An Explanation-Based Approach to Improve Retrieval in Case-Based Planning

Laurie Ihrig & Subbarao Kambhampati*
Department of Computer Science and Engineering
Arizona State University, Tempe, AZ 85287-5406
Email: laurie.ihrig@asu.edu rao@asu.edu

Appears in Current Trends in AI Planning: EWSP '95, IOS Press

Abstract

When a case-based planner is retrieving a previous case in preparation for solving a new similar problem, it is often not aware of the implicit features of the new problem situation which determine if a particular case may be successfully applied. This means that some cases may be retrieved in error in that the case may fail to improve the planner's performance. Retrieval may be incrementally improved by detecting and explaining these failures as they occur. In this paper we provide a definition of case failure for the planner, DERSNLP (derivation replay in SNLP), which solves new problems by replaying its previous plan derivations. We provide EBL (explanation-based learning) techniques for detecting and constructing the reasons for the failure. We also describe how to organize a case library so as to incorporate this failure information as it is produced. Finally we present an empirical study which demonstrates the effectiveness of this approach in improving the performance of DERSNLP.

1 Introduction

Case-based planning provides significant performance improvements over generative planning when the planner is solving a series of similar problems, and when it has an adequate theory of problem similarity [5, 6, 12, 16]. One approach to case-based planning is to store plan derivations which are then used as guidance when solving new similar problems [2, 16]. Recently we adapted this approach, called *derivational replay*, to improve the performance of the partial-order planner, SNLP [6]. Although we found that replay tends to improve overall performance, its effectiveness depends on retrieving an appropriate case. Often the planner is not aware of the implicit features of the new problem situation which determine if a certain case is applicable.

Earlier work in case-based planning has retrieved previous cases on the basis of a static similarity metric which considers the previous problem goals as well as the features of the

*This research is supported in part by NSF research initiation award (RIA) IRI-9210997, NSF young investigator award (NYI) IRI-9457634, and ARPA/Rome Laboratory planning initiative grant F30602-93-C-0039. Thanks to Suresh Katukam and Biplav Srivastava for helpful comments.

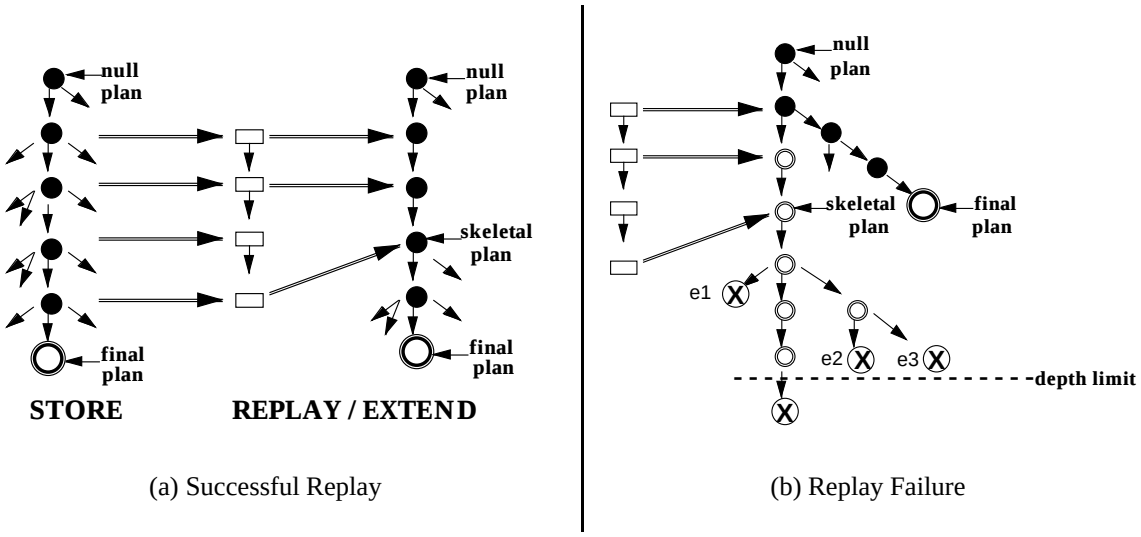
initial state which are relevant to the achievement of those goals [10, 6]. If these are elements of the new problem situation then the case is retrieved and reused in solving the new problem. Usually the new problem contains extra goal conditions not covered by the case. This means that the planner must engage in further planning effort to add constraints (including plan steps and step orderings) which achieve the conditions that are left open. Sometimes an extra goal will interact with the covered goals and the planner will not be able to find a solution to the new problem without backtracking and retracting some of the replayed decisions. In the current work we treat such instances as indicative of a case failure. We provide a framework by which a planner may learn from the case failures that it encounters and thereby improve its case retrieval.

We present the derivation replay framework, DERSNLP+EBL, which extends DERSNLP, a replay system for a partial-order planner, by incorporating explanation-based learning (EBL) techniques for detecting and explaining analytical failures in the planner's search space. These include a method for forming explanations of plan failures in terms of their inconsistent constraints, and regressing these explanations through the planning decisions in the failing search paths [11]. These techniques are employed to construct reasons for case failure, which are then used to annotate the case to constrain its future retrieval. We evaluate the effectiveness of using case failure information in an empirical study which compares the performance of DERSNLP on replay of cases both *with and without* failure information.

