APPLYING MACHINE LEARNING TECHNIQUES TO RULE GENERATION IN INTELLIGENT TUTORING SYSTEMS

by

Matthew Paul Jarvis

A Thesis

Submitted to the Faculty

of the

WORCESTER POLYTECHNIC INSTITUTE

in partial fulfillment of the requirements for the

Degree of Master of Science

in

Computer Science

May 2004

Research Partner: Goss Nuzzo-Jones

APPROVED:

____
Professor Neil T. Heffernan, Thesis Advisor

____
Professor Stanley M. Selkow, Thesis Reader

____
Professor Michael A. Gennert, Head of Department

Keywords: Intelligent Tutoring Systems, Model Tracing, Machine Learning, Artificial Intelligence, Programming by Demonstration.

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.

Acknowledgements

I would like to thank Prof. Neil Heffernan for the substantial amount of time he invested in advising me throughout the development of this Thesis. I would also like to thank my research partner, Goss Nuzzo-Jones, for his significant contribution to the research and development of this project. Also, thanks to Sanket Choksey for his help in integrating this work into the Cognitive Tutor Authoring Tools. Finally, thanks to my family and loved ones for their support during my time at WPI.

Contents

1 Introduction

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 (VanLehn, et. al., 2000) 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 (2004) 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 (Blessing, 2003) 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 (Koedinger, Aleven & Heffernan, 2003) (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 (Cypher, Halbert, 1993) (Lieberman, 2001). 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.

2 Background

The Cognitive Tutor Authoring Tools 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.

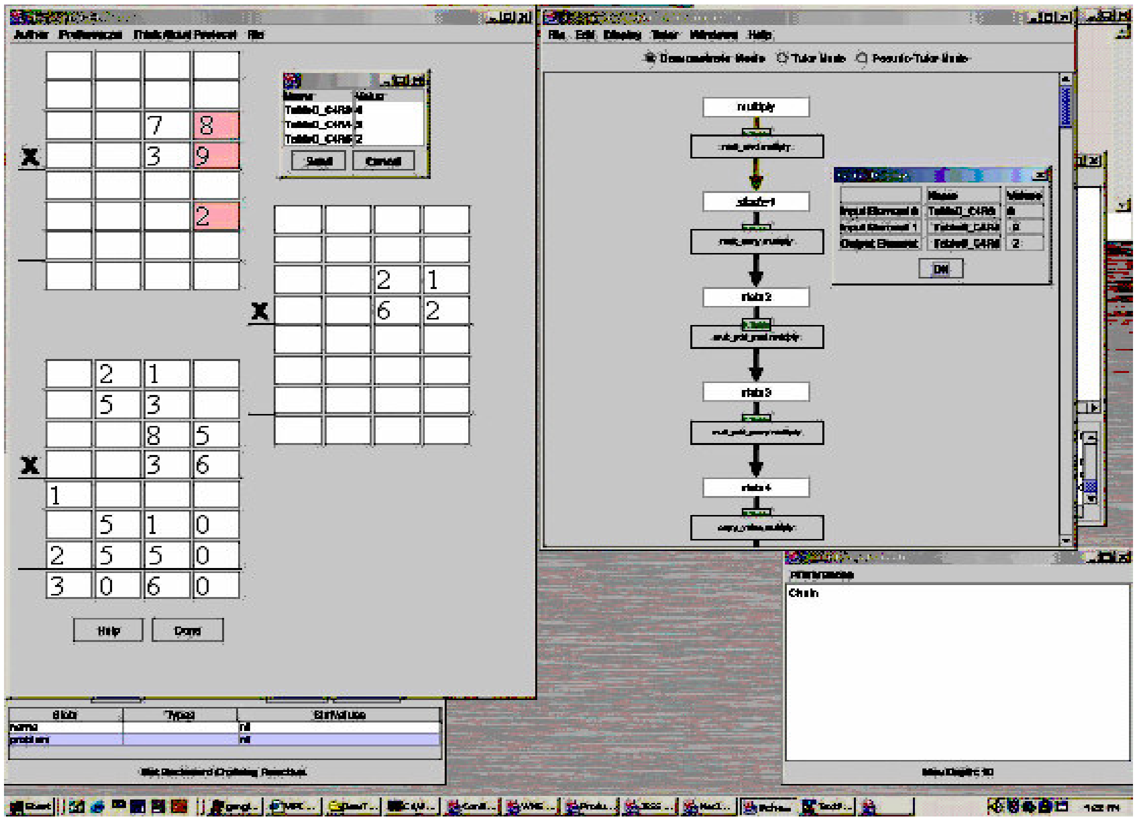
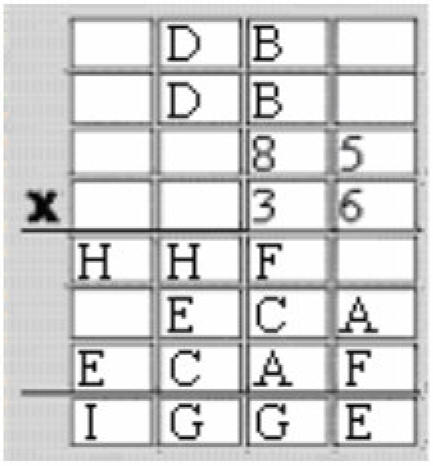


Figure 1 – Example Markup and Behavior Recorder

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

Figure 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.

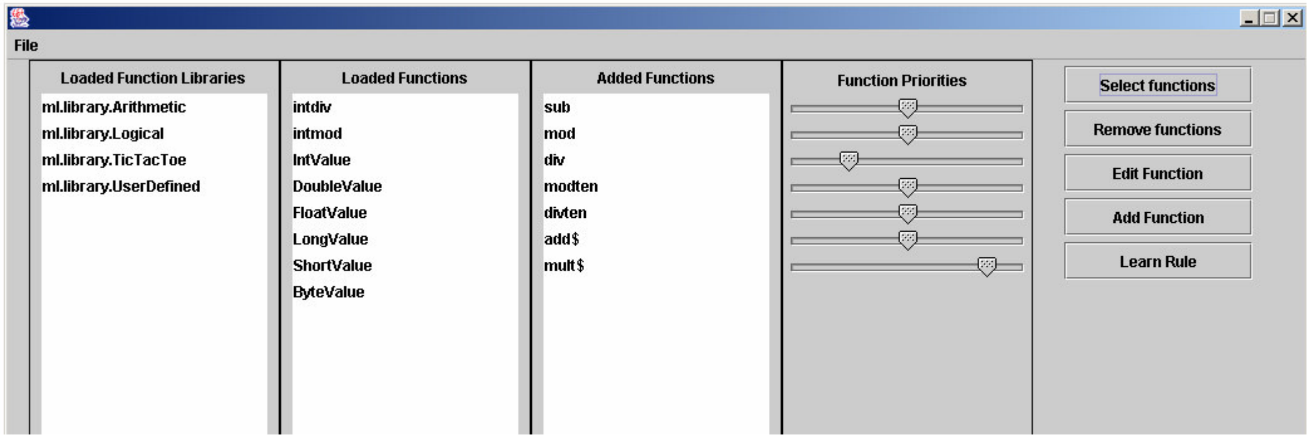


Figure 3 – Function Selection Dialog

The tutoring systems capable of being developed by the Cognitive Tutor Authoring Tools 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 Cognitive Tutor Authoring Tools 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 (Choksey & Heffernan, 2003) (Anderson & Pellitier, 1991) of the Authoring Tools to validate such rules.

3 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 (Quinlan, 1993), FFOIL (Quinlan, 1996) and PROGOL (Muggleton, 1995) 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 (1985) work with state space searches (Korf, 1985). 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.

4 Left-Hand Side Search Algorithm

The algorithm described above generates what are considered the right-hand side of JESS (Java Expert System Shell) 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) (Choksey & Heffernan, 2003). 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.

```
;; 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))
)
```

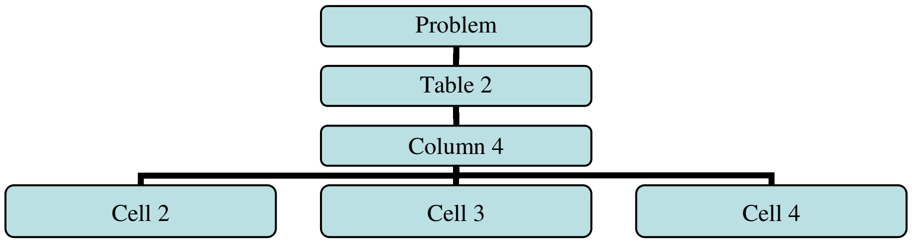
Figure 4 – An actual JESS rule that the system learned. The order of the conditionals on the left hand side has been changed, and indentation added, to make the rule easier to understand.

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.

