# Integrating derivational analogy into a general problem solving architecture*

Jaime Carbonell
Manuela Veloso
Computer Science Department
Carnegie Mellon University
Pittsburgh, PA 15213

March 1988

**Abstract**

True expertise requires knowledge-intensive reasoning as a primary mental activity and a fallback to general problem solving in order to bridge gaps in the domain knowledge or to handle otherwise unexpected situations. This paper unites derivational analogy, a form of case-based reconstructive reasoning driven by rich episodic knowledge, with a general problem solving and planning architecture. The primary learning mechanism consists of extracting and organizing derivational traces of search-intensive problem solving episodes augmented with (and indexed by) explicit justification paths of success and failure conditions. This store of new episodic case knowledge drives the derivational analogy machinery in subsequent reasoning about problems that share substantial parts of the justification structure. The method is being implemented in the PRODIGY problem solving system and demonstrated in the domain of linear algebra.

*This research was supported in part by ONR grants N00014-79-C-0661 and N0014-82-C-50767, DARPA contract number F33615-84-K-1520, and NASA contract NCC 2-463. The views and conclusions contained in this document are those of the authors alone and should not be interpreted as representing the official policies, expressed or implied, of the U.S. Government, the Defense Advanced Research Projects Agency or NASA.

# Contents

# 1 Introduction

The central dream of many AI researchers, present authors included, is the construction of a general purpose learning and reasoning system that given basic axiomatic knowledge of a chosen domain lifts itself up to expert performance through practice solving increasingly complex problems. A small step toward that dream is the development of methods for encapsulating past experience into richly indexed episodic structures that help to avoid past mistakes and enable the reasoner to repeat and generalize past success. In general, we wish to compile past search-intensive behavior into increasingly knowledge-intensive reasoning in the chosen domain, all the while retaining the safety net of the general problem solver.

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 experience over time in any given domain. Inspired by this 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 handcoded at the outset, to machine learning approaches, where incrementally accumulated experience is stored, processed and compiled into generalized reusable "chunks".

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:

- Memorized actual instance solutions annotated with intermediate problem solving states (such as subgoal trees, causes of intermediate planning failure, justifications for each selected planning step, etc.). These are used in analogical reasoning [2, 3] and case-based reasoning [7, 12, 14, 31, 32] to reduce search by using the solutions of similar past problems to guide the planner in constructing the solution to the new problem.

- Reformulated left-hand-sides of operators and inference rules, where the new left-hand-sides are stated in terms of "operational" or initial-state conditions so as to facilitate their selection and application. This is one typical output of explanation-based learning systems [6, 20, 25, 26].

- Explicit control rules (or meta rules) that guide the selection of domain-level subgoals, operators or inference rules in the planning process. These may also be generated by the explanation-based learning process when the basic architecture of the problem solver itself is axiomatized and available to the learning module, along with the domain theory [20, 21].

- Macro-operators composed of sequences of domain-level operators which, if applicable, take "large steps" in the problem space and thereby reduce search [1, 5, 8, 15, 19]. In essence, intermediate decisions corresponding to steps internal to each macro-op are bypassed, in the construction of a parameterized fragment of the proven solution path into a macro-op.

- Generalized "chunking" of all decisions taken by the problem solver, including goal selection, operator selection and other impasse-driven decisions that required search. The output of these internal decisions are at once compiled into new chunks by a background reflex process and become immediately available to the problem solver's recognition process [17, 28].

All of these methods seek to compile existing domain knowledge into more effective form by combining it with search control knowledge acquired through incremental practice. In essence, the idea is to transform book knowledge into practical knowledge that can be applied much more readily, occasionally compromising generality for efficiency of application, but retaining the initial knowledge as a safety net.

Analogy and case-based reasoning are two sides of the same proverbial coin: both rely on encapsulating episodic knowledge to guide complex problem solving, but the former emphasizes the process of modifying, adapting and verifying past derivations (cases), whereas the latter emphasizes the organization, hierarchy indexing and retrieval of case memory. This paper focuses on the smooth integration of derivational analogy into a general purpose problem solving and planning architecture, the PRODIGY system [22]. Whereas both derivational analogy and generalized operator-based planning have been developed and reported previously, never have the twain been totally integrated into a functional and demonstratable system. The integration requires that the general problem solver be able to introspect into its internal decision cycle, recording the justifications for each decision (what subgoal to pursue next, what operator to apply, and what objects to select as variable bindings for each operator). These justifications augment the solution trace and are used both for indexing and to guide the future reconstruction of the solution for subsequent problem solving situations where equivalent justifications hold true.

