Applying Machine Learning Techniques to Rule Generation in Intelligent Tutoring Systems

Matthew P. Jarvis, Goss Nuzzo-Jones, Neil T. Heffernan
(mjarvis@wpi.edu, goss@wpi.edu, nth@wpi.edu)
Computer Science Department
Worcester Polytechnic Institute, Worcester, MA, USA

Abstract: The purpose of this research was to apply machine learning techniques to automate rule generation in the construction of Intelligent Tutoring Systems. By using a pair of somewhat intelligent iterative-deepening, depth-first searches, we were able to generate production rules from a set of marked examples and domain background knowledge. Such production rules required independent searches for both the "if" and "then" portion of the rule. This automated rule generation allows generalized rules with a small number of sub-operations to be generated in a reasonable amount of time, and provides non-programmer domain experts with a tool for developing Intelligent Tutoring Systems.

1 Introduction & Background

The purpose of this research was to develop tools that aid in the construction of Intelligent Tutoring Systems (ITS). Specifically, we sought to apply machine learning techniques to automate rule generation in the construction of ITS. These production rules define each problem in an ITS. Previously, authoring these rules was a time-consuming process, involving both domain knowledge of the tutoring subject and extensive programming knowledge. Model Tracing tutors [12] have been shown to be effective, but it has been estimated that it takes between 200 and 1000 hours of time to develop a single hour of content. As Murray, Blessing, & Ainsworth's [3] recent book has reviewed, there is great interest in figuring out how to make useful authoring tools. We believe that if Intelligent Tutoring Systems are going to reach their full potential, we must reduce the time it takes to program these systems. Ideally, we want to allow teachers to use a programming by demonstration system so that no traditional programming is required. This is a difficult problem. Stephen Blessing's Demonstr8 system [3]had a similar goal of inducing production rules. While Demonstr8 attempted to induce simple production rules from a single example by using the analogy mechanism in ACT-R, our goal was to use multiple examples, rather than just a single example.

We sought to embed our rule authoring system within the Cognitive Tutor Authoring Tools [6] (CTAT, funded by the Office of Naval Research), generating JESS (an expert system language based on CLIPS) rules. Our goal was to automatically generate generalized JESS (Java Expert System Shell) rules for a problem, given background knowledge in the domain, and examples of the steps needed to complete the procedure. This example-based learning is a type of Programming by Demonstration [5] [8]. Through this automated method, domain experts would be able to create ITS without programming knowledge. When compared to tutor development at present, this could provide an enormous benefit, as writing the rules for a single problem can take a prohibitive amount of time.

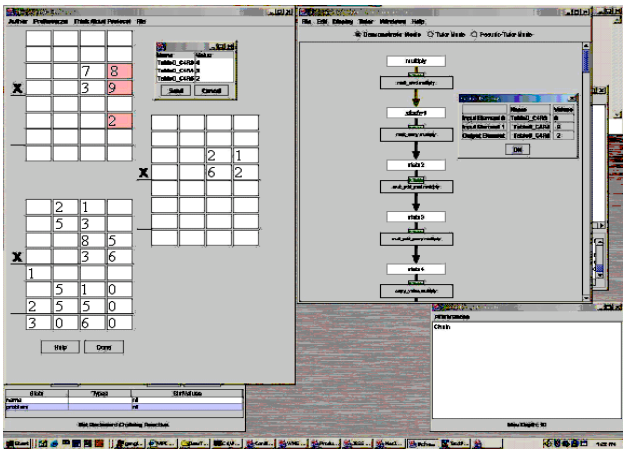
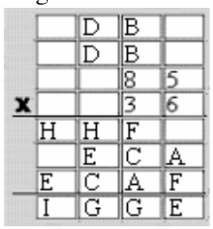


Fig. 1. Example Markup and Behavior Recorder

The CTAT provide an extensive framework for developing intelligent tutors. The tools provide an intelligent GUI builder, a Behavior Recorder for recording solution paths, and a system for production rule programming. The process starts with a developer designing an interface in which a subject matter expert can demonstrate how to solve the problem. CTAT comes with a set of recordable and scriptable widgets (buttons, menus, text-input fields, as well as some more complicated widgets such as tables) (shown in Figure 1) as we will see momentarily. The GUI shown in Figure 1 shows three multiplication problems on one GUI, which we do just to show that this system can generalize across problems; we would not plan to show students three different multiplication problems at the same time.