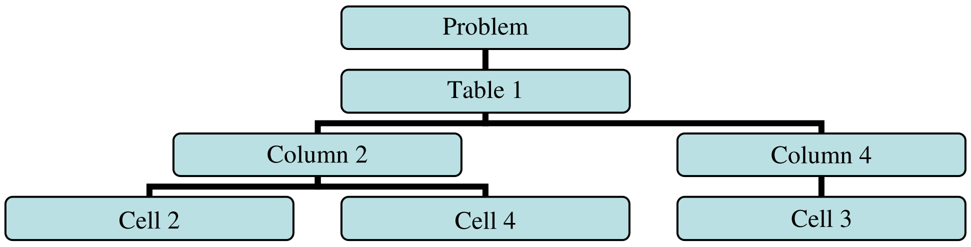
To demonstrate how we automatically generate the left-hand side, we will step through an example JESS rule, given in Figure 4. This "Multiply, Mod 10" rule occurs in the multi-column multiplication problem described below. Left-hand side generation is conducted by first finding all paths searching from 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.

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 here:

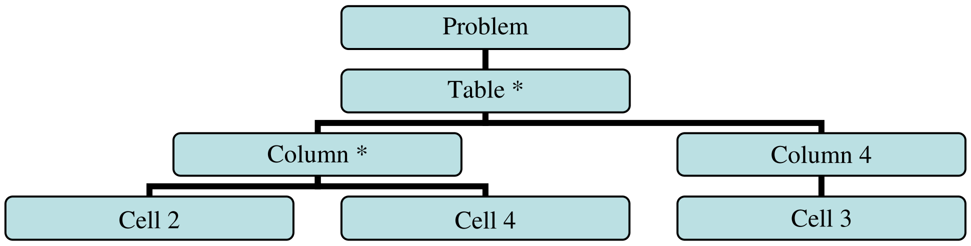


A different set of paths from top to the inputs occurs in the second instance of rule A that occurs in the 2nd column, 7th row. In this example we see that the first input and second input are not always in the same column, but the 2nd input and the output are in the same column as shown graphical here:



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:



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 Cognitive Tutor Authoring Tools. 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 (Choksey & Heffernan, 2003).

5 Methods/Experiments

5.1 Experiment #1: Multi-Column Multiplication

The goal of the 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.

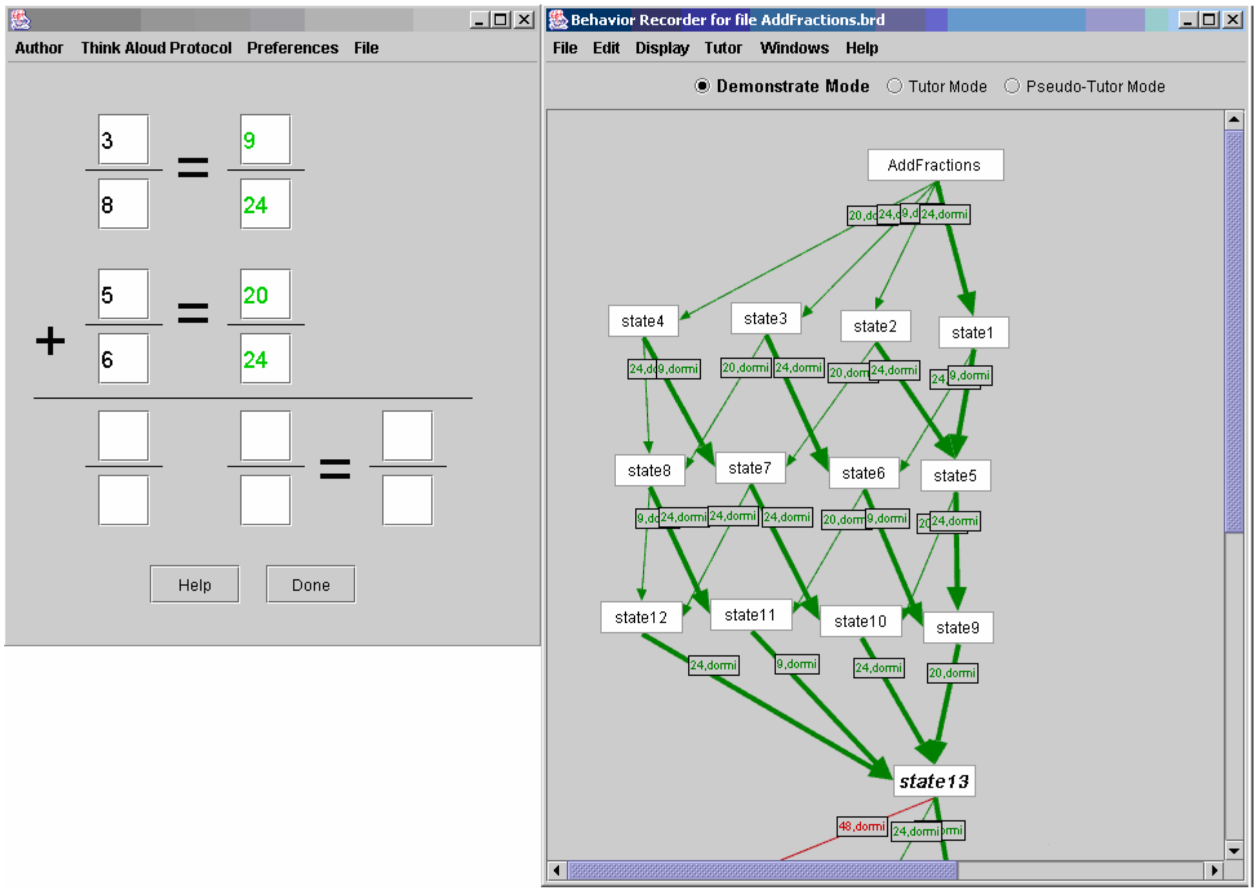
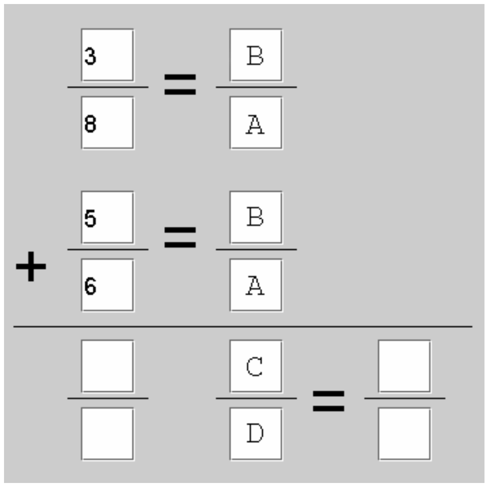


Figure 5 – Fraction Addition Problem

5.2 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 6. The multiple solution paths that are displayed in the Behavior Recorder allow the student to enter the values in any order they wish.



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

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

Figure 6 – Fraction Addition Rules

5.3 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 7). 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)

