Failure Driven Dynamic Search Control for Partial Order Planners: An Explanation based approach

Subbarao Kambhampati [1] and Suresh Katukam and Yong Qu

Department of Computer Science and Engineering, Arizona State University Tempe, AZ 85287-5406

Abstract

Given the intractability of domain-independent planning, the ability to control the search of a planner is vitally important. One way of doing this involves learning from search failures. This paper describes SNLP+EBL, the first implementation of explanation based search control rule learning framework for a partial order (plan-space) planner. We will start by describing the basic learning framework of SNLP+EBL. We will then concentrate on SNLP+EBL's ability to learn from failures, and describe the results of empirical studies which demonstrate the effectiveness of the search-control rules SNLP+EBL learns using our method.

We then demonstrate the generality of our learning methodology by extending it to UCPOP [39], a descendant of SNLP that allows for more expressive domain theories. The resulting system, UCPOP+EBL, is used to analyze and understand the factors influencing the effectiveness of EBL. Specifically, we analyze the effect of (i) expressive action representations (ii) domain specific failure theories and (iii) sophisticated backtracking strategies on the utility of EBL. Through empirical studies, we demonstrate that expressive action representations allow for more explicit domain representations which in turn increase the ability of EBL to learn from analytical failures, and obviate the need for domain specific failure theories. We also explore the strong affinity between dependency directed backtracking and EBL in planning.

[1] Corresponding author's Fax: (602) 965-2751, E-mail: rao@asu.edu., WWW: http://rakaposhi.eas.asu.edu:8001/yochan.html

1 Introduction

Domain independent planners come in two main varieties -- state-space planners that search in the space of world states, and partial order (plan-space) planners that search in the space of partial plans. Several recent studies demonstrate that searching in the space of plans provides a more flexible and efficient framework for planning [1,34]. Despite their many perceived advantages, plan-space planners are not a panacea for computational intractability of domain-independent planning. In particular, it is widely realized [21,47,40,26] that effective search control is critically important for getting efficient planning capabilities out of these planners.

A promising way of controlling the search of a planner is to let the planner learn dynamically from its own failures. There are two complementary ways of doing this. First, the planner can use the information about the failure to decide what part of the search tree to backtrack to. Second, and perhaps more ambitious, it can also learn to avoid similar failures in the future. Both these capabilities can be provided by the general analysis of explanations of the failures encountered by the planner. Explanation based learning techniques (EBL), studied in machine learning [35,10,32], offer significant promise in this direction.

The general idea behind explanation based learning (see Figure 1) is as follows: given a problem the planner searches through the space of possible solutions and returns a solution. The learner then explains the failures and successes in the search tree explored by the planner and, uses these explanations to generate search control rules that may be used to guide the planner to avoid the failing paths and bias it toward the successful paths. The performance of the planner may thus be improved by the use of these learned rules.

Although there has been a considerable amount of research towards applying EBL techniques to planning, almost all of it concentrated on the state-based models of planning, as against the partial-order (plan space) models of planning [31,2]. One of the reasons for the concentration of *explanation based learning* (EBL) work on state-space planners has been the concern that a sophisticated planner may make the learning component's job more difficult (c.f. [32]). However, given the current status of plan-space planning as the dominant planning paradigm, it is important to adapt the speed-up learning techniques to the plan-space planners. In this paper we present an explanation based learning framework for a partial order plan-space planner, that is both clean and elegant. We also show that the framework is capable of significantly improving the performance of a plan-space planner.

First, we will describe SNLP+EBL [29,28], a system that learns search control rules for SNLP, a causal-link partial order planner [30,1]. Learning is initiated whenever the planner detects a failure or crosses the depth limit. In either case, SNLP+EBL explains the failure by isolating a minimal subset of the constraints on the partial plan

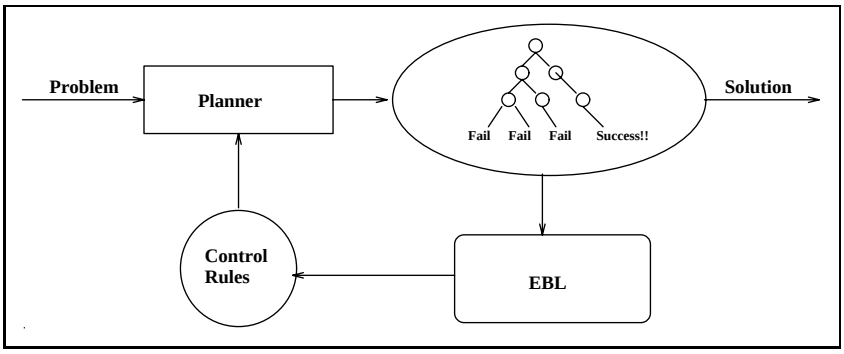


Fig. 1. EBL in Planning

that are together inconsistent. This explanation is then regressed over the planner decisions, and propagated up to compute explanations at the ancestors of the failing plan that are predictive of the failure. These explanations are then generalized and converted into search control rules to avoid similar failures. The regression process is facilitated by formally characterizing all the planner decisions in terms of the constraints that should be present in the partial plan for them to be applicable, and the constraints that are added by them. Apart from facilitating control rule learning, the regression and propagation processes also form the basis for a powerful and efficient form of dependency directed backtracking for the planner. We describe the details of the construction of failure explanations, regression, propagation, control rule generation and rule generalization. We will also discuss the soundness of the control rules generated by SNLP+EBL, and empirically demonstrate their effectiveness in improving the efficiency of the planner.

Although our empirical studies with SNLP+EBL show that control rule learning is an effective way of improving the performance of a plan space planner, they also bring up the critical dependencies between the effectiveness of control rule learning, and a variety of other factors, including the expressiveness of the action representation used by the planner, the types of backtracking strategies being employed by the planner as well as the types of goal selection heuristics used. For example, in our empirical studies with SNLP+EBL, we found that it sometimes had to rely on the presence of a domain specific theory of failure (usually in the form of domain axioms). This brings up the importance of domain representation, and poses the question as to whether a more expressive action description language might allow richer domain representations and thereby reduce the dependence on outside failure theories. Similarly, we noted that the analysis done in learning control rules is very similar to the analysis required to support dependency directed backtracking. Since dependency directed backtracking itself can improve the performance of planning to some extent, it is important to understand how it interacts with the use of learned control rules.

To facilitate the analysis, we started by extending our control rule learning framework to UCPOP, a partial order planner that is can handle a larger class of planning domains including those that contain actions with conditional and

quantified effects, as well as quantified and disjunctive preconditions. The resulting system, UCPOP+EBL [44,42], is used here as the basis for a systematic empirical investigation of the effect of (i) expressive action representations (ii) domain specific failure theories and (iii) sophisticated backtracking strategies on the utility of EBL. We will show that expressive representations allow us to make the relations between the preconditions and effects of the actions more explicit, thereby increasing the effectiveness of learning control rules from analytical failures alone. This in turn reduces the need for domain specific failure theories to guide EBL. We will also demonstrate the strong affinity between dependency directed backtracking and explanation based learning, and clarify as to when EBL can provide savings over and above those offered by dependency directed backtracking.

1.1 Overview

The rest of this paper is organized as follows. Section 2 provides an overview of the architecture of SNLP+EBL. Section 3 reviews the SNLP planning algorithm. Section 4 is the heart of this paper and describes the EBL framework that is used in SNLP+EBL: it classifies the failures encountered by SNLP during the planning process, and describes how explanations are constructed for them. Section 4.2 describes how failure explanations are regressed over planning decisions to propagate the failure explanations to ancestor levels. Section 4.3 explains the propagation process that is used to collect explanations that are emerging from various refinements of a partial plan and take the conjoined explanation up the search tree. It also explains how search control rules are constructed from failure explanations. Section 5 discusses the issues regarding the soundness of the search control rules learned by SNLP+EBL. Section 6 describes how learning from analytical failures alone is sometimes not sufficient, and it describes a novel strategy for learning from depth-limit failures using domain axioms. Section 7 describes the experiments conducted to evaluate the effectiveness of search control rules learned by SNLP+EBL. Section 8 discusses the extensions needed to adapt the explanation-based learning framework to UCPOP, and Section 9 describes an empirical evaluation of the performance of UCPOP+EBL. Section 10 describes the results of a focussed empirical study to analyze the factors affecting the performance of UCPOP+EBL. Section 11 discusses the related work. Finally, Section 12 presents our conclusions, and discusses possible future directions. Appendix A provides the list of symbols used in the paper along with a short description of their intended meanings.

One final note about the organization is in order. Since there is a wide variation in the EBL literature in terms of terminology and architectures used, and since the planning researchers may not be aware of much of this terminology, in this paper, we will attempt to provide a self-contained description. Readers already familiar with EBL techniques may thus find some of the exposition redundant.

2 Architecture of the SNLP+EBL system

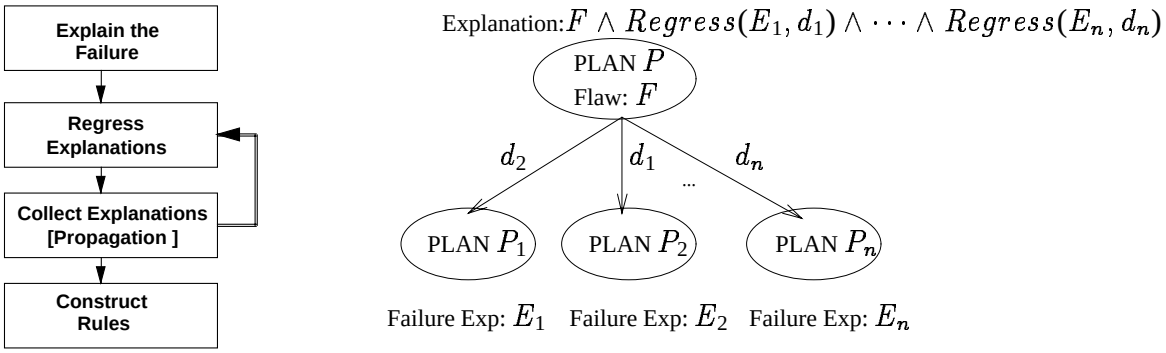
In this section we will provide a broad overview of our control rule learning framework. The SNLP+EBL system consists of two main components: the plan space partial order planner, SNLP, and the learning component for doing EBL. Search consists of selecting a partial plan from the search queue, and refining it by considering a flaw (an unachieved subgoal, or a violation of a previous commitment; see below) in the partial plan, and generating children plans corresponding to all possible ways of resolving the flaw. A partial plan is said to be a solution to the planning problem when it contains no unresolved flaws. The planner does a depth-limited depth first search. [2] During the learning phase, SNLP+EBL invokes the learning component whenever the planner encounters a failure. Figure 2 shows a schematic of the EBL process.

A failure is encountered when SNLP produces a partial plan that cannot be refined further, or the search crosses the depth limit. Once a failure is encountered, SNLP+EBL generates an initial explanation from the failed partial plan. The explanation of failure is a subset of constraints on the partial plan that are together inconsistent (in that as long as they hold, the partial plan cannot be refined into a solution). The failure explanations of the leaf nodes are back-propagated to the ancestor nodes by regressing them over the planner decisions that led to the failing nodes. A partial plan at an intermediate node of the search tree is considered to be failing if all the branches below it are failing. The explanation of failure for the partial plan is the conjunction of the regressed explanations of failures of all its children nodes, along with the description of flaw that they were attempting to resolve (see Figure 2(b)).

Learning is done incrementally and in-step with planning. Specifically, whenever, a partial plan fails, its failure explanation is constructed, regressed over the decision leading to it, and stored at the parent plan. Search resumes at the other unexplored children of the parent plan. When all the children are explored and are found to be failing, the explanation of failure of the parent node is constructed and propagated up to its parent node.

Given the explanation of failure of a partial plan, a search control rule can be generated that recommends rejecting *any* partial plan produced in the future, if it satisfies the failure explanation. To increase the coverage of the rule, the steps and objects comprising the failure explanation are generalized (so that other steps and objects can take their place), without leading to a loss of completeness. The search

[2] Note that the standard implementations of both SNLP and UCPOP use best first search, with a user given heuristic. We implemented depth-first search, along with the ability to EBL and dependency directed backtracking, as described in this paper. Note also that although depth first search is useful for learning search control rules in the first place, once learned, the rules can be used in non-depth-first search regimes also.



(a) Schematic of SNLP+EBL

(b) Computing failure explanation of a plan given the failure explanations of its children

Fig. 2. Schematic overview of control rule learning process in SNLP+EBL and UCPOP+EBL

control rules thus generated are used to improve either the remaining search of the current problem, or the search in the future problems.

In addition to generating the control rules, the regression and propagation processes also facilitate a general and efficient way of dependency directed backtracking, which can significantly improve the performance of the underlying planner. In the next two sections, we discuss the details of the planning and learning processes.

3 The base level planner : SNLP

SNLP is a partial order planner that uses causal links to keep track of its commitments [30,1]. Given a planning problem in terms of a completely specified initial state $\mathcal{I}$, a set $\mathcal{G}$ of goals, and a set of legal operators/actions in the domain, the objective of SNLP is to come up with a sequence of operators drawn from the given set, which, when executed from the initial state, give rise to a state where the conditions specified in $\mathcal{G}$ are all true. The domain operators are specified in the STRIPS operator formalism [36], involving precondition, add and delete lists. Typically, a family of operators are specified compactly as an operator schema, such that any instance of the schema gives rise to a legal operator. Figure 3 shows the example of an operator schema, $Puton(x, y, z)$ in blocks world. In some planners, the add and delete lists are combined into a single list of effects (with the delete list literals appearing negated). For a specification of $Puton(x, y, z)$ in this way, see Figure 16.

SNLP searches in the space of partial plans. Roughly speaking, partial plans are a partially ordered set of actions. SNLP starts with a null plan that consists of a dummy initial and final steps. The effects of the initial step correspond to conditions

Puton (x from y to z)

Preconditions : $Clear(z), Clear(x), On(x, y)$
Add : $On(x, z), Clear(y),$
Delete: $Clear(z), On(x, y)$
Bindings: $(z \not\approx Table), (x \not\approx z)$

Fig. 3. A blocks world operator schema

true in the initial state of the problem. Similarly, the preconditions of the final step correspond to the conditions required in the goal state of the problem. The planning process consists of repeatedly picking up a precondition of a step and ''establishing'' it (making it true), or alternatively resolving conflicts between different establishments. Both establishment and conflict resolution operations are ''refinement'' operations in that they only add further constraints to the partial plan, without deleting any existing constraints.

3.1 Plan Representation

Formally, each partial plan in SNLP is a 7-tuple:

$$\langle \mathcal{S}, O, \mathcal{ST}, \mathcal{B}, \mathcal{L}, \mathcal{E}, \mathcal{C} \rangle$$

where:

-- $\mathcal{S}$ is the set of actions (step-names) in the plan; $\mathcal{S}$ contains two distinguished step names s_0 and s_∞, called the initial and final step of the plan respectively (in the paper s_0 is also referred to as the step 0 or step I, and the s_∞ is also referred to as G).
-- $\mathcal{ST}$ is a symbol table, which maps step names to domain operators. The special step s_0 is always mapped to the dummy operator `start`, and similarly s_∞ is always mapped to `fin`. The formal reason $\mathcal{ST}$ needs to be separate from $\mathcal{S}$ is to allow for the fact that more than one step in the plan may correspond to the same action. However, for simplicity, we will often omit $\mathcal{ST}$ field from the plan specification in this paper.
-- $\mathcal{O}$ is a partial ordering relation, representing the ordering constraints over the steps in $\mathcal{S}$. Ordering constraints between steps are denoted by the relation ''$\prec$''. For example, $(s_1 \prec s_2) \in \mathcal{O}$ means that the step s_1 is constrained to precede s_2 in the plan. Apart from the direction ordering constraints, $\mathcal{O}$ is also implicitly assumed to contain all the transitive ordering constraints that follow from them. Thus, given two ordering constraints $s_1 \prec s_2$ and $s_2 \prec s_3$ that belong to $\mathcal{O}$, the ordering relation $s_1 \prec s_3$ is considered to be present in $\mathcal{O}$.
-- $\mathcal{B}$ is a set of codesignation (binding) and non-codesignation (prohibited bindings) constraints on the variables appearing in the preconditions and postconditions of the operators. The codesignation constraints between variables x and y are

denoted as $x \approx y$ while the non-codesignation constraints between variables are denoted $x \not\approx y$. The notation $mgu(p, q)$ stands for the minimal set of variable bindings that are needed to make the conditions p and q necessarily codesignate. For example, given a partial plan with empty binding set, $mgu(On(x, y), On(u, B)) = (x \approx u) \wedge (y \approx B)$. Like the set of ordering constraints, all the codesignation binding constraints that transitively follow from those present in $\mathcal{B}$ are also implicitly assumed to belong to $\mathcal{B}$.

-- $\mathcal{E}$ is the set of effects of the plan, i.e., tuples $s \xrightarrow{e}$ such that $s \in \mathcal{S}$ and e belongs to the add list (or $\neg e$ belongs to the delete lis)t of the operator corresponding to s. The assertions true in the initial state of the planning problem are treated as the effects of the initial step. We use the special notation `initially–true`(e) to denote that e is an effect of the initial step (i.e., `initially–true`(e) $\equiv s_0 \xrightarrow{e}$).

-- $\mathcal{C}$ is the set of preconditions of steps of the partial plan, i.e, tuples $c@s$ such that c is a precondition of step $s \in \mathcal{S}$ (or to be more precise, the preconditions of the operator corresponding to step s). The assertions in the goal state of the planning problem are treated as the preconditions of the final step s_∞ of the plan.

-- $\mathcal{L}$ is a set of causal links of the form $s \xrightarrow{p} w$ where $s, w \in \mathcal{S}$ and p is an effect of s and a precondition of w. The steps s and w are called, respectively, the source and the destination (or alternately producer and consumer) of the causal link. The causal link $s \xrightarrow{p} w$ is said to *support* the precondition constraint $p@w$. This constraint is satisfied as long as ''s comes before w and gives p to w, and no step in the plan that can possibly come in between s and w is allowed to delete p.'' [3]

A *solution* to the planning problem is a sequence of actions, which when executed from the initial state, results in a state of the world that satisfies all the goals. A partial plan is best understood as a shorthand notation for the set of action sequences that are consistent with the constraints specified in the partial plan. A partial plan is said to be *complete* (i.e., planning can terminate on it) if every topological sort of the partial plan is a solution to the planning problem. The steps, effects, bindings, orderings, causal links, and preconditions can all be seen as constraints on the partial plan in that they constrain the set of solution plans consistent with the partial plan. (For more on the semantics of partial plans, see [25]).

[3] The original description if SNLP [30] also ensures that no intervening step can delete *all* the condition supported by the causal link. This is however not required for completeness [25].

3.2 The planning process

SNLP starts its planning process with the ‘‘null’’ plan

$$\left\langle \begin{array}{c} \mathcal{S}\colon\{s_0, s_\infty\}, \mathcal{O}\colon\{s_0 \prec s_\infty\}, \mathcal{C}\colon\{g_i@s_\infty | g_i \in goal\ state\}, \\ \mathcal{E}\colon\{s \xrightarrow{e} | e \in initial\ state\}, \mathcal{L}\colon\emptyset \end{array} \right\rangle$$

where $\mathcal{C}$ is initialized with the top level goals of the problem (which, by convention are the preconditions of s_∞, initial state conditions are the effects of s_0).

The goal of planning is to add binding constraints to the null plan until it becomes a complete plan (as defined earlier). This process is called plan refinement. To guide the refinement, the incompleteness of a partial plan is characterized in terms of entities called ‘‘flaws’’. Flaws can be seen as the symptoms of incompleteness of the plan; when all the flaws are resolved, the plan becomes complete. There are two types of flaws. The plan is said to contain an *open condition* flaw, if it contains a precondition constraint, that is not supported by any causal link. It is said to contain an *unsafe link flaw* if it contains a causal link constraint, and a step (called a threat) that can possibly come between the producer and consumer of the causal link and delete the condition being supported by the causal link.

It is important to note that flaws can be formally stated in terms of the constraints on the partial plan. For example, an open condition flaw involving the precondition $p@s$ can be described as:

$$p@s \in \mathcal{C} \wedge \nexists_{s' \in \mathcal{S}}\ s' \xrightarrow{p} s \in \mathcal{L}$$

Similarly, an unsafe causal link flaw involving a link $s' \xrightarrow{p} s$ and a threatening step s_t can be described as:

$$s' \xrightarrow{p} s \in \mathcal{L} \wedge s_t \xrightarrow{\neg q} \in \mathcal{E} \wedge mgu(p, q) \in \mathcal{B} \wedge (s_t \prec s') \notin \mathcal{O} \wedge (s \prec s_t) \notin \mathcal{O}$$

It can be easily shown that a partial plan that contains no open condition flaws or unsafe link flaws is complete (i.e., all of its topological sorts correspond to solutions). The planning process thus consists of selecting a flaw from an incomplete partial plan and resolving it by adding constraints to the partial plan. A flaw gets resolved when the constraints comprising its description are no longer true. Notice that since flaw descriptions contain ‘‘$\notin$’’ constraints, flaws can be resolved by adding constraints to the plan.[4]

[4] From a formal view point, this differentiation between flaws and ordinary partial plan constraints is important. In refinement planning, constraints on a partial plan never go

If the flaw is an open condition $c@s$, SNLP establishes it by either using an effect q of an existing step (*simple establishment*) or by a newly introduced step s_e (*step addition*). In either case, the $\mathcal{O}$ and $\mathcal{B}$ fields of the partial plan are updated to make s_e precede s, and q codesignate with c. Finally, to remember this particular establishment commitment, a causal link of the form $s_e \xrightarrow{c} s$ is added to $\mathcal{L}$. In addition to this, in the case of step addition, the $\mathcal{E}, \mathcal{C}$ and $\mathcal{B}$ fields of the partial plan are updated with the effects, preconditions and binding lists of the new step. Notice that in either case the specific flaw will no longer be present since the causal link is added by the establishment operation.

