LEARNING BY ANALOGICAL REPLAY IN PRODIGY: FIRST RESULTS

Manuela M. VELOSO Jaime G. CARBONELL
mmv@cs.cmu.edu jgc@cs.cmu.edu
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

Robust reasoning requires learning from problem solving episodes. Past experience must be compiled to provide adaptation to new contingencies and intelligent modification of solutions to past problems. This paper presents a comprehensive computational model of analogical reasoning that transitions smoothly between case replay, case adaptation, and general problem solving, exploiting and modifying past experience when available and resorting to general problem-solving methods when required. Learning occurs by accumulation and reuse of cases (problem solving episodes), especially in situations that required extensive problem solving, and by tuning the indexing structure of the memory model to retrieve progressively more appropriate cases. The derivational replay mechanism is briefly discussed, and extensive results of the first full implementation of the automatic generation of cases and the replay mechanism are presented. These results show up to a 20-fold performance improvement in a simple transportation domain for structurally-similar problems, and smaller improvements when a rudimentary similarity metric is used for problems that share partial structure in a process-job planning domain and in an extended version of the STRIPS robot domain.

Keywords

Analogy, general-purpose problem solving, learning.

1 Introduction: Motivation and Substrate

Derivational analogy is a general form of case-based reconstructive reasoning that replays and modifies past problem solving traces to solve problems more directly in new but similar situations [Carbonell, 1986]. While generating a solution to a problem from a given domain theory, the problem solver accesses a large amount of knowledge that is not explicitly present in the final solution returned. One can view the problem solving process as a troubled (messy) search for a solution where different alternatives are generated and explored, some failing and others succeeding. Local and global reasons for decisions are recorded incrementally during the search process. A final solution represents a sequence of operations that correspond only to a particular successful search path. Transformational analogy [Carbonell, 1983] and most case-based reasoning systems (as summarized in [Riesbeck and Schank, 1989]) replay past solutions by modifying, *tweaking* the retrieved past solution. Derivational analogy, on the other hand, aims at capturing that extra amount of knowledge present at search time, by compiling the justifications at each decision point and annotating these at different steps of the successful path. When replaying a solution, the derivational analogy engine reconstructs the reasoning process underlying the past solution. Justifications are tested to determine whether modifications are needed, and when they are needed; justifications provide constraints on possible alternative search paths. In the derivational analogy framework, the compilation of the justifications at search time is done naturally without extra effort, as that information is directly accessible by the problem solver. In general, the justifications are valid for the individual problem. No costly attempt is made to infer generalized behavior from a unique problem solving trace. Generalization occurs incrementally as the problem solver accumulates experience in solving similar problems when they occur. In this way we differ from the eager-learning approach of EBL and chunking [Laird *et al.*, 1986].

This work is done in the context of the nonlinear problem solver of the PRODIGY research project. The PRODIGY integrated intelligent architecture was designed both as a unified testbed for different learning methods and as a general architecture to solve interesting problems in complex task domains. The problem solver is an advanced operator-based planner that includes a simple reason-maintenance system and allows operators to have conditional effects. All of PRODIGY's learning modules share the same general problem solver and the same domain representation language. Learning methods acquire domain and problem specific control knowledge.

A domain is specified as a set of operators, inference rules, and control rules. Additionally the entities of the domain are organized in a type hierarchy. Each operator (or inference rule) has a precondition expression that must be satisfied before the operator can be applied, and an effects-list that describes how the application of the operator changes the world. Search control in PRODIGY allows the problem solver to represent and learn control information about the various problem solving decisions. A problem consists of an initial state and a goal expression. To solve a problem, PRODIGY must find a sequence of operators that, if applied to the initial state, produces a final state satisfying the goal expression.

The derivational analogy work in PRODIGY takes place in the context of PRODIGY's **nonlinear** problem solver [Veloso, 1989, Veloso *et al.*, 1990 forthcoming]. The system is called NOLIMIT, standing for Nonlinear problem solver using casual commitment. The basic search procedure is, as in the linear planner [Minton *et al.*, 1989], means-ends analysis in backward chaining mode. Basically, given a goal literal not true in the current world, the planner selects one operator that adds (in case of a positive goal, or deletes, in case of a negative goal) that goal to the world. We say that this operator is *relevant* to the given goal. If the preconditions of the chosen operator are true, the operator can be *applied.* If this is not the case, then the preconditions that are not true in the *state*, become *subgoals*, i.e., new goals to be achieved. The cycle repeats until all the conjuncts from the goal expression are true in the world. NOLIMIT's nonlinear character stems from working with a set of goals in this cycle, as opposed to the top goal in a goal stack. The skeleton of NOLIMIT's search algorithm is shown in Figure 1. Dynamic goal selection enables NOLIMIT to interleave plans, exploiting common subgoals and addressing issues of resource contention.

```
1. Check if the goal statement is true in the current state, or there is a reason to suspend the current search
   path.
       If yes, then either, show the formulated plan, backtrack, or take appropriate action.
2. Compute the set of pending goals $\mathcal{G}$, and the set of possible applicable operators $\mathcal{A}$.
3. Choose a goal $G$ from $\mathcal{G}$ or select an operator $A$ from $\mathcal{A}$ that is directly applicable.
4. If $G$ has been chosen, then
       • expand goal $G$, i.e., get the set $\mathcal{O}$ of relevant instantiated operators for the goal $G$,
       • choose an operator $O$ from $\mathcal{O}$,
       • go to step 1.
5. If an operator $A$ has been selected as directly applicable, then
       • apply $A$,
       • go to step 1.
```

