through PRODIGY/EBL, however, are often overly specific and unnecessarily verbose. The reasoning approaches taken by Cheng's system [Cheng and Irani 1989] and Etzioni's STATIC [Etzioni 1991] derive very general goal ordering rules by analyzing domain operators. STATIC's problem space compilation method derives various types of search control rules in addition to goal ordering rules. However, since both these methods are not based on analysis of problem-solving traces, certain constraints specifically imposed by dynamic attributes[1] of a problem state cannot be detected. Moreover, the above systems require *a priori* domain-specific knowledge. PRODIGY/EBL uses such knowledge in its *compression* phase which is critical for enhancing the utility of the learned rules. STATIC and Cheng's system rely on such knowledge to reason about the effects of the operators. Whenever a new problem domain is given, it is not trivial to determine *a priori* what type of knowledge is needed and in which particular form.

In this paper, we present a new learning method which can derive goal ordering rules from the analysis of problem failures without relying on any explicit *a priori* knowledge beyond that given in the problem description. The condition of a rule derived by this method involves dynamic as well as static attributes[2] of a problem state. These conditions are guaranteed to be correct and much more general than those of PRODIGY/EBL, although slightly less general than those of Cheng and of STATIC. A planning and learning system called PAL has been implemented and tested on PRODIGY's test domains of Blocksworld, Stripsworld, and Schedworld (machine scheduling). The rules derived by PAL performed as well as, and in some cases better than, those of the other systems, while the overhead of learning in PAL is shown to be the lowest.

In the following, we first review the goal interaction problem and discuss the limitations of previous approaches. Then, we give an overview of the PAL's planner and describe PAL's learner and how it learns from failure analysis. Next come the experimental results which compare PAL with PRODIGY/EBL and STATIC. Other related works are then briefly discussed before concluding remarks.

Goal Interaction Problem

Goal interaction refers to the phenomena where the actions for achieving the individual component goals

[1] The attributes that can change by operator application such as the locations of movable objects.

[2] The attributes which are not dynamic such as a room configuration or unchangeable properties of objects.

interfere with one another. Harmful interactions are avoided if the goals are attacked in a right order. Goal ordering rules prescribe the condition under which a certain ordering should be followed. Conditions derived by previous approaches have limitations as illustrated by the following example.

In Stripsworld, condition for goal ordering is sometimes represented by a complicated relational concept. When the goals are (*status dx closed*) and (*inroom ROB rx*), for example, *ROB*(robot) must get into *rx before* closing *dx* if *dx* is the *only* doorway to *rx*. On the other hand, if *dx* is a door not connected to *rx*, *ROB* should get into *rx after* closing *dx*. The control rule learned from the first case should contain in its antecedent the following condition in addition to the goals:

$$\forall d \ (connects \ d \ rx \ ra) \rightarrow d = dx.$$

Similarly, the rule learned from the second case should have the following condition:

$$\forall r \ (connects \ dx \ r \ rb) \rightarrow r \neq rx.$$

The current implementation of STATIC fails to derive these conditions and constructs the over-general rule that prefers (*inroom ROB rx*) to (*status dx closed*) regardless of the relative location of *dx*. Over-general rules can mislead a planner into generating inefficient plans by expanding extraneous nodes. Cheng's system derives the right conditions with the help of domain-specific knowledge. STATIC may similarly need additional domain-specific knowledge to derive the right conditions. When the correct condition for goal ordering involves dynamic attributes of a problem state, these two systems fail because they do not examine problem states or search traces. PRODIGY derives over-specific conditions which overly restrict the applicability of its rules while demanding high matching cost. PAL is able to derive correct conditions through analysis and generalization of the goal stacks retrieved from a failure search tree.