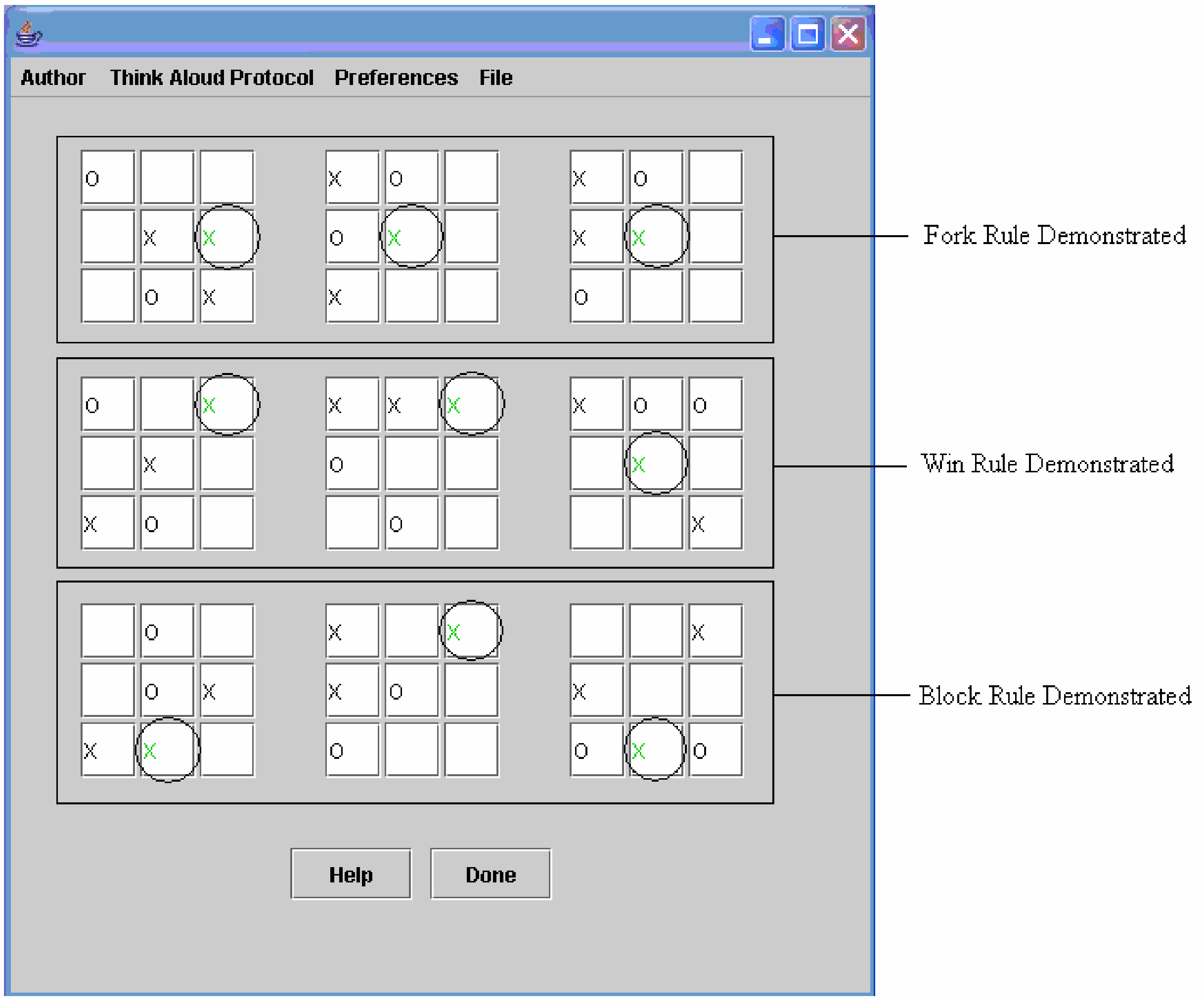


Figure 7 – Tic-Tac-Toe Problem

6 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.

6.1 Experiment #1: Multi-Column Multiplication

Rule Label	Rule Learned	Time Taken 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**	

6.2 Experiment #2: Fraction Addition

Rule Label	Rule Learned	Time Taken 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**	

6.3 Experiment #3: Tic-Tac-Toe

Rule Learned	Time Taken 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**	

7 Discussion

7.1 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 Cognitive Tutor Authoring Tools. 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 biased the search to find the most specific generalized rule that fits all of the provided examples. However, even the most specific generalizations are overly general for many of these rules. This over-generality is primarily due to the nature of the multi-column multiplication problem. Specifically, the inputs and output for many of the rules slide across multiple rows and columns. The left-hand-side search algorithm takes an all-or-nothing approach to generalizing at each level in working memory. For example, the search either matches an input in a single column, or it matches the input in any column. In this experiment, the ideal rules would have some level of generality between these two extremes. For example, some rules would be more accurate if the system was able to learn that a particular input only occurred in columns three and four. However, creating a system that could support this level of specificity would have some drawbacks. In this case, it would be difficult to generate rules that were not overly specific. It is certainly possible that the author would like the input to be found in any column even though there were only examples that showed the input in two different columns. Generating rules of appropriate specificity would require a nearly exhaustive set of examples, which would defeat the purpose of generalizing from the examples.

One possible solution to this problem using the current algorithm would be to design the rules so that there is a single rule associated with each student action. In this case, our algorithm would learn highly specific rules about how to generate the output at each cell on the interface. This would be analogous to writing a constant formula which calculates the value for each cell on the interface. 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.

7.2 Experiment #2: Fraction Addition

The purpose of this experiment was to test the system's behavior when there are multiple paths through working memory to the output elements. For this problem, cells were grouped together into fractions, and there were two different paths to each fraction on the interface. One path to a fraction was through its addition group, and another path was through a set of equivalent fractions. Overall, this experiment was successful in demonstrating that the system is able to take an optimal path through working memory to the inputs and output. While the left-hand-side search algorithm is able to produce many different correct trees through working memory to the examples, these rules are presented to the user in order from most specific to most general. The specificity of the rules is defined by the number of nodes that are in the tree representing the path through working memory. For example, a rule that leads to three fractions through a single addition group is considered to be better than a rule that leads to these fractions through three different equivalent fraction groups. When looking at the rules that were produced, it is clear that the desired effect was achieved. For example, the LCM rule uses equivalent fraction groupings to reach the desired inputs, but the Add rule uses an addition group to reach the inputs. This experiment was successful in demonstrating the system's ability to adjust to different working memory structures and to find optimal paths through working memory to the inputs.

7.3 Experiment #3: Tic-Tac-Toe

The purpose of the tic-tac-toe experiment was to demonstrate the system's capabilities in non-numeric domains. In order to learn these rules, it was necessary to group cells on the interface together into the possible winning combinations (triples). For example, there are four ways to reach the center square of the board: through the middle row, through the middle column, or through either diagonal. In this case there are many possible left-hand-sides that correctly retrieve the inputs and the output. With the exception of some minor over-generalities, the pattern matching on the left-hand-side was successful. However, this experiment revealed some other weaknesses in the system. The system assumes that all inputs will contain some value, and that the right-hand-side will act on these inputs to generate the output. These assumptions are violated in tic-tac-toe, as some inputs are actually cells that are required to be blank, and no inputs are needed to generate the constant output 'X'. Also, the left-hand-side search algorithm is not able to recognize when a specific input should always contain a particular value. In order to generate working rules for tic-tac-toe, it is necessary to note that some inputs should always contain an 'X' and that some inputs should always be blank. These shortcomings make it necessary for the author to modify the generated rules in order to create a working system, but the algorithm is still able to create an accurate structure for these rules that can be modified without significant difficulty. The pattern matching on the left-hand-side of the generated rules is, in fact, highly accurate. For example, the system was able to recognize that the two inputs and the output for the block rules should all occur in the same triple. There was one slight over-generality on the left-hand-side of the Fork rule. The system notices that two of the inputs should be in one triple and that two of the inputs should be found in another triple, but it is not able to recognize that the output should be the intersection of these two triples. Rather, it requires the output to be a part of only one of the two triples. While this experiment demonstrated some clear limitations of the system, it also provides evidence that the left-hand-side search algorithm is able to generate appropriate generalizations from complex working memory structures.

8 Conclusions and Future Work

8.1 Thesis Summary

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 Cognitive Tutor Authoring Tools (Koedinger, Aleven & Heffernan, 2003) 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.

8.2 Limitations and Future Work

Given that we are trying to learn rules with a rather brute force approach, our search is limited to relatively 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 right hand side search), while the system can be left to handle some of the details (like the left hand side search).

Another limitation to this approach is the possibility of creating rules that are overly general on the left-hand-side. An overly general left-hand-side will cause the rule to match unwanted input and output patterns, thereby allowing the rule to fire inappropriately. The current algorithm takes an all-or-nothing approach toward generating paths to inputs at each level. For example, the current system can learn that an input is in a specific column, or it can learn that an input can be found in any column, but it cannot recognize when an input may only be located in two of the four columns on an interface. Future work may involve adding specificity to the left-hand-side search so rules are not overly general. This leads to additional challenges, as the appropriate level of generality on the left-hand-side is highly domain specific. Another possibility for increasing the capability of the left-hand-side search would be to take into account the position of inputs and the output relative to one another. While the current search is able to determine if two inputs are always in the same column, it does not have the capability to recognize other relationships. For example, the current algorithm is not able to recognize when an input is always one column to the left of another.

Despite these limitations, this research is a step toward the goal of providing non-programmers with a tool that will allow them to develop Intelligent Tutoring Systems. This research has demonstrated that it is possible to generate working production rules from a small set of examples. While the rules may be overly general, enhancements to the system that has been developed may make it possible to create complex tutoring systems without expert system programming knowledge.

9 References

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).

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

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

Cypher, A., and Halbert, D.C. Editors. (1993) Watch what I do : Programming by Demonstration. Cambridge, MA: The MIT Press.

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.

Korf, R. (1985) Macro-operators: A weak method for learning. Artificial Intelligence, Vol. 26, No. 1.

Lieberman, H. Editor. (2001) Your Wish is My Command: Programming by Example. Morgan Kaufmann, San Francisco

Muggleton, S. (1995) Inverse Entailment and Progol. New Generation Computing, Special issue on Inductive Logic Programming, 13.

Quinlan, J.R. (1996). Learning first-order definitions of functions. Journal of Artificial Intelligence Research. 5. (pp 139-161)

Quinlan, J.R., and R.M. Cameron-Jones. (1993) FOIL: A Midterm Report. Sydney: University of Sydney.

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.