Creating the interface shown in Figure 1 involved dragging and dropping three tables into a panel, setting the size for the tables, adding the help and "done" buttons, and adding the purely decorative elements such as the "X" and the bold lines under the fourth and seventh rows. Once the interface is built, the developer runs it, sets the initial state by typing in the initial numbers, and clicks "create start state". While in "demonstrate mode", the developer demonstrates possibly multiple sets of correct actions needed to solve the problems. The Behavior Recorder records each action with an arc in the behavior recorder window. Each white box indicates a state of the interface. The developer can click on a state to put the interface into that state. After demonstrating correct actions, the developer demonstrates common errors, and can write "bug" messages to be displayed to the student, should they take that step. The developer can also add a hint message to each arc, which, should the student click on the hint button, the hint sequence would be presented to the student, one by one, until the student solved the problem. A hint sequence will be shown later in Figure 4. At this point, the developer takes the three problems into the field for students to use. The purpose of this is to ensure that the design seems reasonable. His software will work only for these three problems and has no ability to generalize to another multiplication problem. Once the developer wants to make this system work for any multiplication problem instead of just the three he has demonstrated, he will need to write a set of production rules that are able to complete the task. At this point, programming by demonstration starts to come into play. Since the developer already wanted to demonstrate several steps, the machine learning system can use those demonstrations as positive examples (for correct student actions) or negative examples (for expected student errors) to try to induce a general rule.

In general, the developer will want to induce a set of rules, as there will be different rules representing different conceptual steps. Figure 2 shows how the developer could break down a multiplication problem into a set of nine rules. The developer must then mark which actions correspond to which rules. This process should be relatively easy for a teacher. The second key way we make the task feasible is by having the developer tell us a set of inputs for each rule instance. Figure 1 shows the developer click in the interface to indicate to the system that the greyed cells containing the 8 and 9 are inputs to the rule (that the developer named "mult_mod") that should be able to generate the 2 in the A position (as shown in Figure 2). The right hand side of Figure 2 shows the six examples of the "mult_mod" rule with the two inputs being listed first and the output listed last. These six examples correspond to the six locations in Figure 1 where an "A" is in one of the tables.



Rule Label	Rule Action
A	Multiply, Mod 10
B	Multiply, Div 10
C	Multiply, Add Carry, Mod 10
D	Multiply, Add Carry, Div 10
E	Copy Value
F	Mark Zero
G	Add, Add Carry, Mod 10
H	Add, Add Carry, Div 10
I	Add

Rule A Examples	
Inputs	Outputs
8, 9	2
8, 3	4
1, 2	2
1, 6	6
5, 6	0
5, 3	5

Fig. 2. Multiplication Rules

These two hints (labeling rules and indicating the location of input values) that the developer provides for us help reduce the complexity of the search enough to make some searches computationally feasible (inside a minute). The inputs serve as "islands" in the search space that will allow us to separate the right hand side and the left hand side searches into two separate steps. Labeling the inputs is something that the CTAT did not provide, but without which we do not think we could have succeed at all.

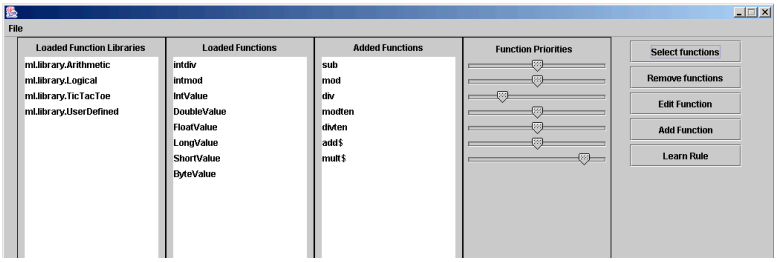


Fig. 3. Function Selection Dialog

The tutoring systems capable of being developed by the CTAT are composed of an interface displaying each problem, the rules defining the problem, and the working memory of the tutor. Most every GUI element (text field, button, and even some entities like columns) have a representation in working memory. Basically, everything that is in the interface is known in working memory. The working memory of the tutor stores the state of each problem, as well as intermediate variables and structures associated with any given problem. Working memory elements (JESS facts) are operated upon by the JESS rules defining each problem. Each tutor is likely to have its own unique working memory structure, usually a hierarchy relating to the interface elements. The CTAT provide access and control to the working memory of a tutor during construction, as well as possible intermediate working memory states. This allows a developer to

debug possible JESS rules, as well as for the model-tracing algorithm [4] [1] of the Authoring Tools to validate such rules.