If the flaw is an unsafe link involving the causal link $s \xrightarrow{p} w$ and the threatening step s_t, it is resolved by either *promoting* s_t to come after w, or *demoting* it to come before s (in both cases, appropriately updating O).[5] In either case, the flaw will not be present in the partial plans after the promotion/demotion refinement, since the threat is no longer unordered with respect to the producer and consumer step. A threat for a causal link is said to be *unresolvable* if both these possibilities make either $\mathcal{O}$ or $\mathcal{B}$ inconsistent.

SNLP does not have to backtrack over the selection of a flaw to maintain completeness, but has to backtrack over the different ways of resolving the flaw (e.g. it has to consider all possible establishment options for each open condition, and both promotion and demotion options for a threat).

We can now summarize the various types of decisions taken by SNLP:

-- If the flaw is an open condition, possible decisions are:
  - *Simple Establishment:* Establish the condition by using an effect of an existing step in the partial plan.
  - *Step Addition:* Establish the condition by selecting an operator in the domain which can give that condition, and introducing that step into the plan.
-- If the flaw is an unsafe link, possible decisions are:
  - *Promotion:* Order the threatening step to come after the consumer of the causal link.
  - *Demotion:* Order the threatening step to come before the contributor of the causal link.

All these decisions are ‘‘refinements’’ to the partial plan in the sense that they add additional constraints to the partial plan, without removing any existing constraints. It is straightforward to formalize these decisions as STRIPS-type operators whose

away, they only accumulate. However, ‘‘flaws’’ do go away during planning. This can be reconciled by the fact that flaws go away as constraints are added.

[5] Readers familiar with SNLP algorithm in [1] will note that by defining a threat in terms of necessary codesignation, rather than possible codesignation, we obviate the need for *separation* as a way of resolving unsafe links. Empirical studies [40] show that this strategy tends to improve the efficiency of SNLP.

Demote($s_1 \xrightarrow{\neg p''}, s_2 \xrightarrow{p'} s_3$)

Resolve the conflict between the link $s_2 \xrightarrow{p'} s_3$ and the effect p'' of step s_1

Preconditions: $s_2 \xrightarrow{p'} s_3 \in \mathcal{L}$

$s_1 \xrightarrow{\neg p''} \in \mathcal{E}$

$mgu(p', p'') \in \mathcal{B}$

$(s_1 \prec s_2) \notin O$

$(s_3 \prec s_1) \notin O$

Effects: $O \leftarrow O + (s_1 \prec s_2)$

Fig. 4. Demotion decision in STRIPS representation

Action	**Precond**	**Add**	**Dele**
Roll(ob)	-	Cylindrical(ob)	Polished(ob) $\wedge$ Cool(ob)
Lathe(ob)	-	Cylindrical(ob)	Polished(ob)
Polish(ob)	Cool(ob)	Polished(ob)	-

Fig. 5. Description of a simple job-shop scheduling domain

preconditions and effects are stated in terms of the constraints on the partial plan. For example, *demotion* decision can be formally represented as shown in the Figure 4. The preconditions of the decision consist of the set of constraints on the plan that give rise to an unsafe link flaw. The effect of the decision is to add a new ordering constraint to the partial plan. In Section 4.2, we will see that this view of planning decisions will be useful in understanding the regression process.

Example: We shall now illustrate SNLP's planning algorithm with an example from a simplified job-shop scheduling domain (which will be used as a running example throughout the discussion of SNLP+EBL). The operators describing this domain are shown in Figure 5 The shop consists of several machines, including a lathe and a roller that are used to reshape objects, and a polisher which is used to polish the surface of a finished object. Given a set of objects to be polished, shaped, etc., the planner's task is to schedule the objects on the machines so as to meet these requirements.

Our planning problem is to polish an object A and make its surface cylindrical. A's temperature is cool in the initial state. Figure 6 shows the complete search tree for the problem. SNLP starts with the null plan, and picks up the open condition flaw $Cylindrical(A)@G$. This flaw is resolved by adding the step 1: `Roll`(A) which has an effect $Cylindrical(A)$. SNLP then resolves the other open condition flaw $Polished(A)@G$ with the step 2: `Polish`(A). Since the step 1: `Roll`(A), deletes

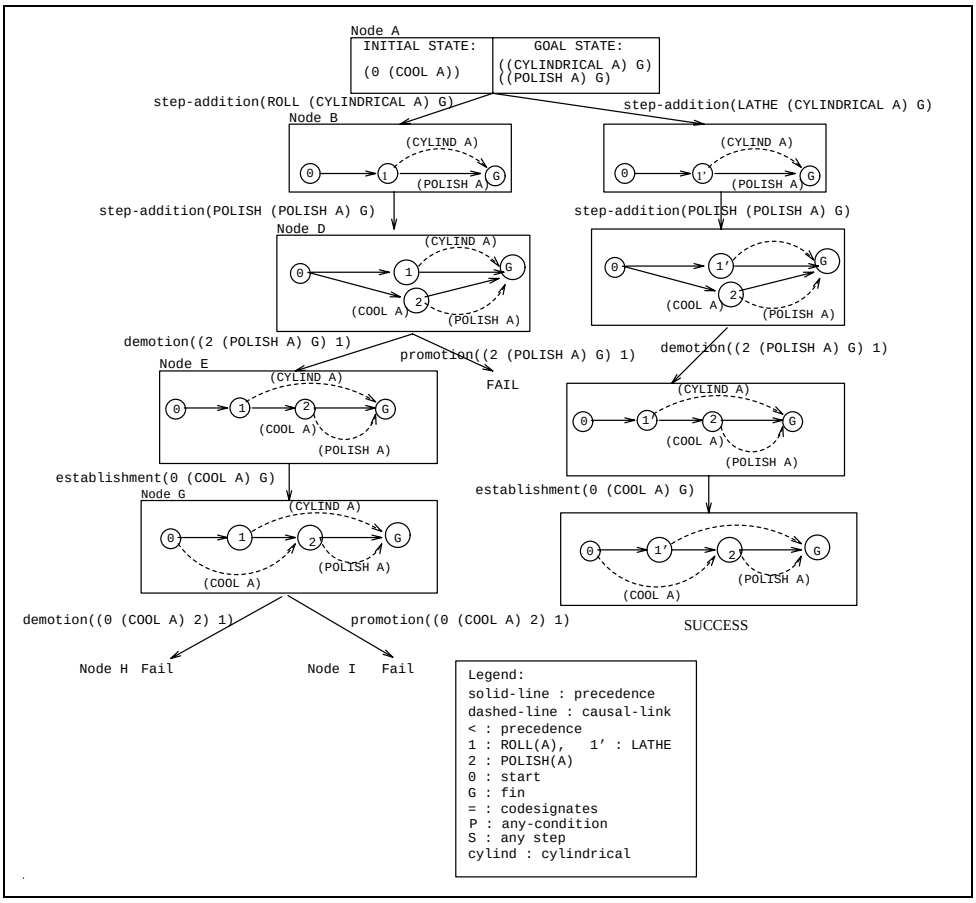


Fig. 6. Search Tree illustrating SNLP planning process. The figure uses a lisp like notation for the plan constraints. Causal link constraints are shown as three element lists, and open conditions and preconditions are shown as two element lists.

$Polished(A)$, it is now a threat to the link 2 $\overset{Polished(A)}{\rightarrow}$ G. This threat is resolved by demoting step 1: `Roll`(A) to come before 2: `Polish`(A). The step 2: `Polish`(A) also introduces a new open condition flaw $Cool(A)$@2. SNLP establishes it using the effects of the initial step 0. Since `Roll`(A) also deletes $Cool(A)$, it threatens this last establishment. When SNLP tries to resolve this threat by demoting step 1 to come before step 0, it fails, since 0 already precedes 1. SNLP backtracks chronologically until the point where it has unexplored alternatives -- node A in this example and explores other possible alternative. It achieves $Cool(A)@G$ using `Lathe`(A) and then achieves $Polished(A)$ using the operator `Polish`(A). It succeeds in this path and returns a solution.

4 Explanation based learning

As illustrated in the job shop scheduling example, when SNLP encounters an inconsistent plan, it considers the search branch passing through that partial plan

to be failing, and backtracks. SNLP+EBL analyzes the failure and generates search control rules that guide the planner to avoid similar failures in future. A search control rule may either be in the form of a selection rule or a rejection rule. Since SNLP+EBL concentrates on learning from failures, it focuses on learning rejection rules.

Search control rules aim to provide guidance to the underlying problem solver at critical decision points. As we have seen, for SNLP these decision points are the selection of flaws (open conditions, unsafe links), establishment choice, including simple-establishment and step-addition (operator selection); threat selection; and threat resolution, including promotion, demotion. Of these, it is not feasible to learn goal-selection and threat-selection rules using the standard EBL analysis since SNLP does not backtrack over these decisions.[6] SNLP+EBL learns search control rules for the other decisions.

In this section, we will describe the process of control rule learning in SNLP+EBL, starting with the initial detection and explanation of failures, and continuing through regression and propagation of leaf node failure explanations to intermediate nodes, and finally learning and generalizing search control rules from these failure explanations.

4.1 Failures and Initial Explanation Construction

SNLP+EBL flags a partial plan to be a failing plan in three situations:

Analytical Failures: These failures are flagged when the planner detects ordering or binding inconsistencies in the partial plan, or finds that there are no establishment choices for resolving an open condition flaw.

Depth Limit Failures: These failures are flagged whenever the search crosses the depth-limit. The idea here is to stop the planner from searching in fruitless paths.

Search Control Rule Failures: These failures are flagged whenever a (previously learned) search control rule rejects the search branch.

In each case, SNLP+EBL attempts to ‘‘explain’’ the failure by listing a set of constraints on the failing partial plan that are together inconsistent. This constraint set is called the *explanation of the failure* of the partial plan. Failure explanations constructed this way are ‘‘*sound*’’ in that partial plans that contain these constraints can never be refined into a solution for the problem.

[6] This doesn’t however mean that threat selection and goal selection order do not affect the performance of the planner. It merely means that the best order cannot be learned through failure based analysis.

We will start by describing the failure explanations for various types of analytical failures:

Ordering failures: These arise when there is a cycle among the orderings of two steps in a partial plan. For example, whenever two steps s_1, and s_2 are ordered such that $(s_1 \prec s_2) \in \mathcal{O} \wedge (s_2 \prec s_1) \in \mathcal{O}$, then this represents an inconsistency in the partial plan. The explanation for the ordering inconsistency is simply the conjunction of inconsistent constraints: $(s_1 \prec s_2) \wedge (s_2 \prec s_1)$ (for simplicity, we avoid mentioning that these constrains are present in the $\mathcal{O}$ part of the partial plan; $s_1 \prec s_2$ is understood as the constraint $(s_1 \prec s_2) \in \mathcal{O}$).

Binding failures: These arise when there is an inconsistency in the bindings; for example, if there exists a variable x in the partial plan such that $(x \approx A) \wedge (x \not\approx A)$. The explanation for this binding inconsistency is: $(x \approx A) \wedge (x \not\approx A)$

Establishment Failures: This failure occurs when the planner encounters an open condition $p@s$ in a partial plan P, which has no simple establishment or step-addition possibilities.[7] In this case, the failure explanation needs to capture two facts: (i) there are no operators in the domain which can give the condition p and (ii) p is not true in the initial state. Of these two, the first clause will remain true even if we change to a new problem (since the domain specification remains the same), and thus does not have to be made part of the failure explanation. In contrast, the second clause may not remain true in a new problem (since the initial state may change from problem to problem), and should thus be a part of the failure explanation. Accordingly, the explanation of failure given by SNLP+EBL for P will be: $p@s \wedge \neg$`initially–true`(p), where `initially–true`(p) is true if p is part of the initial state of the problem.[8]

In contrast to analytical failures, failures flagged at depth limits do not have direct failure explanations. Simply saying that the plan crossed the depth limit does not suffice, and we need to isolate the subset of constraints on the plan that may together be inconsistent. Sometimes, this can be done by analyzing the partial plan at the depth limit with respect to a set of strong consistency checks. In section 6, we will explain an instance of this strategy which uses domain-axiom based consistency checks to explain the implicit failures at depth limits, and construct explanations for these failure.

Finally, SNLP+EBL may reject a specific search branch outright because a previously learned search control rule recommends rejecting this branch. In such a case, the antecedent of the control rule serves as the explanation of failure.

[7] Note that not having establishment possibilities is different from having establishment possibilities all of which eventually end up in failing plans.

[8] Note that it is possible that even if p were true in the initial state, we may still not have been able to use it to establish p at s (perhaps due to interactions with other goals). Thus, by adding $\neg$`initially–true`(p), we may sometimes be taking a failure explanation that is more specific than necessary; see Section 5 for further discussion on this.

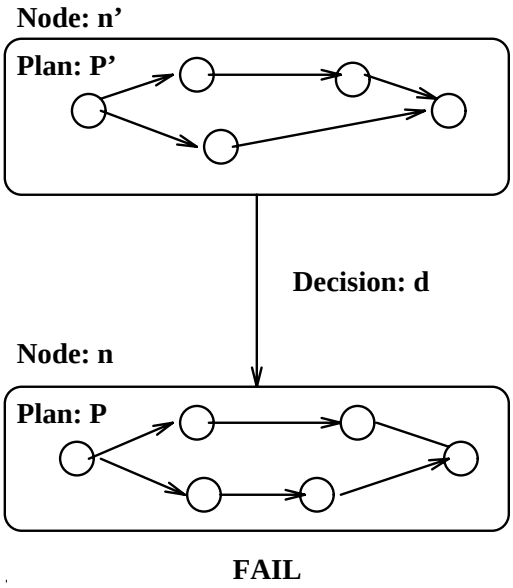


Fig. 7. A part of a failure branch to explain the regression process

4.2 Regression

Consider the scenario shown in Figure 7, where a partial plan P is flagged as failing at the search node n. Suppose SNLP+EBL gave E as the explanation of failure for P. Suppose further that P was produced by refining P' with the decision d. Given that P is failing, ideally we would have liked to avoid generating it in the first place. To do this, we need to isolate the constraints in P' that are predictive of the failure of P. Specifically, we would like to know what is the minimal set of constraints E' that must be present in P' such that after taking the decision d, SNLP will generate a failing plan. Since we know that P is failing because it contains the constraints E, E' must be such that if E' is true before decision d is taken, then E will be true in the resulting partial plan. This process of back-propagating a failure explanation over a decision is called *regression*.

Formally, regression of a constraint c over a decision d is the set of constraints that must be present in the partial plan before the decision d, such that c is present after taking the decision.[9] Regression of this type is typically studied in planning in conjunction with backward application of STRIPS-type operators (with add, delete, and precondition lists), and is quite well-understood (see [36]).

In state based planners, the planner decisions correspond closely to applying domain operators to world states, and thus regression over a decision is very close

[9] Note that in regressing a constraint c over a decision d, we are interested in the weakest constraints that need to be true before the decision so that c will be true after the decision is taken. The preconditions of the decisions must hold in order for the decision to have been taken any way, and thus do not play a part in regression.

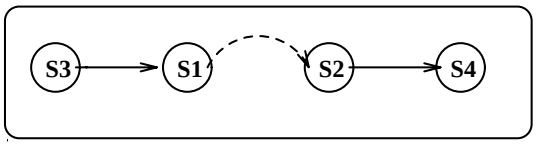


Fig. 8. An example showing transitive constraints

to regression over operators. In contrast, the decisions in partial order planners convert a partial plan to another partial plan. Even here, it is quite easy to formalize regression once we recall (see Section 3.2) that the planning decisions taken by SNLP can be seen as STRIPS-type operators with preconditions and effects that consist of constraints on the partial plan.

Simply put, regressing a constraint c over a decision d results in $True$ if the constraint c is added by d, and c itself if d does not add this constraint.

$Regress(c, d)$ $= True,$ If $c \in add(d)$ (clause (i))
$= c$ otherwise (clause (ii))

One exception to this rule occurs when the constraint being regressed belongs to a constraint family that has transitivity property. A constraint family is said to be transitive if the presence of two constraints $c_1 \wedge c_2$ of that family together imply the presence of a third constraint c_3 (i.e., $c_1 \wedge c_2 \vdash c_3$). Ordering and codesignation constraints in a partial plan are transitive constraints. For example, two ordering constraints $s_1 \prec s_2$ and $s_2 \prec s_3$ imply a third ordering constraint $s_1 \prec s_3$. In contrast, causal links, effects and preconditions are not transitive constraints.

When a planning decision adds a transitive constraint c to the plan, it is in effect adding a set of all the constraints that transitively follow. For example, in Figure 8, steps s_3 and s_4 are not ordered with respect to each other. But if a decision orders steps s_1 and s_2, it also transitively orders steps s_3 and s_4, and this needs to be taken into account in regressing ordering constraints.

Specifically, regression of a transitive constraint c over a decision d has to consider the case where the plan before d has constraint c' and d adds the constraint c'' such that c' and c'' transitively entail c. Thus,

$Regress(c, d)$ $= c'$ if $c'' \in add(d) \wedge (c'' \wedge c') \vdash c$ (clause (iii))

It is possible that there could be multiple different sets of constraints c' such that each set of constraints along with c'' could entail c. In such cases, regression of a constraint c over a decision d results in disjunction of all such sets of constraints c'. For the example in Figure 8, the regression of $(s_3 \prec s_4)$ over the ordering decision $(s_1 \prec s_2)$ results in $(s_3 \prec s_4) \vee [(s_3 \prec s_1) \wedge (s_2 \prec s_4)]$ (with the first part coming from the second clause of the regression, and the second part coming from the third clause).

Demote($s_1 \xrightarrow{\neg p''}, s_2 \xrightarrow{p'} s_3$)

Resolve the conflict between the link $s_2 \xrightarrow{p'} s_3$ and the effect p'' of step s_1

Preconditions: $s_2 \xrightarrow{p'} s_3 \in \mathcal{L}$
$p'' \xrightarrow{s_1} \in \mathcal{E}$
$mgu(p', p'') \in \mathcal{B}$
$(s_1 \prec s_2) \notin O$
$(s_3 \prec s_1) \notin O$

Effects: $O \leftarrow O + (s_1 \prec s_2)$

Constraint	**Result**	**Reason**
$(s' \prec s'')$	$True$	if $(s' \approx s_1) \wedge (s'' \approx s_2)$ clause(i)
	$(s' \prec s'') \vee$	clause (ii)
	$[(s' \prec s_1) \wedge (s_2 \prec s'')]$	clause (iii)

Fig. 9. Regression of various constraints over demotion decision. The precondition, effect, binding and causal link constraints are unaffected.

Figure 9 shows the regression of various constraints over the demotion decision. Since the demotion decision adds only ordering constraints, regression of all the other constrains such as open conditions and causal links over a demotion decision leaves them unchanged (clause (ii)). Since demotion decision adds $(s' \prec s'')$, the regression of $(s' \prec s'')$ over the demotion decision is $True$ (clause (i)). Like any ordering decision, the demotion decision also orders all the steps that precede s_1 to come before all the steps that follow s_2. As shown in Figure 8, say $(s' \prec s_1)$ and $(s_2 \prec s'')$ belong to a partial plan that is present before taking the above demotion decision. After taking the demotion decision to order s_1 to come before s_2, s' is also ordered to come before s''. Since

$$[(s' \prec s_1) \wedge (s_2 \prec s'')] \wedge (s_1 \prec s_2) \Rightarrow (s' \prec s''),$$

the result of the regression of the ordering $(s' \prec s'')$ over the demotion decision is $[(s' \prec s_1) \wedge (s_2 \prec s'')] \vee (s' \prec s'')$. Regression of constraints over a promotion decision is very similar.

Similarly, Figure 10 shows a step addition decision, $stepadd(s_1 \xrightarrow{p''}, s_2@p')$, (which adds a step s_1 into a partial plan to achieve the precondition $p'@s_2$) as well as how various constraints are regressed over it. As shown in the decision, step addition augments the steps, links, orderings, bindings, preconditions as well as effects of the partial plan. In the case of orderings, in addition to ordering the new step to precede the step where the precondition is required, a special ordering is added to make the new step follow the initial step (so as to maintain the invariant that all steps need to follow the initial step, and precede the goal step). In the case of bindings,

in addition to adding enough bindings to make sure that p' and p'' will necessarily codesignate, any pre-specified bindings of the operator (given in its ''*bindings*'' field) corresponding to the new step are also added to the plan. For example, if the new step is the blocks world operator $puton(x, y, z)$, then the internal binding constraints of the operator may be $x \not\approx y, y \not\approx z, x \not\approx z$ and $z \not\approx Table$; all of these are added to the bindings list of the plan.

It is interesting to note the regression of ordering decisions, especially the ordering constraints $s \prec s_n$ in the fifth row of the table in the figure. In this case, regressing $s \prec s_n$ over the step addition decision results in the condition `init–step`(s) (where `init–step`() is a predicate that evaluates to true if s is the initial step of the plan). This is because the step addition decision automatically adds a precedence relation between s_1 and the initial step of the plan.

Regression over the other two planning decisions, viz., promotion and simple establishment, are similar, respectively to regression over demotion and step addition.