PAL's Planner

PAL's planner employs *means-ends analysis* [Newell, Shaw, and Simon 1960] as its basic search strategy. Given a problem with a conjunctive goal condition, PAL processes the component goals from the goal stack in the order presented or as recommended by learned rules. When a goal is true in a state, the goal is removed from the goal stack. Otherwise, for each instantiation of each of the goal's relevant operators, a new copy of the goal stack is created. The instantiated operator and its preconditions are posted on a new copy of the goal stack. These preconditions become subgoals and are treated in the same way as the top level goals. An operator in a goal stack is applied when all its preconditions are satisfied. If an operator is applied, it is deleted from the goal stack together with the goal or subgoal that it achieves. This process recurs until an empty goal stack is found (success) or a dead-end is reached (failure).

```
pushthrudr (?b ?d ?rx)
(sp) (pushable ?b) (connects ?d ?ry ?rx)
(bp)
(dp) (status ?d open) (inroom ?b ?ry)
     (nextto ?b ?d) (nextto ROB ?b)
     (inroom ROB ?ry)
(dl) (nextto ROB ?$) (nextto ?b ?$)
     (nextto ?$ ?b)(inroom ROB ?$) (inroom ?b ?$)
(ma) (inroom ?b ?rx)
(sa) (inroom ROB ?rx) (nextto ROB ?b)
```

Figure 1: An operator used in PAL.

Operator Representation and Control Heuristics

In PAL's planning model, the literals used in plan representation are separated into two categories, *static* and *dynamic*. The static literals in a state description can never be changed throughout legal state transitions dictated by operator applications. Dynamic literals are those that can be deleted and/or added by operators.

Figure 1 illustrates the six-slot operator representation used in PAL. Most of the default search control heuristics are incorporated into this representation scheme. The first three slots are for the preconditions, the *dl* slot is for the delete list, and the *ma* and *sa* slots are for the elements of add list exhibiting major effect and side effect of the operator, respectively. The static literals of the preconditions are separately kept in the *sp* slot. When a static precondition of an operator is unsatisfied, the operator is immediately abandoned because it is not applicable. The preconditions in *bp* slot, although dynamic, play the same role as static preconditions to heuristically restrict the selection of bindings. Only the dynamic preconditions in *dp* slot can become subgoals when they are unsatisfied. A similar scheme is adopted in [Dawson and Siklòssy 1977]. The *ma*, *sa* separation of add list implements operator selection heuristics. An operator is considered relevant to a goal only when it has a literal in its *ma* slot which unifies with the goal. This easily implements the kinds of heuristics that prevent object-moving operators from being used to move the robot, as can be seen in the *pushthrudr* operator. Although we claim not to be using *a priori* domain knowledge explicitly, we may be using it implicitly in the operator descriptions.

Strategy for Handling Goal Interactions

PAL encounters harmful goal interactions in two different forms, namely, protection violation and prerequisite violation. A protection violation is detected when an application of an operator deletes a previously achieved goal. For PAL, a protection violation is a failure and therefore it backtracks. A prerequisite

violation is detected when a precondition of an operator is found deleted by another operator applied for a previous goal. A prerequisite violation, however, is considered a failure only when the violated precondition is impossible to be re-achieved. Such irrecoverable prerequisite violations are frequently observed in the Schedworld domain. If violations are unavoidable with a given ordering of goals, PAL re-orders the goals.

Learning from Goal Interactions

PAL learns goal ordering rules if goals attacked in a certain order lead to failures which involve violations. Learning involves derivation of the general condition under which a certain ordering fails. The central idea of our method is to take into account the mechanism of operator instantiation during failure analysis. We can understand how certain harmful interactions are unavoidable in a given situation by examining how all possible ways of taking actions lead to dead ends. If there are restrictions imposed by the problem state on taking possible actions, then the restrictions manifest themselves in the form of a restricted set of possible bindings of operators. Our method makes use of these observations to derive general control rules without relying on any *a priori* domain-specific knowledge. Due to space limitation, we describe our method for the case of learning from two-goal problems, although the method can be extended to n-goal problems [Ryu 1992].

Mechanism of Operator Instantiation

When an operator relevant to a goal is instantiated, an initial binding is first generated by matching the goal with an add literal. This binding is later completed by matching preconditions with the current state description. A dynamic precondition under a certain binding becomes a subgoal if it is not satisfied by the state under that binding. Given a dynamic precondition p of an operator op, a set D of preconditions other than p is called a *determinant* for p with respect to an add literal a of op if all the variables appearing in p except those in a appear in D. For example, the set of two literals in the *sp* slot of the *pushthrudr* operator in Figure 1 is a determinant for any precondition in the *dp* slot with respect to the add literal in *ma*.[3] PAL assumes that for any operator *op*, its *sp* together with the *bp* slot constitute a determinant for any dynamic precondition in the *dp* slot of *op* with respect to any add literal in its *ma* slot.[4]