We now present the overall structure of a general system and we motivate the need for justifications in case-based learning through derivational analogy.

# 2 The overall picture

The work reported here integrates case-based analogical reasoning into PRODIGY, a general search-intensive problem solver. Episodic knowledge, when applicable, is used to bypass classical search. The knowledge sources in PRODIGY consist of:

- static domain knowledge - a conceptualization of the underlying predicates and objects in the domain in axiomatic form.
- working domain knowledge - a description of the legal actions in the domain, namely in terms of a set of operators, inference rules, and heuristic control rules.
- solvable classes of problems - a categorization of general problems capable of being solved using the static and working domain knowledge.
- episodic knowledge library - a collection of cases and generalization thereof to draw from in derivational analogy, currently under continued development.

These three basic (and one episodic) knowledge sources provide a consistent store of information to be used by the learning module.

The non-episodic portion of PRODIGY's domain knowledge is specified in terms of a set of operators, inference rules, and control rules. The episodic knowledge is encoded as a set of derivational traces with justifications links among the reasoning steps so that the same reasoning strategy can be reconstructed or modified to solve related problems. Moreover, in order to draw meaningful analogies for problem solving, we believe it is necessary to make explicit the underlying reasons for the presence of certain predicates in the non-episodic knowledge as well. That is, a certain amount of higher level knowledge is required so that the problem solver can explain to itself why a retrieved case was (or was not) successful, and thereby know how to modify it to the requirements of the new problem.

This kind of fundamental knowledge is useful even for well known toy domains, let alone areas of realistic expertise. As an example, consider the well known operator PUT-DOWN from the blocks world [8] (see Figure 1).

```
(PUT-DOWN
 (params (<ob>))
 (preconds
   (and
     (object <ob>)
     (holding <ob>)))
 (effects
   ((del (holding <ob>))
    (add (clear <ob>))
    (add (on-table <ob>)))))
```

Figure 1: The PUT-DOWN operator from the blocks world

We can come up with several justifications at different levels of abstraction on why this operator achieves the effects. These justifications can range from specific properties of the object, e.g. being a solid block, to general assumptions, e.g. the presence of Earth gravity. Testing these assumptions may not be relevant at all in the problems that a system was designed to solve. However this implicit information might be crucial when the system is asked to solve problems from a slightly different analog domain, and these causal links to general knowledge are particularly relevant for learning by analogy between far-apart domains or between far-apart classes of problems. As an example, imagine that the system is asked to learn how to solve problems of manipulating blocks in an orbiting space station. (Without gravity, PUT-DOWN will not achieve its goal, as the block will not drop from the hand, nor stay where placed.) However, stacking wooden blocks or plastic ones, or doing so on Tuesday rather than Thursday are probably immaterial differences. In brief, causal justification links bypass the the old frame problem [24] for analogical reasoning by explicitly encoding the limits of justifiability for the initial solution. Other domains within those limits ought to permit direct transfer, but domains outside those limits will require at least local replanning to establish the same effects by other means.

We are aware that unexpected situations may occur (consider a solid block of ice that melts when you hold it for a while...) and that even an elaborated initial set of justifications could prove incomplete. Here we advocate experimentation [4] as the learning tool to use in

case the system runs out of consistent alternatives. Experimentation will be used not only to refine incomplete and incorrect knowledge but also to refine the internal justifications. This approach allows the system to be flexible enough so as not to require an exaggerated initial effort in getting the full justifications at every level of abstraction.

# 3 The basic analogical replay method

We now present the general method of analogical replay process and case learning mechanism in PRODIGY.

- A teacher introduces a problem and guides the planner through its different alternatives while constructing the solution. This process produces the initial entries into the case library. Subsequence sources for analogy can come from the case library (teacher or system-generated solutions). This initial guidance provided by the teacher is done at three different levels:

  1. choice of the next subgoal to work on from the current set of goals,
  2. choice of the operator or inference rules to apply to achieve a particular goal,
  3. choice of particular bindings for the variables.

  The key fact for this step is that the teacher presents *justifications* for her/his choices. In section 6 we formalize these justifications and in section 7 we present how they are used by the learner. The problem and the solution path are stored together with the related justifications.

- Upon demand to solve a new problem, PRODIGY considers its exiting case library of previously solved problems. The generation of the new solution is guided by the justifications that affect the steps of past solution paths. Conservatism (minimal change of the retrieved solution) is an emergent property of this method, as search for alternative methods of solving a problem is only triggered when an exiting method fails (i.e., no analogy is found, or there is no way to repair a failed justification in the retrieved case).