Using Regression in EBL: Now that we have discussed regression of individual constraints over the planning decisions, regression of failure explanations is straightforward. In particular, since the failure explanation is a collection of constraints on the partial plan (Section 4.1), regressing it over a decision simply involves individually regressing each of the constraints comprising it, and conjoining the results. Formally, if $\mathcal{E} = c_1 \wedge c_2 \wedge \ldots \wedge c_i$, then

$$Regress(\mathcal{E}, d) = Regress(c_1, d) \wedge Regress(c_2, d) \wedge \ldots \wedge Regress(c_i, d)$$

One further clarification is needed regarding the use of regression in EBL. As noted above, regression of E over a decision d sometimes results in a disjunction of $E' \vee E'' \vee \ldots$ Since the motivation for using regression is to find out what part of the parent plan is responsible for generating the failure, *we use only that part of the regressed explanation which is present in the parent partial plan*. In Figure 8, when $(s_3 \prec s_4)$ is regressed over the decision to add $(s_1 \prec s_2)$, it results in $(s_3 \prec s_4) \vee [(s_3 \prec s_1) \wedge (s_2 \prec s_4)]$. However, SNLP+EBL considers $[(s_3 \prec s_1) \wedge (s_2 \prec s_4)]$ as the result of regression because the constraints $(s_3 \prec s_1)$ and $(s_2 \prec s_4)$ are present in the partial plan before the ordering decision.

4.3 Propagation of Failure Explanations

In the previous section, we have explained how failure explanations are back-propagated over a single decision. We will now describe how the regressed explanations are combined and propagated up the search tree.

StepAddition($s_n \xrightarrow{p''}, p'@s_d$)
Use the effect p'' of s_n to support the precondition $p'@s_d$

Preconditions: $p'@s_d \in \mathcal{C}$

$s' \xrightarrow{p'} s_d \notin \mathcal{L}$

$p'' \in$ effects of s_n

Effects: $\mathcal{S} \leftarrow \mathcal{S} + s_n$

$\mathcal{L} \leftarrow \mathcal{L} + s_n \xrightarrow{p'} s_d$

$\mathcal{O} \leftarrow \mathcal{O} + (s_n \prec s_d)$

$\mathcal{O} \leftarrow \mathcal{O} + \{(s_0 \prec s_n) | \texttt{init–step}(s_0)\}$ (***)

$\mathcal{B} \leftarrow \mathcal{B} + mgu(p', p'') +$ Internal bindings of s_n

$\mathcal{C} \leftarrow \mathcal{C} + \{p@s_n | p \in preconditions\ of\ s_n\}$

$\mathcal{E} \leftarrow \mathcal{E} + \{s_n \xrightarrow{e} | e \in effects\ of\ s_n\}$

Constraint	**Result**	**Reason**
$q'@s'$	$True$	if $s' \approx s_n$
	$q'@s'$	otherwise
$s' \xrightarrow{q'}$	$True$	if $s' \approx s_n$
	$s' \xrightarrow{q'}$	otherwise
$s' \xrightarrow{q} s''$	$True$	if $s' \approx s_n$
	$s' \xrightarrow{q} s''$	otherwise
$(s' \prec s'')$	$True$	if $s' \approx s_n \wedge s'' \approx s_d$
	$[(s' \prec s_n) \wedge (s_d \prec s'')] \vee$	if $s' \not\approx s_n \wedge s'' \not\approx s_d$
	$(s' \prec s'')$	otherwise
$(s \prec s_n)$	$\texttt{init–step}(s) \vee$	If $s = s_0$
	$(s \prec s_n)$	otherwise
$x \approx y$	True	if $x \approx y \in mgu(p', p'') \cup bindings(s_n)$
	$(x \approx u) \wedge (y \approx v)$	if $u \approx v \in mgu(p', p'') \cup bindings(s_n)$
	$x \approx y$	otherwise
$x \not\approx y$	True	if $x \not\approx y \in bindings(s_n)$
	$x \not\approx y$	otherwise

Fig. 10. Regression of constraints over step addition decision that adds a step s_n to achieve a condition p' at step s_d. The top part describes the step addition decision in terms of its preconditions and effects. The table below shows how individual constraints are regressed.

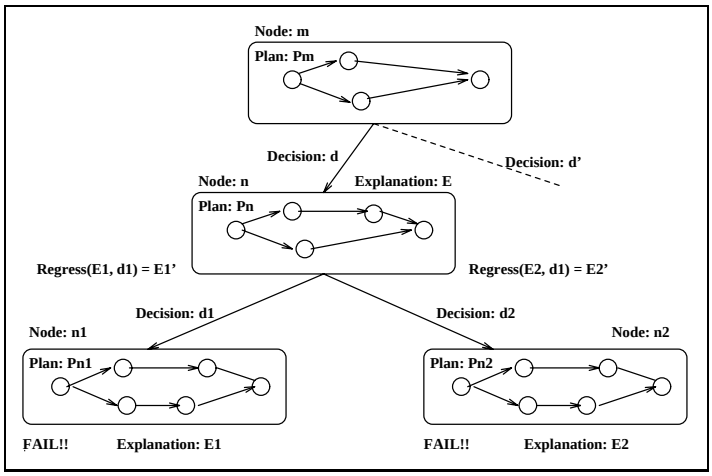


Fig. 11. An example for propagation

Consider the search tree fragment shown in Figure 11, where node m has a descendant n which then leads to two failing nodes $n1$ and $n2$. Since both children of n failed, we would like the planner to avoid generating n in future by rejecting the decision leading to it. To facilitate this, however, we need to compute the constraints in the partial plan P_m at node m that are responsible for failure of node n. In order to compute the failure reasons at node m, we regress the explanation of failure at node n over the decision d, as explained earlier. To do this, we first need to compute the explanation of failure at node n. Suppose that $n1$ and $n2$ have the failure explanations $E1$ and $E2$ respectively. Suppose further that these nodes are generated from node n to resolve a flaw, say F, by taking decisions $d1$ and $d2$ respectively. Assume that these are the only two ways of resolving the flaw. Suppose the result of regressing $E1$ and $E2$ over the decisions $d1$ and $d2$ are $E1'$ and $E2'$ respectively.

To compute the failure explanation E at node n, we note that as long as the flaw F exists at node n, the decisions d_1 and d_2 will be taken and both these will lead to failure. Thus the explanation of the failure at node n is:

$$E(n) = Constraints\ describing\ the\ Flaw \wedge \\ Regress(E1, d1) \wedge Regress(E2, d2)$$

More generally, the propagation rule for computing an explanation at node n, which has a flaw F and m search branches for resolving it, corresponding to the decisions $d_1 \cdots d_m$, and the resulting search nodes $n_1 \cdots n_m$, and failure explanation $E_1 \cdots E_m$ is:

$$E(n) = Constraints\ describing\ the\ flaw\ F\ \wedge_{\forall n_i} Regress(E_i, d_i)$$

It is interesting to note that we conjoin the failure explanations of the children branches with the description of the flaw that is being resolved by all those branches, rather than the preconditions of the decisions that attempt to resolve it.

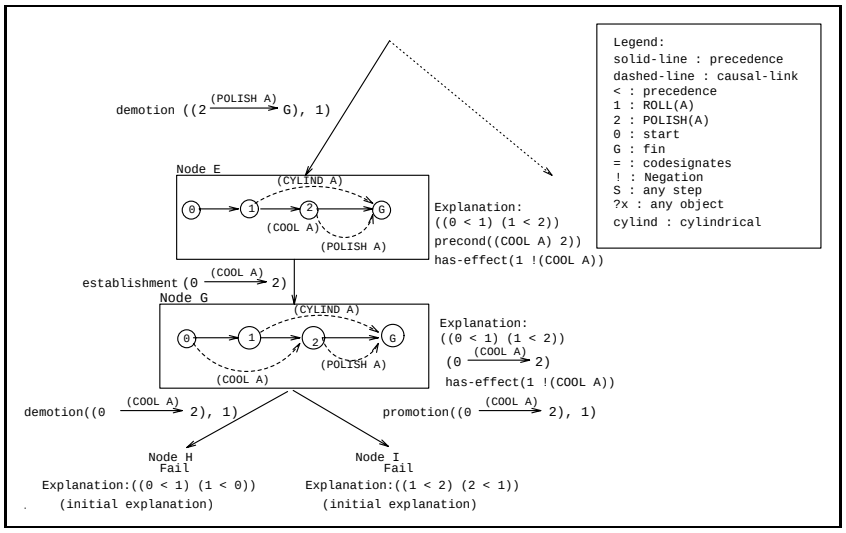


Fig. 12. A partial analysis of failures in the Job-shop Scheduling example. Negated effects are shown with

This makes sense because as long as the flaw is present, these decisions will be used to resolve it, thus resulting in a failure.[10] This method also gives rise to a more general failure explanation than would be obtained by conjoining the preconditions of the decisions with the failure explanations of the corresponding branches. This is because the preconditions of the individual decisions are all supersets of the flaw description.

Coming back to the example in Figure 11, the failure explanation E computed at node n can now be regressed over the decision d leading to n, to compute the constraints under which the decision d will necessarily lead to a failure from the node m.

Example: Let us consider the search tree described in the Figure 12, which shows the lower part of the failure branch of the example in Figure 6. When SNLP failed at node H and I in the Figure 12, EBL explains these failures in terms of ordering inconsistencies as shown in the figure. When we regress the explanation of node H over the $demotion(1 \overset{\neg Cool(A)}{\rightarrow}, 0 \overset{Cool(A)}{\rightarrow} 2)$, it results in the ordering constraint $(0 \prec 1)$. Similarly when we regress the explanation of node I over the $promotion(1 \overset{\neg Cool(A)}{\rightarrow}, 0 \overset{Cool(A))}{\rightarrow} 2)$, it results in the ordering constraint $(1 \prec 2)$. Now, at node G, we have the explanations for the failure of the branches H and I. Thus, the explanation at node G (also shown in Figure 12) is:

$$E(G) = Constraints\ describing\ the\ Unsafe\ link\ flaw\ \wedge (0 \prec 1) \wedge (1 \prec 2)$$

[10] Alert readers might note that this is a simplified model, as the number of resolution possibilities for an open condition flaw depends on the current partial plan. See Section 5 for a discussion of why soundness is preserved depite this.

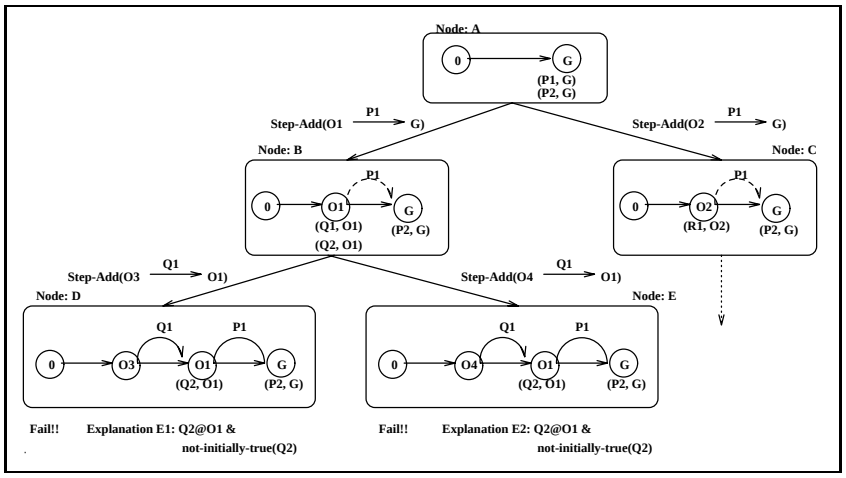


Fig. 13. An example for dependency directed backtracking

$$= (0 \overset{Cool(A)}{\rightarrow} 2) \wedge 1 \overset{\neg Cool(A)}{\rightarrow} \wedge (1 \not\prec 0) \wedge (2 \not\prec 1) \wedge (0 \prec 1) \wedge (1 \prec 2)$$
$$= (0 \overset{Cool(A)}{\rightarrow} 2) \wedge 1 \overset{\neg Cool(A)}{\rightarrow} \wedge (0 \prec 1) \wedge (1 \prec 2)$$

The last step follows from the simplification $(s_1 \not\prec s_2) \wedge (s_2 \prec s_1)$‘$equiv(s_2 \prec s_1)$ (since $(s_2 \prec s_1)$ implies $(s_1 \not\prec s_2)$ in any consistent plan). This explanation can be interpreted as follows: if there are three steps s_0, s_1 and s_2 such that $(s_0 \prec s_1) \wedge (s_1 \prec s_2)$ and if a causal link $s_0 \overset{Cool(A)}{\rightarrow} s_1$ is threatened by the step s_1, prune the node from search space. This failure explanation is sound, because as long as the unsafe link flaw exists in the partial plan, the planner will take $demotion$ and $promotion$ which will lead to failure.

4.4 Avoiding over-specific explanations in propagation

The propagation process as described above may sometimes give over-specific explanations. To see this, consider the example described in Figure 13. Here, both the children of node B fail to resolve an open condition flaw $Q2@O1$, since $Q2$ is not given by either the domain operators or by the initial state. Recall from our discussion in Section 4.1, that SNLP+EBL constructs initial explanations for these failures as shown in the Figure 13. According to the propagation rule described earlier, the explanation at node B will be:

$$E(B)=Q1@O1 \wedge Regress(E1, stepadd(O3 \overset{Q1}{\rightarrow}, Q1@O1)) \wedge Regress(E2, stepadd(O4 \overset{Q1}{\rightarrow}, Q1@O1))$$
(where $E1 = E2 = Q2@O1 \wedge \neg$`initially–true`$(Q2)$

$$=Q1@O1 \wedge Q2@O1 \wedge \neg \texttt{initially–true}(Q2)$$

When computed this way, the explanation at node B has the constraint $Q1@O1$. This is clearly redundant since node B will fail as long as the precondition $Q2@O1$ exists in the plan and $Q2$ is not given by the initial state. Inclusion of $Q1@O1$ is thus going to make the failure explanation over-specific. When this constraint is removed, the failure explanation at node B becomes the same as that at node D. This is reasonable since when the failure explanation at D is regressed over the step addition decision leading from B to D, the failure explanation remains unchanged.

To handle this formally, we change the propagation method such that when regression does not change the explanation of a failure of a node n, the complete explanation of failure at the parent node of n will be the same as explanation of failure of node n. Specifically,

$$E(n) = E1 \quad \text{If } Regress(E1, d1) = E1$$
$$= Constraints\ describing\ the\ flaw \wedge Regress(E1, d1) \wedge Regress(E2, d2) \text{ Otherwise}$$

More generally, the propagation rule for computing an explanation at node n, which has a flaw F and m search branches for resolving it, corresponding to the decisions $d_1 \cdots d_m$, and the resulting search nodes $n_1 \cdots n_m$, and failure explanation $E_1 \cdots E_m$ is:

$$E(n) = E_i \qquad \text{If } Regress(E_i, d_i) = E(n_i)$$
$$= Flaw\ resolved\ at\ n \ \wedge_{\forall n_i} \ Regress(E_i, d_i) \qquad \text{Otherwise}$$

Notice that we do not conjoin results of regression of explanations of other siblings of node n_i, if the regression does not change the explanation E_i over the decision d_i. To see why this is justified, we start by noting that since SNLP+EBL considers only sound failure explanations, E_i, which is the failure explanation of n_i must already be an inconsistent constraint set. When the regression of E_i over d_i leaves E_i unchanged, it implies that E_i is present in the partial plan at node n. This means that the partial plan at node n is already inconsistent and E_i is a sound explanation of failure for n. Thus, n cannot be refined into a solution and consequently there is no point in exploring its other refinements.

4.4.1 Dependency directed backtracking

In the context of on-line learning (i.e., doing learning along with planning), the preceding discussion suggests an elegant methodology for exploiting the explanation and regression procedures to do *dependency directed backtracking*. If an explanation of a node n, $E(n)$, does not change after regressing it over a decision $d(n)$, then the planner can safely prune all other siblings of node n. Thus, we can effectively backtrack over n's parent node, without losing completeness. Furthermore, we can continue the propagation process by regressing $E(n)$ over the decision leading to n's parent, and see if further backtracking is possible. This process stops only when we encounter a decision such that regressing $E(n)$ over it

Procedure Propagate(n_i)

A co-routine with search that recursively propagates the failure explanation
of the node n_i over the decision taken to reach that node,
restarting search appropriately.
$parent(n_i)$: The partial plan that was refined to get n_i.
$d(n_i)$: decision taken get to node n_i from its parent node;
$E(n_i)$: explanation of failure at n_i. It is initially set for the leaf nodes.
Computed recursively for the non-leaf nodes.
$F(n_i)$: The flaw that was resolved at this node.

0. Set $d \leftarrow d(n_i)$
1. $E' \leftarrow$ `Regress`$(E(n_i), d)$
2. If $E' = E$, then (dependency directed backtracking)
   $E(parent(n_i)) \leftarrow E'$; **Propagate**($parent(n_i)$)
3. If $E' \neq E(n_i)$, then
   3.1. If there are unexplored siblings of n_i
      3.1.1 Make a rejection rule rejecting the decision $d(n_i)$, with E'
         as the rule antecedent. Generalize it and store it in the rule set
      3.1.2. $E(parent(n_i)) \leftarrow E(parent(n_i)) \wedge E'$ (Update parent's explanation)
      3.1.3. Restart search at the first unexplored sibling of node n_i
   3.2. If there are no unexplored siblings of n_i,
      3.2.1. Set $E(parent(n_i)) \leftarrow E(parent(n_i)) \wedge E' \wedge F(parent(n_i)$
         (where F is the set of constraints that describe the flaw
         that the decision $d(n_i)$ is resolving)
      3.2.3. **Propagate**($parent(n_i)$)

Fig. 14. The complete procedure for propagating failure explanations

results in a failure explanation that is not the same as $E(n)$.

Example: In the example described in Figure 13, since the explanation of failure at node D did not change after regression over the step addition decision, the planner can prune the other sibling of the node D, i.e. node E, and continue the propagation of explanation above node B with the failure explanation of B set to the same as that of D.

Our implementation of SNLP+EBL folds the propagation into the search process to provide a default dependency directed backtracking. Figure 14 shows the full description of the propagation algorithm.

From the description of the `Propagate` procedure in Figure 14, we note that when DDB occurs, i.e., when the failure explanation E of a plan P regresses unchanged over the decision d leading to that plan, the parent P' of P will have its failure

explanation set to E. This is so even if P is not the first child of P' that has been refined. The the results of regressing the failures of the other children of P' will be discarded as soon as DDB occurs under P. This makes sense because E is in and of itself a set of inconsistent constraints, and can thus completely explain the failure of P'.

4.5 Rule Construction

SNLP+EBL generates search control rules as a part of the propagation of explanations up the search tree. Since SNLP+EBL currently considers only explanations of failure, only rejection rules are learned. The simplest type of search control rules are the pruning rules, which prune a partial plan after it is generated. Suppose, during the propagation process, SNLP+EBL reaches a node n and computes its failure explanation as E. It can then make a pruning rule of the form:

> **If** E holds in the partial plan
> **then Reject** the plan.

Another closely related class of rules reject decisions before they are taken. In the example above, suppose the failure explanation E is regressed over a decision d, resulting in an explanation E'. Then we can form a decision rejection rule:

> **If** E' holds in the partial plan
> **then Reject** decision d

A rejection rule is said to be **sound** (or correct) if for every partial plan P such that P is either pruned by that rule, or a decision d leading to P is rejected by that rule, P cannot be refined further to give rise to a solution for the planning problem. It is easy to see that the rejection rules described above are sound as long as the node failure explanations, on which they are based, are sound. The soundness of failure explanations depends in turn on the soundness of the regression and propagation processes; we will discuss this issue further in Section 5. Presently, we shall illustrate the rule construction process in the context of the job-shop scheduling problem described in Figure 6. Figure 15 illustrates the failure explanations and the control rules learned from the failing subtree in this example. In this example after constructing an initial explanation for node H, a rule can be learned to reject a node, as shown below:

> **if** $(s_0 \prec s_1) \wedge (s_1 \prec s_0)$ holds in a plan
> **then reject** the plan

This rule states that if there exists an ordering cycle in a partial plan of a node, then reject the node. At this point, the planner regresses the explanation over the demotion decision to explain the failure of branch H. After regression, a decision

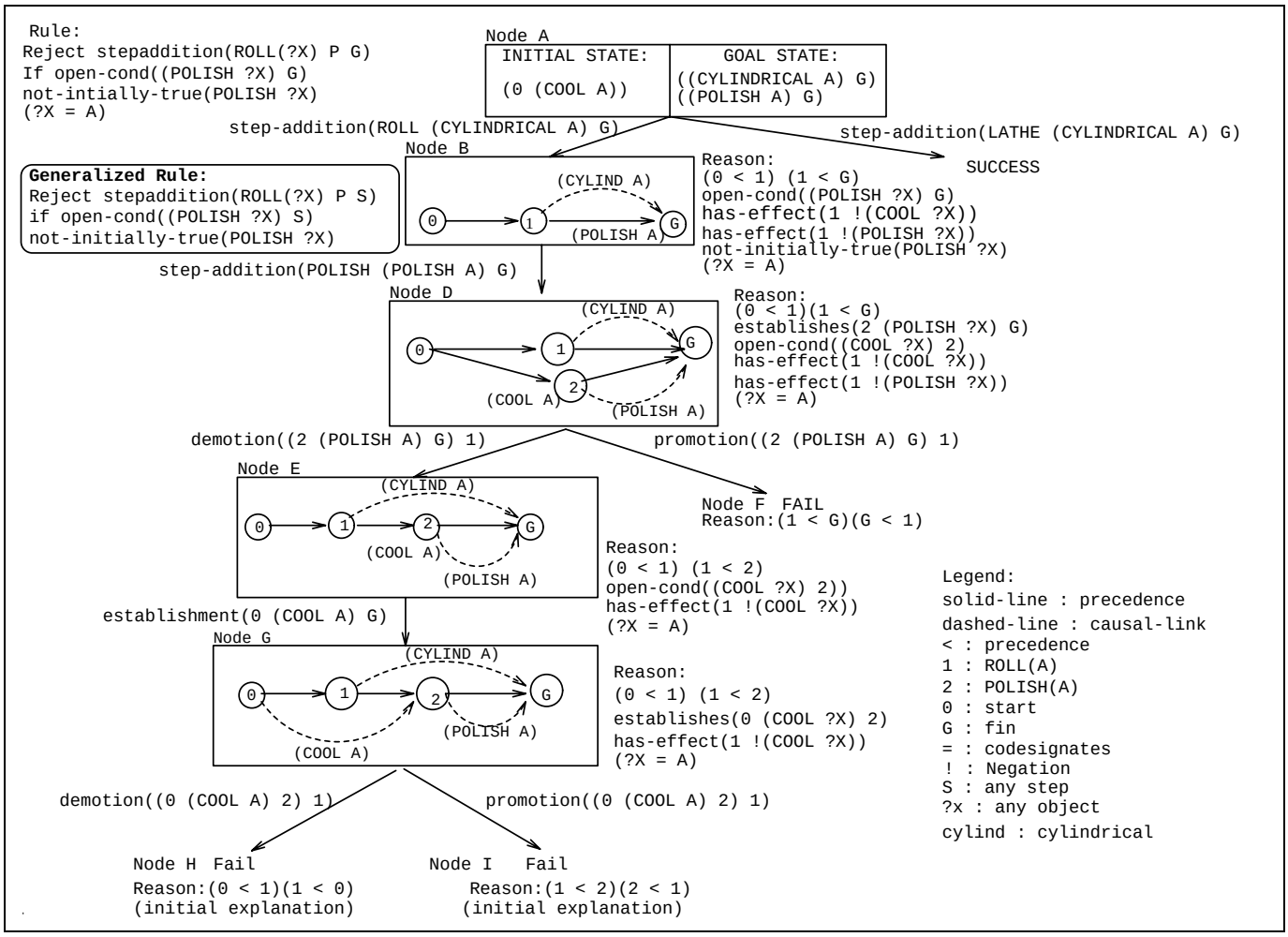


Fig. 15. A complete analysis of failures in the Job-shop Scheduling example

rejection rule can be generated as:

if $(s_0 \prec s_1)$
then reject $demotion(s_0 \overset{Cool(A)}{\rightarrow} s_2, s_1)$

Of course the two rules above are not particularly useful since the planner is going to check the inconsistency in the plan as a matter of course anyway. Specifically, the cost of matching the antecedent of the rule with the current partial plan may offset any possible savings in the search.