Figure 1: A Skeleton of NOLIMIT's Search Algorithm

The algorithm in Figure 1 describes the basic cycle of NOLIMIT as a *mental* planner. *Applying* an operator means *executing* it in the *internal* world of the problem solver, which we refer to, simply by *world* or *state*. Step 1 of the algorithm checks whether the top level goal statement is true in the current state. If this is the case, then we have reached a solution to the problem. Step 2 computes the set of pending goals. A goal is *pending*, iff it is a precondition of a *chosen* operator that is not true in the state. The *subgoaling* branch of the algorithm continues, by choosing, at step 3, a goal from the set of pending goals. The problem solver *expands* this goal, by getting the set of *instantiated operators* that are relevant to it (step 4). NOLIMIT now *commits* to a relevant operator. This means that the goal just being expanded is to be achieved by applying this *chosen* operator. Step 2 further checks for the *applicable* chosen operators. An operator is *applicable*, iff all its preconditions are true in the state. Note that we can apply several operators in sequence by repeatedly choosing step 5 in case there are multiple applicable operators. Such situations occur when fulfilling a subgoal satisfies the preconditions of more than one pending operator. The *applying* branch continues by choosing to apply this operator at step 3, and applying it at step 5, by updating the state. A search path is therefore defined by the follwoing regular expression:

(*goal chosen-operator applied-operator**)*.

PRODIGY's general problem solver is combined with several learning modules. The operator-based problem solver produces a complete search tree, encapsulating all decisions – right ones and wrong ones – as well as the final solution. This information is used by each learning component in different ways: to extract control rules via EBL [Minton, 1988], to build derivational traces (cases) by the derivational analogy engine [Veloso and Carbonell, 1990], to analyze key decisions by the Apprentice knowledge acquisition interface [Joseph, 1989], or to formulate focused experiments [Carbonell and Gil, 1990]. The axiomatized domain knowledge is also used to learn abstraction layers [Knoblock, 1990], and statically generate generate control rules [Etzioni, 1990].

The remainder of this paper is organized as follows. Section 2 discusses the automatic case generation, as fully annotated derivational traces. Section 3 presents the replay mechanism for case utilization, illustrated with results obtained by derivational replay in three different domains. In section 4 we briefly describe the case memory we are developing to address dynamically the indexation and organization of cases. Finally section 5 draws conclusions on this work.

2 The Derivational Trace: Case Generation

The ability to replay previous solutions using the derivational analogy method requires that the problem solver be able to introspect into its internal decision cycle, recording the justifications for each decision during its extensive search process. These justifications augment the solution trace and are used to guide the future reconstruction of the solution for subsequent problem solving situations where equivalent justifications hold true.

Derivational analogy is a *reconstructive* method by which *lines of reasoning* are transferred and adapted to the new problem [Carbonell, 1986]. It is, therefore, necessary to extract and store these lines of reasoning from the search process in an explicit way. The goal is to identify and capture the reasons for the decisions taken by the problem solver at the different choice points encountered while searching for a solution. We identify the following types of choice points [Veloso, 1989]:

- What *goal* to subgoal, choosing it from the set of pending goals.
- What *operator* to choose in pursuit of the particular goal selected.
- What *bindings* to choose to instantiate the selected operator.
- Whether to *apply* an applicable operator or continue *subgoaling* on a pending goal.
- Whether the search path being explored should be *suspended*, continued, or abandoned.
- Upon failure, which *past choice point* to backtrack to, or which *suspended path* to reconsider for further search.

Justifications at these choice points may point to user-given guidance, to preprogrammed control knowledge, to automatically-learned control rules responsible for decisions taken, or to previous cases used as guidance (more than one case can be used to solve a complete problem). They also represent links among the different steps and their related generators, in particular capturing the subgoaling structure. We record failed alternatives (explored earlier) and the cause of their failure. Note that "cause of failure" here refers to the reason why the search path starting at that alternative failed. It does not necessarily mean that the failed alternative is directly responsible for the failure of the global search path. It may be an indirect relationship, but this is the least costly attribution to determine. The current reasons for failure in NoLimit follow from to PRODIGY's search philosophy [Minton *et al.*, 1989]:

No Relevant Operators - When NoLimit reaches an *unachievable* goal, i.e. a goal that does not have any relevant operator that adds it as one of its effects, given the current state and control rules.