- The new solution is stored in the episodic case library, along with its complete set of justifications, and used as a candidate analog for future problem solving. Methods for producing packed encoding for solution derivations to related problems that share substantial substructure offer promise to keep memory requirements if not modest, at least tractable.

- Generalization methods apply to collections of derivational traces (i.e. cases) with the same analog parentage, in order to produce a generalized plan for such commonly recurring problems, whereas those problems with few if any analogs continue to retain their full derivational traces as ground-instance cases.

# 4 An example from elementary linear algebra

Case-based or analogical reasoning works equally well whether one focuses on expert system domains (such as factory scheduling), robotic planning problems, toy domains (such as the blocks world), mundane domains (selection of recepies) or mathematical reasoning. Here we have selected the latter one, in part to be different from other projects. In particular, we focus on the familiar domain of linear algebra, such as Gaussian elimination, Jordan elimination, coping with truncation errors, solving systems of $n$ equations with $n$ unknowns, inversion of matrices, etc. [10].

## 4.1 The initial state representation

To start the problem solving process, let the initial state be any well defined matrix, and let final (desired) state be a particular configuration of that matrix. As a concrete example, consider the following system of equations shown using matrix representation as $Ax = b$,

$$\underbrace{\begin{bmatrix} 2 & -1 & 3 \\ 1 & 2 & -2 \\ 4 & 1 & 2 \end{bmatrix}}_{A} \underbrace{\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}}_{x} = \underbrace{\begin{bmatrix} 2 \\ 1 \\ -3 \end{bmatrix}}_{b}$$

The initial state given to PRODIGY that represents this problem has two predicates:

**mat** with attributes *row*, *column*, and *value*; the predicate (mat 2 3 -2) is true if the matrix element $A(2, 3)$ has the value $-2$.

**dimension** with attributes the number of *rows* and *columns*.

For the example given above, the initial state is the matrix $[A|b]$ corresponding to the following set of formulas:

```
(dimension 3 4)
(mat 1 1 2)     (mat 1 2 -1)  (mat 1 3 3)   (mat 1 4 2)
(mat 2 1 1)     (mat 2 2 2)   (mat 2 3 -2)  (mat 2 4 1)
(mat 3 1 4)     (mat 3 2 1)   (mat 3 3 2)   (mat 3 4 -3).
```

To perform Gaussian elimination in this matrix the goal is stated in terms of a predicate *(triangular upper)* that must be true in the final state (matrix).

## 4.2 The original search tree

When there is no previous case for guidance, PRODIGY generates an augmented *and-or* search tree. If there is a previous case its derivation consists of a successful path through the (earlier) search tree, and analogical problem solving consists of modifying that path (expanding small portions of the search tree) when justifications required in the retrieved derivation are not valid for the new problem situation.

We should note an important difference between PRODIGY and previous problem solvers (such as STRIPS [8], NOAH [30], NASL [18], etc.), where all variables were assumed to be existentially quantified. PRODIGY also permits universally quantified variables as well (with the quantifier in the precondition part of an operator [23]). These are exploited to the hilt in large domains containing regular structure. We introduce a new kind of node in the tree, the *forall-node*. This node has three fields: the list of bound variables (V), the generator (G), and the subexpression (E). As an example, consider the operator TRIANGULIZE in Figure 4. Node 5 in Figure 5 is a forall-node. It is expanded into its children nodes 6, 7, 8, and 9. In the process of constructing a solution by analogy with a past solution path, the links from specific steps to the related *macro* forall-node are taken into consideration. These steps are therefore expanded according to the new state.

## 4.3 Static domain knowledge

The static knowledge consists of a set of axioms or primitives on which the operators are act. Figure 2, for instance, defines the notion of an upper triangular matrix, as a small illustrative example. We assume that the mentioned concepts, such as *table-of, objects, vectors, cprod* (cartesian product), etc, are also present in the system.

```
(FRAME matrix
  (is-a data-structure)
  (definition
    (def1 (table-of objects))
    (def2 (ordered-set-of vectors))
    (def3 (map (cprod setA setB) objects)))
  (characteristics
    (dimensions (number 2)
                (name row column))
    (diagonal (set (elt (row i column j))
      (such-that (equal i j)))))
```

```
(FRAME upper-triangular
  (is-a triangular)
  (definition
    (def1 (forall (elt (row i) (column j))
          (such-that (greater-than i j))
          (elt-value 0)))
    (def2 (forall (elt (row i) (column j))
          (such-that (below diagonal i j))
          (elt-value 0))))
```

