

# Integrating Representation Learning and Skill Learning in a Human-Like Intelligent Agent

Nan Li

May 2013

Computer Science Department  
School of Computer Science  
Carnegie Mellon University  
Pittsburgh, PA

## **Thesis Committee:**

William W. Cohen (Co-Chair)

Kenneth R. Koedinger (Co-Chair)

Tom Mitchell

Pat Langley (Carnegie Mellon University, Silicon Valley Campus)

Raymond J. Mooney (The University of Texas at Austin)

*Submitted in partial fulfillment of the requirements  
for the degree of Doctor of Philosophy.*

Copyright © 2013 Nan Li

This research was sponsored by the National Science Foundation under grant numbers

**Keywords:** intelligent agent, learner modeling, representation learning, complex problem solving

*to my parents*



## Abstract

Building an intelligent agent that simulates human learning of math and science could potentially benefit both cognitive science, by contributing to the understanding of human learning, and artificial intelligence, by advancing the goal of creating human-level intelligence. However, constructing such a learning agent currently requires manual encoding of prior domain knowledge; in addition to being a poor model of human acquisition of prior knowledge, manual knowledge-encoding is both time-consuming and error-prone. Previous research has shown that one of the key factors that differentiates experts and novices is their different representations of knowledge. Experts view the world in terms of deep functional features, while novices view it in terms of shallow perceptual features. Moreover, since the performance of learning algorithms is sensitive to representation, the deep features are also important in achieving effective machine learning.

In this work, we propose an efficient algorithm that acquires representation knowledge in the form of “deep features” for specific domains, and demonstrate its effectiveness in the domain of algebra as well as synthetic domains. We integrate this algorithm into a learning agent, SimStudent, which learns procedural knowledge by observing a tutor solve sample problems, and by getting feedback while actively solving problems on its own. We show that learning representations enhances the generality of the learning agent by reducing the requirements for knowledge engineering. Moreover, we propose an approach that automatically discovers student models using the extended SimStudent. By fitting the discovered model to real student learning curve data, we show that the discovered model is better or as good as human-generated models, and demonstrate how the discovered model may be used to improve a tutoring system’s instructional strategy.



## **Acknowledgments**





# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Motivation . . . . .	1
1.2	Proposed Approach . . . . .	2
1.3	Main Contribution . . . . .	3
<b>2</b>	<b>A Brief Review of SimStudent</b>	<b>5</b>
2.1	Prior Knowledge . . . . .	5
2.2	Learning Task . . . . .	8
2.3	Production Rules . . . . .	9
2.4	Learning Mechanisms . . . . .	10
<b>3</b>	<b>Deep Feature Representation Learning</b>	<b>13</b>
3.1	Representation Learning as Grammar Induction . . . . .	14
3.1.1	A Brief Review of a pCFG Learner . . . . .	14
3.1.2	Feature Learning . . . . .	16
3.1.3	Transfer Learning . . . . .	18
3.2	Experimental Study . . . . .	20
3.2.1	Methods . . . . .	21
3.2.2	Measurements . . . . .	22
3.2.3	Experimental Results . . . . .	22
3.3	Discussion . . . . .	23
<b>4</b>	<b>Learning for Operator Functions</b>	<b>25</b>
4.1	Integrating Representation Learning into Skill Learning . . . . .	25
4.1.1	Extending the Perceptual Representation . . . . .	26
4.1.2	Extending the Perceptual Learner . . . . .	27
4.2	Examples of Integration . . . . .	28
4.3	Experimental Study . . . . .	30
4.3.1	Methods . . . . .	30
4.3.2	Measurements . . . . .	31
4.3.3	Experimental Results . . . . .	32
4.4	Discussion . . . . .	34
<b>5</b>	<b>Learning Perceptual Hierarchies</b>	<b>37</b>

5.1	Learning to Perceive Two-Dimensional Displays . . . . .	37
5.1.1	Problem Definition . . . . .	38
5.1.2	Learning Mechanism . . . . .	41
5.2	Experimental Study in Synthetic Domains . . . . .	45
5.2.1	Methods . . . . .	46
5.2.2	Measurements . . . . .	46
5.2.3	Experimental Results . . . . .	46
5.3	Experimental Study in Three Synthetic Domains . . . . .	47
5.3.1	Methods . . . . .	47
5.3.2	Measurements . . . . .	48
5.3.3	Experimental Results . . . . .	48
5.4	Experimental Study within SimStudent . . . . .	48
5.4.1	Methods . . . . .	48
5.4.2	Measurements . . . . .	49
5.4.3	Experimental Results . . . . .	49
5.5	Discussion . . . . .	49
<b>6</b>	<b>Learning Feature Predicates</b>	<b>51</b>
6.1	Generating Feature Predicates from the Learned Grammar . . . . .	51
6.1.1	Topological Feature Predicates . . . . .	52
6.1.2	Nonterminal Symbol Feature Predicates . . . . .	53
6.1.3	Parse Tree Relation Feature Predicates . . . . .	53
6.2	Experimental Study on Automatically Generated Feature Predicates . . . . .	54
6.2.1	Methods . . . . .	54
6.2.2	Measurements . . . . .	54
6.2.3	Experimental Results . . . . .	55
6.3	Experimental Study on Transferability to Harder Problems . . . . .	56
6.3.1	Methods . . . . .	56
6.3.2	Measurements . . . . .	57
6.3.3	Experimental Results . . . . .	57
6.4	Discussion . . . . .	60
<b>7</b>	<b>Integrating Representation Learning with External World Knowledge</b>	<b>61</b>
7.1	English Article System . . . . .	61
7.2	Integrating Representation Learning with External World Knowledge . . . . .	62
7.3	SimStudent with Probabilistic-Based Conflict Resolution . . . . .	63
7.4	Experimental Study . . . . .	64
7.4.1	Methods . . . . .	64
7.4.2	Experimental Results . . . . .	65
7.5	Discussion . . . . .	66
<b>8</b>	<b>Using SimStudent to Discover Better Learner Models</b>	<b>67</b>
8.1	Methods . . . . .	68
8.2	Dataset . . . . .	69

8.3	Measurements . . . . .	70
8.4	Experimental Results . . . . .	70
8.5	FBI Analysis and LFA on Fraction Addition . . . . .	72
8.6	Detailed Implications for Instructional Decision in Algebra . . . . .	74
8.7	Discussion . . . . .	75
<b>9</b>	<b>Conclusion</b>	<b>77</b>
	<b>Bibliography</b>	<b>81</b>



# List of Figures

2.1	A simple interface to tutor SimStudent in equation solving. . . . .	6
2.2	The interface that shows how SimStudent traces each demonstrated step and learns production rules. . . . .	7
2.3	The perceptual hierarchy associated with the interface in equation solving. . . . .	8
2.4	A production rule for divide. . . . .	9
2.5	A diagram about how SimStudent reacts with the environment. Solid lines show the information that flows through components during execution. Information about the training examples is not presented. . . . .	10
3.1	Correct and incorrect parse trees for $-3x$ . . . . .	15
3.2	Candidate parse trees constructed during learning in algebra. . . . .	17
3.3	Example context free grammar constructed during learning in algebra. . . . .	18
3.4	Learning curves in the last task for four learners in curriculum (a) from task one to task two (b) from task two to task three (c) from task one and two to task three. Both prior knowledge transfer and the feature focus strategy produce faster learning. . . . .	22
4.1	Original and extended production rules for divide in a readable format. Grammar learning allows extraction of information in where-part of the production rule and eliminated the need for domain-specific function authoring (get-coefficient) for use in the how-part. . . . .	26
4.2	A diagram about how the extended SimStudent makes use of the representation acquired by the representation learner, and reacts with the environment. Solid lines show the information that flows through components during execution. Red dashed lines illustrate how the acquired representation is used by the learning components during learning. Information about the training examples is not presented. . . . .	27
4.3	The extended perceptual hierarchy associated with the interface in equation solving. . . . .	28
4.4	Example parse trees learned by the representation learner in three domains, a) fraction addition, b) equation solving, c) stoichiometry. . . . .	29
4.5	Number of domain-specific and domain-general operator functions used in acquired production rules, a) fraction addition, b) equation solving, c) stoichiometry, d) across three domains. . . . .	33

4.6	Number of lines of Java code developed for operator functions used in acquired production rules. . . . .	34
4.7	Learning curves of three SimStudents in three domains, a) fraction addition, b) equation solving, c) stoichiometry. . . . .	35
5.1	An example layout of the interface where SimStudent is being tutored in an equation solving domain. . . . .	41
5.2	Recall scores in a) randomly-generated domains, and three synthetic domains, b) fraction addition, c) equation solving, d) stoichiometry. . . . .	47
5.3	Learning curves of three SimStudents in three domains, a) fraction addition, b) equation solving, c) stoichiometry. . . . .	49
6.1	Learning curves of three SimStudents in equation solving measured by, a) first attempt accuracy, b) all attempt accuracy. . . . .	55
6.2	Learning curves of SimStudents in equation solving measured by a) all test problems, b) hard problems (category 4) only, using all attempt accuracy. . . . .	58
7.1	The parse tree of “ <i>Clocks measure ___ time.</i> ” generated by the Stanford parser. . .	63
7.2	Learning curves of SimStudents in article selection. . . . .	65
8.1	Different parse trees for $-3x$ and $-x$ . . . . .	71
8.2	Error rates for real students and predicted error rates from two learner models. . .	73

# List of Tables

3.1	Probabilistic context-free grammar for coefficients in algebraic equations. . . . .	14
3.2	Method summary . . . . .	21
4.1	Number of training problems and testing problems presented to SimStudent. . . . .	30
4.2	12 curricula of different orders for each domain. . . . .	32
5.1	Part of the two-dimensional probabilistic context free grammar for the equation solving interface . . . . .	40
7.1	Grammar rules in selecting appropriate articles. . . . .	62
8.1	Number of KCs in SimStudent models and Human-Generated Models. . . . .	71
8.2	AIC on SimStudent-Generated models and Human-Generated Models. . . . .	71
8.3	CV RMSE on SimStudent-Generated models and Human-Generated Models. . . . .	72





# Chapter 1

## Introduction

One of the fundamental goals of artificial intelligence is to understand and develop intelligent agents that simulate human-like intelligence. A considerable amount of effort (e.g., [Laird et al., 1987](#), [Anderson, 1993](#), [Langley and Choi, 2006](#)) has been put toward this challenging task. Further, education in the 21<sup>st</sup> century will be increasingly about helping students not just learn content but become better learners. Thus, we have a second goal of improving our understanding of how humans acquire knowledge and how students vary in their abilities to learn.

### 1.1 Motivation

To contribute to both goals, there have been recent efforts (e.g., [Anzai and Simon, 1979](#), [Neves, 1985](#), [Vanlehn, Ohlsson, and Nason, 1994](#), [Matsuda, Lee, Cohen, and Koedinger, 2009](#)) in developing intelligent agents that model human learning of math, science, or a second language. Although such agents produce intelligent behavior with less human knowledge engineering than before, there remains a non-trivial element of knowledge engineering in the encoding of the prior domain knowledge given to the simulated student at the start of the learning process. For example, to build an algebra learning agent, the agent developer needs to provide prior knowledge by coding functions that describe how to extract a coefficient or how to add two algebraic terms. Additionally, the manually-constructed domain-specific prior knowledge may not be reusable in other domains, and thus the need for such prior knowledge hurts the generality of the constructed learning agent.

Moreover, manual encoding of prior knowledge can be time-consuming and may not correspond with human learning. Since real students entering a course do not usually have substantial domain-specific or domain-relevant prior knowledge, it is not realistic in a model of human learning to assume this knowledge is given rather than learned. For example, for students learning about algebra, we cannot assume that they all know beforehand what a coefficient is, or what the difference between a variable term and a constant term is. An intelligent system that models automatic knowledge acquisition with a small amount of prior knowledge could be helpful

both in reducing the effort in knowledge engineering intelligent systems and in advancing the cognitive science of human learning.

## 1.2 Proposed Approach

The goal of this thesis is to build an intelligent agent that is able to learn complex problem solving skills in Science, Technology, Engineering, and Mathematics (STEM) domains from simple demonstrations and feedback. As mentioned above, previous work in this area (e.g., [Anzai and Simon, 1979](#), [Neves, 1985](#), [Vanlehn, Ohlsson, and Nason, 1994](#), [Matsuda, Lee, Cohen, and Koedinger, 2009](#)) has developed intelligent agents that acquire complex skills, but the agents' performance often relies heavily on the pre-programmed representations and features specific to the domain. To address this problem, we propose to develop an unsupervised feature/representation learner, and integrate it into a supervised skill-learning agent to improve learning effectiveness of the agent. Hence, unlike normal feature discovery methods that optimize classification or prediction accuracy, the objective of complex skill learning tasks is to jointly optimize learning of perceptual chunks, action chunks or functions, and search control.

There are three main streams of feature construction algorithms. The first category of work has an unsupervised component that learns key features to identify patterns in data (e.g., images), and then a supervised learning process that makes use of these features to optimize some objective function. An example of such work is deep belief networks [[Hinton, 2007](#)]. Other work takes a joint learning strategy to build latent variable discriminative models such as supervised LDA [[Blei and McAuliffe, 2007](#)]. A third branch in feature construction is supervised feature induction, which searches for new features to optimize an objective function.

Our feature/representation learner falls into the first category, where it focuses on building a generative model,  $\mathcal{G}$ , that best captures the distribution among observations (e.g., algebraic expressions),  $\mathcal{R}$ , which approximates maximum likelihood estimate (MLE) of  $p(\mathcal{R}|\mathcal{G})$ . Although not trained by directly optimizing the learning effectiveness of the intelligent agent, the philosophy behind this strategy is that if we could correctly model these observations (e.g., the right parse structures for the expressions), the acquired representation should contain useful features that aid the skill learning process, and yield faster learning.

The idea of our representation learner comes from previous work in cognitive science [[Chi et al., 1981](#), [Chase and Simon, 1973](#)], which showed that one of the key factors that differentiates experts and novices in a field is their different prior knowledge of world state representation. Experts view the world in terms of deep functional features (e.g., coefficient and constant terms in algebra), while novices only view in terms of shallow perceptual features (e.g., integer in an expression). Representation learning is a major component of human expertise acquisition, but has not received much attention in AI until recently. Learning deep features changes the representation on which future learning is based and, by doing so, improves future learning. However, how these deep features are acquired is not clear. Therefore, we have recently developed a learning

algorithm that acquires representations of the problems in terms of deep features automatically with only domain-independent knowledge (e.g., what is an integer) as input [Li et al., 2010]. We evaluated the effectiveness of the algorithm in learning deep features, but not its impact on future skill learner.

In order to evaluate how the representation learner could affect future learning of an intelligent agent, we further integrated this representation learning algorithm into *SimStudent* [Matsuda et al., 2009], an agent that learns problem-solving skills by examples and by feedback on performance, and demonstrated the extended SimStudent across multiple domains (e.g., algebra, fraction addition, stoichiometry, article selection). The original SimStudent relies on a hand-engineered representation that encodes an expert representation given as prior knowledge. This limits the generality of the learning agent and its ability to model novice students. Integrating the representation learner into the original SimStudent both enhances the generality of learning through reducing the amount of engineering effort and builds a better model of student learning.

We show that the extended SimStudent with better representation learning performs much better than the original SimStudent when neither of them are given domain-specific knowledge. Furthermore, we also show that even compared to the original SimStudent with the domain-specific knowledge, the extended SimStudent is able to learn nearly as well without being given domain-specific knowledge. In addition, we use the extended SimStudent to automatically discover models of real students, and show that the discovered models fit with human student data better than human-generated models. Further analysis of the discovered model reveals insights that could improve instructional strategies in intelligent tutoring systems.

## 1.3 Main Contribution

To summarize, the main contributions of this work are two-fold. By integrating representation learning into skill learning, 1) we improve the generality of the learning agent by reducing the amount of knowledge engineering effort required in constructing the intelligent agent; 2) we get a better modeling of human learning behavior. Note that rather than duplicating how the human brain works, our focus of this work is to build a system that behaves like human students, and use the proposed system to get better understanding of human knowledge acquisition.

In the following part, we start with a brief review of SimStudent. Next, we present the deep feature representation learning algorithm together with its evaluation results. Then, we describe how to integrate the representation learner into SimStudent, and illustrate the proposed approach with an example in algebra. After that, we present experimental results for both the original SimStudent and the extended SimStudent trained with problem sets used by real students during learning across domains, and show that the extended SimStudent is able to achieve performance comparable to or better than the original SimStudent without requiring domain-specific knowledge as input. Later, we explore the generality of the proposed approach in a more ill-defined domain, article selection in English, where no complex problem solving is needed, but where

complex perceptual knowledge and large amounts of background knowledge are needed. Experimental results show that by incorporating world knowledge in the learning agent, the extended SimStudent can successfully learn how to select the correct article given a reasonable number (i.e., 60) of problems.

In the third part of the thesis, we focus on how the extended SimStudent contributes to learning sciences. First, we present how the extended learning agent can provide insights for better understanding of student learning. To study one of the most important variables that affect learning effectiveness, the order of problems presented to students, we conduct a controlled-simulation study with SimStudent, and carefully inspect what causes such effect by looking at SimStudents learning processes and learning outcomes, which are not easily obtainable from human subjects. Furthermore, we present a method for using the extended SimStudent to automatically discover learner models, and show that the learner model discovered by the extended SimStudent is better than or as good as the human-generated models in predicting human student behavior. We conclude my thesis with a summary of findings along with a discussion of possible future directions.

# Chapter 2

## A Brief Review of SimStudent

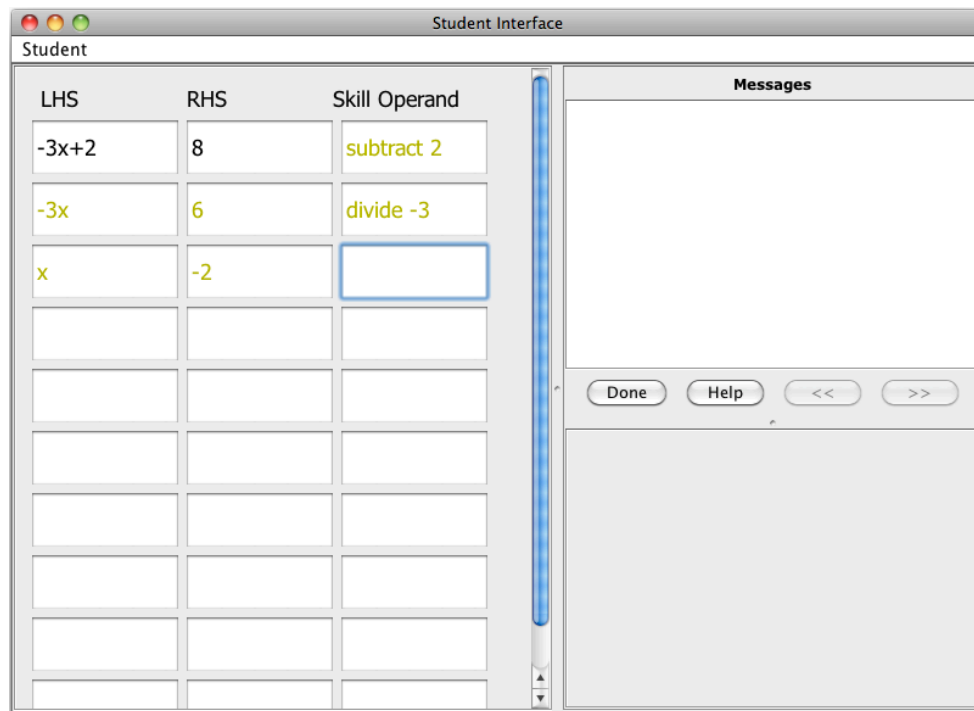
SimStudent is an intelligent agent that inductively learns skills to solve problems from demonstrated solutions and from problem solving experience. It is an extension of programming by demonstration [Lau and Weld, 1998] using a variation of the version space algorithm [Mitchell, 1982], inductive logic programming [Muggleton and de Raedt, 1994], and iterative-deepening depth-first search as underlying learning techniques. Figure 2.1 and 2.2 are screenshots of SimStudent learning to solve algebra equations. Figure 2.1 is an interface used to teach SimStudent equation solving, and Figure 2.2 shows how SimStudent keeps track of the demonstrated steps and acquires skill knowledge based on them. In this thesis, we will use equation solving as an illustrative domain to explain the learning mechanisms. But we would like to point out that the learning algorithms are domain general. In fact, SimStudent has been used and tested across various domains, including multi-column addition, fraction addition, stoichiometry, and so on. In the rest of this subsection, we will briefly review the learning mechanism of SimStudent. For full details, please refer to Matsuda et al. [2009].

### 2.1 Prior Knowledge

Before learning, SimStudent is given a perceptual hierarchy, a set of (ideally simple) *feature predicates* and a set of (ideally simple) *operator functions* as prior knowledge.

A perceptual hierarchy specifies the layout of the elements in the graphical user interface (GUI) that SimStudent is interacting with. The elements in the interface are typically organized in a tree structure. For example, in the algebra domain, interface elements can be of type table, column and cell. In Figure 2.1, the interface contains one table node as the root of the tree. This table node links to three column nodes. For each column node, it has multiple cells as its children, which are also leaves of the tree. Figure 2.3 shows the hierarchically structured elements in the algebra tutor GUI.