2 Right-Hand Side Search Algorithm

We first investigated the field of Inductive Logic Programming (ILP) because of its similar problem setup. ILP algorithms such as FOIL [11], FFOIL [10], and PROGOL [9] were given examples of each problem, and a library of possible logical relations that served as background knowledge. The algorithms were then able to induce rules to cover the given examples using the background knowledge. However, these algorithms all use information gain heuristics, and develop a set of rules to cover all available positive examples. Our problem requires a single rule to cover all the examples, and partial rules are unlikely to cover any examples, making information gain metrics ineffective. ILP also seems to be geared towards the problems associated with learning the left-hand side of the rule.

With the unsuitability of ILP algorithms, we then began to pursue our own rule-generation algorithm. Instead of background knowledge as a set of logical relations, we give the system a set of functions (i.e., math and logical operators). We began by implementing a basic iterative-deepening, depth-first search through all possible function combinations and variable permutations. The search iterates using a declining probability window. Each function in the background knowledge is assigned a probability of occurrence, based on a default value, user preference, and historical usage. The search selects the function with the highest probability value from the background knowledge library. It then constructs a variable binding with the inputs of a given example problem, and possibly the outputs from previous function/variable bindings. The search then repeats until the probability window (depth limit) is reached. Once this occurs, the saved ordering of functions and variable bindings is a rule with a number of sub-operations equal to the number of functions explored. We define the length of the rule as this number of sub-operations. Each sub-operation is a function chosen from the function library (see Figure 3), where individual function probabilities can be initially set (the figure shows that the developer has indicated that he thinks that multiplication is very likely compared to the other functions). The newly generated rule is then tested against the example it was developed from, all other positive examples, and negative examples if available. Should the rule not describe all of the positive examples, or incorrectly predict any negative examples, the last function/variable binding is removed, the probability window is decreased, and the search continues until a function/variable binding permutation meets with success or the search is cancelled.

This search, while basic in design, has proven to be useful. In contrast to the ILP methods described earlier, this search will specifically develop a single rule that covers all examples. It will only consider possible rules and test them against examples once the rule is "complete," or the rule length is the maximum depth of the search. However, as one would expect, the search is computationally prohibitive in all but the simple cases, as run time is exponential in the number of functions as well as the depth of the rule. This combinatorial explosion generally limits the useful depth of our search to about depth five, but for learning ITS rules, this rule length is acceptable since one of the points of intelligent tutoring systems is to create very finely grained rules. The search can usually find simple rules of depth one to three in less than thirty seconds, making it possible that as the developer is demonstrating examples, the system is using background processing time to try to induce the correct rules. Depth four rules can generally be achieved in less than three minutes. Another limitation of the search is that it assumes entirely accurate examples. Any noise in the examples or background knowledge will result in an incorrect rule, but this is acceptable as we can rely on the developer to accurately create examples.

While we have not altered the search in any way so as to affect the asymptotic efficiency, we have made some small improvements that increase the speed of learning the short rules that we desire. The first was to take advantage of the possible commutative properties of some background knowledge functions. We allow each function to be marked as commutative, and if it is, we are able to reduce the variable binding branching factor by ignoring variable ordering in the permutation.

We noted that in ITSs, because of their educational nature, problems tend to increase in complexity inside a curriculum, building upon themselves and other simpler problems. We sought to take advantage of this by creating support for "macro-operators," or composite rules. These composite rules are similar to the macro-operators used to complete sub-goals in Korf's work with state space searches [7]. Once a rule has been learned from the background knowledge functions, the user can choose to add that new rule to the background knowledge. The new rule, or even just pieces of it, can then be used to try to speed up future searches.