Goal Stack Analysis

The reason why all attempts to achieve a subgoal lead to failures is well recorded in the goal stacks of the failure search tree. Consider a simple problem with the initial state of Figure 2 and the goals g_1 = (*status D25 closed*) and g_2 = (*inroom B1 R2*). If g_1 is achieved first, every attempt to achieve g_2 either leads to a goal stack cycle (failure by goal repetition) or violation of g_1 because *D25* is the only pathway for *B1* to be moved into *R2*. PAL extracts and generalizes this configurational constraint and the relative location of the box *B1* by investigating the failure goal stacks.

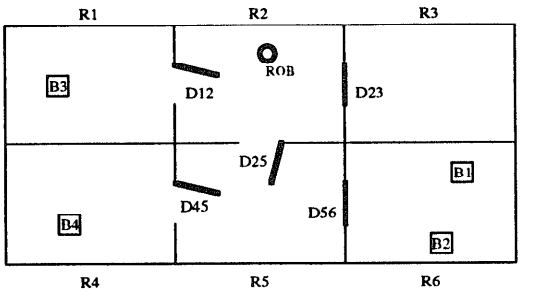

Figure 2: A state in the robot navigation domain.

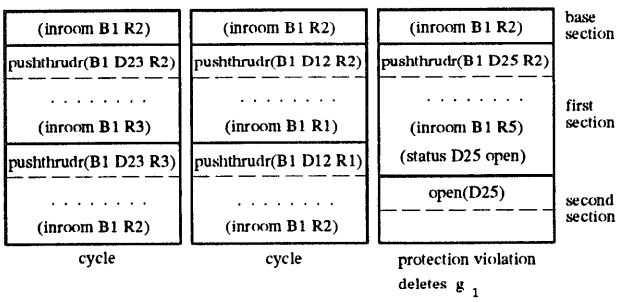

Figure 3: Some failure goal stacks.

Figure 3 shows some of the goal stacks retrieved from the failure search tree obtained during the attempt to achieve g_2 after achieving g_1. Each goal stack is divided into sections for easy reference. The section containing the top level goal is called the *base section*. The remaining part is uniformly divided into sections each of which consists of an operator and its unsatisfied preconditions. They are called the *first section*, *second section*, and so on as indicated in the figure. The subgoal at the top of each section (note the goal stacks are shown upside down) is said to be *active*. Sometimes a section of a failure goal stack contains an active subgoal which cannot be achieved because the binding is not correct. Such an active subgoal, if achieved in other search branch under a right binding, is said to be *achievable*. Each of the active subgoals in the goal stacks of Figure 3 is not achieved in any search branch. This type of active subgoals are called *unachievable*. The unachievable active subgoals constitute the real reason for failure. A failure goal stack whose active subgoals are all unachievable is said to

[3] PAL's determinants are similar to PRODIGY's binding generators.

[4] Under a certain condition, this assumption can be relaxed without compromising the correctness of the learned rule conditions. Details are explained in [Ryu 1992].

be an instance of an *unavoidable* failure. Only from the goal stacks of all the unavoidable failures, can we construct an explanation of how all the alternative attempts to achieve a goal lead to failures. In our example, only the three goal stacks shown in the figure are used for learning.

Let K be a set of all the unavoidable failure goal stacks from the failure search tree obtained during the attempt to achieve the second goal after achieving the first goal. Let e_j^i denote the j-th section of a goal stack $k_i \in K$. Two sections of different goal stacks are identical if the operators and the ordered preconditions in the two sections are exactly the same. The following two lemmas are helpful in further processing the goal stacks.

Lemma 1 *Let k_i and k_h be two different goal stacks in K. If the active subgoals of e_j^i and e_m^h are identical for some $j, m \geq 1$, then $j = m$ and $e_n^i = e_n^h$ for all $1 \leq n \leq j$.*

The lemma should be intuitively obvious because the goal stacks k_i and k_h are expanded copies of the same goal stack (see Section PAL's Planner).