Each feature predicate is a Boolean function that describes relations among objects in the domain. For example, (*has-coefficient*  $-3x$ ) means  $-3x$  has a coefficient. SimStudent uses these feature

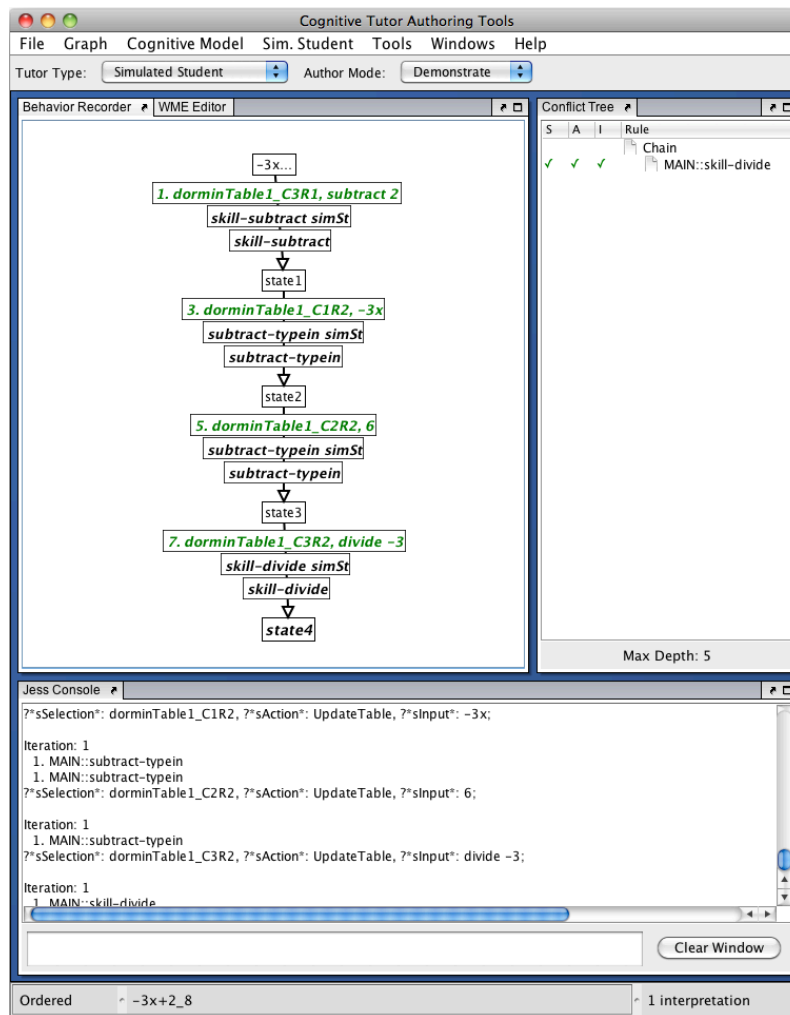


**Figure 2.1:** A simple interface to tutor SimStudent in equation solving.

predicates to understand the state of the given problems.

Operator functions specify basic functions (e.g., add two numbers, get the coefficient) that SimStudent can apply to aspects of the problem representation. Operator functions are divided into two groups, domain-independent operator functions and domain-specific operator functions. Domain-independent operator functions can be used across multiple domains, and tend to be simpler (like standard operations on a programming language). Examples of such operator functions include adding two numbers, (*add 1 2*) or copying a string, (*copy -3x*). These operator functions are not only useful in solving equations, but can also be used in other domains such as multi-column addition and fraction addition. Because these domain-general functions are involved in domains that are acquired before algebra, we can assume that real students know them prior to algebra instruction. Because these domain-general functions can be used in multiple domains, there is a potential engineering benefit in reducing or eliminating a need to write new operator functions when applying SimStudent to a new domain.

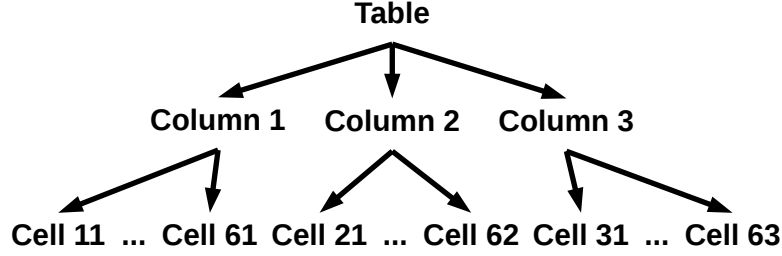
Domain-specific operator functions, on the other hand, are more complicated functions, such as getting the coefficient of a term, (*coefficient -3x*), or adding two terms. Performing such operator functions implies some domain expertise that real students are less likely to have. Domain-specific operator functions tend to require more knowledge engineering or programming effort than domain-independent operator functions. For example, compare the “add” domain-independent operator function with the “add-term” domain-specific operator function. Adding two numbers is one step among the many steps in adding two terms together (i.e., parsing the input terms into sub-terms, applying an addition strategy for each term format, and concatenating



**Figure 2.2:** The interface that shows how SimStudent traces each demonstrated step and learns production rules.

all of the sub-terms together).

Note that operator functions are different from *operators* in traditional planning systems, operator functions have no explicit encoding of preconditions and may not produce correct results when applied in context. Thus, SimStudent is different from traditional planning algorithms, which can engage on speed-up learning. SimStudent engages in knowledge-level learning [Dietterich, 1986], and inductively acquires complex reasoning rules. These rules are represented as *production rules*, which we will explain later.



**Figure 2.3:** The perceptual hierarchy associated with the interface in equation solving.

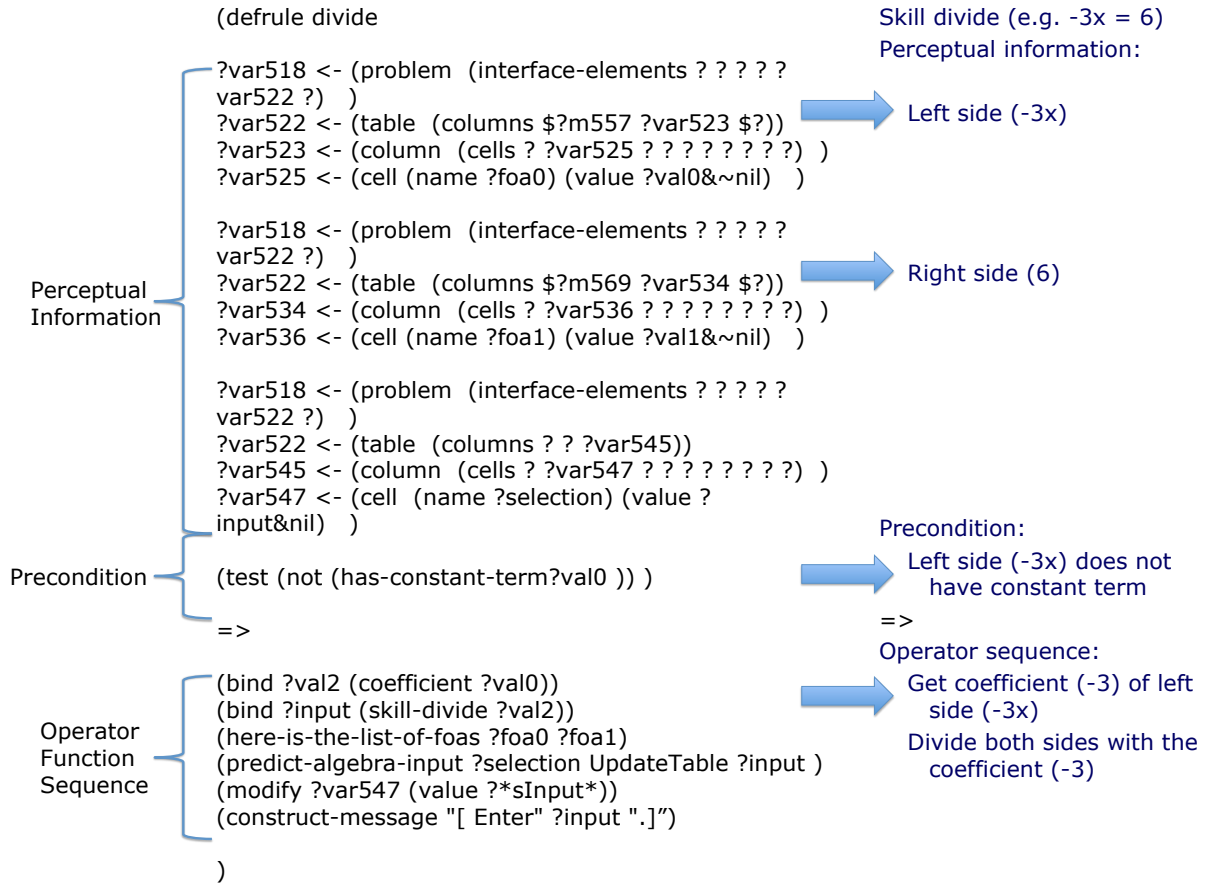
## 2.2 Learning Task

During the learning process, given the current state of the problem (e.g.,  $-3x = 6$ ), SimStudent first tries to find an appropriate production rule that proposes a plan for the next step (e.g., *(bind ?coef (coefficient -3x) (bind ?output (divide ?coef)))*). If it finds one and receives positive feedback, it continues to the next step. If the proposed next step is incorrect, negative feedback is given, and if SimStudent has no other alternatives, a correct next step demonstration is provided. SimStudent will attempt to modify or learn production rules accordingly. Although other feedback mechanisms are also possible, in our case, the feedback is given by automatic cognitive tutors (e.g., [Koedinger and Corbett, 2006](#)), which simulate the tutors used to teach real students.

For each demonstrated step, the tutor specifies 1) *perceptual information* (e.g.,  $-3x$  and  $6$  for  $-3x = 6$ ) from a graphical user interface (GUI) showing where to find information to perform the next step, 2) *a skill label* (e.g., divide) corresponding to the type of skill applied, 3) *a next step* (e.g., *(divide -3)* for problem  $-3x = 6$ ). This simulates the limited information available to real students. Taken together, the three pieces of information form an *example action record* indexed by the skill label,  $R = \langle \text{label}, \langle \text{percepts}, \text{step} \rangle \rangle$ . In the algebra example, an example action record is  $R = \langle \text{divide}, \langle (-3x, 6), (\text{divide } -3) \rangle \rangle$ . For each incorrect next step proposed by SimStudent, an example action record is also generated as a negative example. During learning, SimStudent typically acquires one production rule for each skill label,  $l$ , based on the set of associated (both positive and negative) example action records gathered up to the current step,  $\mathcal{R}_l = (R_1, R_2, \dots, R_n)$  (where  $R_i.\text{label} = l$ ).

In summary, since we would like to model how real students are tutored, the learning task presented to SimStudent is challenging. First, the total number of world states is large. In equation solving, for instance, there are infinite variety of algebraic expressions that can be entered and there are many possible alternative solution strategies. Second, the operator functions given as prior knowledge do not encode any preconditions (neither for applicability nor for search control) or postconditions. Last, the semantics of a demonstrated step is only partially observable. It usually takes more than one operator function to move from one observed state to the next observed state. Correct intermediate outputs of operator functions are unobservable to SimStudent. Taken together, the learning task SimStudent is facing is learning skill knowledge within infinite world



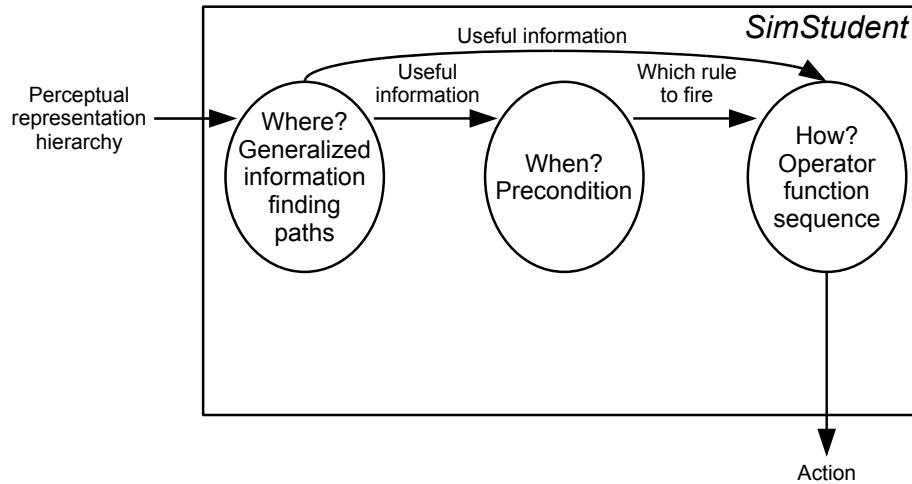


**Figure 2.4:** A production rule for divide.

states given incomplete operator function descriptions and partially observable states.

## 2.3 Production Rules

The output of the learning agent is represented as production rules [Laird et al., 1987, Anderson, 1993]. The left side of Figure 2.4 shows an example of a production rule learned by SimStudent with a simple English description shown on the right. A production rule indicates “where” to look for information in the interface (perceptual information), “how” to change the problem state (an operator function sequence), and “when” to apply a rule (a set of features indicting the circumstances under which performing the how-part will be useful). For example, the rule to “divide both sides of  $-3x=6$  by  $-3$ ” shown in Figure 2.4 can be read as “given a left-hand side (i.e.,  $-3x$ ) and a right-hand side (6) of an equation, when the left-hand side does not have a constant term, then get the coefficient of the term on the left-hand side and write “divide” followed by the coefficient.” The perceptual information part represents paths to identify useful information from the GUI. The precondition (just before “ $\Rightarrow$ ” in Figure 2.4) includes a set of feature tests



**Figure 2.5:** A diagram about how SimStudent reacts with the environment. Solid lines show the information that flows through components during execution. Information about the training examples is not presented.

representing desired conditions in which to apply the production rule. The last part (after “ $\Rightarrow$ ” in Figure 2.4) is the operator function sequence which computes what to output in the GUI.

During execution, as shown in Figure 2.5, SimStudent receives perceptual information from the environment as a hierarchy. The where part finds the useful information from this hierarchy. Next, the when part uses the useful information to decide which production rule to fire. The selected production rule will generate an action that SimStudent is going to execute in the world decided by the how part of the production rule.

## 2.4 Learning Mechanisms

With all the challenges presented, we have developed three learning mechanisms in SimStudent to acquire the three parts of the production rules as shown in Figure 2.5 [Matsuda et al., 2009], where each learning component models one aspect of problem-solving skill acquisition. The first component is a perceptual learner that learns the where-part of the production rule by finding paths to identify useful information in the GUI. Recall that the elements in the interface are typically organized in a tree structure. This tree structure is referred as a *perceptual hierarchy*. For example, the table node has columns as children, and each column has multiple cells as children. The percepts specified in the above production rule are cells associated with the sides of the algebra equation, which are *Cell 11* and *Cell 21* in this case. Hence, the perceptual learner’s task is to find the right paths in the tree to reach the specified cell nodes. There are two ways to reach a percept node in the interface: 1) by the exact path to its exact position in the tree, or 2) by a generalized path to a set of GUI elements that may have a specific relationship with the GUI element where the next step is entered (e.g., cells above next step). A generalized path has one

or more levels in the tree that are bound to more than one node. For example, a cell in the second column and the third row, *Cell 23*, can be generalized to any cell in the second column, *Cell 2?*, or any cell in the table, *Cell ??*. In the example shown in Figure 2.4, the production rule has an over-specific where-part that produces a next step only when the sides of the current step are in row two. The learner searches for the least general path in the version space formed by the set of paths to training examples [Mitchell, 1982]. This process is done by a brute-force depth-first search. For example, if only given the example  $-3x=6$  in row two, the production rule learned as shown in Figure 2.4 has an over-specific where-part. If given more examples in other rows (e.g.,  $4x=12$  in row three), the where-part will be generalized to any row in the table.

The second part of the learning mechanism is a feature test learner that learns the when-part of the production rule by acquiring the precondition of the production rule using the given feature predicates. The acquired preconditions should contain information about both applicability (e.g., getting a coefficient is not applicable to the term  $3x+5$ ) and search control (e.g., it is not preferred to add 5 to both sides for problem  $-3x = 6$ ). The feature test learner utilizes FOIL [Quinlan, 1990], an inductive logic programming system that learns Horn clauses from both positive and negative examples expressed as relations. FOIL is used to acquire a set of feature tests that describe the desired situation in which to fire the production rule. For each rule, the feature test learner creates a new predicate that corresponds to the precondition of the rule, and sets it as the target relation for FOIL to learn. The arguments of the new predicate are associated with the percepts. Each training action record serves as either a positive or a negative example for FOIL based on the feedback provided by the tutor. For example, (*precondition-divide* ?percept<sub>1</sub> ?percept<sub>2</sub>) is the precondition predicate associated with the production rule named “divide”. (*precondition-divide*  $-3x\ 6$ ) is a positive example for it. The feature test learner computes the truthfulness of all predicates bound with all possible permutations of percept values, and sends it as input to FOIL. Given these inputs, FOIL will acquire a set of clauses formed by feature predicates describing the precondition predicate.

The last component is an operator function sequence learner that acquires the how-part of the production rule. For each positive example action record,  $R_i$ , the learner takes the percepts,  $R_i.percepts$ , as the initial state, and sets the step,  $R_i.step$ , as the goal state. We say an operator function sequence *explains* a percepts-step pair,  $\langle R_i.percepts, R_i.step \rangle$ , if the system takes  $R_i.percepts$  as an initial state and yields  $step_i$  after applying the operator functions. For example, if SimStudent first receives a percepts-step pair,  $\langle (2x, 2), (divide\ 2) \rangle$ , both the operator function sequence that directly divides both sides with the right-hand side (i.e., (*bind* ?output (*divide* 2))), and the sequence that first gets the coefficient, and then divides both sides with the coefficient (i.e., (*bind* ?coef (*coefficient* 2x ?coef)) (*bind* ?output (*divide* ?coef))) are possible explanations for the given pair. Since we have multiple example action records for each skill, it is not sufficient to find one operator function sequence for each example action record. Instead, the learner attempts to find a shortest operator function sequence that explains all of the  $\langle percepts, step \rangle$  pairs using iterative-deepening depth-first search within some depth-limit. As in the above example, since (*bind* ?output (*divide* 2)) is shorter than (i.e., (*bind* ?coef (*coefficient* 2x ?coef)) (*bind* ?output (*divide* ?coef))), SimStudent will learn this operator function sequence as the how-part. Later, it meets another example,  $-3x=6$ , and receives another percepts-step pair,  $\langle (-3x, 6), (divide\ -3) \rangle$ . The operator function sequence that divides both sides with the right-hand side is

not a possible explanation any more. Hence, SimStudent modifies the how-part to be the longer operator function sequence *(bind ?coef (coefficient ?rhs)) (bind ?output (divide ?coef))*.

Last, although we said that SimStudent tries to learn one rule for each label, when a new training action record is added, SimStudent might fail to learn a single rule for all example action records when the perceptual information learner cannot find one path that covers all demonstrated steps, or the operator sequence learner cannot find one operator function sequence that explains all records. In that case, SimStudent learns a separate rule just for the last example action record. This breaking a single production rule into a pair of disjuncts effectively splits the example action records into two clusters. Later, for each new example action record, SimStudent tries to acquire a rule for each of the example clusters plus the new example action record. If the new record cannot be added to any of the existing clusters, SimStudent creates another new cluster. As we will see in Chapter 8, this clustering behavior can be used to discover models of student learning.

## Chapter 3

# Deep Feature Representation Learning

Having reviewed SimStudent’s production rule learning mechanisms, we move to a discussion of deep feature knowledge acquisition as representation learning. As mentioned above, representation learning is important both for human knowledge acquisition, and in achieving effective machine learning. Previous studies [Chi et al., 1981, Chase and Simon, 1973] show that one of the key factors that differentiates an expert from a novice is the knowledge of deep features. For instance, in the algebra domain, deep features such as coefficient and constant are usually hard for students to acquire. In addition, the performance of many existing learning algorithms is sensitive to representations. Automatic acquisition of good representation knowledge such as deep features is important in achieving effective learning. We carefully examine the nature of representation learning in algebra equation solving, and discover that it could be modeled as an unsupervised grammar induction problem given observational data (e.g., expressions in algebra). Expressions can be formulated as a context free grammar and deep features are modeled as non-terminal symbols in particular positions in a grammar rule. Table 3.1 illustrates a portion of a grammar for algebra expressions and the modeling of the deep feature “coefficient” as a non-terminal symbol in one of the grammar rules, as indicated by the square brackets (i.e., *[SignedNumber]*).

Viewing representation learning tasks as grammar induction provides a general explanation of how experts acquire perceptual chunks [Chase and Simon, 1973, Koedinger and Anderson, 1990] and explanations for specific novice errors. In this account, some novice errors are the result of acquiring the wrong grammar for the task domain. Let us use the  $-3x$  example again. The correct grammar shown in Table 3.1 produces the correct parse tree shown on the left in Figure 3.1. A novice, however, may acquire different grammar rules (e.g., because of plausible lack of experience with negative numbers) and these result in the incorrect parse tree shown on the right of Figure 3.1. Instead of grouping  $-$  and  $3$  together, this grammar groups  $3$  and  $x$  first, and thus mistakenly considers  $3$  as the coefficient. In fact, a common strategic error students make in a problem like  $-3x=12$  is for the student to divide both sides by  $3$  rather than  $-3$  [Li et al., 2011a]. Based on these observations, we built a representation learner by extending an existing probabilistic context free grammar (pCFG) learner [Li et al., 2009] to support feature learning and transfer learning. The representation learner is domain general. It currently supports domains

**Table 3.1:** Probabilistic context-free grammar for coefficients in algebraic equations.

---

Terminal symbols:	$-$ , $x$ , 0, 1, 2, 3, 4, 5, 6, 7, 8, 9;
Non-terminal symbols:	<i>Expression</i> , <i>SignedNumber</i> , <i>Variable</i> , <i>MinusSign</i> , <i>Number</i> ;
	<i>Expression</i> $\rightarrow$ 0.33, [ <i>SignedNumber</i> ] <i>Variable</i>
	<i>Expression</i> $\rightarrow$ 0.67, <i>SignedNumber</i>
	<i>Variable</i> $\rightarrow$ 1.0, $x$
	<i>SignedNumber</i> $\rightarrow$ 0.5, <i>MinusSign</i> <i>Number</i>
	<i>SignedNumber</i> $\rightarrow$ 0.5, <i>Number</i>
	<i>Number</i> $\rightarrow$ 0.091, <i>Number</i> <i>Number</i>
	<i>Number</i> $\rightarrow$ 0.091, 0
	<i>Number</i> $\rightarrow$ 0.091, 1
	<i>Number</i> $\rightarrow$ 0.091, 2
	<i>Number</i> $\rightarrow$ 0.091, 3
	<i>Number</i> $\rightarrow$ 0.091, 4
	<i>Number</i> $\rightarrow$ 0.091, 5
	<i>Number</i> $\rightarrow$ 0.091, 6
	<i>Number</i> $\rightarrow$ 0.091, 7
	<i>Number</i> $\rightarrow$ 0.091, 8
	<i>Number</i> $\rightarrow$ 0.091, 9
	<i>MinusSign</i> $\rightarrow$ 1.0, $-$

---

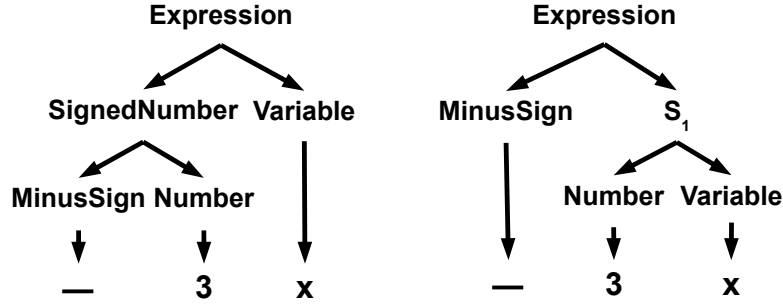
where student input can be represented as a string of tokens, and can be modeled with a context-free grammar (e.g., algebra, chemistry, natural language processing).

