Logic Programming as a Theoretical Tool in Educational Research

John E. McEnaney
Division of Education
Indiana University, South Bend, IN 46634

Abstract

This article makes the claim that much of educational research is predicated on a deductive model of inquiry. According to a deductive model, theory is evaluated by the extent to which explanations and predictions can be deductively derived from theoretical assumptions, known facts, and established principles. Although a wide variety of statistical methods have been developed to avoid sampling errors, the identification and prevention of logical error in theory development has attracted very little attention despite well developed logical methods that have been successfully applied in other fields.

The present article illustrates two applications of logic and logic programming in educational theory. One application is in the analysis and development of existing theory. A logic program is described that tests theoretical arguments for logical errors. A second application is in the development, revision, and testing of computer simulations developed in logic programming languages. A logic program is described that illustrates the power and convenience of Prolog, a logic programming language, for developing production system models of learning and cognition. The article concludes with a brief review of the features of logic programming that recommend it as a theoretical tool in educational research.

A great deal of educational research is predicated on a deductive model of inquiry (Hempel and Oppenheim, 1948; Hempel, 1966, pp. 49-54). According to a deductive model, the capacity of a theory to explain or predict phenomena is dependent on the deduction of conclusions (i.e., explanations or predictions) from the basic principles and facts assumed by the theory. A theory is said to explain phenomena when it leads to conclusions that have been observed. A theory is said to predict phenomena when conclusions suggest answers to as yet uninvestigated empirical questions. Successful theories are those that both explain and make predictions that are subsequently confirmed. This article proposes that formal logic and logic programming can provide educational researchers with a powerful new tools in the development, revision, and testing of theories that adhere to a deductive model of inquiry.
This article consists of three parts. Part 1 defines some basic concepts of logic and logic programming. Part 2 illustrates two applications of logic and logic programming in the evaluation and development of educational theory. Finally, Part 3 considers some of the unique advantages of evaluating and building theories with logic and logic programs.

What Is Logic Programming?

Nearly all logic programming languages are based on clausal form logic (Richards, 1989, p.v). One of the advantages of clausal form logic (CFL) is its simplicity. There are only two kinds of statements in CFL: facts and if-then relations. CFL does not require quantifiers (for any x, there is an x) or other connectives (and, or, if and only if). Despite its simplicity, however, CFL has the same logical power of more complex logics (e.g., predicate logic).

In addition to its simplicity, a second reason CFL has been almost universally adopted in computer-based reasoning systems is the development by Robinson (1965) of a method of inferencing called resolution that is especially well-suited to CFL. The combination of a straightforward logical form and a simple proof procedure that can be applied in a mechanical way has resulted in a powerful new approach to computing that deals with symbols rather than with numbers. This new approach to computing is called logic programming.

The programming language described in this article is Prolog (Coelho and Cotta, 1988; Sterling and Shapiro, 1986), the most widely known and used logic programming language. Like CFL, Prolog statements are of two kinds: facts and if-then relations. For example, the fact that Socrates is a man can be represented in Prolog by the Prolog fact “man(socrates).”, where “manness” is predicated of the object Socrates. The generalization that all men are mortal ((x)(man(x) --> mortal(x)) is represented by an if-then statement that says if something is a man, then it is mortal. In Prolog, an equivalent statement is written in slightly different notation with the symbol “:-” representing logical derivability. The Prolog equivalent of the statement that all men are mortal is “mortal(X) <- man(X).”, where X is a logical variable that provides for universal generalization. Translating the Prolog rule back into English results in “X is mortal if X is a man.”

The two Prolog statements above constitute a simple logic program that corresponds, as indicated below, to the premises of the traditional categorical syllogism.

