TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Planning, as a sub-discipline of AI, has been around for close to twenty years. While the formal foundations of the field have grown increasingly sophisticated, progress has been much slower in terms of applications of AI planning techniques to realistic problems. A main reason for this has been a lack of adequate models of search-control for classical planners. To cope with the computational complexity of domain-independent planning, the planner should be provided with adequate search-control knowledge. A promising way of developing search-control knowledge is to let the planner utilize speedup learning techniques to *learn* such knowledge from its previous problem-solving episodes. One of the well known speedup learning techniques is *Explanation Based Learning* (EBL).

Although there has been a considerable amount of research towards applying speedup learning techniques to planning, almost all of it concentrated on the restrictive state-based models of planning, as opposed to more flexible and efficient, plan-space partial-order models of planning [8, 2]. One reason 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. [9]). This has lead to a somewhat ironic situation: while much of the work on generative planning is based on plan-space partial-order planners, the work on learning to improve planning performance continues to be based on state-space planners. Preferring a state-based planning strategy only to make learning easier seems to be somewhat unsatisfactory, especially given that plan-space planning strategies promise to avoid some of the inefficiencies of the state-based planners in plan-generation.

This thesis investigates adapting Explanation Based Learning (EBL) techniques to a plan-space planning framework. The general idea behind explanation based learning (see Figure 1.1) is as follows: given a problem the planner searches through the space of possible solutions and returns a solution. EBL analyzes failures and successes in the search tree explored by the planner and generates search control rules that guide the planner to avoid the failing paths and bias it toward the successful paths. When the rules are used in subsequent planning episodes, this could improve the performance of the planner.

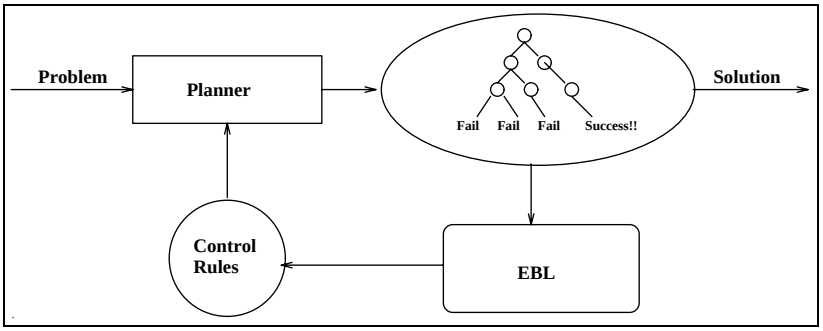


Figure 1.1: EBL in Planning

In this thesis, I will describe SNLP+EBL, a system that learns search control rules for SNLP, a causal-link partial order planner [7, 1]. In this system, when the planner, SNLP encounters a failure during problem solving, it invokes EBL to learn from failures. EBL first explains the reason for failure of the partial plan. Then, it regresses the explanation (the reason for failure) over the planner decisions to explain the reasons for failures at the ancestor levels of the partial plan. To do this, it combines all the explanations that are emerging from various failure refinements of a partial plan and propagates explanations up the search tree. During the propagation, SNLP+EBL also constructs search control rules that guide the planner to avoid the similar failures.

I will start by describing the basic learning framework of SNLP+EBL including the details of failures encountered by SNLP, and the initial explanations for these failure and

regression of failure explanations over the planner decisions, and propagation of these explanations up the search tree in detail. I will explain generalization of explanations to construct search control rules. I will then concentrate on SNLP+EBL's ability to learn from failures. The most obvious types of failures are those detected by the underlying planner, SNLP. I will show that these alone do not provide effective learning opportunities for SNLP+EBL in many domains. This is because often the planner's local consistency checks fail to detect the futility of a particular line of reasoning, possibly leading to infinite looping behavior. Strengthening the planner's consistency checks to detect these failures often results in poorer performance because of the increased cost of the consistency check. To deal with this impasse, I adopt a novel approach of strategically applying stronger consistency checks to the partial plans crossing depth limits, to detect and explain the implicit failures in those plans, and learn useful search control rules from these explanations.

I will describe one particular realization of this strategy that utilizes domain axioms which are readily available physical laws of the domain, (such as ''the same block cannot be on top of two different blocks''), to detect and explain inconsistencies (failures) at some depth-limit search failures. These explanations are then used as the basis for the EBL process of SNLP+EBL. Since the domain axioms are used to explain a failure, rather than detect the failure to begin with, they do not unduly increase the per-node cost of the search. Our experiments show that this method significantly improves SNLP+EBL's ability to learn useful search-control rules.

1.1 Overview

The rest of this thesis is organized as follows: the next chapter reviews the previous work that has been done in speedup learning to improve the performance of state space planners. Chapter 3 reviews the SNLP planning algorithm with an example. Chapter 4 describes the

EBL framework that is used in SNLP+EBL, classifies the failures encountered by SNLP during the planning process. It also describes how to construct explanations for the failures encountered by SNLP. Chapter 7 explains the generalization process to generalize the rules learned by SNLP+EBL. Chapter 5 describes regression rules that are used in SNLP+EBL to explain the failures at ancestor levels. Chapter 6 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. Chapter 8 describes how learning from analytical failures alone is not sufficient, and it describes a novel strategy for learning from depth-limit failures using domain axioms. Chapter 9 describes the experiments conducted to evaluate the effectiveness of search control rules learned by SNLP+EBL and it shows that SNLP+EBL outperforms SNLP in random blocks world problems. Chapter 10 presents the conclusions and future directions.

CHAPTER 2

BACKGROUND

Given a planning problem, a planner searches through the space of possible solutions and returns a solution, if one exists. While searching for a solution, planner also explores some of the paths which do not lead to a solution. In such cases, one way of improving the performance of the planner is to direct the planner to avoid the failures paths and bias it towards a solution. Knowledge based learning approaches such as inductive learning, explanation based learning (EBL) can be used to analyze the paths that are explored by the planner, and learn search control rules which direct the planner to avoid failure paths and guide it towards a solution.

Given the traces of planning episodes, EBL analyzes the traces and explains the reason for the failure or success of the trace. In looking at a problem solver's trace, there is no unique failure or success. For this reason, EBL is, in general, given a target concept along with the problem trace, knowledge about the domain to select what to explain from the trace. Figure 2.1 (adapted from [8]) shows a high-level schema specifying the input and output of EBL. As indicated by the schema, EBL begins with a high-level *target concept* and *training example* for that concept. Using the *domain theory*, a set of axioms describing the domain, EBL explains why the training example is an instance of the target concept. The explanation is essentially a proof that the training example satisfies the target concept. By finding the weakest conditions under which the explanation (proof) holds, EBL will produce a *learned description* that is both a generalization of the training example and a specialization of the target concept. The learned description must satisfy the *operationality criterion,* a test which insures that the description will be an efficient recognizer for the target concept.

Given:

- Target Concept: A concept to be learned.
- Training Example: An example of the target concept.
- Domain Theory: A set of rules and facts to be used in explaining why the training example is an instance of the target concept.
- Operationality Criterion: A predicate over descriptions, specifying the form in which the learned description must be expressed.

Determine:

- A description that is both a generalization of the training example and a specialization of the target concept, which satisfies the operationality criterion.

Figure 2.1: Specification of EBL (taken from Minton's thesis)

The operationality criterion is supposed to insure that each of the resulting *learned description*, which is converted into a rule, can be efficiently tested to guide the planner during the problem solving phase. However, the scheme above completely ignores the cumulative cost of testing the rule. Thus, although the rules may individually be less expensive to test than the original target concept definition, testing their description may be considerably more expensive.

To apply the rules learned by the system effectively, learning systems consider various costs such as application frequency, match cost and benefit of each of these rules to keep the learned rules in the search-control knowledge. Such systems have been shown to improve the performance of the base-level planner in many domains.

Given this general description of EBL, one can visualize a standard EBL methodology for learning search-control rules. This involves (i) identifying *target concepts* worth learning from, (ii) analyzing the search tree of the planner to locate and explain the instances of these target concepts and (iii) *regressing* the explanations through the successive decisions in the search tree to learn a variety of search control rules.

One such learning system, PRODIGY+EBL developed by Minton [8] learns search control rules and improved performance of the base planner, PRODIGY. In that, PRODIGY+EBL learns from variety of target concepts like failures, success, goal interaction. In similar to PRODIGY+EBL, another learning system that is developed by Bhatnagar[2] also learns search control rules for a state space planner, FailSafe. This system learns useful but potentially 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 pruned space. To learn quickly, the technique over-generalizes by assuming that the learned censors are preservable, i.e., remain unviolated along atleast one solution path. A recovery mechanism heuristically detects violations of this assumption and selectively specializes censors that violate the assumption.

PRODIGY+EBL, and FailSafe learning systems are based on state space planners. Since most of the recent research in planning concentrated on generative planning based on plan space planners, in this thesis I adapt the general framework of EBL for a plan space planner, SNLP, and show that it improves the performance of the base level planner.

CHAPTER 3

ARCHITECTURE OF THE SNLP+EBL SYSTEM

The SNLP+EBL system consists of two main components: the plan-space partial order planner, SNLP, and the learning component for doing EBL. This chapter describes SNLP's architecture, how SNLP interacts with the learner as problems are solved. In later chapters, I will describe the failures encountered by the planner and initial explanations for these failures, regression of explanations over the planner decisions, propagation of explanations up the search tree, rule construction and generalization, which are key parts of the learner.

The planner, SNLP, invokes the learner when it encounters a partial plan which cannot be refined further, and gives the learner an opportunity to learn from the failure. The learner generates an initial explanation from the failed partial plan, and regresses the explanation over the decision taken by the planner to get to the partial plan. It keeps the regressed explanation at the immediate ancestor of the failed partial plan, as a reason for the failure of this branch. When all refinements of a partial plan fail, EBL collects and propagates the explanation of a failure partial plan up the search tree. In the process of propagation, EBL also generalizes the explanation as described in later chapters. Generalized explanations are used to construct search control rules which are then used by the planner. Search control rules typically prune certain paths which are guaranteed to fail, thereby improving the performance of the planner.