Appendix A – Function Libraries Used in Experiments

A.1 Arithmetic Library

```
public class Arithmetic {

        //"$" in the function name denotes a commutative function

        public Number modten(final Number X) {
                return new Integer(X.intValue() % 10);
        }

        public Number divten(final Number X) {
                return new Integer(X.intValue() / 10);
        }

        public Number add$(final Number X,final Number Y) {
                return new Double(X.doubleValue() + Y.doubleValue());
        }

        public Number mult$(final Number X,final Number Y) {
                return new Double(X.doubleValue() * Y.doubleValue());
        }

        public Number sub(final Number X,final Number Y) {
                return new Double(X.doubleValue() - Y.doubleValue());
        }

        public Number intdiv(final Number X,final Number Y) {
                if(Y.intValue() == 0) {
                        return new Integer(Integer.MAX_VALUE);
                }
                else {
                        return new Integer(X.intValue() / Y.intValue());
                }
        }

        public Number intmod(final Number X,final Number Y) {
                if(Y.intValue() == 0) {
                        return new Integer(Integer.MAX_VALUE);
                }
                else {
                        return new Integer(X.intValue() % Y.intValue());
                }
        }

        public Number div(final Number X,final Number Y) {
                if(Y.doubleValue() == 0) {
                        return new Double(Double.POSITIVE_INFINITY);
```

```
		}
		else {
			return new Double(X.doubleValue() / Y.doubleValue());
		}
	}

	public Number mod(final Number X,final Number Y) {
		if(Y.doubleValue() == 0) {
			return new Double(Double.POSITIVE_INFINITY);
		}
		else {
			return new Double(X.doubleValue() % Y.doubleValue());
		}
	}

	public Integer IntValue(final Number X) {
		return new Integer(X.intValue());
	}

	public Double DoubleValue(final Number X) {
		return new Double(X.doubleValue());
	}

	public Float FloatValue(final Number X) {
		return new Float(X.floatValue());
	}

	public Long LongValue(final Number X) {
		return new Long(X.longValue());
	}

	public Short ShortValue(final Number X) {
		return new Short(X.shortValue());
	}

	public Byte ByteValue(final Number X) {
		return new Byte(X.byteValue());
	}
}
```

A.2 Tic-Tac-Toe Library

```
public class TicTacToe {

        public String MarkX() {
                return "X";
        }

        public String MarkO() {
                return "O";
        }
}
```

Appendix B – Actual Production Rules Learned

B.1 Multi-Column Multiplication

(Note: Comments have been added to all rules)

Multiply Mod

```
(defrule mult_mod
        ;select the problem fact (the root of working memory)
        (addition (problem ?factMAIN__problem1))
        ;select any table from the interface
        ?factMAIN__problem1 <- (problem (interface-elements $?_blank_mf572
                ?factMAIN__table0 $?_blank_mf573))
        ;select any column from that table
        ?factMAIN__table0 <- (table (columns $?_blank_mf574 ?factMAIN__column0
                $?_blank_mf575))
        ;select the fourth column from that table
        ?factMAIN__table0 <- (table (columns ?_blank_580 ?_blank_581 ?_blank_582
                ?factMAIN__column4 $?_blank_mf583))
        ;from any column, select the fourth cell
        ?factMAIN__column0 <- (column (cells ?_blank_584 ?_blank_585 ?_blank_586
                ?factMAIN__cell4 $?_blank_mf587))
        ;from the fourth cell, get the value and assign it to ?v#0#1
        ?factMAIN__cell4 <- (cell (value ?v#0#1&~nil))
        ;from the fourth column, select the third cell
        ?factMAIN__column4 <- (column (cells ?_blank_588 ?_blank_589 ?factMAIN__cell3
                $?_blank_mf590))
        ;from the third cell, get the value and assign it to ?v#0#0
        ?factMAIN__cell3 <- (cell (value ?v#0#0&~nil))
        ;from any column, select any cell
        ?factMAIN__column0 <- (column (cells $?_blank_mf591 ?factMAIN__cell0
                $?_blank_mf592))
        ;from that cell, assign the cell name to ?sai_name so the output can be placed here
        ?factMAIN__cell0 <- (cell (name ?sai_name) (value nil))
        ?selection-action-input <- (selection-action-input)
=>
        (bind ?v#1#0 (call (new ml.library.Arithmetic) mult$ (new java.lang.Integer (integer
                ?v#0#0)) (new java.lang.Integer (integer ?v#0#1))))
        (bind ?v#1#1 (call (new ml.library.Arithmetic) modten (new java.lang.Integer (integer
                ?v#1#0))))
        ;insert the calculated output into the interface
        (modify ?factMAIN__cell0 (value ?v#1#1))
        (modify ?selection-action-input (selection ?sai_name) (action "UpdateTable") (input
                ?v#1#1))
)
```

Muliply Carry

```
(defrule mult_carry
        ;select the problem fact (the root of working memory)
        (addition (problem ?factMAIN__problem1))
        ;select any table from the interface
        ?factMAIN__problem1 <- (problem (interface-elements $?_blank_mf593
                ?factMAIN__table0 $?_blank_mf594))
        ;from that table, select any column
        ?factMAIN__table0 <- (table (columns $?_blank_mf595 ?factMAIN__column0
                $?_blank_mf596))
        ;from that table, select the third column and the fourth column
        ?factMAIN__table0 <- (table (columns ?_blank_597 ?_blank_598 ?factMAIN__column3
                ?factMAIN__column4 $?_blank_mf599))
        ;from any column, select the fourth cell
        ?factMAIN__column0 <- (column (cells ?_blank_600 ?_blank_601 ?_blank_602
                ?factMAIN__cell4 $?_blank_mf603))
        ;from that cell, select the value and assign it to ?v#0#1
        ?factMAIN__cell4 <- (cell (value ?v#0#1&~nil))
        ;from the fourth column, select the third cell
        ?factMAIN__column4 <- (column (cells ?_blank_604 ?_blank_605 ?factMAIN__cell3
                $?_blank_mf606))
        ;from that cell, select the value and assign it to ?v#0#0
        ?factMAIN__cell3 <- (cell (value ?v#0#0&~nil))
        ;from the third column, select any cell
        ?factMAIN__column3 <- (column (cells $?_blank_mf607 ?factMAIN__cell0
                $?_blank_mf608))
        ;from that cell, assign the cell name to ?sai_name so the output can be placed here
        ?factMAIN__cell0 <- (cell (name ?sai_name) (value nil))
        ?selection-action-input <- (selection-action-input)
=>
        (bind ?v#1#0 (call (new ml.library.Arithmetic) mult$ (new java.lang.Integer (integer
                ?v#0#0)) (new java.lang.Integer (integer ?v#0#1))))
        (bind ?v#1#1 (call (new ml.library.Arithmetic) divten (new java.lang.Integer (integer
                ?v#1#0))))
        ;insert the calculated output into the interface
        (modify ?factMAIN__cell0 (value ?v#1#1))
        (modify ?selection-action-input (selection ?sai_name) (action "UpdateTable") (input
                ?v#1#1))
)
```

Multiply Add Mod

```
(defrule mult_add_mod
        ;select the problem fact (the root of working memory)
        (addition (problem ?factMAIN__problem1))
        ;select any table from the interface
        ?factMAIN__problem1 <- (problem (interface-elements $?_blank_mf928
                ?factMAIN__tableW $?_blank_mf929))
        ;from that table, select any column
        ?factMAIN__tableW <- (table (columns $?_blank_mf930 ?factMAIN__column0
                $?_blank_mf931))
        ;from that table, select any column
        ?factMAIN__tableW <- (table (columns $?_blank_mf932 ?factMAIN__column2
                $?_blank_mf933))
        ;from that table, select the third column
        ?factMAIN__tableW <- (table (columns ?_blank_936 ?_blank_937
                ?factMAIN__column3 $?_blank_mf938))
        ;from any column, select the fourth cell
        ?factMAIN__column0 <- (column (cells ?_blank_939 ?_blank_940 ?_blank_941
                ?factMAIN__cell4 $?_blank_mf942))
        ;from that cell, select the value and assign it to ?v#0#0
        ?factMAIN__cell4 <- (cell (value ?v#0#0&~nil))
        ;from the third column, select any cell
        ?factMAIN__column3 <- (column (cells $?_blank_mf943 ?factMAIN__cell0
                $?_blank_mf944))
        ;from the third column, select the third cell
        ?factMAIN__column3 <- (column (cells ?_blank_945 ?_blank_946 ?factMAIN__cell3
                $?_blank_mf947))
        ;from any cell in the third column, select the value and assign it to ?v#0#2
        ?factMAIN__cell0 <- (cell (value ?v#0#2&~nil))
        ;from the third cell in the third column, select the value and assign it to ?v#0#1
        ?factMAIN__cell3 <- (cell (value ?v#0#1&~nil))
        ;from any column, select any cell
        ?factMAIN__column2 <- (column (cells $?_blank_mf948 ?factMAIN__cellW
                $?_blank_mf949))
        ;from that cell, assign the cell name to ?sai_name so the output can be placed here
        ?factMAIN__cellW <- (cell (name ?sai_name) (value nil))
        ?selection-action-input <- (selection-action-input)
=>
        (bind ?v#1#0 (call (new ml.library.Arithmetic) mult$ (new java.lang.Integer (integer
                ?v#0#1)) (new java.lang.Integer (integer ?v#0#2))))
        (bind ?v#1#1 (call (new ml.library.Arithmetic) add$ (new java.lang.Integer (integer
                ?v#0#0)) (new java.lang.Integer (integer ?v#1#0))))
        (bind ?v#1#2 (call (new ml.library.Arithmetic) modten (new java.lang.Integer (integer
                ?v#1#1))))
        ;insert the calculated output into the interface
        (modify ?factMAIN__cellW (value ?v#1#2))
        (modify ?selection-action-input (selection ?sai_name) (action "UpdateTable") (input
                ?v#1#2)))
```

