Machine Learning, 10, 249-278 (1993)

Derivational Analogy in PRODIGY: Automating Case Acquisition, Storage, and Utilization

MANUELA M. VELOSO MMV@CS.CMU.EDU
JAIME G. CARBONELL JGC@CS.CMU.EDU
School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213

Abstract. Expertise consists of rapid selection and application of compiled experience. Robust reasoning, however, requires adaptation to new contingencies and intelligent modification of past experience. And novel or creative reasoning, by its real nature, necessitates general problem-solving abilities unconstrained by past behavior. This article presents a comprehensive computational model of analogical (case-based) 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 of new cases, 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 discussed in some detail, and extensive results of the first full implementation are presented. These results show up to a large performance improvement in a simple transportation domain for structurally similar problems, and smaller improvements when less strict similarity metrics are used for problems that share partial structure in a process-job planning domain and in an extended version of the STRIPS robot domain.

Keywords. General problem solving, derivational analogy, case-based reasoning, search and retrieval costs, replay, learning by analogy.

1. Introduction

Whereas classical AI techniques for problem solving and planning require vast amounts of search to produce viable solutions for even moderately complex problems, humans typically require much less search as they accrue and reuse experience over time in any given domain. Inspired by the ubiquitous observation, researchers in various subdisciplines of AI sought methods of encapsulating more knowledge to reduce search, ranging from expert systems, where all knowledge is laboriously hand-coded at the outset, to machine learning approaches, where incrementally accumulated experience is stored and processed for future reuse.

The machine learning approaches typically start with a general problem-solving engine and accumulate experience in the process of solving problems the hard way (via extensive search), or via demonstrations of viable solutions by an external (human) teacher. The knowledge acquired can take many forms, ranging from explicit provably correct control rules (meta rules, or chunks) (Cheng & Carbonell, 1986; DeJong & Mooney, 1986; Fikes & Nilsson, 1971; Korf, 1985; Laird et al., 1986, Minton, 1985; Minton, 1988; Mitchell et al., 1983; Mitchell et al., 1986; Newell, 1980; Shell & Carbonell, 1989) to actual instance solutions for use in analogical or case-based reasoning (CBR) (Carbonell, 1983; Carbonell,

1986; Doyle, 1984; Hammond, 1986; Kolodner, 1984; Riesbeck & Schank, 1989; Schank, 1982; Schank, 1983; Simpson, 1985; Sycara, 1987). However, they all seek to compile existing factual domain knowledge into more effective form by combining it with search control knowledge acquired through incremental practice.

Analogical reasoning in general is concerned with transferring *episodic* past experience to guide problem solving. The pure CBR approach rejects the operator-based problem-solving approach (Riesbeck & Schank, 1989). Knowledge is specified as a set of previously solved problems (cases) in the domain of interest, and solving a problem consists of retrieving a similar past case and adapting it to the new situation. To guarantee the success of the adaptation phase, CBR requires accurate similarity metrics and incurs high retrieval costs. This approach emphasizes, therefore, the organization, hierarchical indexing, and retrieval of the case memory.

We have explored machine learning techniques for compiling past experience in the PRODIGY system that integrate both knowledge and case-based reasoning for solving large-scale problems efficiently (Carbonell & Veloso, 1988; Veloso & Carbonell, 1991a). 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). When generating a solution to a novel problem from a given operator-based 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. The purpose of solving problems by analogy is to reuse past experience to guide the generation of the solution for the new problem, avoiding a completely new search effort. Transformational analogy (Carbonell, 1983) and most CBR systems (as summarized in Riesbeck & Schank, 1989) replay past solutions by modifying (*tweaking*) the retrieved final solution plan as a function of the differences recognized between the past and the current new problem. However, when the case is created during the original problem solving, local and global reasons for decisions are naturally accessible during the search process. A final solution represents a sequence of operations that corresponds only to a particular successful search path. Derivational analogy aims at capturing that extra amount of knowledge present at search time, by compiling the justifications at each decision point and annotating these at the 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 essence, derivational analogy can benefit from past successes, failures, and interactions.

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 the context of a general problem solver, we approach analogy as a closed interaction between the case memory management and the problem-solving engines. The problem solver

is seen as both the generator of new cases and the module that acknowledges or rejects the similar cases proposed by the case memory manager. We claim that no sophisticated initial measures for similarity and relevance of information are needed, as the memory manager will adapt its similarity computation based on positive and negative feedback on the utility of retrieved cases as provided by the problem solver. We show how we take advantage of the integration of general problem solving and analogical reasoning to overcome some crucial issues and difficulties in scaling up a knowledge or case-based system. The primary issues addressed in our work are:

- How the integrated analogical problem solver can generate cases automatically from problem-solving experience.
- How the analogical problem solver achieves a reduction in search effort by replaying past annotated problem solving episodes (derivational traces).
- How the analogical problem solver can help in refining the similarity metric based on the solutions(s) encountered and the utility of the suggested guiding case(s).
- How the cost of retrieving a past case can be offset by the expected search effort reduction.

Hence, our approach differs from pure CBR in the following ways:

- The substrate problem solver for the analogical engine is a rich general-purpose nonlinear means-ends analysis reasoner, as opposed to a special-purpose one, or to no reasoning engine at all beyond localized solution tweaking.
- The analogical reasoning mechanism developed is completely domain independent and applies to any domain-specific case library.
- Cases are not simply copied and tweaked, but they guide replay and can be invoked in recursive subgoal reduction, i.e., when a subgoal is reduced, memory may be asked for additional guiding cases.
- Case memory is dynamically organized in response to feedback from the problem solver on the utility of the suggested guidance. In fact, the similarity metric is adapted in response to accumulated experience.

The model presented here is a major advance beyond the original derivational analogy framework as presented in Carbonell (1986), including:

- 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.
- 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.
- Evidence of the feasibility of the extended development framework in a variety of different domains (currently extended to a 1000-case library in a complex domain).
- Development of a memory model that dynamically addresses the indexing and organization of cases, by maintaining a closely coupled interaction with the analogical problem solver.

This article is organized as follows. In section 2 we introduce the automatic case generation, as fully annotated derivational traces of problem-solving search episodes. Section 3 describes the mechanisms for case utilization. In section 4 we present the case retrieval strategy and we discuss two different similarity metrics. The replay strategy is illustrated with results obtained by derivational replay in three different domains. Section 5 summarizes our overall case memory model that we are currently developing to address dynamically the indexing and organization of cases. Finally, section 6 draws conclusions on the work and mentions future work. An appendix provides a brief introduction to the PRODIGY architecture.

In this article we use examples from an extended version of the STRIPS world (Fikes & Nilsson, 1971; Minton et al., 1989), a process-job planning and scheduling domain (Minton et al., 1989) and a simple transportation domain (Veloso, 1989). Due to the lack of space, we present in full detail only the simplest version from one of these domains (as we can reduce it to three operators) and refer the reader to the references for a complete description of the other two domains. The extended-STRIPS domain consists of a set of rooms connected through doors. A robot can move around among the rooms carrying or pushing objects along. Doors can be locked or unlocked. Keyes to the doors lay in rooms and can be picked up by the robot. In the process-job planning domain, parts are to be shaped, polished, painted, or treated in some other way. Machines are scheduled to accomplish multiple part-processing requirements in parallel.

2. The derivational trace: Case generation

Derivational analogy is a *reconstructive* method by which *lines of reasoning* are transferred and adapted to the new problem (Carbonell, 1986). 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.

In PRODIGY (Minton et al., 1989) 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 (Veloso, 1989). 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 use 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 statement. 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 & Carbonell, 1990), to analyze key decisions by a knowledge acquisition interface (Joseph, 1989), or to formulate focused experiments

(Carbonell & Gil, 1990). The axiomatized domain knowledge is also used to learn abstraction layers (Knoblock, 1991), and statically generate control rules (Etzioni, 1990a). (For additional details on the PRODIGY architecture, see the appendix).

The derivational analogy work in PRODIGY takes place in the context of PRODIGY's nonlinear problem solver (Veloso, 1989). 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 (MEA) 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.

Automatically generating a case from a problem-solving episode is immediately related to identifying and capturing the reasons for the decisions taken by the problem solver at the different choice points encountered while searching for a solution. In the nonlinear search procedure of NOLIMIT, 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.

These choice points characterize a nonlinear problem solver that uses casual commitment (Minton, 1988) in its search cycle, i.e., mentally applies operators, and considers a set, as opposed to a rigid FILO linear order (a stack), of pending goals (see appendix section A.1).