State Loop - If the application of an operator leads into a previously visited state, then NoLimit abandons this path, as a redundant sequence of operators was applied.

Goal Loop - When NoLimit encounters an unmatched goal that was already previoulsy posted in the search path (i.e. when a pending goal becomes its own subgoal).

NoLimit abandons a search path either due to any of these failures, or at a situation that is heuristically declared not promising (e.g. a search path that is too long).

A search path follows the sequence of decisions presented in the algorithm of Figure 1. Hence, a step of the search path can only be either a goal choice, an operator choice, or the application of an operator. To generate a case from a search tree episode, we take the successful solution path annotated with both justifications for the successful decisions taken, and record of the remaining alternatives that were not explored or that were abandoned and their corresponding reasons. We show below the different justifications annotated at a goal, operator, and applied operator decision nodes.

2.1 Justifications at the Different Decision Nodes

According to the search algorithm presented in Figure 1, a goal is selected from the set of pending goals. NoLimit may either apply an operator whose preconditions are satisfied (if any), i.e. its left hand side is true in the current state, or continue subgoaling in an unmatched precondition of a different chosen operator. Figure 2 (a) shows the skeleton of a goal decision node.

```
Goal Node                    Applied Operator Node          Chosen Operator Node
    :step                        :step                          :step
    :sibling-goals               :sibling-goals                 :sibling-relevant-ops
    :sibling-applicable-ops      :sibling-applicable-ops        :why-this-operator
    :why-subgoal                 :why-apply                     :relevant-to
    :why-this-goal               :why-this-operator
    :precond-of
       (a)                           (b)                            (c)
```

Figure 2: Justification Record Structure: (a) At a Goal Decision Node; (b) At a Chosen Operator Decision Node; (c) At an Applied Operator Decision Node

The different slots capture the context in which the decision is taken and the reasons that support the choice:

Step shows the goal selected at this node.

Sibling-goals enumerates the set of pending goals, i.e. goals that arose from unmatched preconditions of operators chosen as relevant to produce previous goals; the goal at this node was selected from this set; the other goals in this set were therefore sibling goals to work on. NOLIMIT annotates the reason why these alternatives were not pursued further according to its search experience (either not tried, or abandon as described above).

Sibling-applicable-ops shows the relevant operators that could have been applied instead of subgoaling on this goal.

Why-Subgoal presents the reason why NOLIMIT decided to subgoal instead of applying an operator (in case there was one).

Why-This-Goal explains why this particular goal was selected from the set of alternative sibling goals.

Precond-of captures the subgoaling structure; it refers to the operator(s) that gave rise to this goal as an unmatched precondition.

The reasons annotated at the slots *why-subgoal* and *why-this-goal* can range from arbitrary choices to a specific control rule or guiding case that dictated the selection. These reasons are tested at replay time and are interpretable by NOLIMIT.

An operator may be applied if previously selected as relevant for a pending goal and all its preconditions are satisfied in the current state. Figure 2 (b) shows the skeleton of an applied operator decision node. The different slots have an equivalent semantics to the ones at a goal decision node.

An operator is chosen because it is relevant to a pending goal. Figure 2 (c) shows the skeleton of a chosen operator decision node. Alternative instantiated relevant operators are listed in the slot *sibling-relevant-ops*. The slot *why-this-operator* captures reasons that supported the choice of both the current operator and instantiated bindings. The subgoaling link is kept in the slot *relevant-to* that points at the pending goal that unifies with an effect of this operator.

To illustrate the automatic generation of an annotated case, we now present an example.

2.2 Example

Consider the set of operators in Figure 3 that define the *ONE-WAY-ROCKET* domain.

```
(LOAD-ROCKET                      (UNLOAD-ROCKET                    (MOVE-ROCKET
 (params ((<obj> OBJECT)           (params ((<obj> OBJECT)           (params nil)
          (<loc> LOCATION))                 (<loc> LOCATION))         (preconds
 (preconds                         (preconds                           (at ROCKET locA))
  (and                              (and                              (effects
   (at <obj> <loc>)                  (inside <obj> ROCKET)             (add (at ROCKET locB))
   (at ROCKET <loc>)))               (at ROCKET <loc>)))               (del (at ROCKET locA))))
 (effects                          (effects
  (add (inside <obj> ROCKET))       (add (at <obj> <loc>))
  (del (at <obj> <loc>))))          (del (inside <obj> ROCKET))))
```

Figure 3: The *ONE-WAY-ROCKET* Domain

Variables in the operators are represented by framing their name with the signs "<" and ">". In this domain, there are variable objects and locations, and one specific constant ROCKET. An object can be loaded into the ROCKET at any location by applying the operator LOAD-ROCKET. Similarly, an object can be unloaded from the ROCKET at any location by using the operator UNLOAD-ROCKET. The operator MOVE-ROCKET shows that the ROCKET can move only from a specific location *locA* to a specific location *locB*. Although NoLimit will solve much more complex and general versions of this problem, the present minimal form suffices to illustrate the derivational analogy procedure in the context of nonlinear planning.