Figure 2: Static knowledge - a snapshot

## 4.4 Working domain knowledge

We introduce below a subset of the base-level operators currently used in the linear algebra domain. These operators specify the legal operations to perform on a system of linear equations. The name of each predicate is an abbreviation of its function; for instance the operator UPDATE-ELT-SC-ADD stands for *update an element by scaling and adding*. The goal of updating an element (*clearedelt*) on the working row *wrow* and working column *wcol* to the new value *val* is achieved by applying this operator UPDATE-ELT-SC-ADD. Note that in order to change the value of a coefficient in a system of linear equations we affect the whole equation (row of the matrix), so that whereas the application of an operator may achieve a desired goal, it will also have multiple, possibly undesired, side-effects. Such is true for most operators, and the name of the game in linear algebra is to protect desired portions of the state while achieving necessary changes elsewhere. The result of applying this

operator is to scale a pivot row in the matrix (*prow*) with the constant $c$ and substituting the working row *wrow* by the sum of the scaled *prow* and *wrow*.

```
; Updates an element by scaling and adding
; the pivot row with the working row
(UPDATE-ELT-SC-ADD
 (params (<wrow> <wcol> <val>))
 (preconds
   (and
   (rowpivotfor <prow> <wrow> <wcol> <val>)
   (const <wrow> <wcol> <val> <c>)
   (scaled&added <prow> <c> <wrow>)))
 (effects
   ((add (clearedelt <wrow> <wcol> <val>)))))
```

```
; Scales the pivot row and adds
; it to the working row
(SCALE-ADD
 (params (<prow> <wrow> <c>))
 (preconds
   (and
   (dimension <n> <m>)
   (forall (<col>) (range-f <col> 1 <m>)
      (and
      (mat <prow> <col> <pval>)
      (mat <wrow> <col> <val>)
      (mult-f <pval> <c> <mval>)
      (changed <wrow> <col> <val> <nval>)))))
 (effects
   ((add (scaled&added <prow> <c> <wrow>)))))
```

```
; Selects a row for pivoting
; different from the working row
(CHOOSE-ROW-PIVOT
 (params (<prow> <wrow>))
 (preconds
   (and
   (row <prow>)
   (diff-f <prow> <wrow>)))
 (effects
   ((add (rowpivotfor <prow> <wrow> <wcol> <val>)))))
```

```
; Substitutes an element by a new one
(PERFORM-UPDATE
 (params (<row> <col> <old> <new>))
 (preconds
   (mat <row> <col> <old>))
 (effects
   ((add (changed <row> <col> <old> <new>))
    (del (mat <row> <col> <old>))
    (add (mat <row> <col> <new>)))))
```

Additional operators (by name only) consist of:

**UPDATE-ELT-SC** Updates an element by scaling its row.

**UPDATE-ELT-ADD** Updates an element by adding two rows.

**ADD-ROWS** Adds two rows and substitutes one by the obtained sum.

**SCALE-ROW** Multiplies a row by a constant and substitutes the row by the obtained result.

**SELECT-WROW** Selects a working row.

**CREATE-CONST** Creates a multiplicative constant.

### 4.5 Solvable classes of problems

The problems we have been dealing with lay within the area of elementary linear algebra. We show in Figure 3 a simplified hierarchy of the classes of solvable problems. The current set of operators is directed to solving a system of linear equations. The actions specified are however general enough to be applied to other classes of problems.

Figure 3: Solvable classes of problems

## 5 Introducing justifications

Different kinds of justifications are required by the derivational analogy process to insure applicability (or to guide modification) of the retrieved derivational trace. This section introduces and motivates each type of causal justification link.

### 5.1 Justifications at the operator level

An operator or inference rule in PRODIGY is encoded as a data structure with three primary fields, as we saw in the examples of the previous sections: parameters (params), preconditions (preconds), and effects (effects). We add the additional field for justifications (justifs). Justifications are both links to external domain knowledge and to other operators on which the present operator depends. These links may be viewed as higher-order (immutable) preconditions, and come in three varieties:

**assumes** expects a list of operators that are assumed to be also valid in this domain. For instance, unlocking a door is useful to exit a room only if opening that door is also a valid operation.

**refers-to** points to static domain knowledge underlying the meaning of this operator. For instance, robotic manipulation operators refer to the definition gravity (including the value of the gravity constant, and the direction of the gravity vector).