3.1 The base level planner : SNLP

SNLP is a causal-link plan-space planner as, described in [7, 1]. SNLP starts with a null plan that consists of a set of initial state conditions and a set of flaws[1], where the set of flaws

[1] A flaw is a precondition of a step or an unsafe link in a partial plan

are initialized with the goal state conditions of the problem at hand. SNLP refines a partial plan by adding constraints to remove a flaw from the partial plan until it finds all flaws are removed or the partial plan has an inconsistency. If it finds an inconsistency in a partial plan, it backtracks in chronological fashion and refines other unexplored possible partial plans until it expands all possible partial plans or it finds a solution to the problem. Each partial plan during refinement in SNLP can be seen as 6-tuple: $\langle \mathcal{S}, O, \mathcal{B}, \mathcal{L}, \mathcal{E}, \mathcal{F} : \langle \mathcal{G}, \mathcal{T} \rangle \rangle$ where:

- $\mathcal{S}$ is the set of actions (step-names) in the plan; $\mathcal{S}$ contains two distinguished step names t_I and t_G[2].
- O is a partial ordering relation, representing the ordering constraints over the steps in $\mathcal{S}$.
- $\mathcal{B}$ is a set of codesignation (binding) and non-codesignation (prohibited bindings) constraints on the variables appearing in the preconditions and post-conditions of the operators.
- $\mathcal{L}$ 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. This constraint is interpreted as: ''s comes before w and gives p to w. No step in the plan that can possibly come in between s and w is allowed to necessarily add or delete p.''
- $\mathcal{E}$ is the set of effects of the plan, i.e., `has–effect`(e, s) such that $s \in \mathcal{S}$ and e is an effect (add or delete list literal) of s.
- $\mathcal{G}$ is the set of preconditions of steps of the partial plan, i.e, $precond(c,\ s)$ such that c is a precondition of step $s \in \mathcal{S}$ and there is no link supporting c at s in $\mathcal{L}$.

[2]To simplify, we removed symbol table $\mathcal{ST}$ from the table which maps step names to domain operators which represent initial step and goal step of a plan

- $\mathcal{T}$ is a set of threats, i.e., tuples of the form $\langle s \xrightarrow{p} w, t \rangle$ such that $s, t, w \in \mathcal{S}$, $s \xrightarrow{p} w \in \mathcal{L}$, t has an add or delete list literal q such that q necessarily codesignates with p, and t can possibly come in between s and w (t is called a *threat* for $s \xrightarrow{p} w$). The threat is resolved by either *promoting* t to come after w, or demoting it to come before s (in both cases, appropriately updating O). A threat for a causal link is said to be *unresolvable* if all of these possibilities make either O or $\mathcal{B}$ inconsistent. The Flaw list in a partial plan consists of the list of preconditions, $\mathcal{G}$, and the list of unsafe links, $\mathcal{T}$.

3.1.1 The problem solving process

SNLP starts its planning process with the null plan

$$\langle \{t_I, t_G\}, \{t_I \rightarrow \texttt{start}, t_G \rightarrow \texttt{fin}\}, \{\langle g_i, t_G \rangle\}, \rangle$$

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

The planning process consists of selecting a flaw from the flaws list and add constraints to the partial plan such that the flaw is removed. As explained earlier, there are two types of flaws exist in a partial plan. If the flaw is $precond(c, s)$, SNLP establishes it by using an effect q of an existing step (*simple establishment*) or newly introduced step s_e (*step addition*). In either case, the O and B fields of the partial plan are updated to make s_e precede s, and q codesignates with c. Finally, to remember this particular establishment commitment, a causal link of the form $s_e \xrightarrow{c} s$ is added to $\mathcal{L}$, where s_e, c, s are the source, the condition and the destination of the causal link, respectively.

SNLP does not backtrack over the selection of a flaw, but backtracks over the ways of resolving the flaw (e.g. it considers all possible establishment options for each precondition). After each establishment, the planner checks to see if there are any threats to

the new or existing causal links, and updates the threat list $\mathcal{T}$. If the selected flaw is a threat of the form $\langle s \xrightarrow{p} w, t \rangle$, then the planner resolves it by ordering the threat t to come after the step w (*promotion*) or by ordering the threat t to come before the step s (*demotion*). A partial plan is said to be complete if no flaws exist in the partial plan.

To summarize, the decisions taken by SNLP to resolve flaws are:

- If the flaw is a precondition of a step, possible decisions are:

  Simple Establishment: Select a *precondition of a step* from $\mathcal{G}$ and establish it by using an effect of an existing step in the partial plan.

  Step Addition: Select a *precondition of a step* from $\mathcal{G}$ and establish it by using an effect of a newly introduced step.

- if the flaw is an unsafe link, possible decisions are:

  Promotion: Select a threat to be resolved and order the threatening step to come after the destination of the causal link.

  Demotion: Select a threat to be resolved and order the step that is threatening the causal link to come before the source of the causal link.

All these decisions refine a partial plan to another partial plan. Thus, these decisions can be seen as STRIPS operators working on *plan states*. For example, $demotion$ decision can be seen in STRIPS representation as shown in the Figure 3.1.

Thus the $demotion$ decision requires a causal link $s_2 \xrightarrow{p'} s_3 \in \mathcal{L}$, and a step $s_1 \in \mathcal{S}$ such that it is not ordered with respect to s_2, and s_1 has an effect p'' that unifies with p'. The first two conditions of the $demotion$ indicates the threat flaw. The last two preconditions of the $demotion$ are to check demotion is possible or not. The effect of the $demotion$ decision is that the step s_1 is ordered to come before the source, i.e. s_2 of the causal link. We will exploit this STRIPS operator representation of planning decisions in later chapters.

Demotion(s_1, s_2)

Preconditions:

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

has–effect$(s_1, p'') \wedge \Box(p' \approx p'')$

$(s_1 \prec s_2) \notin O$

$(s_2 \prec s_1) \notin O$

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

Figure 3.1: Demotion decision in STRIPS representation

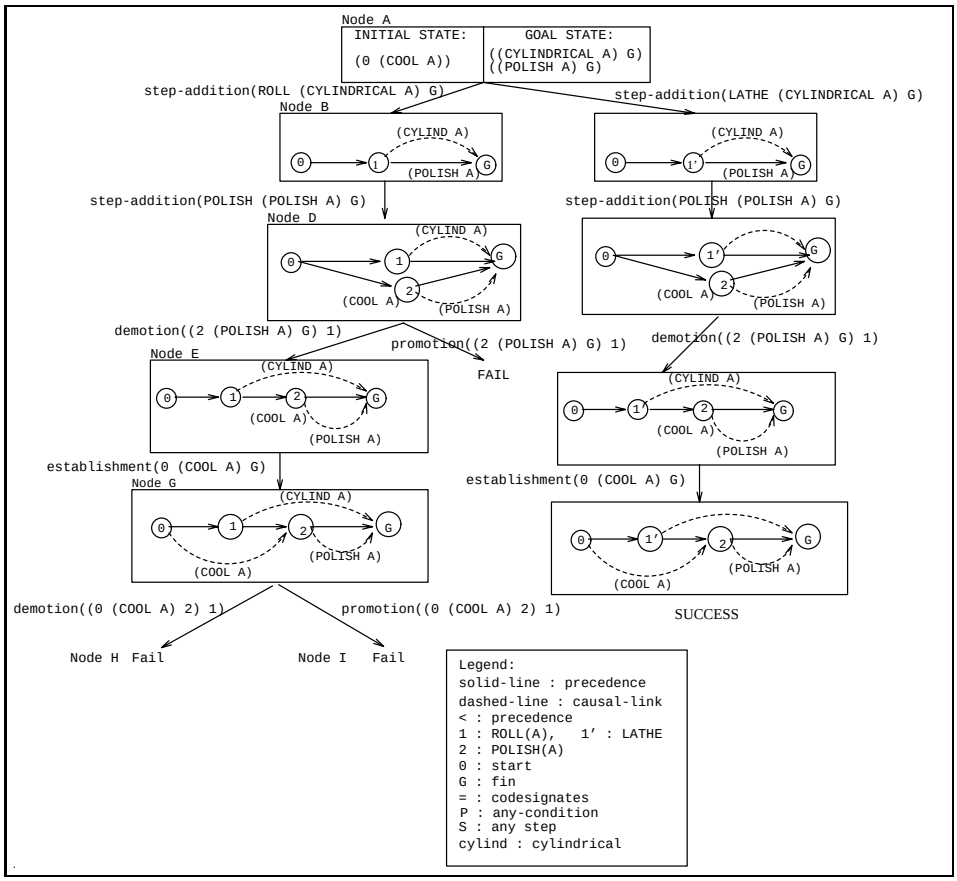


Figure 3.2: Search Tree illustrating SNLP planning process

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)	-

Figure 3.3: Blocks world domain

Example: Now, let us illustrate SNLP's planning algorithm on a simple example from a job-shop scheduling. We will be using this as a running example throughout the thesis. The shop consists of several machines, including a lathe and a roller that are used to reshape objects, and a polisher. Given a set of objects to be polished, shaped, etc., the task is to schedule the objects on the machines so as to meet these requirements. The operators in this domain are simplified to give a complete trace of the planning process for the above example.

The search tree in Figure 3.2 illustrates SNLP planning process in terms of an example from a simple job-shop scheduling domain with the operators shown below:

The initial planning problem is to polish an object A and make its surface cylindrical. The object's temperature is cool in the initial state. Figure 3.2 shows the complete search tree for the problem. SNLP, given the above problem, starts with $\langle t_I, t_G, \langle\, precond((CylindricalA), t_G)\, precond((PolishA), t_G)\rangle\rangle$ as the initial partial plan. SNLP picks up $precond(Cylindrical(A), \mathtt{G})$, from the set of flaws $\mathcal{G}$, and establishes the precondition with the help of the step 1: $\mathtt{Roll}(A)$. It then establishes the other precondition $precond(Polished(A), \mathtt{G})$ with the step 2: $\mathtt{Polish}(A)$. Since step 1 i.e. $\mathtt{Roll}(A)$, deletes $Polish(A)$, it is now a threat to the link 2 $\stackrel{Polish(A)}{\rightarrow}$ G. SNLP resolves this threat by demoting step 1: $\mathtt{Roll}(A)$ to come before 2: $\mathtt{Polish}(A)$. The step $\mathtt{Polish}(A)$ also introduces a new precondition $precond(Cool(A), 2)$. SNLP establishes it using the effects of the start state. Since $\mathtt{Roll}(A)$ also deletes $Cool(A)$, it threatens the last establishment. When SNLP tries to deal with this threat by demoting 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 $precond((Cool(A), G)$ using `Lathe`(A) and then achieves $Polished(A)$ using the operator `Polish`(A). It succeeds in this path and returns a solution.