Lemma 2 *Let p be an active subgoal of the j-th section which is not the top section of a goal stack in K. Then, each of all the instantiated relevant operators considered for p during search appears in the $(j+1)$-th section of a goal stack in K.*

The goal stacks in K show all the unsuccessful attempts to achieve goals from the top level to the lowest level until failures are explicitly detected. Based on the fact that the goal stacks of alternative attempts share identical portions from the base up to the sections of different attempts (Lemma 1), PAL re-arranges these stacks into a tree structure. We can view each goal stack as a chain of sections with each of the sections connected by edges, starting from the base to the top section. The base section, shared by all the goal stacks, becomes the root of a tree to be constructed. Then, starting from the first section for each level, any identical sections seen at a level are merged into a single node, with the sections at the next level becoming children of the merged node.

Figure 4 (a) shows the tree (called C-tree) derived from our failure goal stacks. Each node in the tree corresponds to a section of a goal stack. However, in each node of the tree, the operators are replaced by their determinants (*sp* and *bp* slots) and the preconditions other than the active ones are simply removed. The determinant in a node can be considered a condition under which the node's subgoal emerges given the goal of its parent node. The leaf node of protection violation does not contain any determinant because the deletion causing the violation is solely determined by the initial binding. This node does not contain a subgoal either. Instead, it contains the literal which is deleted.

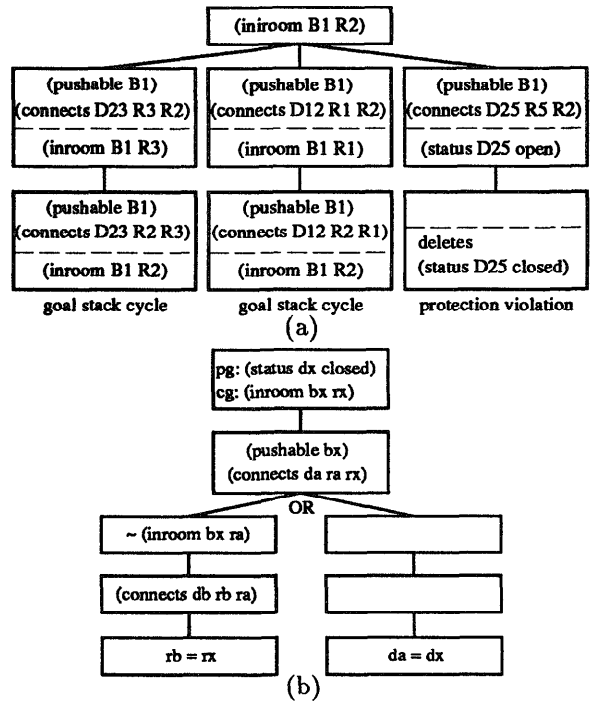

Figure 4: The C-tree and F-tree of our example.

Generalization

PAL generalizes the C-tree by substituting constants with variable symbols starting from the root in a top-down manner. The constants resulting from instantiation of operator variables are denoted by upper case letters. Only these constants are generalized to variables. Variables are denoted by lower case letters. The generalization process can be viewed as the reverse of the operator instantiation. First, the goal at the root is variabilized and the constant/variable substitution used is passed down and applied to the children nodes. This is the reverse of the initial binding generated when the relevant operators for the goal are instantiated. The already achieved goal (g_1) is also variabilized and kept separately. Next, the determinants of sibling nodes under the root are generalized to look identical if they are just different instantiations of the same determinant of an operator. Once a determinant of a node is generalized, the subgoal in the same node can be accordingly generalized by applying the constant/variable substitution used in variabilizing the determinant. After all the nodes at the same level are generalized, the nodes are split into parent and child with the determinant as a parent and the subgoal as a child. Negation symbols are attached in front of the generalized subgoals to reflect the fact that they are not satisfied. All the determinants generalized to look identical are then merged into a single node moving their children (subgoals) under the merged node. Any

identical subgoal nodes are also merged at this time and different children are put under an OR branch to indicate the fact that different subgoals can be created from a single operator depending on the bindings. Then, only the substitutions used for variabilizing the subgoals are passed down and applied to the nodes at the next level and the same process is repeated. The children of a subgoal node are always put under an AND branch because *all* the relevant operators for the subgoal should lead to failures. When a leaf node is generalized, the repeated subgoals (cycle failure) and violated goals are replaced by the respective conditions for failure. Such failure conditions are easily derived by examining the repeated subgoal and the delete/add lists of the operator involved in the violation.