The rest of the paper is organized as follows. In Section 2, we first describe DERSNLP which implements our approach to derivation replay in the partial-order planner, SNLP. We discuss a definition of case failure for DERSNLP, and show how DERSNLP recovers from the case failures it encounters. Then, in Section 3 we briefly describe the explanation-based learning techniques that we have developed in [11], including the construction of failure explanations, and their regression and propagation up the search tree. We then show how reasons for case failure are constructed using these techniques, and how these failure reasons are used to refine the labeling of cases in the library. Section 4 describes an empirical study demonstrating the relative effectiveness of replay when this failure information is utilized. Finally, in Section 5 we discuss the relationship to previous work in case storage and retrieval.

2 Derivation Replay in Partial-Order Planning

Derivational analogy is a case-based planning technique which includes all of the following elements [2, 16, 17]: a facility within the underlying planner to generate a trace of the derivation of a plan, the indexing and storage of the derivation trace in a library of previous cases, the retrieval of a case in preparation for solving a new problem, and finally, a replay mechanism by which the planner utilizes a previous derivation as a sequence of instructions to guide the search process in solving a new problem. Figure 1a illustrates the replay of a derivation trace. Whenever a new problem is attempted and a solution is achieved, a trace of the decisions that fall on the derivation path leading from the root of the search tree to the final plan in the leaf node is stored in the case library. Then, when a similar problem is encountered, this trace is used as a sequence of instructions to guide the new search process. DERSNLP employs an *eager* replay strategy. With this strategy, control is shifted to the series of instructions provided by the previous derivation, and is returned to from-scratch planning only after all of the valid instructions in the trace have been replayed [6, 7]. This means that the plan which is produced through replay, called the *skeletal plan*, contains all of the constraints that were added on the guidance of the one previous trace. When the skeletal



(a) Successful Replay

(b) Replay Failure

Figure 1: *Schematic characterization of derivation storage and replay. Each time that a plan is derived, the decisions contained in the plan derivation (shown as filled circles to the left in Figure (a)) are stored as a sequence of instructions (shown as open rectangles) which are used to guide the new search process. The guidance provided by replay is considered successful if the skeletal plan that replay produces can be extended (by the addition of constraints) into a solution to the new problem. If replay has been successful, the skeletal plan lies on the new derivation path leading to the solution (the new derivation path is shown to the right of Figure (a) as a sequence of filled circles). A replay failure is indicated when the skeletal plan has to be backtracked over in order to reach a solution. In Figure (b), a solution to the new problem could be reached only by backtracking over the skeletal plan, which now lies outside the new plan derivation (shown as filled circles). Explanations are constructed for the failing plans in the leaf nodes of the subtree underneath the skeletal plan, and are regressed up the tree to compute the reasons for case failure.*

plan contains open conditions relating to extra goals not covered by the earlier case, further planning effort is required to extend this plan into a solution for the new problem.

In the current work *we define replay success and failure in terms of the skeletal plan. Replay is considered to fail if the skeletal plan which contains all of the constraints prescribed by the old case cannot be extended by the addition of further constraints into a solution for the new problem* (See Figure 1b). In such instances, the planner first explores the failing subtree underneath the skeletal plan, then backtracks over the replayed portion of the search path until a solution can be found. Replay failure results in poor planning performance, as illustrated in Figure 2, since it entails the additional cost of retrieving a trace from the library, as well as the cost of validating each of the decisions in the trace. This means that when replay fails and the planner has to backtrack over the skeletal plan performance may be worse than in from-scratch planning.

When a case fails, and the planner goes on to find a new solution, the final plan that it reaches does not contain some of the constraints that are present in the skeletal plan. The derivation path which leads from the root of the search tree to the final plan in the leaf node thus avoids (or *repairs*) the failure encountered in replaying the old case. Consider a simple example taken from the logistics transportation domain of [16] (See Figure 3a). The goal is to have package OB1 located at the destination location l_d. The package is initially at location

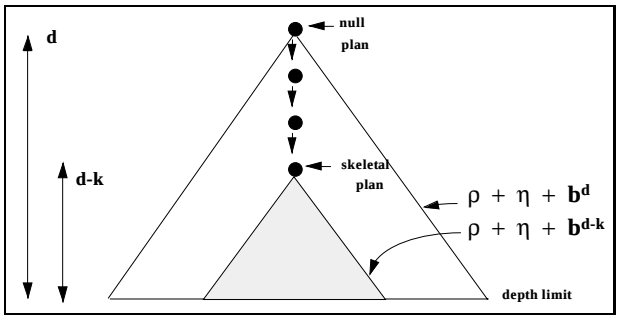
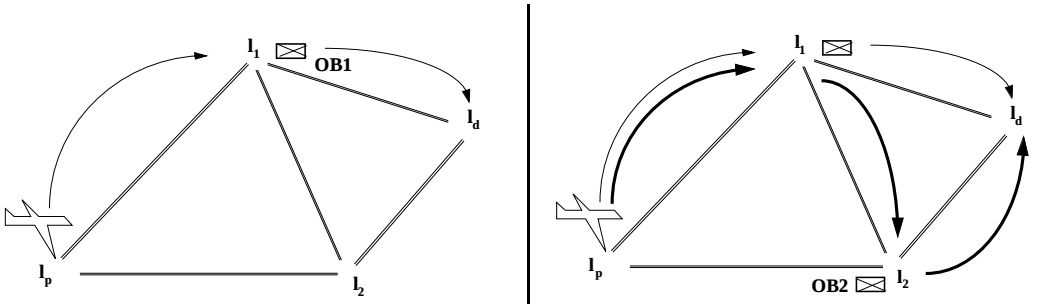