**prob-class** enumerates the class of problems where this operator is assumed to be applicable. For instance, matrix manipulation operators list mathematical (and perhaps other) domains, but not selection of recipes for low cholesterol intake.

```
(TRIANGULIZE
 (params ())
 (preconds
   (and
   (dimension <n> <m>)
   (pred-f <n1> <n>)
   (forall (<k>) (range-f <k> 1 <n1>)
   (and (workrow <r> <k>)
   (forall (<col>) (range-f <col> <st> <k>)
      (clearedelt <r> <col> 0))))))
 (effects
   ((add (triangular upper))))
 (justifs
   ((assumes INTERCHANGE-ROWS)
    (refers-to ((upper-triangular (definition def1)))
    (probl-class ((system-of-equations))))))
```

Figure 4 : The operator TRIANGULIZE

As an example we show the operator TRIANGULIZE in Figure 4. This operator refers to the definition of an upper triangular matrix named def1 (see Figure 2). The state achieved by this operator can be transformed into the referred definition by applying the operator INTERCHANGE-ROWS. This is the meaning underlying the *assumes* property.

## 5.2 Justifications at the goal choice

While the planner runs, there is a current (partially ordered) set of active goals, rather than a totally-ordered goal stack typical of more limited linear planners. Since the set can be dynamically (re)ordered, PRODIGY can interleave subgoals and produce non-linear plans. When being instructed by an external teacher (or while blindly searching for a solution under weak heuristic guidance) justifications for each goal choice are recorded. The planner itself is able to come out with some of these justifications by exploring the search space, or by post-facto explanation based learning methods [20]. Five different types of goal justifications are presented:

**unique** this means that this was the only available alternative.

**stack** this means that this goal was the latest to be posted, and in the absence of any reason to the contrary, becomes the default selection.

**arbitrary** this goal was picked up arbitrarily from the goal set.

**protect** enumerates a list of previously achieved subgoals that the system can assure will not to be destroyed by working on this subgoal. In essence, a goal selection may be dictated by protection criteria, even though in isolation of the problem solving context there may have been a locally superior selection.

**necess-before** lists subgoals that have to be worked out after the current subgoal in order for these latter to be achievable; in this way we capture the concept of serializable goals [16].

## 5.3 Justifications at the choice of alternatives among operators

There may be several operators that can be potentially applied when working out a goal. The justification for operator choice, and for its successful application is recorded using the following three predicates:

**unique** no more operators were available; the system may learn in the future alternative operators; therefore when solving a problem by analogy with this current one, the information that this chosen operator was unique at that time is taken into consideration.

**arbitrary** expects a set of operators as an attribute; its meaning is that it was an arbitrary choice among the specified set of operators.

**failure** this predicate lists the set of operators known to have failed and the reason why they failed; we store either the related end condition responsible for the failure or the fact that this operator lead the system into a *loop*; in this case we store a reference to the subgoal on which it looped. Thus, an operator may be chosen because alternatives that appeared better are known to fail for recorded (well defined) reasons. In the subsequent analogical reconstruction, if these failure reasons hold, the same selection is made, and if not, the potentially better operators can be reconsidered.

Note that the justifications *arbitrary* and *failure* are not mutually exclusive. In fact the union of the two sets of operators mentioned should generate the total set of alternative operators. We are planning to introduce other justifications for the use of an operator in terms of statistical analysis of its frequency of successful application. Consider the situation where an operator was chosen arbitrarily. Consider further that in the sequence of solving similar problems, we are faced always with the same set of operators and we get a continuous success by choosing the same operator *arbitrarily* without exploring the alternative operators. We claim that the justification for the use of this operator is becoming *stronger* than simply an arbitrary choice. We will therefore introduce a measure of its frequency of success related to a specific class of solvable problems.

## 5.4 Justifications for the bindings of variables

The planner has the choice of applying an operator with a different set of bindings for its variables. We approach separately the choice of bindings for a particular operator and the choice of an operator among a set of alternative operators (see section 5.3). We consider that any specific constraint that might exist for the choice of bindings is encoded as a precondition of the operator. Therefore the justifications show only the set of bindings that have failed and the set of bindings not explored. The union of these two sets and the chosen binding is

the set of all possible bindings for that operator. In order to use these justifications while learning by analogy, the system should be able to reason about the set of failed bindings and generalize from the reasons of failure. We postpone this point for future research.