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 past cases used as guidance (more than one case can be used to solve a complete problem). They also represent links within the different choices and their related generators, in particular capturing the subgoaling structure. At choice points, we also 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 the best attribution so far. The current reasons for failure in NOLIMIT follow, according to PRODIGY's search strategy:

No Relevant Operators—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—NoLimit encounters an unmatched goal that was already previously 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 step of the search path can only be either a goal choice, an instantiated operator choice, or the application of an operator. Each step taken corresponds to a decision. To generate a case from a search tree episode, we take the successful solution path annotated with the justifications for the successful decisions taken, and with the 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 the goal, operator, and applied operator decision nodes.

2.1. Justification structures at decision nodes

In a casual-commitment search approach, justifications on decisions made arise in a natural way. Examples of these justifications are links between choices capturing the subgoaling structure, records of explored failed alternatives, and pointers to applied control guidance.

Figure 1 shows the skeleton of the different decision nodes. The different justification slots capture the context in which the decision is taken and the reasons that support the choice.

The *choice* slots show the selection done, namely, the selected goal or operator. The *sibling*-slots enumerate the alternatives to the choice made. At a goal node and an applied operator node (see Figure 1 (a) and (b)), the goals left in the current set of goals that need still to be achieved, constitute the sibling-goals annotation. For completeness, the problem solver may postpone applying an operator whose preconditions are satisfied and continue subgoaling on a still unachieved goal. These possible applicable operators are the contents of the alternative sibling-applicable-ops slot. At a chosen operator node, the sibling operators are the possible other different operators that are also relevant to the goal being expanded, i.e., the operators that, if applied, will achieve that goal. NoLimit annotates the reason why

```
Goal Node                  Applied Op Node            Chosen Op Node
 :choice                    :choice                    :choice
 :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 1. Justification record structure: (a) At a goal decision node; (b) At an applied operator decision node; (c) At a chosen operator decision node.

these alternatives were not pursued further according to its search experience (either not tried, or abandoned due to a described failure reason). The *why-* slots present the reasons (if any) the particular decision was taken. The reasons 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 the analogical problem solver. Finally the subgoaling structure is captured by the slot *precond-of* at a goal node, and the slot *relevant-to* at a chosen operator node. At reconstruction time, these slots play an important role in providing the set of relevant operators for a given goal, and the set of instantiated preconditions of an operator.

The problem and the generated annotated solution become a *case* in memory. The case corresponds to the search tree compacted into the successful path as a sequence of annotated decision nodes as presented in figure 1. According to the case utilization method (see section 3) that we present below, we note that a case is not used as a simple "macro-operator" (Fikes & Nilsson, 1971; Minton, 1985). A case is selected based on a partial match to a new problem-solving situation. Hence, as opposed to a macro-operator, a case *guides* and *does not dictate* the reconstruction process. Intermediate decisions corresponding to choices internal to each case can be bypassed or adapted, if their justifications no longer hold.

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

2.2. An example in a simple transportation domain

The simplicity of this example is for pedagogical reasons, rather than to show a situation where learning is absolutely needed in order to deal with new problems. In this section the example illustrates the automatic case-generation process, where the justifications annotated are simple and the subgoaling structure in particular can be fully presented. Later in this article, the example is briefly pursued to show the reuse of a case. Results are shown of the search reduction achieved when these simple justifications are tested and guide the reconstruction of structurally similar new problems. Clearly, the system solves much more complex and general versions of the domain.[1] The present minimal form suffices also to illustrate the casual-commitment strategy in nonlinear planning, allowing full interleaving of goals and subgoals.

Consider a generic transportation domain with three simple operators that load, unload, or move a ROCKET, as shown in figure 2.

The operator MOVE-ROCKET shows that the ROCKET can move only from a specific location *locA* to a specific location *locB*. This transforms this current general domain into a ONE-WAY-ROCKET domain. 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.

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 3.

Without any analogical guidance (or other form of 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

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

(UNLOAD-ROCKET
 (params
  ((<obj> OBJECT)
   (<loc> LOCATION)))
 (preconds
  (and
   (inside <obj> ROCKET)
   (at ROCKET <loc>)))
 (effects
  ((add (at <obj> <loc>))
   (del (inside <obj> ROCKET)))))

(MOVE-ROCKET
 (params nil)
 (preconds
  (at ROCKET locA))
 (effects
  ((add (at ROCKET locB))
   (del (at ROCKET locA)))))
```

Figure 2. The ONE-WAY-ROCKET domain.

```
(has-instances OBJECT obj1 obj2)
(has-instances LOCATION locA locB)

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

Figure 3. A problem in the ONE-WAY-ROCKET world.

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 tree is needed to reach a solution.

Figure 4 shows the conceptual tree, i.e., the subgoaling structure, for this problem. The top node *FINISH* represents the final operator that is applied to show that the user-given problem is completely solved. The numbers show the execution order of the plan steps.

NoLimit solves this problem, where linear planners fail (but where of course other complete planners also succeed), because it switches attention to the conjunctive goal (*at obj2 locB*) before completing the first conjunct (*at obj1 locB*). This is shown in figure 4 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 subgoals in the subgoaling stack of this goal. NoLimit changes its focus of attention to the other top-level goal (*at obj2 locB*), 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 the search.

While solving this problem, NoLimit automatically annotates the decisions taken with justifications that reflect its experience while searching for the solution. Figure 5 shows an example of a case generated from a problem-solving episode for this two-object problem. We represent only the choice and the subgoaling links for each node. These are extracted from the conceptual tree, as shown in figure 4, which is incrementally expanded at search time. Figure 6 represents the complete goal-decision node, *cn6*, to show the record

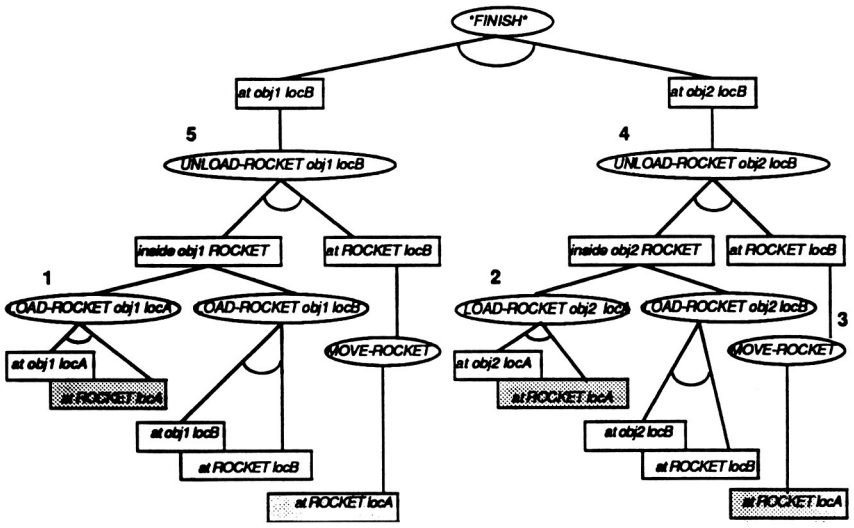


Figure 4. The complete conceptual tree for a successful solution path. The numbers at the nodes show the execution order of the plan steps.

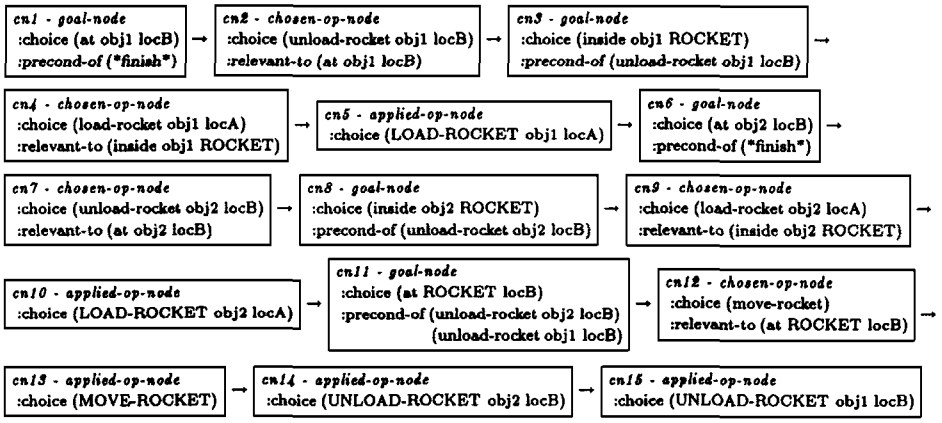


Figure 5. A case as a sequence of annotated decision nodes.

of a failure. It corresponds to the situation where the correct decision of choosing to work on the goal (*at obj2 locB*) was taken after having failed when working first on (*at ROCKET locB*). The decision node stored for the goal (*at obj2 locB*) 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*). This goal needs to be achieved in order to load *obj2* into the *ROCKET*.