3 Left-Hand Side Search Algorithm

The algorithm described above generates what are considered the right-hand side of JESS production rules. JESS is a forward-chaining production system, where rules resemble first-order logical rules (given that there are variables), with a left and right hand side. The left-hand side is a hierarchy of conditionals which must be satisfied for the right-hand side to execute (or "fire" in production system parlance) [4]. As previously mentioned, the tutoring systems being constructed retain a working memory, defining the variables and structures for each problem in the tutor being authored. The left-hand side of a production rule being activated in the tutoring system checks its conditionals against working memory elements. Each conditional in the hierarchy checks against one or more elements of working memory; each element is known as a fact in working memory. Within each conditional is pattern-matching syntax, which defines the generality of the conditional. As we mentioned above, working memory elements, or facts, often have a one-to-one correspondence with elements in the interface. For instance, a text field displayed on the interface will have a corresponding working memory element with its value and properties. More complex interface elements, such as tables, have associated working memory structures, such as columns and rows. A developer may also define abstract working memory structures, relating interface elements to each other in ways not explicitly shown in the interface.

To generate the left-hand side in a similarly automated manner as the right-hand side, we must make create a hierarchy of conditionals that generalizes the given examples, but does not "fire" the right-hand side inappropriately. Only examples listed as positive examples can be used for the left-hand side search, as examples denoted as negative are incorrect in regard to the right-hand side only. For our left-hand side generation, we make the assumption that the facts in working memory are connected somehow, and do not loop. They are connected to form "paths" (as can be seen in the Figure 4) where tables point to lists of columns which in turn point to lists of cells which point to given cell which has a value.

To demonstrate how we automatically generate the left-hand side, we will step through an example JESS rule, given in Figure 4. This "Multiply, Mod 10" rule occurs in the multi-column multiplication problem described below. Left-hand side generation is conducted by first finding all paths searching from the "top" of working memory (the "?factMAIN_problem1" fact in the example) to the "inputs" (that the developer has labeled in the procedure shown in Figure 1) that feed into the right-hand side search (in this case, the cells containing the values being operated on by the right-hand side operators.) This search yields a set of paths from the "top" to the values

themselves. In this multiplication example, there is only one such path, but in Experiment #3 we had multiple different paths from the "top" to the examples. Even with the absence of multiple ways to get from "top" to an input, we still had a difficult problem.

Once we combine the individual paths, and there are no loops, the structure can be best represented as a tree rooted at "top" with the inputs and the single output as leaves in the tree. This search can be conducted on a *single* example of working memory, but will generate rules that have very specific left-hand sides which assume the inputs and output locations will always remain fixed on the interface. This assumption of fixed locations is violated somewhat in this example (the output for A moves and so does the second input location) and massively violated in tic-tac-toe. Given that we want parsimonious rules, we bias ourselves towards short rules but risk learning a rule that is too specific unless we collect multiple examples.

```
;; define a rule called mult_mod (see Multi-Column Multiplication problem for details)

(defrule mult_mod
        ;; begin left-hand side ("if")
        (addition (problem ?factMAIN__problem1 ))
        ;; select any table from working memory
        ?factMAIN__problem1 <- (problem
                (interface-elements $? ?factMAIN__tableAny1 $? ))
                ; ;bind an interface element ?factMAIN__tableAny1

                ;; and if that element is a table then…
                ;; [going to first get the top input that is always on the far right]
                ?factMAIN__tableAny1 <- (table
                        (columns ? ? ? ?factMAIN__column4  $? ))
                         ;;then select the 4th  (right most) column from that table
                        ?factMAIN__column4 <- (column
                                (cells ? ? ?factMAIN__cell3  $? ))
                                ;; select the cell in the 3rd row of the fourth column
                                ;;and bind the value in that cell to ?v#0#1, while also ;;checking that its not
                                nil
                                ?factMAIN__cell3 <- (cell
                                        (value ?v#0#1   &   ~nil))
                                        ;bind and check not nil
                                        ;; from that same table, select any column
                ?factMAIN__tableAny1 <- (table
                        (columns $? ?factMAIN__columnAny1 $? ))
                        ;; select any cell from that  column
                        ?factMAIN__columnAny1 <- (column
                                (cells $? ?factMAIN__cellAny1 $? ))

                                ;; select the name of any cell in any column of the
                                ;; selected table. Check to make sure this cell is empty,
                                ;; as it is the output cell
                                ?factMAIN__cellAny1 <- (cell
                                        (name ?sai_name) (value nil))

                        ;; select the 4th cell from that column
                        ?factMAIN__columnAny1 <- (column
                                (cells ? ? ? ?factMAIN__cell4  $? ))

                                ;; select the value of the 4th  cell of the any column of
                                ;; the selected table. This cell is required to be in the
                                ;; same column as the above cell (cell1).  Check to make
                                ;; sure that cell is not empty
                                ?factMAIN__cell4 <- (cell
                                        (value ?v#0#0&~nil))

        ?selection-action-input <- (selection-action-input)
        ;this line indicates it's a correct rather than buggy rule
        ;; end left-hand side
=>
        ;; begin right-hand side ("then")
        ;; Execute right-hand side operations on selected arguments
        (bind ?v#1#0 ((new ml.library.Arithmetic) mult$
                (new java.lang.Integer (integer ?v#0#0))
                (new java.lang.Integer (integer ?v#0#1))))

        (bind ?v#1#1 ((new ml.library.Arithmetic) modten
                 (new java.lang.Integer (integer ?v#1#0))))
```

```
;; output computed value to appropriate cell and update working memory
(modify ?factMAIN__cellAny1 (value ?v#1#1))
(modify ?selection-action-input
;these lines related to model tracing and are not discussed herein
            (selection ?sai_name)
            (action "UpdateTable")
            (input ?v#1#1)))
```