CHAPTER 4

EXPLANATION BASED LEARNING

As observed in earlier chapter, EBL analyzes failures and successes in the search tree generated by the planner and generates search control rules that guide the planner to avoid similar failures and bias it towards a success.

Search control rules aim to provide guidance to the underlying problem solver at critical decision points. As we have seen above, for SNLP these decision points are the selection of flaws, establishment, 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 never backtracks over these decisions.[1] SNLP+EBL system learns search control rules for the other decisions. A search control rule may either be in the form of a selection rule or a rejection rule. In our current work, we have concentrated on learning rejection rules (although the basic framework can be extended to include selection rules).

Unlike systems such as PRODIGY/EBL, which commence learning only after the planning is completed, SNLP+EBL does adaptive (intra-trial) learning (c.f. [2]), which combines a form of dependency directed backtracking with the generation of search-control rules. The planner does depth first search both in the learning and non-learning phases. During the learning phase, SNLP+EBL invokes the learning component whenever the planner encounters a failure. Figure 4.1 shows a schematic flow diagram of the EBL process.

SNLP+EBL starts by generating an explanation of a failure when it is encountered. It

[1]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.

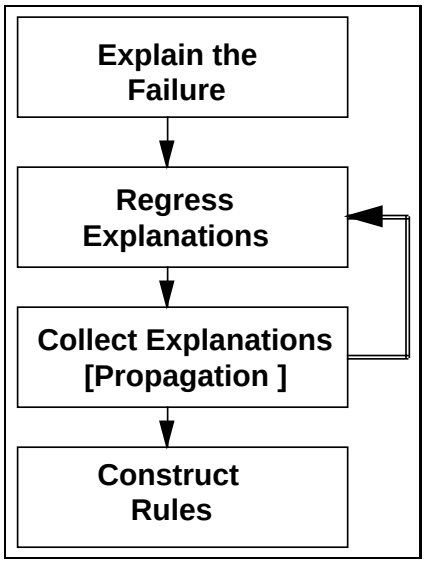


Figure 4.1: EBL Framework

then regresses the explanation over the planner decisions to explain failure at higher levels. After regression, it conjoins all the explanations coming from all the lower levels of a node and propagates it up the search tree. During propagation of the explanation up the search tree, it constructs search control rules which are in turn used by the planner to avoid similar failures.

In the following chapters, we will discuss, in detail, each of these phases of EBL framework.

4.1 Failures and Initial Explanation Construction

SNLP encounters two kinds of failures during its planning process. These are:

- Analytical Failures

  These failures are due to ordering or binding inconsistencies in the partial plan or due to unachievable open conditions.

- Depth Limit Failures

As said above, a pre-set depth limit has been set during the planning process and whenever the planner crosses the depth limit, it is considered as a failure partial plan.

Whenever the planner encounters a failure, EBL constructs an explanation for the failure and regresses the explanation over the planner decisions that led to the failure. An explanation for a failure of a partial plan is a minimal set of constraints (steps, orderings, bindings, effects, preconditions and causal links) that are together inconsistent.

4.1.1 Analytical failures

These are the failures SNLP can detect during its planning process. When SNLP encounters an inconsistency in the partial plan, it declares the partial plan as a failure and backtracks from that point to explore any unexplored paths. SNLP can detect three kinds of inconsistencies in the plan and these are :

- Ordering Inconsistencies

  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 $(s_1 \prec s_2) \wedge (s_2 \prec s_1)$, then this an inconsistency in the partial plan is detected. Explanation for the ordering inconsistency is:
  $(s_1 \prec s_2) \wedge (s_2 \prec s_1)$

- Binding inconsistencies

  These arise when there is an inconsistency in the bindings. For example, if there exists a variable X in the partial plan such that $(X \approx A) \wedge (X \not\approx A)$, then this is an inconsistency in the partial plan.
  An explanation for this binding inconsistency is:
  $(X \approx A) \wedge (X \not\approx A)$

- Failure to establish from initial state

  When SNLP+EBL selects a precondition flaw to remove, it always attempts to establish it from initial state irrespective of initial state conditions. For example, a flaw, $\texttt{precond}(has\_airport(Tempe), s)$ is selected to be achieved by establishment. SNLP+EBL first establishes a causal link from initial state of the form $(s_0 \overset{has\_airport(Tempe)}{\rightarrow} s)$ irrespective of initial state conditions. After establishing the causal link, it checks whether the condition $has\_airport(Tempe)$ is present in the initial state or not. If it is not present in the initial state, there exists an inconsistency, because the causal link states that the initial state gives the condition and the initial state conditions state that the condition is not given by it. SNLP declares it as a failure and an explanation for this inconsistency can be constructed as:
  $(s_0 \overset{has\_airport(Tempe)}{\rightarrow} s)$
  $\neg\texttt{initially\_true}(has\_airport(Tempe))$
  The above explanation states that there exists a link $s_0 \overset{has\_airport(Tempe)}{\rightarrow} s$ and the condition $has\_airport(Tempe)$ is not present in the initial state of the problem. The reason for SNLP+EBL to first establish a causal link and then find an inconsistency in the partial plan is to explore all branches that are possible irrespective of the initial conditions. This makes the learner consider all the failures that are encountered by SNLP uniformly irrespective of initial state conditions.

4.1.2 Depth-Limit Failures

When SNLP crosses the preset depth limit, it declares the partial plan at the depth limit as a failure partial plan and backtracks from that point. The partial plan at the depth limit may contain inconsistencies which are not recognized by SNLP's consistency checks. By applying stronger consistency checks on constraints of the partial plan at depth limit,

we may be able to recognize such inconsistencies in the partial plan. In chapter 8, we will explain an instance of this strategy which uses domain-axiom consistency checks to explain the implicit failures at depth limits, and construct explanations for these failure. Once an explanation is constructed, EBL regresses it over the planner decisions that led to the failure. In next chapter, we will discuss the regression process in detail.

CHAPTER 5

REGRESSION

Consider the case where SNLP found an inconsistency in a partial plan P at a node n (see Figure 5.1). Further suppose that the parent node of n, node n', contains the decision d resulting in failure and E is the failure explanation. We would like to know what constraints of the partial plan, P' at node n', are necessarily responsible for causing the failure at node n after taking the decision d. The process of computing the constraints at node n' that caused the failure after taking the decision d is called the *regression*.

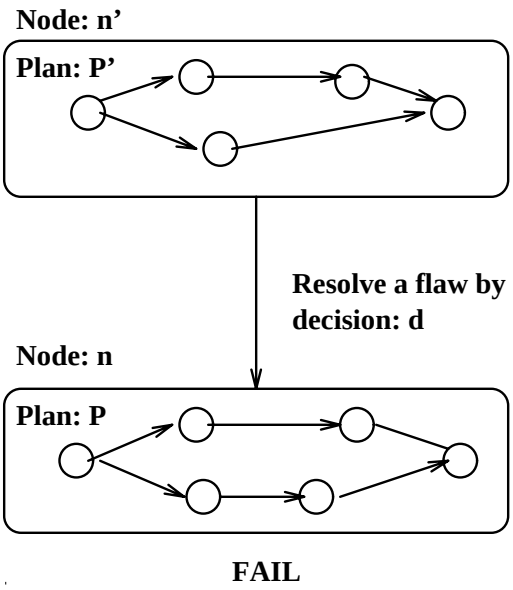


Figure 5.1: A part of a failure branch to explain the regression process

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 the decision d adds c to the partial plan. In state based planners, the decisions correspond closely to applying operators to world states, and thus regression of an explanation over a decision is very close to regression over operators. In contrast, the decisions in partial order planners convert a

partial plan to another partial plan. We thus need to provide a set of rules for regressing arbitrary constraints of an explanation over the planner decisions.

In discussing regression rules, it is useful to distinguish between two types of constraints: **transitive** and **non transitive** constraints. A constraint is said to be transitive if the presence of two constraints $c_1 \wedge c_2$, together imply a third constraint c_3, which is not explicitly stated in the partial plan. In other words, $c_1 \wedge c_2 \vdash c_3$. For example, ordering and binding constraints in a partial plan are transitive constraints while causal links, are non transitive constraints. For example, in Figure 5.2, 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. In contrast, adding a causal link or a precondition of a step does not create any further causal links or preconditions.

Regression of non-transitive constraints are easy to handle [11]. Suppose we want to regress a constraint c over a decision d. If c is added by d, then the regression of c over d is $True$. Otherwise the regression of c results in itself. Thus we have

Regress(c, d)

= $True$, if $c \in add(d)$ (clause (i))

= c, otherwise (clause (ii))

For a transitive constraint c, regression 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 we need an additional rule:

Regress(c, d)

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

It is also 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

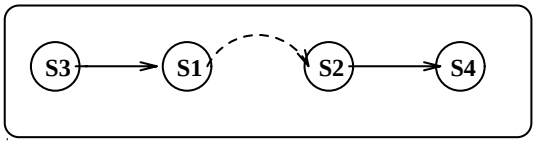


Figure 5.2: An example showing transitive constraints

c over a decision d results in disjunction of all such sets of constraints c'. For the example in Figure 5.2, assume that s_3 precedes s_4 before ordering. Then the regression of $(s_3 \prec s_4)$ over the ordering decision $(s_1 \prec s_2)$ results in $(s_3 \prec s_4) \vee [(s_3 \prec s_1) \wedge (s_2 \prec s_4)]$.

Regression of an explanation over a decision d is the conjunction of regressing each constraint of the explanation over the decision d. 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) $\wedge$ …$\wedge$ Regress(c_i, d)

The following table explains the regression of various constraints over the demotion decision.