Figure 2: *Successful replay reduces the size of the search. However it entails the additional cost, ρ , of retrieving a trace from the library, as well as the cost, η, of validating each of the decisions in the trace. When replay fails performance may be worse than in from-scratch planning.*



(a) Previous Case

(b) New Problem with Extra Goal

Figure 3: *An illustration of replay success and failure. In the previous case (shown in Figure (a)) a plan has been derived to accomplish the transport of a single package, OB1, to the destination airport l_d. Decisions were made which establish the plane's route (indicated in Figure (a) by the curved arrows), as well as the loading and unloading of the package. The new problem, illustrated in Figure (b), contains an extra goal which involves the transport to l_d of a second package, OB2, which is initially located off the previous route. The planner must backtrack and choose an alternative route (shown in bold) in order to accomodate the extra goal.*

l_1. There is a plane located at l_p which can be used to transport the package. A previous case which solves this problem will contain decisions which add steps that determine the plane's route to the destination airport as well as steps which accomplish the loading of the package at the right place along this route. These decisions may be readily extended to load and unload extra packages which lie along the same route. However, if the new problem involves the additional transport of a package which is off the old route, the planner may not be able reach a solution without backtracking over some of the replayed decisions (See Figure 3b). The new derivation path makes some alternative choices in achieving the goal which was covered by the previous case. Although it arrives at a plan which contains more steps, this plan may be readily extended to solve the extra goal.

Having defined case failure and provided an example, we are now in a position to describe how the planner learns the reasons underlying a case failure. Specifically, we use EBL techniques to accomplish this learning. In the next section, we show how the EBL techniques developed in [11] are employed to construct reasons for case failure.

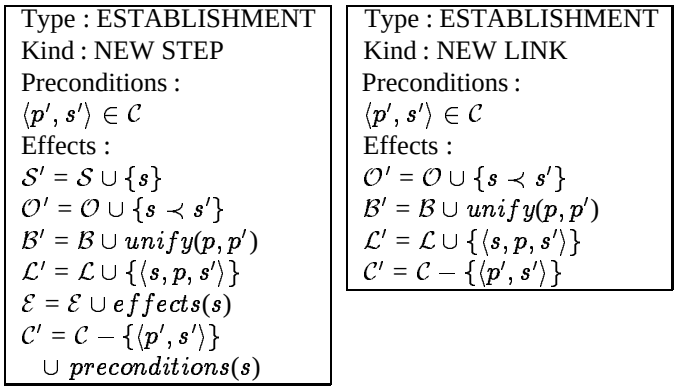


Figure 4: *Planning decisions are based on the current active plan* $\langle \mathcal{S}, \mathcal{O}, \mathcal{B}, \mathcal{L}, \mathcal{E}, \mathcal{C} \rangle$ *and have effects which alter the constraints so as to produce the new current active plan* $\langle \mathcal{S}', \mathcal{O}', \mathcal{B}', \mathcal{L}', \mathcal{E}', \mathcal{C}' \rangle$.

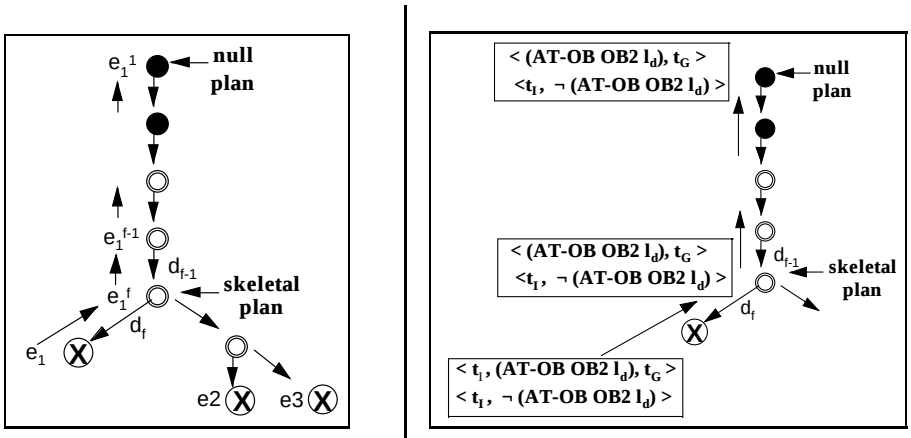
3 Learning from Case Failure

DERSNLP+EBL constructs case failure reasons incrementally as the skeletal plan is extended by the addition of constraints and failures are encountered. As Figure 1b illustrates, DERSNLP+EBL forms explanations for the failures that occur in the subtree rooted at the skeletal plan. [1] Path failure explanations identify a minimal set of conflicting constraints in the plan which are together inconsistent. These are then regressed up the tree to construct reasons for case failure.