Fig. 4. An actual JESS rule that we learned. The order of the conditionals on the left hand side has been changed, and indentation added, to make the rule easier to understand

One of these trees would be what would come about if we were only looking at the first instance of rule A, as shown in Figure 2, where you would tend to assume that the two inputs and the output will always be in the same last column as shown graphically in Figure 5.

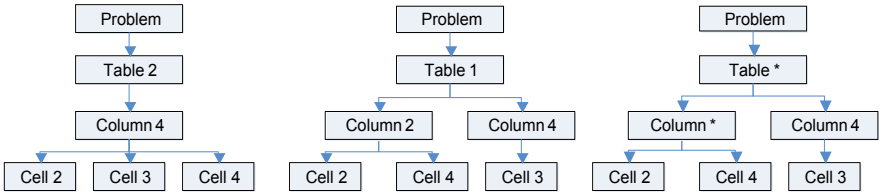


Fig. 5. Left-hand side trees

A different set of paths from top to the inputs occurs in the second instance of rule A that occurs in the 2nd column, 7th row. In this example we see that the first input and second input are not always in the same column, but the 2nd input and the output are in the same column as shown in Figure 5.

One such path is the series of facts given in the example rule, from problem to table, to two possible columns, to three cells within those columns. Since this path branches and contains no loops, it can best be represented as a tree. This search can be conducted on a single example of working memory, but will generate a very specific left-hand side. To create a generalized left-hand side, we need to conduct this path search over multiple examples.

Despite the obvious differences in the two trees shown above, they represent the left-hand side of the same rule, as the same operations are being performed on the cells once they are reached. Thus, we must create a general rule that applies in both cases. To do this, we merge the above trees to create a more general tree. This merge operation marks where facts are the same in each tree, and uses wildcards (*) to designate where a fact may apply in more than one location. If a fact cannot be merged, the tree will then split. A merged example of the two above trees is show in Figure 5.

In this merged tree (there are many possible trees), the "Table 1" and "Table 2" references have been converted to a wildcard "Table *." This generalizes the tree so that the wildcard reference can apply to any table, not a single definite one. Also the "Column 2" reference in the first tree has been converted to a wildcard. This indicates that that column could be any column, not just "Column 2". This allows this merged tree to generalize the second tree as well, for the wildcard could be "Column 4." This is one possible merged tree resulting from the merge operation, and is likely to be generalized further by additional examples. However, it mirrors the rule given in Figure 4, with the exception that "Cell 2" is a wildcard in the rule.

We can see the wildcards in the rule by examining the pattern matching operators. For instance, we select any table by using:

```
?factMAIN__problem1 <- (problem
        (interface-elements $? ?factMAIN__tableAny1 $? ))
```

The "$?" operators indicate that there may be any number of interface elements before or after the "?factMAIN_tableAny1" that we select. To select a fact in a definite position, we use the "?" operator, as in this example:

```
?factMAIN__tableAny1 <- (table
        (columns ? ? ? ?factMAIN__column4  $? ))
```

This selects the 4th column by indicating that there a three preceding facts (three "?'s") and any number of facts following the 4th ("$?").