(iii) Demotion($s_1,\ s_2 \xrightarrow{p} s_3$)

`Preconditions`: $(s_1 \prec s_2) \notin O$

$(s_2 \prec s_1) \notin O$

$s_2 \xrightarrow{p} s_3 \in \mathcal{L}$

`has–effect`(s_1, p)

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

Constraint	**Result**	**Reason**
$precond(p', s')$	$precond(p', s')$	clause(ii)
$s' \xrightarrow{p'} s''$	$s' \xrightarrow{p'} s''$	clause(ii)
`has–effect`(s', p')	`has–effect`(s', p')	clause(ii)
$unifies(p', p'')$	$unifies(p', p'')$	clause(ii)
$(s' \prec s'')$	$True$ $(s' \prec s'')\vee$ $[(s' \prec s_1) \wedge (s_2 \prec s'')\ ]$	if $(s' \approx s_1) \wedge (s'' \approx s_2)$ clause(i) clause (ii) clause (iii)

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 results in

themselves (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 5.2, 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, if we consider regression of constraints over a step addition, $step\text{-}add(s_1, precond(p', s_2))$ which adds a step s_1 into a partial plan to achieve $precond(p', s_2)$, the table of regression results are shown below. Like regression of any constraint over any decision, all the constraints that are not added by the step addition are regressed to themselves and the constraints that are added by the step addition are regressed to $True$.

(i) StepAddition($s_1, precond(p', s_2)$)

`Preconditions`: $precond(p', s_2) \in \mathcal{G}$

$s' \xrightarrow{p'} s_2 \notin \mathcal{L}$

`has–effect`(s_1, p'')

$unifies(p', p'')$

`Effects`: $\mathcal{S} \leftarrow s_1$

$\mathcal{L}' \leftarrow s_1 \xrightarrow{p'} s_2$

$O' \leftarrow (0 \prec s_1) \wedge (s_1 \prec s_2)$

$\mathcal{B}' \leftarrow unify(p', p'')$

$\mathcal{G}' \leftarrow preconditionsof s_1 - p'$

$\mathcal{F}' \leftarrow effects(s_1)$

Constraint	Result	Reason
$precond(q', s')$	$True$ $precond(q', s')$	if $s' \approx s_1$, clause (i) otherwise, clause(ii)
has–effect(s', q')	$True$ has–effect(s', q')	if $s' \approx s_1$, clause (i) otherwise, clause(ii)
$s' \xrightarrow{q} s''$	$True$ $s' \xrightarrow{q} s''$	if $s' \approx s_1$, clause(ii) otherwise, clause(ii)
$(s' \prec s'')$	$True$ $(s_2 \prec s'')$ $(s' \prec s'')$	if $s' \approx s_1 \wedge s'' \approx s_2$, clause(ii) if $s' \approx s_1 \wedge s'' \not\approx s_2$, clause(iii) otherwise, clause(ii)
$(s_0 \prec s')$	$True$ $(s_0 \prec s')$	if $s_0 \approx s_1$ otherwise, clause(ii)

Regression rules over simple establishment are very similar to regression rules over step addition and regression rules over promotion decision are same as regression rules over the demotion decision.

One final observation regarding the use of regression in EBL is that its use in regression differs from the complete goal regression. As noted earlier, regression of E over a decision d sometimes results in a disjunction of $E' \vee E'' \vee ...$ 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 explanation which is present in the parent partial plan. In Figure 5.2, the result of regression of $(s_3 \prec s_4)$ over the decision to add $(s_1 \prec s_2)$ is $(s_3 \prec s_4) \vee [(s_3 \prec s_1) \wedge (s_2 \prec s_4)]$. SNLP+EBL considers $(s_3 \prec s_4)$ as the result of regression because $(s_3 \prec s_4)$ is present in the partial plan.

CHAPTER 6

PROPAGATION OF FAILURE EXPLANATIONS

In earlier chapters, we looked at failures encountered by SNLP and initial explanations for these failures. We also looked at regression of explanations over the planner decisions. Next we describe is how the regressed explanations are combined and propagated up the search tree.

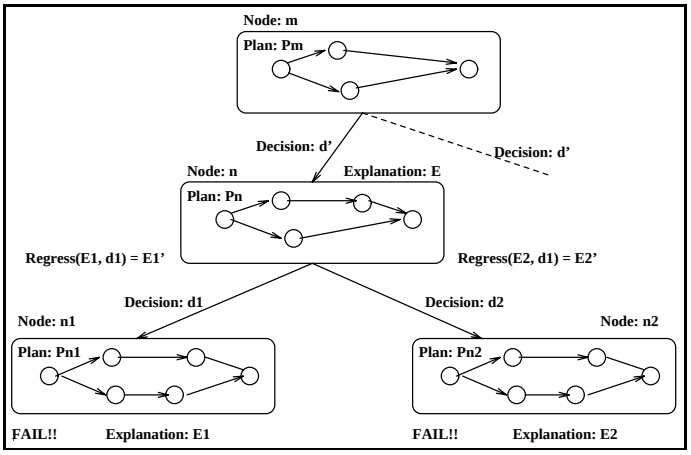


Figure 6.1: An example for propagation

Consider the example shown in Figure 6.1. Since both the children nodes $n1$ and $n2$ of a node n failed, we would like the planner to avoid the decision d to generate node n. 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 constraints at node m we regress the explanation of failure at node n over the decision, as explained earlier. To do this, we first need to compute the explanation of failure at node n. Suppose that $n1$ and $n2$ are failure nodes with partial plans P_{n1} and P_{n2} and failure explanations are $E1$ and $E2$ respectively. Suppose further that these nodes are generated from node n to remove a flaw, say F, by taking decisions $d1$ and $d2$ respectively. Assume that these are the only two

different ways of removing the flaw[1]. As explained earlier, $E1$ and $E2$ are regressed over $d1$ and $d2$ to give rise to $E1'$ and $E2'$. Since both of these branches failed to remove the flaw F, and these are the only two choices, the partial plan P at node n cannot be refined to a successful partial plan. To compute the 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:

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

In other words, the general propagation rule to compute an explanation of a failure at node n which has failing children $n_1..n_p$ corresponding to decisions $d(n_1), .., d(n_p)$ is

$$E(n) = Constraints\ describing\ the\ Flaw \wedge$$
$$Regress(E(n_1), d(n_1)) \wedge Regress(E(n_2), d(n_2)) \wedge \ldots \wedge Regress(E(n_p), d(n_p))$$
$$E(n) = Constraints\ describing\ the\ Flaw \wedge$$
$$\bigwedge \forall n_i \wedge Regress(E(n_i), d(n_i)) \rightarrow (i)$$

In the above rule (i), $n_1, n_2, .. n_p$ are *all possible* children nodes of node n and $d(n_i)$ denotes the decision that is taken to get to the node n_i from its parent node n and $E(n_i)$ denotes the explanation at node n_i. The resultant explanation E at node n can now be regressed over d 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 6.2, which shows the lower part of the failure branch of the example that is described in chapter 3.1.1. When SNLP failed at node H and I in the Figure 6.2, EBL explains these failures in terms of ordering inconsistencies as shown in the figure. When we regress the explanation of

[1]refer to refinement search

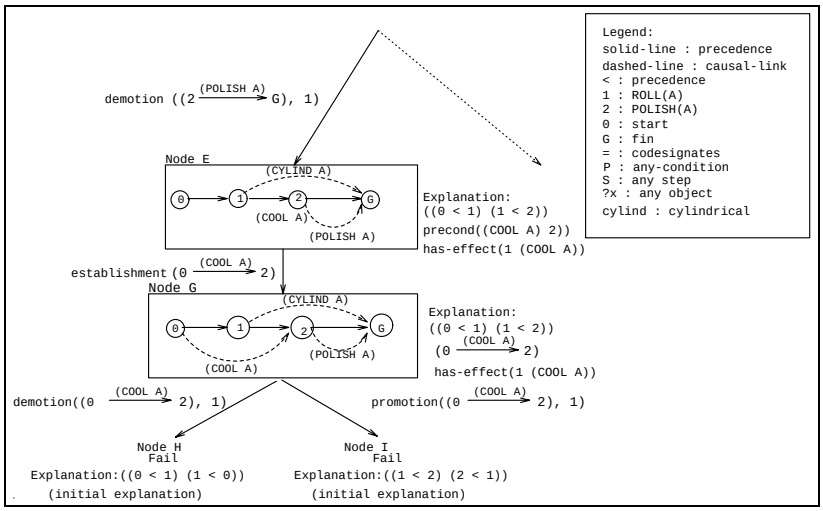


Figure 6.2: Failure branch of the the example described in Figure 3.2.

node H over the $demotion((0 \overset{(Cool\ A)}{\rightarrow} 2), 1)$, it results in the ordering constraint $(0 \prec 1)$. Similarly when we regress the explanation of node I over the $promotion((0 \overset{(Cool\ A)}{\rightarrow} 2), 1)$, it results in the ordering constraint $(2 \prec 1)$. Now, at node G, we have two explanations for the failure of the branches H and I. According to (i), the explanation at node G is:

$$
\begin{aligned}
&E(G) \\
&\quad = Constraints\ describing\ the\ Unsafe\ link\ flaw\ \wedge (0 \prec 1) \wedge (1 \prec 2) \\
&\quad = (0 \overset{(Cool\ A)}{\rightarrow} 2) \wedge \texttt{has\_effect}(1,\ (Cool\ A)) \wedge (0 \prec 1) \wedge (1 \prec 2)
\end{aligned}
$$

The above resultant explanation at node G is also shown[2] in the Figure 6.2. The explanation at node G can be interpreted as, if there are three steps s_1, s_2 and s_3 such that $(s_1 \prec s_2) \wedge (s_2 \prec s_3)$ and if an unsafe link of the form $((s_0 \overset{(CoolA)}{\rightarrow} s_2), s_1)$, exists in a partial plan, prune the node from search space, because as long as the unsafe link flaw exist in the partial plan, the planner will take $demotion$ and $promotion$ which will lead to failure.

[2]In the Figure 6.2, the unsafe link flaw is represented as $(0 \overset{(Cool\ A)}{\rightarrow} 2) \wedge \texttt{has\_effect}(1, (Cool\ A))$.

6.1 Avoiding overspecific explanations in propagation

The propagation process as described above may sometimes give overspecific explanations. To see this, consider the example described in Figure 6.3.