However, rules learned from intermediate nodes of a search tree could be more useful. For example, SNLP+EBL could learn a rule at node B to reject a plan in the search tree if the explanation at node B holds in that plan. In other words:

If $(s_0 \prec s_1) \wedge (s_1 \prec G) \wedge$
$Polished(A)@G \wedge$
$s_1 \overset{\neg Cool(A)}{\rightarrow} \wedge$

$s_1 \overset{\neg Polished(A)}{\longrightarrow} \wedge$
$\neg$initially–true($Polished(A)$)
then Reject the plan

This rule says that if there is a step s_1 in a given partial plan which deletes $Cool(A) \wedge Polished(A)$ and it comes in between two steps s_0 and G, and G requires a precondition $Polished$ (A), and $Polished(A)$ is not true in the initial state, then reject the node.

Similarly, the explanation regressed over the establishment decision at B can be used to learn a useful step establishment rejection rule at A (since A still has unexplored alternatives):

If $Polished(A)@G \wedge$
$\neg$initially–true($Polished(A)$)
then reject $stepadd(Roll(A), Cylindrical(A)@G)$

This rule states that $Roll$ should be rejected as a choice for establishing $Cylindrical(A)$ at G, if $(Polish\ A)$ is also a goal at the same step.

4.6 Generalization

Consider the step addition rejection rule discussed at the end of the previous section. This rule states that if the object A needs to be $Polished$ and it is not $Polished$ initially, then the planner should not consider $Roll$ to achieve $Cylindrical(A)$ at step G. If the planner is given the same problem again, SNLP can use the advice of the above rule and avoid adding the step $Roll$ to achieve $(CylindricalA)$, which is guaranteed to fail. Since the planner is left only with one other operator, $Lathe$, it applies this operator and succeeds. Thus, a rule advises the planner not to generate branches that will lead to failures and, consequently improves the planner's performance.

Now, assume that the planner is given a new problem which involves making an object B $Cylindrical$ and $Polished$. This new problem has the same goals as the earlier problem but it involves a different object B instead of the object A. SNLP cannot take the advice from the rule above because it is applicable only if we are making the object A, $Cylindrical$ and $Polished$ (and only if these are the top-level goals of the plan). However, it is clear that the rule can advice the planner not to add the operator $Roll$ to achieve $(Cylindrical\ B)$, even if we are dealing with object B. To make this rule applicable in cases where we are dealing with other objects, we need to remove the specific object names such as A, and step names such as G from the rule and replace it with variables, while preserving the correctness of the rule.

An object variable matches with any object of the domain and a step variable with any step of a partial plan as long as all the other constraints of a rule hold in the partial plan. A variablized rule can thus have different instantiations corresponding to different possible substitutions of constants for variables. For a generalized rule to be correct (i.e. taking its advice will not affect the completeness of the underlying planner), all of the instances of the generalized rule must be sound.

The more variables a rule contains, the more applicable it can be in new situations. Thus, ideally, we would like to remove *all* the step-names and object-names from a rule and replace these names with variables. This strategy works fine in the case of the job-shop scheduling rule above. In particular, it is easy to see that all instantiations of this following rule will be sound

If $Polished(x)@s \wedge$
$\neg$`initially–true`$(Polished(x))$
then reject $stepadd(Roll(x), Cylindrical(x)@s)$

The rule states that we should never attempt to achieve the precondition $Cylindrical(x)$ at a step s through the operator $Roll(x)$ if $Polished(x)$ is required at s and $Polished(x)$ is not true in the initial state. [11]

Unfortunately however, variablizing every constant is not guaranteed to preserve the soundness of a rule, since some specific constants (objects, steps), may have to be present in the rule for the failure to occur. To see this, consider a blocks world problem where the table is clear in the initial state, and the goal is to achieve $\neg Clear(Table)$. SNLP+EBL fails on this problem since a $Table$ is always clear. [12] Using the standard failure explanation for unestablishable goals discussed earlier, SNLP+EBL constructs the failure explanation:

$\neg Clear(Table)@G \wedge$ `initially–true`$(Clear(Table))$

and a control rule:

If $\neg Clear(Table)@G \wedge$ `initially–true`$(Clear(Table))$

[11] Recall that the rule is qualified with $\neg$`initially–true`$(Polished(x))$ because the planner did not consider simple establishment from initial state in its original search. In this case, this qualification turns out to be unnecessary as the rule holds even if $Polished(x)$ was true in the initial state. However, we could not have avoided this qualification without further hypothetical reasoning during learning phase.

[12] Although the standard implementation of SNLP does not handle negated preconditions, it is quite easy to extend it to do so. In particular, we allow a goal $\neg p@s$ to be established by the effect q of a step s' if q is in the delete list of s' and q can be made to necessarily codesignate with p. In case where s' happens to be the initial step, we make the closed world assumption with respect to the initial state, and thus can support p as long as the initial state does not contain a condition r that can necessarily codesignate with p.

then prune the plan

It is easy to see that we can variablize the specific step G to be any step (since the planner will fail as long as $\neg Clear(Table)$ is a precondition of *any* step in the plan. However, we cannot generalize the constant $Table$ to, say, an object variable x, since $Clear(x)$ can then match $Clear(A)$ where A happens to be a block. It will thus wrongly prune a partial plan with a precondition $Clear(A)@G$, leading to a loss of completeness.

The generalization process also needs to generalize step names occurring in the failure explanation. Since the explanations qualify the steps in terms of their effects and conditions, and their relations to other steps, in most cases, the step names, including the final step, can be variablized. The only exception arises in the case of the initial step, which may or may not be generalized based on the specifics of the situation.

The standard EBL methodology for ensuring correct generalization is to use a two-pass process. In the first pass, a proof is constructed for the target concept. In the second pass the structure of the proof tree is retained, the operations in the proof tree are variablized (the specific instances are replaced by fresh copies of the corresponding operation schemas), the proved target concept is variablized, and regressed through the generalized proof tree to compute the weakest conditions under which the variablized target concept can be proved again [31,35,45]. In the context of SNLP+EBL, the ''proof tree'' is the part of the search tree that terminates in failing nodes, and the operations are the planner decisions, and the generalization process will involve variablizing the planner decisions in the failing search tree, starting with variablized failure explanations of the leaf nodes and regressing them through the decisions. While this can be done (see [29]), it turns out to be more cumbersome than is necessary for our purposes. In the following we will discuss two simpler ways in which we can ensure that generalization is done correctly in the context of SNLP+EBL.

4.6.1 Correct Generalization through name-insensitive theories

The simplest way to ensure that objects will not be variablized incorrectly is to use domain theories that are *name insensitive*. A domain theory is considered name insensitive if none of the operators in the domain refer to specific objects by *name*.[13] If a domain operator needs to qualify an object, it can do so by listing the properties that the object needs to possess. Figure 16 describes two ways of writing the $Puton$ operator in the standard blocks world. The operator on the left is name sensitive as it names the specific constant $Table$. The one on the right is

[13] Similar constraints should also apply to any other components of domain theory such as domain axioms; see Section 6.

Puton (x from y to z)
Preconditions : $Clear(z), Clear(x), On(x, y)$
Effect : $\neg On(x, y), Clear(y), \neg Clear(z), On(x, z)$
Bindings: $(z \not\approx Table), (x \not\approx z)$

(a) A name-sensitive version of $Puton$ operator

Puton(x from y to z)
Preconditions : $Clear(z), Clear(x), \neg IsTable(z)$
Effect : $\neg On(x, y), Clear(y), \neg Clear(z), \neg On(x, z)$
Bindings: $(x \not\approx z)$

(b) A name-insensitive version of $Puton$ operator

Fig. 16. Examples of Name Sensitive vs. Name Insensitive operators

name insensitive. Instead of naming a specific constant, it qualifies x through the precondition $\neg IsTable(x)$.

The example also shows that even if a domain theory is not name-insensitive as given, it can be easily transformed into one which is name-insensitive (by inventing appropriate unary property predicates, and using them to augment the preconditions of the operators).

The advantage of name-insensitive domain theories is that any specific objects in the failure explanations computed by SNLP+EBL using such theories will be fully qualified in terms of the properties of the objects that are important. Because of this, during generalization phase we can variablize *every* specific object in the failure explanation. Coming back to the blocks world problem with the goal $\neg Clear(Table)@G$ discussed earlier, if SNLP+EBL uses the name insensitive $Puton$ operator (See Figure 16), then it can be verified that the failure occurs when SNLP+EBL tries to establish $\neg IsTable(Table)$ for this step. The failure explanation will be:

$$\neg IsTable(Table)@Puton(x, y, Table) \wedge \texttt{initially\_true}(IsTable(Table))$$

When this is regressed over the step addition decision involving $Puton$ operator, and conjoined with the flaw description, we get the explanation of the failure at the root node as:

$$\underbrace{\neg Clear(Table)@G}_{Flaw\ resolved} \wedge \texttt{initially\_true}(IsTable(Table))$$

At this point, the objects in this explanation can be variablized without loss of soundness, resulting in a generalized failure explanation:

$$\neg Clear(x)@G \wedge \texttt{initially\_true}(IsTable(x))$$

This explanation is sound because something that is a block cannot match x in this explanation.

Until now we talked about name-insensitivity with respect to object names. To handle proper generalization of step names, we need to ensure that the planner decisions are also expressed in a name-insensitive fashion. In particular, the planner decision should not name any specific steps without qualifying them. It turns out that the decisions are already name insensitive. We already noted that SNLP uses four types of decisions -- promotion, demotion, step addition and simple establishment. Of these, the only decision that could be name sensitive turns out to be the step addition decision, since it adds an ordering between the new step and the distinguished initial step of the plan. However, from Figure 10, we note that step addition is made name-insensitive by qualifying the initial step with the unary predicate `init–step`().

Thus, when an ordering ($0 \prec s_n$) is regressed over a step addition decision that adds a new step s_n, it regresses to the constraint `init–step`(0). This then becomes part of the final explanation (since no planning decision has an effect related to `init–step`), and appropriately qualifies any role played by the specific step 0 in the final explanation. Thus, we can generalize all the step names, secure in the guarantee that if the step 0 is playing an important part, then the constraint `init–step`() would be there to properly qualify the variablized version of this step.

4.6.2 Correct generalization in name-sensitive theories

In the previous section, we showed that name-insensitive domain theories make generalization very simple, as generalization simply involves replacing all the constants in the failure explanation with variables. We also noted that (a) the planner decisions are already name-insensitive and (b) the domain operators can easily be made name-insensitive by typing objects. However, SNLP+EBL does have the ability to correctly generalize explanations even if the domain theories are name-sensitive. The process is slightly more complex, and involves keeping track of which constants *can be* generalized and which cannot be. SNLP+EBL does this by flagging objects ‘‘special’’ if they cannot be generalized. Given a failure explanation, all the objects that are not flagged special can be variablized without losing correctness.

An object is flagged special when:

(i) A constraint *c* involving that object is regressed over a step-addition decision and
(ii) The constraint *c* is present in the operator schema of the operator added by the step-addition decision.

In the above, only step-addition decision is considered since it is the only decision that brings in new operators, and only operators can violate name-sensitivity property with respect to objects. This method can be shown to produce the same

results as the 2-pass generalization methods used by the EBG systems [35]. To see how this works, consider the example of attempting to achieve $\neg Clear(Table)$. Suppose we are using the name sensitive $Puton$ operator in Figure 16. In this case, the step addition decision of adding $Puton(u, v, w)$ to achieve $\neg Clear(Table)@G$ fails due to a binding inconsistency, and the failure explanation will be:

$$(w \approx Table) \wedge (w \not\approx Table)$$

When this is regressed over the step addition decision, we note that the constraint $w \approx Table$ is added directly by the step addition decision, and thus regresses to true. The constraint ($w \not\approx Table$) is added indirectly by the step addition decision, since this was a part of the $Puton$ operator schema. While this constraint clearly disappears after the regression, we need to note that it was specifically added only with respect to the object $Table$. Thus, SNLP+EBL flags $Table$ special. This ensures that if $Table$ appears in the failure explanation of any ancestor node, it will not be generalized. In particular, as discussed earlier, the failure explanation at the rootnode will be:

$$\neg Clear(Table)@G \wedge \texttt{initially\_true}(Clear(Table))$$

and SNLP+EBL will rightly avoid generalizing $Table$.

This approach flags objects as special *only* when it is clearly necessary. To see this, consider a slightly different blocks world example. Suppose we have a goal to make $On(A, Table) \wedge On(A, B)$, and the initial state has $On(A, Table)$. Suppose SNLP+EBL first makes $On(A, Table)$ true by simple establishment, and then makes $On(A, B)$ true by adding a step $Puton(A, Table, B)$. At this point, the partial plan will fail due to an unresolvable unsafe link, and the explanation of failure will be:

$$0 \overset{On(A,Table)}{\rightarrow} G \wedge (0 \prec Puton() \prec G) \wedge Puton() \overset{\neg On(A,Table)}{\rightarrow}$$

When this explanation is regressed over the step addition decision involving $Puton$ operator, $Table$ will not be flagged special since none of the constraints in the failure explanation are added indirectly by the operator $Puton$. Thus, we will rightly be able to generalize $Table$, and learn a rule which says that a partial plan will fail if it has two preconditions $On(x, y)@s \wedge On(x, z)@s$

4.7 Rule Storage

Once a rule is generalized, it is entered into the corpus of control rules available to the planner. These rules thus become available to the planner in guiding its search

in the other branches during the learning phase, as well as subsequent planning episodes. In storing rules in the rule corpus, SNLP+EBL makes some bounded checks to see if an isomorphic rule is already present in the stored rules. In this research, we ignored issues such as monitoring the utility of learned rules, and filtering bad rules. Part of the reason for this was our belief that utility monitoring models developed for state-space planners [20,33] will also apply for plan-space planners. Another reason is that since we only learn rejection rules based on search failures, in general we have fewer search control rules compared to planners such as PRODIGY [31] that learn from a variety of target concepts.

5 On the Soundness of Search control rules learned by SNLP+EBL

As explained in chapter 4.6, a rule is said to be sound if it does not affect the completeness of the underlying planner. Formally, a search control rule is said to be **sound** if and only if whenever the rule prunes a partial plan P, it is necessarily the case that P could not have been refined into a solution. In this section, we shall argue that the rules learned by SNLP+EBL are sound.

In order to preserve soundness, a rule should guarantee that it is not removing any solution from the search space. In the case of partial plans at the leaf nodes of the search tree, which are flagged by SNLP+EBL as failing, SNLP+EBL constructs their failure explanations as the set of inconsistent constraints present in the failing plans. Since a plan with inconsistent constraints cannot be refined to a successful plan, rules that are constructed from the initial explanations are sound.

The situation is more complex for rules that are learned from the intermediate nodes in the search tree. A rule that is learned from an explanation at an intermediate node of the search tree is sound only if the explanation of the node accounts for the failures of all possible branches under that node. This is complicated by the fact that sometimes, the possible branches will vary when the problem details change. For example, when the initial state changes, there may be more possible establishment opportunities from the initial state. The issue thus becomes one of ensuring that all possible and potential search branches are properly accounted for in generating a failure explanation. In the following paragraphs, we explain how this is done in SNLP+EBL.

A failing intermediate node in the search tree of SNLP+EBL may be classified into two types. The node is of conflict resolution type when all the branches below it correspond to resolution of some unsafe link flaw. The second type of nodes, called establishment type nodes are such that all the branches under than correspond to different ways of establishing a goal.

In the case of conflict resolution type nodes, the planner has only two choices to

order the threat by promotion or by demotion. To ensure that both possibilities are always explicitly considered, we simply modified SNLP such that it always generates two branches to account for promotion and demotion irrespective of the constraints in the partial plan. If this leads to an ordering cycle, it is then detected and the resulting plan is flagged as a failing plan.

In the case of establishment type nodes, there are two types of branches: step addition branches or simple establishment branches. Since the number of operators available in the domain are fixed, the number of branches that are generated by the step addition choices are fixed irrespective of the details of the current problem. It should be noted that we need to explicitly consider all failing operators, even if their internal binding constraints turn out to be inconsistent with the current plan. Since standard implementation of SNLP avoids generating such branches; we changed it such that the steps are first introduced, and then the failures are flagged. Note that this is reasonable, since the binding constraints on an operator can be seen as a form of preconditions that need to be satisfied, and as as such should be worked on after the operator is introduced.

The number of simple establishment branches under a node are however not fixed since the number of steps in a partial plan that can give an open condition depends on the constraints in the partial plan. Simple establishments can be separated into two categories, (i) establishments from initial state and (ii) establishments from steps other than initial state. We will treat these two cases in turn.

Simple establishments from initial state: Since the initial state changes from problem to problem, the number of simple establishment branches from the initial state may vary too. For example, suppose we are trying to establish a condition P at a step s and we fail. Suppose further that in the current partial plan, P is not true in the initial state. It is possible that had initial state given p, the failure would have been avoided. To handle this, we can do one of two following things:

-- Qualify the Explanation: Qualify the failure explanation with a constraint, $\neg$`initially_true`(P). This essentially states that any rule learned here will only be applicable if P is not true in the initial state.
-- Counterfactual Reasoning: The approach of qualifying explanations may lead us sometimes to over specific explanations. For example, it may be that the simple establishment from initial state to achieve P at s would have failed even if P were true in the initial state. In such cases, we can get more general but sound explanations by doing counterfactual reasoning i.e. assume that P is given by initial state and check the simple establishment from initial state still fail. If it fails, the qualification is not necessary.

Since counterfactual reasoning can be expensive, we use the first approach of qualifying the explanation in our current implementation.

The approach of qualifying the explanation may not work well in cases where the failed condition is non-propositional. For example, suppose we were trying to achieve $P(x)$ at a step s. Suppose that initial state currently has $P(A)$ and we fail to achieve $P(x)$ at s under these conditions. Now, to ensure soundness, we need to qualify that the initial state gives only $P(A)$ and not $P(B)$, $P(C)$ etc. In other words, the qualification constraint is

$$x \not\approx A \Rightarrow \neg \texttt{initially\_true}(P(x))$$

While this is possible to do, the resulting explanations may become too specific and expensive to match. In our current implementation, we simply avoid learning from any failure branches corresponding to uninstantiated goals. Fortunately, efficient planning anyway demands that the planner prefer working on maximally instantiated open conditions since such goals will have least number of possible establishment branches [21]. Therefore, this restriction does not seem to affect the efficiency of the learner.

Simple establishments from steps in the partial plan: In order to generate a sound explanation at a node, simple establishment branches pose a problem since the number of simple establishments from existing steps (other than the initial state) can depend on the current ordering and binding constraints in the plan. In SNLP+EBL, we consider only those simple establishment branches that are generated by the planner.

At first glance, this looks insufficient. Specifically, since some of the simple establishment possibilities are not considered by the planner because of the existing constraints, the question arises as to whether we need to consider them too (using explanation qualification or counterfactual reasoning techniques as above) to make a sound explanation.

Let us illustrate this issue with an example. Suppose we want to achieve a condition P at a step s_1, and the current plan has an instance of operator O (say with step name s_2) which gives P except that it is coming after s_1. Suppose we find that all the other establishment choices are failing, and SNLP+EBL computes the explanation of failure propagated from those branches. Our worry is whether we need to qualify this explanation in some way to take care of the fact that P could have been achieved at s_1 by s_2, had s_2 not been following s_1.