Suppose we want NoLimit to solve the problem of moving two given objects *obj*1 and *obj*2 from the location *locA* to the location *locB* as expressed in Figure 4.

```
Initial State:                    Goal Statement:
        (at obj1 locA)                 (and (at obj1 locB)
        (at obj2 locA)                      (at obj2 locB))
        (at ROCKET locA)
```

Figure 4: A Problem in the *ONE-WAY-ROCKET* World

Without any control knowledge the problem solver searches for the goal ordering that enables the problem to be solved. Accomplishing either goal individually, as a linear planner would do, inhibits the accomplishment of the other goal. A precondition of the operator LOAD-ROCKET cannot be achieved when pursuing the second goal (after completing the first goal), because the ROCKET cannot be moved back to the second object's initial position (i.e. *locA*). So interleaving of goals and subgoals at different levels of the search is needed to find a solution. An example of a solution to this problem is the following plan: `(LOAD-ROCKET obj1 locA), (LOAD-ROCKET obj2 locA) (MOVE-ROCKET), (UNLOAD-ROCKET obj1 locB), (UNLOAD-ROCKET obj2 locB).`

NoLimit solves this problem, because it switches attention to the conjunctive goal *(at obj2 locB)* before completing the first conjunct *(at obj1 locB)*. This is shown in Figure 5 by noting that; after the plan step 1 where the operator (LOAD-ROCKET obj1 locA) is applied as relevant to a subgoal of the top-level goal *(at obj1 locB)*, NoLimit suspends processing and changes its focus of attention to the other top-level goal and applies, at plan step 2, the operator (LOAD-ROCKET obj2 locA) which is relevant to a subgoal of the goal *(at obj2 locB)*. In fact NoLimit explores the space of possible attention foci and only after backtracking does it find the correct goal interleaving. The idea is to learn next time from its earlier exploration and reduce search dramatically.

While solving this problem, NoLimit automatically annotates the decisions taken with justifications that reflect its experience while searching for the solution. As an example, suppose that the correct decision of choosing to work on the goal *(inside obj1 ROCKET)* was taken after having failed when working first on *(at ROCKET locB)*. The decision node stored for the goal *(inside obj1 ROCKET)* is annotated with sibling goal failure as illustrated in Figure 6. *(at ROCKET locB)* was a sibling goal that was abandoned because NoLimit encountered an unachievable predicate pursuing that search path, namely the goal *(at ROCKET locA)*.

The problem and the generated annotated solution become a *case* in memory. The case corresponds