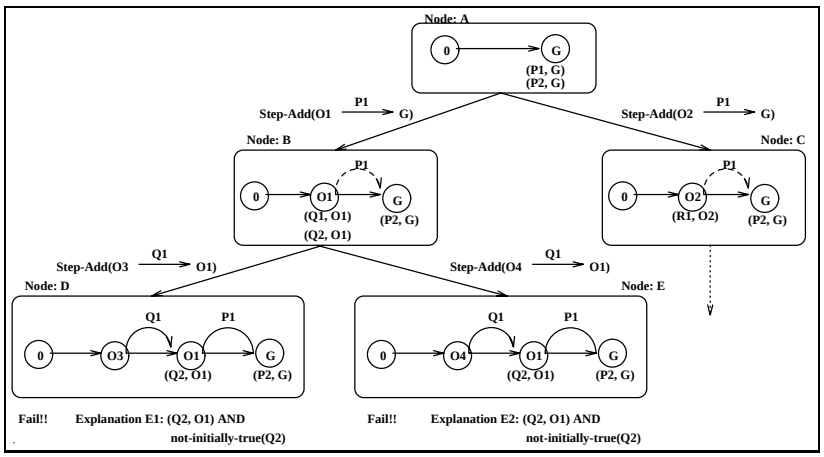


Figure 6.3: An example for dependency directed backtracking

In Figure 6.3, both children of the node B failed to achieve a precondition flaw $precond(Q2, O1)$, since $Q2$ is not given by either the domain operators or by the initial state. Based on previous discussion, EBL constructs initial explanations for these failures as shown in the Figure 6.3. According to the propagation rule described earlier, the explanation at node B will be:

$$E(B) = precond(Q1, O1) \wedge \text{Regress(E1, Step-Add}(O3 \xrightarrow{Q1} P1))$$
$$\wedge\ \text{Regress(E2, Step-Add}(O4 \xrightarrow{Q1} P1))$$
$$= precond(Q1,\ O1) \wedge precond(Q2,\ O1) \wedge \neg \texttt{initially\_true}(Q2)$$

The explanation above at node B has an additional flaw $precond(Q1, O1)$, which is certainly redundant, since it is clear that the node B will fail as long as the precondition $precond(Q2,\ O1)$ exists in the flaw list and $Q2$ is not given by the initial state[3]. Clearly in

[3]The assumption is that the domain operators do not change from problem to problem. Search control rules generated by EBL are sound, only if the domain description does not change.

this case the explanation of failure at node B is same as the explanation at child node D. To handle this, we change the propagation such that when regression does not change an 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,

$$E(n) =$$
$$Regress(E1, d1), If \;\; Regress(E1, d1) = E1 \rightarrow (i)$$
$$Constraints\ describing\ the\ flaw \wedge$$
$$Reg(E1, d1) \wedge Reg(E2, d2), Otherwise \rightarrow (ii)$$

In other words, the general propagation rules to compute an explanation at node n is:

$$E(Parent(n)) =$$
$$Regress(E(n_i), d(n_i)), If \;\; Regress(E(n_i), d(n_i)) = E(n_i) \rightarrow (i)$$
$$Constraints\ describing\ the\ flaw \wedge$$
$$\forall n_i \wedge Regress(E(n_i), d(n_i)), Otherwise \rightarrow (ii)$$

In rule (i), 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 $E(n_i)$ over the decision $d(n_i)$. This is because $E(n_i)$ is an inconsistent constraints set (i.e. $P(n_i)$ has no potential solutions). When the regression of $E(n_i)$ is not changed when it is regressed over the $d(n_i)$ implies that $E(n_i)$ is present in the partial plan at node n. This means that the partial plan at node n, $P(n)$, has a set of inconsistent constraints, same as $E(n_i)$. Consequently, $P(n)$ also has no potential solutions. Thus, no refinement of the plan $P(n)$ will lead to a solution [4].

[4]Decisions in a refinement planning add constraints to a plan and do not remove any constraints from it. Note that flaws are not part of constraints of a plan.

6.1.1 Dependency directed backtracking

The preceding discussions suggests a methodology for exploiting the explanation and regression procedures to do dependency directed backtracking. In particular, suppose we are folding the propagation into the search process. 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 the other siblings of node n. Thus, an explanation $E(n)$ can be taken all the way up without expanding the outstanding siblings until $E(n)$ changes after regression over a decision.

In the example described in Figure 6.3, since the explanation of failure at node D did not change after regression over *step-add*, 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 Explanation same as the explanation of failure node D.

The actual implementation of SNLP+EBL folds the propagation into the search process to provide a default dependency directed backtracking. Figure 6.4 shows the full description of the propagation algorithm.

6.2 Rule Construction

In SNLP+EBL, rules are learned during propagation of explanations up the search tree. Since SNLP+EBL explains only failures in the current implementation, only rejection rules are learned[5]. Given a failure explanation E for a node N, a rule can be learned to reject node N as follows:

IF E

REJECT $NODE$

[5]The SNLP+EBL framework can be easily extended to explain successes and learn preference rules

Procedure Propagate($E(n_i), d(n_i)$)
($d(n_i)$: decision taken get to node n_i from its parent node;
$P(n_i)$: partial plan at node n_i; $E(n_i)$: explanation of failure at n_i).
0. Set $d \leftarrow d(n_i)$
1. $E' \leftarrow$ `Regress`($(E(n_i), d)$
2. If $E' = E$, then set $d \leftarrow d(Parent(n_i))$; Goto Step 1. (*a form of DDB*)
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 antecedent generalize it and store it in the rule set
      3.1.2. $E(Parent(n_i)) \leftarrow E(Parent(n_i)) \wedge E'$
      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 $E(Parent(n_i)) \leftarrow E(Parent(n_i)) \wedge E' \wedge$
      Constraints that describe the flaw that the decision $d(n_i)$ is removing
      3.2.3. Set $n_i \leftarrow Parent(n_i)$, $E(n_i) = E(Parent(n_i))$
      Set $d(n_i) \leftarrow d(Parent(n_i))$; Goto Step 1.

Figure 6.4: Propagating Failure Explanations

The rule above states that if an explanation E holds at a node, then that node can be pruned from the search tree. We can also construct another rules that reject decisions from consideration. Suppose the failure explanation E is regressed over a decision d and the resultant explanation is E'. Then we can learn a rule as follows:

IF E'

REJECT d

The rule above states that if E' holds at a node and if d is a decision choice then the planner can reject the decision d from the choices.

Now, let us look at the example explained in Figure 3.1.1. and see how rules are learned. In this example after constructing an initial explanation for node H, a rule can be learned to reject a node, as shown below:

IF $(s_0 \prec s_1) \wedge (s_1 \prec s_0) \in \mathcal{O}$

REJECT NODE

This rule states that if there exists an ordering cycle in a partial plan of a node, then reject the node. Since the planner is going to check the inconsistency in the plan, learning this particular rule will not be very useful in improving the performance. Specifically, the match cost is not offset by the savings.

At this point, the planner regresses the explanation over the demotion decision to explain the failure of branch H. After regression, a rule can be generated as

IF $(s_0 \prec s_1)$
REJECT $demotion(s_0 \overset{(Cool\ A)}{\rightarrow} s_2, s_1)$

The rule states that if $(s_0 \prec s_1) \in \mathcal{O}$, then do not take the demotion decision. Similarly, SNLP+EBL could learn a rule at node B to reject a node in search tree if the explanation at node B holds with the node. In other words:

IF $(s_0 \prec s_1) \wedge (s_1 \prec G) \wedge$
$precond((Polish\ A), G) \wedge$
has–effect(s_1, $(Cool\ A)$) $\wedge$
has–effect(s_1, $(Polish\ A)$) $\wedge$
$\neg$initially–true($Polish\ A$)
REJECT NODE

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

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). This rule is shown to the left of node A. It says that $Roll$ should be rejected as a choice

for establishing any condition at goal step G, if ($Polish\ A$) is also a goal at the same step. Notice that the rule does not mention the specific establishment ($Cylindrical\ A$), that lead to the introduction of $Roll$. This is correct because the failure explanation at node B does not involve ($Cylindrical\ A$).[6]

The rules above are in terms of object A and these are not applicable, if we are solving the similar problem with another object B. This limitation on rule applicability can be overcome and in the next chapter we will look at how to generalize these rules such that these rules are applicable in more situations.

[6]It is interesting to note that in a similar situation, Prodigy [9] seems to learn a more specific rule which depends on establishing ($Cylindrical\ A$).

CHAPTER 7

GENERALIZATION

Until now, we have seen how SNLP+EBL learns search control rules from the failures encountered by SNLP. Now, let us look at the usage of these search control rules and their role in improving the performance.

Consider the following rule that is generated by SNLP+EBL, which is shown in the Figure 7.1.

This rule states that if the object A needs to be $Polished$ and it is not $Polished$ initially, then the planner should not add $Roll$ to achieve ($Cylindrical\ A$) at step $Goal$. 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 performance.

Now, assume that the planner is given a new problem which involves making an object B, $Cylindrical$ and $Polished$. This new problem has 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 above rule because the above rule states that 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

IF $precond((Polish\ A), Goal) \in \mathcal{G}\ \wedge$
$\neg$`initially–true`$(Polish\ A)$
REJECT $stepadd((Roll\ A) \overset{(Cylindrical\ A)}{\rightarrow} Goal)$

Figure 7.1: Rejection rule

plan). However, it is clear that the above 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 $Goal$ from the rule and replace with variables[1], while preserving the soundness of the rule.

For a moment, let us discuss about the soundness of a rule. Whenever the constraints of a rule are applicable in a partial plan, the planner SNLP, takes the advice from the rule and rejects generating a branch by adding the decision of the rule to the partial plan. In general, a rule is said to be sound if it does not effect the completeness of the underlying planner. Soundness can be defined in multiple ways:

- Strong Soundness: there are no solutions under the branch that is rejected by the planner.
- Minimal Soundness: there are no minimal solutions existing under the branch that is rejected by the planner (where a solution is minimal if no operator sequence produced by removing steps from it is a solution).
- Solution Soundness: there could be solutions under the branch that is rejected by the planner, but there exists atleast one other solution in the over all search space.

All the above criterion for soundness of a rule preserve the completeness of the underlying planner (that is, if there exists a solution for a problem, the planner is guaranteed to find it). For example, if the planner takes the advice of the rule in Figure 7.1 and rejects adding the operator $Roll$ to achieve $Cylindrical$, then the rule guarantees that there exists atleast one solution in other branches of the search tree. But the rules learned by SNLP+EBL are sound according to criterion (i), which guarantee that there are no solutions

[1] 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.