Since a plan failure is explained by a subset of plan constraints, failure explanations are represented in the same manner as a partial plan. DERSNLP represents its partial plans as a 6-tuple, $\langle \mathcal{S}, \mathcal{O}, \mathcal{B}, \mathcal{L}, \mathcal{E}, \mathcal{C} \rangle$, where [1]: $\mathcal{S}$ is the set of actions (step-names) in the plan, each of which is mapped onto an operator in the domain theory. $\mathcal{S}$ contains two dummy steps: t_I whose effects are the initial state conditions, and t_G whose preconditions are the input goals, G. $\mathcal{B}$ is a set of codesignation (binding) and non-codesignation (prohibited binding) constraints on the variables appearing in the preconditions and post-conditions of the operators which are represented in the plan steps, $\mathcal{S}$. $\mathcal{O}$ is a partial ordering relation on S, representing the ordering constraints over the steps in $\mathcal{S}$. $\mathcal{L}$ is a set of causal links of the form $\langle s, p, s' \rangle$ where $s, s' \in \mathcal{S}$. A causal link contains the information that s causes (contributes) p which unifies with a precondition of s'. $\mathcal{E}$ contains step effects, represented as $\langle s, e \rangle$, where $s \in \mathcal{S}$. $\mathcal{C}$ is a set of open conditions of the partial plan, each of which is a tuple $\langle p, s \rangle$ such that p is a precondition of step s and there is no link supporting p at s in $\mathcal{L}$.

The explanation for the failure of the partial plan contains the constraints which contribute to an inconsistency in the plan. These inconsistencies appear when new constraints are added which conflict with existing constraints. DERSNLP makes two types of planning decisions, *establishment* and *resolution*. Each type of decision may result in a plan failure. For example, an *establishment* decision makes a choice as to a method of achieving an open condition, either through a new/existing plan step, or by adding a causal link from the initial state. When an attempt is made to achieve a condition by linking to an initial state effect, and this condition is not satisfied in the initial state, the plan then contains a contradiction. An

[1] Depth limit failures are ignored. This means that the failure explanations that are formed are not sound in the case of a depth limit failure. However, soundness is not crucial for the current purpose, since explanations are used only for case retrieval and not for pruning paths in the search tree.



(a) The Regression Process

(b) A Detailed Example

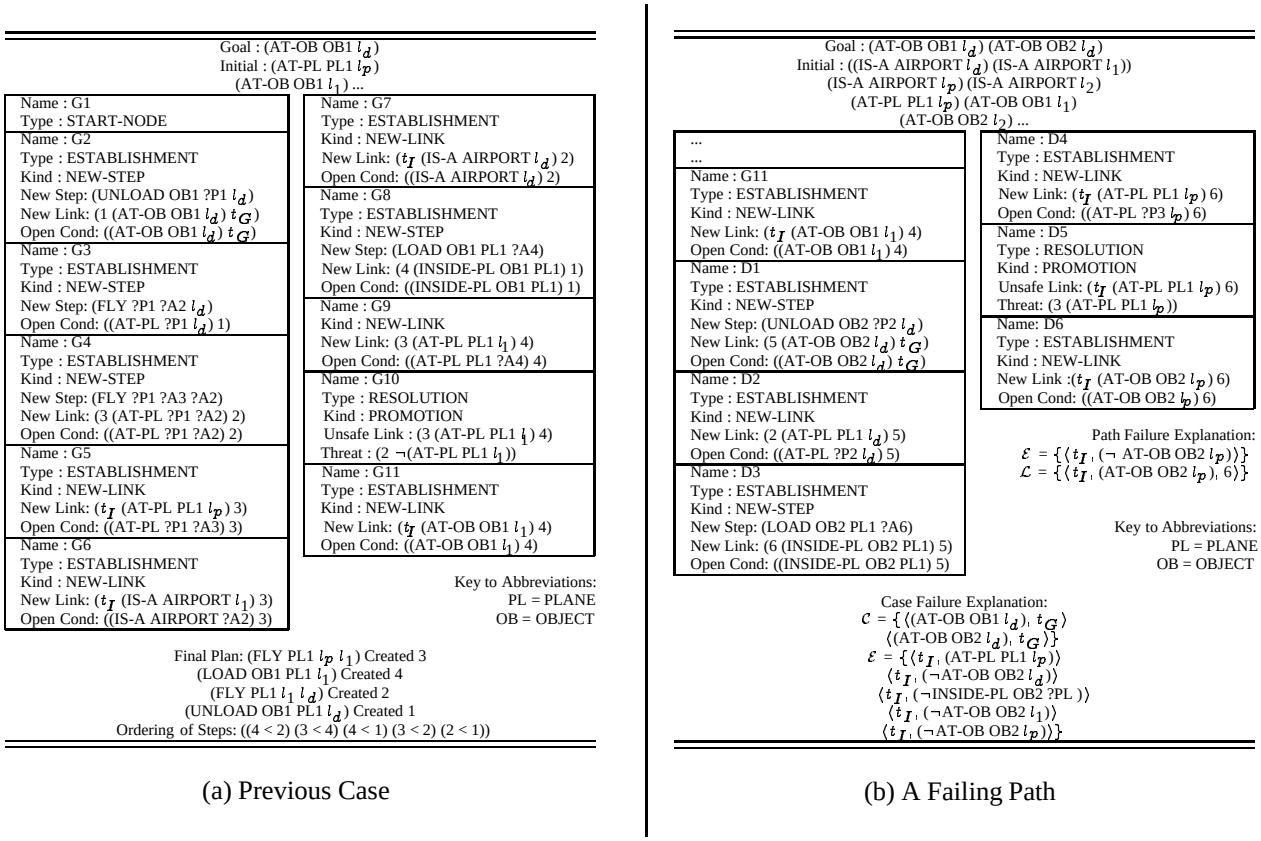
Figure 5: *Explanations are constructed for the failing plans in the leaf nodes of the subtree rooted at the skeletal plan, and are regressed up the tree. The path failure explanation at the root of the tree is computed as* $e_1^1 = d_1^{-1}(d_2^{-1} \cdots (d_f^{-1}(e_1)) \cdots)$*. The case failure reason represents a combined explanation for each of the path failures.*