```
Frame of class goal-decision-node
 :choice  (at obj2 locB)
 :sibling-goals
       (((at ROCKET locB) (:no-relevant-ops (at ROCKET locA))))
 :sibling-applicable-ops NIL
 :why-subgoal NIL
 :why-this-goal NIL
 :precond-of  (*finish*)
```

Figure 6. Saving a goal decision node with its justifications. An example: The goal decision node cn6.

The generated case corresponds to the search tree compacted into the successful path annotated with the justifications that resulted in the sequence of correct decisions that led into a solution to the problem. In essence, a case, as shwon in figure 5, is a sequence of decision nodes such as the one illustrated in figure 6.

3. The derivational replay: Case utilization

The general replay mechanism involves a complete interpretation of the justification structures in the next context, and development of adequate actions to be taken when transformed justifications are no longer valid. 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). For all the choice points, the problem solver also tests the validity of the justifications (it semantic applicability, or rather its "desirability" in the new situation). In case the choice remains valid in the current problem state, it is merely copied, and in case it is not valid the implemented system has two alternatives:

1. Replan at the particular failed choice, e.g., 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, or

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.

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 and proceeds as if nothing had gone wrong earlier. If the extra steps performed do not interfere with the already replayed case steps, the extension occurs without further problems. It may also happen that future steps in the case continue to fail and the case is abandoned.

Justification structures also encompass the record of past failures in addition to the subgoaling links (Kambhampati, 1989). This allows both the early pruning of current alternatives that were experienced to have failed in the past, and the exploration of alternatives for which the past reason of failure does not exist in the current situation. Furthermore, the replay mechanism in the context of casual commitment as opposed to least commitment allows naturally to combine guidance from several past problem-solving episodes (Veloso, 1991). Replicated adapted decisions can be interleaved and backtracked upon within the totally ordered reasoning plan.

3.1. Pursuing the ONE-WAY-ROCKET example

Let us return to the *ONE-WAY-ROCKET* problem introduced in section 2.2 to illustrate briefly the derivational replay process. We show the results obtained in the problems of moving three objects and four objects from *locA* into *locB* in table 1. Each row of the table refers to one new problem, namely, the two- (2objs), three- (3objs), and four-object (4objs) problems. We show the average running time of NoLimit without analogy (base search) and using analogical guidance from one of the other cases.[2] We performed a large number of runs where the choices were taken randomly. Hence, the numbers shown represent the average among a large spectrum of possible search situations.

The solution is replayed whenever the same step is a possible step and the justifications hold. For example, in using the two-object case as guidance to the three- (or four-) object problem, the failure justification for moving the rocket—"no-relevant-ops (at ROCKET locA)"—is tested, and this step is not replayed until all the objects are loaded into the rocket. The improvements obtained are high, as the new cases are extensions of the previous cases used for guidance. Maximal improvement is achieved when the case and the new problem

Table 1. Replaying a justified past solution.

		Replayed Cases		
New Prob	Base Search	Case 2objs	Case 3objs	Case 4objs
2objs	4.5s	2s	2s	2s
3objs	14.75s	4.75s	3.25s	3.25s
4objs	117.5s	7.75s	7.75s	5.75s

differ substantially (two-objects and four-objects, respectively). Allen and Langley (1990) obtained similar results in simple domains by replaying one-step past cases as opposed to a complete sequence of problem-solving decisions.

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 gaining insight solving the reduced problem from scratch (i.e., building a reference case), and finally solving the original four-object problem by analogy with the simpler problem. The running time of this two-step process still adds up to less than trying to solve the extended problem directly, without analog for guidance: 4.5 s + 7.75 s = 12.25 seconds (solving the two-object from scratch (4.5 s) + the derivational replay of the two-object for the four-object problem (7.75 s)) versus 117.5 seconds for solving the four-object problem 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.

To show some additional results from two other substantially more complicated domains, we first discuss the case retrieval strategy followed.

4. Case retrieval: The similarity metric

Several research projects study the problem of assigning adequate similarity metrics (recent work includes Bareiss & King, 1989; Kolodner, 1989; Porter et al., 1989). Our approach relies on an incremental understanding of an increasingly more appropriate similarity metric. In Veloso and Carbonell (1989), we introduced our proposed memory model, SMART (standing for Storage in Memory and Adaptive Retrieval over Time). NOLIMIT, the nonlinear analogical problem solver, provides SMART the information about the utility of the candidate cases suggested as similar in reaching a solution. This information is used to refine the case library organization and in particular the similarity metric. In this section we analyze two similarity metrics with different degrees of problem-context sensitivity. We first introduce a simple direct similarity metric and proceed to refine it by analyzing the derivational trace produced by the analogical problem solver.

4.1. A direct similarity metric

Let $\mathcal{S}$ be the initial state and $\mathcal{G}$ be the goal statement, both given as conjunctions of literals. A *literal* is an instantiated predicate, i.e., literal = (predicate argument-value*). As an example, (inroom key12 room1) is a literal where inroom is the predicate and key12 and room1 are its instantiated arguments.

Each past case P in memory is indexed by the corresponding initial state and goal statement, respectively $\mathcal{S}^P$ and $\mathcal{G}^P$. When a new problem P' is given to the system in terms of its $\mathcal{G}^{P'}$ and $\mathcal{S}^{P'}$, retrieving one (or more) analog consists in finding a *similar* past case by comparing these two inputs $\mathcal{G}^{P'}$ and $\mathcal{S}^{P'}$ to the indices of past cases.

Definition 1. *We say that a conjunction of literals* $L = l_1, \ldots, l_n$ **directly matches** *a conjunction of literals* $L' = l_1', \ldots, l_m'$ *under a substitution* σ *with* **match value** δ, *if there are* δ **many** *literals in L that directly match some literals in L' under* σ. *A literal l directly matches a literal l', iff*

- *The predicate of l is the same as the predicate of l'.*
- *Each argument of l is of the same class (type) as its corresponding argument of l'.*

In this case, there is a substitution σ, *such that* $l = \sigma(l')$.

As an example, the literal (inroom box1 room1) *directly matches* the literal (inroom boxA roomX), where box1 and boxA are both of class BOX and room1 and roomX are of class ROOM. Under the substitution σ = {box1/boxA, room1/roomX}, (inroom box1 room1) = σ ((inroom boxA roomX)).

We first compute a simple partial match value between problems as the sum of the match value of their corresponding initial states and goal statements calculated independently, as presented in definition 2.

Definition 2. *Let P and P' be two particular problems, respectively with intiial states* $\mathcal{S}^P$ *and* $\mathcal{S}^{P'}$ *and goal* $\mathcal{G}^P$ *and* $\mathcal{G}^{P'}$. *Let* $\delta_{\mathcal{G}}^{\sigma(P),P'}$ *be the match value of* $\mathcal{G}^P$ *and* $\mathcal{G}^{P'}$, *under some substitution* σ. *Let* $\delta_{\mathcal{S}}^{\sigma(P),P'}$ *be the match value of* $\mathcal{S}^P$ *and* $\mathcal{S}^{P'}$, *under the substitution* σ. *Then we say that the two problems P and P'* **directly match** *with* **match value** $\delta^{\sigma(P),P'} = \delta_{\mathcal{G}}^{\sigma(P),P'} + \delta_{\mathcal{S}}^{\sigma(P),P'}$ *under substitution* σ.

The partial match value of two problems is substitution dependent. As an example, consider the goal $\mathcal{G}$ = {(inroom key12 room1), (inroom box1 room1)}, and the goal $\mathcal{G}'$ = {(inroom key13 room4), (inroom key14 room2), (inroom box53 room4)}. Then $\mathcal{G}$ directly matches $\mathcal{G}'$ *with match value* $\delta = 2$ under the substitution σ = {key12/key13, room1/room4, box1/box53}, and *match value* $\delta = 1$ under the substituion σ' = {key12/key14, room1/room2}.

In a first experiment we used this direct similarity metric to evaluate the partial match between problems, not considering therefore any relevant correlations between the initial states and the goal statements. The procedure in figure 7 retrieves the set of the *most similar* past cases.

*4.1.1. Examples in the process-job planning and extended-*STRIPS *domains*

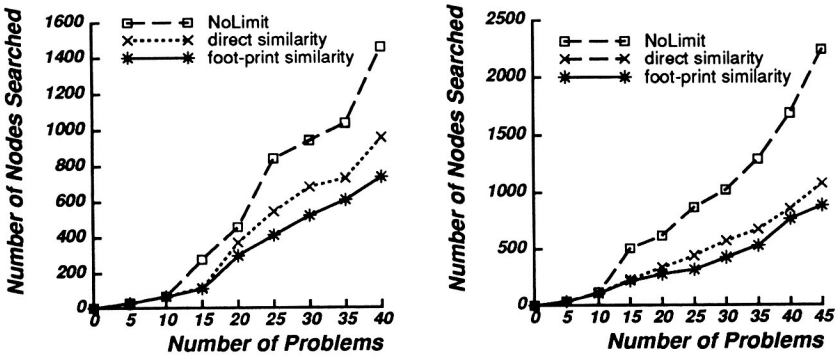
We ran NOLIMIT without analogy over a set of problems in the process-job planning and in the extended-STRIPS domains.[3] We accumulated a library of cases. In order to factor away other issues in memory organization, the case library was simply organized as a linear list of cases. We then ran again a new set of problems using the case library.

The dotted curves in figures 8(a) and (b) show the results for these two domains. We plotted the average cumulative number of nodes searched. We note from the results that analogy showed an improvement over base search (dashed curves): a factor of 1.5-fold higher

Input: A case library $\mathcal{L} = \mathcal{P}_1, \ldots, \mathcal{P}_p$, and a new problem $\mathcal{P}'$.
Output: The set of problems from $\mathcal{L}$ and corresponding substitutions that make them the most similar ones to $\mathcal{P}'$.