## 5.5 An example of a justified solution path

Consider the example of the plan generated to perform Gaussian elimination on a matrix representing a system of equations. In fact we deal with a generalized version of Gaussian elimination where we admit that rows can be interchangeable (see the operator TRIANGULIZE in Figure 4). Furthermore assume that we also want the system matrix to have diagonal unit elements. On Figure 5 we sketched an initial piece of the search tree for a system of three unknowns. We refer to the goals by the corresponding numbered nodes. PRODIGY is given initially goals 1 and 2. Below we introduce the justifications at the goal choice. Note that the legal actions on the matrix affect the elements on an entire row - the element we wish to change, and by side-effect all the other elements as well. This is what is encoded as the justification (necess-bef 2) at node 0, for example, when node 1 is chosen to be worked out before node 2. The predicate (workrow r k) on node 5 is true if row $r$ was chosen to have $k$ many null elements in it. Note that the values of $r$ are different for nodes 8 and 9. In fact the operator SELECT-WROW is responsible for achieving the subgoal (workrow r k) and its preconditions guarantee that $r$ is bound to different values for different $k$ values.

| At node | Goal selected | Selected among goals | Justification |
|---|---|---|---|
| 0 | 1 | 1,2 | (necess-bef 2) |
| 1 | 3 | 3,4,5,2 | (necess-bef 4 binding) |
| | 4 | 4,5,2 | (necess-bef 5 binding) |
| | 5 | 5,2 | (stack) |
| 5 | 7 | 6,7,8,9,2 | (necess-bef 9 binding) |
| | | | (arbitrary 6,8) |
| | 9 | 6,8,9,2 | (arbitrary 6,8) |
| | 11 | 6,8,11,12,2 | (arbitrary 6,8,12) |
| | 6 | 6,8,12,2 | (necess-bef 8 binding) |
| | | | (protect 10 11) |
| | 8 | 8,12,2 | (necess-bef 12) |
| | 10 | 10,12,2 | (necess-bef 12) |
| | | | (protect 11) |

...

The solution path also mentions the operator choices. As an example, note that TRIANGULIZE is chosen because it is the *unique* relevant operator. On the other hand there are three candidate operators to clear an element: UPDATE-ELT-SC, UPDATE-ELT-ADD, and UPDATE-ELT-SC-ADD. This last operator is chosen as the only successful one (the other fail).

Figure 5: A piece of the search tree for generalized Gaussian elimination

# 6 Analogical reconstruction and learning: The algorithm

Learning by analogy involves the recognition of a previously solved similar problem, the derivation of the new solution by reconstruction of the old one, and finally the capability of generalizing individual solutions achieved to higher levels of abstraction. As stated in the introduction, our current work in PRODIGY addresses the problem of rational reconstruction of a solution from the derivational trace of an earlier similar case. We do not address here the equally important problem of indexing and retrieving from an extensive case library. The retrieved case re-establishes the earlier problem solving context, and the process of reconstruction, essentially replays the solution path (without regenerating a potentially vast search tree with innumerable unsuccessful side branches).

The stored derivational trace contains the full justifications for all the steps required to derive the old solution. We introduced in the previous section these justifications at the different points in the decision cycle of the reasoner. We now present how the system intelligently uses the past reasoning process to reconstruct a solution for a new problem without going through a painful blind planning process again. One can envision two approaches:

**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 when to consider alternatives of arbitrary decisions and to re-explore failed paths if their causes for failure are not present in the new situation.

Presently we are implementing only the satisficing approach in full, 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. At goal choice, operator choice, or variable bindings choice, PRODIGY tests the validity of all the justifications. 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 to establish 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 pieces, it would learn that these were just as good if not better than rectangular blocks for the purpose of stacking tall towers. Thus, the justification would be generalized for future reference.

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

The second case occurs for example, when the assumptions for the applicability of an operator fail. The system may then try 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 this 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 the general this fails, and a totally different approach is required (at least for the subgoal in question).

The experimentation case can be blind 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 possible re-establishments of these looser conditions via steps 1 or 2 above.

## 6.1 Some learning examples

The process of reconstructing a new solution may lead to the learning of new operators. Consider the example in the domain of linear algebra presented above. Further consider the step of the solution path for solving a system of linear equations that involves the selection of a pivot row. Suppose in the previous solution the justification for the choice of a particular pivot row is just *arbitrary*. Repeating the same choice on a different matrix may lead to choosing a zero element for pivot element. The solution fails further down in the selection of a multiplicative constant; division by zero is attempted. At this point, the system takes the failed precondition (divisor different from zero in the division operator) and moves it back up the derivational trace (in an "on-demand" version of weakest precondition propagation) to point where the binding occurred for the variable in question, in this case in the operator for choosing a pivot row. A new, corrected version of the operator is learned by copying the CHOOSE-ROW-PIVOT operator and adding the precondition that failed.