We convert the trees generated by our search and merge algorithm to JESS rules by applying these pattern matching operations. The search and merge operations often generate more than one tree, as there can be multiple paths to reach the inputs, and to maintain generality, many different methods of merging the trees are used. This often leads to more than one correct JESS rule being provided.

We have implemented this algorithm and the various enhancements noted in Java within the CTAT. This implementation was used in the trials reported below, but remains a work in progress. Following correct generation of the desired rule, the algorithm outputs a number of JESS production rules. These rules are verified for consistency with the examples immediately after generation, but can be further tested using the model trace algorithm of the authoring tools [4].

4 Methods/Experiments

4.1 Experiment #1: Multi-Column Multiplication

The goal of our first experiment was to try to learn all of the rules required for a typical tutoring problem, in this case, Multi-Column Multiplication. In order to extract the information that our system requires, the tutor must demonstrate each action required to solve the problem. This includes labeling each action with a rule name, as well as specifying the inputs that were used to obtain the output for each action. While this can be somewhat time-consuming, it eliminates the need for the developer to create and debug his or her own production rules.

For this experiment, we demonstrated two multiplication problems, and identified nine separate skills, each representing a rule that the system was asked to learn (see Figure 2). After learning these nine rules, the system could automatically complete a multiplication problem. These nine rules are shown in Figure 2.

The right-hand sides of each of these rules were learned using a library of Arithmetic methods, including basic operations such as add, multiply, modulus ten, among others. Only positive examples were used in this experiment, as it is not necessary (merely helpful) to define negative examples for each rule. The left-hand side search was given the same positive examples, as well as the working memory state for each example.

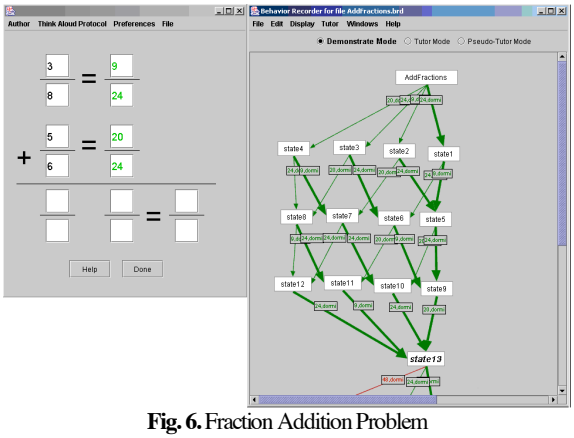
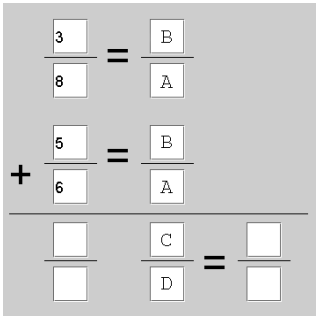


Fig. 6. Fraction Addition Problem

4.2 Experiment #2: Fraction Addition

Our second experiment was to learn the rules for a solving a fraction addition problem. These rules were similar to the multiplication rules in the last experiment, but had a slightly different complexity. In general, the left-hand side of the rules was simpler, as the interface had fewer elements and they were organized in a more definite way. The right-hand-side of the rules were of similar complexity to many of the rules in multiplication.

We demonstrated a single fraction addition problem using the Behavior Recorder and identified the rules shown in Figure 7. The multiple solution paths that are displayed in the Behavior Recorder allow the student to enter the values in any order they wish.



Rule B Examples	
Inputs	**Outputs**
8,6,3	9
6,8,5	20

Rule Label	Rule Action
A	Least Common Multiple
B	Least Common Multiple, Divide, Multiply
C	Add Numerators
D	Copy Value

Fig. 7. Fraction Addition Rules

4.3 Experiment #3: Tic-Tac-Toe

In this experiment, we attempted to learn the rules for playing an optimal game of Tic-Tac-Toe (see Figure 8). The rules for Tic-Tac-Toe differ significantly from the rules of the previous problem. In particular, the right-hand side of the rule is always a single operation, simply a mark "X" or a mark "O." The left-hand side is then essentially the entire rule for any Tic-Tac-Toe rule, and the left-hand sides are more complex than either of the past two experiments. In order to correctly learn these rules, it was necessary to augment working memory with information particular to a Tic-Tac-Toe game. Specifically, there are eight ways to win a Tic-Tac-Toe game: one of the three rows, one of the three columns, or one of the two diagonals. Rather than simply grouping

cells into columns as they were for multiplication, the cells are grouped into these winning combinations (or "triples"). The following rules to play Tic-Tac-Toe were learned using nine examples of each:

- Rule #1: Win (win the game with one move)
- Rule #2: Play Center (optimal opening move)
- Rule #3: Fork (force a win on the next move)
- Rule #4: Block (prevent an opponent from winning)

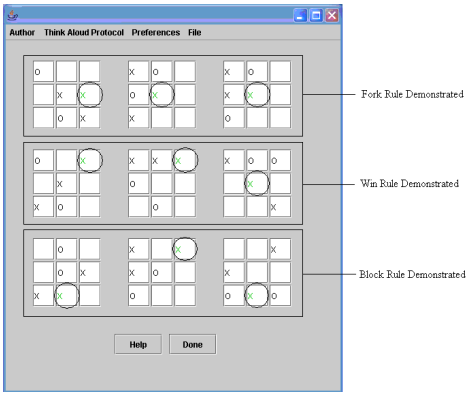


Fig. 8. Tic-Tac-Toe Problem

5 Results

These experiments were performed on a Pentium IV, 1.9 GHz with 256 MB RAM running Windows 2000 and Java Runtime Environment 1.4.2. We report the time it takes to learn each rule, including both the left-hand-side search and the right-hand-side search.

5.1 Experiment #1: Multi-Column Multiplication

Rule Label	Rule Learned	Time to Learn (seconds)	Number of Steps
A	Multiply, Mod 10	0.631	2
B	Multiply, Div 10	0.271	2
C	Multiply, Add Carry, Mod 10	20.249	3
D	Multiply, Add Carry, Div 10	18.686	3
E	Copy Value	0.190	1
F	Mark Zero	0.080	1
G	Add, Add Carry, Mod 10	16.354	3
H	Add, Add Carry, Div 10	0.892	3
I	Add	0.160	1
	Total Time:	57.513	

5.2 Experiment #2: Fraction Addition

Rule Label	Rule Learned	Time to Learn (seconds)	Number of Steps
A	LCM	0.391	1
B	LCM, Divide, Multiply	21.461	3
C	Add	2.053	1
D	Copy Value	0.060	1
	Total Time:	**23.965**	

5.3 Experiment #3: Tic-Tac-Toe

Rule Learned	Time to Learn (seconds)	Number of Steps
Win	1.132	1
Play Center	1.081	1
Fork	1.452	1
Block	1.102	1
Total Time:	**4.767**	

6 Discussion

6.1 Experiment #1: Multi-Column Multiplication

The results from Experiment #1 show that all of the rules required to build a Multi-Column Multiplication tutor can be learned in a reasonable amount of time. Even some longer rules that require three mathematical operations can be learned quickly using only a few positive examples. The rules learned by our algorithm will correctly fire and model-trace within the CTAT. However, these rules often have over general left-hand sides. For instance, the first rule learned, "Rule A", (also shown in Figure 4), may select arguments from several locations. The variance of these locations within the example set leads the search to generalize the left-hand side to select multiple arguments, some of which may not be used by the rule. During design of the left-hand side search, we intentionally biased the search towards more general rules. Despite these over-generalities, this experiment presents encouraging evidence that our system is able to learn rules that are required to develop a typical tutoring system.

6.2 Experiment #2: Fraction Addition

The rules for the Fraction Addition problem had, in general, less complexity than the Multi-Column Multiplication problem. The right hand sides were essentially much simpler, and the total number of rules employed much lower. The left-hand sides did not suffer from the over-generality experience in Multi-Column Multiplication, as the number of possible arguments to the rules was much fewer. This experiment provides a fair confirmation of the capabilities of both the left and right hand side searches with regard to the Multi-Column Multiplication problem.