<table>
<thead>
<tr>
<th>Prolog</th>
<th>English</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>man(socrates).</td>
<td>Socrates is a man.</td>
<td>(minor premise)</td>
</tr>
<tr>
<td>mortal(X) &lt;- man(X).</td>
<td>All men are mortal.</td>
<td>(major premise)</td>
</tr>
</tbody>
</table>

Prolog can do more than simply represent statements, however. Prolog can also derive conclusions based on the premises that constitute the logic program. For example, the two-statement program above can be asked whether the mortality of Socrates follows from the premises. This query takes the form “?- mortal(socrates).”. The system’s response to this query is “yes.”; according to the premises, Socrates is mortal. In more general terms, a logic program is a set of premises and a built-in proof procedure that allows a computer to determine whether other statements are derivable from those premises.
Applications of Logic Programs in Educational Research

Recall that the deductive model of inquiry treats theories as composed of two parts: 1) a set of basic facts and principles and 2) a deductive logic that allows explanations and predictions to be derived. The set of basic facts and principles of a theory correspond to the set of facts and rules assumed as premises by a Prolog program. The deductive logic used to generate explanations and predictions of a theory corresponds to the Prolog resolution proof procedure. According to the deductive model of inquiry, therefore, theories and logic programs share the same conceptual structure and, under appropriate circumstances, it is reasonable to think of logic programs as theories and, in turn, build theories as logic programs. In addition, the conceptual overlap of logic programs and theories suggests that logic programming may offer theoretical tools that will be useful in the analysis, testing and revision of theories.

As an illustration of the applications of logic programming in educational research, the remainder of this section will describe two Prolog programs. One of the two programs is actually a Prolog theorem prover for the propositional calculus but, when coupled with theory-specific input, this program can determine whether or not some outcome is predicted (or explained) by a theory. The review of the first program will focus on its application as a tool to analyze theoretical arguments that can be formulated in the propositional calculus.

The second of the two programs is a production system model, developed as a logic program, that examines the role of various strategies in simulating solutions to the Towers of Hanoi problem, a puzzle widely used in the exploration of human problem solving. The review of this model will focus on the characteristics of the logic programming environment that make it especially well suited to the rapid prototyping and development of production system models of learning and cognition.

Analyzing Theoretical Arguments with Logic Programs

Educational research often relies on complex arguments. Most research begins with certain assumptions and what are taken to be established facts and principles. These theoretical premises lead researchers to theoretical predictions that are tested by further empirical study. The philosophy of science that supports this approach is largely Popperian (1968); the goal of research is to falsify theoretical predictions. Errors attributable to statistical contingencies are usually accounted for in a systematic manner, and methods have been developed to avoid such errors (e.g., alpha levels and power analysis).

But another important source of error is usually not addressed in any systematic fashion. Despite the availability of well established logical methods, research in education rarely attempts to systematically identify or prevent logical errors that arise from the complex arguments employed in the development of theories and their predictions. One role of logical methods (including logic programming) is to provide educational researchers with a systematic method of identifying and avoiding logical errors.

Consider, for example, the following argument. Although this argument was specifically developed for pedagogical purposes and is somewhat compressed, I believe its complexity is no greater than arguments that are commonly employed in the literature of educational research. Read this argument and decide for yourself; is this a valid argument? Or is there a logical error that invalidates the conclusion drawn?
A disabled reader either suffers from inadequate phonic decoding ability, or from inadequate general language development. It has, however, been shown that if a disabled reader suffers from inadequate general language development, then that individual's WISC-R verbal score is significantly lower than the performance score. Since it has also been shown that the absence of adequate phonic decoding skills results in poor recall of orthographically regular nonwords, it follows that if a disabled reader does not demonstrate a significant difference between verbal and performance scores, then this disabled reader will demonstrate poor recall of orthographically regular nonwords.

The point I want to make with the argument above is that sometimes the validity of an argument is by no means self-evident, and under such circumstances the possibility of logical error poses a real danger to the validity of research findings. Fortunately, there are established logical methods that can be applied to identify and prevent such errors.

The first step in carrying out a logical analysis of the argument above is to translate it into a set of propositions that will be represented by symbols (uppercase letters). In essence, the argument consists of only four propositions which are related in certain specific ways using the logical connectives and (represented by the symbol "&"), or ("v"), not ("~"), and if...then or implies ("=").

The first premise of the argument takes the form \( P \lor L \) (\( P \lor L \)), where \( P \) represents the proposition "A disabled reader suffers from inadequate phonic decoding ability." and \( L \) represents the proposition "A disabled reader suffers from inadequate general language development." This disjunction of two possibilities indicates that at least one of these two statements must be the case (and possibly both).