## 3.1 Representation Learning as Grammar Induction

Before introducing the representation learning algorithm, we first briefly review the pCFG learner [Li et al., 2009] it is based on. The pCFG learner is a variant of the inside-outside algorithm [Lari and Young, 1990].

### 3.1.1 A Brief Review of a pCFG Learner

The input to the pCFG learner is a set of observation sequences,  $\mathcal{O}$ . Each sequence is a string of tokens directly from user input (e.g.,  $-3x$ ). The output is a pCFG that can generate all input observation sequences with high probabilities. The system consists of two parts, a greedy structure hypothesizer (GSH), which creates non-terminal symbols and associated grammar rules as needed, to cover all the training examples, and a Viterbi training step, which iteratively refines the probabilities of the grammar rules.



**Figure 3.1:** Correct and incorrect parse trees for  $-3x$ .

### Greedy Structure Hypothesizer (GSH)

GSH creates a context-free grammar in a bottom-up fashion. Pseudo code for the GSH algorithm is shown in algorithm 3.1. It starts by initializing the rule set  $S$  to rules associated with terminal characters (e.g.,  $-$ ,  $3$  and  $x$  in  $-3x$ ) in the observation sequences,  $\mathcal{O}$ . Next the algorithm (line 4) detects whether there are possible recursive structures embedded in the observation sequence by looking for repeated symbols. If so, the algorithm learns a recursive rule for them (e.g.,  $Number \rightarrow 0.091, Number\ Number$ ). If the algorithm fails to find recursive structures, it starts to search for the character pair that appears in the plans most frequently (line 6), and constructs a grammar rule for the character pair. To build a non-recursive rule, the algorithm will introduce a new symbol and set it as the head of the new rule. After getting the new rule, the system updates the current observation set  $\mathcal{O}$  with this rule by replacing the character pairs in the observations with the head of the rule (line 9).

After learning all the grammar rules, the structure learning algorithm assigns initial probabilities to these rules. If there are  $k$  grammar rules with the same head symbol, then each of them are assigned the probability  $\frac{1}{k}$ . To break ties among grammar rules with the same head, GSH adds a small random number to each probability and normalizes the values again. This output of GSH is a redundant set of grammar rules, which is sent to the Viterbi training phase.

### Refining Schema Probabilities – Viterbi Training Phase

The Viterbi training algorithm tunes the probabilities associated with the initial set of rules generated by the GSH phase. It considers the parse trees  $\mathcal{T}$  associated with the observation sequence as hidden variables, and carries out an iterative refinement process to update the parse trees as well as the grammar rules. Each iteration involves two steps.

In the first step, the algorithm computes the most probable parse tree for each observation using the current rules. Any subtree of a most probable parse tree is also a most probable parse subtree. Therefore, for each observation sequence, the algorithm builds the most probable parse tree in a

---

**Algorithm 3.1:** *GSH* constructs an initial set of grammar rules,  $S$ , from observation sequences,  $O$ .

---

**Input:** Observation Sequence Set  $O$ .

```

1  $S :=$  terminal symbol grammar rules;
2 while not-all-sequences-are-parsable( $O$ ,  $S$ ) do
3   if has-recursive-rule( $O$ ) then
4      $s :=$  generate-recursive-rule( $O$ );
5   else
6      $s :=$  generate-most-frequent-rule( $O$ );
7   end
8    $S := S + s$ ;
9    $O :=$  update-plan-set-with-rule( $O$ ,  $S$ );
10 end
11  $S =$  initialize-probabilities( $S$ ); return  $S$ 

```

---

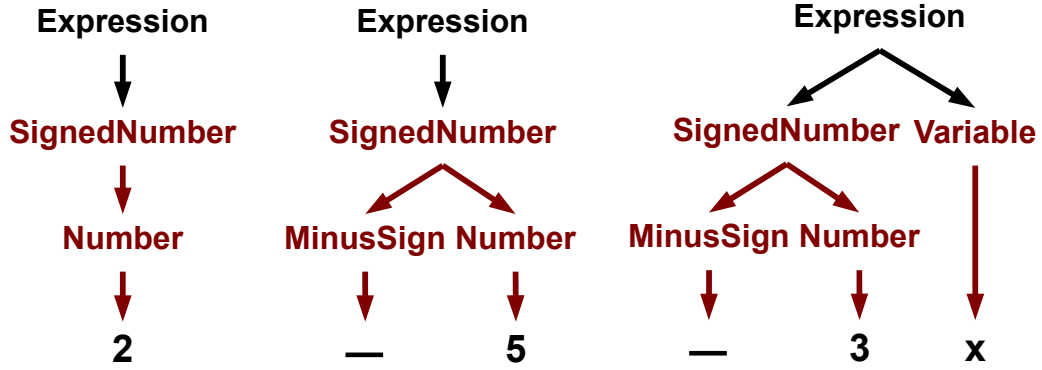
bottom-up fashion until reaching the start symbol. After getting the parse trees for all observations, the algorithm moves on to the second step. In this step, the algorithm updates the selection probabilities associated with the grammar rules. For a grammar rule with head  $h$ , the new probability of being chosen is simply the total number of times that rule appears in the Viterbi parse trees divided by the total number of times  $h$  appears in the parse trees. (This learning procedure is a fast approximation of expectation-maximization [Dempster et al., 1977], which approximates the posterior distribution of trees given parameters by the single MAP hypothesis.) After finishing the second step, the algorithm starts a new iteration until convergence. The output of the algorithm is a set of probabilistic grammar rules.

For example, as shown in Figure 3.2, the pCFG learner is given three observation sequences, 2,  $-5$ , and  $-3x$ , and it knows that 2, 3, and 5 are numbers. In the first step, the structure hypothesizer finds that the  $\langle \text{MinusSign}, \text{Number} \rangle$  pair appears more often than the  $\langle \text{Number}, \text{Variable} \rangle$  pair. It creates a rule that reduces an automatically-generated non-terminal symbol, *SignedNumber*, into *MinusSign* and *Number*, and replaces  $-5$  and  $-3$  in the observation sequences with the non-terminal symbol *SignedNumber*. Then, using the updated sequences, the GSH continues to find frequently appeared pairs and creates non-terminal symbols as well as grammar rules as needed. This procedure continues until every training example has at least one parse tree, as shown in Figure 3.2. The hypothesized grammar rules as presented in Figure 3.3 are then sent to the Viterbi training step, where the probabilities associated with grammar rules are refined, and redundant grammar rules are removed.

### 3.1.2 Feature Learning

Having reviewed Li et al.’s [2009] pCFG learning algorithm, we are ready to describe how it is extended to support representation learning without SimStudent. The input of the system is a set of pairs such as  $\langle -3x, -3 \rangle$ , where the first element is the input to a feature extraction mechanism



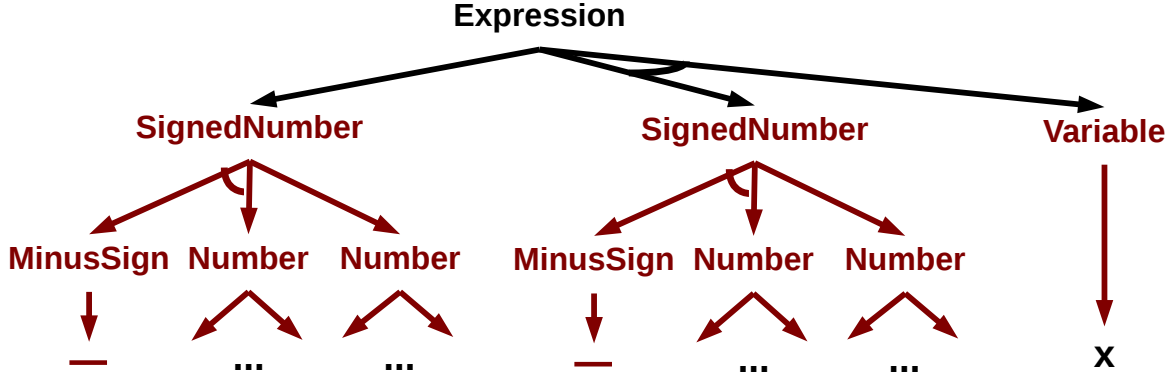


**Figure 3.2:** Candidate parse trees constructed during learning in algebra.

(e.g., coefficient), and the second is the extraction output (e.g., -3 is the coefficient of  $-3x$ ). The output is a pCFG with a non-terminal symbol in one of the rules set as the target feature (as shown by *SignedNumber* in Table 3.1). The learning process contains two steps. The system first acquires the grammar using Li et al.’s [2009] algorithm. After that, the representation learner tries to identify a non-terminal symbol in one of the rules as the target feature. To do this, the system builds parse trees for all of the observation sequences, and picks the non-terminal symbol that corresponds to the most training records as the deep feature. To produce this output, the representation learner uses the pCFG learner to produce a grammar, and then searches for non-terminal symbols that correspond to the extraction output (e.g., the -3 in  $-3x$ ). The process is done in three steps.

The system first builds the parse trees for all of the observation sequences based on the acquired rules. For instance, in algebra, suppose we have acquired the pCFG shown in Table 3.1. The associated parse tree of  $-3x$  is shown at the left side of Figure 3.1. Next, for each sequence, the learner traverses the parse tree to identify the non-terminal symbol associated with the target feature extraction output, and the rule to which the non-terminal symbol belongs. In the case of our example, the non-terminal symbol is *SignedNumber*, the associated feature extraction output is -3, and the rule is  $Expression \rightarrow 1.0, SignedNumber Variable$ . For some of the sequences, the feature extraction output may not be generated by a single non-terminal symbol, which happens when the acquired pCFG does not have the right structure. For example, the parse tree shown in the right side of Figure 3.1 is an incorrect parse of  $-3x$ , and there is no non-terminal symbol associated with -3. In this case, no non-terminal symbol is associated with the target feature for the current sequence, and this sequence will not be counted towards the identification of the target feature. Last, the system records the frequency of each symbol rule pair, and picks the pair that matches the most training records as the learned feature. For instance, if most of the input records match with *SignedNumber* in  $Expression \rightarrow 1.0, SignedNumber Variable$ , this symbol-rule pair will be considered as the target feature pattern.

After learning the feature, when a new problem comes, the system will first build the parse tree



**Figure 3.3:** Example context free grammar constructed during learning in algebra.

of the new problem based on the acquired grammar. Then, the system recognizes the subsequence associated with the feature symbol from the parse tree, and returns it as the target feature extraction output (e.g., -5 in  $-5x$ ). This model presented so far learns to extract deep features in a mostly unsupervised way without any goals or context from SimStudent problem solving. Later, we describe how to extend its ability by integrating it into SimStudent’s supervised skill learning process.

### 3.1.3 Transfer Learning

In order to achieve effective learning, we further extend the representation learner to support transfer learning within the same domain and across domains. Different grammars sometimes share grammar rules for some non-terminal symbols. For example, both the grammar of equation solving and the grammar of integer arithmetic problems should contain the sub-grammar of signed number. We extend the representation learning algorithm to transfer solutions to common sub-grammars from one task to another. Note that the tasks can be either from the same domain (e.g. learning what is an integer, and learning what is a coefficient), or from different domains (e.g. learning what is an integer, and learning what is a chemical formula). We consider two learning protocols: one in which the tutor provides hints to a shared grammar by highlighting subsequences that should be associated with one non-terminal symbol; and one in which the shared grammar is present, but no hints are provided. For transfer learning with sub-grammar hints, we apply what we will call a *feature focus mechanism* to the acquisition process. For transfer learning without sub-grammar hints, we extend the system to make use of grammar rule application frequencies from previous tasks to guide future learning, as explained below.

## Explicitly Labeled Common Sub-grammars

We first consider the situation where SimStudent’s tutor provides a hint toward a shared sub-grammar (the deep feature). In the original learning algorithm, during the process of grammar induction, the learner acquires some grammar that generates the observation sequences, without differentiating potential feature subsequences (e.g. coefficients or constant terms) from other subsequences in the training examples. It is possible that two grammars can generate the same set of observation sequences, but only one grammar has the appropriate feature symbol embedded in it. We cannot be sure that the original learner will acquire the right one.

However, it may be reasonable to assume that a tutor explicitly highlights example subsequences as targeted features as in a teacher giving examples of coefficient by indicating that  $-3$  is the coefficient of  $-3x$  and  $-4$  is the coefficient of  $-4x$ . With this assumption, the representation learner can focus on creating non-terminal symbols for such feature subsequences. We develop this *feature focus mechanism* as follows. First, we call one copy of the original learner to acquire the subgrammar for the highlighted subsequences (i.e., the deep feature). That is, the representation learner extracts all the feature subsequences from training sequences, and then learns a sub-grammar for it. The representation learner then replaces the feature subsequence with a special *semantic terminal symbol*, and invokes the original learner on this problem. Since this semantic terminal symbol is viewed as a terminal character in this phase of learning, it must be properly embedded in the observation sequence. Finally, the two grammars are combined, and the semantic terminal is relabeled as a *non-terminal symbol* and associated with the start symbol for the grammar for the feature. Note that if we consider the acquisition of the deep feature grammar is the first subtask, and learning the whole grammar is a second task, we can view this process as a subtask transfer within the same task.

## Learning and Transfer of Common Sub-grammars without Hints

Aiding transfer learning by providing hints for common sub-grammars requires extra work for the tutor. A more powerful learning strategy should be able to transfer knowledge without adding more work for the tutor. Therefore, we consider a second learning protocol, where the shared grammar is present, but no hint to it is provided. An appropriate way of transferring previously acquired knowledge to later learning could improve the speed and accuracy of that later learning. The intuition here is that the perceptual chunks or grammar acquired with whole-number experience will aid grammar acquisition of negative numbers that, in turn, will aid algebra grammar acquisition. Our solution involves transferring the acquired grammar, including the application frequency of each grammar rule, from previous tasks to future tasks.

More specifically, during the acquisition of the grammar in previous tasks, the learner records the acquired grammar and the number of times each grammar rule appeared in a parse tree. When facing with a new task, the learning algorithm first uses the existing grammar from previous tasks to build the smallest number of most probable parse trees for the new records. This process is done in a top-down fashion. For each sequence/subsequence, the algorithm first tries to see whether the given sequence/subsequence can be reduced to a single most probable parse tree. If

it succeeds, the algorithm returns. If it fails, the algorithm separates the sequence/subsequence into two subsequences, and recursively calls itself. After building the smallest number of most probable parse trees for the training subsequences, the system switches to the original GSH and acquires new rules based on the partially parsed sequences.

For example, if the representation learner has acquired what is a signed number (e.g.,  $-3$ ) in a previous task (as shown in red in Figure 3.3), when facing with a new task of learning what is an expression (e.g.,  $-3x$ ), the learner first tries to build a parse tree for the whole term (e.g.,  $-3x$ ). But it fails because the grammar for signed number can only build the parse trees for some subsequence (e.g.,  $-3$  in  $-3x$ ). Nevertheless, the grammar learner does get some partially parsed sequences (e.g., the partial reduced sequence for  $-3x$  is *SignedNumber*  $x$  as shown in red in Figure 3.2), and calls the original grammar learner on these partially parsed sequences.

During the Viterbi training phase, the learning algorithm estimates rule frequency using a Dirichlet distribution based on prior tasks; that is, it adds the applied rule frequency associated with the training problems of the current task to the recorded frequency from previous tasks. Note that it is possible that after acquiring new rules with new examples, in the Viterbi training phase, the parse trees for the training examples in the previous tasks have changed, and the recorded frequencies are no longer accurate, so this is not equivalent to combining the examples from the old task with the examples of the new task. By recording only the frequencies, instead of rebuilding the parse trees for all previous training examples in each cycle, we save both space and time for learning.

## 3.2 Experimental Study

To evaluate the proposed representation learner, we carry out two controlled experiments. We compare four alternatives of the proposed approach: 1) without transfer learning and no feature focus; 2) without transfer learning, but with feature focus; 3) with transfer learning (from unlabeled sub-grammars), and without feature focus; 4) with transfer learning from unlabeled sub-grammars and feature focus. Learners without labeled feature have no way of knowing what the feature is; instead, we report the accuracy that would be obtained using the non-terminal symbol that mostly frequently corresponds to the feature sub-grammar in the training examples. Note that we do not compare the proposed representation learner with the inside-outside algorithm, as Li et al. have shown that the base learner (i.e. the learner with no extension) outperforms the inside-outside algorithm.<sup>1</sup> All the experiments were run on a 2.53 GHz Core 2 Duo MacBook with 4GB of RAM.

In order to understand the generality and scalability of the proposed approach, we first design and carry out experiments in synthetic domains. The experiment results show that the learner with both transfer learning and feature focus (+*Transfer* +*Feature Focus*) has the steepest learning curve. Learners with a single extension (-*Transfer* +*Feature Focus*, and +*Transfer* -*Feature Focus*) have a slower learning curve comparing with the learner with both extensions (+*Transfer*

<sup>1</sup><http://rakaposhi.eas.asu.edu/nan-tist.pdf>

**Table 3.2:** Method summary

Three tasks:	T1, learn signed number T2, learn to find coefficient from expression T3, learn to find constant from equation
Three curricula:	T1 $\rightarrow$ T2 T2 $\rightarrow$ T3 T1 $\rightarrow$ T2 $\rightarrow$ T3
Number of training condition:	10
Training size in all but last tasks:	10
Training size in the last task:	1, 2, 3, 4, 5
Testing size:	100

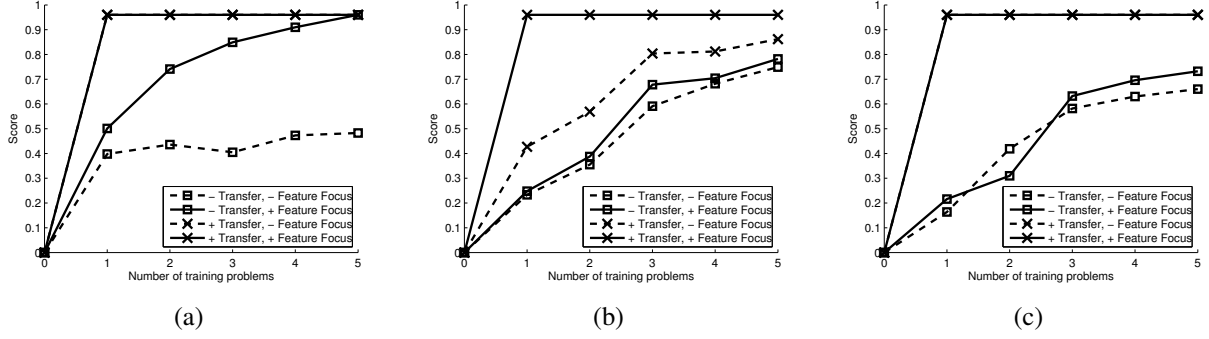
+*Feature Focus*), but both outperform the base learner (-*Transfer -Feature Focus*). All learners acquire the targeted feature within a reasonable amount of time, i.e., 1 – 266 milliseconds per training record with domains of size smaller than or equals to 25. For full details, please refer to [Li et al., 2012b].

In order to understand whether the proposed algorithm is a good model of real students, we carry out a controlled simulation study in algebra. Accelerated future learning, in which learning proceeds more effectively and more rapidly because of prior learning, is an interesting measure of robust learning. Two possible causes for accelerated future learning are a better learning strategy or stronger prior knowledge. Learning with feature focus is an example of using a better learning strategy during knowledge acquisition. Transfer learning from unlabeled sub-grammars is an example of developing stronger prior knowledge from previous training to prepare for better future learning. The objective of this study is to test 1) whether the proposed model could yield accelerated future learning with stronger prior knowledge and better learning strategies, 2) if so, how prior knowledge and learning strategies affect the learning outcome. In other words, can we model how students may learn later tasks more effectively after prior unsupervised or semi-supervised experience?

### 3.2.1 Methods

In order to understand the behavior of the proposed model, we designed three curricula. Three tasks are used across the three curricula. Task one is to learn about signed numbers. Task two is to learn how to recognize a coefficient from expressions in the form of  $\{\text{SignedNumber } x\}$ . Task three is to learn how to recognize a constant in the left-hand side from equations in the form of  $\{\text{SignedNumber } x - \text{Integer} = \text{SignedNumber}\}$ . The three curricula contain 1) task one, task two; 2) task two, task three; 3) task one, task two, and task three.

There were also 10 training sequences to control for a difference in training problems. The training data were randomly generated following the grammar corresponding to each task. For instance, task two’s grammar is shown in Table 3.1. In all but the last task, each learner was



**Figure 3.4:** Learning curves in the last task for four learners in curriculum (a) from task one to task two (b) from task two to task three (c) from task one and two to task three. Both prior knowledge transfer and the feature focus strategy produce faster learning.

given 10 training problems. For the last task, each learner was given training records.

To measure learning gain under each training condition both systems were tested on 100 expressions in the same form of the training data in the last task. For each testing record, we compared the feature extracted by the oracle grammars with that recognized by the acquired grammars. Note that in task two, 4% of the testing problems in task two were  $x$  and  $-x$ . To assess the accuracy of the model, we asked both systems to extract the feature from each problem. We then used the oracle grammar to evaluate the correctness of output. A brief summary of the method is shown in Table 3.2.

### 3.2.2 Measurements

To assess the learning outcome, we measure the learning rate in the last task of each curriculum to evaluate the effectiveness of the learners. The experiment tests whether the proposed model is able to yield accelerated future learning, that is, a faster learning rate in the last task either because of transferring prior learning or because of a better learning strategy. We compare the same four learners used in the previous simulations, that is, the combinations of transfer or not and feature focus or not. To evaluate the learning rate, we report learning curves for all four learners by the number of training problems given in the last task. The accuracy of the feature extraction task is averaged over 10 training conditions.