Although it is not too hard to take the failures of impossible simple establishments into account (similar to the way we explicitly consider impossible promotion and demotion decisions), this is not required to ensure soundness. Even though certain simple establishments are not generated by the planner because of the constraints in the partial plan, the planner considers all the step-addition possibilities involving the same operators. If it fails to establish an open condition, it must be because it can

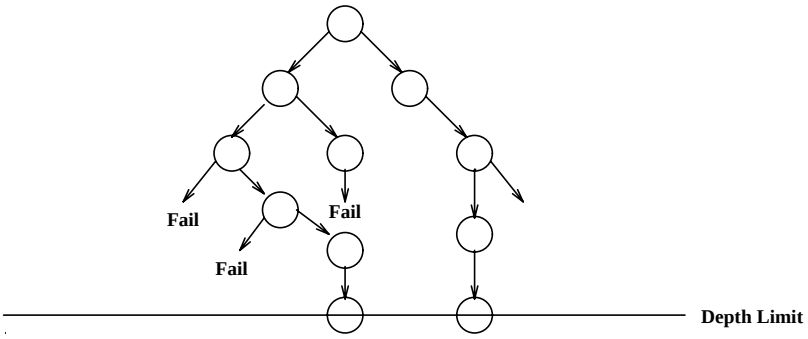


Fig. 17. A search tree showing depth limit failures

not do so even if it were allowed to have fresh copies of all the operators. The failure explanations from the step addition branches will subsume the failure explanations that could have been produced from the impossible simple establishments.

6 Learning from Depth Limit Failures

In earlier sections, we described the framework for learning search control rules from failures that are recognized by SNLP. The rules learned from analytical failures detected by SNLP+EBL were successful in improving performance of SNLP in some synthetic domains (such as $D^m S^{2*}$ described in [1]). Unfortunately however, learning from analytical failures alone turned out to be less useful in many domains.[14] The reason is that often the planner crosses the depth limit, without encountering any failure (see Figure 17). An important reason for this turns out to be that, in many cases, SNLP goes into an unpromising branch and continues adding locally useful, but globally useless constraints (steps, orderings, bindings) to the plan, without making any progress towards a solution [26].

An example here might help to see why SNLP gets into infinite loops. In Figure 18, SNLP achieves $On(A, B)$ at G by establishing it from initial state. Then it tries to achieve $On(B, C)$ at G by introducing a new step $s1$ (which corresponds to an operator $Puton(B, y, C)$), and ordering $s1$ to come in between initial state $S0$ and goal state G. But the newly added step $s1$ requires $Clear(B)$ as one of its preconditions. Intuitively, it is clear that this plan is doomed to failure since we cannot both protect $On(A, B)$ and achieve $Clear(B)$, as required by the plan in the situation before step $s1$.

Since there are no ordering or binding inconsistencies in the partial plan, SNLP tries to continue refining the plan, possibly crossing depth limit before attempting

[14] UCPOP+EBL did overcome this problem to a certain extent when the operator representation is beefed up to make explicit some of the implicit domain characteristics; see Section 10

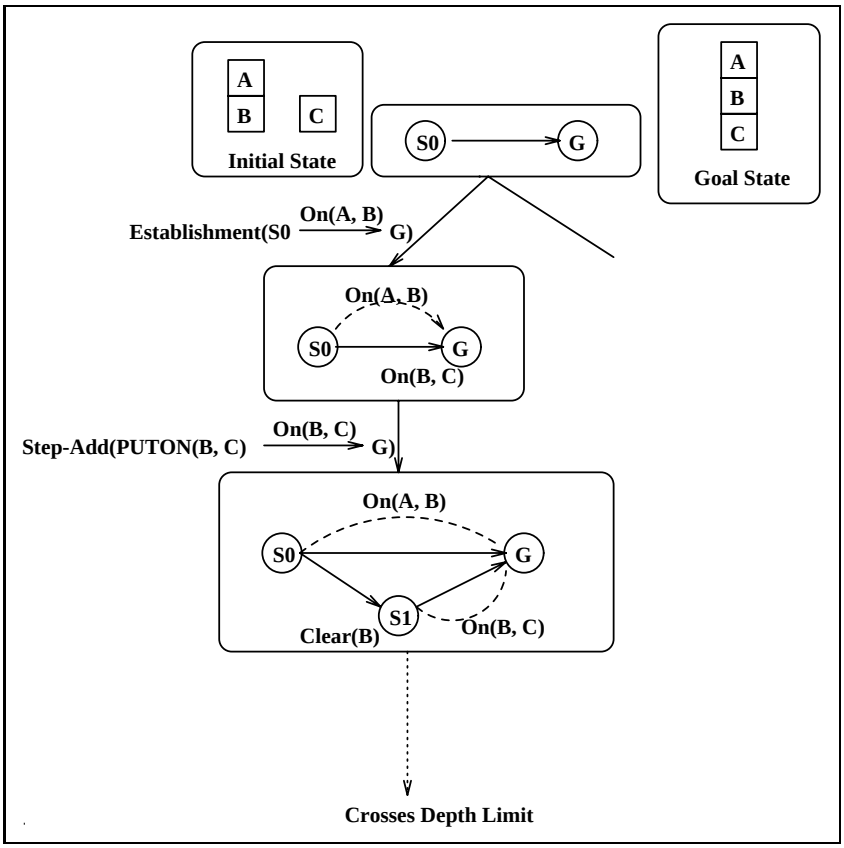
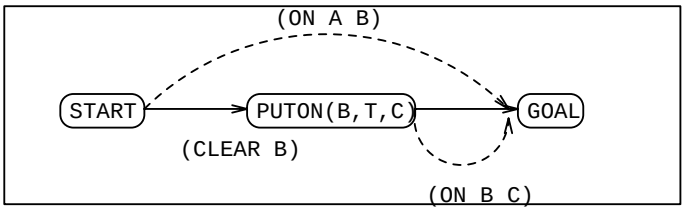


Fig. 18. An example showing a branch of a search tree that may possibly cross depth limit

$Clear(B)$. In this section, we will describe how SNLP+EBL explains implicit failures at depth limits and learn from these failures.

In general, if a branch of a node crosses a depth limit, the partial plan at the depth limit is assumed to be unpromising and is removed from consideration. But when a branch crosses a depth limit, there is no obvious failure explanation. As noted in Section 4.3, an explanation at an intermediate node in a search tree is the conjoined explanation of all the possible children explanations and the constraints describing the flaw. If we do not explain the reason for pruning the partial plan at depth limit, then we cannot construct an explanation for a node which has some branches that failed analytically, and some that crossed a depth limit. This limits EBL to learn effective search control rules.

As mentioned in Section 4.1, sometimes it is possible to use strong consistency checks based on the domain-theory as well as the meta-theory of the planner to show that the partial plan at the depth-limit contains a failure that the planner's consistency checks have not yet detected. Consider the previous example, the partial plan at node C is shown below:



Given the blocks world domain axiom that no block can have another block on top of it, and be clear at the same time, and the SNLP meta-theory that a causal-link, $s_1 \xrightarrow{c} s_2$, once established, will protect the condition c in every situation between s_1 and s_2, we can see that the above partial plan can never be refined into a successful plan. To generalize and state this formally, we define the *np–conditions*, or necessarily persistent conditions, of a step s' in a plan $\mathcal{P}$ to be the set of conditions supported by any causal link, such that s' necessarily intercedes the source and destination of the causal link.

$$\textit{np–conditions}(s') \;=\; \{c | s_1 \xrightarrow{c} s_2 \in \mathcal{L} \wedge s_1 \prec s' \wedge s' \prec s_2\}$$

Given the *np–conditions* of a step, we know that the partial plan containing it can never be refined into a complete plan as long as $precond(s') \cup$ *np–conditions*(s') is inconsistent with respect to domain axioms.[15] However, SNLP's local consistency checks will not recognize this, leading it sometimes into an indefinite looping behavior of repeatedly refining the plan in the hopes of making it complete. In the example above, this could happen if SNLP tries to achieve $Clear(B)$ at step 1 by adding a new step $s_3 : Puton(x, B, z)$, and then plans on making $On(x, B)$ true at s_3 by taking A off of B, and putting x on B. When such looping makes SNLP cross depth-limit, SNLP+EBL uses the *np–conditions* based consistency check, to detect and explain this implicit failure, and learn from that explanation.

To keep the consistency check tractable, SNLP+EBL utilizes a restricted representation for domain axioms (first proposed in [11]): each domain axiom is represented as a conjunction of literals, with a set of binding constraints. The table below lists a set of domain axioms for the blocks world. The first one states that y cannot have

[15] In fact the plan will also fail if $effects(s') \cup$ *np–conditions*(s') is inconsistent. However, given any action representation which makes STRIPS assumption, (i.e., every literal whose truth value is affected by an action must necessarily occur in the effects list of the action) these inconsistencies will any way be automatically detected by the normal threat detection and resolution mechanisms.

x on top of it, and be clear, unless y is the table.[16]

$$\boxed{\begin{array}{c} On(x,y) \wedge clear(y)[y \not\approx Table] \\ On(x,y) \wedge On(x,z)[y \not\approx z] \\ On(x,y) \wedge On(z,y)[x \not\approx z, y \not\approx Table] \end{array}}$$

A partial plan is inconsistent whenever it contains a step s such that the conjunction of literals comprising any domain-axiom are unifiable with a subset of conditions in *np–conditions*$(s) \cup precond(s)$. Given this theory, we can now explain and learn from the blocks-world partial plan above. The initial explanation of this failure is:

$$0 \overset{On(A,B)}{\rightarrow} G \wedge (0 \prec 1) \wedge (1 \prec G) \wedge Clear(B)@1 \wedge (B \not\approx Table)$$

This explanation can be regressed over the planning decisions to generate rules. The type of analysis described above can be used to learn from some of the depth-limit failures. In the blocks world, use of this technique enabled SNLP+EBL to produce several useful search control rules. Figure 19 lists a sampling of these rules. The first one is an establishment rejection rule which says that if $On(x,y) \wedge On(y,z)$ is required at some step, then reject the choice of establishing $On(x,y)$ from the initial state, if initial state is not giving $On(y,z)$. Note the presence of the constant $Table$ in the antecedent of this rule. This is in accordance with our description in the generalization section (since the constraint $y \not\approx Table$, which is part of the initial failure explanation, is regressed over the step addition decision that adds $Puton()$ step, making $Table$ a special, non-generalizable constant). [17] Notice also that all the first and the third rules qualify the step s with the constraint `init–step`(s). Once again this makes sense since learning those rules involves regressing a constraint of the form $0 \prec Puton()$ over the step addition decision, giving rise to `init–step`(0), which later gets variablized to `init–step`(s).

Handling Multiple Failure Explanations: When we learn by analyzing plans at depth-limits, it is sometimes possible that the analysis unearths multiple failure explanations. In this case, a question arises as to which explanation should be

[16] Note that this particular domain axiom is not name insensitive (From the point of view of generalization, this particular domain axiom is not name-insensitive. It can however be made name insensitive by converting $Table$ into a unary predicate as discussed in Section 4.6.1

[17] Note that the the constraint $y \not\approx Table$ can be dropped without affecting the correctness of the rule. The constraint comes in because the initial explanation with respect to domain-axiom based failure says that $On(x,y)$ and $Clear(y)$ can't be true simultaneously unless $y \not\approx Table$. It turns out that we can simplify this constraint away since $On(y,z)$ is one of the other goal conjuncts, and if y were equal to Table, then the problem could anyway not have been solved. In our current system, we do not do this type of simplification.

(1) **Reject Simple establishment** $s \overset{On(x,y)}{\rightarrow} s_1$
If $\texttt{init–step}(s) \wedge On(y,z)@s_1 \wedge$
$\neg\texttt{initially–true}(On(y,z)) \wedge (y \not\approx Table)$
(2) **Reject promotion** $s_1 \prec s_3$
If $clear(x_2)@s_3 \wedge$
$s_1 \overset{On(x_1,x_2)}{\rightarrow} s_2 \wedge$
$(s_3 \prec s_2) \wedge (x_2 \not\approx Table)$
(3) **Reject step addition** $puton(x',y) \overset{Clear(z)}{\rightarrow} s_1$
If $s \overset{On(x,y)}{\rightarrow} s_2 \wedge \texttt{init–step}(s) \wedge$
$(s_1 \prec s_2) \wedge (y \not\approx Table)$

Fig. 19. A sampling of rules learned using domain-axioms in Blocks world domain

Test Set	SNLP		SNLP+EBL		SNLP+Domax	
	% Solv	C. time	% Solv	C. time	% Solv	C. time
I (30 prob)	60%	1767	100%	195	97%	582
II (100 prob)	51%	6063	81%	2503	74%	4623

Table 1
Results from the blocks world experiments

preferred. SNLP+EBL handles this by preferring the explanations containing steps introduced at shallower depths (in our case, smaller step numbers). This allows the failure explanation to regress to higher levels in the search tree, thereby learning more effective control rules. As a side effect, it also helps the dependency directed backtracking component during the learning phase.

7 Experimental evaluation of SNLP+EBL

To evaluate the effectiveness of the rules learned by SNLP+EBL, we conducted experiments on random problems in blocks world. The problems all had randomly generated initial states consisting of 3 to 8 blocks (using the procedure outlined in Minton's thesis [31]). The first test set contained 30 problems all of which had random 3-block stacks in the goal state. The second test set contained 100 randomly generated goal states (using the procedure in [31]) with 2 to 6 goals . For each test set, the planner was run on a set of randomly generated problems drawn from the same distribution (20 for the first set and 50 for the second). Any learned search-control rule, which has been used at least once during the learning phase, is stored in the rule-base. This resulted in approximately 10 stored rules for the first set, and 15 stored rules for the second set. In the testing phase, the two test set problems were run with SNLP , SNLP+EBL (with the saved rules) as well as

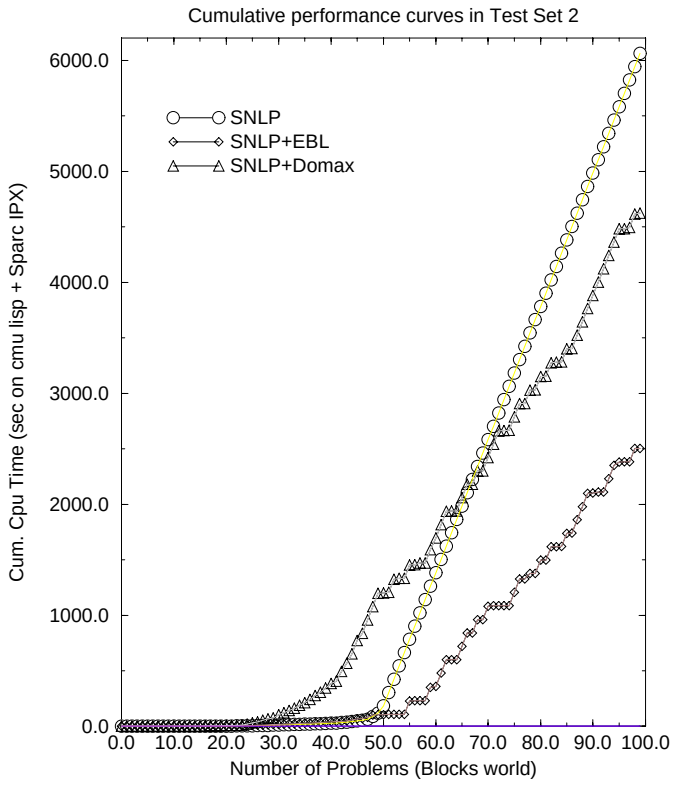


Fig. 20. Cumulative performance curves for Test Set 2

SNLP+Domax, a version of SNLP which uses domain axioms to prune inconsistent plans as soon as they are generated. A cpu time limit of 120 seconds was used in each test set.

Table 1 describes the results of these experiments. Figure 20 shows the cumulative performance graphs for the three methods in the second test set. Our results clearly show that SNLP+EBL was able to outperform SNLP significantly on these problem populations (p-value for this was .24 for sign test and .00 for signed rank test [15]). A closer analysis of the second set revealed that SNLP+EBL outperformed SNLP in 36 problems, resulting in a cumulative saving of 3,587 cpu. sec. SNLP on the other hand outperformed SNLP+EBL in 43 instances, but the cumulative difference in this case was a mere 27 sec. Similarly between SNLP+Domaxand SNLP+EBL, SNLP+EBLdoes better 76 of the problems, with a 2120 cumulative difference, while SNLP+Domaxout performs SNLPin 3 problems with 1 sec difference The results about SNLP+Domax also show that learning search-control rules is better than using domain-axioms directly as a basis for stronger consistency check on every node during planning (p-value for this was .00 for both sign test and signed rank test). This is not surprising since checking consistency of every plan during search can increase the refinement cost unduly. EBL thus provides a way of strategically applying stronger consistency checks. Finally, the fact that SNLP+EBL fails to

```
BW-prop
Operator: NEWTOWER
:precondition (and (on ?x ?z) (clear ?x) (neq ?x ?z)
                   (block ?x) (block ?z) (Tab Table))
:effect (and (on ?x Table) (clear ?z) (not (on ?x ?z)))))
Operator: PUTON
:precondition (and (on ?x ?z) (clear ?x)  (clear ?y) (neq ?x ?y) (neq ?x ?z)
                   (neq ?y ?z) (block ?x) (block ?y) (block z))
:effect  (and (on ?x ?y) (not (on ?x ?z)) (clear ?z) (:not (clear ?y)))))
```

```
BW-cond
Operator: PUTON
:precondition (and (on ?x ?z) (clear ?x) (clear ?y)
                   (neq ?x ?z) (neq ?x ?z)
                   (neq ?x ?y) (block ?x))
:effect (and (on ?x ?y) (not (on ?x ?z))
             (when (block ?z) (clear ?z))
             (when (block ?y) (not (clear ?y)))))
```

```
BW-quant
Operator: PUTON
:precondition
      (and (on ?x ?z) (neq ?x ?y) (neq ?x ?z) (neq ?y ?z)
      (or (tab ?y)
          (:forall (block ?b) (:not (on ?b ?y))))
      (:forall (block ?c) (:not (on ?c ?x))))
:effect (:and (on ?x ?y) (:not (on ?z ?x))))
```

Fig. 21. Three different encodings of the Blocks world domain

solve 19% of the test problems in the second set shows that there may be other avenues for learning search control rules.

8 Extensions needed to support EBL in UCPOP

In this section, we will describe how the EBL framework is extended to *UCPOP*, giving rise to UCPOP+EBL. This discussion will also demonstrate that it is relatively straightforward to extend our framework to other plan space planners.

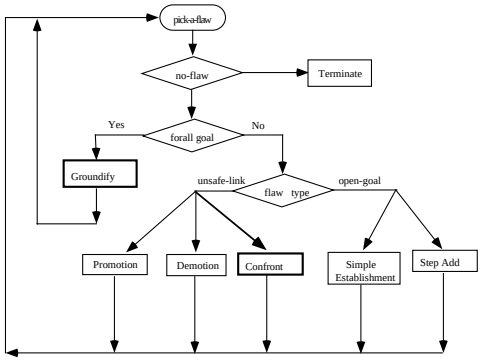
Like SNLP [30], UCPOP [39] searches in a space of partial plans, refining (adding constraints to) a partial plan until it becomes a complete solution to the planning problem. Figure 22(a) shows the description of a simple example domain for UCPOP called the briefcase domain [39,38], which involves moving objects from one location to another with the help of a briefcase. Note that the actions contain conditional and quantified effects (e.g. $\forall_{obj(x)}\ [in(x) \rightsquigarrow at(x,m) \wedge \neg at(x,l))]$). Similarly, it is also possible for actions to have quantified preconditions (e.g. $\forall_{block(x)} Clear(x)$) or disjunctive preconditions (e.g $Clear(x) \vee On(x,Table)$). The ability to handle quantified (universal or existential) and disjunctive preconditions, and universally quantified and conditional effects [18] allows UCPOP to represent many domains more compactly, and to introduce more of the domain physics into the operator description explicitly. As an example, Figure 21, shows three different descriptions of blocks world domain theory that make use of the expressiveness of the UCPOP operator language to varying extents. Note that in contrast to the first description which requires two different operators, and four different predicates, the third one requires only three predicates and a single operator.

The presence of more expressive preconditions and postconditions also means that the flaw resolution procedure used by SNLP needs to be extended to work for UCPOP. This in turn may require extensions to the EBL framework. In the following sections, we shall discuss these differences in detail and explain the

[18] UCPOP, which is based on Pednault's ADL theory of planning, [37], does not allow non-deterministic postconditions. This means that disjunctive and existentially quantified effects are not allowed.

mov-b(l, m) ;; Move B from **l** to **m**
precond: $m \neq l \wedge at(B, l)$
eff: $at(B, m) \wedge \neg at(B, l)$
$\forall_{obj(x)} in(x) \Rightarrow at(x, m) \wedge \neg at(x, l)$
take-out(o) ;;Take out **o** from **B**
precond: $in(o)$
eff: $\neg in(o) \wedge \neg closed(B)$
close-b() ;; Close the briefcase
precond: $\neg closed(B)$
eff: $closed(B)$

(a) The Briefcase Domain



(b) Flowchart of refinement process in UCPOP

Fig. 22. Example domain and Flowchart of UCPOP

extensions made to the EBL framework to handle them. Figure 22(b) shows a flow chart of the plan-refinement process in UCPOP.

8.1 Operator preconditions

As we remarked, UCPOP operators can have quantified and disjunctive preconditions. They necessitate changes to the way open condition flaws are resolved. We shall see that the only significant change to EBL is necessitated by the treatment of universally quantified preconditions.

8.1.1 Universally quantified preconditions

When we have operators with universally quantified preconditions of the form $\forall_{type(x)} cond(x)$, they give rise to universally quantified open conditions of the form $\forall_{type(x)} cond(x)@s$. UCPOP handles such preconditions by making the assumption that the domains have finite and static universes (static universe assumption means that the operators do not create new objects). Under these assumptions, a quantified formula is just a shorthand notation for a conjunction involving individual objects. Thus, universally quantified open conditions are handled by converting the quantified formula into a conjunction over the objects that are spanned by the quantification, and treating the individual non-quantified open-conditions in the same way as SNLP.