The second premise of the argument takes the form \( \text{If } L, \text{ then } W, \text{ or } L \text{ implies } W \) (\( L \rightarrow W \)) where \( L \) is the same proposition as in the first premise and \( W \) represents the new proposition "A disabled reader's WISC-R verbal score is significantly lower than her performance score."

The third premise takes the form \( \text{If } P, \text{ then } R, \text{ or } P \text{ implies } R \) (\( P \rightarrow R \)), where \( P \) represents the proposition \( P \) in premise 1 and \( R \) represents the fourth and final proposition "The disabled reader has poor recall of orthographically regular nonwords."

The argument consists of these three premises and a conclusion that takes the form not \( W \) implies \( R \) (\( \neg W \rightarrow R \)) which translates back into English as "If the disabled reader's verbal score is not significantly lower than her performance score, then she will have poor recall of orthographically regular nonwords."

Summarizing the analysis to this point: the argument consists of four propositions that are related in specific ways. The first step in the analysis has been to translate those four propositions and the relations between those propositions into a symbolic shorthand. This translation is represented in Table 1 with the English original on the right and the propositional equivalents on the left.

With the argument represented in the symbolic manner in Table 1, it becomes possible to apply the formal methods of the propositional calculus to determine the validity of the argument. One way to demonstrate the validity of the argument is to generate a proof by hand. Lines 4 - 8 on the left is such a proof, with each line representing an intermediate step in the construction of the proof. Obviously, however, demonstrating the validity of an argument by generating a proof in propositional logic requires a knowledgeable theorem prover, and since formal logic is not usually a part of the training of educational researchers it might be objected that such a method is not generally useful.
Table 1.
Propositional analysis of the sample argument

<table>
<thead>
<tr>
<th>Logical Equivalents</th>
<th>English Original</th>
</tr>
</thead>
<tbody>
<tr>
<td>1) $P \lor L$</td>
<td>A disabled reader (either) suffers from inadequate phonic decoding ability ($P$), OR (a disabled reader suffers) from inadequate general language development ($L$).</td>
</tr>
<tr>
<td>2) $L \rightarrow W$</td>
<td>(It has, however, been shown that) IF a disabled reader suffers from inadequate general language development ($L$), THEN the disabled reader’s WISC-R verbal score is significantly lower than the performance score ($W$).</td>
</tr>
<tr>
<td>3) $P \rightarrow R$</td>
<td>(Since it has been shown that) IF adequate phonic decoding skills are absent ($P$), THEN poor recall of orthographically regular nonwords results ($R$).</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Conditional Proof</th>
</tr>
</thead>
<tbody>
<tr>
<td>4) $-W$</td>
</tr>
<tr>
<td>5) $-W \rightarrow -L$</td>
</tr>
<tr>
<td>6) $-L$</td>
</tr>
<tr>
<td>7) $P$</td>
</tr>
<tr>
<td>8) $R$</td>
</tr>
</tbody>
</table>

| (It follows that) IF the disabled reader does NOT demonstrate a significantly lower verbal score ($-W$), THEN the disabled reader will demonstrate poor recall of |

But there is another way to demonstrate the validity of the argument that does not require competence in formal logic. The alternative is to evaluate the questionable argument with an appropriate logic program. If the argument is valid, the argument as a whole will be a theorem of the propositional calculus. If the argument is not valid, it will not be a theorem. A logic program that can assess the theoremhood of formulas will, therefore, provide exactly what is needed. Moreover, as a logic programming language, Prolog is an ideal language for developing the theorem prover required, and just such a theorem prover is depicted in Table 2; a program which has been adapted from one developed by Pereira (1976). Sample input submitted to this program and the resulting output are depicted in Table 3 (the original argument discussed above is input 1.)
Table 2.
A Propositional Calculus Theorem Prover (Adapted from L. Pereira, 1976)

```
:- op(700,xfy,<=>)  /* defines the symbol for equivalence */
:- op(650,xfy,=>)  /* defines the symbol for implication */
:- op(600,xfy,v)  /* defines the symbol for disjunction */
:- op(550,xfy,\&)  /* defines the symbol for conjunction */
:- op(500,_,-)  /* defines the symbol for negation */

formulas :- repeat,
  write('Enter the formula followed by a period, then hit Return.'), nl,
  read(T), (T = = stop, write('END'), nl, nl; (test(T), fail)).

test(T) :- (false(T), write('Formula entered is not a valid theorem.'), !; 
  write('Formula entered is a valid theorem.'), nl, nl.

false('FALSE') :- !.
false('TRUE') :- !.

false(P <=> Q) :- false((P => Q) & (Q = > P)).
false(P => Q) :- false(- P v Q).
false(P v Q) :- false(P), false(Q).
false(P & Q) :- (false(P); false(Q)).
false(- P) :- false(P).
false(- (P <=> Q)) :- false(- (P = > Q) v - (Q = > P)).
false(- (P => Q)) :- false(- (P v Q)).
false(- (P v Q)) :- false(- P & - Q).
false(- (P & Q)) :- false(- P v - Q).
```