explanation for the failure is constructed which identifies the two conflicting constraints: $\langle \emptyset, \emptyset, \emptyset, \{\langle t_I, p, s \rangle\}, \{\langle t_I, \neg p \rangle\}, \emptyset \rangle$.

As soon as a path failure is detected and an explanation is constructed, the explanation is regressed through the decisions in the failing path up to the root of the search tree. In order to understand the regression process, it is useful to think of planning decisions as STRIPS-style operators acting on partial plans (See Figure 4). The preconditions of these operators are specified in terms of the plan constraints that make up a plan flaw, which is either an open condition or a threat to a causal link. The effects are the constraints that are *added* to the partial plan to eliminate the flaw.

Each of the conflicting constraints in the failure explanation is regressed through the planning decision, and the results are sorted according to type to form the new regressed explanation. As an example, consider that a new link from the initial state results in a failure. The explanation, e_1 is: $\langle \emptyset, \emptyset, \emptyset, \{\langle t_I, p, s \rangle\}, \{\langle t_I, \neg p \rangle\}, \emptyset \rangle$ When e_1 is regressed through the final decision, d_f to obtain a new explanation, e_1^f, the initial state effect regresses to itself. However, since the link in the explanation was added by the decision, d_f, this link regresses to the open condition which was a precondition of adding the link. The new explanation, e_1^f, is therefore $\langle \emptyset, \emptyset, \emptyset, \emptyset, \{\langle t_I, \neg p \rangle\}, \{\langle p, s \rangle\} \rangle$. The regression process continues up the failing path until it reaches the root of the search tree. This process is illustrated graphically in Figure 5a. A more detailed example is provided in Figure 5b. When all of the paths in the subtree underneath the skeletal plan have failed, the failure reason at the root of the search tree provides the reason for the failure of the case. It represents a combined explanation for all of the path failures. The case failure reason contains only the aspects of the new problem which were responsible for the failure. It may contain only a subset of the problem goals. Also, any of the initial state effects that are present in a leaf node explanation, are also present in the reason for case failure.

An Example of Case Failure: Figure 6a provides a trace of DERSNLP's decision process in arriving at a solution to the simple problem taken from the logistics transportation domain.



(a) Previous Case

(b) A Failing Path

Figure 6: *An Example Trace of a Failing Path for* DERSNLP

We will use this trace as an example and again assume that this derivation is replayed in solving a new problem that contains an extra goal. The trace in Figure 6a derives a plan to solve a problem which has a single goal, that OB1 be at the destination airport, l_d.

Starting with the null plan DERSNLP first chooses a method of achieving this open condition. The first step to be added to the plan unloads the package, OB1, at its destination, l_d. Planning continues by adding a second step to fly a plane to the package destination. This step accomplishes a precondition of the `unload` action, which is that the plane has to be at l_d in order to unload the package. As planning continues more steps are added, and some of the open conditions are linked to existing steps, or to the initial world state. Eventually a threat is detected. The flight to the final destination location threatens a precondition of the `load` action, since the plane has to be at the package location in order to load the package. This threat is resolved by promoting the flight to the package destination to follow the loading of the package. The final plan is shown at the bottom of Figure 6a.

If the derivation in Figure 6a is replayed in solving a new problem where there is an extra package to be transported to the same destination, and this package lies outside the plane's route, replay will fail. Figure 6b illustrates how the decisions that DERSNLP makes in extending the derivation result in a path failure. Although the subtree underneath the skeletal plan fails, for illustration purposes only one failing path is traced in Figure 6b. The figure shows only the latter portion of the path, starting with the skeletal plan. DERSNLP first attempts to extend the replayed path by adding the `unload` and then the `load` actions. A precondition of a `load` is that the plane is at some location. DERSNLP links the plane's location to the initial world state. The plane originates at l_p. This decision is then followed