```
procedure Retrieve_Most_Similar_Past_Cases(L, P'):
1.  current_max_match ← 0
2.  Most_Similar ← {}
3.  for i ← 1 to p do
4.      Compute δ_G^{P_i,P'} for all the possible goal substitutions.
5.      for each substitution σ such that δ_G^{σ(P_i),P'} ≠ 0 do
6.          Apply substitution to the initial state S_i.
7.          Compute δ_S^{σ(P_i),P'}.
8.          δ^{σ(P_i),P'} ← δ_G^{σ(P_i),P'} + δ_S^{σ(P_i),P'}
9.          if δ^{σ(P_i),P'} > current_max_match
10.             current_max_match ← δ^{σ(P_i),P'}
11.             Most_Similar ← {(P_i, σ)}
12.         if δ^{σ(P_i),P'} = current_max_match
13.             Add (P_i, σ) to Most_Similar.
14. Return Most_Similar
```

Figure 7. Retrieving the most similar past cases.



(a) Process-Job Planning Domain (b) Extended-STRIPS Domain

Figure 8. Comparison in the process-job planning and extended-STRIPS domains.

for the process-job planning and scheduling domain and 2.0-fold for the extended-STRIPS domain. (We will see later the meaning of the solid curves.) In general the direct similarity metric leads to acceptable results. However, analyzing the results, we notice that the straightforward similarity metric does not always provide the best guidance when there are several conjuncts in the goal statement.

The problem of matching conjunctive goals turns out to be rather complex. Since conjunctive goals may interact, it is not at all clear that problems are more similar based simply on the *number* of literals that match the initial state and the goal statements. Noticing therefore

that matching conjunctive goals involves reasoning over a large lattice of problem configurations, we developed a new similarity metric by refining the indexing based on the derivational trace of a past solution.

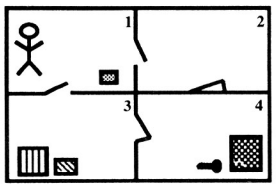
4.2. The foot-print similarity metric

The derivational trace identifies for each goal the set of *weakest preconditions* necessary to achieve that goal. Then recursively we create the *foot-print* of a user-given goal conjunct by doing a goal regression, i.e., projecting back its weakest preconditions into the literals in the initial state (Waldinger, 1981; Mitchell et al., 1986). The literals in the initial state are therefore *categorized* according to the goal conjunct that employed them in its solution.

Definition 3. *For a given problem P and corresponding solution, a literal in the initial state is in the* **foot-print** *of a goal conjunct g, iff it is in the set of the weakest preconditions of g according to the derivational trace of the solution.*

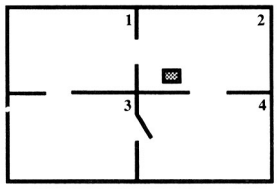
The purpose of retrieving a similar past case is to provide a problem-solving episode to be replayed for the construction of the solution to a new problem. We capture into the similarity metric the role of the initial state in achieving the different goal conjuncts with respect to a particular solution found. Details of particular initial state configurations are not similar per se. Instead they are similar as a function of their relevance in the solution encountered.

In figure 9(a) we show the initial state and in figure 9(b) the goal statement of an example problem from the extended-STRIPS domain. Rooms are numbered at the corner of their picture and doors are named according to the rooms they connect. Doors may be open, closed, or locked. In particular, door24 connects the rooms 2 and 4 and is locked. Door34




```
(inroom robot room1)   (inroom box1 room1)
(inroom box2 room3)    (inroom box3 room3)
(inroom box4 room4)    (inroom key24 room4)
(door-open door12)     (door-open door13)
(door-closed door34)   (door-locked door24)
```

(a) Initial State




```
(and (inroom box1 room2)
     (door-open door34))
```

(b) Goal Statement

Figure 9. Problem situation in the extended-STRIPS domain. The goal statement is a partial specification of the final desired state: the location of other objects and the status of other doors remained unspecified.

is closed and, for example, door12 is open. The number of the boxes can be inferred by the attached description of the initial state.

Assume NoLimit solved the problem in figure 9 by first pushing box1 from room1 into room2. In order to open door34, the robot then goes to room3 back through room1. The actual solution searched for and found would be the plan (GOTO-BOX box1) (PUSH-THRU-DOOR box1 door12) (GO-THRU door12 room1) (GOTO-DOOR door13) (GO-THRU door13 room3) (GOTO-DOOR door34) (OPEN-DOOR door34). For this particular solution, for example, the key24 for the locked door24 does not play any role in achieving the goals and it is not, therefore, a *relevant* literal in the initial state of this problem, if this problem-solving episode is to be replayed. In figure 10(a) we show the actual foot-print of the initial state corresponding to this first solution to the problem. The foot-print-state-goal slot associates each literal in the initial state with list of goals that it contributed to achieve. Note that *nil* means that the literal was not used for any goal.

However, NoLimit could have encountered a different solution to this problem. The robot can push box1 opportunistically on its way to open door34. It pushes then box1 through door24 into room2, after unlocking door34. The actual solution searched for and found would be the plan (GOTO-BOX box1) (PUSH-THRU-DOOR box1 door13) (PUSH-TO-DOOR box1 door34) (OPEN-DOOR door34) (PUSH-THRU-DOOR box1 door34) (GOTO-KEY key24) (PICK-UP key24) (GOTO-DOOR door24) (UNLOCK-DOOR door24) (OPEN-DOOR

```
(INDEXED-PROBLEM strips-4-0
 (has-instances
   (agent robot)
   (box box1 box2 box3 box4)
   (key key24)
   (room rm1 rm2 rm3 rm4)
   (door door12 door13 door24 door34))
 (goal (and (inroom box1 rm2) (door-open door34)))
 (foot-print-state-goal
  ((arm-empty) ((inroom box1 rm2)))
  ((connects door12 rm1 rm2) ((inroom box1 rm2)))
  ((connects door13 rm1 rm3) ((door-open door34)))
  ((connects door34 rm3 rm4) ((door-open door34)))
  ((connects door24 rm2 rm4) nil)
  ((pushable box1) ((inroom box1 rm2)))
  ((door-open door12) ((inroom box1 rm2)
                       (door-open door34)))
  ((door-open door13) ((door-open door34)))
  ((door-closed door13) ((door-open door34)))
  ((door-locked door24) nil)
  ((inroom robot rm1) ((inroom box1 rm2)
                       (door-open door34)))
  ((inroom box2 rm3) nil)
  ((inroom box3 rm3) nil)
  ((inroom box4 rm4) nil)
  ((inroom key24 rm4) nil)))
```

(a)