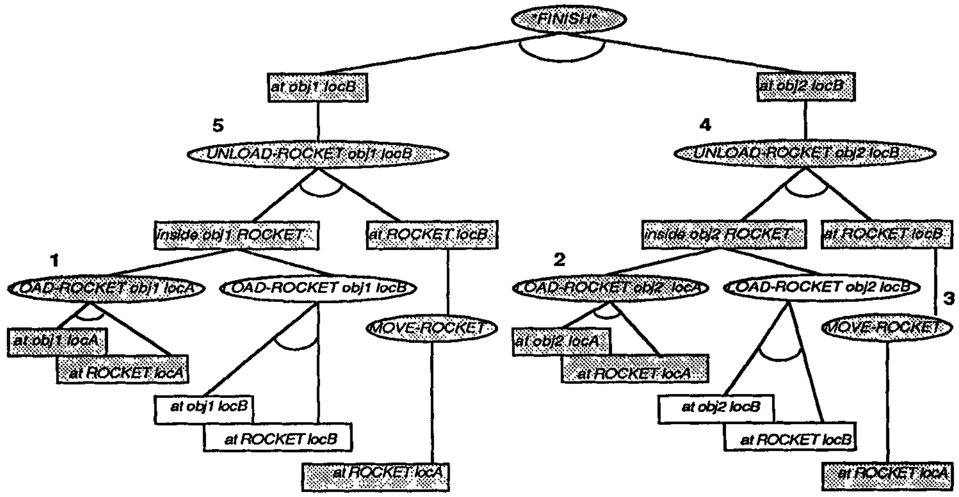

Figure 5: The Complete Conceptual Tree for a Successful Solution Path. The numbers at the nodes show the execution order of the plan steps. Shaded nodes correspond to the choices to which the problem solver committed.

```
Frame of class goal-decision-node
 :step  (inside obj1 ROCKET)
 :sibling-goals (((inside obj2 ROCKET) not-tried)
                 ((at ROCKET locB) (:no-relevant-ops (at ROCKET locA))))
 :sibling-applicable-ops NIL
 :why-subgoal NIL
 :why-this-goal NIL
 :precond-of  (UNLOAD-ROCKET obj1 locB)
 step of next-decision-node  (LOAD-ROCKET obj1 locA)
```

Figure 6: Saving a Goal Decision Node with its Justifications

to the search tree compacted into the successful path annotated with the justifications that resulted in the sequence of correct decisions that lead into a solution to the problem. In essence, a case is a sequence of decision nodes such as the one illustrated in Figure 6.

3 The Derivational Replay: Case Utilization

When solving new problems similar to past cases, one can envision two approaches for derivational replay:

A. *The satisficing approach* - Minimize planning effort by solving the problem as directly as possible, recycling as much of the old solution as permitted by the justifications.

B. *The optimizing approach* - Maximize plan quality by expanding the search to consider alternatives of arbitrary decisions and to re-explore failed paths if their causes for failure are not present in the new situation.

At present we have implemented in full the satisficing approach, although work on establishing workable optimizing criteria may make the optimizing alternative viable (so long as the planner is willing to invest the extra time required). Satisficing also accords with observations of human planning efficiency and human planning errors.

In the satisficing paradigm, the system is fully guided by its past experience. The syntactic applicability of an operator is always checked by simply testing whether its left hand side matches the current state. Semantic applicability is checked by determining whether the justifications hold (i.e. whether there is still a *reason* to apply this operator). In case the choice remains valid in the current problem state, it is merely copied, and in case it is not valid the system has three alternatives:

1. Replan at the particular failed choice, e.g. re-establishing the current subgoal by other means (or to find an equivalent operator, or equivalent variable bindings) substituting the new choice for the old one in the solution sequence,
2. Re-establish the failed condition by adding it as a prioritized goal in the planning, and if achieved simply insert the extra steps into the solution sequence, or
3. Attempt an experiment to perform the partially unjustified action anyway; if success is achieved the system refines its knowledge according to the experiment. For instance, if the justification for stacking blocks into a tower required objects with flat top and bottom surfaces, and there were none about (so the first fix does not work) nor is there a way to make surfaces flat (so the second fix also fails), the robot could attempt to forge ahead. If the objects were spherical it would fail, but if they were interlocking LEGO^{TM} pieces, it would learn that these were just as good if not better than rectangular blocks for the purpose of stacking objects to build tall towers. Thus, the justification could be generalized for future reference.

In the first case (substitution), deviations from the retrieved solution are minimized by returning to the solution path after making the most localized substitution possible.

The second case occurs, for example, when the assumptions for the applicability of an operator fail. The system then tries to overcome the failed condition, and if it succeeds, it returns to the exact point in the derivation to proceed as if nothing had gone wrong earlier. Failures however, can be serious. Consider as an example, applying to the context of matrix calculus, some previously solved problems on scalars that rely on commutativity of multiplication. Facing the failure to apply a commutation operator in the matrix context, the system may try to overcome this difficulty by checking whether there is a way of having two matrices commute. In general this fails; the case must be abandoned; and a totally different approach is required.

The experimentation case enables uncertain attempts to perform the same action with partially unjustified conditions, or can digress from the problem at hand to perform systematic relaxation of justifications, and establish whether a more general (more permissive) set of conditions suffices for the instance domain. Then, returning to the problem at hand, it may find possible substitutions or perhaps even re-establishments of these looser conditions via steps 1 or 2 above.

The fact that these different situations can be identified by the problem solver when trying to replay a past case is the motivation and support for our proposed memory model. Memory organization is in a closely coupled dynamic relationship with the problem solving engine.

3.1 The One-Way-Rocket Domain, An Example

Let us return to the *ONE-WAY-ROCKET* problem shown in section 2.2 to illustrate the derivational replay process. We show the results obtained in the problems of moving three objects and four objects from *locA* into *locB* in Tables 1 and 2. Each row of the tables refers to one new problem, namely the two- (2objs), three- (3objs), and four-object (4objs) problems. We show the number of search steps in the final solution, the average running time of NOLIMIT without analogy (blind search), and using analogical guidance from one of the other cases.