### 3.2.3 Experimental Results

As shown in Figure 3.4(a), with curriculum one, all four learners acquire better knowledge with more training examples. With five training problems, either transfer or feature focus are sufficient to acquire knowledge of score 0.96, while the base learner with neither was only able to achieve a score around 0.5. None of the learners are able to learn the coefficient of  $x$  and  $-x$  are 1 and

$-1$ , as it requires the feature learner to generate the number 1 in the coefficient, but 1 is not presented in  $x$  and  $-x$ . This difference between problems of the form  $Ax$  and problems of the form  $-x$  turns out to be an important distinction in human learning as well, which we will show in Chapter 8.

Both learners with transfer learning (*+Transfer -Feature Focus*, and *+Transfer +Feature Focus*) have the steepest learning curve. In fact, they reach a score of 0.96 with only one training example. The feature focus learner (*-Transfer +Feature Focus*) learns more slowly than the learners with transfer learning (*+Transfer -Feature Focus*, and *+Transfer +Feature Focus*), but is able to reach a score of 0.96 after five training examples. Learners that transfer prior grammar learning achieve faster future learning than those without transfer learning. The base learner (*-Transfer -Feature Focus*) learns most slowly. A careful inspection shows that without feature focus and transfer learning, the base learner is not able to acquire a grammar rule with a non-terminal symbol generally corresponding with the feature “coefficient”, though it does learn to identify positive coefficients (like many novice students). This causes the failure of identifying the feature symbol. Comparing the base learner (*-Transfer -Feature Focus*) and the learner with feature focus (*-Transfer +Feature Focus*) we can see that a better learning strategy also yields a steeper learning curve.

Similar results are also observed with curriculum two and curriculum three. In curriculum two, one interesting point is that, in some conditions, if a transfer learner, (*+Transfer -Feature Focus*) remembers the wrong knowledge acquired from task two, and transferred this knowledge to task three, the learner will perform even worse than the learner with no prior knowledge (*-Transfer -Feature Focus*). This indicates that more knowledge does not necessarily lead to steeper learning curves. Transferring incorrect knowledge leads to less learning.

In all three curricula, the transfer learner (*+Transfer -Feature Focus*) always outperforms the learner with the semantic non-terminal constraint (*-Transfer +Feature Focus*). This suggests that prior knowledge is more effective in accelerating future learning than this learning strategy.

### 3.3 Discussion

The main contribution of this work is to propose a computational model of representation learning. We exploit the connection between representation learning and grammar induction by extending an existing pCFG algorithm [Li et al., 2009] to support feature learning and transfer learning. It shares some ideas from previous work on grammar induction (e.g., [Wolff, 1982, Langley and Stromsten, 2000, Stolcke, 1994, Vanlehn and Ball, 1987]), which searches for the target grammar by adding or merging non-terminal symbols. Roark and Bacchiani [2003], Hwa [1999], and others have also explored transfer learning for pCFG. But most of the above approaches focus on the grammar induction task, rather than applying the techniques to representation learning as we do here.

Previous work in cognitive science has shown that “chunking” is an important component of human knowledge acquisition. Theories of the chunking mechanisms [Chase and Simon, 1973,

[Richman et al., 1995](#), [Gobet and Simon, 2000](#)] have been constructed. EPAM [[Chase and Simon, 1973](#)] is one of the first chunking theories proposed to explain key phenomena of expertise in chess. Learning occurs through the incremental growth of a discrimination network, where each node in the network is a chunk. It has been shown that chunks can be used to suggest plans, moves and so on. A later version of EPAM, EPAM-IV [[Richman et al., 1995](#)], extends the basic chunking mechanism to support a retrieval structure that enables domain-specific material to be rapidly indexed. In these theories, chunks usually refer to perceptual chunks. In addition, CHREST [[Gobet and Simon, 2000](#)] proposes a template theory, where the discrimination network contains both perceptual chunks and action chunks. A more detailed review of these work can be found in [[Gobet, 2005](#)]. Our work is similar to these works as we are also modeling the learning of perceptual chunks, a kind of representation learning, but differs from these theories since none of the above theories uses pCFG learning to model the acquisition of perceptual chunks.

In summary, we present a computational model of representation learning that yields accelerated future learning. We provide an empirical evaluation of our computational model, and compare four alternative versions of the proposed model. Results show how both stronger prior knowledge and a better learning strategy can yield accelerated future learning, and indicate that stronger prior knowledge produces faster learning outcomes compared with a better learning strategy.



# Chapter 4

## Learning for Operator Functions

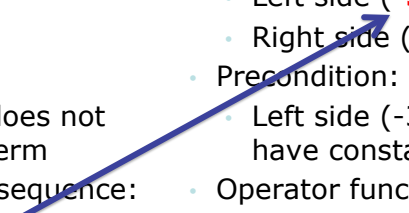
Given the promising results shown above, we believe the proposed representation learner is effective in acquiring representation knowledge, and is a good model of real students. To evaluate how the deep feature representation learner could affect the problem-solving learning of an intelligent agent, in this chapter, we present an integration of representation learning into such an agent, SimStudent.

### 4.1 Integrating Representation Learning into Skill Learning

As we have mentioned above, SimStudent is able to acquire production rules in solving complicated problems, but requires a set of operator functions given as prior knowledge. Some of the operator functions are domain-specific and require expert knowledge to build. In contrast, the representation learner acquires deep features that are essential for effective learning without requiring prior knowledge engineering. In order to both reduce the amount of prior knowledge engineering needed for SimStudent and to build a better model of real students, we present a novel approach that integrates representation learning into SimStudent.

Figure 4.1 shows a comparison between a production rule acquired by the original SimStudent and the corresponding production rule acquired by the extended SimStudent. As we can see, the coefficient of the left-hand side (i.e.,  $-3$ ) is included in the perceptual information part in the extended production rule. Therefore, the operator function sequence no longer needs the domain-specific operator, “get-coefficient”. To achieve this, we extend the representation learning algorithm as described below.

Previously, the perceptual information encoded in production rules was associated with elements in the graphical user interface (GUI) such as text field cells in the algebra equation solving interface. This assumption limited the granularity of observation SimStudent could achieve. In fact, the deep features we have discussed previously are perceptual information obtained at a fine-grained level. Representing these deep perceptual features may enhance the performance of the learning agent, and may eliminate or reduce the need for authors/developers to manually

- 
- |  |  |
|--|--|
| <ul style="list-style-type: none"> <li>• Original:</li> <li>• Skill divide (e.g. <math>-3x = 6</math>)</li> <li>• Perceptual information: <ul style="list-style-type: none"> <li>• Left side (<math>-3x</math>)</li> <li>• Right side (<math>6</math>)</li> </ul> </li> <li>• Precondition: <ul style="list-style-type: none"> <li>• Left side (<math>-3x</math>) does not have constant term</li> </ul> </li> <li>• Operator function sequence: <ul style="list-style-type: none"> <li>• Get coefficient (<math>-3</math>) of left side (<math>-3x</math>)</li> <li>• Divide both sides with the coefficient (<math>-3</math>)</li> </ul> </li> </ul> | <ul style="list-style-type: none"> <li>• Extended:</li> <li>• Skill divide (e.g. <math>-3x = 6</math>)</li> <li>• Perceptual information: <ul style="list-style-type: none"> <li>• Left side (<b><math>-3</math></b>, <math>-3x</math>)</li> <li>• Right side (<math>6</math>)</li> </ul> </li> <li>• Precondition: <ul style="list-style-type: none"> <li>• Left side (<math>-3x</math>) does not have constant term</li> </ul> </li> <li>• Operator function sequence: <ul style="list-style-type: none"> <li>• <del>Get coefficient (<math>-3</math>) of left side (<math>-3x</math>)</del></li> <li>• Divide both sides with the coefficient (<b><math>-3</math></b>)</li> </ul> </li> </ul> |
|--|--|

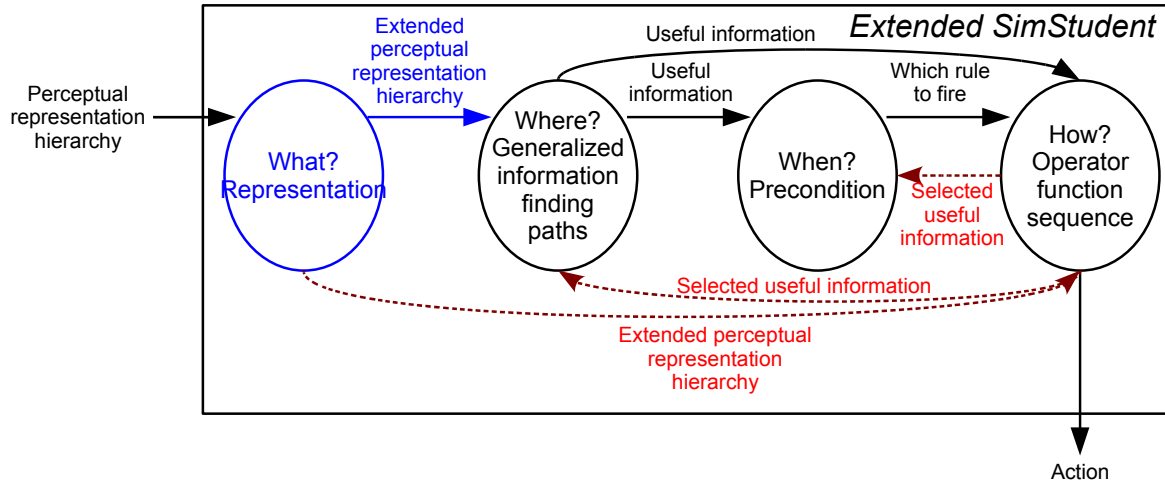
**Figure 4.1:** Original and extended production rules for divide in a readable format. Grammar learning allows extraction of information in where-part of the production rule and eliminated the need for domain-specific function authoring (get-coefficient) for use in the how-part.

encode domain-specific operator functions to extract appropriate information from appropriate parts of the perceptual input.

Figure 4.2 shows a high-level diagram that illustrates how the extended percept hierarchy is used by the learning components with red dashed lines. We first extend the perceptual representation hierarchy by the representation acquired by the representation learner, and then send the extended hierarchy to the how learner. The how learner finds an operator function sequence with the extended hierarchy, and selects a subset of the elements in the extended hierarchy. The where and when learners then carry out their learning processes with this selected set of useful information.

### 4.1.1 Extending the Perceptual Representation

More specifically, to improve perceptual representation, we first extend the percept hierarchy of GUI elements to further include the most probable parse tree for the content in the leaf nodes (e.g., text fields) by appending the parse trees as an extension of the GUI path leading to the associated leaf nodes. All of the inserted nodes are of type “subcell”. In the algebra example, this extension means that for cells that represent expressions corresponding to the sides of the equation, the extended SimStudent appends the parse trees for these expressions to the cell nodes. Let’s use  $-3x$  as an example. In this case, as presented in Figure 4.3, the extended hierarchy includes the parse tree for  $-3x$  as shown at the left side of Figure 3.1 as a subtree connecting to the cell node associated with  $-3x$ . With this extension, the coefficient ( $-3$ ) of  $-3x$  is now explicitly represented in the percept hierarchy. If the extended SimStudent includes this subcell as a percept in production rules, as shown at the right side of Figure 4.1, the new production rule does not



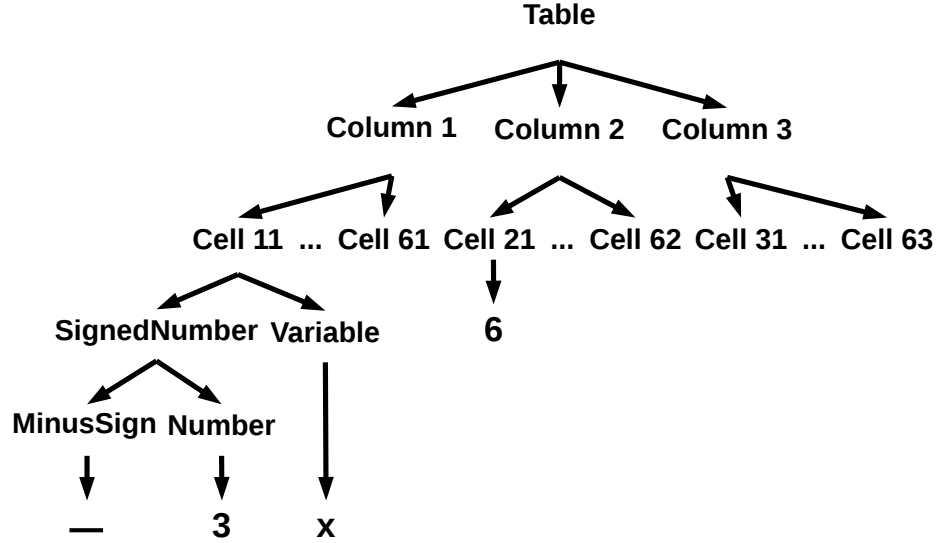
**Figure 4.2:** A diagram about how the extended SimStudent makes use of the representation acquired by the representation learner, and reacts with the environment. Solid lines show the information that flows through components during execution. Red dashed lines illustrate how the acquired representation is used by the learning components during learning. Information about the training examples is not presented.

need the first domain-specific operator function “coefficient”.

### 4.1.2 Extending the Perceptual Learner

However, extending the percept hierarchy presents challenges to the original perceptual learner. First of all, since the extended subcells are not associated with GUI elements, we can no longer depend on the tutor to specify relevant perceptual input for SimStudent, nor can we simply specify all of the subcells in the parse trees as relevant perceptual information; otherwise, the acquired production rules would include redundant information that would hurt the generalization capability of the perceptual learner. For example, consider problems  $-3x=6$  and  $4x=8$ . Although both examples could be explained by dividing both sides with the coefficient, since  $-3x$  has eight nodes in its parse tree, while  $4x$  has five nodes, the original perceptual learner will not be able to find one set of generalized paths that explain both training examples. Moreover, not all of the subcells are relevant percepts in solving the problem. Including unnecessary perceptual information into production rules could easily lead to computational issues. Second, since the size of the parse tree for an input depends on the input length, the assumption of fixed percept size made by the “where” learner no longer holds. In addition, how the inserted percepts should be ordered is not immediately clear. To address these challenges, we extend the original perceptual learner to support acquisition of perceptual information with redundant and variable-length percept lists.

To do this, SimStudent first includes all of the inserted subcells as candidate percepts, and calls the operator function sequence learner to find an operator function sequence that explains all

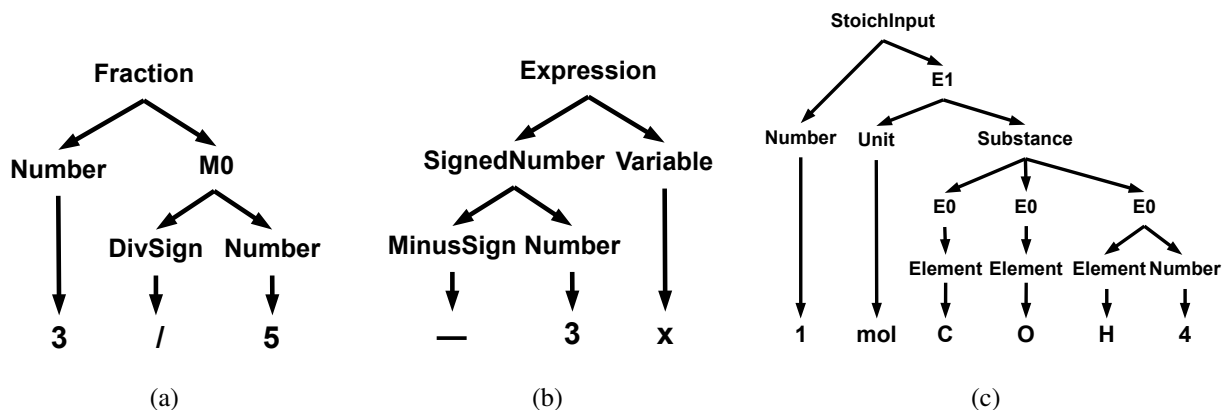


**Figure 4.3:** The extended perceptual hierarchy associated with the interface in equation solving.

of the training examples. In our example, the operator function sequence for (*divide -3*) would only contain one operator function “divide”, since *-3* is already included in the candidate percept list. The perceptual learner then removes all of the subcells that are not used by the operator function sequence from the candidate percept list. Hence, subcells such as *-*, *3* and *x* would not be included in the percept list any more. Since all of the training example action records share the same operator function sequence, the number of percepts remaining for each example action record should be the same. Next, the percept learner arranges the remaining subcell percepts based on their order of being used by the operator function sequences. After this process, the percept learner now has a set of percept lists that contains a fixed number of percepts ordered in the same fashion. We can then switch to the original percept learner to find the least general paths for the updated percept lists. In our example for skill “divide”, as shown at the right side of Figure 4.1, the perceptual information part of the production rule would contain three elements, the left-hand side and right-hand side cells which are the same as the original rule, and a coefficient subcell which corresponds to the left child of the variable term. Note that since we removed the redundant subcells, the acquired production rule now works with both  $-3x=6$  and  $4x=8$ .

## 4.2 Examples of Integration

Here are a few more examples to demonstrate how the extended perceptual learner enables the removal of domain-specific operator functions, while maintaining efficient skill knowledge acquisition. Figure 4.4 shows the parse trees of example input strings acquired by the representation learner. The deep features are associated with nonterminal symbols in the parse trees.



**Figure 4.4:** Example parse trees learned by the representation learner in three domains, a) fraction addition, b) equation solving, c) stoichiometry.

In fraction addition, one of the important operator functions in this domain is getting the denominator of the addend (i.e., (*get-denominator ?val*)). Figure 4.5(a) shows an example parse tree for  $3/5$ . The extended SimStudent can directly get the denominator 5 from the non-terminal symbol *Number* in rule  $M0 \rightarrow 1.0, DivSign, Number$ . Then, the operator function (*get-denominator ?val*) is replaced by a more general operator function (*copy-string ?val*).

Another important domain-specific operator function in equation solving is getting the coefficient of some expression (i.e., (*get-coefficient ?val*)). With the representation learner, the coefficient of an expression can be extracted by directly taking the signed number (i.e., *SignedNumber*) in rule  $Expression \rightarrow 1.0, SignedNumber, Variable$ . Again, the domain-specific operator function (*get-coefficient ?val*) is replaced by the domain-general operator function (*copy-string ?val*).

In stoichiometry, (*molecular-ratio ?val0 ?val1*) is a domain-specific operator function. Instead of programming this operator function, after integrated with representation learning, the output can now be generated by taking the “*Number*” in grammar rule  $E0 \rightarrow 0.5 Element, Number$ , and then concatenating with the unit *mol* and the individual substance “*Element*”. Thus, the original operator function (*molecular-ratio ?val0 ?val1*) is replaced by the domain-general operator function concatenation (i.e., (*concat ?val2 ?val3*)).

As we can see from the above examples, without representation learning, SimStudent needs three different operator functions across three domains. After integrated with representation learning, these three seemingly distinctive operator functions can all be removed, and replaced by domain-general operator functions, (*copy-string ?val*) and (*concat ?val0 ?val1*). It is not hard to see that the amount of knowledge engineering effort in developing the three original operator functions is larger than that of implementing copying and concatenating strings.

Domain Name	# of Training Problems	# of Testing Problems
Fraction Addition	40	6
Equation Solving	24	11
Stoichiometry	16	3

**Table 4.1:** Number of training problems and testing problems presented to SimStudent.

## 4.3 Experimental Study

To further quantitatively evaluate the amount of required prior knowledge encoding and the learning effectiveness of SimStudent, we carry out a controlled simulation study in three domains: fraction addition, equation solving, and stoichiometry.

### 4.3.1 Methods

We compare three versions of SimStudent: two original SimStudents without representation learning, and one extended SimStudent with representation learning. One of the original SimStudents is given both domain-general and domain-specific operator functions (*O+Strong Ops*). The other is given only domain-general operator functions (*O+Weak Ops*). The extended SimStudent is also only given domain-general operator functions (*E+Weak Ops*).

In each domain, the three SimStudents are trained on 12 problem sequences over the same set of problems in different orders. Both training and testing problems are gathered from classroom studies on human students. SimStudent is tutored by automatic tutors that are similar to those used by human students. The number of training and testing problems is listed in Table 4.1.

**Fraction Addition:** In the fraction addition domain, SimStudent was given a series of fraction addition problems of the form

$$\frac{numerator_1}{denominator_1} + \frac{numerator_2}{denominator_2}$$

All numerators and denominators are positive integers. The problems are of three types in the order of increasing difficulty: 1) *easy problems*, where the two addends share the same denominators (i.e.,  $denominator_1 = denominator_2$ , e.g.,  $1/4 + 3/4$ ), 2) *normal problems*, where one denominator is a multiple of the other denominator (i.e.,  $GCD(denominator_1, denominator_2) = denominator_1$  or  $denominator_2$ , e.g.,  $1/2 + 3/4$ ), 3) *hard problems*, where no denominator is a multiple of the other denominator (e.g.,  $1/3 + 3/4$ ). In this case, students need to find the common denominator (e.g. 12 for  $1/3 + 3/4$ ) by themselves. Both the training and testing problems were selected from a classroom study of 80 human students using an automatic fraction addition tutor. The number of training problems is 20, and the number of testing problems is 6.

**Equation Solving:** The second domain in which we tested SimStudent is equation solving. Equation solving is a more challenging domain since it requires more complicated prior knowl-

edge to solve the problem. For example, it is hard for human students to learn what is a coefficient, and what is a constant. Also, adding two terms together is more complicated than adding two numbers.

In this experiment, we evaluated SimStudent based on a dataset of 71 human students in a classroom study using an automatic tutor, CTAT [Aleven et al., 2009]. The problems are also in three types: 1) problems of the form  $S_1 + S_2V = S_3$ , 2),  $V/S_1 = S_2$ , 3)  $S_1/V = S_2$ , where  $S_1$  and  $S_2$  are signed numbers, and  $V$  is a variable. Note that the terms in the above problem forms can appear in any order, and surrounded with parenthesis. There were 12 training problems, and 11 testing problems in the experiment.