In particular, the open condition $\forall_{type(x)}\ cond(x)@s$, is converted into an equivalent set of unquantified preconditions of the form $cond(o_1)@s \wedge cond(o_2)@s \cdots \wedge cond(o_n)@s$ (where $o_1 \cdots o_n$ are the only objects of category ‘‘type’’). Notice that once the quantified conditions are instantiated this way, the filter condition (‘‘type’’) does not appear in the partial plan.

From the EBL point of view, the only thing that is required is that the search tree explicitly model this instantiation process, so that explanations of failures containing precondition constraints can be appropriately regressed. Since the instantiation process occurs at the time the step is first introduced into the plan, the logical place to model it would be in the step addition decision. Specifically, the effects of the step addition decision are changed to include:

$$\mathcal{C} \leftarrow \mathcal{C} + \left\{ cond(x)@s \;\middle|\; \begin{array}{c} \forall_{type(x)} cond(x) \in preconditions\ of\ s \\ \wedge \texttt{initially\_true}(type(x)) \end{array} \right\}$$

where s is a new step being added, which contains the quantified precondition. From this, we note that when an open condition $cond(A)@s$ is regressed over the step addition decision that introduced the operator having the quantified precondition $\forall_{type(x)} cond(x)@s$, it results in $cond(A)@s \vee \texttt{initially\_true}(type(A))$ (the idea being that the precondition constraint would have been automatically added by the instantiation decision if x is an object of type $type$). As before (Section 4.2), UCPOP+EBL picks the disjunct that holds in the current example.

Example: To illustrate how EBL can handle regression over universally quantified decisions, consider a variant of the blocks world domain, that contains, in addition to the predicates $clear$ and on, a predicate $allpainted$. Suppose that the goal $allpainted$ can only be achieved by operator $spraypaint$, with precondition $\forall_{block(x)}\ clear(x)$. Consider the problem of achieving the conjunctive goal: $on(A, B) \wedge allpainted$, given that in the initial state we have A on top of B. Figure 23 shows the search tree explored by UCPOP+EBL in attempting to solve this problem. UCPOP+EBL first establishes $On(A, B)$ through simple establishment from the initial state. Then, it adds the step $s1{:}\, spraypaint$ to the plan. Next, it works on the quantified precondition $\forall_{block(x)} Clear(x)@s1$, and instantiates it into non-quantified goals $clear(A)@s_1 \wedge clear(B)@s_1$ (since A and B are the only blocks in the problem). We know that the resulting plan is inconsistent since it is impossible to both protect $On(A, B)$ and achieve $Clear(B)$ in the situation before $s1$. Suppose this failure is discovered (either by further planning, or by the use of the domain-axiom based consistency checks discussed in Section 6), and the following failure explanation is provided:

$$E_3 = 0 \overset{on(A,B)}{\rightarrow} G \wedge clear(B)@s_1 \wedge (0 \prec s_1) \wedge (s_1 \prec G)$$

When this explanation is regressed over the quantification instantiation decision, the constraint $clear(B)@s_1$ regresses to $\texttt{initially\_true}(block(B))$. Continuing this process, the failure explanation at P_2 becomes:

$$E_2 = 0 \overset{on(A,B)}{\rightarrow} G \wedge \texttt{initially\_true}(block(B)) \wedge allpainted@G \wedge \texttt{init\_step}(0)$$

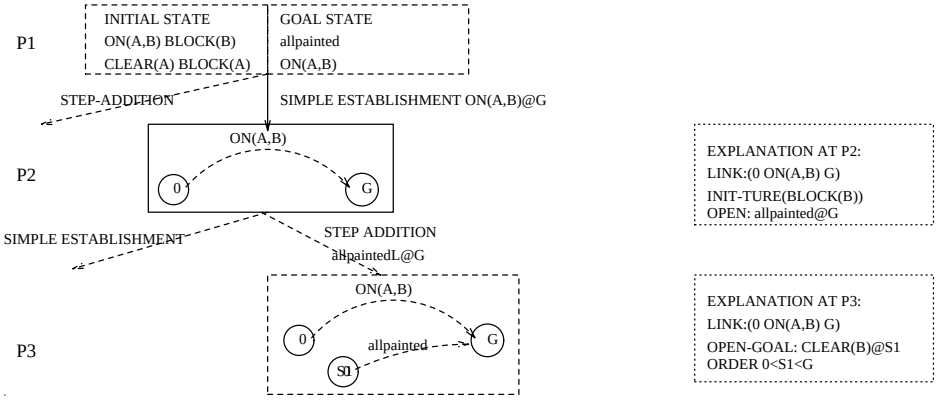


Fig. 23. Regression of quantified precondition over quantification instantiation decision in blocks world domain

Regressing E_2 over the simple establishment decision, we can learn the simple establishment rejection rule:

If $allpainted@G \wedge$ `initially–true`$(block(B)) \wedge$ `init–step`(0)
then Reject Simple establishment$(0 \overset{On(A,B)}{\rightarrow} G)$

Which states that the simple establishment of $On(A, B)$ from initial state can be rejected as long as $allpainted$ is a precondition to be achieved, and B is a block. Using the same generalization framework as that used by SNLP+EBL, this rule can be generalized to

If $allpainted@s' \wedge$ `initially–true`$(block(y)) \wedge$ `init–step`(s)
then Reject Simple establishment$(s \overset{On(x,y)}{\rightarrow} s')$

To see that this process gives rise to sound rules, note that in a different situation, if we want the goals $On(A, Table) \wedge allpainted$ (with $On(A, Table)$ being initially true), then the planner can in fact succeed through the simple establishment branch. Thus, we do not want the rejection rule above to be applicable in such a case (as it would then wrongly reject the decision). Indeed, the qualification `initially–true`$(block(y))$ ensures that the explanation will not be applicable in this situation (since if y is a table, then `initially–true`$(block(table))$ will not hold.

8.1.2 Disjunctive preconditions

UCPOP operators can have disjunctive preconditions of the form $(p \vee q)$, which give rise to open condition flaws of the form $(p \vee q)@s$. As UCPOP assumes a completely specified initial state, and deterministic actions, it can handle disjunctive preconditions by planning to make either of the preconditions true. Thus, given an open condition flaw $(p \vee q)@s$, UCPOP makes two partial plans, one containing the flaw $p@s$ and another containing the flaw $q@s$, and puts both plans on the

search queue.

From the point of view of EBL, disjunctive preconditions do not create any special problems. Specifically, if both the search branch containing $p@s$ and the search branch containing to $q@s$ fail with failure explanations E_1 and E_2 respectively, then the combined failure explanation for the node containing $(p \vee q)@s$ is computed as:

$$(p \vee q)@s \wedge E_1 \wedge E_2$$

8.1.3 Existentially Quantified Preconditions

Finally, UCPOP operators can have existentially quantified preconditions such as $\exists_{type(x)} cond(x)$, which give rise to open conditions of type $\exists_{type(x)} cond(x)@s$. Although existentially quantified preconditions can be handled as large disjunctive preconditions, there is another easier method. Specifically, they can be handled just as any other partially instantiated open conditions. For example, the existentially quantified open condition $\exists_{type(x)} cond(x)@s$ is equivalent to $(type(x) \wedge cond(x))@s$, which even SNLP can handle (an open condition $cond(x)@s$ can be established by the effect $cond(y)$ of a step s' if y can be bound to x). Thus, once again, existentially quantified preconditions do not necessitate any changes to EBL.

Since existentially quantified preconditions are treated as partially instantiated open conditions, they will inherit the soundness problems associated with the failures involving the latter (see Section 5). In particular, the number of establishment possibilities for an existentially quantified goal, and consequently its failure explanation, can be dependent on the number of objects of that type in the domain. For example, if the initial state has two blocks, A and B, then it is not possible to satisfy the conjunctive goal $on(A, B) \wedge \exists_{block(x)} On(x, Table) \wedge Clear(x)$. But if there is another block C, then the failure will not happen.

To ensure the soundness of failure explanations learned through search trees involving establishment of existentially quantified goals, we could either qualify the failure explanation with a constraint stating that no more objects of that type will be available, or do some counterfactual reasoning (as explained in Section 5). No such problem arises in the case of universally quantified preconditions since for a universally quantified condition to fail, it is enough that any one of the instantiated preconditions fail.

8.2 Operator effects

We will now discuss the changes brought about by the fact that UCPOP operators can have quantified and conditional effects.

8.2.1 Conditional effects

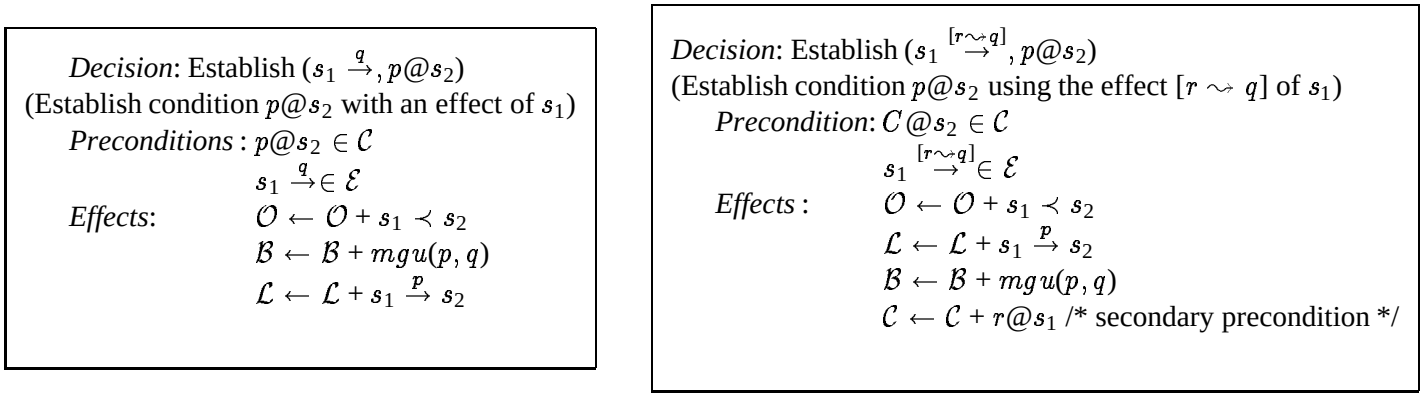
In *UCPOP*, the effects of an operator are represented in the form of $[prec \rightsquigarrow effect]$. While the preconditions of an operator should be true for it to be applicable, a particular effect $[prec \rightsquigarrow effect]$ will be true only when the antecedent conditions *prec* are true in the state proceeding the operator (these are thus called the secondary preconditions; [37]). As a special case, we can think of non-conditional effects used by SNLP as effects with secondary precondition ''True''. The presence of conditional effects has direct ramifications on establishment and threat resolution processes of the planner. It also has indirect ramifications on the type of analytical failures encountered by the planner.

Consequence of Conditional Effects on Establishment: Because of the conditional effects, unlike SNLP, which selects an *operator* to support a link in the establishment decision, *UCPOP* selects an *effect* of an operator to support a link, and to make sure that this conditional effect will occur it adds the antecedent of the conditional effect as an extra precondition for the plan. Specifically, if UCPOP is establishing the precondition $p@s$ with the help of the conditional effect $[r \rightsquigarrow q]$ of step s', it not only ensures that s' comes before s, and that q necessarily codesignates with p, but it also adds an additional precondition $r@s'$ to the plan. This is treated as any other open condition flaw in subsequent iterations.

From EBL point of view, the presence of conditional effect changes the characterization of the establishment decisions, (both *simple establishment* and *step addition*), which in turn leads to changes on regression of failure explanations over establishment decisions. Figure 24 shows the the description of simple establishment decision in *UCPOP* and SNLP. The main differences are that in the case of UCPOP, the establishment decision also augments the precondition constraints of the plan (by adding the secondary preconditions of the conditional effect used in the establishment), and this needs to be taken into account during regression. For example, when the precondition constraint $r@s_1$ is regressed over the establishment decision $establish(s_1 \overset{[r \rightsquigarrow q]}{\rightarrow}, q@s_2)$, it results in $True$.

Consequence of Conditional Effects on Threat Resolution: The presence of conditional effects also affects the way threats are resolved. For example, suppose the effect $[r \rightsquigarrow \neg q]$ of a step s_t is threatening the causal link $s_1 \overset{q}{\rightarrow} s_2$. In addition to the standard promotion and demotion possibilities, we can also resolve this threat by ensuring that s_t does not delete q. This can be done by simply adding $\neg r@s_t$ as an additional precondition of s_t. This way of resolving a threat is called on''confrontation.''

From the EBL point of view, confrontation is an additional planning decision, and failure explanations will have to be appropriately regressed over this decision.



(a) *Simple Establishment Decision in SNLP*

(b) *Simple Establishment Decision in UCPOP*

Fig. 24. Comparison of Simple establishment Decisions in SNLP and *UCPOP*

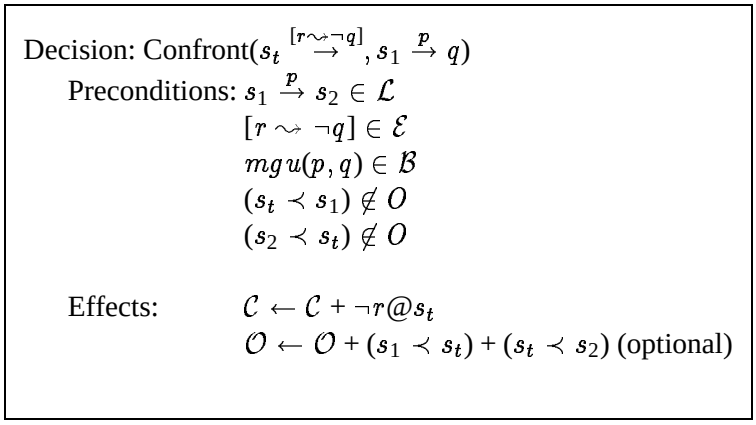


Fig. 25. Confrontation Decision

Figure 25 characterizes the preconditions and effects of the confrontation decision. Note that from the point of view of completeness, it is not strictly necessary to order s_t between s_1 and s_2 during confrontation. But, doing so will help in reducing the redundancy in the search space.

Consequence of Conditional Effects on Analytical Failures: Presence of operators with conditional effects has an indirect ramification on the types of analytical failures detected by the planner. In addition to ordering, binding and establishment failures that are automatically detected by SNLP, UCPOP can also detect failures involving inconsistent precondition constraints. For example, suppose a step s has two conditional effects $s \xrightarrow{[p \rightsquigarrow q]}$ and $s \xrightarrow{[p \rightsquigarrow \neg r]}$. It is possible that during planning, UCPOP attempts to use the first effect to establish the precondition q of some step, thereby adding a precondition constraint $p@s$. Suppose later UCPOP finds that there is a conflict between the second effect and some causal link, and decides to resolve it by confrontation. This leads it to add a precondition constraint $\neg p@s$. Now the two constraints $p@s$ and $\neg p@s$ are mutually incompatible, and UCPOP

can thus singal a failure, with the failure explanation being $p@s \wedge \neg p@s$.

8.2.2 Universally quantified conditional effects

Operators in UCPOP can have universally quantified effects of the form $s \overset{\forall_{type(x)}\,[p(x) \rightsquigarrow e(x)]}{\longrightarrow}$. Universally quantified effects are treated by UCPOP as if they are a shorthand notation for a conjunction of non-quantified effects of the form

$$s \overset{[p(o_1) \rightsquigarrow e(o_1)]}{\longrightarrow} \wedge \cdots \wedge s \overset{[p(o_n) \rightsquigarrow e(o_n)]}{\longrightarrow},$$

where $o_1, \cdots, o_n$ are all the objects of type $type$.

From the EBL point of view, regression over step-addition decisions that add steps with universally quantified effects can be handled in a way that is very similar to the handling of regression over steps with universally quantified preconditions. Specifically, when an effect $s \overset{[p(a) \rightsquigarrow e(a)]}{\longrightarrow}$ is regressed over a step addition decision that added a step having a universally quantified effect $s \overset{\forall_{type(x)}\,[p(x) \rightsquigarrow e(x)]}{\longrightarrow}$, it regresses to `initially–true`$(type(a))$.

8.3 Example

We shall now pull together the discussion regarding UCPOP+EBL, through an example problem from the briefcase domain described in Figure 22. The problem involves getting an empty briefcase to the office, while leaving everything else at home. Suppose the pay check P is in the briefcase, and the briefcase is at home in the initial state. The goal for this problem is specified by the formula $\forall_{obj(x)} at(x, H) \wedge at(B, O) \wedge closed(B)$. Figure 26 shows a trace of UCPOP solving this problem (for now ignore the explanation boxes on the right). The process starts with UCPOP selecting the precondition $\forall_{obj(x)} at(x, H)$ for establishment. Since this is a quantified condition, and since P is the only object in the domain, the condition $at(P, H)$ is added in place of the quantified condition. Next, UCPOP picks up the open conditions $closed(B)@G$, and $at(P, H)@G$ and establishes them with the help of the effects of the initial state. The condition $at(B, O)@G$ is established by adding a step `move–B`(H, O). At this point, the effect $\forall_{obj(x)} in(x, B) \Rightarrow \neg at(x, H)$ of the `move–B`(H, O) action threatens the link $0 \overset{at(P,H)}{\longrightarrow} G$. This is handled by the confrontation choice, which adds the secondary precondition $\neg in(P, B)$ to the `move–B` step. To establish this condition, the action `take–out`(P) is added to the plan. The effect $\neg closed(B)$ of this action threatens the link $0 \overset{closed(B)}{\longrightarrow} G$. This threat cannot be resolved since the promotion, demotion and confrontation choices all fail. The search process will eventually backtrack over the decision to establish $closed(B)$ from initial state, and find a solution in other branches.

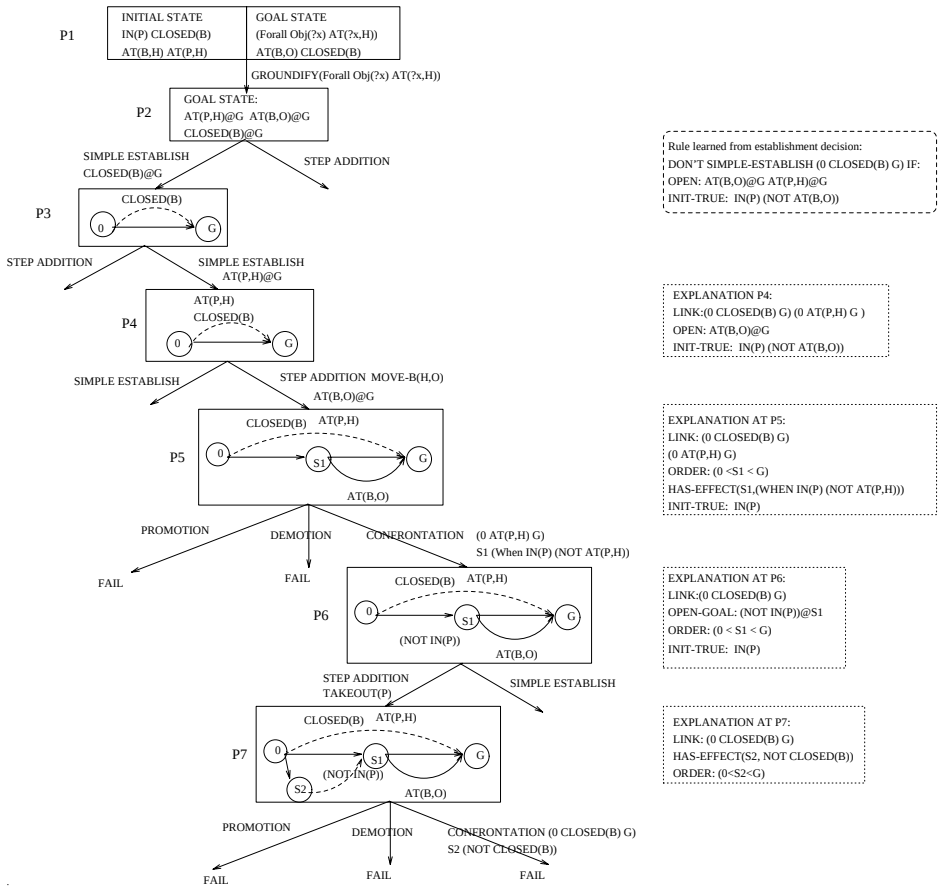


Fig. 26. Trace of UCPOP+EBL solving a problem in the briefcase domain: A failing path. (used as the Running Example)

Since all the branches under the plan P_7 fail, the partial plan P_7 itself fails. The failure explanation is constructed by conjoining the regressed failure explanations of the three branches with the description of the flaw in P_7 that is being removed. Specifically, we get the failure explanation of P_7 as

$$E_7 : \left\{ \begin{array}{c} \underbrace{0 \stackrel{closed(B)}{\longrightarrow} G \wedge (s_2 \not\prec 0) \wedge (G \not\prec s_2) \wedge \texttt{has\_effect}(s_2, \neg closed(B))}_{\text{unsafe link flaw}} \\ \wedge \underbrace{(0 \prec s_2) \wedge (s_2 \prec G)}_{\text{regressed from children}} \end{array} \right.$$

(The ordering constraints simplify to $0 \prec s_2 \prec G$.) This explanation is then regressed over the step addition decision that adds `take_out`(P), and the process continues as shown in Figure 26, eventually computing the failure explanation of

P_3 as

$$E_3 : \begin{cases} 0 \overset{closed(B}{\rightarrow} G \wedge at(B, O)@G \wedge at(P, H)@G \\ \qquad \wedge \texttt{init\_true}(in(P)) \\ \qquad \wedge \texttt{init\_true}(\neg at(B, O)) \end{cases}$$

When E_3 is regressed over the simple establishment decision under P_2, we get

$$at(B, O)@G \wedge at(P, H)@G \wedge \texttt{init\_true}(in(P)) \\ \wedge \texttt{init\_true}(\neg at(B, O)) \wedge \texttt{init\_true}(closed(B)).$$

This leads to a useful control rule, shown at the top right corner of Figure 26, which states that simple establishment of $closed(B)@G$ should be avoided when the paycheck is in the briefcase, briefcase is not at office, and we want the briefcase to be at the office and paycheck to be left at home. The generalized form of the rule learned at P_2 will be:

if $at(B, x_o)@s \wedge at(x_p, x_h)@s \wedge (x_o \not\approx x_h)$
$\wedge \texttt{initially\_true}(in(x_p)) \wedge \texttt{initially\_true}(\neg at(B, x_o))$
then Reject simple establishment of $closed(x)@s$ from initial state.