```
(INDEXED-PROBLEM strips-4-1
 (has-instances
   (agent robot)
   (box box1 box2 box3 box4)
   (key key24)
   (room rm1 rm2 rm3 rm4)
   (door door12 door13 door24 door34))
 (goal (and (inroom box1 rm2) (door-open door34)))
 (foot-print-state-goal
  ((arm-empty) ((inroom box1 rm2)))
  ((connects door12 rm1 rm2) nil)
  ((connects door13 rm1 rm3) ((door-open door34)
                              (inroom box1 rm2)))
  ((connects door34 rm3 rm4) ((door-open door34)
                              (inroom box1 rm2)))
  ((connects door24 rm2 rm4) (inroom box1 rm2))
  ((pushable box1) ((inroom box1 rm2)))
  ((door-open door12) nil)
  ((door-open door13) ((inroom box1 rm2)
                       (door-open door34)))
  ((door-closed door13) ((inroom box1 rm2)
                         (door-open door34)))
  ((door-locked door24) ((inroom box1 rm2)))
  ((inroom robot rm1) ((inroom box1 rm2)
                       (door-open door34)))
  ((inroom box2 rm3) nil)
  ((inroom box3 rm3) nil)
  ((inroom box4 rm4) nil)
  ((inroom key24 rm4) ((inroom box1 rm2)))))
```

(b)

Figure 10. Two foot-prints in the extended-STRIPS domain for different solutions to the same problem. The initial state literals are associated with the goal conjunct(s) they contributed to achieve, according to the goal regression of the different plans encountered.

door24) (GOTO-BOX box1) (PUT-DOWN key24) (PUSH-THRU-DOOR box1 door24). Now in this way of solving the problem, the key24 for the locked door24 is a relevant literal in the initial state of this problem if this problem-solving episode is to be replayed. Figure 10(b) shows the actual foot-print of the initial state for this solution.

We formally define the new similarity metric that evaluates the degree (or value) of match of the initial state as a function of the goal conjuncts that directly matched. This similarity emphasizes goal-oriented behavior (Kedar-Cabelli, 1985; Hammond, 1986) even more than the one introduced earlier by focusing only on the goal-relevant portions of the initial state (Hickman & Larkin, 1990; Pazzani, 1990), as determined by the problem solver for each case in the library.

Definition 4. *We say that the initial state* $\mathcal{S}$ **foot-print matches** *an initial state* $\mathcal{S}'$ *under a substitution* σ *and given matched goals* $g_1^m, \ldots, g_k^m$ **with match value** δ, *iff there are* δ **many** *literals* l *in* $\mathcal{S}$, *such that (i)* l *directly matches some literal* l' *in* $\mathcal{S}'$ *under* σ, *and (ii)* l *is in the foot-print of some goal* g_i^m, *for* $i = 1, \ldots, m$.

When assigning a match value to two problems, we do not consider now only the *number* of goals and literals that match in the initial state. Instead, we also use the unified goals themselves to determine the match degree of the initial state.

We change steps 4 and 8 of the procedure presented in figure 7 according to definition 4. Step 4 computes the match value for the goal statements but further return which goals matched, and in step 8 we use these goals to compute the match value for the initial states. The rest of the algorithm is invariant to selection of similarity metric.

4.2.1. Further search-reduction examples

We ran new experiments with this foot-print similarity metric in the extended-STRIPS and process-job planning domains. The solid curves in figures 8(a) and 8(b) show the results for these two domains. We note that the results with the foot-print similarity metric show a 2.0-fold improvement over the base search for the process-job planning and scheduling domain and a 2.6-fold-improvement for the extended-STRIPS domain. The curves obtained do not represent the best improvement expected, as the set of 40 problems used does not completely cover the full range of problems in either domain. One of the directions of our current research is to develop techniques for learning similarity metrics by further automatically analyzing the analogical replay mechanism.[4]

To scale the system well in both the size and diversity of domains, we have currently a 1000-case library in a complex logistics transportation domain. In this domain, packages are to be moved among different cities. Packages are carried within the same city in trucks and across cities in airplanes. Trucks and airplanes may have limited capacity. At each city there are several locations, e.g., post offices and airports. Although our analysis in this large-scaled domain is not complete yet, the results so far show high positive transfer, including total memory retrieval and problem-solving times, thus demonstrating the scale capabilities of our methods.

4.3. Trading off retrieval and search costs

In pure general-purpose problem solvers, the cost of search is exponential in the length of the solution. (We refer to systems that search without any control knowledge to prune the search space of possible operators.) In pure case-based reasoning systems the cost of retrieval is very high, as the system fully relies on retrieving the *best* case in memory to maximize its chance of successful adaptation.

In the analogical version of PRODIGY, where we integrate a search-based problem solver with an analogical reasoner, we balance the cost of retrieving and the predicted search cost (Veloso & Carbonell, 1991b). We show how we balance the cost of retrieval as a function of the degree of partial match. In the retrieval procedure of figure 7, suppose that the memory is organized in a discrimination network (as we are currently developing). The organization of the memory is such that the indices for the cases are less relevant as we move away from the root of the discrimination network. Given a new problem P' with initial state $\mathcal{S}^{P'}$ and goal $\mathcal{G}^{P'}$, we can compute the absolute maximum possible match value, *absolute_max_match* = length $(\mathcal{G}^{P'})$ + length $(\mathcal{S}^{P'})$.

In general, we integrate analogy and search to reduce the size of the search space in terms of the number of nodes searched and consequently achieve an improvement in running time. Harandi and Bhansali (1989) concluded that analogy would be useful if the time to find analogs is small and the degree of similarity is high. Hickman, Shell, and Carbonell (1990) also show that internal analogy can reduce the search cost. We show now that there is an optimal range of retrieval time to spend searching for candidate analogs. Intuitively, the deeper that memory is searched, the better the analog and the less search required by the problem solver. However, searching memory also takes time. Is there, hence, an optimal amount of effort to spend searching memory?

We assume that the memory is organized in such a way that the confidence on the match degree increases monotonically with retrieval time (Kolodner, 1984; Schank, 1982), though not necessarily in a linear manner. Assume also that there is always one (or more) case available to return when retrieval is halted. If the retrieval time increases, the match value between the case returned and the new problem might increase. We now formalize this model. Let

- t_r be the time spent to retrieve a similar past case,
- δ_{t_r} be the match value between the case retrieved and the new problem, as a function of t_r,
- m be the absolute_max_match as introduced above, and
- d be the percentage of deviation from the absolute_max_match of the match value of the case retrieved if the retrieval time is null (or close to null).

To capture the fact that the math value may increase with the retrieval time, we say that

$$\delta_{t_r} = m(1 - dC^{-\alpha t_r}), \tag{1}$$

where C and α are constants.

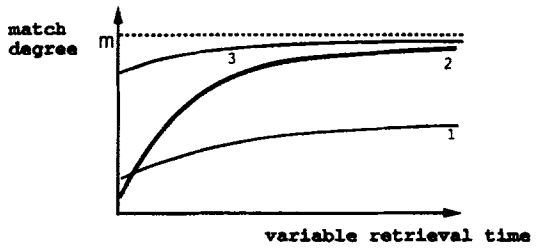


Figure 11. Three different curves for the match value as a function of the retrieval time.

Figure 11 sketches three possible curves for this match value δ_{t_r} as a function of the retrieval time t_r. Curves 1 and 2 show situations where the initial match is poor, i.e., with low match degree. However, for curve 1, the rate of match-degree improvement is very low (low α), while for curve 2 the match degree increases fast with the retrieval time. Situations 1 and 2 depict two different rates of improvement for the match result while traversing down the discrimination net. Curve 3 plots a situation where the initial match is immediately high and continues to improve gradually towards the maximum.

In the situations captured by curves 1 and 3, the system should not invest a long time in retrieving a *better, or best*, similar past case. In both cases termination will occur because the rate of improvement, α, is low. In case 1, the system should solve the problem by base search, as there are no good cases, and in case 3 it should immediately start derivational replay on the retrieved high-match case, rather than waste time seeking a marginally better one. Situation 2 illustrates the case where retrieval time is more wisely invested.

Given the fact that the match degree is on average directly related to search savings in problem solving, we now show analytically that there is an optimal amount of effort to spend in searching memory for candidate analogs.