Table 3.
Sample Input and Output from the Theorem Prover Program Depicted in Table 2

<table>
<thead>
<tr>
<th>Premise(s)</th>
<th>Conclusion</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input 1:. ((P v L) &amp; (L = &gt; W) &amp; (P = &gt; R))</td>
<td>=&gt; (-W = &gt; R)</td>
</tr>
<tr>
<td>Output 1.: Formula entered is a valid theorem.</td>
<td></td>
</tr>
<tr>
<td>Input 2:. ((P v L) &amp; (L = &gt; W))</td>
<td>=&gt; (-W = &gt; P)</td>
</tr>
<tr>
<td>Output 2.: Formula entered is a valid theorem.</td>
<td></td>
</tr>
<tr>
<td>Input 2:. ((P v L) &amp; (L = &gt; W))</td>
<td>=&gt; (-W = &gt; R)</td>
</tr>
<tr>
<td>Output 2.: Formula entered is not a valid theorem.</td>
<td></td>
</tr>
</tbody>
</table>
The purpose of this example has been to demonstrate how logical analysis, either by hand or through the convenience of a logic program, can be employed to test the validity of complex theoretical arguments. Any theory that can be formulated as a set of propositions can be mechanically tested for logical errors by a computational theorem prover. Moreover, a computer-driven theorem prover is not limited to just testing the validity of theoretical arguments. Theorem provers have been developed (Lenat, 1982, 1983) that are capable of generating theorems on their own. In other words, the computer is capable of generating novel theoretical predictions from a set of premises; the program can actually be constructed to carry out new theoretical work.

Logic programs that prove theorems seem to offer at least two applications that should be of interest to educational researchers. One potential application of logic programs is analytic. An appropriately constructed logic program can test the validity of complex theoretical arguments. The second potential application of logic programs goes beyond mere analysis in the sense that the purpose of the program is to generate predictions. While it seems unlikely that mechanical theorem provers will demonstrate the kind of elegance and insight most favored by theoreticians, the benefit of such a system is that it makes theory accessible to anyone who can use the program. Even a theoretically naive researcher can test complex theoretical arguments or produce potentially interesting empirical predictions. Thus, while such programs will probably never replace the insightful theoretician, they do have the potential to make significant contributions to the progress of what Kuhn (1962) calls “normal” science, the plodding pedestrian task of elaborating, extending and testing established theory.

Logic Programs as Production Systems

The application of formal logic and logic programming described above begins with the assumption that the theory of interest already exists. The application that has been proposed is to translate existing theory into a logical form for the purpose of analysis or further development. The systematic, constructive character of formal logic suggests, however, that logic and logic programs may provide valuable tools for building theory as well as analyzing it.

One widely used approach to theory development in psychology and education is the production system model (Neches, Langley, and Klahr, 1987). A production system has two basic components: working memory and production memory. The production system as a whole models a problem solver or learner. Working memory in the production system represents declarative knowledge. The production memory represents procedural knowledge that allows the system to manipulate input or its own internal state (working or production memory). Production systems simulate problem solving by manipulating input that represents the problem to be solved. Production systems simulate learning by modifying their own working or production memories in response to input.