Multiply Add Carry

```
(defrule mult_add_carry
        ;select the problem fact (the root of working memory)
        (addition (problem ?factMAIN__problem1))
        ;select any table from the interface
        ?factMAIN__problem1 <- (problem (interface-elements $?_blank_mf68
                ?factMAIN__tableW $?_blank_mf69))
        ;from that table, select any column
        ?factMAIN__tableW <- (table (columns $?_blank_mf70 ?factMAIN__column0
                $?_blank_mf71))
        ;from that table, select the second and third column
        ?factMAIN__tableW <- (table (columns ?_blank_72 ?factMAIN__column2
                ?factMAIN__column3 $?_blank_mf73))
        ;from any column, select the fourth cell
        ?factMAIN__column0 <- (column (cells ?_blank_74 ?_blank_75 ?_blank_76
                ?factMAIN__cell4 $?_blank_mf77))
        ;from that cell, select the value and assign it to ?v#0#0
        ?factMAIN__cell4 <- (cell (value ?v#0#0&~nil))
        ;from the third column, select any cell
        ?factMAIN__column3 <- (column (cells $?_blank_mf78 ?factMAIN__cell0
                $?_blank_mf79))
        ;from the third column, select the third cell
        ?factMAIN__column3 <- (column (cells ?_blank_80 ?_blank_81 ?factMAIN__cell3
                $?_blank_mf82))
        ;from any cell in the third column, select the value and assign it to ?v#0#2
        ?factMAIN__cell0 <- (cell (value ?v#0#2&~nil))
        ;from the third cell in the third column, select the value and assign it to ?v#0#1
        ?factMAIN__cell3 <- (cell (value ?v#0#1&~nil))
        ;from the second column, select any cell
        ?factMAIN__column2 <- (column (cells $?_blank_mf83 ?factMAIN__cellW
                $?_blank_mf84))
        ;from that cell, assign the cell name to ?sai_name so the output can be placed here
        ?factMAIN__cellW <- (cell (name ?sai_name) (value nil))
        ?selection-action-input <- (selection-action-input)
=>
        (bind ?v#1#0 (call (new ml.library.Arithmetic) mult$ (new java.lang.Integer (integer
                ?v#0#1)) (new java.lang.Integer (integer ?v#0#2))))
        (bind ?v#1#1 (call (new ml.library.Arithmetic) add$ (new java.lang.Integer (integer
                ?v#0#0)) (new java.lang.Integer (integer ?v#1#0))))
        (bind ?v#1#2 (call (new ml.library.Arithmetic) divten (new java.lang.Integer (integer
                ?v#1#1))))
        ;insert the calculated output into the interface
        (modify ?factMAIN__cellW (value ?v#1#2))
        (modify ?selection-action-input (selection ?sai_name) (action "UpdateTable") (input
                ?v#1#2))
)
```

Copy Value

```
(defrule copy_value
        ;select the problem fact (the root of working memory)
        (addition (problem ?factMAIN__problem1))
        ;select any table from the interface
        ?factMAIN__problem1 <- (problem (interface-elements $?_blank_mf950
                ?factMAIN__tableW $?_blank_mf951))
        ;select any column from that table
        ?factMAIN__tableW <- (table (columns $?_blank_mf952 ?factMAIN__columnW1
                $?_blank_mf953))
        ;select any column from that table
        ?factMAIN__tableW <- (table (columns $?_blank_mf953 ?factMAIN__columnW2
                $?_blank_mf954))
        ;select any cell from the first column that was selected
        ?factMAIN__columnW1 <- (column (cells $?_blank_mf955 ?factMAIN__cellW1
                $?_blank_mf956))
        ;select the value from that cell and assign it to ?v#0#0
        ?factMAIN__cellW1 <- (cell (value ?v#0#0&~nil))
        ;select any cell from the second column that was selected
        ?factMAIN__columnW2 <- (column (cells $?_blank_mf957 ?factMAIN__cellW1
                $?_blank_mf958))
        ;from that cell, assign the cell name to ?sai_name so the output can be placed here
        ?factMAIN__cellW2 <- (cell (name ?sai_name) (value nil))
        ?selection-action-input <- (selection-action-input)
=>
        (bind ?v#0#1 (call (new ml.library.Arithmetic) IntValue (new java.lang.Integer (integer
                ?v#0#0))))
        ;insert the calculated output into the interface
        (modify ?factMAIN__cellW2 (value ?v#0#1))
        (modify ?selection-action-input (selection ?sai_name) (action "UpdateTable") (input
                ?v#0#1))
)
```

Mark Zero

```
(defrule mark_zero
        ;select the problem fact (the root of working memory)
        (addition (problem ?factMAIN__problem1))
        ;select any table from the interface
        ?factMAIN__problem1 <- (problem (interface-elements $?_blank_mf956
                ?factMAIN__tableW $?_blank_mf957))
        ;from that table, select the third and fourth columns
        ?factMAIN__tableW <- (table (columns ?_blank_958 ?_blank_959
                ?factMAIN__column3 ?factMAIN__column4 $?_blank_mf960))
        ;from the fourth column, select the sixth cell
        ?factMAIN__column4 <- (column (cells ?_blank_961 ?_blank_962 ?_blank_963
                ?_blank_964 ?_blank_965 ?factMAIN__cell6 $?_blank_mf966))
        ;from that cell, select the value and assign it to ?v#0#0
        ?factMAIN__cell6 <- (cell (value ?v#0#0&~nil))
        ;from the third column, select the fifth cell
        ?factMAIN__column3 <- (column (cells ?_blank_967 ?_blank_968 ?_blank_969
                ?_blank_970 ?factMAIN__cell5 $?_blank_mf971))
        ;from that cell, assign the cell name to ?sai_name so the output can be placed here
        ?factMAIN__cell5 <- (cell (name ?sai_name) (value nil))
        ?selection-action-input <- (selection-action-input)
=>
        (bind ?v#1#0 (call (new ml.library.Arithmetic) intdiv (new java.lang.Integer (integer
                ?v#0#0)) (new java.lang.Integer (integer ?v#0#0))))
        (bind ?v#1#1 (call (new ml.library.Arithmetic) intmod (new java.lang.Integer (integer
                ?v#0#0)) (new java.lang.Integer (integer ?v#1#0))))
        ;insert the calculated output into the interface
        (modify ?factMAIN__cell5 (value ?v#1#1))
        (modify ?selection-action-input (selection ?sai_name) (action "UpdateTable") (input
                ?v#1#1))
)
```

Add Mod

```
(defrule add_mod
        ;select the problem fact (the root of working memory)
        (addition (problem ?factMAIN__problem1))
        ;select any table from the interface
        ?factMAIN__problem1 <- (problem (interface-elements $?_blank_mf972
                ?factMAIN__tableW $?_blank_mf973))
        ;from that table, select any column
        ?factMAIN__tableW <- (table (columns $?_blank_mf974 ?factMAIN__columnW
                $?_blank_mf975))
        ;from that column, select cells five, six, seven, and eight
        ?factMAIN__columnW <- (column (cells ?_blank_976 ?_blank_977 ?_blank_978
                ?_blank_979 ?factMAIN__cell5 ?factMAIN__cell6 ?factMAIN__cell7
                ?factMAIN__cell8 $?_blank_mf980))
        ;from cell five, select the value and assign it to ?v#0#2
        ?factMAIN__cell5 <- (cell (value ?v#0#2&~nil))
        ;from cell seven, select the value and assign it to ?v#0#1
        ?factMAIN__cell7 <- (cell (value ?v#0#1&~nil))
        ;from cell six, select the value and assign it to ?v#0#0
        ?factMAIN__cell6 <- (cell (value ?v#0#0&~nil))
        ;from cell eight, assign the name to ?sai_name so the output can be placed here
        ?factMAIN__cell8 <- (cell (name ?sai_name) (value nil))
        ?selection-action-input <- (selection-action-input)
=>
        (bind ?v#1#0 (call (new ml.library.Arithmetic) add$ (new java.lang.Integer (integer
                ?v#0#0)) (new java.lang.Integer (integer ?v#0#1))))
        (bind ?v#1#1 (call (new ml.library.Arithmetic) add$ (new java.lang.Integer (integer
                ?v#0#2)) (new java.lang.Integer (integer ?v#1#0))))
        (bind ?v#1#2 (call (new ml.library.Arithmetic) modten (new java.lang.Integer (integer
                ?v#1#1))))
        ;insert the calculated output into the interface
        (modify ?factMAIN__cell8 (value ?v#1#2))
        (modify ?selection-action-input (selection ?sai_name) (action "UpdateTable") (input
                ?v#1#2))
)
```