Consider that a search tree can be seen as an OR-tree, branching alternatively among possible goal orderings and possible operators to achieve a goal. Let b be the average branching factor of the search tree, let l be the solution length for a given problem, and S be the search effort without analogy. Then the complexity of S is $S = \mathcal{O}(b^{\mathcal{O}(l)})$. (From now on, for simplicity, we skip the order of notation, $\mathcal{O}$.) Assume that the effect of analogical reasoning is captured in a decrease of the average branching factor b (Hickman et al., 1990). This reduction of the search effort is in direct relationship with the match degree of the guiding case(s). Let $S_{analogy}$ be the search spent with analogy. We can then say that, for some linear function f,

$$S_{analogy} = ((1 - f(\delta_{t_r}))b)^l. \tag{2}$$

The purpose of the integrated analogical reasoner is to improve the effort to reach a solution: memory search time plus problem solving search time. The objective is to find the situation when this sum is much smaller than brute-force problem solving search without any analogical guidance. We capture this goal in the inequality below, where we do not represent, for simplicity, the function f introduced in equation (2):

$$t_r + ((1 - \delta_{t_r})b)^l \ll b^l. \tag{3}$$

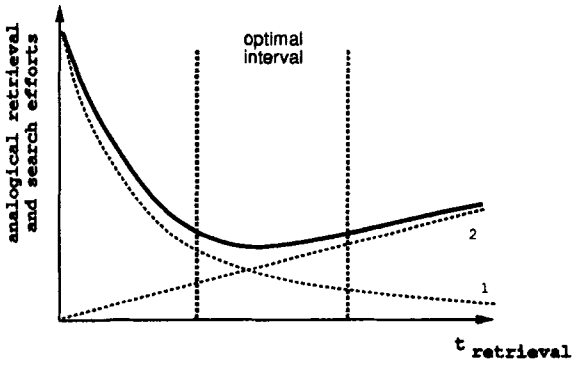


Figure 12. Retrieval time (curve 2) plus analogical search effort (curve 1).

Substituting equation (1) into equation (3), we get the final equation as a function of the retrieval time t_r:

$$t_r + (1 - m(1 - dC^{-\alpha t_r}))^l b^l \ll b^l \tag{4}$$

Figure 12 sketches the left-hand side of inequality 4. Analyzing this qualitative curve, we conclude that there is an optimal retrieval time interval, which is a function of the dynamic match rate α. Retrieval should then stop when a given threshold is reached, namely, when the derivative of the expected search savings approaches the incremental memory search cost.

5. Case storage: Interpreting the utility of guidance provided

Currently the cases in the case library are indexed by their goal conjuncts and initial state (Veloso, 1991). Cases are clustered by goal, and within the same cluster, cases are all equally close to each other.

We view the final desired behavior of the system dynamically reorganizing its case library as the resulting interaction of the two functional modules, namely, the *problem solver* and *the memory manager.* In a nutshell, the problem solver has the ability

1. to ask the memory manager for advice on how to solve a problem (i.e., guidance based on past experience, stored as fully annotated derivational traces),
2. to replay the past solutions received as analogs and create an annotated solution for the new problem based both on the guidance received from the memory manager, and on the domain theory available, and
3. to return to the memory manager information about the utility of the guidance received for creating the solution (i.e., the relevance of the retrieved cases) and the new justified case (a new fully annotated derivational trace).

Memory organization will be in a closely coupled dynamic relationship with the problem-solving engine. SMART, the memory manager, has the ability

1. to search its case library for one or more cases solved in the past that best relate to the new problem presented by the problem solver.
2. to reorganize and create new links between the cases stored as a function of the feedback received from the problem solver on the utility of the guidance provided (the retrieved case) in solving the new problem.

The problem solver and SMART communicate as shown in figure 13, where W_i is the initial world, G is the goal to be achieved, W_f is the final world, *Sol* is the solution found, *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.

In the context of the discussion above on the actions taken at replay time (see section 3), we identify four situations (see figure 14) that encode the *utility* as judged by the problem solver on the guidance received from memory (Veloso & Carbonell, 1989):

Fully-used (see figure 14(a)). In this situation the problem solver is able to replay the previous case, fully validating the justifications.

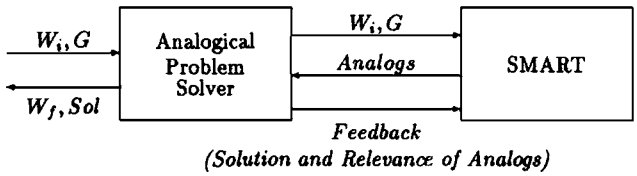


Figure 13. Interaction of the problem solver and the memory manager.

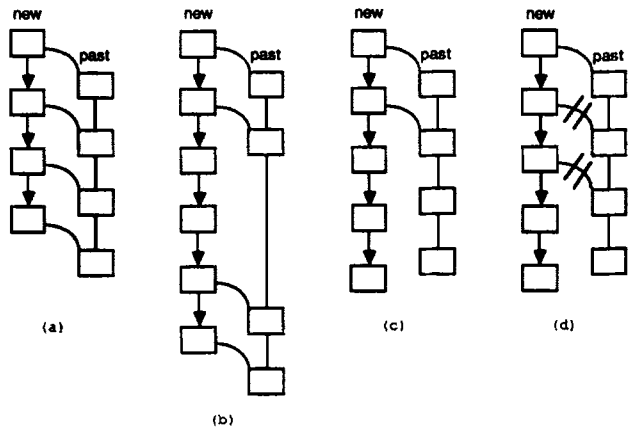


Figure 14. Four situations to encode the usefulness of the guidance received: (a) Fully-used: past case is fully copied; (b) Extension: past case is copied but additional steps are performed in the new case; (c) Locally divergent: justifications do not hold and invalidate copying part of the past case; (d) Globally divergent: extra steps are performed that undo previously copied steps.

Extension (see figure 14(b)). The guiding case is fully used, but there is some extra work done to re-establish a failed condition. These extra steps taken do not invalidate the successfully used guidance; they are spliced into the resulting solution.

Locally Divergent (see figure 14(c)). The case suggested and the current one diverge due to some failed justification after an initial successful replay. The two cases fully diverge from that point, though again the new steps performed do not interfere with the earlier steps performed under the case guidance.

Globally Divergent (see figure 14(d)). The replay diverges from the retrieved case, and fully-justified decisions prior to the divergence point must be undone, because the problem solver switches to a different strategy (e.g., attempts different operators for the top-level goals).

These four situations determine the reorganization of memory when the new case is to be stored back into memory. We are exploring precise algorithms to address each of these situations in particular. An informal discussion follows below on how we expect these situations to be handled.

If a case was *fully used* under a particular substitution, SMART will generalize its data structure over this match, updating the indices to access these cases (Veloso & Carbonell, 1989). 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 should also occur on the common parts of the case. The situations where the two cases *diverge* represent a currently incorrect memory concept of similarity or lack of knowledge. The fact that the retrieval mechanism suggested a past case as being most similar to the new problem, and that 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 distinguish two categories of indices for stored cases, namely, *problem-* and *search-*dependent ones. *Problem*-dependent indices derive from the description of the initial state of the world and the goal statement. *Search*-dependent indices are related to the justification structure built by the problem solver during its search process for a solution to the problem. As an example, suppose that at a certain choice point, only one alternative is available, i.e., there is only one way of achieving a particular goal. The justification at a choice point referring to a *unique* alternative available is a strong index for this case. This choice point is a bottleneck in the search path. In the new situation, if this justification is still valid and this unique alternative fails, the remaining portion of the solution cannot be replayed easily. Hence, we plan to extract memory indices from the justification structure, and use them at retrieval time to prune the set of candidate analogs more adequately.

6. Conclusion

The results reported here demonstrate the feasibility of derivational analogy as a means to integrate general problem solving with case-based reasoning. In summary, we have shown that

- A general problem solver, such as NoLimit, can successfully create its own case library by recording solution traces and their accompanying justification structure.
- Derivational analogy can exploit past cases to solve new but similar problems, and do so significantly faster than standard problem solving without the benefit of accumulated cases.
- A rich derivational structure can yield improvements over direct trial-and-error replay of cases for related but non-identical problems. The justification structure permits the case transfer to be partial when total transfer cannot be justified.
- The integration of the two problem-solving paradigms, namely, general problem solving and case-based reasoning, can be explored to minimize the trade-off between memory and search. Problem-solving search cost can be significantly reduced by replaying past similar derivational traces of problem-solving episodes, and incrementally better similarity metrics can be learned by interpreting the behavior of the problem solver replaying retrieved cases.
- Finally, an efficient balancing of the costs of retrieval and search can help the integrated system to dynamically scale up its case library.

However, the research into full-fledged case-based reasoning and machine learning in the context of NoLimit, the PRODIGY nonlinear problem solver is still in progress. The tight integration between the analogical replay and the problem solver enables cases to be used at any level of problem solving, from the entire solution structure to the achievement of individual subgoals deep in the goal tree. A new large or multi-part problem may therefore be solved by appealing to multiple smaller cases. The full implementation of the SMART memory model is also in progress.