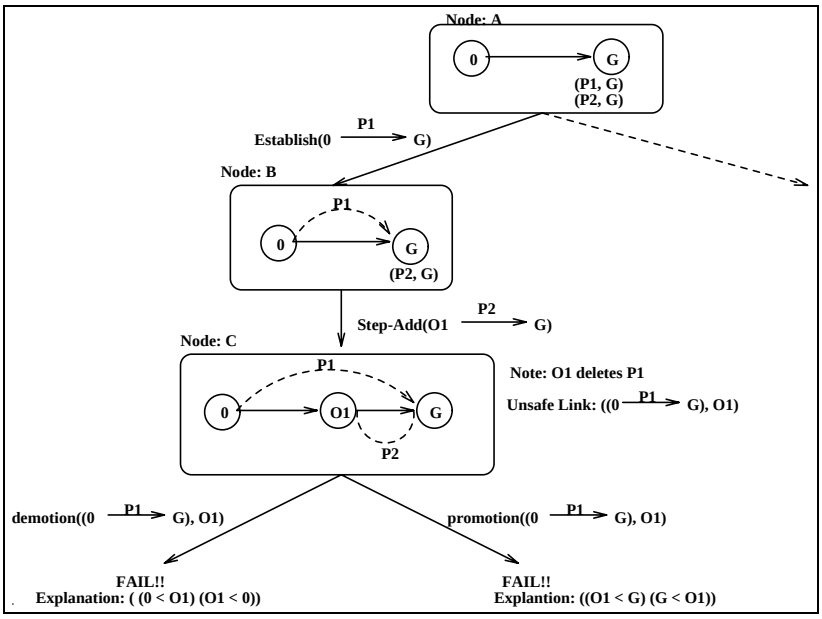


Figure 7.2: An example to explain generalization process in SNLP+EBL

under the rejected branch. In this chapter, we will describe how to generalize the rules that are learned by SNLP+EBL such that the rules are applicable in more than one problem while preserving the soundness of the rule.

We start by noting that generalization involves replacing specific step-names such as $Goal$ and object-names such as A with variables, without losing the soundness of a rule. For a generalized rule to be sound (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. Ideally, we would like to remove all the step-names and object-names from a rule and replace these names with variables. But this may not preserve the soundness of the rule, since some names may have to be present in the rule for the failure to occur. We will show later a situation where replacing all the names of a rule with variables will lead to the loss of the soundness of the rule.

Traditionally, the generalization in EBL is done by starting with a variablized explanation and variablized decisions and redoing the complete regression process, specializing the

explanation along the way [8]. In SNLP+EBL, we use single pass generalization scheme, where we begin by variablizing the initial explanation that is constructed by SNLP+EBL. Consider the example described in Figure 7.2, where the goals required are $P1$ and $P2$ and initially only $P1$ is true. The planner removes the flaw $precond(P1,\ G)$ and establishes $(0 \stackrel{P1}{\rightarrow} G)$ from the initial state (denoted by 0, in the figure). It then adds the new operator[2] $O1$ to remove the flaw $precond(P2,\ G)$. But the operator $O1$ deletes $P1$, which threatens the establishment from initial state. Planner tries to order $O1$ to come before the initial state by demotion and orders $O1$ to come after G by promotion. Both these branches fail because of ordering inconsistencies. The initial explanation constructed by the planner for demotion failure is

$$E = (0 \prec s_1 : O1) \wedge (s_1 \prec 0)$$

Immediately, after constructing the above initial explanation, SNLP+EBL variablizes by replacing names with variables where 0 in the explanation is replaced with $?s_0$, $O1$ is replaced with $?s_1$ and to remember the bindings, these are kept in a list of bindings.

$$\mathcal{E} = (?s_0 \prec ?s_1) \wedge (?s_1 \prec ?s_0) \ \wedge$$
$$\langle (?s_0 \rightarrow 0), (?s_1 \rightarrow O1) \rangle$$

After variablizing the initial explanation, and keeping the bindings in a separate list, it regresses the explanation over variablized decisions (for example, $demotion(?s_0 \stackrel{P1}{\rightarrow} ?s_g, ?s_1)$ with $\langle (?s_0 \rightarrow 0), (?s_g \rightarrow G), (?s_1 \rightarrow O1) \rangle$ corresponds to the variablized version of $demotion(0 \stackrel{P1}{\rightarrow} G, O1)$) [3]. As explained earlier, SNLP+EBL conjoins explanations from various branches of a node and propagates the explanation up the search tree. At any point, SNLP+EBL generates a rule as explained earlier, *but removes the bindings between names*

[2] steps are instances of operators

[3] Note that the predicates of a causal link are not variablized. Since the predicates will be again introduced into explanations during regression, variablizing the predicates will not help much in generalization

and variable, thus leaving the variables in the rule. Since an object variable could be matched with any object of the domain, the rule can be applied in more than one problem, if the other constraints of the rule are met by the partial plan.

If a failure occurrs because of a specific object or a step in the partial plan, then removing all the bindings between names and variables will lead to unsound rules. If a specific object name or a step is required, then the regression of a binding over a planner decision will result in $True$ and replacing all the instances of the variable with the name in the explanation.

To automate this process, we just need to formalize the regression of bindings between names and variables over the planner decisions. Consider a step-name binding $\langle(?s_0 \rightarrow 0)\rangle$ over various decisions. As noted earlier, regression of a constraint c over a decision d regresses to itself, if d did not add the constraint c. During planning, whenever the planner, SNLP, introduces a new step s_1 into a partial plan, s_1 is replaced with $?s_1$ and this replacement is noted down by adding a step-binding $\langle ?s_1 \rightarrow s_1\rangle$ into the partial plan. Step addition is the only one decision that introduces the bindings between step names and variables. Therefore, from earlier discussion of regression in Chapter 5

Regress($\langle ?s_1 \rightarrow s_1\rangle$, $stepadd(s_1, precond(p', s_2))$)

= $True$

Regress($\langle ?s \rightarrow s\rangle$, $stepadd(s_1, \langle p', s_2\rangle)$)

= ($\langle ?s \rightarrow s\rangle$)

where step addition decision is given by StepAdd($s_1, precond(p', s_2)$)

Preconditions: $precond(p', s_2) \in \mathcal{G}$

$s' \xrightarrow{p'} s_2 \notin \mathcal{L}$

has–effect(s_1, p'')

$unifies(p', p'')$

$$\texttt{Effects:}\ \mathcal{S} \leftarrow s_1$$
$$\mathcal{L}' \leftarrow s_1 \xrightarrow{p'} s_2$$
$$O' \leftarrow (0 \prec s_1) \wedge (s_1 \prec s_2)$$
$$\mathcal{B}' \leftarrow unify(p', p'')$$
$$\mathcal{G}' \leftarrow precond(s_1) - p'$$
$$\mathcal{F}' \leftarrow effects(s_1)$$

But from the above definition, it is clear that $stepadd$ orders the newly added step s_1 to come after initial step 0 and adds $(0 \prec s_1)$ into the partial plan. When it orders, it also adds a step-binding $\langle ?s_0 \rightarrow 0 \rangle$ into the partial plan in order to treat all the steps in a partial plan as variables and these step-bindings are noted down in the list of step-bindings. When it regresses $\langle ?s_0 \rightarrow 0 \rangle$ over the $stepadd$, it results in $True$, since it is added by the decision. Since, this binding requires only the initial state to be constant, when we regress, all the instances of $?s_0$ in the explanation are replaced with 0 (or I, a short hand notation of initial state).

The general idea behind the above regression process of bindings over various decisions is that the binding results in itself, if the decision does not add that binding. It results in a specific constant if that specific step name is required by a decision.

Let us consider the example described in Figure 7.2 to explain the above regression process. Initial explanations at leaf nodes are generalized by replacing step-names with variables and regressed over the variablized $demotion$ and $promotion$ decisions respectively. The resultant explanation at node C is (see Figure 7.2):

$E(C)$

$= Constraints\ of\ Flaw \wedge Regress(E1, demotion) \wedge Regress(E2, promotion)$

$= (?s_0 \xrightarrow{P1} s_g) \wedge \texttt{has\_effect}(?s_1,\ P1) \wedge$

$(?s_0 \prec ?s_1) \wedge$

$(?s_1 \prec ?s_g) \wedge$

$\langle (?s_0 \rightarrow 0), (?s_1 \rightarrow O1), (?s_g \rightarrow G) \rangle$

When the above explanation is regressed over $stepadd(s_1, precond(P2, s_g))$, it results in the following explanation.

$Regress(E(C), stepadd)$

$= (I \stackrel{P1}{\rightarrow} ?s_g) \ \wedge \ \langle (P2, ?s_g) \rangle \ \wedge \ \langle (?s_g \rightarrow G) \rangle$

In the above explanation, $?s_0$ is replaced with I, because when we regressed $\langle ?s_0 \rightarrow 0 \rangle$ results in $True$ and replaces all the instances of $?s_0$ in the explanation with I. If a step is specified by a name instead of variable, the step can be matched with the particular instance of the step and not any other step in the partial plan. In our example, I in the explanation matches with only the initial step and not with any other step in the plan. Thus, if a specific step name is required, regression process introduces that name into the explanation.

Let us look at the generalization of object-names. If an initial explanation requires an object to be a specified by a name, it cannot be generalized. For example, consider the following initial explanation[4].

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

This explanation states that if we need $Clear(B)$ at a step 1 where B is not a $Table$ and if the step 1 comes in between steps 0 and G, and $(0 \stackrel{On(A,B)}{\rightarrow} G)$ exists in the partial plan, it will fail. The variablized version of the above explanation does not replace the $Table$ with a variable, since the failure occurred because of B not being a $Table$. Therefore, the variablized version of the above explanation is:

$(?s_0 \stackrel{On(?A,?B)}{\rightarrow} ?s_G) \ \wedge$

[4]we will show that when SNLP+EBL learns from depth-limit failures, it starts with these kind of initial explanations.

$\langle Clear(?B), ?s_1 \rangle \ \wedge$

$(?s_0 \prec ?s_1), (?s_1 \prec ?s_G) \ \wedge$

$(?B \not\approx Table) \ \wedge$

$\langle (?s_0 \rightarrow 0), (?s_1 \rightarrow 1), (?s_g \rightarrow G)(?A \rightarrow A), (?B \rightarrow B) \rangle$

Whenever SNLP+EBL generates a rule from a generalized explanation, it removes only the codesignation bindings between the names and variables. It does not remove any non-codesignation bindings since these bindings are produced by the decisions or initial explanations as explained above[5]. By using the above explanation, if SNLP+EBL generates a rule, it will be as follows, which has no codesignation bindings between names and variables.