As a final step, PAL removes redundant literals from the resulting tree. A literal in a determinant node is redundant if it already appears in one of its ancestor nodes. A literal $\neg p$ in a subgoal node is redundant if p is deleted by every operator which achieves g_1. The reason is that our tree is intended to represent the condition on the state after g_1 is achieved. In our example, $\neg$(*status da open*) is redundant under the condition *da* = *dx* because the operator achieving the violated goal (*status dx closed*) deletes it. The tree after generalization, called an F-tree, is shown in Figure 4 (b). The *pg* and *cg* in the root represent the previously achieved goal (g_1) and the current goal (g_2), respectively.

An F-tree is comparable to a failure subgraph of STATIC's problem space graph (PSG) where goal/subgoal relationships are represented by connecting the goals and preconditions of operators with their relevant operators successively [Etzioni 1990]. However, the difference is that an F-tree summarizes consequences of different sets of bindings of each single operator observed in a failure example, while it is not feasible for STATIC to consider the implications of all possible bindings of operators in PSG without guidance from examples. The F-tree of Figure 4 (b) specifies the relative location of the target object *bx* (dynamic attribute) as one of the reasons for the observed violation failure. Ordering constraints depending on the dynamic attributes of a problem state are difficult to be detected by problem space analysis as exhibited by STATIC and Cheng's system.

When an F-tree is matched against a new problem, an initial binding is first generated by matching the two goals in the root with the problem goals. This initial binding is then passed down to children nodes for further matching. When a node is matched, the partial bindings passed from its parent are used and extended by matching new variables in the node. When a set of partial bindings generated in a node is passed down through AND branches, all the partial bindings must be successfully completed in every subtree under the branches. If it is passed through OR branches, each partial binding must be completed in at least one subtree under the branches. Otherwise, the matching is considered to have failed. The interpretation of the F-tree of Figure 4 (b) is given in Figure 5 in the self-explanatory PRODIGY's rule format.

```
(and
 (current-node n2)
 (candidate-goal n2 (status dx closed))
 (candidate-goal n2 (inroom bx rx))
 (known n2 (pushable bx))
 (for-all (da ra) such-that
  (known n2 (connects da ra rx))
    (or (is-equal da dx)
        (and (known n2 (not (inroom bx ra)))
             (for-all (rb) such-that
              (known n2 (connects db rb ra))
                (is-equal rb rx))))))
```

Figure 5: The rule condition represented by the F-tree of our example.

The following theorem states that if an F-tree matches the state after achieving the first goal of a problem, then all the attempts to achieve the next goal will confront the same type of failures summarized in the F-tree.

Theorem 1 *Consider a problem with two ordered goals g_a and g_b. If the state after achieving g_a matches an F-tree, then g_b cannot be achieved from that state provided that the given operators are coherent.*[5]

For an F-tree to be more useful, it should be possible to use it even before the first goal is achieved. For this, an F-tree has to be *isolated* as explained in the following. Given a goal g, a subset A of given operators is called the goal achievement closure (*GAC*) of g, if (1) all the operators that can achieve g are in A, (2) all the operators that can achieve any precondition of an operator in A are also in A, and (3) A is the minimal such set. An F-tree with the two goals *pg* and *cg* is said to be *isolated* from *pg* if none of the generalized subgoals in the tree can possibly be achieved by any operator in the *GAC* of *pg*.

Theorem 2 *Consider a problem with the ordered pair of goals g_a and g_b. Suppose an F-tree F with the two goals pg and cg match the initial state of the problem, and the two goals pg and cg match the goals g_a and g_b, respectively. If F is isolated from its pg, then F matches the state after achieving g_a.*

Intuitively, the operators applied for g_a are those in the *GAC* of *pg* which matches g_a, and thus none of them can achieve the subgoals in the F-tree. Then, by Theorem 1, g_b cannot be achieved from the state reached by achieving g_a. If an isolated F-tree matches a problem, we want to change the goal ordering assuming that the conditions of Theorem 1 and Theorem 2 are met. Only the isolated F-trees are qualified as rules.

[5] All the operators of our test domains are coherent. The definition of coherency is omitted for space limitation. Details are explained in [Ryu 1992].