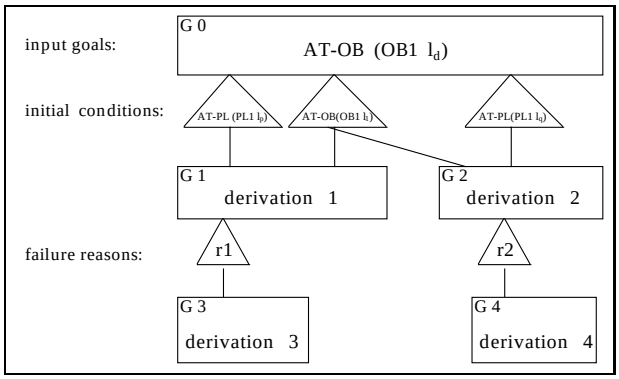


Figure 7: *Local organization of the case library.*

by a decision to link the object's location to the initial state. This results in an inconsistency in the plan since the object is not at l_p initially (See path failure explanation in Figure 6b). The full case failure reason is shown at the bottom of Figure 6b. It gives the conditions under which a future replay of the same case will result in failure. A summary of the information content of the explanation is: *There is an extra package to transport to the same destination location, and that package is not at the destination location, is not inside the plane, and is not located on the plane's route.* This explanation identifies implicit features of the problem description which are predictive of failure and are used to censor retrieval.

Library Organization: Each time a case fails the failure reason is used to annotate the case in the library. The library fragment depicted in Figure 7 is indexed by a discrimination net and represents all of the cases which solve a single set of input goals. Individual cases which solve these goals are represented one level lower in the net. Each case is labeled by the relevant initial state conditions. When one of these cases is retrieved for replay and the case fails, it is then annotated with the reason for the case failure. When the case is retrieved again this failure reason is tested in the new problem situation, and, if satisfied, the retrieval process returns the alternative case that repairs the failure. The case failure reason is thus used to direct retrieval away from the case which will repeat a known failure, and towards the case that avoids it.

4 Empirical Evaluation of the Retrieval Strategy

In this section we describe an empirical study which was conducted with the aim of demonstrating the advantage of retrieving cases on the basis of previous replay failures. We chose domains in which randomly generated problems contained interacting goals, and tested planning performance when DERSNLP was solving n-goal problems from scratch and through replay of a similar (n-1)-goal problem. We compared the performance improvements provided by replay when information about the previous case failure was not used in case retrieval vs when it was used.

4.1 Experimental Method

Domains: Experiments were run on problems drawn from two domains. The first was the artificial domain, ($\theta_2 D^m S^1$), originally described in [1] and shown in Figure 8. Testing was done on problems which were randomly generated from this domain with the restriction that they always contain the goal G_α. Notice that if G_α is one of a set of problem goals, and it

$(\theta_2 D^m S^1)$:
$(A_i^\alpha \ \underline{precond}: \ \{I_i, P_\alpha\} \ \underline{add}: \ \{G_i\} \ \underline{delete}: \ \{I_j | j < i\})$
$(A_i^\beta \ \underline{precond}: \ \{I_i P_\beta\} \ \underline{add}: \ \{G_i\} \ \underline{delete}: \ \{I_j | j < i\})$
$(A_\alpha \ \underline{precond}: \ \{\} \ \underline{add}: \ \{G_\alpha\} \ \underline{delete}: \ \{P_\beta\} \cup \{G_i | \forall i\})$

Figure 8: *The specification of Barrett and Weld's Transformed ($D^m S^1$)Domain*

is not true initially, then any other goal, G_i, that is present in the set must be achieved by the operator A_i^α, and not by A_i^β. This means that any time a case is replayed that previously solved a goal, G_i, through an action A_i^β, and G_α as an extra goal not covered by the case, then replay will fail.

The logistics transportation domain of [16] was adopted for the second set of experiments. Eight packages and one airplane were randomly distributed over four cities. Problem goals represented the task of getting one or more packages to a single destination airport. The `fly` operator was augmented with a delete condition which prevented planes from visiting the same airport more than once. This meant that replay failed for DERSNLP if there was an extra package to be transported which was off the previous route taken by the plane.

Retrieval Strategy: Cases were initially retrieved on the basis of a *static* similarity metric which takes into account the new goals that were covered by the case as well as all of their relevant initial state conditions. The metric was similar to the validation structure-based similarity metric of [9] and the foot-printed similarity metric of [16]. Prior studies show it to be a reasonably effective metric. In *learning* mode, cases were also retrieved on the basis of the same metric. However, in this mode, the failure reasons attached to the case were used to censor its retrieval. Each time that a case was retrieved in learning mode, these failure conditions were also tested. If each failure reason was not satisfied in the new problem specification, the retrieval mechanism returned the case for replay. If, on the other hand, a failure reason was found to be true in the new problem context, then the case that repaired the failure was retrieved. Following retrieval, the problem was solved both by replay of the retrieved case as well as by planning from scratch.

Experimental Setup: Each experiment consisted of three phases, each phase corresponding to an increase in problem size. Goals were randomly selected for each problem, and, in the case of the logistics domain, the initial state was also randomly varied between problems. In an initial training session that took place at the start of each phase n, 30 n-goal problems were solved from scratch, and each case was stored in the case library. Following training, the testing session consisted of randomly generating problems in the same manner but with an additional goal. Each time that a new $(n+1)$ goal problem was tried, an attempt was made to retrieve a similar n-goal problem from the library. Cases were chosen so as to have all but one of the goals matching. This meant that the skeletal plan that was returned by the replay procedure contained one open condition representing a goal corresponding to an extra input goal.