IF $(?s_0 \overset{On(?A,?B)}{\rightarrow} ?s_G) \ \wedge$

$\langle Clear(?B), ?s_1 \rangle \ \wedge$

$(?s_0 \prec ?s_1), (?s_1 \prec ?s_G) \ \wedge$

$(?B \not\approx Table) \ \wedge$

REJECT $NODE$

7.1 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 thesis, we ignored issues such as monitoring the utility of learned rules, and filtering bad rules. Part of the reason for this

[5]From our generalization method, it should be clear that if two variables are not denoted by same symbol, then these two variables are two different steps or objects.

was our belief that utility monitoring models developed for state-space planners [4, 10] would also apply for plan-space planners.

CHAPTER 8

LEARNING FROM DEPTH LIMIT FAILURES

In earlier chapters, we described the framework for learning search control rules from failures that are recognized by SNLP. As mentioned in Chapter 4, the only failures explained by standard SNLP+EBL are the ordering and binding inconsistencies, which it detects during threat resolution (the unestablishable condition failure is rare in practical domains). The rules learned from such failures 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 infeasible in many domains. The reason is that the planner crosses a pre-set[1] depth limit before it encounters a failure or a success as shown in the following Figure 8.1.

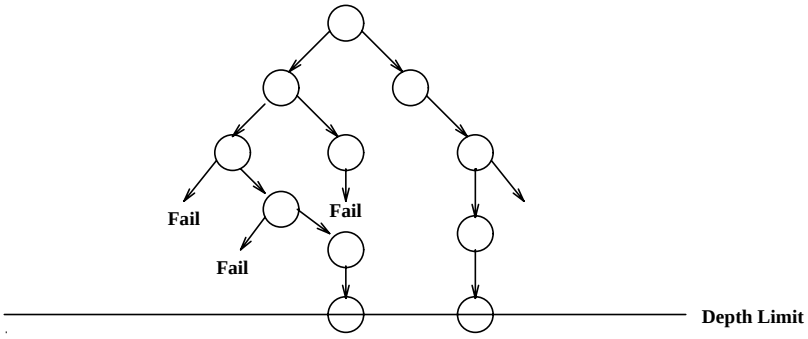


Figure 8.1: A search tree showing depth limit failures

The main 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 solution.

An example here might help to see why SNLP gets into infinite loops. In Figure 8.2, SNLP achieves $On(A, B)$ at G by establishing it from initial state. Then it tries to

[1] depth limit is set to avoid searching for infinitely long plans

achieve $On(B, C)$ at G by introducing a new step $S1$ (which corresponds to an operator $Puton(B, C)$), 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. Since there are no ordering or binding inconsistencies in the partial plan, SNLP without realizing any inconsistencies in the partial plan, it tries to achieve all the preconditions of all steps in the partial plan. But before it attempts to achieve $Clear(B)$, it could possibly cross depth limit. In this chapter, we will describe how SNLP+EBL explains implicit failures at depth limits and learn from these failures.

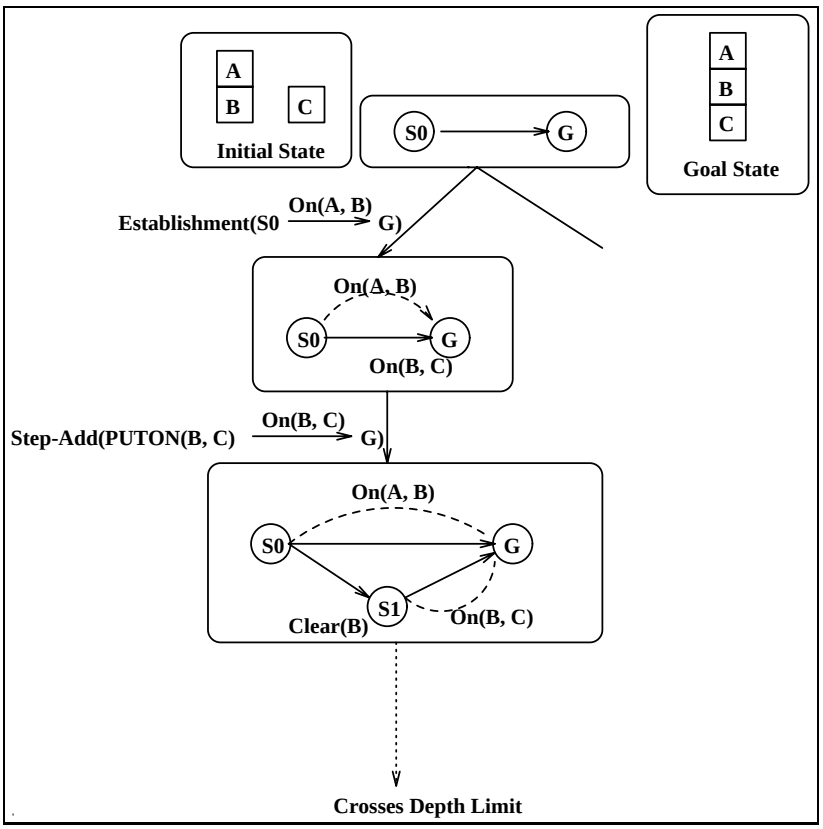
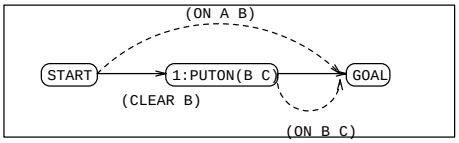


Figure 8.2: An example showing a branch of a search tree that may possibly cross depth limit

In general, if a branch of a node crosses a depth limit, the partial plan at the depth limit is non-optimal and extending such plans to generate a complete plan may not be desired, and such plans can be pruned from consideration. But when a branch crosses a depth limit, no

analytical explanation can be given, since the partial plan at depth limit is not inconsistent according to planner's consistency checks. As noted in Chapter 6, 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 a branch that crossed a depth limit. Since most of the branches of a problem cross depth limits in recursive domains, it limits EBL's ability to learn effective search control rules.

Since depth-limit failures are not analytical, no domain-independent explanation can be given to these failures. However, 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 $precond(s')$ ∪ *np–conditions*(s') is inconsistent with respect to domain axioms.[2] 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 $s_3 : Puton(x, y)$, 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 [3]): 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 x on top of it, and be clear, unless y is the table.

$On(x, y) \wedge clear(y)[y \not\approx Table]$
$On(x, y) \wedge On(x, z)[y \not\approx z]$
$On(x, y) \wedge On(z, y)[x \not\approx z, y \not\approx Table]$

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) ∪ $precond(s)$. Given this theory, we can now explain and learn from the blocks-world partial plan above. The initial explanation of this failure is: $\texttt{start} \overset{\texttt{On(x,y)}}{\rightarrow} \texttt{G} \wedge (\texttt{start} \prec \texttt{1}) \wedge (\texttt{1} \prec \texttt{G}) \wedge \texttt{precond(Clear(y), 1)} \wedge \texttt{y} \not\approx \texttt{Table}$. 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

[2] In fact the plan will also fail if $effects(s')$ ∪ *np–conditions*(s') is inconsistent. However, given any action representation which makes STRIPS assumption, these inconsistencies will any way be automatically detected by the normal threat detection and resolution mechanisms.

(1) **Reject establishment** $\texttt{start} \overset{On(x,y)}{\rightarrow} s_1$
If $\texttt{precond}(On(y,z), s_1) \wedge$
$\neg\texttt{initially\_true}(On(y,z)) \wedge \neg\texttt{binds}(y, Table)$
(2) **Reject promotion** $s_1 \prec s_3$
If $\texttt{precond}(clear(x_2), s_3) \wedge$
$\texttt{establishes}(s_1, On(x_1, x_2), s_2) \wedge$
$\texttt{precedes}(s_3, s_2) \wedge \neg\texttt{binds}(x_2, Table)$
(3) **Reject step addition** $puton(x', y) \overset{Clear(z)}{\rightarrow} s_1$
If $\texttt{establishes}(\texttt{start}, On(x,y), s_2) \wedge$
$\texttt{precedes}(s_1, s_2) \wedge \neg\texttt{binds}(y, Table) \wedge$

Figure 8.3: A sampling of rules learned using domain-axioms in Blocks world domain

failures[3] In the blocks world, use of this technique enabled SNLP+EBL to produce several useful search control rules. Figure 8.3 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 $On(y,z)$.

[3]When the analysis unearths multiple failure explanations for a depth-limit plan, we prefer the explanations containing steps introduced at shallower depths (in our case, smaller step numbers). This tends to help the dependency direct backtracking component during the learning phase

CHAPTER 9

EXPERIMENTS

Until now, we have described the learning framework in SNLP+EBL and showed how these rules could improve the performance of the system. Providing search control rules to a planner alone is not sufficient. Planner needs to match every rule from the rule storage at every decision point. The cost to match rules could offset the benefits provided by the rules. In this chapter, we present the results of two sets of experiments conducted to evaluate the effectiveness of the rules learned by the system. The objective of the experiments was to show that the rules learned by SNLP+EBL improve the performance of the planner over the baselevel planner.

First, we have conducted experiments on random blocks world problems without any constraints on the problem characteristics. In these problems, the planner did not do much of search as these problems turned out to be very easy to solve. For this reason, we have conducted two sets of experiments in blocks world domain. In the first set of the experiments the planner was given random initial states and goal states with 3-block (e.g (On A B) (On b C)) stacks. In the second set of the experiments the planner was given random problems with a constraint on the minimum size of the problem.

Setup: Initial states and goal states of the problems in each test case are generated using the procedure defined in [8]. This procedure takes two parameters MAX-BLOCKS and MAX-GOALS. First, the number of blocks, from 3 to MAX-BLOCKS is generated randomly. To generate the initial state, the following sub-procedure is used, For each block, with probability 1/3 put it on the table, otherwise randomly select a previously generated stack, and place the block on top. To create goal states, the same subprocedure used to generate the initial state is used. Then a set of goals is selected as follows. Each of assertions in

the goal state is filtered with probability 2/3 if it is true in the initial state. Then a random number, from 1 to MAX-GOALS, of these assertions are conjoined to form goal state. If all the goals are already in the initial state the procedure is repeated.