**Stoichiometry:** Lastly, we evaluated SimStudent in a chemistry domain, stoichiometry. Stoichiometry is a branch of chemistry that deals with the relative quantities of reactants and products in chemical reactions. We selected stoichiometry because it is different from equation solving and fraction addition in nature. In the stoichiometry domain, SimStudent was asked to solve problems such as “How many moles of atomic oxygen (O) are in 250 grams of  $P_4O_{10}$ ? (Hint: the molecular weight of  $P_4O_{10}$  is 283.88 g  $P_4O_{10}$  / mol  $P_4O_{10}$ ).”. 8 training problems and 3 testing problems were selected from a classroom study of 81 human students using an automatic stoichiometry tutor [Mclaren et al., 2008].

To solve the problems, SimStudent needs to acquire three types of skills: 1) unit conversion (e.g. 0.6 kg  $H_2O$  = 600 g  $H_2O$ ), 2) molecular weight (e.g. There are 2 moles of  $P_4O_{10}$  in  $283.88 \times 2$  g  $P_4O_{10}$ ), 3) composition stoichiometry (e.g. There are 10 moles of O in each mole of  $P_4O_{10}$ ). The problems are of three types ordered in increasing difficulty, where each later type adds one more skill comparing with its former type.

To generate different curricula given to SimStudent, for each domain, we first group the problems of the same type together. Since there are three types of problems, we have three groups in each domain: *group - 1*, *group - 2*, and *group - 3*. Then, there are six different orders of these three groups. For each order (e.g. [*group - 1*, *group - 2*, *group - 3*]), we generate one blocked-ordering curriculum by repeating the same problems in each group right after that group’s training was done (e.g., [*group - 1*, *group - 1’*, *group - 2*, *group - 2’*, *group - 3*, *group - 3’*]). To generate the interleaved-ordering curriculum, the same problems will be repeated once the whole set of problems were done (e.g. [*group - 1*, *group - 2*, *group - 3*, *group - 1’*, *group - 2’*, *group - 3’*]).

After this manipulation, we end up having 12 curricula of different orders for each domain as shown in Table 4.2. Six of them are blocked-ordering curricula, whereas the other six were interleaved-ordering curricula. SimStudent was trained and tested on all these curricula. the results are the average over all curricula.

### 4.3.2 Measurements

We evaluate the performance of SimStudent with two measurements. We use the number of domain-specific and domain-general operator functions used in three domains to measure the



Blocked-Ordering Curricula	Interleaved-Ordering Curricula
1, 1', 2, 2', 3, 3'	1, 2, 3, 1', 2', 3'
1, 1', 3, 3', 2, 2'	1, 3, 2, 1', 3, 2'
2, 2', 1, 1', 3, 3'	2, 1, 3, 2', 1', 3'
2, 2', 3, 3', 1, 1'	2, 3, 1, 2', 3', 1'
3, 3', 1, 1', 2, 2'	3, 1, 2, 3', 1', 2'
3, 3', 2, 2', 1, 1'	3, 2, 1, 3', 2', 1'

**Table 4.2:** 12 curricula of different orders for each domain.

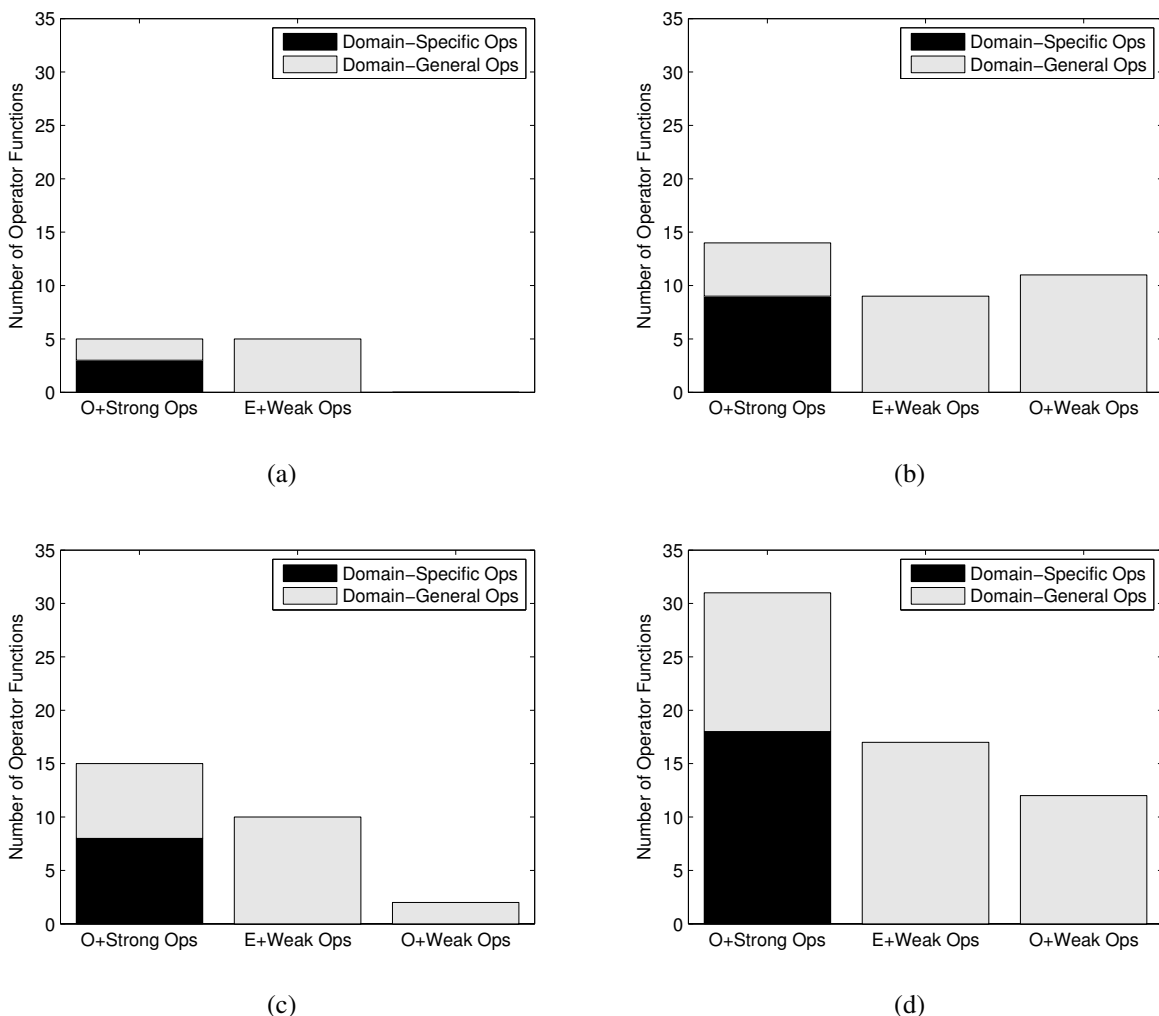
amount of prior knowledge engineering needed. In addition, we count the number of lines of Java code developed for each operator functions, and use this as a secondary measurement to assess the amount of knowledge engineering. To assess learning effectiveness, we define a *step score* for each step in the testing problem. Among all next steps proposed by SimStudent, we count the number of next steps that are correct, and compute the step score as the number of correct next steps proposed divided by the total number of correct steps plus the number of incorrect next steps proposed. This measurement evaluates the quality of production rules in terms of both precision and recall. For example, if there were four possible correct next steps, and SimStudent proposed three, of which two were correct, and one was incorrect, then only two correct next steps were covered, and thus the step score is  $2/(4+1)=0.4$ . We report the average step score over all testing problem steps for each curriculum.

### 4.3.3 Experimental Results

Not surprisingly, only the original SimStudent given the strong set of operator functions (*O+Strong Ops*) uses domain-specific operator functions. As shown in Figure 4.5, across three domains, it requires at least as many operator functions as the extended SimStudent without domain-specific operator functions (*E+Weak Ops*). Moreover, since domain-specific operator functions are not reusable across domains, the original SimStudent with domain-specific operator functions (*O+Strong Ops*) requires nearly twice as many operator functions (31 vs. 17) as that of the extended SimStudent (*E+Weak Ops*) needed.

As presented in Figure 4.6, since the domain-specific operator functions often require more knowledge engineering effort, not surprisingly, in all three domains, the original SimStudent given domain-specific operator functions (*O+Strong Ops*) requires more than twice as much coding compared to the extended SimStudent given only domain-general operator functions (*E+Weak Ops*). The total number of lines of code required for the operator functions used by the extended SimStudent (*E+Weak Ops*) is 645, whereas the total number of lines of code programmed for the original SimStudent (*O+Strong Ops*) is 6789, which is more than ten times the size of the code needed by the extended SimStudent. In the equation solving domain, the original SimStudent (*O+Strong Ops*) needed 4548 lines of code, whereas the extended SimStudent (*E+Weak Ops*) only needs 555 lines of code. The original SimStudent with only domain-general

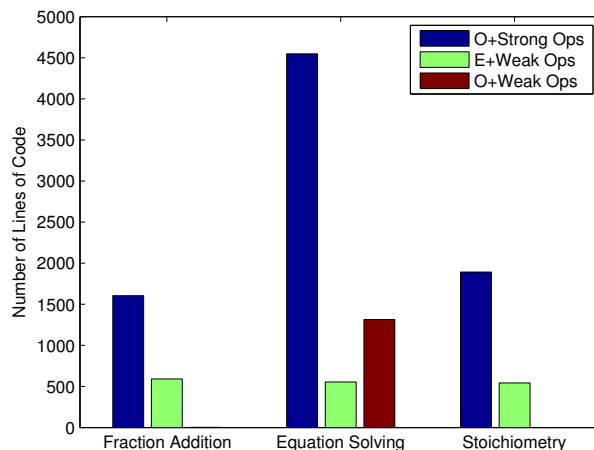




**Figure 4.5:** Number of domain-specific and domain-general operator functions used in acquired production rules, a) fraction addition, b) equation solving, c) stoichiometry, d) across three domains.

operator functions (*O+Weak Ops*) needs more knowledge engineering than the extended SimStudent (*E+Weak Ops*) in equation solving, but requires less knowledge engineering in the other two domains. However, as we will see later, it performs much worse than the extended SimStudents (*E+Weak Ops*).

Learning curves of the three SimStudents are presented in Figure 4.7. Across three domains, without domain-specific prior knowledge, the original SimStudent (*O+Weak Ops*) is not able to achieve a step score more than 0.3. Given domain-specific operator functions, the original SimStudent (*O+Strong Ops*) is able to perform reasonably well. It obtains a step score around 0.85 in equation solving. However, its performance is still not as good as the extended SimStudent. Given all training problems, the extended SimStudent (*E+Weak Ops*) performs slightly



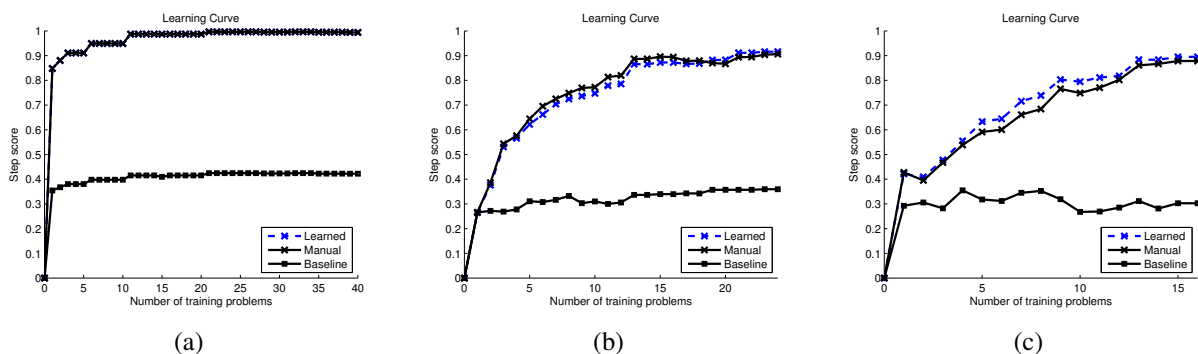
**Figure 4.6:** Number of lines of Java code developed for operator functions used in acquired production rules.

better than the original SimStudent with domain-specific prior knowledge in equation solving. It (*E+Weak Ops*) achieves significantly ( $p < 0.0001$ ) better step scores than the original SimStudent given domain-specific operator functions (*O+Strong Ops*) in two other domains. Hence, we conclude that the extended SimStudent acquires skill knowledge, which is as or more effective than the original SimStudent, while requiring less prior knowledge engineering.

## 4.4 Discussion

The main objective of this work is to reduce the amount of knowledge engineering effort needed in building an intelligent learning agent by integrating representation learning into skill learning. There has been considerable research on learning within agent architectures. Soar [Laird et al., 1986] uses a chunking mechanism to acquire knowledge that constrains problem-space search. Another architecture ACT-R [Anderson, 1993] creates new production rules through a compilation process that gradually transforms declarative representations into skill knowledge [Taatgen and Lee, 2003]. Anderson and Thompson [Anderson and Thompson, 1989] developed an analogical problem solving mechanism, and integrated it into an earlier version of ACT-R to assist skill learning. ICARUS [Langley and Choi, 2006] acquires complex hierarchical skills in the context of problem solving. Unlike those theories, SimStudent puts more emphasis on knowledge-level learning (cf., [Dietterich, 1986]) achieved through induction from positive and negative examples. It integrates ideas of theories of perceptual chunking [Richman et al., 1995] as a basis for improving knowledge representations that, in turn, facilitate better learning of problem solving skills.

Another closely related research area is learning procedural knowledge by observing others' behavior. Classical approaches include explanation-based learning [Segre, 1987, Mooney, 1990],



**Figure 4.7:** Learning curves of three SimStudents in three domains, a) fraction addition, b) equation solving, c) stoichiometry.

learning apprentices [Mitchell et al., 1985] and programming by demonstration [Cypher et al., 1993, Lau and Weld, 1998]. Neves [Neves, 1985] proposed a program that learns production rules from worked-out solutions and by working problems, and demonstrated the algorithm in algebra, but did not show results across domains. Most of these approaches used analytic methods to acquire candidate procedures. Other works on transfer learning (e.g., [Raina et al., 2006, Niculescu-Mizil and Caruana, 2007, Torrey et al., 2007, Richardson and Domingos, 2006]) also share some resemblance with our work. They focus on improving the performance of learning by transferring previously acquired knowledge from another domain of interest. However, to the best of our knowledge, none of the above approaches uses the transfer learner to acquire a better representation that reveals essential percept features, and to integrate it into an intelligent agent.

To sum up, we propose a novel approach that integrates representation learning into an intelligent agent, SimStudent, as an extension of the perception module. We show that after the integration, the extended SimStudent is able to achieve better or comparable performance without requiring any domain-specific operator function as input.

Other research in cognitive science also attempts to use probabilistic approaches to model the process of human learning. Kemp and Xu [Kemp and Xu, 2008] applied a probabilistic model to capture principles of infant object perception. Kemp and Tenenbaum [Kemp and Tenenbaum, 2008] used a hierarchical generative model to show the acquisition process of domain-specific structural constraints. But again, neither of the above approaches tend to use the probabilistic model as a representation acquisition component in a learning agent. Additionally, research on deep architectures [Bengio, 2009] shares a clear resemblance with our work and has been receiving increasing attention recently. Theoretical results suggest that in order to learn complicated functions such as AI-level tasks, deep architectures that are composed of multiple levels of non-linear operation are needed. Although not having been studied much in the machine learning literature due to the difficulty in optimization, there are some notable exceptions in the area including convolutional neural networks [LeCun et al., 1989, Lecun et al., 1998, Simard et al., 2003, Ranzato et al., 2007], sigmoidal belief networks learned using variational approximations [Dayan et al., 1995, Hinton et al., 1995, Saul et al., 1996, Titov and Henderson, 2007],

and deep belief networks [[Hinton, 2007](#), [Bengio et al., 2010](#)]. While both the work in deep architectures and our work are interested in modeling complicated functions through non-linear features, the tasks we work on are different. Deep architectures are used more often in classification tasks whereas our work focuses on simulating human problem solving and learning of math and science.

[Ohlsson's 2008](#) reviews how different learning models are employed during different learning phases in intelligent systems. Our work on integrating representation learning and skill learning also reflects how one learning mechanism is able to aid other learning processes in an intelligent system.

# Chapter 5

## Learning Perceptual Hierarchies

With the above extension, we have shown that we are able to reduce the amount of knowledge engineering effort in encoding domain-specific operator functions. In this chapter, we shift our focus on reducing the knowledge engineering effort in constructing the perceptual hierarchy. As mentioned before, in order to identify useful information from the interface, SimStudent’s perceptual learner needs to be given a perceptual hierarchy that helps it to understand of the way the interface is organized. This is a similar problem people face daily. Every day, people view and understand many novel two-dimensional (2-D) displays such as tables on webpages and software user interfaces. How do humans learn to process such displays?

As an example, Figure 2.1 shows a screenshot of one interface to an intelligent tutoring system that is used to teach students how to solve algebraic equations. The interface should be viewed as a table of three columns, where the first two columns of each row contain the left-hand side and right-hand side of the equation, and the third column names the skill applied. In tutoring, students enter data row by row, a strategy that requires a correct intuitive understanding of how the interface is organized. More complicated displays may contain multiple tables, and require information sharing and coordination to fully understand the whole display.

Incorrect representation of the interface may lead to inappropriate generalization of the acquired skill knowledge, such as generalizing the skill for adding two numerators to adding two denominators in fraction addition. A good representation of the display on which future learning is based is essential in achieving effective learning. How such representation is acquired remains unknown. Past instances of SimStudent have used a hand-coded hierarchical representation of the interface, which is both time-consuming, and less psychologically plausible. Here we consider replacing that hand-coded element with a learned representation.

### 5.1 Learning to Perceive Two-Dimensional Displays

More generally, we consider using a two-dimensional variant of a probabilistic context-free grammar (pCFG) to model how a user perceives the structure of a user interface, and propose

a novel 2-D pCFG learning algorithm to model acquisition of this representation. Our learning method exploits both the spatial layout of the interface, and temporal information about when users interact with the interface. The alphabet of the grammar is a vocabulary of symbols representing primitive interface-element types. For example, in Figure 2.1, the type of the cells in the first two columns is *Expression*, and the type of the last cell in the each column is *Skill*. (In SimStudent, these primitive types can be learned from prior experience.) We extend the ordinary one-dimensional (1-D) pCFG learner [Li et al., 2010] as described in Chapter 3 to acquire two-dimensional grammar rules, using a two-dimensional probabilistic version of the Viterbi training algorithm to learn parameter weights and a structure hypothesizer that uses spatial and temporal information to propose grammar rules.

We then integrate this two-dimensional representation learner into SimStudent. Previously, we had to manually encode such representation, which is both time consuming and error prone. We now extend SimStudent by replacing the hand-coded display representation with the statistically learned display representation. We demonstrate the proposed algorithm in tutoring systems, and for simplicity will refer to terminal symbols in the grammar as interface element, but we emphasize that the proposed algorithm should work for two-dimensional displays of other types as well. We evaluate the proposed algorithms in both synthetic domains and real world domains, with and without integration into SimStudent. Experimental results show that the proposed learning algorithm is effective in acquiring user interface layouts. The SimStudent with the proposed representation learner acquired domain knowledge at similar rates to a system with hand-coded knowledge. The main contribution of this work is to use probabilistic grammar induction to model learning to perceive two-dimensional visual displays.

### 5.1.1 Problem Definition

To learn the representation of a 2-D display, we first need to formally define the input and output of the problem.

**Input:** The input to the algorithm is a set of records,  $\mathcal{R} = \{R_1, R_2, \dots, R_n\}$ , associated with examples shown on the display observed by people. Figure 2.1 shows one problem example in this algebra tutor interface. Each record,  $R_i$  ( $i = 1, 2, \dots, n$ ), records how and when the elements in the display are filled out by users. Thus,  $R_i$  is a sequence of tuples,  $\langle T_{i1}, T_{i2}, \dots, T_{im} \rangle$ , where each tuple,  $T_{ik}$  ( $k = 1, 2, \dots, m$ ), is associated with one display element that is used in solving the problem. The tuples in a record are ordered by time. For example, to solve the problem,  $-3x+2 = 8$ , shown in Figure 2.1, the cells in the first three rows (except for the last cell of the third row) are used. We do not assume that meta-elements such as columns and rows are given, but we will assume that each display element occupies a rectangular region, and that we can detect when regions are adjacent. In this case,  $R_i$  will contain 12 tuples,  $\langle T_{i1}, T_{i2}, \dots, T_{i12} \rangle$ , that correspond to the eight cells, *Cell 11*, *Cell 12*, *Cell 13*, *Cell 21*, *Cell 22*, *Cell 23*, *Cell 31*, and *Cell 32*, and the four buttons, *done*, *help*,  $<<$ , and  $>>$ .

Each tuple consists of seven items,

$$T_{ik} = \langle type, x_{left}, x_{right}, y_{up}, y_{bottom}, timestamp_{start}, timestamp_{end} \rangle$$

where *type* is the type of the input to the display element,  $x_{left}$ ,  $x_{right}$ ,  $y_{up}$ , and  $y_{bottom}$  define the  $x$  and  $y$  coordinates of the space the element ranges over, and  $timestamp_{start}$  and  $timestamp_{end}$  are the start and ending time when the display element is filled out by the user. For example, given the problem  $-3x+2 = 8$ , the tuple associated with *Cell 11* is  $T_{i1} = \langle Expression, 0, 1, 0, 1, 0, 0 \rangle$ . The timestamp of *Cell 11* is 0, since both *Cell 11* and *Cell 21* were entered first by the tutor as the given problem. As mentioned above, we have developed a 1-D pCFG learner that acquires parse structures of 1-D strings. The type of the input is the non-terminal symbol associated with the parse tree of the content. Hence, the type of  $-3x+2$  is *Expression*.

**Output:** Given the input, the objective of the grammar learner is to acquire a 2-D pCFG,  $\mathcal{G}$ , that best captures the structural layout given the training records, that is,

$$\arg \max_{\mathcal{G}} p(\mathcal{R} | \mathcal{G})$$

under the constraint that all records share the same parse structure (i.e., layout). We will explain this in more detail in the algorithm description section.

The output of the layout learner is a two-dimensional variant of pCFG [Chou, 1989], which we define below. When used to parse a display, this grammar will generate a tree-like hierarchical grouping of the display elements.