If during the testing session, a case that was similar to the new problem was found which had previously failed and that case was annotated with a failure reason, then the problem was solved in learning, static and from-scratch modes, and the problem became part of the 30-problem set. With this method, we were able to evaluate the improvements provided by failure-based retrieval when retrieval on the static metric alone was ineffective, and when failure conditions were available. If no similar case with a failure reason was found in the library, then the problem was attempted through replay, and if a replay failure occurred, the

Phase	$(\theta_2 D^m S^1)$			Logistics		
	Learning	**Static**	**Scratch**	**Learning**	**Static**	**Scratch**
(1) Two Goal						
%Solved	100%	100%	100%	100% (6.0)	100% (6.0)	100% (6.0)
nodes	90	240	300	1773	1773	2735
time(sec)	1	4	2	30	34	56
(2) Three Goal						
% Solved	100%	100%	100%	100% (8.2)	100% (8.2)	100% (8.2)
nodes	120	810	990	6924	13842	20677
time(sec)	2	15	8	146	290	402
(3) Four Goal						
% Solved	100%	100%	100%	100% (10.3)	100% (10.3)	100% (10.3)
nodes	150	2340	2533	290	38456	127237
time(sec)	3	41	21	32	916	2967

Table 1: *Performance statistics in $(\theta_2 D^m S^1)$ and Logistics Transportation Domain (Average solution length is shown in parentheses next to %Solved for the logistics domain only)*

case that repaired the failure was also stored in the library. In order that the new case differed from the old only in the quality of the advice that it contained and not in the amount of information, prior to storing the derivation trace it was stripped of the decisions relating to the extra goal. The new case was then stored in the library, and the old case was annotated with the failure reason, as well as a pointer to the new case. This meant that $(n + 1)$-goal problems were always solved by replaying n-goal cases.

4.2 Experimental Results

The results of the experiments are shown in Tables 1 and 2. Each table entry represents cumulative results obtained from the sequence of 30 problems corresponding to one phase of the experiment. The first row of Table 1 shows the percentage of problems correctly solved within the time limit (550 seconds). The average solution length is shown in parentheses for the logistics domain (solution length was omitted in $(\theta_2 D^m S^1)$ since all of the problems generated within a phase have the same solution length). The second and third rows of Table 1 contain respectively the total number of search nodes visited for all of the 30 test problems, and the total CPU time (including case retrieval time). DERSNLP in learning mode was able to solve as many of the multi-goal problems as in the other two modes and did so in substantially less time. Case retrieval based on case failure resulted in performance improvements which increased with problem size (See Table 1). Comparable improvements were not found when retrieval was based on the static similarity metric alone. This should not be surprising since cases were retrieved that had experienced at least one earlier failure. This meant that testing was done on cases that had some likelihood of failing if retrieval was based on the static metric.

In the simple artificial domain, $(\theta_2 D^m S^1)$, cases retrieved on the basis of the static similarity metric alone always resulted in a replay failure. Moreover, cases that were retrieved on the basis of failure conditions were always successfully replayed. In the more complex transportation domain, the likelihood of replay success or failure depends on the parameters that govern the problem generation, including the number of cities and the number of packages to be transported. Results from this domain, however, were even more compelling. Retrieval based on failure analysis showed substantial improvements over

Phase	($\theta_2 D^m S^1$)		Logistics	
	Learning	**Static**	**Learning**	**Static**
Two Goal				
% Succ	100%	0%	53%	53%
% Der	60%	0%	48%	48%
% Rep	100%	0%	85%	85%
Three Goal				
% Succ	100%	0%	80%	47%
% Der	70%	0%	63%	50%
% Rep	100%	0%	89%	72%
Four Goal				
% Succ	100%	0%	100%	70%
% Der	94%	0%	79%	62%
% Rep	100%	0%	100%	81%

Table 2: *Measures of Effectiveness of Replay*

retrieval based on the static metric alone. Notice, in particular, the last phase, which included testing on four-goal problems. On these problems, the planner showed up to two orders of magnitude improvement in CPU time in learning mode.

Table 2 records three different measures which reflect the effectiveness of replay. The first is the percentage of *successful* replay. Recall that replay of a trace is considered here to be successful if the skeletal plan is further refined to reach a solution to the new problem. The results on the percentage of successful replay point to the greater efficiency of replay in learning mode. In the ($\theta_2 D^m S^1$) domain, replay was entirely successful in this mode. In the transportation domain, retrieval based on failure did not always result in successful replay, but did so more often than in static mode.

The greater effectiveness of replay in learning mode is also indicated by the two other measures contained in the subsequent two rows of Table 2. These are respectively, the percentage of plan-refinements on the final derivation path that were formed through guidance from replay, and the percentage of the total number of plans created through replay that remained in the final derivation path. The case-based planner in learning mode showed as much or greater improvements according to these measures, demonstrating the relative effectiveness of guiding retrieval through a learning component based on replay failures.

4.3 Summary

The results from our experiments in both the artificial domain, ($\theta_2 D^m S^1$), and the more complex logistics transportation domain, demonstrate that replay performance can be significantly improved by taking into account the failures that are encountered during replay. Performance improvements were greatly enhances when the planner was able to avoid case failures that were predicted on the basis of previously experienced failures. These results validate our choice of definition of case failure, as well as our method of constructing case failure reasons. The results indicate that the case failure annotations were successful in predicting future case failure. Moreover, when case failure was predicted, the failure annotations enabled the retrieval of alternative cases which provided substantially better performance improvements through replay. Our results demonstrate that when failure information is available, the improvements gained by utilizing this information are such that they more than offset the

added cost entailed in testing failure reasons and retrieving on the basis of failure information.