For each test set, the planner was run on a set of randomly generated problems to learn search control rules. During the testing phase, the two test sets of problems were run with SNLP, SNLP+EBL (with the saved rules). To evaluate the relative effect of EBL vs domain-axiom based check, we also have run a different version of SNLP, SNLP+Domax, which uses domain axioms based consistency checks. Specifically, SNLP+Domax uses domain axioms based consistency checks to see a partial plan is inconsistent with the domain theory. If it finds inconsistency in the partial plan, it prunes the branch as soon as it is generated. Applying such strong consistency checks at every decision point while planning increases the cost of generating a node. Thus, we predicted that the use of domain axioms based consistency checks increases the cost of the planning unduly and does not perform better than SNLP+EBL, which uses domain axioms to generate search control rules at the depth limits to explain failures.

A cpu time limit of 120 seconds was used in each test set. If the planner takes more than 120 seconds to solve a problem, the planner aborts the search and returns as failed. We chose to compare the cpu time limits so that it takes the matching cost of the rules[1] into account while providing the performance improvements by the search control rules.

Test Set I: This set consists of 30 problems that had randomly generated initial states consisting of 3 to 8 blocks and the goal states consisting of only one 3-block stack. SNLP+EBL was run on 20 random problems to learn search control rules from the failures encountered by the planner. SNLP+EBL learned 10 search control rules from these 20 problems and these rules are used during the testing phase.

[1]matching cost of a rule is the cost that is taken by the planner to match a rule against the partial plan at a decision point

Test Set II: This set consists of 100 problems had randomly generated initial states consisting of 3 to 8 blocks and random goal states. SNLP+EBL was run on 50 random problems to learn search control rules from the failures encountered by the planner. SNLP+EBL learned 15 search control rules from these 50 problems and these rules are used during the testing phase.

In both of these test sets, a rule is added into the rule storage, if it is used atleast once while learning the rules. Since very few number of rules learned in each test set, we did not do any sophisticated utility analysis like average matching cost into consideration to add a rule into the rule storage.

Results: Table 9.1 describes the results of these experiments. Figure 9.1 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 in terms of cpu time significantly on these problem populations. 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 generated during planning. 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 solve 19% of the test problems in the second set shows that there may be other avenues for learning search control rules. We intend to explore these in our future work (see chapter 10).

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 9.1: Results from the blocks world experiments

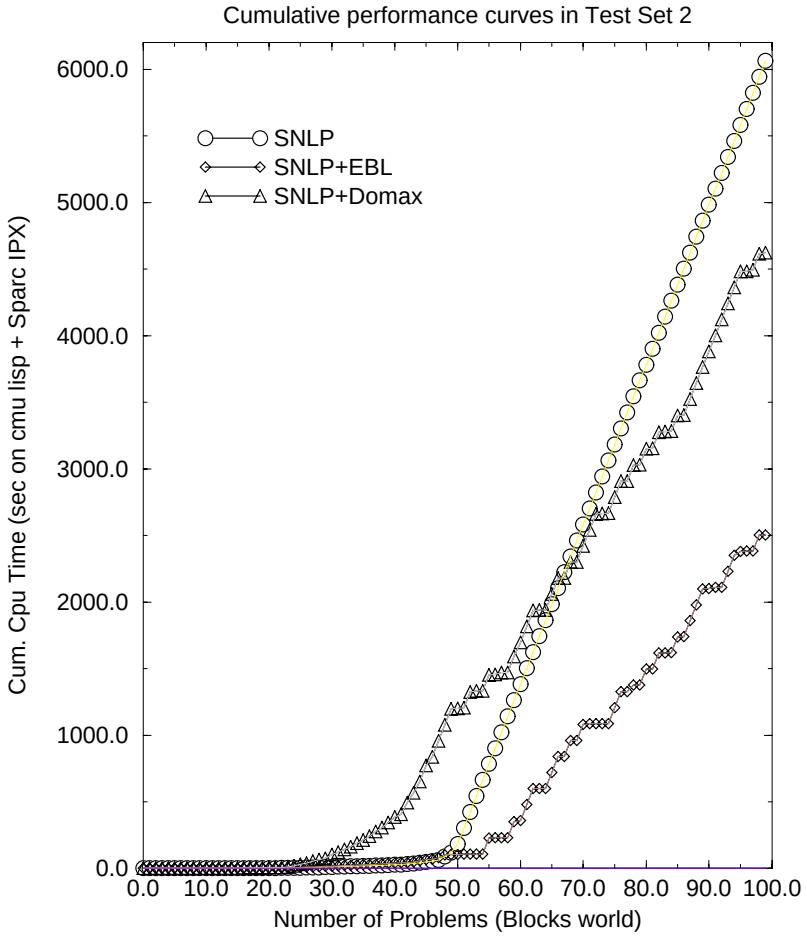


Figure 9.1: Cumulative performance curves for Test Set 2

CHAPTER 10

CONCLUSIONS AND FUTURE WORK

In this thesis, we presented SNLP+EBL, the first systematic implementation of EBL search-control learning in a partial-order planner. We have described the details of the initial explanation construction, regression and explanation propagation, and rule learning process in SNLP+EBL. We have then proposed a general methodology for learning from planning failures, viz., using a battery of stronger consistency checks based on the meta-theory of the planner, and the domain theory of the problem, to detect and explain failures at depth limits. We described a specific instantiation of this method, which uses domain axioms to look for inconsistencies in the plans at depth limits, and presented experimental results showing that the search control rules that SNLP+EBL learns using this technique enable it to outperform SNLP.

The current work shows that EBL provides a way of strategically applying domain-axiom-based consistency checks. Finally, in this thesis, I 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 [4, 10] will also apply for plan-space planners. In our future work, better utility models can be incorporated into SNLP+EBL.

Learning from domain-axiom failures alone may not be sufficient in domains which do not have any strong implicit domain theory. In future, identifying other types of stronger consistency checks which can be used to complement the domain-axiom techniques in such domains and learn search control rules will be an interesting area to explore. These include using the full finite-domain variable consistency checks to learn to avoid certain types of variable-based looping, utilizing domain-specific theories of loop detection to

avoid step-based looping, as well as using strong causal link consistency checks as a way of learning about conflict deferment [12]. Applying the EBL methodolgy that is used in SNLP+EBL for a planner with better representation of actions and allows universal and existential conditions such as UCPOP will be interesting. Currently, my colleague Yong Qu is conducting experiments on the effectiveness of EBL on UCPOP.

APPENDIX A

Issues Regarding Soundness of SNLP+EBL's Search Control Rules

As explained in chapter 7, a rule is said to be sound if it does not affect the completeness of the underlying planner. It can be defined in different ways as follows:

- Strong Soundness: there are no solutions under the branch that is rejected by the planner.
- Minimal Soundness: there are no minimal solutions existing under the branch that is rejected by the planner (where a solution is minimal if no operator sequence produced by removing steps from it is a solution).
- Solution Soundness: there could be solutions under the branch that is rejected by the planner, but there exists atleast one other solution in the over all search space.

All the above criteria for soundness of a rule preserve the completeness of the underlying planner (that is, if a problem is solvable, the planner is guaranteed to solve it). In chapter 7, we argued that the rules learned by SNLP+EBL preserve strong soundness.

In order to preserve the strong soundness, a rule should guarantee that it is not removing any solution from the search space. When the planner fails during the search, it invokes the learner to learn a search control rule. Learner constructs an initial explanation for the failure as explained in 4.1. The initial explanation for any failure is a set of inconsistent constraints. Since a plan with inconsistent constraints cannot be refined to a successful plan, rules that are constructed from the initial explanations are sound.

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 all the failures of the branches under that node. For this, an explanation at any intermediate node should be the conjunction of the failure explanations emerging from all the branches under that node. To construct a sound explanation at any intermediate node, the planner should thus explore all branches of that node and explain the failures for all those branches. However, the planner may not

generate some branches under a node because of existing constraints in the partial plan of the node. For example, suppose s_1 and s_2 are two steps in a partial plan and $s_1 \prec s_2$ and s_2 has an effect p, and s_1 needs the condition p. The planner will not generate a simple establishment branch to achieve p at s_1 from the step s_2. If we learn a rule here, we want to know whether that the rule would be applicable in another situation where $s_1 \not\prec s_2$. The issue thus becomes one of ensuring that all possible and potential search branches are properly accounted for in generating a failure explanation. In following paragraphs, we explain how this is done.

While planning, at any node the planner, SNLP, resolves a conflict or achieves an open condition at a step. For resolving a conflict, the planner has only two choices to order the threat by promotion or by demotion. To resolve a conflict, we modified SNLP such that it always generates two branches to account for promotion and demotion irrespective of the constraints in the partial plan.

For achieving an open condition at a step, the planner has two choices to achieve the open condition by step addition or by simple establishment. Since the number of operators available in the domain are fixed, the number of branches that are generated by the step addition are fixed irrespective of the constraints in the partial plan. However, the number of simple estbalishment branches under a node are 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 seperated into two categories, establishments from initial state and establishments from steps other than initial state. We will treat these two cases differently.

Simple establishments from initial state: Since initial state changes from problem to problem, the number of simple establishment branches from initial state may vary too. For example, suppose we are trying to achieve 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 gave 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).
- Counterfactual Reasoning: The approach of qualifying explanations may lead us sometimes to over specific explanations. For example, it may be that the simple estblishment 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 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 implementation.

The approach of qualifying the explanation will 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, we need to qualify that the initial state gives only $P(A)$ and not $P(B)$, $P(C)$ etc,. In other words, the qualification is

$x \not\approx A \Rightarrow \neg$`initially–true`($P(x)$)

While this is possible to do, the resulting explanations can be too specific and also 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. Therefore, this restriction does not affect the efficiency of the learner. Now, we have completed explaining

about simple establishments from the initial state. The final issue is simple establishments from steps other than the initial state.

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 establishment possibilities change from problem to problem. In SNLP+EBL, we currently consider only those explanations that account for failures of establishments that are generated by the planner.

It may look as if we need to consider explanations of all possible establishment failures and conjoin these explanations along with other explanations of the node. All possible establishments include the establishments that are actually generated by the planner as well as establishments that could have been possible, if certain ordering constraints or binding constraints are not present in the partial plan. This is not required because even though certain simple establishments are not generated by the planner because of the constraints in the partial plan, planner considers all the step-addition possibilities involving the same operators. If it fails to establish an open condition from a step because of ordering or binding constraints, it must be because it can not do so even if it were allowed to have fresh copies of all the operators. Thus, learner need not conjoin the explanations of the branches that did not give the condition because of ordering or binding constraints in the partial plan.

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

[4] 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

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

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

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

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

[9] 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.

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

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

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