One reason to consider logic programming as a theoretical tool in educational research is that such an approach to computation provides an ideal environment for the development of production system models. The reason is that the declarative knowledge of working memory in a production system is functionally equivalent to a set of Prolog facts and a production is functionally equivalent to a Prolog rule. If the production conditions apply, then execute the production; if the antecedents of the Prolog rule are satisfiable, then the consequent follows. In fact, a set of productions can be viewed as a specialized “logic” where the execution of a production goal corresponds to the “derivation” of that goal from the
production conditions. The isomorphism of production systems and logic means that the tools of formal logic, including logic programming, can be applied in developing and analyzing production systems.

Moreover, the conceptual consistency of production systems and logic programs means that a logic programming environment avoids a central problem of the translation of theory into simulation. The problem is that programming languages rarely are based on the same conceptual structure as the original theory. The translation of theory into simulation under such circumstances results in an approximation to the original theory rather than an actual implementation. Developing production system models in a logic programming environment minimizes the gap between theory and simulation since both are based on the same conceptual foundations. The elimination of the gap between theory and simulation has the further benefit of allowing simulations to make substantive contributions to theory development and revision. The conceptual equivalence of theory and simulation means the internal operation of the simulation may be theoretically interpretable. In other words, the simulation can go beyond simply being a “black box” that provides output, given some input.

In the remainder of this section I intend to demonstrate how the Prolog programming language is especially well-suited to the development of production system models by describing a simple yet potentially useful model that simulates the solution of the Towers of Hanoi puzzle, a task often employed in investigations of problem solving.

The Towers of Hanoi puzzle (see Figure 1) consists of a board with three pegs and a set of different sized disks stacked pyramid-style on one peg. The goal of the puzzle is to move the stack of disks to another peg. What makes the puzzle difficult is that certain rules must be observed in moving the disks. One rule is that when a disk is removed from a peg it must be placed on another peg before a second disk can be moved. The second rule is that a disk can never be placed on top of a smaller disk.

![Figure 1](image.png)

Figure 1. The Tower of Hanoi puzzle depicted in initial state (Figure 1A) and goal state (Figure 1B) configurations.

The Towers of Hanoi problem is a convenient example because it provides a fairly constrained domain to explore problem solving. The initial state is a pyramidal stack of disks on one peg. The goal state is a similar stack on another peg. The state space of the Towers problem is represented graphically in Figure 2, where the initial state is any one of the three vertices of the triangular state space and the goal state is one of the two remaining vertices. As is apparent from Figure 2, there are 27 possible states in the system (for the
Figure 2. The problem state space of the 3-disk towers problem with states indicated by nodes and possible transitions indicated by connecting lines.

The three-disk problem and state transitions are significantly constrained by the specified rules (there are never more than 3 possible state transitions, represented by the lines connecting nodes, from any given state).

The first step in developing a production system model of the Towers problem has, in fact, already been taken with the specification of the problem space and the implicit identification of the primitive concepts employed, including pegs, disks, states, and state transitions. From one perspective these primitives are logical entities, and the goal of building the model is to specify the logic of the problem and its solution. From the production system perspective, state transitions are production system goals and states (composed of disks distributed across pegs) make up production conditions.

Knowledge about the problem state, operations that can be performed that lead to state transitions, and current system state indicators constitute the productions employed by the system. It is convenient to identify three kinds of productions: state productions, search productions, and move productions. State productions represent declarative knowledge in the system, including initial, goal, and current and past system states. State productions are represented using Prolog lists (indicated with square brackets), each of which represents a peg. For example, the state at the top of the triangle in Figure 1 (which happens to be the initial state - state zero) is represented by “state(0,[d1,d2,d3],[1,1,1]).”, where d1 is the smallest disk and d3 is the largest and left to right within the list represents top to bottom on the peg. This production represents declarative knowledge because it has no conditions. Since it has no conditions it always applies, so it represents an invariant state
Table 4.
Productions employed in the Towers of Hanoi problem solver