**Two-Dimensional pCFG:** 2-D pCFG is an extended version of 1-D pCFG. Each 2-D pCFG,  $\mathcal{G}$ , is defined by a four-tuple,  $\langle \mathcal{V}, \mathcal{E}, Rules, S \rangle$ .  $\mathcal{V}$  is a finite set of non-terminal symbols that can be further decomposed to other non-terminal or terminal symbols.  $\mathcal{E}$  is a finite set of terminal symbols, that makes up the actual content of the “2-D sentence”. In our algebra example, the terminal symbols of the visual display are the input types associated with the display elements (e.g., *Expression*, *Skill*). *Rules* is a finite set of 2-D grammar rules.  $S$  is the start symbol.

Each 2-D grammar rule is of the form

$$V \rightarrow p, [direction] \gamma_1 \gamma_2 \dots \gamma_n$$

where  $V \in \mathcal{V}$ ,  $p$  is the probability of the grammar rule used in derivations<sup>1</sup>, and  $\gamma_1, \gamma_2, \dots, \gamma_n$  is either a sequence of terminal symbols or a sequence of non-terminal symbols. Without loss of generality, in this case, we only consider grammar rules that have one or two symbols at the right side of the arrow.

*direction* is a new field added for the 2-D grammar. It specifies the spatial relation among its children. The value of the direction field can be  $d$ ,  $h$ , or  $v$ .  $d$  is the default value set for grammar rules that have only one child, in which case there is no direction among the children.  $h$  ( $v$ )

<sup>1</sup>The sum of the probabilities associated with rules that share the same head,  $V$ , equals to 1.

**Table 5.1:** Part of the two-dimensional probabilistic context free grammar for the equation solving interface

---

Terminal symbols:	<i>Expression, Skill</i> ;
Non-terminal symbols:	<b>Table, Row, Equation, Exp, Ski</b>
<b>Table</b> → 0.7, [ <i>v</i> ]	<b>Table Row</b>
<b>Table</b> → 0.3, [ <i>d</i> ]	<b>Row</b>
<b>Row</b> → 1.0, [ <i>h</i> ]	<b>Equation Ski</b>
<b>Equation</b> → 1.0, [ <i>h</i> ]	<b>Exp Exp</b>
<b>Exp</b> → 1.0, [ <i>d</i> ]	<i>Expression</i>
<b>Ski</b> → 0.5, [ <i>d</i> ]	<i>Skill</i>

---

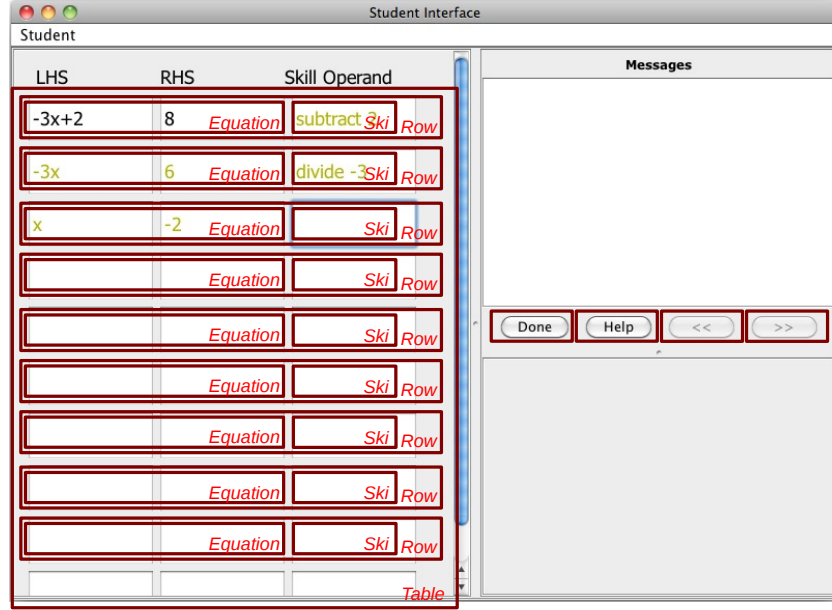
means the children generated by the grammar rule should be placed horizontally (vertically) with respect to each other. An example of a two-dimensional pCFG of the equation solving interface is shown in Table 5.1<sup>2</sup>. The corresponding layout is presented in Figure 5.1. The rows in the table are placed vertically with respect to other rows. Thus, the direction field in the grammar rule “**Table** → 0.7, [*v*] **Table Row**” is set to be *v*. On the other hand, the equation should be placed horizontally with the skill cell in the third column, so the direction field of “**Row** → 1.0, [*h*] **Equation Ski**” is *h*. These three direction values form the original direction value set.

Since the interface elements may not form a rectangle sometimes (e.g., the table and the buttons in the equation solving interface), we further extend the direction field to have two additional values *pv* and “*ph*”. *pv* (*ph*) means that the children of the grammar rule should be placed vertically (horizontally) with respect to each other, but the parts in the interface associated with these children do not have to form a rectangle. As shown in Figure 5.1, the table in the left side and the buttons in the right side can be placed horizontally, but do not form a rectangle. In this case, the grammar rule should use *ph* instead of *h* as the directional field value. These direction values are less preferred than the original values. Grammar rules that have such direction values will only be added if no more rules with directions *d*, *h*, or *v* can be found.

**Layout:** Given the 2-D pCFG, the final output of the display representation is a hierarchical grouping of the display elements, which we will call a layout, *L*. Figure 5.1 shows an example layout of the equation solving interface. The left side of the interface contains a row-ordered table, where each row is further divided into an equation and a skill. The right side of the interface contains a list of buttons that can be pressed by students to ask for help or to indicate when he/she considers the problem is solved.

<sup>2</sup>The non-terminal symbols are replaced with meaningful names here. The symbols in the learned grammars are synthetic-generated symbols.





**Figure 5.1:** An example layout of the interface where SimStudent is being tutored in an equation solving domain.

### 5.1.2 Learning Mechanism

Now that we have formally defined the learning task, we are ready to describe the 2-D display layout learner. In Chapter 3, we have proposed a 1-D representation learner [Li et al., 2010], and have shown that the 1-D representation learner acquires knowledge more effectively and runs faster than the inside-outside algorithm [Lari and Young, 1990]<sup>3</sup>. Hence, we further extend the one-dimensional grammar learner to acquire a 2-D pCFG from two-dimensional training records.

Algorithm 5.1 shows the pseudo code of the 2-D display layout learner. The learning algorithm iterates between a greedy structure hypothesizer (GSH) and a Viterbi training phase. The GSH tries to construct non-terminal symbols as well as grammar rules that could parse all input records,  $\mathcal{R}$ . The set of constructed rules are then set as the start point for the Viterbi training algorithm. Next, the Viterbi training algorithm iteratively re-estimates the probabilities associated with all grammar rules until convergence. If the grammar rules are not sufficient in generating a layout in the Viterbi training algorithm, GSH is called again to add more grammar rules. This process continues until at least one layout can be found.

Since an appropriate way of transferring previously acquired knowledge to later learning process could potentially improve the learning speed, we further designed a learning mechanism that transfers the acquired grammar with the application frequency of each rule from previous tasks to future tasks. Due to the limited space, we will not present the detail of this extension in this

<sup>3</sup>[rakaposhi.eas.asu.edu/nan-tist.pdf](http://rakaposhi.eas.asu.edu/nan-tist.pdf).

---

**Algorithm 5.1:** *2D-Layout-Learner* constructs a set of grammar rules,  $\mathcal{Rules}$ , from the training records,  $\mathcal{R}$ , and a set of terminal symbols  $\mathcal{E}$ .

---

**Input:** Record Set  $\mathcal{R}$ , Terminal Symbol Set  $\mathcal{E}$

```

1  $\mathcal{Rules} := \phi$ ;
2 while not-all-records-have-one-layout( $\mathcal{R}$ ,  $\mathcal{Rules}$ ) do
3    $\mathcal{Rules} := \text{GSH}(\mathcal{R}, \mathcal{E}, \mathcal{Rules})$ ;
4    $\mathcal{Rules} := \text{Viterbi-training}(\mathcal{R}, \mathcal{Rules})$ ;
5 end
6 return  $\mathcal{Rules}$ 

```

---

paper.

## Viterbi Training

Given a set of grammar rules from the GSH step, the Viterbi training algorithm tunes the probabilities on the grammar set, and removes unused rules.<sup>4</sup> We consider an iterative process. There are two steps in each iteration.

One key difference between learning the parse trees of 1-D strings and learning the GUI element layout is that the parse trees for different input contents are different (e.g.,  $-3x$  vs.  $5x+6$ ), whereas the GUI elements should always be organized in the same way even if the input contents in the GUI elements have changed from problem to problem. For instance, students will always perceive the equation solving interface as multiple rows, where each row consists of an equation along with a skill operation, no matter which problem they are given. Therefore, instead of finding a grammar that parses the interface given specific input, the learning algorithm should acquire one layout for the interface across different problems. This effectively adds a constraint on the learning algorithm.

In the first step, the algorithm computes the most probable parse trees,  $\mathcal{T}$ , for all training records using the current rules, under the constraint that the parse structure among these trees should be the same, that is,

$$\begin{aligned}
\mathcal{T} &= \arg \max_{\mathcal{T}} p(\mathcal{T} \mid \mathcal{R}, \mathcal{G}, S) \\
&= \bigcup_{i=1,2,\dots,n} \arg \max_{T_i} p(T_i \mid R_i, \mathcal{G}, S) \\
&\text{s.t. } \text{parse}(T_1) = \text{parse}(T_2) = \dots = \text{parse}(T_n) \forall T_i \in \mathcal{T}
\end{aligned}$$

where  $T_i$  is the parse tree with root  $S$  for record  $R_i$  given the current grammar  $\mathcal{G}$ , and  $\text{parse}(T_i)$  denotes the parse structure of  $T_i$  ignoring the symbols associated with the parse nodes<sup>5</sup>

<sup>4</sup>More detailed discussion on why a Viterbi training algorithm instead of the standard CKY is used can be found in [Li et al., 2011b], which is mainly because of overfitting.

<sup>5</sup>In the case that some record uses less elements than the other records (e.g., simpler problems that require less

Since any subtree of a most probable parse tree is also a most probable parse subtree, we have

$$p(T_i | R_i, \mathcal{G}, S_i) = \max_{rule, idx} \begin{cases} p(rule | \mathcal{G}) \times p(T_{i,1} | R_{i,1}, \mathcal{G}, S_{i,1}) \times p(T_{i,2} | R_{i,2}, \mathcal{G}, S_{i,2}) & \text{if rule is } S_i \rightarrow p(rule|\mathcal{G}), [direction] S_{i,1} S_{i,2}, \\ p(rule | \mathcal{G}) \times p(T_{i,1} | R_i, \mathcal{G}, S_{i,1}) & \text{if rule is } S_i \rightarrow p(rule|\mathcal{G}), [direction] S_{i,1}, \\ p(rule | \mathcal{G}) & \text{if rule is } S_i \rightarrow p(rule|\mathcal{G}), [direction] E_{i,1}, \text{ and } E_{i,1} \in \mathcal{E}. \end{cases}$$

where  $rule$  is the rule that is used to parse the current record  $R_i$ ,  $p(rule | \mathcal{G})$  is the probability of  $rule$  used among all grammar rules (in all directions) that have head  $S_i$ ,  $R_{i,1}$  and  $R_{i,2}$  are the split traces based on the direction of the rule,  $direction$ , and the place of the split,  $idx$ , and  $T_i$ ,  $T_{i,1}$  and  $T_{i,2}$  are the most probable parse trees for  $R_i$ ,  $R_{i,1}$  and  $R_{i,2}$  respectively. Using this recursive equation, the algorithm builds the most probable parse trees in a bottom-up fashion.

After getting the parse trees for all records, the algorithm moves on to the second step. In this step, the algorithm updates the selection probabilities associated with the rules. For a rule with head  $V$ , the new probability of getting chosen is simply the total number of times that rule appearing in the Viterbi parse trees divided by the total number of times that  $V$  appears in the parse trees, that is,

$$p(rule_i | \mathcal{G}) = \frac{|rule_i \text{ appearing in parse trees}|}{|V_i \text{ appearing in parse trees}|}$$

where  $rule_i$  is of the form  $V_i \rightarrow p, [direction], \gamma_1, \gamma_2, \dots, \gamma_n, n = 1 \text{ or } 2$ .

After finishing the second step, the algorithm starts a new iteration until convergence. This learning procedure is a fast approximation of expectation-maximization, which approximates the posterior distribution of trees given parameters by the single MAP hypothesis. The output of the algorithm is an updated 2-D pCFG,  $\mathcal{G}$ , and the most probable layout of the interface. For elements that have never been used in the training examples, the acquired layout will not include them in it as there is no information for them in the record. But the acquired grammar may be able to generalize to those elements. For example, if the acquired grammar learns a recursive rule across rows, it will be able to generalize to more rows than the training records have reached.

The complexity of the Viterbi training phase is  $\mathcal{O}(|iter| \times |\mathcal{R}| \times |\mathcal{Rules}_{nt}| \times |\max R_i.length|!)$ , where  $|iter|$  is the number of iterations,  $|\mathcal{R}|$  is the number of records,  $|\mathcal{Rules}_{nt}|$  is the number of rules that reduce to non-terminal symbols,  $|\max R_i.length|$  is the length of the longest record. In practice, since the number of rules generated by GSH is small, and we cache previously calculated parse trees in memory, as we will see in the experiment section, all learning tasks are completed within a reasonable amount of time.

steps),  $parse(T_i)$  is considered equal to  $parse(T_j)$  as long as the parse structures of the shared elements are the same.

---

**Algorithm 5.2:** *GSH* constructs a set of grammar rules,  $\mathcal{Rules}$ , a set of terminal symbols  $\mathcal{E}$ , and from the training records,  $\mathcal{R}$ .

---

**Input:** Record Set  $\mathcal{R}$ , Terminal Symbol Set  $\mathcal{E}$ , Grammar Rule Set  $\mathcal{Rules}$

```

1 if is-empty-set( $\mathcal{Rules}$ ) then
2   |  $\mathcal{Rules} := \text{generate-terminal-grammar-rules}(\mathcal{E});$ 
3 end
4 while not-all-records-are-parsable( $\mathcal{R}$ ,  $\mathcal{Rules}$ ) do
5   | if has-recursive-structure( $\mathcal{R}$ ) then
6     |  $rule := \text{generate-recursive-rule}(\mathcal{R});$ 
7   | else
8     |  $rule := \text{generate-most-frequent-non-added-rule}(\mathcal{R});$ 
9   | end
10  |  $\mathcal{Rules} := \mathcal{Rules} + rule;$ 
11  |  $\mathcal{R} := \text{update-record-set-with-rule}(\mathcal{R}, rule, \mathcal{Rules});$  // First, update the
    | record set using  $rule$ ; second, update the record set using
    | all acquired  $\mathcal{Rules}$ 
12 end
13  $\mathcal{Rules} = \text{initialize-probabilities}(\mathcal{Rules});$ 
14 return  $\mathcal{Rules}$ 

```

---

### Greedy Structure Hypothesizer (GSH)

As with the standard Viterbi training algorithm, the output of the algorithm converges toward only a local optimum. It often requires more iteration to converge if the starting point is not good. Moreover, since the complexity of the Viterbi training phase increases as the number of grammar rules increases, we designed a greedy structure hypothesizer (GSH) that greedily adds grammar rules for frequently observed “adjacent” symbol pairs. Note that instead of building a structure learner from scratch, we extend the greedy structure hypothesizer described in Chapter 3 to accommodate the 2-D space. Extending other learning mechanisms is also possible. To formally define adjacency, let’s first define two terms, *temporally adjacent*, and *horizontally (vertically) adjacent*.

**Definition 1.** Two tuples,  $T_{i1}$  and  $T_{i2}$ , are temporally adjacent, iff the two tuples’ time intervals overlap, i.e.

$$[T_{i1}.timestamp_{start}, T_{i1}.timestamp_{end}) \cap [T_{i2}.timestamp_{start}, T_{i2}.timestamp_{end}) \neq \emptyset$$

**Definition 2.** Two tuples,  $T_{i1}$  and  $T_{i2}$ , are horizontally adjacent, iff the spaces taken up by the two tuples are horizontally next to each other, and form a rectangle, i.e.

$$\begin{aligned}
T_{i1}.x_{right} &= T_{i2}.x_{left} \text{ or } T_{i2}.x_{right} = T_{i1}.x_{left} \\
T_{i1}.y_{up} &= T_{i2}.y_{up} \\
T_{i1}.y_{bottom} &= T_{i2}.y_{bottom}
\end{aligned}$$

**Definition 3.** Two tuples,  $T_{i1}$  and  $T_{i2}$ , are vertically adjacent, iff the spaces took up by the two tuples are vertically next to each other, and form a rectangle, i.e.

$$\begin{aligned} T_{i1}.y_{bottom} &= T_{i2}.y_{up} \text{ or } T_{i2}.y_{bottom} = T_{i1}.y_{up} \\ T_{i1}.x_{left} &= T_{i2}.x_{right} \\ T_{i1}.x_{right} &= T_{i2}.x_{left} \end{aligned}$$

Now, we can define what is a 2D-mergeable pair.

**Definition 4.** Two tuples,  $T_{i1}$  and  $T_{i2}$ , are 2D-mergeable, iff the two tuples are both temporally adjacent and horizontally (vertically) adjacent.

The structure hypothesizer learns grammar rules in a bottom-up fashion. The pseudo code of the structure hypothesizer is shown in Algorithm 5.2. The grammar rule set,  $\mathcal{Rules}$ , is initialized to contain rules associated with terminal symbols, when GSH is called for the first time. Then the algorithm detects whether there are recursive structures embedded in the records (e.g., **Row**, **Row**, ...**Row**), and learns a recursive rule for it if finds one (e.g., **Table**  $\rightarrow$  0.7,  $[v]$  **Table Row**). If the algorithm fails to find recursive structures, it starts to search for the 2D-mergeable pair (e.g.,  $\langle \mathbf{Equation}, \mathbf{Ski} \rangle$ ) that appears in the record set most frequently, and constructs a grammar rule (e.g., **Row**  $\rightarrow$  1.0,  $[h]$  **Equation Ski**) for that 2D-mergeable pair. The direction field value is set based on whether the 2D-mergeable pairs are horizontally or vertically adjacent. If the Viterbi training phase cannot find a layout based on these rules, less frequent pairs are added later. When there is no more pair that is 2D-mergeable, it is possible that some training record has not been fully parsed, since some symbol pairs that are horizontally (vertically) ordered may not form rectangles. The grammar rules constructed for these symbol pairs in this case will use the extended direction values (e.g., ph, pv). After getting the new rule, the system updates the current record set with this rule by replacing the pairs in the records with the head of the rule.

After learning the grammar rules, the GSH assigns probabilities associated with these grammar rules. For each rule with head  $V$ ,  $p$  is assigned to 1 divided by the number of rule that have  $V$  as the head. In order to break the symmetry among all rules, the algorithm adds a small random number to each probability and normalizes the values again. This structure learning algorithm provides a redundant set of grammar rules to the Viterbi algorithm.

## 5.2 Experimental Study in Synthetic Domains

In order to evaluate whether the proposed layout learner is able to acquire the correct layout, we carry out three experiments in progressively more realistic settings. All experiments are performed on a machine with a 3.06 GHz CPU and 4 GB Memory. The time the layout learner takes to learn ranges from less than 1 millisecond to 442 milliseconds per training record.

### 5.2.1 Methods

In this section, we use the 1-D layout learner (i.e., 1-D pCFG learner) as a baseline, and compare it with the proposed 2-D layout learner. In order to make the training records learnable by the 1-D layout learner, we first transform each training record into a row-ordered 1-D record, and then call the 1-D layout learner on the transformed records.

### 5.2.2 Measurements

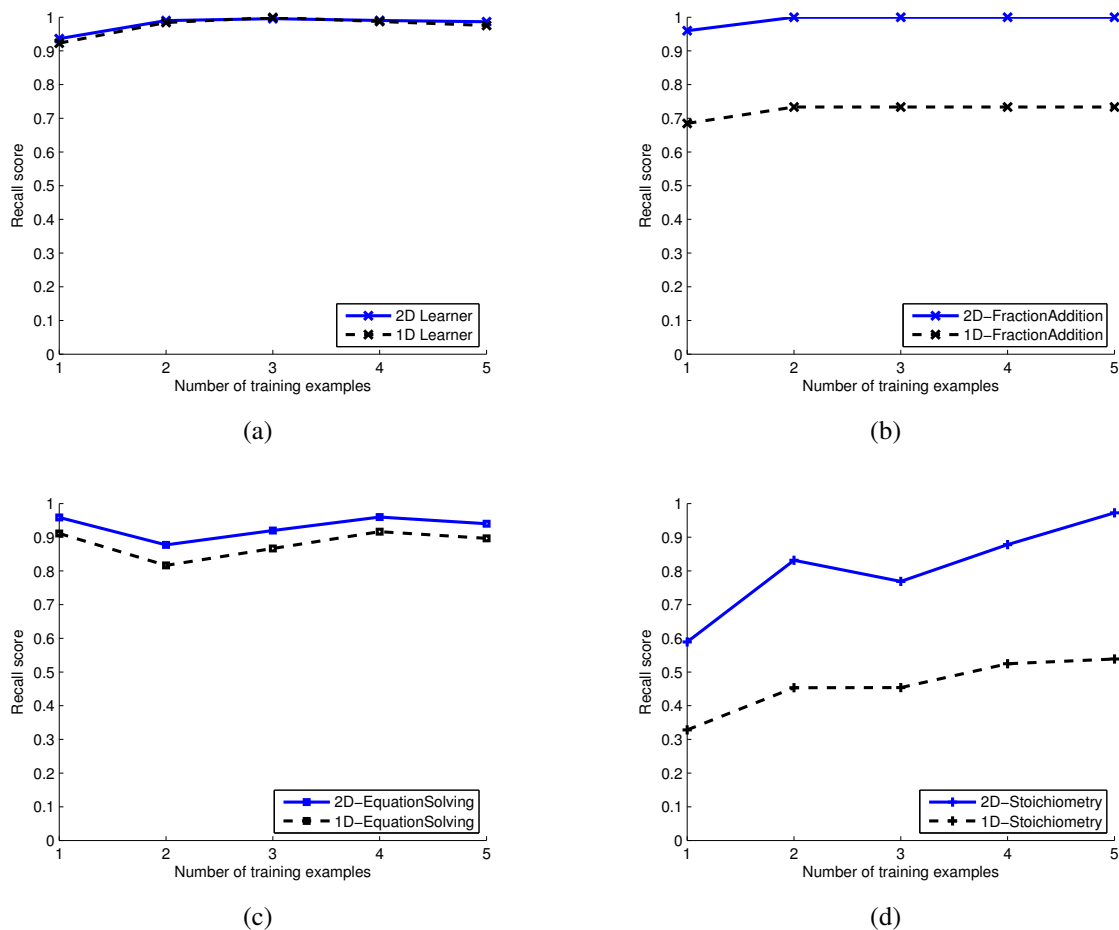
We evaluate the quality of the learned parses with the most widely-used evaluation measurements [Harrison et al., 1991]: (1) the *Crossing Parentheses* score, which is the number of times that the learned parse has a structure such as  $((A\ B)\ C)$  and the oracle parse has one or more structures such as  $(A\ (B\ C))$  which “cross” with the learned parse structure; (2) the *Recall* score, which is the number of parenthesis pairs in the intersection of the learned and oracle parses ( $L$  intersection  $O$ ) divided by the number of parenthesis pairs in the oracle parse  $O$ , i.e.,  $(L \text{ intersection } O) / O$ . To better understand the crossing parentheses score, we further normalize it so that it ranges from zero to one.