6.3 Experiment #3: Tic-Tac-Toe

To learn appropriate rules for Tic-Tac-Toe, we employed abstract working memory structures, relating the interface elements together. Specifically, we created working memory elements relating each "triple" or set of three consecutive cells together. These triples are extremely important when creating a "Win" or "Block" rule. With these additions to working memory, our left-hand side search was able to create acceptably general rules for all four skills listed. However, as in the case of Multi-Column Multiplication, some rules were over-general, specifically the "Fork" rule, with which our search is unable to recognize that the output cell is always the intersection of two triples. This observation leads to the most optimal rule; our search generates working but over-general pattern matching. Nonetheless, this experiment demonstrates an encouraging success in regard to generating complex left-hand sides of JESS rules.

7 Conclusions

Intelligent tutoring systems provide an extremely useful educational tool in many areas. However, due to their complexity, they will be unable to achieve wide usage without a much simpler development process. The CTAT [6] provide a step in the right direction, but to allow most educators to create their own tutoring systems, support for non-programmers is crucial. The rule learning algorithm presented here provides a small advancement toward this goal of allowing people with little or no programming knowledge to create intelligent tutoring systems in a realistic amount of time. While the algorithm presented here has distinct limitations, it provides a significant stepping-stone towards automated rule creation in intelligent tutoring systems.

7.1 Future Work

Given that we are trying to learn rules with a brute force approach, our search is limited to short rules. We have experimented with allowing the developer to control some of the steps in our algorithm while allowing the system to still do some search. The idea is that developers can do some of the hard steps (like the RHS search), while the system can be left to handle some of the details (like the LHS search). In order to use machine learning effectively, we must get the human computer interaction "correct" so that the machine learning system can be easily controlled by developers. We believe that this research is a small step toward accomplishing this larger goal.

7.2 Acknowledgements

This research was partially funded by the Office of Naval Research (ONR) and the US Department of Education. The opinions expressed in this paper are solely those of the authors and do not represent the opinions of ONR or the US Dept. of Education.

References

1. Anderson, J. R. and Pellitier, R. (1991) A developmental system for model-tracing tutors. In Lawrence Birnbaum (Eds.) *The International Conference on the Learning Sciences*. Association for the Advancement of Computing in Education. Charlottesville, Virginia (pp. 1-8).
2. Blessing, S.B. (2003) A Programming by Demonstration Authoring Tool for Model-Tracing Tutors. In Murray, T., Blessing, S.B., & Ainsworth, S. (Ed.), Authoring Tools for Advanced Technology Learning Environments: Toward Cost-Effective Adaptive, Interactive and Intelligent Educational Software. (pp. 93-119). Boston, MA: Kluwer Academic Publishers
3. Choksey, S. and Heffernan, N. (2003) An Evaluation of the Run-Time Performance of the Model-Tracing Algorithm of Two Different Production Systems: JESS and TDK. Technical Report WPI-CS-TR-03-31. Worcester, MA: Worcester Polytechnic Institute
4. Cypher, A., and Halbert, D.C. Editors. (1993) Watch what I do : Programming by Demonstration. Cambridge, MA: The MIT Press.
5. Koedinger, K. R., Aleven, V., & Heffernan, N. T. (2003) Toward a rapid development environment for cognitive tutors. 12th Annual Conference on Behavior Representation in Modeling and Simulation. Simulation Interoperability Standards Organization.
6. Korf, R. (1985) Macro-operators: A weak method for learning. Artificial Intelligence, Vol. 26, No. 1.
7. Lieberman, H. Editor. (2001) Your Wish is My Command: Programming by Example. Morgan Kaufmann, San Francisco
8. Muggleton, S. (1995) Inverse Entailment and Progol. New Generation Computing, Special issue on Inductive Logic Programming, 13.
9. Quinlan, J.R. (1996). Learning first-order definitions of functions. Journal of Artificial Intelligence Research. 5. (pp 139-161)
10. Quinlan, J.R., and R.M. Cameron-Jones. (1993) FOIL: A Midterm Report. Sydney: University of Sydney.
11. VanLehn, K., Freedman, R., Jordan, P., Murray, C., Rosé, C. P., Schulze, K., Shelby, R., Treacy, D., Weinstein, A. & Wintersgill, M. (2000). Fading and deepening: The next steps for Andes and other model-tracing tutors. *Intelligent Tutoring Systems: 5th International Conference*, Montreal, Canada. Gauthier, Frasson, VanLehn (eds), Springer (Lecture Notes in Computer Science, Vol. 1839), pp. 474-483.