In the learning process the new knowledge acquired is considered *conjectured* until a formal proof for its truth may be constructed. The example just described shows an operator that was learned and whose truth value can be proved to be true immediately for scalar arithmetic.

In a more complex variant of this example, PRODIGY encounters difficulties after selecting a very small pivot element (leading to truncation errors after repeated divisions). This

time the added condition that the selected pivot be within several orders of magnitude as the other elements in the column remains a conjecture - and in fact is only valid as a function of the precision with which the scalar calculations are made.

Other learning examples deal with the notion of symmetry and the resultant ability to perform Jordan elimination [10] given the fact that the system knows how to perform Gaussian elimination. We are working on testing the analogical methods to arrive at a plan for generating a lower triangular matrix from the plan to generate an upper triangular one. We are also working on generating a diagonal matrix from the plan to generate a triangular one. And, finally we are investigating matrix inversion, again derived from earlier experience (e.g. the diagonalization trace). In other words, we are investigating the use of derivational analogy to rediscover algorithms in linear algebra, given structurally similar algorithms from this domain. At present it is too early to report on the success (or lack thereof) of our method as applied to this task.

# 7 Concluding remarks

In this paper, we have reported on the total integration of derivational analogy into the general purpose PRODIGY problem solving architecture, including the operator based domain knowledge encoding in PRODIGY, the causal justification structure encapsulating episodically-required domain expertise, the use of this knowledge to guide problem solving behavior, experimentation to test the validity of conjectured justifications, and the ability to fall back upon the general problem solver either when justifications fail or when episodic expertise is lacking.

The primary research thrusts we envision for the near future include:

- Demonstrate cross-domain generality: In addition to linear algebra, PRODIGY solves problems in domains as diverse as robotics root planning and factory scheduling. We expect that the derivational analogy method can function equally well in any of these domains.

- Unify derivational analogy with explanation based learning (EBL): PRODIGY supports a general EBL implementation [20], and derivational analogy is an alternate method to EBL for compiling episodic acquired knowledge. Therefore, PRODIGY provides a common framework for comparative analysis and possible integration.

- Integrate derivational analogy with abstraction planning and domain-level experimentation: Parallel developments in PRODIGY have produced a multi-level dynamic abstraction planner [13] and a proactive experimentation method to refine empirically a partial domain theory [4]. In principal, the power of both these methods could be augmented significantly by judicious exploitation of derivational analogy, but in practice such integration is yet to be realized.

As Artificial Intelligence systems grow to face increasingly complex task domains, they must acquire and compile domain expertise incrementally through experience. We have provided a first step in this direction via the integration of derivational analogy into a

general problem solver, and expect to see many subsequent developments in this promising direction.

# References


[1] Anderson, J. R. (1983). *The Architecture of Cognition*, Harvard University Press, Cambridge, Mass.

[2] Carbonell, J. G. (1983). Learning by Analogy: Formulating and Generalizing Plans from Past Experience. In R. S. Michalski, J. G. Carbonell and T. M. Mitchell (Eds.) *Machine Learning, An Artificial Intelligence Approach*, Tioga Press, Palo Alto, CA.

[3] Carbonell, J. G. (1986). Derivational Analogy: A Theory of Reconstructive Problem Solving and Expertise Acquisition. In R. S. Michalski, J. G. Carbonell, and T. M. Mitchell (Eds.). *Machine Learning, An Artificial Intelligence Approach, Volume II*, Morgan Kaufmann.

[4] Carbonell, J. G. and Gil, Y. (1987). Learning Domain Knowledge by Experimentation. *Proceedings of the Fourth International Machine Learning Workshop*, Irvine, CA.

[5] Cheng, P. W. and Carbonell, J. G. (1986). Inducing Iterative Rules from Experience: The FERMI Experiment. *Proceedings of AAAI-86*.

[6] DeJong, G. F. and Mooney, R. (1986). Explanation-Based Learning: An Alternative View. *Machine Learning Journal*, 1, 2.

[7] Doyle, J. (1984). Expert Systems Without Computers. *AI Magazine*, 5, 2, 59-63.