Add Carry

```
(defrule add_carry
        ;select the problem fact (the root of working memory)
         (addition (problem ?factMAIN__problem1))
        ;select any table from the interface
        ?factMAIN__problem1 <- (problem (interface-elements $?_blank_mf981
                ?factMAIN__tableW $?_blank_mf982))
        ;from that table, select any column
        ?factMAIN__tableW <- (table (columns $?_blank_mf983 ?factMAIN__columnW
                $?_blank_mf984))
        ;from that column, select cells five, six, and seven
        ?factMAIN__columnW <- (column (cells ?_blank_985 ?_blank_986 ?_blank_987
                ?_blank_988 ?factMAIN__cell5 ?factMAIN__cell6 ?factMAIN__cell7
                $?_blank_mf989))
        ;from cell seven, select the value and assign it to ?v#0#2
        ?factMAIN__cell7 <- (cell (value ?v#0#2&~nil))
        ;from cell six, select the value and assign it to ?v#0#1
        ?factMAIN__cell6 <- (cell (value ?v#0#1&~nil))
        ;from cell five, assign the name to ?sai_name so the output can be placed here
        ?factMAIN__cell5 <- (cell (name ?sai_name) (value nil))
        ?selection-action-input <- (selection-action-input)
=>
        (bind ?v#1#0 (call (new ml.library.Arithmetic) add$ (new java.lang.Integer (integer
                ?v#0#1)) (new java.lang.Integer (integer ?v#0#2))))
        (bind ?v#1#1 (call (new ml.library.Arithmetic) divten (new java.lang.Integer (integer
                ?v#1#0))))
        ;insert the calculated output into the interface
        (modify ?factMAIN__cell5 (value ?v#1#1))
        (modify ?selection-action-input (selection ?sai_name) (action "UpdateTable") (input
                ?v#1#1))
)
```

Add Final

```
(defrule add_final
        ;select the problem fact (the root of working memory)
        (addition (problem ?factMAIN__problem1))
        ;select any table from the interface
        ?factMAIN__problem1 <- (problem (interface-elements $?_blank_mf990
                ?factMAIN__tableW $?_blank_mf991))
        ;from that table, select the first column
        ?factMAIN__tableW <- (table (columns ?factMAIN__column1 $?_blank_mf992))
        ;from the first column, select cells five, seven, and eight
        ?factMAIN__column1 <- (column (cells ?_blank_993 ?_blank_994 ?_blank_995
                ?_blank_996 ?factMAIN__cell5 ?_blank_997 ?factMAIN__cell7
                ?factMAIN__cell8 $?_blank_mf998))
        ;from the seventh cell, select the value and assign it to ?v#0#1
        ?factMAIN__cell7 <- (cell (value ?v#0#1&~nil))
        ;from the fifth cell, select the value and assign it to ?v#0#0
        ?factMAIN__cell5 <- (cell (value ?v#0#0&~nil))
        ;from the eighth cell, assign the name to ?sai_name so the output can be placed here
        ?factMAIN__cell8 <- (cell (name ?sai_name) (value nil))
        ?selection-action-input <- (selection-action-input)
=>
        (bind ?v#1#0 (call (new ml.library.Arithmetic) add$ (new java.lang.Integer (integer
                ?v#0#0)) (new java.lang.Integer (integer ?v#0#1))))
        (bind ?v#1#1 (call (new ml.library.Arithmetic) modten (new java.lang.Integer (integer
                ?v#1#0))))
        ;insert the calculated output into the interface
        (modify ?factMAIN__cell8 (value ?v#1#1))
        (modify ?selection-action-input (selection ?sai_name) (action "UpdateTable") (input
                ?v#1#1))
)
```

B.2 Fraction Addition

LCM Rule

```
(defrule LCM
        ;select the problem fact (the root of working memory)
        (fraction-addition (problem ?factMAIN__problem1))
        ;select any list of equivalent fraction groups from the problem
        ?factMAIN__problem1 <- (problem (equivalent-fractions $?_blank_mf572
                ?factMAIN__equal-fractions0 $?_blank_mf573))
        ;select another list of equivalent fraction groups from the problem
        ?factMAIN__problem1 <- (problem (equivalent-fractions $?_blank_mf572
                ?factMAIN__equal-fractions1 $?_blank_mf573))
        ;from the first group of equal fractions, select the first and second fractions
        ?factMAIN__equal-fractions0 <- (equal-fractions ?factMAIN__fraction0
                ?factMAIN__fraction1)
        ;from the first fraction, select the denominator
        ?factMAIN__fraction0 <- (fraction (denominator ?factMAIN__textField0))
        ;from the denominator’s textField, select the value and assign it to ?v#0#0
        ?factMAIN__textField0 <- (textField (value ?v#0#0&~nil))
        ;from the second fraction, select the denominator
        ?factMAIN__fraction1 <- (fraction (denominator ?factMAIN__textField1))
        ;from the denominator’s textField, select the name and set it to ?sai_name so the output
                ;can be placed here
        ?factMAIN__textField1 <- (textField (name ?sai_name) (value nil))
        ;from the second group of equal fractions, select the first fraction
        ?factMAIN__equal-fractions1 <- (equal-fractions ?factMAIN__fraction2
                $?_blank_mf574)
        ;from that fraction, select the denominator
        ?factMAIN__fraction2 <- (fraction (denominator ?factMAIN__textField2))
        ;from the denominator’s textField, select the value and assign it to ?v#0#1
        ?factMAIN__textField2 <- (textField (value ?v#0#1&~nil))
        ?selection-action-input <- (selection-action-input)
=>
        (bind ?v#1#0 (call (new ml.library.Arithmetic) lcm$ (new java.lang.Integer (integer
                ?v#0#0)) (new java.lang.Integer (integer ?v#0#1))))
        ;insert the calculated output into the interface
        (modify ?factMAIN__textField1 (value ?v#1#0))
        (modify ?selection-action-input (selection ?sai_name) (action "UpdateTextField") (input
                ?v#1#0))
)
```

Div-Mult Rule

```
(defrule div-mult
        ;select the problem fact (the root of working memory)
        (fraction-addition (problem ?factMAIN__problem1))
        ;select any list of equivalent fraction groups from the problem
        ?factMAIN__problem1 <- (problem (equivalent-fractions $?_blank_mf572
                ?factMAIN__equal-fractions0 $?_blank_mf573))
        ;from this group, select the first and second fractions
        ?factMAIN__equal-fractions0 <- (equal-fractions ?factMAIN__fraction0
                ?factMAIN__fraction1)
        ;from the first fraction, select the numerator
        ?factMAIN__fraction0 <- (fraction (numerator ?factMAIN__textField0))
        ;from the numerator, select the value in the textField and assign it to ?v#0#0
        ?factMAIN__textField0 <- (textField (value ?v#0#0&~nil))
        ;from the first fraction, select the denomintor
        ?factMAIN__fraction0 <- (fraction (denominator ?factMAIN__textField1))
        ;from the denominator, select the value in the textField and assign it to ?v#0#1
        ?factMAIN__textField1 <- (textField (value ?v#0#1&~nil))
        ;from the second fraction, select the numerator
        ?factMAIN__fraction1 <- (fraction (numerator ?factMAIN__textField2))
        ;from the numerator, select the name, and set it to ?sai_name so the output can be placed
                ;here
        ?factMAIN__textField2 <- (textField (name ?sai_name) (value nil))
        ;from the second fraction, select the denominator
        ?factMAIN__fraction1 <- (fraction (denominator ?factMAIN__textField3))
        ;from the denominator, select the value in the textField and assign it to ?v#0#2
        ?factMAIN__textField3 <- (textField (value ?v#0#2&~nil))
        ?selection-action-input <- (selection-action-input)
=>
        (bind ?v#1#0 (call (new ml.library.Arithmetic) div (new java.lang.Integer (integer
                ?v#0#2)) (new java.lang.Integer (integer ?v#0#1))))
        (bind ?v#1#1 (call (new ml.library.Arithmetic) mult$ (new java.lang.Integer (integer
                ?v#0#0)) (new java.lang.Integer (integer ?v#1#0))))
        ;insert the calculated output into the interface
        (modify ?factMAIN__textField2 (value ?v#1#1))
        (modify ?selection-action-input (selection ?sai_name) (action "UpdateTextField") (input
                ?v#1#1))
)
```

Add Rule

```
 (defrule add
        ;select the problem fact (the root of working memory)
         (fraction-addition (problem ?factMAIN__problem1))
        ;select the second addition group of fractions
        ?factMAIN__problem1 <- (problem (addition-groups ?_blank_572
                ?factMAIN__addition-group0 $?_blank_mf573))
        ;from this addition group, select the first, second, and third fractions
        ?factMAIN__addition-group0 <- (addition-group (fractions ?factMAIN__fraction0
                ?factMAIN__fraction1 ?factMAIN__fraction2))
        ;from the first fraction, select the numerator
        ?factMAIN__fraction0 <- (fraction (numerator ?factMAIN__textField0))
        ;from this numerator, select the value in the textField and assign it to ?v#0#0
        ?factMAIN__textField0 <- (textField (value ?v#0#0&~nil))
        ;from the second fraction, select the numerator
        ?factMAIN__fraction1 <- (fraction (numerator ?factMAIN__textField1))
        ;from this numerator, select the value in the textField and assign it to ?v#0#1
        ?factMAIN__textField1 <- (textField (value ?v#0#1&~nil))
        ;from the third fraction, select the numerator
        ?factMAIN__fraction2 <- (fraction (numerator ?factMAIN__textField2))
        ;from this numerator, select the name of the textField and assign it to ?sai_name so the
                ;output can be placed here.
        ?factMAIN__textField2 <- (textField (name ?sai_name) (value nil))
        ?selection-action-input <- (selection-action-input)
=>
        (bind ?v#1#0 (call (new ml.library.Arithmetic) add (new java.lang.Integer (integer
                ?v#0#0)) (new java.lang.Integer (integer ?v#0#1))))
        ;insert the calculated output into the interface
        (modify ?factMAIN__textField2 (value ?v#1#0))
        (modify ?selection-action-input (selection ?sai_name) (action "UpdateTextField") (input
                ?v#1#0))
)
```