9 Experimental Evaluation of UCPOP+EBL

To evaluate the performance of UCPOP+EBL we conducted experiments in two different domains -- the first one is a variant of the briefcase domain (Figure 22(a)) that has multiple locations, and multiple objects to be transported among those locations using the briefcase. This domain is similar in character to the logistics transportation domains, except with conditional and quantified effects. We generated 100 random problems containing between 3 to 5 objects, 3 to 5 locations and between 3 to 5 goal conjuncts. The second domain is the blocks world domain called `BW-quant` described in Figure 21. We generated 100 random problems using the procedure described in [31]. The problems contained between 3 to 6 blocks, and 3 to 4 goals. In each domain, we compared the performance of the from-scratch planner with that of the planner using the search control rules generated by UCPOP+EBL. Table 2 shows the results of these experiments. As can be seen, UCPOP+EBL achieves significant savings in performance in both the domains, both in terms of the number of problems solved, and the speedup obtained. To gauge the cost of learning itself, we also ran UCPOP+EBL in an ''*online learning*'' mode, where it continually learns control rules and uses them in the future problems. The statistics in the *online learning* column show that the cost of learning does not outweigh its benefits.

Domain	From-scratch		Online Learning		Using learned rules	
	% Solv	cpu	% Solv	cpu	% Solv	cpu
Brief Case Dom	49%	6568	92%	1174 (*5.6X*)	98%	1146 (*5.6X*)
Blocks World	53%	7205	100%	191(*38X*)	100%	190 (*38x*)

Table 2
Performance of UCPOP+EBL in Blocks world (`BW-quant`) and Briefcase Domain

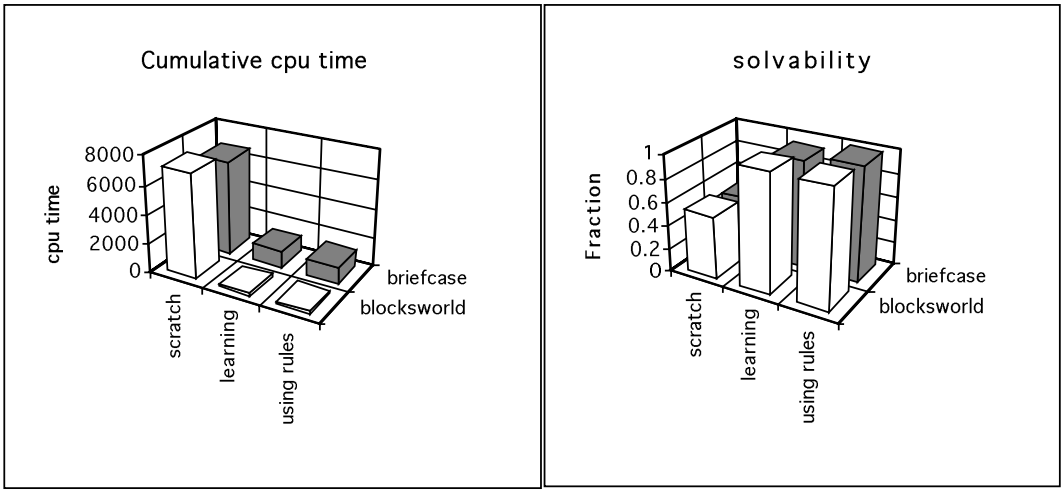


Fig. 27. Plots summarizing the performance of UCPOP+EBL in Blocks world (`BW-quant`) and Briefcase domain. Note that both the time taken for solving the problems, as well as the number of problems solved in within the resource limits improve through EBL. The plots labeled ‘‘learning’’ show the total cost for learning the rules and using them in an on-line fashion, while the plots labeled ‘‘using rules’’ show the cost of using the previously learned rules

10 Factors influencing the effectiveness of EBL in Planning

Although learning search control rules is an attractive way of improving planning performance, there are a variety of factors that can affect the utility of control rule learning in a given domain. In particular, the nature of the domain theory, and the nature of the base level search strategy used by the planner can have a significant impact on the effectiveness of learning control rules from analytical failures. Furthermore, as discussed elsewhere, the ability of EBL to learn control rules crucially depends on detecting and explaining failures in the partial plans before they cross depth limits [28]. This in turn depends on the nature of the domain theory (viz, how much information is left implicit and how much is represented explicitly), and the availability of domain specific theories of failure (c.f. [28,2]). Finally, the availability of sophisticated dependency directed backtracking strategies can directly compete with the performance improvements produced through learned control rules.

We have used our implementation of UCPOP+EBL to investigate the effect of these factors. In this section, we will describe the results from these studies, and analyze them.

Domains: To evaluate the effect of expressive representations on the performance of EBL systems, we tried three different domain theories of the blocks world domain, as shown in Figure 21. The first, called `BW-prop`, contains two types of predicates, On and $Clear$, and two actions, $puton$ and $newtower$ with no conditional effects. The second, called `BW-cond`, contains the same two predicates, but with a single $puton$ action with conditional effects. The third domain, called `BW-quant`, contains a single predicate On, with the condition $Clear(x)$ replaced by the quantified formula $Table(x) \vee \forall_y \neg On(y, x)$. Note that `BW-prop` is forced to make a choice between its two actions, while `BW-cond` and `BW-quant` don't have to make such premature commitment. Similarly, because of their restricted language, `BW-cond` and `BW-prop` are forced to hide the relation between $Clear$ and On, while the expressive language of `BW-quant` allows it to make the relation explicit.

Experimental setup: The experiments consisted of three *phases*, each corresponding to the use of one of the three domain descriptions above. In each phase, the same set of 100 randomly generated blocks world problems were used to test the from-scratch and learning performance of UCPOP+EBL, and statistics regarding the number of problems solved and the cumulative cpu time were collected. To understand the effect of domain specific failure theories on the performance, we ran UCPOP+EBL in three different *modes*. In the first mode, UCPOP+EBL's learning component was turned off, and the problems were solved from scratch by the base level planner, UCPOP. In the second mode, UCPOP+EBL was trained over the problems, and the control rules it learned from the analytical failures alone were used in solving the test set. The third mode was similar to the second mode except that UCPOP+EBL was also provided domain specific theories of failure in the form of domain axioms (such as the one stating that $\forall_{x,y} clear(x) \supset \neg On(y, x)$), which could be used to detect and explain failures that would otherwise not be detected by the base level planner.

Since it is well-known that the performance of a plan-space planner depends critically on the order in which open condition flaws are handled (goal selection order) [21], we experimented with two goal-selection strategies -- one which corresponds to a LIFO strategy and one that works on goals with the least number of variables left uninstantiated, called MIGF strategy. [19]

[19] Since partially instantiated goals have larger number of establishment possibilities, this goal selection strategy approximates the least-cost flaw refinement strategy, [21]

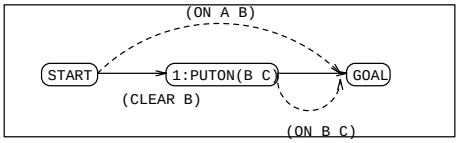
Domain	I. Scratch		II. EBL with analytical failures			III. EBL with dom. spec. fail. theories		
	% Solv	cpu	% Solv	cpu	# rules.	% Solv	cpu	# rules.
Achieving most instantiated goals first (MIGF)								
BW-prop	51%	7872	69%	5350 (*1.5x*)	24	68%	5410 (*1.5x*)	28
BW-cond	89%	2821	88%	2933 (*0.96x*)	15	91%	2567 (*1.1x*)	37
BW-quant	53%	7205	100%	210(*34x*)	4	100%	210 (*34x*)	4
Achieving goals in a LIFO order								
BW-prop	10%	13509	10%	13505 (*1x*)	30	10%	13509 (*1x*)	30
BW-cond	42%	9439	60%	6954 (*1.4x*)	14	75%	4544 (*2.1x*)	36
BW-quant	81%	3126	89%	2136 (*1.5x*)	32	94%	1699 (*1.8x*)	37

Table 3
Performance of UCPOP+EBL in the three blocks world domains (with DDB disabled)

Results: Table 3 shows the statistics from these experiments. The performance of the base level planner varies considerably across the three domains and the two goal selection strategies. What is more interesting, the effectiveness of the learned control rules in improving the planning performance also varies significantly, giving a speedup anywhere between 0.95x and 38x. We will now analyze this variation with respect to several factors.

10.1 Expressive Domain Theories and Analytical Failures

The results show that the magnitude of improvement provided by UCPOP+EBL when learning from analytical failures alone (mode II) depends on the nature of the domain theory. For example, we note that search control rules learned from analytical failures improve performance more significantly in `BW-quant` than they do in `BW-prop` and `BW-cond`(see Table 3). This can be explained by the fact that in the latter two domains, in many cases, the base level planner is not able to detect any analytical failures in various branches before the planner crosses the depth-limit. In contrast, the explicitness of the domain description in `BW-quant` enables the planner to detect and explain analytical failures in many more situations. To illustrate this point, consider the blocks world partial plan:



Given the relation between $Clear$ and On predicates, it is clear that this plan cannot be refined into a solution (since it is impossible to protect the condition $On(A, B)$ while still ensuring the precondition $Clear(B)$ of step 1). However, since the relation between $Clear$ and On is not explicit in `BW-prop` and `BW-cond`, the planner may not recognize this failure in those domains before crossing the depth limit (unless some domain specific theories of failure are provided). In contrast, in `BW-quant`, the precondition $Clear(B)$ will be expressed as the quantified condition $\forall_y \neg On(y, B)$, and the planner will immediately notice an analytical failure, when trying to add a step to achieve $\neg On(A, B)$ at step 1 (since any step giving $\neg On(A, B)$ will threaten the causal link supporting $On(A, B)$.

10.2 Expressive domain theories and domain specific failure theories

We note that the availability of domain specific theories of failure does not uniformly improve performance of EBL. In particular, we see a bigger improvement in `BW-cond` than we do in `BW-quant` (Table 3). UCPOP+EBL improves the

solvability from 60% to 75% in LIFO goal ordering and 88% to 91% in the MIGF goal order. The number of rules learned also increases from 14 to 36, and 15 to 37 respectively in MIGF and LIFO goal orders. In contrast, the improvements are smaller in the case of `BW-quant`. This can be explained by the fact that the information in the domain axioms (which constitute our domain-specific theories of failure), is subsumed to a large extent by the information in the quantified preconditions and effects of the actions in `BW-quant`. The situation is opposite in the case of `BW-cond`, and it benefits from the availability of domain specific theories of failure.

10.3 Importance of Explainable Failures

Another interesting point, brought about by the results above is the correlation between the performance of the base level planner and the performance of the planner in the presence of learned control rules. From Table 3, we note that the planner performs poorly in the `BW-quant` domain, compared to `BW-cond` domain in the from-scratch mode (with 53% solvability as against 89%), but out-performs it with learning (100% solvability as against 91%). At first glance, this might suggest the hypothesis that the planner that makes more mistakes in the from-scratch phase has more *opportunities* to learn from. This hypothesis is not strictly true -- in particular, it is not the number of mistakes, but rather the number of *explainable mistakes* that provide learning opportunities. As an example, `BW-prop`, which also does worse than `BW-cond` in from-scratch mode, continues to do worse with learning.

10.4 Effect of Sophisticated Backtracking Strategies

One other factor that influences the utility of control rules learned from EBL is the default backtracking strategy used by the planner. In Section 4.4.1, we noticed that the analysis being done by UCPOP+EBL in learning control rules also helps it do a powerful form of dependency directed backtracking (DDB). To understand how much the improvement brought about by dependency directed backtracking affects the utility of control rules learned through the EBL analysis, we repeated our experiments while making the base level planner use dependency directed backtracking. Table 4 shows these results, and Figure 28 compares the performance of UCPOP+EBL when DDB, EBL and domain specific failure theories are used (for the LIFO goal order case).

The first thing we note is that the impact of EBL reduces in the presence of DDB in both goal ordering strategies (compare the numbers in Tables 4 and 3. This should not in itself be surprising since both EBL and DDB draw strength from the same processes of regression and propagation of failure explanations (see Section 4). For

Domain	I. Scratch		II. EBL with analytical failures			III. EBL with dom. spec. fail. theories		
	% Solv	cpu	% Solv	cpu	# rules.	% Solv	cpu	# rules.
Achieving most instantiated goals first (MIGF)								
BW-prop	71%	5093	61%	6613 (*0.8x*)	24	70%	5193 (*1.0x*)	28
BW-cond	89%	2837	88%	2983 (*0.8x*)	15	95%	1835 (*1.6x*)	37
BW-quant	100%	197	100%	190(*1.03x*)	4	100%	190 (*1.03x*)	4
Achieving goals in a LIFO order								
BW-prop	22%	12001	21%	12054 (*0.97x*)	30	21%	12080 (*0.98x*)	36
BW-cond	42%	9439	60%	7666 (*1.2x*)	14	75%	4544 (*2.1x*)	29
BW-quant	90%	1640	96%	1175 (*1.4x*)	32	98%	1146 (*1.4x*)	37

Table 4
Performance of UCPOP+EBL in the three blocks world domains, when the base level planner uses a dependency directed backtracking strategy

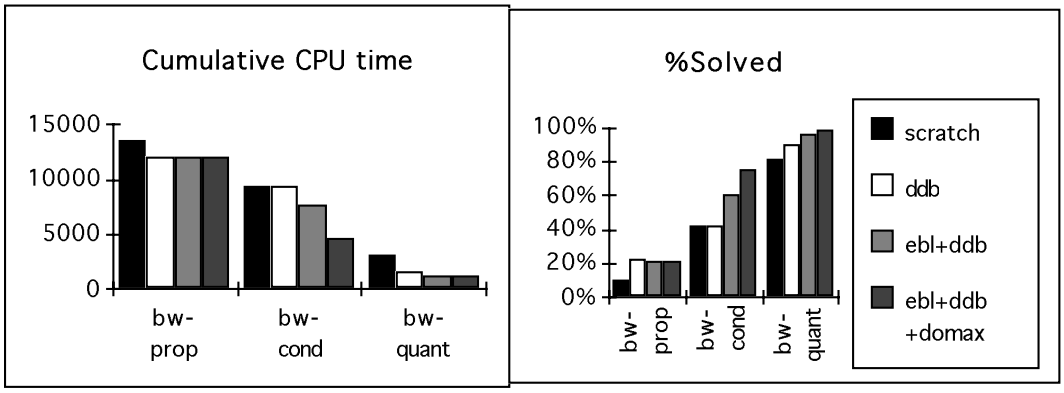


Fig. 28. Plots summarizing the interaction between DDB, EBL and domain specific failure theories (in the LIFO goal order)

example, in comparing the statistics in Table 3 and Table 4, we note that there is a strong correlation between the planner's ability to improve its performance through DDB, and its ability to learn useful control rules from analytical failures alone. For example, `BW-cond` is unable to improve its performance with DDB or with control rules learned from analytical failures in the ''most instantiated goals first'' case. This is not surprising since lack of detectable analytical failures will hurt both the effectiveness of DDB and that of EBL. (Similar relation has been observed in the constraint satisfaction literature between back-jumping and learning [9]).

What is interesting is that EBL is able to outperform DDB at least in some cases. Specifically, in the case of the LIFO goal selection strategy, control rules do bring out significant additional savings over DDB (see Table 3). This tendency was also confirmed by our experiments in the briefcase domain (not shown in the table). It can be explained by the fact that although DDB captures much of the analysis done while learning control rules, it is effective *only after a failure has been encountered, and backtracking is started*. In contrast, control rules attempt to steer the planner away from the failing branches in the first place. In cases when the average match cost of the control rules is smaller than cost of backtracking from the first failure, EBL can outperform DDB. Another factor is that while DDB typically tends to exploit only the analytical failures detected by the planner, the control rules learning may also benefit from domain specific failure theories.

11 Related Work and Discussion

11.1 Relation to other search control rule learning frameworks

As we mentioned earlier, significant research has been done towards applying EBL techniques to state-space planners. Two such systems are closely related to our work. The first, PRODIGY+EBL, was developed by Minton [31] . It learns

search control rules and improves performance of a state-space planer using means ends analysis. The second, FAILSAFE, was developed by Bhatnagar and Mostow [2,3]. It learns search control rules for a forward searching state-space planner. The primary difference between these efforts and our work is that we adapt EBL to plan-space planners. In some ways, the SNLP+EBL/UCPOP+EBL frameworks can be seen as a generalization of the EBL techniques for state space planning. In particular, recent work [27] shows that partial order and state space planning approaches can be cast and combined in a single refinement planning framework, that is not very different from the one used in this paper.

Ignoring the differences brought about by the differences in base-level planners, the learning strategies used by our systems have some interesting relations to PRODIGY+EBL and FAILSAFE. We discuss these briefly below:

Learning opportunities: PRODIGY+EBL learns from a variety of target concepts including failures, successes and subgoal interactions, while FAILSAFE uses failures and subgoal interactions alone. In contrast, both SNLP+EBL and UCPOP+EBL restrict their learning to failures. Since partial order planner do not need to backtrack on goal ordering decisions, target concepts based on subgoal interactions are not relevant for SNLP+EBL and UCPOP+EBL. While learning from successes is important, we believe that macro learning strategies such as reuse and replay are more effective in doing this [17] (see Section 11.2).

Interaction between Learner and planner: In PRODIGY+EBL, the learning starts *after* the planning phase is completed. In contrast, both FAILSAFE, and our systems SNLP+EBL, UCPOP+EBL do on-line learning, where the learning component is activated any time the planner encounters a failure. One advantage of the on-line learning process in SNLP+EBL and UCPOP+EBL is that the regression and propagation analysis also provides a powerful framework for dependency directed backtracking, that can speedup the base-level planner.

Properties of learned rules: As we discussed in Section 5, SNLP+EBL and UCPOP+EBL aim to learn search control rules that are sound in that they do not affect the completeness of the planner. While PRODIGY+EBL shares this goal, FAILSAFE does not. Specifically, FAILSAFE learns over-general control rules, called censors, by declaring failures early on during the search, building incomplete proofs of the failures, and learning censors from these proofs. The censors speed up search by pruning away more and more of the space until a solution is found in the remaining space. To learn quickly, the technique over-generalizes by assuming that the learned censors are preservable, i.e., remain un-violated along at least one solution path. A recovery mechanism heuristically detects violations of this assumption and selectively specializes censors that violate the assumption. It

remains to be seen as to what extent such an adaptive learning mechanism could be useful for plan-space planners (see Section 12.1).

Initiating learning: The three systems follow slightly different methods in initiating the learning cycle. PRODIGY+EBL initiates learning by analyzing the full search tree, looking for instances of its target concepts in the search tree. Once it finds these target concepts, it then specializes them with respect to the search tree to learn search control rules. Both SNLP+EBL/UCPOP+EBL and FAILSAFE initiate learning any time they encounter a failure or cross depth limits. They however differ in terms of the way failures are explained. In the case of SNLP+EBL/UCPOP+EBL, failure explanations are a minimal set of mutually inconsistent constraints. When such explanations cannot be extracted SNLP+EBL/UCPOP+EBL avoid learning from that failure. In contrast, since FAILSAFE allows overgeneral search control rules, it does not have to rely on its failure explanations being sound. In fact, FAILSAFE does not guarantee any formal properties of soundness for the failure explanations.

Monitoring Utility: Although search control rules can improve the planner's performance, like any other deductive learning strategy, they suffer from the potential utility problem. Specifically, it is possible to populate the rule database with many rules which have little search reduction potential, and whose amortized match cost far outstrips their potential search improvements. To prune rules of questionable utility, PRODIGY+EBL tracks the usage statistics associated with the control rules, including the application frequency, match cost and reduction in search entailed by the rule, when it is applicable. More recent work such as that by Gratch and DeJong [20] provides a sound statistical basis for such utility models. SNLP+EBL and UCPOP+EBL currently do not use any sophisticated models for tracking the utility of learned rules. The only mechanism that is currently used involves learning control rules only when the size of the search tree pruned by those rules in the training case is above a specified threshold. One reason why we did not find utility problem a bottleneck until now may be that SNLP+EBL and UCPOP+EBL learn only from a single target concept -- failures for which sound explanations can be given. We believe however that as the complexity of the domains increase, the utility models such as those developed in [20] can be profitably adapted to our EBL framework (see Section 12.1).

11.2 Relation to EBL methods that do not learn search control rules

Although our work is the first to adapt failure based search control rule learning to partial order planning, the general explanation based generalization framework has been applied to partial order planners in the past. Kambhampati and Kedar [24]

describe how partial order plans can be generalized in a variety of ways based on their explanations of success. Similar methods were also developed independently by Chien and DeJong [5,6].

While learning search control rules is one possible way of exploiting the explanation based analysis of failures, this is by no means the only way. In particular, it is possible to use similar analyses in conjunction with other types of speedup learning frameworks, such as plan reuse and/or replay. As an example, recently, we adapted the SNLP+EBL framework to learn to improve case-retrieval based on previous replay failures [17].