[8] Fikes, R. E. and Nilsson, N. J. (1971). STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving. *Artificial Intelligence*, 2, 189-208.

[9] Gentner, D. (1980). The Structure of Analogical Models in Science. *Bolt Beranek and Newman*, 4451.

[10] Golub, G. H. and Van Loan, C. F. (1985). *Matrix Computations*, The Johns Hopkins University Press, MD.

[11] Greiner, R. (1985). Learning by Understanding Analogies. *Proceedings of the Third International Machine Learning Workshop*, 50-52, Skytop, PA.

[12] Hammond, K. (1986). Case-based Planning: An Integrated Theory of Planning, Learning and Memory. *Ph.D. Thesis*, Yale University.

[13] Knoblock, C. A. (1988). Automatic Generation of Abstractions for Planning. *AAAI-88*, submitted.

[14] Kolodner, J. L. (1980). Retrieval and Organizational Strategies in Conceptual Memory: A Computer Model. *Ph.D. Thesis*, Yale University.

[15] Korf, R. E. (1985). Macro-operators: A Weak Method for Learning. *Artificial Intelligence*, 26, 35-77.

[16] Korf R. E. (1987). Planning as Search: A Quantitative Approach. *Artificial Intelligence*, September, 33, 1, 65-68.

[17] Laird, J. E., Rosenbloom, P. S. and Newell, A. (1986). Chunking in SOAR: The Anatomy of a General Learning Mechanism. *Machine Learning*, 1, 11-46.

[18] McDermott, D. V. (1978). Planning and Acting. *Cognitive Science*, 2, 2, 71-109.

[19] Minton, S. (1985). Selectively Generalizing Plans for Problem Solving. *Proceedings of AAAI-85*, 596-599.

[20] Minton, S. (1988). Learning Effective Search Control Knowledge: An Explanation-Based Approach. *Ph.D. Thesis*, Computer Science Department, Carnegie Mellon University.

[21] Minton, S. and Carbonell, J.G. (1987). Strategies for Learning Search Control Rules: An Explanation-Based Approach. *Proceedings of IJCAI-87*, Milan, Italy.

[22] Minton, S., Carbonell J. G., Knoblock C. A., Kuokka, D. R., Etzioni, O., and Gil, Y. (1988). Explanation-Based Learning: Optimizing Problem Solving Performance through Experience. *Paradigms for Machine Learning*, forthcoming.

[23] Minton, S., Knoblock, C. A., Kuokka, D. R., Gil, Y., and Carbonell, J. G. (1988). PRODIGY 1.0: The Manual and Tutorial. *Technical Report*, Computer Science Department, Carnegie Mellon University, forthcoming.

[24] McCarthy, J. and Hayes, P. J. (1969). Some Philosophical Problems from the Standpoint of Artificial Intelligence. In B. Meltzer and D. Michie (Eds.) *Machine Intelligence 4*, Edinburgh University Press, Edinburgh.

[25] Mitchell, T. M., Utgoff, P. E. and Banerji, R. B. (1983). Learning by Experimentation: Acquiring and Refining Problem-Solving Heuristics. In R. S. Michalski, J. G. Carbonell and T. M. Mitchell (Eds.) *Machine Learning, An Artificial Intelligence Approach*, Tioga Press, Palo Alto, CA.

[26] Mitchell, T. M., Keller, R. M. and Kedar-Cabelli, S. T. (1986). Explanation-Based Learning: A Unifying View. *Machine Learning*, 1, 47-80.

[27] Mostow, J. (1988). Automated Replay of Design Plans: Some Issues in Derivational Analogy. *Artificial Intelligence Journal*, to appear.

[28] Newell, A. (1980). Physical Symbol Systems. *Cognitive Science*, 4, 2, 135-184.

[29] Rosenbloom, P. S. and Newell, A. (1983). The Chunking of Goal Hierarchies: A Generalized Model of Practice. *Proceedings of the 1983 International Machine Learning Workshop*, 183-197, Urbana-Champaign, IL.

[30] Sacerdoti, E. D. (1977). *A Structure for Plans and Behavior*, North-Holland, Amsterdam.

[31] Schank, R. C. (1982). *Dynamic Memory*, Cambridge University Press.

[32] Schank, R. C. (1983). The Current State of AI: One Man's Opinion. *Artificial Intelligence Magazine*, IV, 1, 1-8.

[33] Shell, P. and Carbonell, J. G. (1988). Towards a General Framework for Composing Disjunctive and Iterative Macro-operators. *AAAI-88*, submitted.