### 5.2.3 Experimental Results

In the first experiment, we randomly generate 50 oracle two-dimensional grammars. For each oracle grammar, we randomly generate a sequence of 15 training layouts<sup>6</sup> based on the oracle grammar. Each randomly-generated oracle grammar forms an and-or tree, where each non-terminal symbol can be decomposed by either a non-recursive or a recursive rule. Each grammar has 50 non-terminal symbols in it. For each layout, we give the layout learners a fixed number of training records. The two layout learners (i.e., the 1-D layout learner with row-based transformation and the 2-D layout learner) are trained on the 15 layouts sequentially using a transfer learning mechanism developed for the layout learner. The transfer learning mechanism is not described here due to the limited space. Then, we generate another layout with a fixed number of testing records by the oracle grammar, and test whether the grammars acquired by the two layout learners are able to correctly parse the testing records.

Figure 5.2(a) presents the recall scores of the layout learners averaged over 50 grammars. Both learners perform surprisingly well. They are able to achieve close to one recall scores, and close to zero crossing parentheses scores with only five training examples per layout. To better understand the result, we take a close look at the data. Since the oracle grammar is randomly generated, the probability of getting a hard-to-learn grammar is very low. In fact, many of the training records are traces of single rows or columns, which makes learning easy. Hence, to challenge the layout learner more, we carried out a second experiment.

<sup>6</sup>Some layouts may be the same.



**Figure 5.2:** Recall scores in a) randomly-generated domains, and three synthetic domains, b) fraction addition, c) equation solving, d) stoichiometry.

## 5.3 Experimental Study in Three Synthetic Domains

In addition, we examine three tutoring systems used by human students: fraction addition, equation solving, and stoichiometry.

### 5.3.1 Methods

We manually construct an oracle grammar that is able to parse these three domains. Moreover, the oracle grammar can further generate variants of the existing user interfaces. For example, instead of adding two fractions together, the oracle grammar can generate interfaces that can be used to add three fractions. We carry out the same training process based on this manually-constructed oracle grammar, and test the quality of the acquired grammar in three domain variants.

The interface of the fraction addition tutor has four rows, where the upper two rows are filled with the problem (e.g.,  $\frac{3}{5} + \frac{2}{3}$ ), and the lower two rows are empty cells for the human students to fill in. The equation solving tutor’s interface is shown in Figure 2.1. The interface of the stoichiometry domain contains four tables of different sizes. The four tables are used to provide given values, to perform conversion, to self-explain for the current step, and to compute intermediate results. All tables are of column-based orders.

### 5.3.2 Measurements

The crossing parentheses and recall are used to evaluate the quality of the acquired layouts.

### 5.3.3 Experimental Results

Figure 5.2(b), 5.2(c), 5.2(d) show the recall scores of the three domains averaged over 50 runs. Both learners achieve better performance with more training examples. We also see that the 2-D layout learner has significantly ( $p < 0.0001$ ) higher recall scores than the 1-D layout learner in all three domains. Both fraction addition and stoichiometry contain tables/subtables of column-based orders. The row-based transformation of the 1-D layout learner removes the column information, and thus hurts the learning performance.

The crossing parentheses scores for both learners are always close to zero across three domains, which indicates the acquired grammar does not generate bad “crosses” often. But since a “cross” occurs only when the parents in the learned parse and in the oracle parse match, thus the crossing parentheses score is meaningful only when the recall scores are high. The low crossing parentheses score for the 1-D layout learner may be partially due to its relatively low recall.

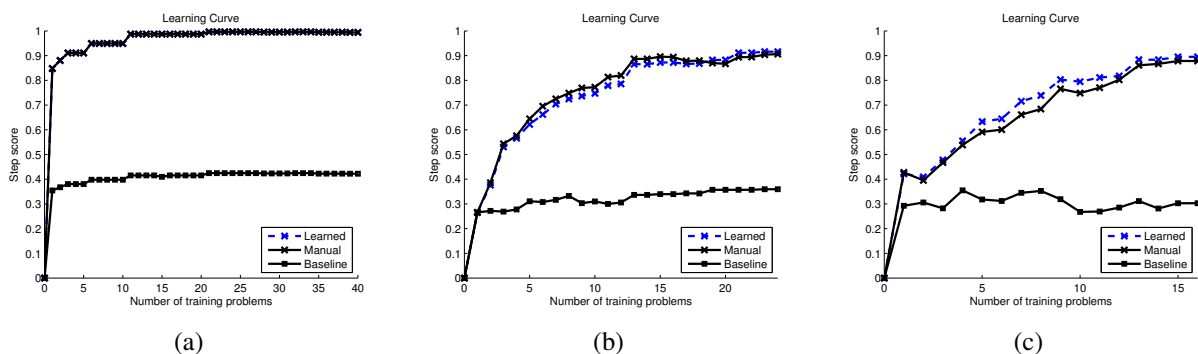
## 5.4 Experimental Study within SimStudent

In order to understand how display representation learning affects an agents learning effectiveness, the last experiment that we carry out is within SimStudent.

### 5.4.1 Methods

We use the actual tutor interfaces in three tutoring domains. The 2-D layout learner is first trained on no more than five problems used to tutor human students, and sends its output to SimStudent. An automatic tutor (also used by human students) then teaches the SimStudent with the constructed/acquired layouts with one set of problems, and tests SimStudents’ performance on another set of problems. Both the training and testing problems are problems used by human students. In each domain, SimStudent is trained on 12 problem sequences. Three SimStudents





**Figure 5.3:** Learning curves of three SimStudents in three domains, a) fraction addition, b) equation solving, c) stoichiometry.

are compared in the experiment. One SimStudent (*manual*) is given the manually-constructed layout, one SimStudent (*learned*) is given the acquired layout, and one SimStudent (*baseline*) is given a row-based layout<sup>7</sup>.

## 5.4.2 Measurements

To measure learning gain, we calculate step scores of the testing problems after SimStudent is tutored on each training problems, and report the average step score over all testing problem steps for each curriculum.

## 5.4.3 Experimental Results

Figure 5.3 shows the learning curves of the three SimStudents across three domains. In all three cases, the SimStudent with a row-based layout (*baseline*) performs significantly ( $p < 0.0001$ ) worse than the other two SimStudents. This shows the importance of the layout in achieving effective learning. Both the SimStudent with the manually-constructed layout (*manual*) and the SimStudent with the learned layout (*learn*) perform well across three domains. There is no significant difference between the two SimStudents, which suggests that the acquired layouts are as good as the manually constructed layouts.

## 5.5 Discussion

The main focus of this chapter is to model how human learns to perceive two-dimensional displays through a 2-D grammar induction technique. One closely related research area that also

<sup>7</sup>A fully flat layout performs so badly that SimStudent cannot finish learning.

uses two-dimensional pCFGs is learning to recognize equations (e.g., [Chou, 1989](#), [Vanlehn and Ball, 1987](#)) or images (e.g., [Siskind et al., 2007](#)). Algorithms in this direction often assume the structure of the grammar is given, and use a two-dimensional parsing algorithm to find the most likely parse of the observed image. Our system differs from their approaches in that we model the acquisition of the grammar structure, and apply the technique to another domain, learning to perceive user interface.

Research on extracting structured data on the web (e.g., [Arasu and Garcia-Molina, 2003](#), [Cafarella et al., 2008](#), [Crescenzi et al., 2001](#)) shares a clear resemblance with our work, as it also concerns on understanding structures embedded in a two-dimensional space. It differs from our work in that webpages have an observable hierarchical structure in the form of their HTML parse trees, whereas we only observe the 2-D visual displays, which have no such structural information.

Our work in this chapter continues the effort on modeling two-dimensional data in a new domain, where structural information is not provided, but temporal information is available for understanding the layout of a two-dimensional display. We show that the proposed layout learner is able to acquire the high-quality layouts with a small number of training examples. We further integrate the proposed approach into SimStudent, and demonstrate the intelligent agent with the acquired layouts is able to perform equally well comparing with the agent given manually constructed layouts.

# Chapter 6

## Learning Feature Predicates

Having demonstrated that with representation learning, the knowledge engineering effort needed for the perceptual learner and the operator sequence learner is reduced. In this chapter, we move to the discussion on how to reduce the knowledge engineering effort of the feature test learner. We present both a new method for discovering perceptual feature predicates in an unsupervised way and an evaluation of this method in the context of complex skill learning. This method creates feature predicates from non-terminals in the parse tree and from relationships between non-terminals expressed in the grammar rules. We provide these automatically generated feature predicates as prior knowledge to the skill learning component of *SimStudent* [Matsuda et al., 2009].

More specifically, we automatically generate, from the acquired representation, a set of predicates that can be used by the inductive logic programming (ILP) component that learns when to apply a skill. It is important and interesting that this integration of an unsupervised representation learner and a supervised skill learner makes it possible, for the first time, for a computer to learn a complex skill without domain-specific feature or representation engineering. Prior skill learning efforts have always required such engineering. We evaluate the quality of the automatically generated feature predicates in the algebra equation solving domain, and report the results in the experiment section.

### 6.1 Generating Feature Predicates from the Learned Grammar

Having removed the dependency on domain-specific operator functions, we would like to further reduce the knowledge engineering required by eliminating *SimStudent*'s dependency on manually-constructed feature predicates. As implied by its name, the representation learner acquires information that reveals essential features of the problem. It is natural to think that these acquired representations can be used in describing desired situations to fire a production rule. In this work, we automatically generate, from the acquired grammar, a set of predicates that can

be used by the inductive logic programming (ILP) component. These automatically generated feature predicates can then replace manually constructed feature predicates.

Hence, we make use of the domain-specific information in the grammar acquired by the representation learner to automatically generate a set of feature predicates. There are two main categories of the automatically generated feature predicates: topological feature predicates, and nonterminal symbol feature predicates. A third category, parse tree relation feature predicates, considers a combination of the information used in the first two. All these types of predicates are applicable to a general pCFG and the parse trees it generates. The truthfulness of the feature predicates is decided by the most probable parse tree of the problem.

### 6.1.1 Topological Feature Predicates

Topological feature predicates evaluate whether a node with the value of its first arguments exists at some location in the parse tree generated from the second argument (e.g., (*is-left-child-of* -3 -3x)). There are four generic topological feature predicates: (*is-descendent-of* ?val0 ?val1), (*is-nth-descendent-of* ?val0 ?val1), (*is-tree-level-m-descendent-of* ?val0 ?val1) and (*is-nth-tree-level-m-descendent-of* ?val0 ?val1). These four generic feature predicates are used to generate a wide variety of useful topological constraints based on different  $n$  and  $m$  values. An automatically-generated predicate is created for each  $m$  between 0 and  $M-1$ , where  $M$  is the maximum number of nonterminal symbols on the right side of the grammar rules, and for each  $n$  between 0 and  $N-1$ , where  $N$  is the maximum height of the parse trees encountered.

The level specificity in the desired location varies from the most general topological predicate, (*is-descendent-of* ?val0 ?val1), to the most specific (*is-nth-tree-level-m-descendent-of* ?val0 ?val1). (*is-descendent-of* ?val0 ?val1) determines whether ?val0 exists anywhere in the subtree rooted at ?val1. For example, since 3 is a grandchild of -3x in the parse tree shown in the left side of Figure 3.1, (*is-descendent-of* 3 -3x) is true. The next two topological feature predicates each incorporate one of the two pieces of information available about the location of a node: the depth of the node in the parse tree and in which subtree its located when the child nodes are ordered. (*is-nth-descendent-of* ?val0 ?val1) is slightly more specific than (*is-descendent-of* ?val0 ?val1). It tests whether ?val0 exists anywhere in the subtree rooted at the  $n^{th}$  child of ?val1. In the correct parse tree of -3x, 3 appears in the left subtree of -3x, therefore, (*is-0th-descendent-of* 3 -3x) is true. (*is-tree-level-m-descendent-of* ?val0 ?val1) represents a similar level of specificity to (*is-nth-descendent-of* ?val0 ?val1), in that it incorporates one of the two pieces of information available. It tests whether ?val0 appears at the  $m^{th}$  level in the subtree rooted at ?val1. For instance, 3 appears at level two of the parse tree so (*is-tree-level-2-descendent-of* 3 -3x) is true. The last topological feature predicate (*is-nth-tree-level-m-descendent-of* ?val0 ?val1) considers both the tree level  $m$  and the descendent index  $n$ . It defines whether ?val0 exists  $m-1$ <sup>1</sup> levels down in the subtree rooted at the  $n^{th}$  child of ?val1. If  $m=1, n=0$ , (*is-nth-tree-level-m-descendent-of* ?val0 ?val1) is equivalent to (*is-left-child-of* ?val0 ?val1).

<sup>1</sup>We are considering nodes  $m-1$  levels down in the tree rooted at the  $n^{th}$  child, rather than  $m$ , because the  $n^{th}$  child is already 1 level down in the tree rooted at ?val1.

### 6.1.2 Nonterminal Symbol Feature Predicates

Nonterminal symbol feature predicates are defined based on the nonterminal symbols used in the grammar rules. For example, *-3* is associated with the nonterminal symbol *SignedNumber* based in the grammar shown in Table 3.1. There are three generic nonterminal symbol feature predicates: *(is-symbol-x ?val0 ?val1)*, *(has-symbol-x ?val0 ?val1)*, and *(has-multiple-symbol-x ?val0 ?val1)* where *x* can be instantiated to any nonterminal symbols in the grammar.

*(is-symbol-x ?val0 ?val1)* describes whether *?val0* is associated with symbol *x* in the parse tree of *?val1*. For instance, *(is-symbol-SignedNumber -3 -3x)* tests whether *-3* is associated with *SignedNumber* in the parse tree of *-3x*. *(has-symbol-x ?val0 ?val1)* tests whether any node in the subtree of *?val0* is associated with symbol *x* in the parse tree of *?val1*. Although *-3* is **not** associated with symbol *Number*, it has a child, *3*, that **is** associated with symbol *Number*. In this case, *(is-symbol-Number -3 -3x)* is false, but *(has-symbol-Number -3 -3x)* is true. The last symbol feature predicate *(has-multiple-symbol-x ?val0 ?val1)* operates similarly to *(has-symbol-x ?val0 ?val1)*, but examines whether there are multiple **separate** nodes in the subtree of *?val0* which are associated with the symbol *x*. For the purposes of this predicate, two nodes *A* and *B* in a parse tree are separate iff *A* is not in *B*'s subtree and *B* is not in *A*'s subtree. In math and logic, an exact number is often less significant than whether a number falls into the category of zero, one, or many/infinite. The *(has-multiple-symbol-x)* predicate thus covers the category of many/infinite, without the need to create individual predicates for specific numbers of nodes which are associated with some symbol. *(has-multiple-symbol-Number 4-3 x+(4-3))* would return true because *4-3* has 2 nodes, *4* and *3*, each of which is associated with the symbol *Number* and neither is in the other's subtree.

### 6.1.3 Parse Tree Relation Feature Predicates

Topological feature predicates examine the position of a particular input in the overall parse tree and symbol feature predicates examine the symbol associated with a particular input. The third class of feature predicates, parse tree relation predicates, examine **both** the positions of nodes in the tree **and** their associated symbols. These allow SimStudent to examine the surrounding nodes in the parse tree and determine if they have a particular symbol from the grammar associated with them.

For the algebra study, three such predicates were used which represent examining the nearest nodes in the parse which are not in the input's subtree: *(parent-is-symbol-x ?val0 ?val1)*, *(sibling-is-symbol-x ?val0 ?val1)*, and *(uncle-is-symbol-x ?val0 ?val1)* (or *aunt*). As their names imply, these predicates examine whether a parent/sibling/uncle(aunt) node of the input is associated with the symbol *x*, where *x* could be any nonterminal symbol in the grammar. As an example, referring again to the left side of Figure 3.1, consider the predicate *(sibling-is-symbol-MinusSign 3 -3x)*. This would return true because in the parse tree for *-3x*, the node representing the number *3* does have a sibling whose associated symbol is minus sign. In ongoing work, these types of predicates have been generalized to encompass arbitrary relations between nodes in the tree

in much the same way that *(is-child-of ?val0 ?val1)* has been generalized to *(is-tree-level-1-descendent-of ?val0 ?val1)*. An arbitrary relationship representing the relative position of any two nodes in a parse tree can be described by the predicate *(i-j-relation-is-symbol-x ?val0 ?val1)*. This represents examining whether the nodes reached by moving up  $i$  times in the tree, then down  $j$  times are associated with the symbol  $x$ . Using this notation, *(1-1-relation-is-symbol-x ?val ?val1)* is equivalent to *(sibling-is-symbol-x ?val0 ?val1)*.

## 6.2 Experimental Study on Automatically Generated Feature Predicates

In order to evaluate whether the extended SimStudent is able to acquire correct knowledge with automatically generated feature predicates, we carry out an experiment in equation solving.

### 6.2.1 Methods

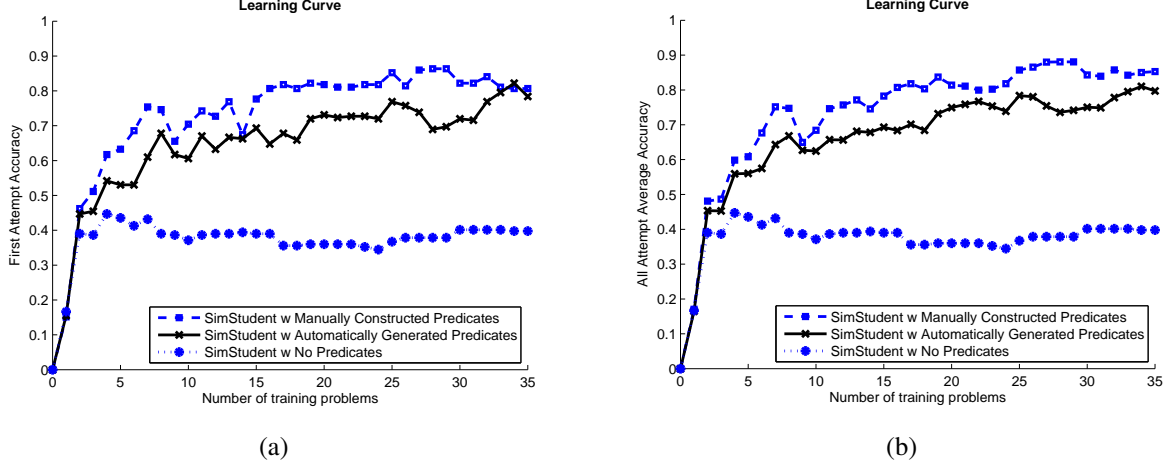
The representation learner was first trained on a sequence of feature learning tasks (i.e., what is a signed number, what is a term, and what is an expression). Then, SimStudent was tutored by an automatic tutor, CTAT [Alevan et al., 2009], which was used by 71 human students in a classroom study, to solve basic algebra problems. All of the training and testing problems were extracted from the same classroom study. There were four sets of training problems. Each set has 35 training problems. The testing problem set contains 11 problems. In this way, we have provided Simstudent with the same information and training as would be provided to human students.

We compared three SimStudents: one SimStudent given the manually constructed feature predicates known to be useful in solving algebra problems, one SimStudent given the automatically generated feature predicates, and one SimStudent given no feature predicates.

### 6.2.2 Measurements

We evaluated the effectiveness of SimStudent in two aspects: the amount of knowledge engineering needed, and the speed of learning. To assess the knowledge engineering effort required, we counted the number of lines of Java code a developer needed to write for each feature predicate, and reported the total number of lines developed for all feature predicates used in the acquired rules.

To measure learning gain, we calculate a first attempt accuracy and an step score for each step in the testing problem, and report the average first attempt accuracy and step score of all testing problems.



**Figure 6.1:** Learning curves of three SimStudents in equation solving measured by, a) first attempt accuracy, b) all attempt accuracy.

### 6.2.3 Experimental Results

Since there is no manual encoding of domain knowledge needed for the automatically generated feature predicates, the number of lines of domain-specific code needed in equation solving is 0. On the other hand, the manually constructed feature predicates required 2093 lines of Java code, which is also a measure of the amount of knowledge engineering saved by automatic feature predicate generation.

The second study we carried out focused on evaluation of learning speed. The average learning curves for the three SimStudents are shown in Figures 6.1(a) and 6.1(b). As we can see, there is a huge gap between the SimStudents with and without manually constructed feature predicates (i.e., the two blue lines). The goal of our algorithm is to fill in the gap without requiring extra knowledge engineering.

As shown in the figures, the SimStudent with automatically generated feature predicates has a slower learning curve than the SimStudent with manually constructed feature predicates. It does, however, gradually catch up after being trained on more problems. This is to be expected because while the manually constructed feature predicate directly evaluates information that is known to be applicable in solving the problem, the automatically generated feature predicates evaluate a larger set of information obtained from the parse trees. Much of this information does not turn out to be relevant to solving the problem. It therefore takes more examples for the SimStudent with automatically generated feature predicates to learn to solve the problems because it must first determine which of the automatically generated predicates are relevant. As the results show, after being trained on 35 problems, the SimStudent with automatically generated feature predicates achieved comparable performance to that with manually constructed feature predicates. This is the case for both measurements (i.e., 0.77 vs. 0.75 for first attempt accuracy, 0.83 vs. 0.79 for all attempt accuracy). Taken together, we conclude that with automatic feature predicate generation,



we are able to obtain nearly comparable performance while significantly reducing the amount of knowledge engineering effort needed.