Copy Value Rule

```
(defrule copy-value
        ;select the problem fact (the root of working memory)
         (fraction-addition (problem ?factMAIN__problem1))]
        ;select the second addition group of fractions
        ?factMAIN__problem1 <- (problem (addition-groups ?_blank_572
                ?factMAIN__addition-group0 $?_blank_mf573))
        ;from that group of fractions, select the first, second, and third fractions
        ?factMAIN__addition-group0 <- (addition-group (fractions ?factMAIN__fraction0
                ?factMAIN__fraction1 ?factMAIN__fraction2))
        ;from the first fraction, select the denominator
        ?factMAIN__fraction0 <- (fraction (denominator ?factMAIN__textField0))
        ;from that denominator, select the value in the textField and assign it to ?v#0#0
        ?factMAIN__textField0 <- (textField (value ?v#0#0&~nil))
        ;from the second fraction, select the denominator
        ?factMAIN__fraction1 <- (fraction (denominator ?factMAIN__textField1))
        ;from that denominator, select the value in the textField and assign it to ?v#0#1
        ?factMAIN__textField1 <- (textField (value ?v#0#1&~nil))
        ;from the third fraction, select the denominator
        ?factMAIN__fraction2 <- (fraction (denominator ?factMAIN__textField2))
        ;from that denominator, select the name of the textField and assign it to ?sai_name so the
                ;output can be placed here
        ?factMAIN__textField2 <- (textField (name ?sai_name) (value nil))
        ?selection-action-input <- (selection-action-input)
=>
        (bind ?v#1#0 (call (new ml.library.Arithmetic) intValue (new java.lang.Integer (integer
                ?v#0#0))))
        ;insert the calculated output into the interface
        (modify ?factMAIN__textField2 (value ?v#1#0))
        (modify ?selection-action-input (selection ?sai_name) (action "UpdateTextField") (input
                ?v#1#0))
)
```

B.3 Tic-Tac-Toe

Win Rule

```
(defrule Win
        ;select the problem fact (the root of working memory)
         (MAIN::addition (problem ?factMAIN__problem1 ))
        ;select any tic-tac-toe game from the problem
        ?factMAIN__problem1 <- (MAIN::problem (ttt-games $? ?factMAIN__ttt-gameW1000
                $? ))
        ;from that tic-tac-toe game, select one of the triples
        ?factMAIN__ttt-gameW1000 <- (MAIN::ttt-game (triples $? ?factMAIN__tripleW1001
                $? ))
        ;from that triple, select any of the cells
        ?factMAIN__tripleW1001 <- (MAIN::triple (cells $? ?factMAIN__cellW1002 $? ))
        ;from that triple, select any of the cells again
        ?factMAIN__tripleW1001 <- (MAIN::triple (cells $? ?factMAIN__cellW1003 $? ))
        ;from that triple, select a third cell
        ?factMAIN__tripleW1001 <- (MAIN::triple (cells $? ?factMAIN__cellW1004 $? ))
        ;from the first cell, select the value and assign it to ?v#0#0
        ?factMAIN__cellW1002 <- (MAIN::cell (value ?v#0#0&~nil))
        ;from the second cell, select the value and make sure it is not nil
        ?factMAIN__cellW1003 <- (MAIN::cell (value ~nil))
        ;select the name of the third cell and assign it to ?sai_name so the output can be placed
                ;here.
        ?factMAIN__cellW1004 <- (MAIN::cell (name ?sai_name) (value nil))
        ?selection-action-input <- (selection-action-input)
=>
        (bind ?v#0#2 ((new ml.library.TicTacToe) MarkX (new java.lang.Integer (integer
                ?v#0#0))))
        ;insert the calculated output into the interface
        (modify ?factMAIN__cellW1004 (value ?v#0#2))
        (modify ?selection-action-input (selection ?sai_name) (action "UpdateTable") (input
                ?v#0#2))
)
```

Block Rule

```
(defrule block
	;select the problem fact (the root of working memory)
	(MAIN::addition (problem ?factMAIN__problem1))
	;select any tic-tac-toe game from the problem
	?factMAIN__problem1 <- (MAIN::problem (ttt-games $? ?factMAIN__ttt-gameW1000
		$? ))
	;from this tic-tac-toe game, select any triple
	?factMAIN__ttt-gameW1000 <- (MAIN::ttt-game (triples $? ?factMAIN__tripleW1001
		$? ))
	;from that triple, select any cell
	?factMAIN__tripleW1001 <- (MAIN::triple (cells $? ?factMAIN__cellW1002 $? ))
	;from that triple, select a second cell
	?factMAIN__tripleW1001 <- (MAIN::triple (cells $? ?factMAIN__cellW1003 $? ))
	;from that triple, select a third cell
	?factMAIN__tripleW1001 <- (MAIN::triple (cells $? ?factMAIN__cellW1004 $? ))
	;from the first cell, select the value and assign it to ?v#0#0
	?factMAIN__cellW1002 <- (MAIN::cell (value ?v#0#0&~nil))
	;from the second cell, select the value and make sure it is not nil
	?factMAIN__cellW1003 <- (MAIN::cell (value ~nil))
	;select the name of the third cell and assign it to ?sai_name so the output can be placed
		;here
	?factMAIN__cellW1004 <- (MAIN::cell (name ?sai_name) (value nil))
	?selection-action-input <- (selection-action-input)
=>
	(bind ?v#0#2 ((new ml.library.TicTacToe) MarkX (new java.lang.Integer (integer
?v#0#0))))
	;insert the calculated output into the interface
	(modify ?factMAIN__cellW1004 (value ?v#0#2))
	(modify ?selection-action-input
		(selection ?sai_name)
		(action "UpdateTable")
		(input ?v#0#2)
	)
)
```

Play Center Rule

```
(defrule playcenter
        ;select the problem fact (the root of working memory)
        (MAIN::addition (problem ?factMAIN__problem1))
        ;select any tic-tac-toe game from the problem
        ?factMAIN__problem1 <- (MAIN::problem (ttt-games $? ?factMAIN__ttt-gameW1000
                $? ))
        ;from the tic-tac-toe game, select the second triple
        ?factMAIN__ttt-gameW1000 <- (MAIN::ttt-game (triples ? ?factMAIN__triple2 $? ))
        ;from this triple, select the second cell
        ?factMAIN__triple2 <- (MAIN::triple (cells ? ?factMAIN__cell2 ? ))
        ;select the name of the second cell and assign it to ?sai_name so the output can be placed
                ;here
        ?factMAIN__cell2 <- (MAIN::cell (name ?sai_name) (value nil))
        ?selection-action-input <- (selection-action-input)
=>
        (bind ?v#0#2 ((new ml.library.TicTacToe) MarkX (new java.lang.Integer (integer
                ?v#0#0))))
        ;insert the calculated output into the interface
        (modify ?factMAIN__cellW1004 (value ?v#0#2))
        (modify ?selection-action-input
        (selection ?sai_name)
        (action "UpdateTable")
        (input ?v#0#2))
)
```

Fork Rule

```
(defrule fork
        ;select the problem fact (the root of working memory)
        (MAIN::addition (problem ?factMAIN__problem1))
        ;select any tic-tac-toe game from the problem
        ?factMAIN__problem1 <- (MAIN::problem (ttt-games $? ?factMAIN__ttt-gameW1000
                $? ))
        ;from that tic-tac-toe game, select any triple
        ?factMAIN__ttt-gameW1000 <- (MAIN::ttt-game (triples $? ?factMAIN__tripleW1001
                $? ))
        ;from the tic-tac-toe game, select another triple
        ?factMAIN__ttt-gameW1000 <- (MAIN::ttt-game (triples $? ?factMAIN__tripleW1002
                $? ))
        ;from the first triple, select any cell
        ?factMAIN__tripleW1001 <- (MAIN::triple (cells $? ?factMAIN__cellW1002 $? ))
        ;from the first triple, select a second cell
        ?factMAIN__tripleW1001 <- (MAIN::triple (cells $? ?factMAIN__cellW1003 $? ))
        ;from the first triple, select a third cell
        ?factMAIN__tripleW1001 <- (MAIN::triple (cells $? ?factMAIN__cellW1004 $? ))
        ;from the second triple, select any cell
        ?factMAIN__tripleW1002 <- (MAIN::triple (cells $? ?factMAIN__cellW1005 $? ))
        ;from the second triple, select another cell
        ?factMAIN__tripleW1002 <- (MAIN::triple (cells $? ?factMAIN__cellW1006 $? ))
        ;select the value in the first cell and assign it to ?v#0#0
        ?factMAIN__cellW1002 <- (MAIN::cell (value ?v#0#0&~nil))
        ;select the value in the second cell and make sure it isn’t nil
        ?factMAIN__cellW1003 <- (MAIN::cell (value ~nil))
        ;select the name of the third cell and assign it to ?sai_name so the output can be placed
                ;here
        ?factMAIN__cellW1004 <- (MAIN::cell (name ?sai_name) (value nil))
        ;select the value in the fourth cell and make sure it isn’t nil
        ?factMAIN__cellW1005 <- (MAIN::cell (value ~nil))
        ;select the value in the fifth cell and make sure it isn’t nil
        ?factMAIN__cellW1006 <- (MAIN::cell (value ~nil))
        ?selection-action-input <- (selection-action-input)
=>
        (bind ?v#0#2 ((new ml.library.TicTacToe) MarkX (new java.lang.Integer (integer
                ?v#0#0))))
        ;insert the calculated output into the interface
        (modify ?factMAIN__cellW1004 (value ?v#0#2))
        (modify ?selection-action-input (selection ?sai_name) (action "UpdateTable") (input
                ?v#0#2))
)
```