New Problem	Blind Search (s)	Following Case 2objs (s)	Improvement	Following Case 3objs (s)	Improvement	Following Case 4objs (s)	Improvement
2objs (18 steps)	18	8	2.3x	8	2.3x	8	2.3x
3objs (24 steps)	59	31	1.9x	13	4.5x	13	4.5x
4objs (30 steps)	470	110	4.3x	58	8.1x	23	20.4x

Table 1: Replaying Direct Solution

Table 1 shows the results obtained when the justifications are not fully tested. The solution is simply replayed whenever the same step is possible (but not necessarily desirable). For example, if using the two-object case as guidance to the three- (or four-) object problem, after two objects are loaded into the rocket, the step of moving the rocket is tested and replayed because it is also a syntatically possible step. This is not the right step to take, as there are more objects to load into the rocket in the new extended cases. NOLIMIT must backtrack across previously replayed steps, namely across the step of moving the rocket.

On the other hand, in Table 2, we show the results obtained from further testing the justifications before applying the step. In this case, the failure justification for moving the rocket - "no-relevant-ops" - is tested and this step is not replayed until all the objects are loaded into the rocket. Testing justifications shows maximal improvement in performance when the case and the new problem differ substantially (two-objects and four-objects respectively).

From these results we also note that it is better to approach a complicated problem, like the four-object problem, by first generating automatically a reduced problem [Polya, 1945], such as the two-object problem, then gain insight solving the reduced problem from scratch (i.e. build a reference case), and

New Problem	Blind Search (s)	Following Case 2objs (s)	Impro-vement	Following Case 3objs (s)	Impro-vement	Following Case 4objs (s)	Impro-vement
2objs (18 steps)	18	8	2.3x	8	2.3x	8	2.3x
3objs (24 steps)	59	19	**3.1x**	13	4.5x	13	4.5x
4objs (30 steps)	470	30	**15.2x**	30	**15.2x**	23	20.4x

Table 2: Testing the Justifications: no-relevant-ops

finally solve the original four-object problem by analogy with the simpler problem. The running time of the last two steps in this process is significantly less than trying to solve the extended problem directly, without analog for guidance. (18 seconds + 30 seconds = 48 seconds – see Table 2 – for solving the two-objects from scratch + derivational replay to the four-object case, versus 470 seconds for solving the four-object case from scratch.)

We note that whereas we have implemented the nonlinear problem solver, the case formation module, and the analogical replay engine, we have not yet addressed the equally interesting problem of automated generation of simpler problems for the purpose of gaining relevant experience. That is, PRODIGY will **exploit** successfully the presence of simpler problems via derivational analogy, but cannot **create** them as yet.

3.2 Process-Job Planning and extended-STRIPS Domains, More Examples

We also ran two other experiments to test empirically the benefits of the replay mechanism. We ran NoLIMIT without analogy in a set of problems in the process-job planning and in the extended-STRIPS domains [1]. We accumulated a library of cases, i.e. annotated derivational solution traces. We then ran again the set of problems using the case library. In particular, if the set of cases is C, and the new problem is P, corresponding to case C_P, then we searched for a similar case in the set $C - C_P$. We used a rudimentary fixed similarity metric that matched the goal predicates, allowed substitutions for elements of the same type, and did not consider any relevant correlations. Figures 7 and 8 show the results for these

[1]This set is a sampled subset of the original set used by [Minton, 1988].

two domains. We plotted the average cumulative number of nodes searched.

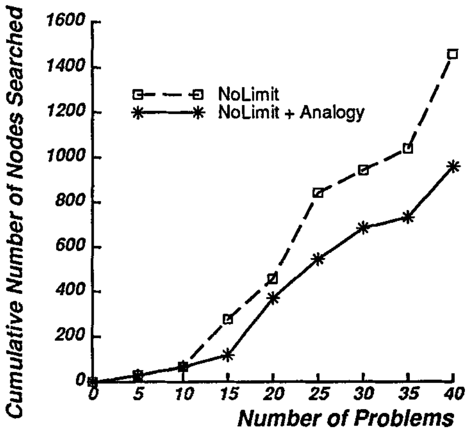

Figure 7: Comparison in the Process-Job Planning and Scheduling Domain

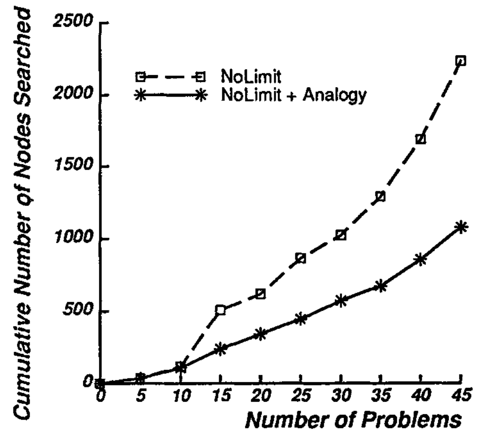

Figure 8: Comparison in the Extended-STRIPS Domain

We note from the results that analogy showed an improvement over basic blind search: a factor of 1.5 fold up for the process-job planning and scheduling domain and 2.0 fold for the extended-STRIPS domain. We noticed few individual problems in the case library that provided bad guidance. In general, however, the simple similarity metric lead to acceptable results. We expect to achieve even better results when a more sophisticated metric is used through dynamic memory organization, as discussed below. We also expect faster indexing than the current linear comparison search, and therefore expect higher performance improvements after completing the implementation of the more sophisticated memory model.