We showed how derivational analogy differs from standard case-adaptation methods in several dimensions. The most significant one is that a case consists not of a rigid data structure, which may be retrieved, applied, and at best "tweaked," but rather of a network of fully justified advice to the problem solver, directing its decision making in future similar situations. Another dimension is the coverage attained from a case. It may be used in whole or in part, due to the flexibility of the replay procedure.

Finally, we also note that derivational analogy differs from other learning methods in PRODIGY. Static analysis and construction of abstraction hierarchies (Etzioni, 1990a; Knoblock, 1991) provide *eager* learning mechanisms. Domain definitions are precompiled into more efficient forms. Explanation-based learning (Minton, 1988) is dynamically triggered but when applied performs a full weakest-precondition proof procedure requiring a complete domain theory. Therefore, it combines aspects of *eager* and *lazy* learning. Derivational analogy takes the extreme point in the spectrum, as no processing is done on the cases as they are recorded. Instead, all the processing is done at replay time (initial retrieval, derivational replay, and memory-feedback-adjustment). Learning, therefore, is done only on an "if-needed" basis. It is the ultimately lazy machine learning method. Note that we are not claiming superiority of one method over others. Instead, we believe that each method has its role, depending upon the characteristics of the domain, the problem, and the kind of solution required. PRODIGY, in fact, includes all these learning methods, and determining the appropriate principles for effective integration and selection among them is a high-priority topic on our research agenda.

Appendix: The PRODIGY architecture

PRODIGY is a general problem solver combined with several learning modules. 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.1. The problem solver

PRODIGY's basic reasoning engine is a general-purpose problem solver and planner that searches for sequences of operators (i.e., plans) to accomplish a set of goals from a specified initial state description. Search control in PRODIGY is governed by a set of *control rules* that apply at each decision point, and may consist of heuristic preferences or definitive selections. Control rules may be domain independent or (more typically) domain specific. The control language allows the problem solver to represent and learn control information about the various problem-solving decisions, such as selecting which goal/subgoal to address next, which operator to apply, what bindings to select for the operator, or where to backtrack in case of failure. Different disciplines for controlling decisions can be incorporated (Drummond & Currie, 1989).

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. Precondition expressions are well-formed formulas in a typed first-order predicate logic encompassing negation, conjunction, disjunction, and existential and universal quantification. The effects-list indicates atomic formulas that should be added or deleted from the current state when the operator is applied, reflecting the actions of the operator in the world. In addition one can also include *conditional* effects that specify transformations to the world as a function of its current state at application time.

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 a **nonlinear** problem solver capable of searching through the space of all alternative instantiated operators and all possible orderings of the set of pending goals (Veloso, 1989). The system is called NOLIMIT, standing for **Nonlin**ear problem solver using casual com**mit**ment. Nonlinear planning was developed to deal with problems like Sussman's anomaly, which could not be solved by rudimentary linear planners such as STRIPS (Fikes & Nilsson, 1971; Sussman, 1973). Least-commitment planners handle this anomaly by deferring decisions while building the plan (Sacerdoti, 1975; Wilkins, 1989). These planners typically output a partially ordered plan as opposed to a totally ordered one, and consequently the term *nonlinear plan* is used. However, the essence of the *nonlinearity* is not in the fact that the plan is partially ordered,

but in the fact that a plan need not be a linear concatenation of complete subplans,each for a goal presumed independent of all others (Veloso, 1989). We follow instead a casual-commitment approach (Minton et al., 1989), as opposed to a least-commitment approach, to the nonlinear planning problem. Alternatives are generated and tested incrementally and all decision points (operator selections, goal orderings, backtracking points, etc.) are open to introspection and reconsideration.

The basic search procedure is, as in the linear planner (Minton et al., 1989), a 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 proceeds in this apparently simple way. Its nonlinear character stems from working with a **set** of goals in this cycle, as opposed to the top goal in a goal stack. Dynamic goal selection enables NoLimit to interleave plans, exploiting common subgoals and addressing issues of resource contention. The skeleton of NoLimit's search algorithm is shown in figure 15.

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. We can run NoLimit in *multiple-solutions* mode, where NoLimit shows the solution found and continues searching for more solutions, which it groups into *buckets* of solutions. Each *bucket* has different solutions that use the same set of plan steps (instantiated operators). The set of different totally ordered solutions within a bucket forms a potential partially ordered solution.

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

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 15. A skeleton of NoLimit's search 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 an *applicable* chosen operator. An operator is *applicable*, iff all its preconditions are true in the state. The operator considered to be applicable is the last chosen operator not applied yet in the current search path. 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 following regular expression: (*goal chosen-operator applied-operator**)*.

A.2. The learning modules

PRODIGY's general problem solver is combined with several learning modules. The PRODIGY architecture, in fact, was designed both as a unified testbed for different learning methods and as a general architecture to solve interesting problems in complex task domains. Let us now focus on the global architecture itself, as diagrammed in figure 16.

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, to

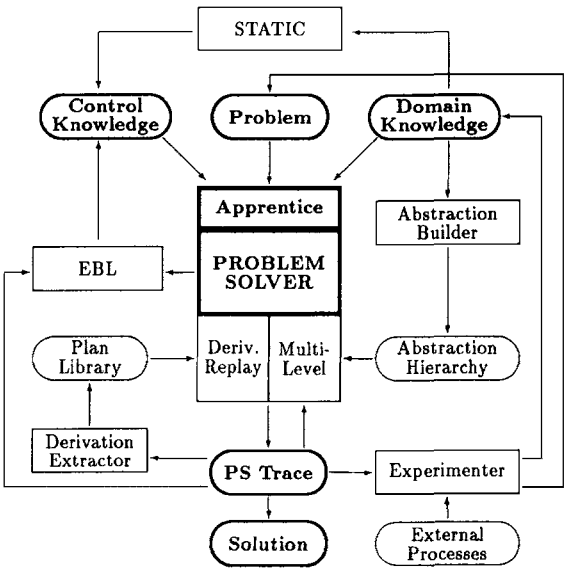


Figure 16. The PRODIGY architecture: Multiple learning modules unified by a common representation language and a general problem solver.

build derivational traces (cases) by the derivational analogy engine, to analyze key decisions by the APPRENTICE knowledge acquisition interface, or to formulate focused experiments.

In addition to the central problem solver, PRODIGY integrates the following learning components:

APPRENTICE: A graphic-based user-interface that can participate in an apprentice-like dialogue, enabling the user to evaluate and guide the system's problem solving and learning (Joseph, 1989).

EBL: An explanation-based learning facility (Minton, 1988) for acquiring control rules from a problem-solving trace. Explanations are constructed from an axiomatized theory describing both the domain and relevant aspects of the problem solver's architecture. The resulting descriptions are expressed in control rule form.

STATIC: A method for learning control rules by analyzing PRODIGY's domain descriptions prior to problem solving (Etzioni, 1990b).

ANALOGY: A derivational analogy engine (Carbonell & Veloso, 1988; Veloso & Carbonell, 1989) that is able to replay entire solutions to similar past problems, calling the problem solver recursively to reduce any new subgoals brought about by known differences between the old and new problems. (This article presents this module.)

ALPINE: A multi-level abstraction planning capability (Knoblock, 1991). First, the axiomatized domain knowledge is divided into multiple abstraction layers based on an in-depth analysis of dependencies and interactions in the domain. Then, during problem solving, PRODIGY proceeds to build abstract solutions and refine them by adding details from the domain, solving new subgoals as they arise.

EXPERIMENTATION: A learning-by-experimentation module for refining domain knowledge that is incompletely or incorrectly specified (Carbonell & Gil, 1990). Experimentation is triggered when plan execution monitoring detects a divergence between internal expectations and external observations. The main focus of experimentation is to refine the factual domain knowledge, rather than the control knowledge.

Acknowledgments

The authors thank Craig Knoblock, Alicia Pérez, and Yolanda Gil for comments on this work and for helping revising this document, Daniel Borrajo for a major part of NOLIMIT's implementation, and the whole PRODIGY research group for helpful discussions. The authors also thank the reviewers for their comments and suggestions. 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 U.S. Government.

Notes

1. In Veloso et al. (1990), we show several examples in a complex logistics transportation domain.