A. State and search productions

initial_state([d1,d2,d3],[d1,d2,d3],[d1,d2,d3]).  /* defines the initial state */
state([d1,d2,d3],[d1,d2,d3],[d1,d2,d3]).  /* first item in the state “trail” */
current_state(0).  /* identifies current state */
depth_of_search(2).  /* sets depth of backward search */
goal_state([d1,d2,d3],[d1,d2,d3],[d1,d2,d3]).  /* defines goal state */

p(1,[d1,d2,d3]).  /* These are redundant representations of the */
p(2,[]).  /* current state of the system that are employed */
p(3,[]).  /* as a matter of convenience */

B. Move productions

mdr :-
    moves(ML),  /* identifies possible moves */
pick_a_move2(ML,[A to B]),  /* tries to move disks to peg 3 */
check_move(A,B),  /* checks past moves */
move_disk(A,B),  /* moves disk & updates system */
!.

mdr :-
    moves(ML),  /* identifies possible moves */
pick_a_move3(ML,[A to B]),  /* tries to move disks to peg 2 */
check_move(A,B),  /* checks past moves */
move_disk(A,B),  /* moves disk & updates system */
!.

mdr :-
    moves(ML),  /* identifies possible moves */
pick_a_move4(ML,[A to B]),  /* tries to move disks to peg 1 */
check_move(A,B),  /* checks past moves */
move_disk(A,B),  /* moves disk & updates system */
!.

mdr :-
    moves(ML),  /* picks a move randomly */
pick_a_move(ML,[A to B]),  /* tries to move disks to peg 3 */
check_move(A,B),  /* checks past moves */
move_disk(A,B),  /* moves disk & updates system */
!.
rather than a "rule." For the purposes of the present example, the initial state is set to the state at the top of the triangle in Figure 1 ("initial_state([d1,d2,d3],[],[])") and the goal state is set to the lower right-hand vertex ("goal_state([],[],[d1,d2,d3])").

As suggested above, the initial state production (see Table 4A) specifies the distribution of disks at the start of the trial. In addition, the path of the system through the problem space is recorded with a "trail" of numbered states that depict the sequence of state transitions as the system tries to solve the problem. The search production specifies the depth of backward search employed by the system to prevent duplication of recent moves. In other words, the search production prevents the system from simply bouncing back and forth between nearby states or from circling around a constellation of points within limits specified by the depth of search parameter. Finally, move productions represent the rules and strategies that actually determine the movement of disks. State productions (working memory) are depicted in Table 4A. The four possible move productions are depicted in Table 4B.

The system begins from whatever state has been defined as the initial state. Each move production (mdr) begins with the moves sub-production that generates a list of possible moves. The mdr productions are selected according to the order in which they are listed in the program; in this system it is the order of the productions that provides the mechanism for resolving production conflicts. The pick_a_move sub-productions preferentially select certain moves so each of the four mdr productions has a "preference" for a certain kind of move. The check_move production carries out a backward search and discards the move selected if it violates the backward search criteria. When a move has been selected that satisfies the backward search criteria, the move_disk production moves the disk and updates the state of the system to reflect the move.

Moves require the most complex sets of conditions since they integrate all three kinds of productions. Following each move, the system compares the current state to the goal state. When the goal state is achieved the system halts. Output from the system is a description of the path from the initial to the goal state. As a convenience, an option is provided that allows any number of similar problem solving trials to be conducted. Each trial begins at the state specified by the initial state production and concludes when the system has achieved the goal state. Upon conclusion of a trial, the solution (the complete

![Figure 3. Single solution path based on a fixed strategy that attempts to move disks to peg 3, peg 2, and peg 1 in that order of priority.](https://example.com/figure3_image)
Sample output of Towers program

**Trial 1:**

depth_of_state_search(2).

state(0, [d1, d2, d3], [], []).
state(1, [d2, d3], [], [d1]).
state(2, [d2, d3], [d1], []).
state(3, [d3], [d1], [d2]).
state(4, [d3], [], [d1, d2]).
state(5, [d1, d3], [], [d2]).
state(6, [d1, d3], [d2], []).
state(7, [d3], [d2], [d1]).
state(8, [d3], [d1, d2], []).
state(9, [], [d1, d2], [d3]).
state(10, [], [d2], [d1, d3]).
state(11, [d1], [d2], [d3]).
state(12, [d1], [], [d2, d3]).
state(13, [], [], [d1, d2, d3]).