4 Towards an Integrated Memory Model

We view the ultimate desired behavior of the analogical reasoning system to emerge from the interaction of two functional modules, namely the *problem solver* and *the memory manager.* We call the memory manager, SMART, for Storage in Memory and Adaptive Retrieval over Time. NOLIMIT and SMART communicate as shown in Figure 9, where W_i is the initial world, G is the goal to be achieved, W_f is the final world, *Analogs* are the retrieved candidate cases, and *Feedback* represents both the new solved problem and information about the utility of the candidate cases in reaching a solution.

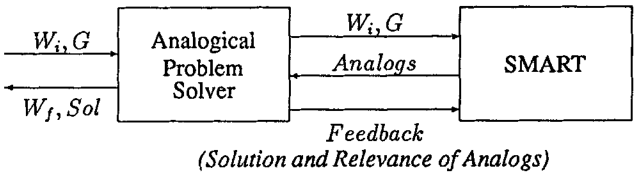
W_i, G
W_f, Sol
Analogical Problem Solver
W_i, G
Analogs
SMART
Feedback
(Solution and Relevance of Analogs)

Figure 9: Interaction of the Problem Solver and the Memory Manager

In encoding the utility of the guidance received from SMART, we foresee four different situations that can arise, as shown in Figure 10.

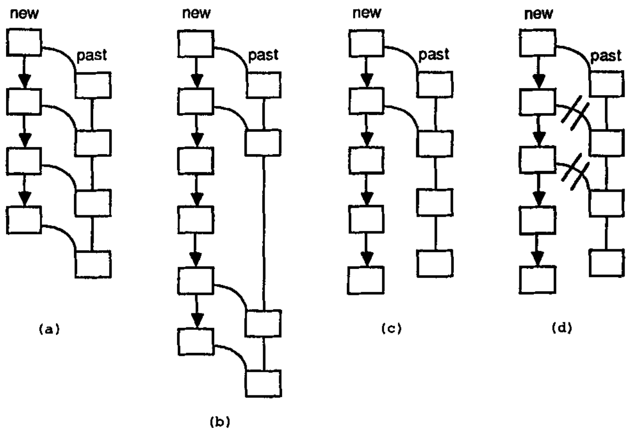

Figure 10: Four Situations to Encode the Utility of the Guidance Received: (a) Fully-sufficient: past case is fully copied; (b) Extension: past case is copied but additional steps are performed in the new case; (c) Locally-divergent: justifications do not hold and invalidate copying part of the past case; (d) Globally-divergent: extra steps are performed that undo previously copied steps.

These four situations determine the reorganization of memory when the new case is to be stored in memory. We are exploring algorithms to address each of these situations. If a case was ***fully-sufficient*** under a particular match, SMART will generalize its data structure over this match updating the indices to access these cases [Veloso and Carbonell, 1989, Veloso and Carbonell, 1990]. If the new case is an *extension* of the previous case, the conditions that lead into the adaptation and extension work are used to differentiate the indexing of the two cases. Generalization will also occur on the common parts of the case. The situations where the two cases diverge represent a currently incorrect metric of similarity or lack of knowledge. The fact that the retrieval mechanism suggested a past case as most similar to the new problem and the problem solver could not fully use the past case or even extend it, indicates either the sparsity of better cases in memory, or a similarity function that ignores an important discriminant condition. SMART will have to either specialize variables in the memory data structures due to previous overgeneralization or completely set apart the two cases in the decision structure used for retrieval. We plan to extract memory indices from the justification structure, and use them at retrieval time to more adequately prune the set of candidate analogs.

5 Conclusion

Whereas much more work lies ahead in reconstructive problem solving exploiting past experience, the results reported here demonstrate the feasibility of derivational analogy as a means to integrate general problem solving with analogical reasoning.

The research into full-fledged case-based reasoning and machine learning in the context of the PRODIGY nonlinear planner and problem solver, however, is far from complete. The full implementation of the SMART memory model, for instance, must be completed. This will enable us to scale up from the present case libraries of under a hundred individual cases to much larger case libraries numbering in the thousands of cases. We are investigating domains such as logistics and transportation planning whose inherent complexity requires large case libraries and sophisticated indexing methods.