2. These numbers are meaningful for their relative and not absolute values, as they were obtained using an initial version of the analogical reasoner code. Actual values using a much more optimized code are up to 12 times lower in absolute value.
3. This set is a sampled subset of the original set used by Minton (1988).
4. In fact, we currently have generated a more sophisticated similarity metric, also derived from the derivational trace, where better improvements are noticed (Veloso, 1991).

References

Allen, J., & Langley, P. (1990). Integrating memory and search in planning. In *Proceedings of the DARPA Workshop on Innovative Approaches to Planning, Scheduling, and Control* (pp. 301–312). San Diego: CA: Morgan Kaufmann.

Bareiss, R., & King, J.A. (1989). Similarity assessment in case-based reasoning. In *Proceedings of the Second Workshop on Case-Based Reasoning* (pp. 67–71). Pensacola, FL: Morgan Kaufmann.

Carbonell, J.G. (1983). Learning by analogy: Formulating and generalizing plans from past experience. In R.S. Michalski, J.G. Carbonell, & T.M. Mitchell (Eds.), *Machine learning, an artificial intelligence approach* (Vol. 1), pp. 137–162. Palo Alto, CA: Tioga Press.

Carbonell, J.G. (1986). Derivational analogy: A theory of reconstructive problem solving and expertise acquisition. In R.S. Michalski, J.G. Carbonell, & T.M. Mitchell (Eds.), *Machine learning, an artificial intelligence approach* (Vol. 2), pp. 371–392. San Mateo, CA: Morgan Kaufmann.

Carbonell, J.G., & Gil, Y. (1990). Learning by experimentation: The operator refinement method. In R.S. Michalski & Y. Kodratoff (Eds.), *Machine learning: An artificial intelligence approach*, (Vol. 3), pp. 191–213. Palo Alto, CA: Morgan Kaufmann.

Carbonell, J.G., & Veloso, M.M. (1988). Integrating derivational analogy into a general problem solving architecture. In *Proceedings of the First Workshop on Case-Based Reasoning* (pp. 104–124). Tampa, FL: Morgan Kaufmann.

Cheng, P.W., & Carbonell, J.G. (1986). Inducing iterative rules from experience: The FERMI system. In *Proceedings of AAAI-86* (pp. 490–495). Philadelphia, PA.

DeJong, G.F., & Mooney, R. (1986). Explanation-based learning: An alternative view. *Machine Learning*, *1*(2), 145–176.

Doyle, J. (1984). Expert systems without computers. *AI Magazine*, *5*(2), 59–63.

Drummond, M., & Currie, K. (1989). Goal ordering in partially ordered plans. In *Proceedings of IJCAI-89* (pp. 960–965). Detroit, MI.

Etzioni, O. (1990a). *A structural theory of explanation-based learning* (Technical Report CMU-CS-90-185). Ph.D. thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA.

Etzioni, O. (1990b). Why Prodigy/EBL works. In *Proceedings of AAAI-90* (pp. 916–922). Boston, MA.

Fikes, R.E., & Nilsson, N.J. (1971). Strips: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, *2*, 189–208.

Hammond, K. (1986). *Case-based planning: An integrated theory of planning, learning and memory.* Ph.D. thesis, Department of Computer Science, Yale University, New Haven, CT.

Harandi, M.T., & Bhansali, S. (1989). Program derivation using analogy. In *Proceedings of IJCAI-89* (pp. 389–394). Detroit, MI.

Hickman, A.K., & Larkin, J.H. (1990). Internal analogy: A model of transfer within problems. In *The 12th Annual Conference of The Cognitive Science Society* (pp. 53–60). Hillsdale, NJ: Lawrence Erlbaum Associates.

Hickman, A.K., Shell, P., & Carbonell, J.G. (1990). Internal analogy: Reducing search during problem solving. In C. Copetas (Ed.), *The Computer Science Research Review 1990.* The School of Computer Science, Carnegie Mellon University, Pittsburgh, PA.

Joseph, R.L. (1989). Graphical knowledge acquisition. In *Proceedings of the 4th Knowledge Acquisition For Knowledge-Based Systems Workshop*, Banff, Canada.

Kambhampati, S. (1989). *Flexible reuse and modification in hierarchical planning: A validation structure based approach*. Ph.D. thesis, Computer Vision Laboratory, Center for Automation Research, University of Maryland, College Park, MD.

Kedar-Cabelli, S. (1985). Purpose-directed analogy. In *Proceedings of the Seventh Annual Conference of the Cognitive Science Society* (pp. 150–159).

Knoblock, C.A. (1991). *Automatically generating abstractions for problem solving*. (Technical Report CMU-CS-91-120). Ph.D. thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA.

Kolodner, J. (1989). Judging which is the "best" case for a case-based reasoner. In *Proceedings of the Second Workshop on Case-Based Reasoning* (pp. 77–81). Pensacola, FL: Morgan Kaufmann.

Kolodner, J.L. (1984). *Retrieval and organization strategies in conceptual memory*. Hillsdale, NJ: Lawrence Erlbaum Associates.

Korf, R.E. (1985). Macro-operators: A weak method for learning. *Artificial Intelligence, 26*, 35–77.

Laird, J.E., Rosenbloom, P.S., & Newell, A. (1986). Chunking in soar: The anatomy of a general learning mechanism. *Machine Learning, 1*, 1–46.

Minton, S. (1985). Selectively generalizing plans for problem solving. In *Proceedings of AAAI-85* (pp. 596–599).

Minton, S. (1988). *Learning effective search control knowledge: An explanation-based approach*. (Technical Report CMU-CS-88-133). Ph.D. thesis, Computer Science Department, Carnegie Mellon University, Pittsburgh, PA.

Minton, A., Knoblock, C.A., Kuokka, D.R., Gil, Y., Joseph, R.L., & Carbonell, J.G. (1989). PRODIGY 2.0: The manual and tutorial (Technical Report CMU-CS-89-146). School of Computer Science, Carnegie Mellon University, Pittsburgh, PA.

Mitchell, T.M., Keller, R.M., & Kedar-Cabelli, S.T. (1986). Explanation-based generalization: A unifying view. *Machine Learning, 1(1)*, 47–80.

Mitchell, T.M., Utgoff, P.E., & Banerji, R. (1983). Learning by experimentation: Acquiring and refining problem-solving heuristics. In R.S. Michalski, J.G., J.G. Carbonell, & T.M. Mitchell (Eds.), *Machine learning, an artificial intelligence approach* (pp. 163–190), Palo Alto, CA: Tioga Press.

Newell, A. (1980). Physical symbol systems. *Cognitive Science, 4(2)*, 135–184.

Pazzani, M. (1990). *Creating a memory of causal relationships: An integration of empirical and explanation-based learning methods*. Hillsdale, NJ: Lawrence Erlbaum Associates.

Polya, G. (1945). *How to solve it*. Princeton, NJ: Princeton University Press.

Porter, B., Bareiss, R., & Holte, R. (1989). Knowledge acquisition and heuristic classification in weak-theory domains (Technical Report AI-TR-88-96). Department of Computer Science, University of Texas at Austin.

Riesbeck, C.K., & Schank, R.C. (1989). *Inside case-based reasoning*. Hillsdale, NJ: Lawrence Erlbaum Associates.

Sacerdoti, E.D. (1975). The nonlinear nature of plans. In *Proceedings of IJCAI-75* (pp. 206–213). Tbilisi, USSR.

Schank, R.C. (1982). *Dynamic memory*. Cambridge: Cambridge University Press.

Schank, R.C. (1983). The current state of AI: One man's opinion. *Artificial Intelligence Magazine, 4(1)*, 1–8.

Shell, P., & Carbonell, J.G. (1989). FRuleKit: A frame-based production system. User's manual. Internal report, Department of Computer Science, Carnegie Mellon University, Pittsburgh, PA.

Simpson, R.L. (1985). *A computer model of case-based reasoning in problem solving: An investigation in the domain of dispute mediation*. Ph.D. thesis, School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA.

Sussman, G.J. (1973). A computational model of skill acquisition (Technical Report AI-TR-297). Artificial Intelligence Laboratory, MIT, Cambridge, MA.

Sycara, E.P. (1987). *Resolving adversarial conflicts: An approach to integrating case-based and analytic methods*. Ph.D. thesis, School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA.

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

Veloso, M.M. (1991). Replaying multiple cases in analogical problem solving (Technical Report forthcoming). School of Computer Science, Carnegie Mellon University, Pittsburgh, PA.

Veloso, M.M., & Carbonell, J.G. (1989). Learning analogies by analogy—The closed loop of memory organization and problem solving. In *Proceedings of the Second Workshop on Case-Based Reasoning* (pp. 153–158). Pensacola, FL: Morgan Kaufmann.

Veloso, M.M., & Carbonell, J.G. (1990). Integrating analogy into a general problem-solving architecture. In M. Zemankova & Z. Ras (Eds.), *Intelligent systems* (pp. 29–51). Chicester, England: Ellis Horwood.

Veloso, M.M., & Carbonell, J.G. (1991a). Learning by analogical replay in PRODIGY: First results. In *Proceedings of the European Working Session on Learning* (pp. 375-390). Porto, Portugal: Springer-Verlag.

Veloso, M.M., & Carbonell, J.G. (1991b). Variable-precision case retrieval in analogical problem solving. In *Proceedings of the 1991 DARPA Workshop on Case-Based Reasoning* (pp. 93-106). Washington, DC: Morgan Kaufmann.

Veloso, M.M., Pérez, M.A., & Carbonell, J.G. (1990). Nonlinear planning with parallel resource allocation. In *Proceedings of the DARPA Workshop on Innovative Approaches to Planning, Scheduling, and Control* (pp. 207-212). San Diego, CA: Morgan Kaufmann.

Waldinger, R. (1981). Achieving several goals simultaneously. In B.L. Webber & N.J. Nilsson (Eds.), *Readings in Artificial Intelligence* (pp. 250-271). Los Altos, CA: Morgan Kaufmann.

Wilkins, D.E. (1989). Can AI planners solve practical problems? (Technical Note 468R). SRI International. Stanford, CA.

Received September 12, 1990
Accepted October 17, 1991
Final Manuscript May 7, 1992