11.3 Relation to speedup learning methods that use static analyses

Although most EBL systems base their learning on the experiences of the underlying problem solver, this is not always required. In particular, it is theoretically possible to derive equivalent (and some times more general) search control knowledge by simply analyzing the domain theory used by the problem solver. Such static analyses have been the basis of some EBL systems, such as the PRODIGY/STATIC system by Etzioni [13]. This system analyzes a structure called the ''Problem Space Graph (PSG)'' -- which is a graphical structure capturing the precondition/effect dependencies between the actions in the domain -- to detect necessary interactions between different types of subgoals. This analysis is used to come up with operator rejection and goal ordering rules for a state-based means-ends analysis planner. Although goal-ordering rules are not relevant for partial order planners, operator rejection rules will of course be relevant.[20] For example, the rule rejecting the operator $Roll(x)$ in job shop scheduling domain (see Figure 15) can be learned through analysis of PSG. It is not clear whether this type of analysis can be extended to learn simple establishment possibilities, such as the first rule in Figure 19.

As Etzioni points out [41], both static and dynamic search control rule learning methods have their advantages. For example, while static methods can improve the performance of the planner before the first failure is even encountered, the dynamic methods have the ability to exploit the problem distribution and the default behavior of the problem solver. Etzioni and Perez [41] describe a way of combining static and dynamic analyses to exploit both their advantages.

There has been some work on using PSG-like structures in partial order planning. Smith and Peot [47] propose structures called ''Operator Graphs.''[21], and show that

[20] It is not clear from Etzioni's papers whether the goal-ordering rules or the operator rejection rules had more effect on the performance of PRODIGY.

[21] Unlike PSGs, that are domain-specific, the operator graphs are problem-specific, and are constructed for each problem. There are also some differences between PSGs and operator graphs in terms of when the construction is terminated.

operator graphs can provide a variety of search control strategies for partial order planners [19, Section 3], including analysis on which types of unsafe link conflicts can be postponed indefinitely, thereby improving performance of the planner. It would be interesting to see how our dynamic search control rule learning methods for partial order planning, can be integrated with the operator graph methods such as those being developed by Smith and Peot [47].

In [14], Etzioni also develops a structural theory of EBL that attempts to explain what features of the problem spaces (domains) are predictive of the effectiveness of EBL. Our work complements and extends this theory in several directions. In particular, the empirical results discussed in Section 10 show a high correlation between the effectiveness of EBL and the presence of explainable failures, and discuss how these are affected by the domain encoding and availability of domain specific failure theories. We conclude that encodings that allow more analytical failures to be explicitly detected by the planner facilitate learning without recourse to additional domain specific failure theories. In addition, as we discuss below, our experiments confirm an open hypothesis regarding the effect of DDB on the effectiveness of EBL.

11.4 Relation to work on dependency directed backtracking

One of the important side-benefits of the EBL framework used in SNLP+EBL and UCPOP+EBL is a systematic method for doing dependency directed backtracking in planning. Although there has been a significant amount of work on dependency directed backtracking in the the constraint satisfaction community[48], very little such work has been done in planning. There do exist planning systems, such as OPLAN-2 [7], that claim to use some form of ‘‘intelligent’’ backtracking. Typically, such methods are driven by a carefully constructed decision dependency graph (c.f. [8]). The regression and propagation based approach for dependency directed backtracking provides an interesting alternative that does not require explicit construction of decision graphs.

The affinity between dependency directed backtracking and learning has been observed in the CSP literature. In particular, Dechter and her co-workers [9,16] have done several empirical studies on the relative tradeoffs offered by the use of failure explanations in guiding dependency directed backtracking, vs. using them to learn node rejection rules. Their conclusions are similar to those we reached in our experiments (Section 10.4).

In [14], Etzioni hypothesizes that DDB reduces the impact of EBL, but leaves the verification of the hypothesis for future work. Our experimental results in Section 10.4 can be seen as a partial confirmation and refinement of Etzioni’s hypothesis. In particular, we not only show that DDB and EBL derive their

effectiveness from the same computational sources of regression and propagation, we also point out that when the cost of matching control rules is smaller than the cost of backtracking from the first failure, EBL can potentially outperform DDB.

It is also possible to exploit the DDB component of UCPOP+EBL more effectively than we did in the current work. Sadeh et. al. [46] discuss a variety of techniques for improving DDB, including changing the flaw resolution order such that the flaw whose resolution lead to the latest failure is tried first after DDB (this is an instance of the ‘‘fail-first’’ principle [48]). In [43], we also discuss the relations between DDB as presented here, and many other intelligent backtracking schemes, including dynamic backtracking [18].

11.5 Effect of the default search strategy on performance of EBL

The discussion about the effect of sophisticated backtracking strategies on the impact of EBL brings to fore the more general issue of the impact of the default search strategy used by the planner on the effectiveness of EBL. Like most previous EBL frameworks, UCPOP+EBL also uses a depth first regime [2,32]. The main reason for the use of depth-first search is to force the planner to continue each line of enquiry until it encounters a failure. These failures then help the EBL component to formulate search control rules that will avoid unpromising branches of inquiry in future. Best-first search regimes often change search direction before a given line of enquiry leads to failure, and thus they do not provide effective support for EBL. Note that depth-first search is only required during learning phase. Once learned, the search control rules can be used in non-depth-first search regimes also.

12 Conclusions and Future Work

In this paper, we presented SNLP+EBL, the first systematic framework for learning from failures of a partial order planner, using an explanation based analysis. We have described the various ways in which failures are detected and explained during planning. We then discussed how the failure explanations of the interior nodes are computed through regression and propagation, and how the resulting explanations are converted into search control rules. We have shown that the search control rules generated by SNLP+EBLare sound, and explained how they can be generalized without loss of soundness. Our discussion shows that name-insensitive theories are particularly useful in facilitating simple generalization algorithms.

We have also noted that the regression and propagation processes can facilitate a powerful form of dependency directed backtracking. We have then presented experimental results showing that the search control rules that SNLP+EBL learns

using our techniques enable it to outperform SNLP.

Next, we demonstrated the extensibility of our EBL framework by showing how it can be easily adapted to UCPOP, a more powerful descendant of SNLP, that allows quantified and conditional formulas in its operator language. We described empirical studies in a quantified encoding of blocks world domain (`BW-quant`) and a simple transportation domain, that demonstrate that UCPOP+EBL provides significant performance improvements.

Finally, we presented an empirical analysis of the factors that influence the effectiveness of explanation based search control rule learning for partial order planners. We used UCPOP+EBL as a basis to investigate the effect of expressive action representations, heuristic goal selection strategies and sophisticated backtracking algorithms on the effectiveness of control rule learning. In particular, we showed that expressive action representations facilitate the use of richer domain descriptions, which in turn increase the effectiveness of learning control rules from analytical failures. This reduces the need for domain specific failure theories to guide EBL. We also noted the strong affinity between dependency directed backtracking and control rule learning, and showed that despite this affinity, control rules can still improve the performance of a planner using dependency directed backtracking.

12.1 Limitations and Future Directions

There are several ways in which our learning framework can be improved. Our approaches until now have concentrated on finding and explaining inconsistencies in partial plans generated by the base-level planner. Unfortunately, this is still inadequate in doing effective learning in some domains. In the following, we discuss two extensions that we are currently pursuing.

The first approach is to expand the notion of failure to include not just the inconsistencies among the constraints of the partial plan, but also the inability of the partial plan to lead to a useful solution. For example, partial order planners exhibit ‘‘looping’’ whrein they spend inordinate amounts of time doing many locally relevant but globally unpromising refinements to a partial plan. Recently, we showed that much of the looping in partial order planning can be tied to production of non-minimal plans (i.e., plans with redundant steps), and developed conditions under which such pruning strategies do not lead to loss of completeness [26]. We are currently investigating if it is possible to learn effective search control rules from such pruning techniques.

The second approach for improving the chances of failure detection is to relax the requirement for soundness of failure explanations. Although proving that a partial plan is inconsistent is hard, often we may know that the presence of a set of features is losely ‘‘indicative’’ of the unpromising nature of the partial plan. For example,

FAILSAFE system [2] constructs explanations that explicate *why the current node is not the goal node,* inspite of many refinements.[22]

Relaxing soundess requirement on failure explanations will allow UCPOP+EBL to learn with incomplete explanations, thus improving the number of learning opportunities. We are currently experimenting with a variant of this approach, where such partial explanations of failure are associated with numerical certainty factors between 0 and 1 (to signify their level of soundess). The explanation of failure of an interior node will have a certainty factor that depends on the certainty factors of the explanations of failure of its children nodes. Similarly, the search control rules learned from these failure explanations will also inherit the certainty factors of the explanations.

Of course, learning with unsound explanations of failure will lead UCPOP+EBL to learn unsound search control rules, which, if used as pruning rules, can affect the planners completeness. We propose to handle this by considering such search control rules to black-list, rather than prune plan refinements.

Although sacrificing soundness seems like a rather drastic step, it should be noted that ''correctness'' and ''utility'' of a search control rule are not necessarily related. Utility is a function of the problem distribution that is actually encountered by the planner, and thus, it is possible for a rule with lower certainty factor to have higher positive impact on the efficiency than one that is correct.[23]

A complementary approach to improving the effectiveness of EBL involves combining it with inductive learning methods. In particular, EBL methods can be used to isolate the features of the problem that are relevant to the failure and then inductive methods can be used to generalize over these partial explanations of failure (or success). Borrajo and Veloso [4] discuss an approach of this type in the context of a state-space planner, while Estlin and Mooney present a similar method in the context of partial order planning [12]. It would be interesting to see how such hybrid methods can be adapted to UCPOP+EBL.

---

[22] It is tempting to use the complete description of the unpromising plan as its own explanation of failure. However, this can seriously inhibit any useful learning from taking place. Once a partial plan P is given the constraints comprising P as the explanation of its failure, given the way the explanations of failure of the interior nodes are computed by the `propagate` procedure, no ancestor P' of P can ever have an explanation of failure simpler than P' itself. Thus, it is critically important to blame the failure on some (rather than all) constraints of the plan.

[23] As an analogy, consider a physician who has two diagnostic rules, one that is completely certain, but is about a relatively rare disease (e.g. ebola virus syndrome), and another which has low certainty, but is about a frequently occuring disease (e.g. common cold). Clearly, the latter rule may be much more useful for the physician practising in a US city, than the latter.

Another issue that needs to be carefully addressed is that of utility of learned rules. The utility problem was not critical until now as UCPOP+EBL learns only from failures, and leads to few rules because of the practical limit on the number explainable failures detected by the base-level planner. However, as we allow extensions such as learning with partially sound failure explanations, we are likely to learn rules whose match cost may outweigh their computational advantages. We believe however that the existing approaches to utility management in EBL will still be applicable for UCPOP+EBL. It would be interesting to integrate the rule utility monitoring approaches such as those embodied in the COMPOSER system [20] into UCPOP+EBL.

The framework for EBL and DDB, presented in this paper, applies with very little changes to constraint satisfaction problems. In particular, in [43], we adapt our framework to general refinement search, which subsumes many models of planning and constraint satisfaction problems. This formalization brings to fore the many similarities between the EBL and DDB work in CSP, Planning and Machine Learning communities, and facilitates cross-fertilization of the ideas from these hither-to disparate research streams.

Acknowledgements

This research is supported in part by NSF research initiation award (RIA) IRI-9210997, NSF young investigator award (NYI) IRI-9457634 and ARPA/Rome Laboratory planning initiative grant F30602-93-C-0039. We thank Bulusu Gopi Kumar for his feedback in the initial stages of this research, and Laurie Ihrig for her many critical comments on the previous drafts of this paper. Discussions with her significantly clarified our thinking on the issues of soundness of the search control rules. We also thank Tony Barrett, Dan Weld for making their implementations of SNLP and UCPOP available publicly, and Steve Minton for his clarifications on the PRODIGY/EBL system.

A List of Symbols

Symbol	Denotes
$\mathcal{C}$	The set of precondition constraints on the partial plan
$\mathcal{E}$	The set of effect constraints on the partial plan
$\mathcal{L}$	The set of causal link constraints on the partial plan
$\mathcal{O}$	The set of binding constraints on the partial plan
$\mathcal{S}$	The set of steps in the partial plan
$\mathcal{B}$	The set of binding (codesignation and non-codesignation) constraints on the partial plan
$s_0, 0$	Alternative names for the initial step of a partial plan. The effects of this step correspond to the assertions in the initial state of the planning problem.
s_∞, G	Alternative names for the final step of a partial plan. The preconditions of this step correspond to the assertions in the goal state of the planning problem.
$s\!:\!O$	Denotes that a step name s is mapped to an operator O
$x \approx y$	A codesignation constraint among variables, saying that x and y must take the same values
$x \not\approx y$	Non-codesignation constraint among variables, saying that x and y must not take the same values
$s \prec s'$	An ordering constraint among steps, saying that s must precede s'.
F	The set of constraints describing a flaw
$E, E', E'', E_i, \cdots$	Symbols for denoting failure explanations of partial plans (a failure explanation is a set of constraints on the partial plan that are inconsistent)
$d, d', d'', d_i, \cdots$	Symbols for denoting the planning decisions taken to refine one partial plan into another.
$P, P', P'', P_i, \cdots$	Symbols for denoting partial plans.
$n, n', n'', n_i, \cdots$	Symbols for denoting search nodes generated by SNLP+EBL (which contain the partial plans as well as any other search tree related information)
$P(n)$	The partial plan corresponding to a search node n
$d(n)$	The planning decision that lead to the search node n from its parent
$E(n)$	The failure explanation for the search node n
$parent(n)$	The search node from which n was produced (through the

Symbol	Denotes
$s \xrightarrow{e}$	The effect constraint denoting that the operator corresponding to the step s in the partial plan must have an effect s
$s \overset{[p \rightsquigarrow e]}{\longrightarrow}$	A conditional effect constraint denoting that the operator corresponding to the step s has a conditional effect ''If p then e''
`initially–true`(p)	Same as $0 \xrightarrow{s}$, denoting that the initial state must have an assertion p
$p@s$	The precondition constraint, denoting that the condition p must be true before the step s in the plan
$s \xrightarrow{p} q$	The causal link constraint saying that step s gives the condition p to the step q in the partial plan
$mgu(p, q)$	The set of binding constraints on variables of the partial plan such that the condition p and q necessarily codesignate
`init–step`(s)	A predictate that evaluates to true only if s is the initial step of the plan
$Regress(c, d)$	The result of regressing the constraint c over the planning decision d.
$Regress(E, d)$	The result of regressing the failure explanation E over the planning decision d.

References

[1] A. Barrett and D.S. Weld. Partial Order Planning: Evaluating Possible Efficiency Gains. *University of Washington, Technical Report 92-05-01, 1992*

[2] N. Bhatnagar. *On-line Learning From Search Failures* PhD thesis, Rutgers University, New Brunswick, NJ, 1992.

[3] N. Bhatnagar and J. Mostow. *On-line Learning From Search Failures Machine Learning*, Vol. 15, pp. 69-117, 1994.

[4] D. Borrajo and M. Veloso. Incremental learning of control knowledge for nonlinear problem solving. In *Proc. European Conference on Machine Learning*, 1994.

[5] S.A. Chien. *An Explanation-Based Learning Approach to Incremental Planning*. PhD thesis, TR. UIUCDCS-R-90-1646 (Ph.D. Thesis). Dept. of Computer Science, University of Illinois, Urbana, IL, 1990.

[6] S. Chien and G. DeJong. Constructing Simplified Plans via Truth Criteria Approximation. In *Proc. 2nd Intl. Conference on Artificial Intelligence Planning Systems* (pp. 19-24), June 1994.

[7] K. Currie and A. Tate. O-Plan: The Open Planning Architecture. *Artificial Intelligence*, 51(1), 1991.

[8] L. Daniel. Planning: Modifying non-linear plans *University Of Edinburgh, DAI Working Paper: 24*

[9] R. Dechter. Enhancement schemes for learning: Back-jumping, learning and cutset decomposition. *Artificial Intelligence*, Vol. 41, pp. 273-312, 1990.

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

[11] M. Drummond and K. Currie. Exploiting Temporal coherence in nonlinear plan construction. *Computational Intelligence*, 4(2):341-348, 1988.

[12] T. Estlin and R. Mooney. Hybrid Learning of Search Control for Partial-order planning. In *Proc. 3rd European Workshop on Planning*, 1995.

[13] O. Etzioni. Acquiring search control knowledge via static analysis *Artificial Intelligence*, Vol. 62, No. 2, 1993.

[14] O. Etzioni. A structural theory of explanation-based learning. *Artificial Intelligence*, Vol. 60, No. 1, 1993.

[15] O. Etzioni and R. Etzioni. Statistical methods for analyzing speedup learning experiments. *Machine Learning*, Vol 14, 1994.

[16] D. Frost and R. Dechter. Dead-end driven learning. In *Proc. AAAI-94*, 1994.

[17] L. Ihrig and S. Kambhampati. Integrating EBL with Replay to improve planning performance. In *Proc. 3rd European Planning Workshop*, 1995.

[18] M. Ginsberg and D. McAllester. GSAT and Dynamic Backtracking. In *Proc. KRR*, 1994.

[19] M. Goldszmidt, A. Darwiche, T. Chavez, D. Smith and J. White. Decision-theory for Crisis Management ROME Laboratory Technical Report, RL-TR-94-235. 1994.

[20] J .Gratch and G. DeJong COMPOSER: A Probabilistic Solution to the Utility problem in Speed-up Learning. In *Proc. AAAI 92*, pp:235--240, 1992

[21] D. Joslin and M. Pollack. Least-cost flaw repair: A plan refinement strategy for partial order planning. *Proceedings of AAAI-94*, 1994.

[22] S. Kambhampati and S. Kedar. Explanation-based generalization of partially ordered plans. In *Proc. AAAI-91*, pp. 679--685, July 1991.

[23] S. Kambhampati and J.A. Hendler. during Plan reuse. Controlling refitting during Plan reuse In *Proc. IJCAI-89*, 1989.

[24] S. Kambhampati and S. Kedar. A unified framework for explanation-based generalization of partially ordered and partially instantiated plans. *Artificial Intelligence*, Vol. 67, No. 1, pp. 29-70, 1994.

[25] S. Kambhampati, C. Knoblock and Q. Yang. Planning as Refinement Search: A Unified framework for evaluating design tradeoffs in partial order planning. ASU-CSE-TR 94-002. To appear in *Artificial Intelligence* special issue on Planning and Scheduling. 1995.

[26] S. Kambhampati. Admissible Pruning strategies based on plan minimality for plan-space planning. In *Proc. IJCAI-95*, 1995.

[27] S. Kambhampati and B. Srivastava. Universal Classical Planner: An Algorithm for unifying state-space and plan-space planning. In *Proc. 3rd European Workshop on Planning Systems*, 1995.

[28] S. Katukam and S. Kambhampati. Learning explanation based search control rules for partial order planning. In *Proc. AAAI-94*, 1994.

[29] S. Katukam. *Learning EBL Based Search Control Rules for Partial Order Planni ng* Masters Thesis, Arizona State University, June 1995.

[30] D. McAllester and D. Rosenblitt Systematic Nonliner Planning In *Proc. AAAI-91*, 1991.

[31] S. Minton. *Learning Effective Search Control Knowledge: An Explanation- Based Approach.* PhD thesis, Carnegie-Mellon University, Pittsburgh, PA, 1988.

[32] S. Minton, J.G Carbonell, Craig A. Knoblock, D.R. Kuokka, Oren Etzioni and Yolanda Gil. Explanation-Based Learning: A Problem Solving Perspective. *Artificial Intelligence*, 40:63--118, 1989.

[33] S. Minton. Quantitative Results Concerning the Utility of Explanation Based Learning *Artificial Intelligence*, 42:363--391, 1990.

[34] S. Minton, J. Bresina and M. Drummond. Total Order and Partial Order Planning: a comparative analysis. Journal of Artificial Intelligence Research 2 (1994) 227-262.

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

[36] N.J. Nilsson. *Principles of Artificial Intelligence.* Tioga, Palo Alto, 1980.

[37] E.P.D. Pednault. Synthesizing Plans that contain actions with Context-Dependent Effects. *Computational Intelligence*, Vol. 4, 356-372 (1988).

[38] E.P.D. Pednault. Generalizing nonlinear planning to handle complex goals and actions with context dependent effects. In *Proc. IJCAI-91*, 1991.

[39] J.S. Penberthy and D. Weld. UCPOP: A Sound, Complete, Partial Order Planner for ADL. In *Proc. KR-92*, November 1992.

[40] M.A. Peot and D.E. Smith. Threat-Removal Strategies for Nonlinear Planning. In *Proc. Eleventh AAAI*, 1993.

[41] A. Perez and O. Etzioni. DYNAMIC: A new role for training problems in EBL. In *Proc. of 9th Intl. Machine Learning Conference*, 1992.

[42] Y. Qu and S. Kambhampati. Learning control rules for expressive planners: Factors influencing performance. In *Proc. 3rd European Planning Workshop,* 1995.

[43] S. Kambhampati. Formalizing Dependency directed backtracking and explanation based learning in refinement search. ASU CSE Tech. Report 96-001, 1996.

[44] Y. Qu. Learning Search control Rules for Plan-space Planners: Factors Affecting the Performance Masters Thesis, Arizona State University, June 1995.

[45] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach* Prentice Hall, 1995.

[46] N. Sadeh, K. Sycara and Y. Xiong. Backtracking techniques for job shop scheduling constraint satisfaction problem. *Artificial Intelligence*, Vol 76, July 1995.

[47] D.E. Smith and M.A. Peot Postponing Threats in Partial-Order Planning. In *Proc. AAAI-93*, pp:500--506, 1993.

[48] E. Tsang. *Foundations of Constraint Satisfaction*, (Academic Press, San Diego, California, 1993).