Finally, we summarize new contributions in this work beyond the original derivational analogy framework as presented in [Carbonell, 1986]:

- Elaboration of the model of the derivational trace, i.e. identification and organization of appropriate data structures for the justifications underlying decision making in problem solving episodes. Justifications are compiled under a lazy evaluation approach.
- Development of a memory model that dynamically addresses the indexation and organization of cases, by maintaining a closely-coupled interaction with the analogical problem solver.
- Full implementation of the refined derivational analogy replay and memory model in the context of a nonlinear planner (as opposed to the original linear one). Hence the refined framework deals with a considerably larger space of decisions and with more complex planning problems.

Acknowledgments

The authors thank the whole PRODIGY research group for helpful discussions: Daniel Borrajo, Yolanda Gil, Robert Joseph, Dan Kahn, Craig Knoblock, Dan Kuokka, Steve Minton, Alicia Pérez, and Mei Wang.

This research was sponsored in part by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 4976, Amendment 20, under contract number F33615-87-C-1499, monitored by the Avionics Laboratory, Air Force Wright Aeronautical Laboratories, Aeronautical Systems Division (AFSC), United States Air Force, Wright-Patterson AFB, Ohio 45433-6543, and in part by the Office of Naval Research under contracts N00014-86-K-0678. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the US Government.

References

[Carbonell and Gil, 1990] J. G. Carbonell and Y. Gil. Learning by experimentation: The operator refinement method. In R. S. Michalski and Y. Kodratoff, editors, *Machine Learning: An Artificial Intelligence Approach, Volume III*. Morgan Kaufmann, Palo Alto, CA, 1990.

[Carbonell, 1983] J. G. Carbonell. Learning by analogy: Formulating and generalizing plans from past experience. In R. S. Michalski, J. G. Carbonell, and T. M. Mitchell, editors, *Machine Learning, An Artificial Intelligence Approach, Volume I*. Tioga Press, Palo Alto, CA, 1983.

[Carbonell, 1986] J. G. Carbonell. Derivational analogy: A theory of reconstructive problem solving and expertise acquisition. In R. S. Michalski, J. G. Carbonell, and T. M. Mitchell, editors, *Machine Learning, An Artificial Intelligence Approach, Volume II*. Morgan Kaufman, Los Altos, CA, 1986.

[Etzioni, 1990] O. Etzioni. Why Prodigy/EBL works. In *Proceedings of AAAI-90*, 1990.

[Joseph, 1989] R. L. Joseph. Graphical knowledge acquisition. In *Proceedings of the* 4^{th} *Knowledge Acquisition For Knowledge-Based Systems Workshop*, Banff, Canada, 1989.

[Knoblock, 1990] Craig A. Knoblock. Learning abstraction hierarchies for problem solving. In *Proceedings of Eighth National Conference on Artificial Intelligence*, Boston, MA, 1990.

[Laird *et al.*, 1986] J. E. Laird, P. S. Rosenbloom, and A. Newell. Chunking in SOAR: The anatomy of a general learning mechanism. *Machine Learning*, 1:11–46, 1986.

[Minton *et al.*, 1989] S. Minton, C. A. Knoblock, D. R. Kuokka, Y. Gil, R. L. Joseph, and J. G. Carbonell. PRODIGY 2.0: The manual and tutorial. Technical Report CMU-CS-89-146, School of Computer Science, Carnegie Mellon University, 1989.

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

[Polya, 1945] G. Polya. *How to Solve It*. Princeton University Press, Princeton, NJ, 1945.

[Riesbeck and Schank, 1989] C. K. Riesbeck and R. C. Schank. *Inside Case-Based Reasoning*. Lawrence Erlbaum Associates, Inc., Hillsdale, New Jersey, 1989.

[Veloso and Carbonell, 1989] M. M. Veloso and J. G. Carbonell. Learning analogies by analogy: The closed loop of memory organization and problem solving. In *Proceedings of the Second Workshop on Case-Based Reasoning*. Morgan Kaufmann, May 1989.

[Veloso and Carbonell, 1990] M. M. Veloso and J. G. Carbonell. Integrating analogy into a general problem-solving architecture. In Maria Zemankova and Zbigniew Ras, editors, *Intelligent Systems*, 1990.

[Veloso *et al.*, 1990 forthcoming] M. M. Veloso, D. Borrajo, and A. Perez. NoLimit - the nonlinear problem solver for Prodigy: User's and programmer's manual. Technical report, School of Computer Science, Carnegie Mellon University, 1990, forthcoming.

[Veloso, 1989] M. M. Veloso. Nonlinear problem solving using intelligent casual-commitment. Technical Report CMU-CS-89-210, School of Computer Science, Carnegie Mellon University, 1989.