	EBL	STATIC	PAL
Blocks	762/19=40	9.9/18=0.55	4.3/10=0.43
Strips	1057/30=35	40.5/35=1.16	8.7/14=0.62
Sched	1374/37=37	19.4/46=0.42	4.9/20=0.25

Table 1: Learning overhead in cpu seconds divided by the number of control rules derived.

	EBL	STATIC	PAL
Blocks	4	9	10
Strips	12	16	14
Sched	20	19	20

Table 2: Numer of goal ordering rules derived.

Comparison to PRODIGY/EBL and STATIC

PAL was trained and tested on PRODIGY's test domains.[6] The training and test problem sets are based on those used in [Etzioni 1990]. Since the purpose of Minton's and Etzioni's experiments was to test the effectiveness of various types of control rules, their problem set contains many single-goal problems which do not serve our purpose. From both the training and test sets, single-goal problems are replaced by multiple-goal problems to better see the effects of learning and using goal ordering rules. Table 1 shows that PAL spent the shortest average learning time.[7] PRODIGY/EBL's learning time includes the cost of utility evaluation. PAL's learning time includes the overhead of deriving non-isolated F-trees which are abandoned. Since STATIC and PRODIGY/EBL derive rules for other types of search control as well as goal ordering, the above comparison is not really fair. However, we believe that learning goal ordering rules is a more complicated task than learning other types of control rules. Table 2 shows the number of derived goal ordering rules.

We used PRODIGY's planner as a testbed. For efficiency of the planner in other aspects of search such as selection and rejection of operators or bindings, we had the planner equipped with STATIC's control rules excluding the goal ordering rules. We empirically observed that STATIC's rules were in general more effective than PRODIGY/EBL's. The effectiveness of the goal ordering rules derived by each system is then measured by successively loading and testing each set of goal ordering rules. Table 3 compares the test results of running the planner (learning module turned off) without and with different sets of goal ordering rules. The planner was observed to run more efficiently with more effective goal ordering rules, and to generate better (shorter) solutions with less number of expanded nodes.

ordering rules	cpu secs	nodes	sol. length
none	1224	4587	529 (59)
EBL	1044	3551	499 (60)
STATIC	590	1832	495 (63)
PAL	587	1831	495 (63)

(a) 100 Schedworld problems

ordering rules	cpu secs	nodes	sol. length
none	109	5929	1009
EBL	90	4536	899
STATIC	74	3416	925
PAL	63	2601	837

(b) 100 Blocksworld problems

ordering rules	cpu secs	nodes	sol. length
none	622	16309	2695
EBL	523	13806	2656
STATIC*	591	15599	2693
STATIC	748	19632	2769
PAL	492	13317	2618

(c) 100 Stripsworld problems

Table 3: Effects of different sets of goal ordering rules.

The Schedworld is distinguished from the other domains in that problem solving generates very shallow search trees. STATIC performed well in this domain because the ordering constraints of the domain can be captured by operator analysis due to the shallowness. The solution lengths are for the 59 problems which were solvable within resource limits with each of the rule sets. The numbers in parentheses are the numbers of solved problems. In Blocksworld and Stripsworld, PAL's rules performed most effectively. STATIC's rules were not as effective as PAL's in Blocksworld because the current version of STATIC fails to derive the rule preferring $(on\ y\ z)$ to $(on\ x\ y)$, which are goals frequently seen in this domain. In Stripsworld, STATIC generated a few over-general rules (four out of sixteen) including those mentioned earlier. The effect of over-general rules, given a set of test problems, can be either beneficial or harmful depending on the distribution of the problems. In Table 3 (c), STATIC* denotes STATIC rules excluding the over-general rules.

Related Work

The SteppingStone system [Ruby and Kibler 1991] learns subgoal sequences which are used *after* failures are observed, while PAL's rules are used *before* problem-solving to avoid failures. ALPINE [Knoblock 1990] learns abstraction hierarchies for planning by analyzing *possible* goal interactions. PAL and STATIC learn only from *necessary* goal interactions. ALPINE

[6] Cheng's system [Cheng 1991] is excluded from this comparison because its variabilization method for the derived goal ordering constraints is not implemented. In a separate experiment, PAL's learning outperformed Cheng's pre-processing [Ryu 1992].