**Trial 2:**

depth_of_state_search(2).

state(0, [d1, d2, d3], [], []).
state(1, [d2, d3], [], [d1]).
state(2, [d3], [d2], [d1]).
state(3, [d3], [d1, d2], []).
state(4, [], [d1, d2], [d3]).
state(5, [], [d2], [d1, d3]).
state(6, [d1], [d2], [d3]).
state(7, [d1], [], [d2, d3]).
state(8, [], [], [d1, d2, d3]).

**Trial 3:**

depth_of_state_search(2).

state(0, [d1, d2, d3], [], []).
state(1, [d2, d3], [], [d1]).
state(2, [d2, d3], [d1], []).
state(3, [d3], [d1], [d2]).
state(4, [d3], [], [d1, d2]).
state(5, [d1, d3], [], [d2]).
state(6, [d1, d3], [d2], []).
state(7, [d3], [d2], [d1]).
state(8, [d3], [d1, d2], []).
state(9, [], [d1, d2], [d3]).
state(10, [], [d2], [d1, d3]).
state(11, [d2], [], [d1, d3]).
state(12, [d2], [d1], [d3]).
state(13, [d1], [d2], [d3]).
state(14, [], [], [d1, d2, d3]).

**Trial 4:**

depth_of_state_search(2).

state(0, [d1, d2, d3], [], []).
state(1, [d2, d3], [], [d1]).
state(2, [d3], [d2], [d1]).
state(3, [d3], [d1, d2], []).
state(4, [], [d1, d2], [d3]).
state(5, [], [d2], [d1, d3]).
state(6, [d2], [], [d1, d3]).
state(7, [d2], [d1], [d3]).
state(8, [d1], [d2], [d3]).
state(9, [], [], [d1, d2, d3]).

The path from initial_state to goal_state is written to a unique file for later examination.

Sample output from various trials that employ different search strategies are depicted in Figures 3, 4, and 5. The solution depicted in Figure 3 is based on the move hierarchy depicted in Table 4B. The first attempt is to move a disk to peg 3. If a move to peg 3 fails, the system tries a move to peg 2. If moves to pegs 2 and 3 both fail, the system moves a disk to peg 1. Since the strategy applied is a fixed one, the solution path generated is also fixed; every attempt to solve the problem using this strategy results in the same solution path. The solutions depicted in Figure 4 are based on a strategy that begins with an attempt to move a disk to peg 3, followed by a random move selection if a move to peg 3 is ruled out. Introduction of a random element in the system means multiple solution paths are possible.
Figure 4. Three solution paths based on a strategy that begins with an attempt to move a disk to peg 3, followed by a random move selection if a move to three is ruled out.

Figure 5. Three solution paths using the Figure 5 strategy but with a depth of search limited to examining only the last move rather than the last two moves as in Figures 4 and 5.
Finally, Figure 5 depicts three solution paths taken by the system using the Figure 4 strategy but with a depth of search limited to examining only the last move rather than the last two (depth of backward search is the last two moves in Figures 3 and 4).

The solution paths in Figures 3, 4, and 5 are graphic examples of the kind of influence search strategies and backward searching have on problem solving. The solution path in Figure 3 represents the one, and only, path that is possible under the fixed strategy adopted in that trial. The three solution paths depicted in Figure 4, on the other hand demonstrate that although random choices will usually result in less efficient problem solving, random choices may provide important opportunities to improve problem solving since, on occasion, a system that employs a random element will generate a better solution than a reliable but less than optimal fixed strategy. Finally, Figure 5 depicts three solution paths that demonstrate the critical role of backward search in enhancing the efficiency of the solution.

The model of problem solving presented above is a simple one. It has not been presented with the intention of contributing to either the theory of problem solving in general or the Towers of Hanoi problem in particular. Rather, the purpose of presenting this model has been to illustrate the kind of development environment logic programming provides for the rapid prototyping and modification of production system models.