Appendix C – Code Documentation

C.1 Main Algorithm

Class RuleGeneratingFrame

Main functionality:

This class controls the sliding probability window for the right-hand-side search. When the user indicates that they are ready to begin searching, the RuleGeneratingFrame creates an instance of BForce. It then calls the search method of the BForce object, gradually decreasing the probability window until the user decides to terminate the search, or until the search method indicates that a working sequence of operators and variable bindings have been found. At this point, the BForce object continues the left-hand-side search.

Other functionality:

The RuleGeneratingFrame also provides an interface which displays the examples that were provided for the current rule. Using this interface, the user can start and terminate the search for the desired rule. After a set of working rules are found, they are displayed to the user using the DisplayRules class. The user selects a rule from the DisplayRules dialog, and it is displayed in this interface. The user then has the option to test this rule on the given examples and to enter this rule into the production system.

Class BForce

Main functionality:

The purpose of this class is to perform the right-hand-side search that is described in Section 3. The main function of this class is the search function, which will recursively call itself until it has found a working sequence of operators and variable bindings, or until there are no more functions and variable bindings possible within the specified probability range. The probability range is specified by the RuleGeneratingFrame, and this range is gradually decreased until a working rule is found or until the user terminates the search. The search consists of two nested for loops. The outside loop iterates through each possible function. For each function that is selected, the inner loop iterates through all the possible variable bindings for the selected function. Initially, the only variables that can be inserted into the functions are the inputs that were specified by the author. Each time a new function is applied, the output of that function is added to the list of possible variable bindings. For each function and variable binding that is applied, the probability of the current set of operations is calculated and tested against the probability window. If the probability is within the window, then the output of the current function and variable binding is tested against the desired output of all the examples. If the outputs do not match, the search function calls itself in order to test if another function can be added to the current sequence that will be successful. If the probability is greater than the maximum probability, the function recursively calls itself in order to add a new operation. It is not necessary to test the current set of operations in this case because they were already tested in a previous iteration. Finally, if the probability is less than the minimum probability, the function is removed from the current set of operators, and no recursive call is made.

When the search function is successful in finding a sequence of operators and variable bindings that match all the examples, it continues by creating an instance of the LHS class. It then calls the printJessRules function of the LHS object, passing the information necessary to generate the right-hand-side of the rule.

Other functionality:

The BForce class also uses functions to iterate though the permutations of variable bindings that are available. The getFirstPermutation and the getNextPermutation are used by the search function to iterate through the possible variable bindings for each function.

Class LHS

Main functionality:

The LHS class takes control of the search process after the right-hand-side search has been completed successfully. The main function in the LHS class is the printJessRules function. This function is responsible for controlling the left-hand-side search and for generating the actual text of the resulting Jess rule. For each of the inputs, and for each of the examples, the system finds a set of concrete paths from the top of working memory to that input. These concrete paths are then sent to the MergeFactsPath class to generate a set of merged paths for each input. Each merged path represents a common path that can be taken from the problem fact to the input for all the examples. These paths are then passed to the MergeInputPaths class which creates a set of possible trees, each representing a Jess production rule. The MergeInputPaths class then converts these trees into actual Jess rules, and returns these rules to LHS class. The rules are then presented to the user in order of generality by the DisplayRules class.

Class MergeFactsPath

Main functionality:

The MergeFactsPath class is responsible for the first step of merging the concrete paths to each of the inputs and the output. The main function of this class is the mergePaths function. This function takes a list of concrete paths to an input and creates a generalized path to that input. The mergePaths function looks at each concrete path and determine where the paths vary. For example, the input may move across different columns, but may always be found in the third cell down. In this case, the mergePaths function will create a generalized path that matches any column and the third cell down. These merged paths are then passed on to the MergeInputPaths class which will create a set of merged trees.

Other functionality:

The MergeFactsPath class contains a function named searchPath which is used to generate a concrete path from one fact in working memory to another. This function is used to generate all the concrete paths from the problem fact to each input and output. This class also contains functions to generate a signature for each path through working memory. These signatures represent straight-line paths in working memory which will later be converted into trees for the MergeInputPaths class.

Class MergeInputPaths

Main functionality:

The MergeInputPaths class has two main responsibilities. First, this class is responsible for creating a set of merged trees from the merged paths that are provided to it. The second responsibility is to convert these trees into actual Jess rules. The first goal is accomplished by iteratively merging inputs into a merged tree. When merging a path into a tree, the system begins with the root, and it iteratively attempts to merge nodes. If two nodes are equal constants, then they will always be merged. If two nodes are equal wildcards, then the system checks each of the concrete paths at that point. If the concrete paths are always equal at that point, then the system branches and creates two trees: one that merges the nodes and one that splits them. Otherwise, the system splits the nodes and continues with the next input path.

To create a Jess rule from the resulting trees, the MergeInputPaths class uses two functions. The treeToJessRule and the treeToJessRuleRecursive functions iterate through each node of the tree generating the appropriate Jess code for each node. The code that is generated depends upon the type of the current node (variable or fixed). For example, to find the fourth column, the system generates "columns(? ? ? ?col4 $?)". In order to match any column, the system generates "columns($? ?colAny $?)".

Other functionality:

The main search algorithm of the MergeInputPaths class takes advantage of the mergePossible and checkConcretePaths functions when it is deciding whether or not to merge two nodes. The mergePossible function compares two nodes and returns one of three values. If the two nodes must be merged, the function returns 2. If the two nodes may or may not be merged, the function returns 1. If the two nodes cannot be merged, the function returns 0. The checkConcretePaths function is used to determine whether or not two wildcard nodes are able to be merged. For example, if Input#1 and Input#2 can both be found in any column, the nodes should only be merged if the two inputs are always found in the same column. The checkConcretePaths function returns true if all the concrete paths travel through the same point at the selected node, and it returns false otherwise.

C.2 Other Classes

Class FunctionClass

Main functionality:

The purpose of the FunctionClass is to manage the operators that can be used to generate the right-hand-side of the rule. This class stores a list of the all operators that were selected by the user as well as pertinent information about each operator. This class manages the function probabilities, and has a method which determines if a given function is commutative. It also contains methods that are able to determine the arguments required by each method as well as their return type. Finally, this class contains a function which allows for the invocation of each method in the list. The BForce class uses this class to perform the right-hand-side search.

Class DisplayRules

Main functionality:

The DisplayRules class provides a simple interface for displaying the learned rules to the user. Given that there may be more than one possible left-hand-side that matches the examples, it is possible for there to be multiple correct rules of varying generality. The DisplayRules class displays each of these rules to the user, and gives the user the option to select one of these rules.

Class MacroOperator

Main functionality:

The MacroOperator class provides functionality which allows a sequence of operators to be treated as a single function. This class stores the list of operators and variable bindings that are necessary to execute the macro-operator. It provides functions which are able to save and retrieve a macro-operator to or from a file. Like the FunctionClass object, the MacroOperator class provides methods to retrieve information about the arguments and return type of the macro-operator. Finally, this class provides a method to invoke the macro-operator on a list of arguments.

Class MLWindow

Main functionality:

The MLWindow class provides the user with an interface for selecting functions that can be used to learn the right-hand-side of the rule. The user is presented with a list of function libraries. By selecting a library, the user is able to add functions from that library to the list of functions to be used in the search. After the user has selected the functions, they are presented with a list of sliders that are used to adjust the probability that each function will be used. The user then indicates that they are ready to learn the rule, and the information provided by the user is passed to the RuleGeneratingFrame, and then to the BForce class to perform the right-hand-side search.

Additional functionality:

The MLWindow also gives a user the option to add new functions to a library or to edit existing functions. This functionality is included in the SourceEditor class described below.

Class SourceEditor

Main functionality:

This class provides the user with the option of adding or modifying a function in some function library. The user is presented with a simple interface containing a text area that is used to add or modify code. The user is able to compile any modifications that are made so they can confirm that the modified function will work within the system. Finally, the user is provided with the option of saving or discarding the changes that were made to the original function library.