[7] Since PAL was run on a different machine (Sun Sparcstation), we re-scaled our original data (3.1, 6.2, and 3.5) based on the comparison of the data given in [Etzioni 1990] with the data obtained by running the same set of sample problems on our machine. Note, however, that data in Table 3 are original, not re-scaled.

is closer to Cheng's method in that both are problem specific methods. Currently, it cannot generalize an abstraction hierarchy for later use in similar problems. Kambhampati's validation structure based approach [Kambhampati 1990] deals with generalizing reusable plans in hierarchical planning. Kambhampati's other method for generalizing partially ordered plan [Kambhampati and Kedar 1991] uses the Modal Truth Criterion [Chapman 1987] to analytically generalize interaction-free plans. However, the generalized plans of both these approaches are described in terms of operators rather than goals. The goal ordering rules of PAL can be viewed as plans at higher level of abstraction with broader applicability.

Conclusion

We have presented a learning algorithm which is both efficient and effective for deriving sufficient conditions for ordering conjunctive goals. We have shown that some complicated relational concepts representing conditions for search control can be derived by a relatively simple method using a minimal amount of information collected from a search tree, namely the failure goal stacks. Unlike other systems, PAL does not rely on any explicit *a priori* domain-specific knowledge which is often difficult to provide. One of the drawbacks of PAL's approach is that the generality of the rules it derives is sometimes limited by the specifics of the training problem instances, while STATIC and Cheng's system do not have such dependency. However, PAL can learn rules whose conditions depend on the dynamic attributes of a problem state, which is beyond the scope of those two systems. Moreover, PAL does not suffer from the problems of conflict or over-generality observed with PRODIGY and STATIC.

Acknowledgments

Authors are grateful to the PRODIGY group at Carnegie Mellon University for providing us with the PRODIGY system. Thanks also go to Steve Minton and Oren Etzioni for providing their data and giving helpful hints for testing their systems.

References

Chapman, D. 1987. Planning for Conjunctive Goals. *Artificial Intelligence* 32:337-377.

Cheng, J. and Irani, K.B. 1989. Ordering Problem Subgoals. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, 931-936. Detroit, Michigan.

Cheng, J. 1991. Deriving Subgoal Ordering Constraints Prior to Problem Solving and Planning. Ph.D. diss., EECS Dept., The University of Michigan.

Dawson, C. and Siklòssy, L. 1977. The Role of Preprocessing in Problem Solving Systems: "An Ounce of Reflection is Worth a Pound of Backtracking." In *Proceedings of the Fifth International Joint Conference on Artificial Intelligence*, 465-471.

Etzioni, O. 1990. A Structural Theory of Explanation-Based Learning. Ph.D. diss., School of Computer Science, Carnegie Mellon University.

Etzioni, O. 1991. STATIC: A Problem-Space Compiler for Prodigy. In *Proceedings of the Ninth National Conference on Artificial Intelligence*, 533-540.

Kambhampati, S. 1990. Mapping and Retrieval During Plan Reuse: A Validation Structure Based Approach. In *Proceedings of the Eighth National Conference on Artificial Intelligence*, 170-175.

Kambhampati, S. and Kedar, S. 1991. Explanation-Based Generalization of Partially Ordered Plans. In *Proceedings of the Ninth National Conference on Artificial Intelligence*, 679-685.

Knoblock, C. 1990. Learning Abstraction Hierarchies for Problem Solving. In *Proceedings of the Eighth National Conference on Artificial Intelligence*, 923-928.

Minton, S. 1988. Qualitative Results Concerning the Utility of Explanation-Based Learning. In *Proceedings of the National Conference on Artificial Intelligence.* St. Paul, MN.

Minton, S. 1989. *Learning Search Control Knowledge: an Explanation-Based Approach.* Kluwer Academic Publishers.

Newell, A., Shaw, J.C. and Simon, H.A. 1960. Report on a General Problem-Solving Program. In *Proceedings of International Conference on Information Processing*, 256-264. Paris, France.

Ruby, D. and Kibler, D. 1991. SteppingStone: An Empirical and Analytical Evaluation. In *Proceedings of the Ninth National Conference on Artificial Intelligence*, 527-532.

Ryu, K.R. 1992. Learning Search Control Knowledge for Planning with Conjunctive Goals. Ph.D. diss., EECS Dept, The University of Michigan.

Sussman, G.J. 1975. *A computer model of skill acquisition.* New York: American Elsevier.