5 Related Work and Discussion

The current work complements and extends earlier treatments of case retrieval [10, 16, 17]. Replay failures are explained and used to avoid the retrieval of a case in situations where replay will mislead the planner. CHEF [4] learns to avoid execution-time failures by simulating and analyzing plans derived by reusing old cases. In contrast, our approach attempts to improve planning efficiency by concentrating on search failures encountered in plan generation. We integrate replay with techniques adopted from the planning framework provided by SNLP+EBL [11]. This framework includes methods for constructing conditions for predicting analytical failures in its search space.

In this paper, we explain and learn only from analytical failures. However, our approach may be also be extended to explain failures related to plan quality [15]. If plans are rejected by an independent plan evaluation which identifies a combination of plan constraints which are to be blamed for the quality of the plan, then explanations may be constructed that identify those constraints. This would allow the retrieval mechanism to learn to avoid cases which result in similar bad plans, and thereby improve the quality of the solutions, as well as the planning performance.

Although EBL techniques have been previously used to learn from problem-solving failures [14], the goal of EBL has been to construct generalized control rules that can be applied to each new planning decision. Here we use the same analysis to generate case-specific rules for case retrieval. Rather than learn from all failures, we only concentrate on learning from failures that result in having to backtrack over the replayed portion of the search path. As learned information is used as a censor on retrieval rather than as a pruning rule, soundness and completeness of the EBL framework may not be as critical. Furthermore, keeping censors on specific cases avoids the utility problem commonly suffered by EBL systems. See [13, 8, 3] for discussions on the issue of the relative tradeoffs offered by case-based and EBL methods in learning to improve planning performance .

6 Conclusion

In this paper, we described a framework for a case-based planning system that is able to exploit case failure to improve case retrieval. A case is considered to fail in a new problem context when the skeletal plan produced through replay can not be extended by further planning effort to reach a solution. EBL techniques are employed to explain the failure of the subtree directly beneath the skeletal plan. This failure explanation is then propagated up the tree to the root, where it is in terms of the problem specification and is used to explain the case failure and to predict similar failures that may occur in the future. Failing cases are annotated with the reasons for failure which direct the retrieval process on to an alternative case which avoids the failure. Our results demonstrate the effectiveness of this approach.

References

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

[2] J. Carbonell. Derivational analogy: A theory of reconstructive problem solving and expertise acquisition. In Ryszard Michalski, Jaime Carbonell, and Tom M. Mitchell, editors, *Machine Learning: an Artificial Intelligence approach: Volume 2*. Morgan-Kaufman, 1986.

[3] A. Francis and S. Ram. A comparative utility analysis of case-based reasoning and control-rule learning systems. In *Proceedings of the Workshop on Case-Based Reasoning*. AAAI, 1994. Seattle, Washington.

[4] K. Hammond. Chef: A model of case-based planning. In *Proceedings AAAI-86*, pages 261--271. AAAI, 1986. Philadelphia, Pennsylvania.

[5] K. Hammond. Explaining and repairing plans that fail. *Artificial Intelligence*, 45:173--228, 1990.

[6] L. Ihrig and S. Kambhampati. Derivation replay for partial-order planning. In *Proceedings AAAI-94*, 1994.

[7] L Ihrig and S. Kambhampati. On the relative utility of plan-space vs state-space planning in a case-based framework. Technical Report 94-006, Department of Computer Science and Engineering, 1994. Arizona State University.

[8] S. Kambhampati. Utility tradeoffs in incremental modification and reuse of plans. In *Proc. AAAI Spring Symp. on Computational Considerations in Supporting Incremental Modification and Reuse*, 1992.

[9] S. Kambhampati. Exploiting causal structure to control retrieval and refitting during plan reuse. *Computational Intelligence Journal*, 10(2), 1994.

[10] S. Kambhampati and J. A. Hendler. A validation structure based theory of plan modification and reuse. *Artificial Intelligence*, 55:193--258, 1992.

[11] S. Katukam and S. Kambhampati. Learning ebl-based search control rules for partial order planning. In *Proceedings AAAI-94*, 1994.

[12] J. Koehler. Avoiding pitfalls in case-based planning. In *Proceedings of the 2nd Intl. Conf. on AI Planning Systems*, pages 104--109, 1994.

[13] S. Minton. Issues in the design of operator composition systems. In *Proceedings of the International conference on Machine Learning*, 1990.

[14] J. Mostow and N. Bhatnagar. Failsafe: A floor planner that uses ebg to learn from its failures. In *Proceedings IJCAI-87*, pages 249--255, 1987.

[15] A. Perez and J. Carbonell. Control knowledge to improve plan quality. In *Proceedings of the 2nd Intl. Conf. on AI Planning Systems*, pages 323--328, 1994.

[16] M. Veloso. *Learning by analogical reasoning in general problem solving*. PhD thesis, Carnegie-Mellon University, 1992.

[17] M. Veloso and J. Carbonell. Derivational analogy in prodigy: Automating case acquisition, storage and utilization. In *Machine Learning*, pages 249--278, 1993.