The model described above was prototyped by a nonprofessional programmer in only a few hours. As is evident from the Prolog code in Table 4, Prolog is a very high-level programming language that is generally more readable than code in other programming languages. Another aspect of Prolog that makes it ideal for applications in educational research is that it is especially well suited to building models of learning. The power of Prolog in simulating learning is a consequence of the fact that Prolog makes no distinction between data and programs. Since data and programs are represented in the same way, a Prolog program can just as easily manipulate itself as it can data. This means that a Prolog program can not only add to its “static” declarative knowledge, it can also add to or otherwise modify its procedural knowledge. For example, after only minor modifications to the production system described above, a system was developed that could alter its solution strategy by reordering the mdr productions. Since the order of the mdr productions determines how production conflicts are resolved, it is apparent that such reordering can have an important influence on the performance of the system.

**Why Logic Programming?**

A reader familiar with computational models of cognition will find similarities between logic programs and other kinds of computational models. It seems reasonable to ask, therefore, why logic programs should be preferred to, for example, a production system developed in some other programming language (e.g., Karat, 1982), a PDP model (McClelland & Rumelhart, 1986) or some other kind of computer simulation. It is the purpose of this final section to present some features of logic programs that offer unique advantages in the development, testing, and revision of models of learning and cognition that are not available in other computational approaches.

Foremost among the unique features of logic programs as cognitive models is that they build on the well developed methods of formal logic. Since Prolog is a subset of the predicate calculus, Prolog programs can be analyzed with the exceptional rigor and power of predicate logic. A related advantage is that formal logic provides a basis for systematic meta-theoretical analysis. The same formal methods that are employed in the systematiza-
tion of theory can, in turn, be used to systematize a metatheoretical framework that will allow different theories to be analyzed and compared.

As indicated above, one advantage of developing production systems in a logic programming language (as opposed to some other language) is that the gap between theory and simulation is minimized. In addition, from a more practical perspective, logic programming provides a model of computation based on the same conceptual structures that underlie production systems. This correspondence between production system concepts and the programming environment promotes both faster learning of logic programming methods and more rapid prototyping of production system models.

An advantage logic programs have over PDP and other connectionist models is that a logic-based approach relies on symbol-based modeling of cognition. A central tenet of connectionism is to eliminate symbols in favor of non-symbolic connections. Although connectionists have successfully modeled a wide variety of "rote" tasks in learning and cognition, their capacity to model meaningful learning seems to be severely limited. Fodor and Pylyshyn (1988) have, for example, presented a convincing argument against the suitability of connectionist models in explaining linguistic competence. Moreover, the case against connectionism in educational research is even stronger since educational theory requires explanation at a symbolic psychological level if that explanation is to have any instructional relevance. Since logic programs provide a suitable explanatory framework, they have a distinct advantage over connectionist models from an educator's perspective.

Another feature recommending logic programs is that they integrate so well with the scientific framework typical of so much research in education: the deductive model of inquiry. Theories developed as logic programs can thereby provide for a clearer and more systematic philosophy of science for educational research.

A final feature recommending logic programs is that logic programming (and Prolog in particular) has recently become one of the most well developed and widely used tools in the field of artificial intelligence (AI). Research in AI has, moreover, increasingly tended to overlap with research in psychology and education. Logic programming thus offers educators, psychologists, and AI researchers a lingua franca that can serve to bring together common interests, problems, and solutions.

**Summary, Limitations and Conclusions**

This paper makes the claim that much of educational research is predicated on a deductive model of inquiry. According to such a model, theory is evaluated by the extent to which explanations and predictions can be deductively derived from theoretical assumptions, known facts, and established principles. Although a variety of powerful statistical methods have been developed to avoid sampling error, the identification and prevention of logical error in theory development has attracted very little attention despite well developed logical methods that have been successfully applied in other fields. The present article illustrates two applications of logic and logic programming in educational theory. One application is in the analysis and development of existing theory. The second application is in the development, revision, and testing of logic-based theory, including computer simulations developed in logic programming languages.

The attention focused in this paper on the deductive model of inquiry should not be construed as a claim that the deductive model is the only model of inquiry appropriate for educational research. The choice of a model of inquiry, naturally, depends on the question being addressed. The present research is limited in its application to research questions that
can be adequately addressed within a deductive framework. Given this limitation, it seems reasonable to conclude that, on the evidence of the examples above, educational researchers who employ a deductive approach to research would do well to consider adding formal logic and logic programming to their arsenal of theoretical tools.

References