## 6.3 Experimental Study on Transferability to Harder Problems

Having evaluated the effectiveness of the proposed approach on problems at the same level of difficulty, we further evaluate the transferability of the proposed approach to harder problems, by training SimStudent on sequences of problems of increasing difficulty. The main claim of the work is that by integrating representation learning into skill learning, we are able to develop intelligent agents that learn to solve both easy and hard problems, which (1) require less knowledge engineering effort, and (2) maintain equally good performance, compared with human-engineered intelligent agents. Problem representations are used to learn skill knowledge for simpler problems. This skill knowledge is then automatically built upon to develop a SimStudent capable of representing and solving much more complicated problems. This process is performed without manually constructed extensions to prior domain knowledge as required by the original SimStudent. The results further indicate that while the original SimStudent given human-engineered prior domain knowledge performed better than the extended SimStudent without prior domain knowledge on easier problems, it performs worse on the harder problems, due to the fact that the human-engineered prior domain knowledge was built for easier problems, and is not easily extensible to harder ones.

To determine whether the skill knowledge of a SimStudent incorporating deep features, obtained from simple problems, can be transferred to problems of greater complexity, we carry out an experiment in algebra equation solving. Original and extended SimStudents are trained on sequences of increasingly difficult algebra problems and their performance is compared.

### 6.3.1 Methods

Two extended versions of SimStudent are tested in this study, one with *only* an extension to the memory element hierarchy and one with automatically generated feature predicates as well. Both used only a set of domain-independent operators. To construct the extended SimStudents, a representation learner is trained on a series of feature learning tasks (i.e. what is a signed number, what is a term, what is an expression, what is a complex expression). The learned grammar for algebra problems obtained from this is then incorporated into SimStudent, as described in the Chapter 4 on SimStudent with integrated representation. The two extensions are compared with an original SimStudent, which is provided a set of domain-specific operator functions, and feature predicates known to be useful in algebra equations solving and an original SimStudent with the domain-independent operator functions and a set of domain-independent feature predicates.



All versions are tutored using an automatic tutor, CTAT [Aleven et al., 2009], which is used by 71 human students in a classroom study. Four training sets, each consisting of 47 problems are constructed for use in teaching the SimStudents. Each training set consists of problems in four difficulty categories and ordered in increasing difficulty where the fourth category represents a much more significant increase in problem difficulty. A separate test set consisting of 19 problems, also of varying difficulty and with a distribution weighted toward more difficult problems, is constructed for use in evaluation of performance. Problems for both the training and test sets are likewise obtained from actual classroom studies.

### 6.3.2 Measurements

We assess the accuracy of the SimStudents' skill acquisition by measuring each SimStudent's *first attempt accuracy* and *step score* for each step in the test problems. First attempt accuracy is the percentage of the time, which the first action proposed, by SimStudent is correct. Since, for a given problem step, there may be multiple correct courses of action and SimStudent may propose more than one action at any given step, a more nuanced measure of accuracy is required to evaluate SimStudent's overall mastery of the skills represented in the problem domain. Step score is the same measurement we used in previous studies, which is number of correct steps proposed by SimStudent, divided by the number of possible correct steps plus the number of proposed steps, which were incorrect.

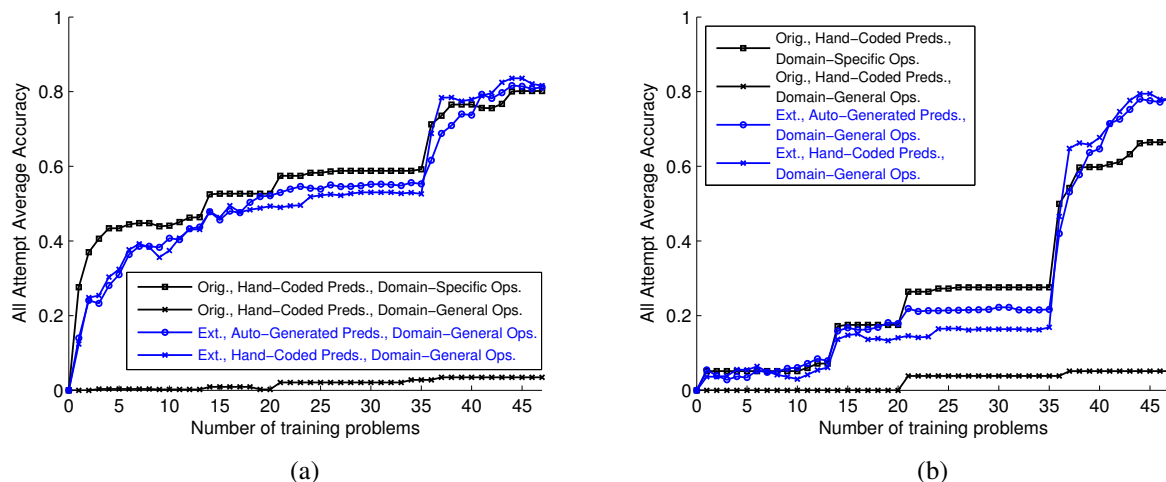
Last, to measure the amount of domain-specific prior knowledge encoding required for SimStudent, we count the number of lines of Java code used in the implementation.<sup>2</sup> There are two locations where this information is used, the operators and the feature predicates. Each are measured separately and reported as such.

### 6.3.3 Experimental Results

#### Knowledge Engineering Effort

We compare the lines of code needed to encode the operator functions and feature predicates. The addition of representation learning reduces the effort in coding operator functions from 2287 lines to only 247 lines. (This replicates results from [Li et al., 2012a] but with a larger training set that indicates more difficult problems.) The addition of predicate learning reduces the effort in coding feature predicates from 1981 lines of code to zero lines. We find that this approach completely removes the prior knowledge engineering effort need to author feature predicates. Since one of the important applications of SimStudent is to enable end-users to create intelligent tutoring systems without heavy programming, this reduction of programming effort makes SimStudent a better authoring tool for intelligent tutoring system. Moreover, by requiring less prior knowledge engineering, SimStudent becomes a more complete model of human skill acquisition.

<sup>2</sup>Although the line of code is not the ideal measurement of knowledge engineering effort due to individual differences among agent developers, this still serves as a good indication.



**Figure 6.2:** Learning curves of SimStudents in equation solving measured by a) all test problems, b) hard problems (category 4) only, using all attempt accuracy.

## Speed of Learning

Average learning curves for all attempt accuracy are shown in Figure 6.2(a). Both extended SimStudents perform similarly to the original SimStudent with domain-specific operators and feature predicates. This is the case in both first attempt and all attempt accuracy. The extended SimStudent with representation learning and automatically generated predicates achieves a final all attempt accuracy of 0.81 and first attempt accuracy of 0.83 after all 47 problems, while the extended SimStudent with representation learning and hand-coded domain-specific predicates achieved scores of 0.82 and 0.80 respectively. These are comparable to the original SimStudent with both domain-specific operators and feature predicates whose accuracy is 0.80 and 0.82. We see that the original SimStudent when given only the weak operators, as supplied to the extensions, and domain-independent feature predicates fails to produce any significant level of skill mastery.

Extended SimStudents, incorporating deep features, do learn slightly slower than the original SimStudent, but catch up readily with more training examples. This is most noticeable at the beginning of training when the system has limited information from a small number of training examples. The hand-coded feature predicates and operators built for the original SimStudent were designed specifically to handle information relevant to solving algebra problems of the types on which it is being trained. The deep features and associated automatically generated feature predicates employed in the extended SimStudent include a wider range of information obtained from parse trees and the topology of parse trees in general. Not all of this information is applicable to a given problem solving step in algebra. It may, however, be relevant to some algebra skills or to domains aside from algebra, such as stoichiometry, which are likewise modeled accurately using context free grammars. Identifying the subset of this broader range of information which is applicable to a given algebra skill requires more examples. The examples help

eliminate those predicates which are not relevant in this domain or for this particular skill.

Also clearly visible from the graphs are the locations at which the SimStudents began to be trained on harder problems at problems 14, 20, and 35. Since the testing set consists of problems of all difficulty levels, the SimStudents perform poorly on the tests at the beginning when they have only been exposed to simple or moderate difficulty problems. As they are trained on harder problems, their performance improves until they acquire a similar level of skill mastery to the original. This transfer of knowledge from simple to harder problems is accomplished without further knowledge engineering work on the part of the system designers. The overall approach of learning representations and incorporating them into SimStudent remains effective as the complexity of the skills within the domain increases, ultimately performing slightly better.

### **Extensibility to Harder Problems**

Through the first three categories of problems, the original SimStudent generally performs better than the extended versions (0.96 vs. 0.90/0.91 on all attempt accuracy), but it is overtaken by the extended SimStudents when trained on the hardest problems (0.66 vs. 0.78/0.78 on all attempt accuracy). The learning curves for only hard problems measured by all attempt accuracy are shown in Figure 6.2(b). There are two reasons for this disparity. First, the hand-coded, domain-specific operators were originally designed for the first three categories of problems. Hence, they are not readily extensible to the harder problems, which cause difficulty during the training process for the original SimStudent. The extended SimStudents, on the other hand, are unaffected by errors in the more complex operators since they use only simple domain-independent operators. Second, as in the case of the equations  $-3x = 6$  and  $-3(x + 4) = 6$ , automatically generated predicates identify abstract similarities in more complex examples. This allows the extended SimStudents to more easily transfer their previously acquired skill knowledge from simple problems to harder ones.

Errors in encoding the prior domain knowledge for use in SimStudent's operators and in tutor feedback mechanisms underline the reduced learning rate on harder problems. First, the hand-coded, domain-specific operators for algebra occasionally fail to correctly handle the input they are given. Consequently, an operator sequence which we would expect to produce a correct next step for a given problem state does not do so. During the testing process this is a relatively minor difficulty since that particular problem step will simply be marked as wrong and the testing will proceed to the next step. During the training phase these errors are much more problematic. Should an operator sequence fail to produce a correct next step in an instance when it would otherwise have been correct were the domain specific operators perfect, erroneous negative feedback would be provided to SimStudent. This incorrect negative feedback can weaken SimStudent's skill knowledge. Second, complex algebra problems can be written in many equivalent forms. As the complexity of the problems increases so does the variety of different forms. The feedback system must correctly identify *all* such equivalent forms in order to provide universally accurate feedback. Though the system may not identify all possible forms as equivalent using algebraic transformations, exact, character-by-character matches will always be identified as correct. The extended SimStudents are unaffected by errors in the more complex operators since they use only

simple domain-independent operators. Extended SimStudents also tend to be hurt much less by errors in matching the form of the input. Input produced by extended SimStudents is very often an exact match since it is constructed from strings directly represented in the parse trees of the problem. The combination of these two factors helps the extended SimStudents edge out the original by the end of the training.

## 6.4 Discussion

The main contribution of this work is to reduce the amount of knowledge engineering required in building an intelligent agent by automatically generating feature predicates. Although there has been considerable work on representation change (e.g., [Muggleton and Buntine, 1988](#), [Martín and Geffner, 2004](#), [Utgoff, 1984](#), [Fawcett, 1996](#)) in machine learning, little has occurred in the context of representation learning. Additionally, research on deep architectures [[Bengio, 2009](#)] and Markov logic networks [[Richardson and Domingos, 2006](#)] shares a clear resemblance with our work, but the tasks on which we work are different. These works are used more often in classification tasks whereas our work focuses on simulating human learning of math and science.

Research on ILP (e.g., [Quinlan, 1990](#), [Raedt and Dehaspe, 1997](#), [Srinivasan, 2004](#)) is also closely related to our work, as SimStudent uses FOIL as its “when” learner. ILP systems acquire logic programs that separate positive examples from negative ones given an encoding of the known background knowledge. Our work differs from these systems in that it automatically generates the encoding based on a learned grammar, and calls an existing ILP algorithm to acquire the “when” part of the production rule.

One open question in this work is that how many feature predicates should be generated. More specifically, what would be the appropriate  $M$  and  $N$  (as described in the above section) for the current learning task. The bigger  $M$  and  $N$  are, the more expressive the set of generated feature predicates are. However, setting  $M$  and  $N$  with large values could lead to an explosion on the total number of feature predicates generated. We believe that by setting  $M$  and  $N$  to be larger than the needed values would not further increase the learning effectiveness. Instead, more training examples might be needed due to the larger search space. One possible future study is to systematically search for the appropriate  $M$  and  $N$  values, or to add a feature selection phase before calling the feature test learner.

In this chapter, the proposed feature generation mechanism is integrated into an intelligent agent, SimStudent. In the experimental study, we show that the SimStudent with automatically generated feature predicates is able to achieve comparable performance without requiring any manually-constructed feature predicates as input.

## Chapter 7

# Integrating Representation Learning with External World Knowledge

Till now, we have shown that by integrating representation learning with skill learning, SimStudent can achieve comparable or better learning speed with largely reduced knowledge engineering effort. However, most of these domains are well-defined problem-solving domains, where little real-world background knowledge is needed.

In this work, we explore the generality of the proposed approach in another domain, article selection in English, where no complex problem solving is needed, but where complex perceptual knowledge and large amounts of background knowledge are needed. Specifically, representation learning in this world-knowledge rich domain requires the ability to parse sentences and the extensive understanding of semantics of English words and phrases. There has been a long-standing interest in the natural language processing community to learn how to parse sentences correctly. Therefore, we apply one of the widely-used linguistic tools, *the Stanford parser* [Klein and Manning, 2003], to the sentences in the problems, and integrate the perceptual representations (parse trees) of the sentences into SimStudent.

Additionally, although linguistic theory has long assumed that knowledge of language is characterized by a categorical system of grammar, many previous studies have shown that language users reliably and systematically make probabilistic syntactic choices [Hay and Bresnan, 2006]. To accommodate to this probabilistic character, we further extend SimStudent to accept less-accurate production rule conditions, and learn to prioritize learned rules using historical accuracy statistics. Experimental results show that the extended SimStudent can successfully learn how to select the correct article given a reasonable number (i.e., 60) of problems.

### 7.1 English Article System

Before describing our simulated student, let us first take a look at the domain. The learning task is acquiring the English article system. There are more than 40 grammar rules to decide which

**Table 7.1:** Grammar rules in selecting appropriate articles.

Rule Name	Content	Article
generic-singular	Use “a/an” when a singular count noun is indefinite.	a/an
generic-noncount	Use “no article” with a noncount noun that is indefinite.	no article
generic-plural	Use “no article” with a plural noun that is indefinite.	no article
number-letter	Use “a/an” for single letters and numbers.	a/an
already-mentioned	Use “the” when the noun has already been mentioned.	the
same	Use “the” with the word “same”.	the

article to choose.

In the current study, we took the problems from a previous study on human students [Wylie et al., 2010]. There are six mostly-used grammar rules taught in the study, as shown in Table 7.1. Each problem consists of one or two sentences and an empty space to be filled with an article that best completes the sentence (e.g., *Clocks measure \_\_\_ time.*). There are three choices available, *a/an*, *the* and *no article*. In the clock example, since time is uncountable, *no article* should be selected based on the rule “generic-noncount”.

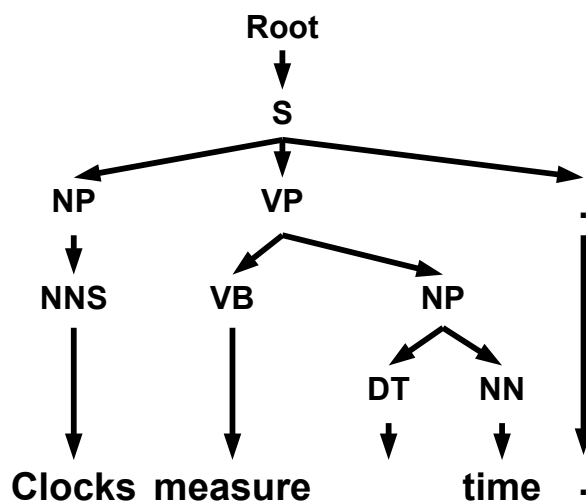
There are priorities among these six grammar rules. For example, if given the problem *He drives \_\_\_ same car as he did last year*, both the condition of the rule “generic-singular” and the condition of the rule “same” are satisfied, but since the rule “same” has a higher priority, article *the* should be selected.

## 7.2 Integrating Representation Learning with External World Knowledge

In spite of the promising results we have shown, the domains we have tested so far are mainly well-defined domains (e.g., fraction addition, equation solving, stoichiometry), where the perceptual representation can be captured by a pCFG, and learning such representation does not require large amounts of external world background knowledge. Article selection in English is quite different from these domains. To solve this task, it requires complex prior perceptual knowledge as well as large amounts of world knowledge.

Therefore, we use an existing linguistic tool, the Stanford parser, to automatically generate the parse structure of the input sentence for SimStudent. The parse tree for the clock example is shown in Figure 7.1. We give these parse trees to SimStudent as the perceptual hierarchies. Based on these hierarchies, SimStudent learned that the noun that the article is pointing to is the last sibling of the article in the subtree. In the example, the non-terminal node *NP* has two children, hence, the word *time* is the noun that the article is pointing at.

Moreover, SimStudent automatically generated a set of feature predicates based on the parse tree. For example, in the parse tree shown in Figure 7.1, each non-terminal symbol (e.g., *NN*) is



**Figure 7.1:** The parse tree of “Clocks measure \_\_\_ time.” generated by the Stanford parser.

associated with a feature predicate (e.g., *(is-NN ?val0 ?val1)*). Given the parse tree, *(is-NN time Clocks-measure-time)* returns true. Topological based feature predicates such as (e.g., *(is-child-of ?val0 ?val1 ?val2)*) can also be generated, but were not used in article selection.

Lastly, we use Wiktionary<sup>1</sup>, which is a collaborative project for creating a free lexical database in every language, complete with meanings, etymologies, and pronunciations, to generate two feature predicates (i.e., *(is-countable ?val)*, *(is-uncountable ?val)*) that evaluate whether a noun is countable or not. Note that since one word may have multiple senses, it can be both countable and uncountable at the same time.

### 7.3 SimStudent with Probabilistic-Based Conflict Resolution

As mentioned before, although grammar rules are often modeled as a categorical system, previous studies have shown that people systematically make probabilistic choices. To incorporate this feature, we developed two conflict resolution strategies that prioritize rules based on historical accuracy statistics. SimStudent associates each production rule with a utility. When multiple production rules are applicable, the production rule that has the highest utility is applied first.

To implement the conflict resolution strategy, we lowered the accuracy requirement of the preconditions learned by FOIL, so that preconditions that are less accurate are also included in the production rule. This change allows SimStudent to learn more general production rules. Therefore, there are more situations where more than one production rules are applicable. However, some of them may be incorrect.

<sup>1</sup><http://www.wiktionary.org/>



Next, SimStudent computes the utility associated with each production rule based on the correctness of the rule’s application history. We designed two ways of computing the utility. The first approach is developed based on ACT-R’s conflict resolution strategy [Belavkin and Ritter, 2004], where the utility associated with production rule  $i$ ,  $U_i$ , is calculated based on the following equation.

$$U_i = P_i G - C_i,$$

where,  $P_i$  stands for the probability of success of the production rule  $i$ ,  $C_i$  is the average cost of the production rule, and  $G$  is a goal value.

In the above approach,  $P_i$  considers all successful applications are equally important. One interesting question to ask is that whether the importance of the rule application result decays as time passes. Hence, in the second approach, instead of directly computing the probability of success, SimStudent weighs recent successes more than the past ones. Each time a rule is applied correctly, it is given a constant reward,  $R$ , and the utilities of all other rules decay by another constant,  $d$ . In case of an incorrect application, the same constant value,  $R$ , is removed from the utility function. Therefore, the utility of production rule  $i$  at time  $t$ ,  $U_{i,t}$ , is calculated by

$$\begin{aligned} U_{i,t} &= D_{i,t} G - C_i, \\ D_{i,0} &= 0, \\ D_{i,t+1} &= (-1)^{failure} R + d D_{i,t}, \end{aligned}$$

where *failure* is an integer that equals to 1 if the rule application is incorrect, and 0 if correct,  $R$  is the reward/punishment given to the production rule, and  $d$  is the rate of decaying.

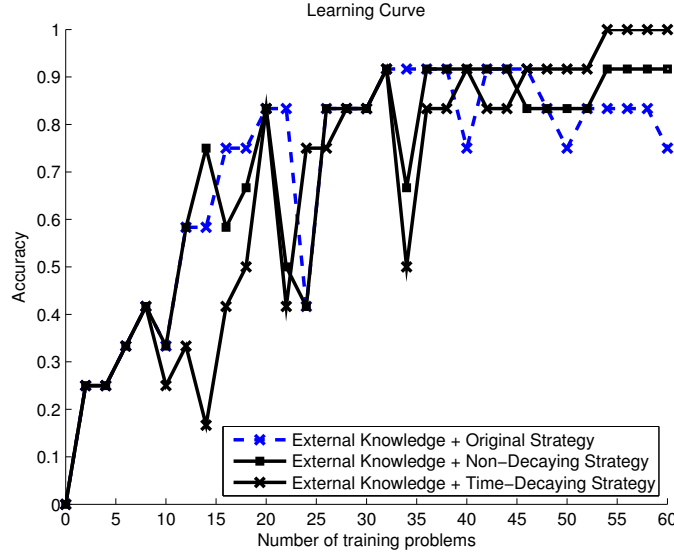
## 7.4 Experimental Study

To evaluate the effectiveness of the proposed approach, we carry out two experiments to test, 1) whether the extended SimStudent can learn the six grammar rules; 2) whether the extended SimStudent can better predict human student behavior than human-generated models.

### 7.4.1 Methods

We use data collected from Wylie et al.’s [2010] recent study on second language learning. The study is conducted at the University of Pittsburgh’s English Language Institute. Students ( $N = 99$ ) are adult English language learners (*meanage* = 27.9, *SD* = 6.6) and participate as part of their regular grammar class. Data collection is completed within one 50-minute class period. Pre- and post-test items are identical in the form of the practice problems students has seen during tutoring without feedback and hints. All of the student behaviors are recorded during the process, and encoded with rules applied to the problems and whether students answers are correct.





**Figure 7.2:** Learning curves of SimStudents in article selection.

SimStudent is taught by an automated tutor that simulates the tutor used by human students, and is trained on the same 60 problems that are provided to human students. The production rules acquired are evaluated by 12 problems given to human students as test problems.

## 7.4.2 Experimental Results

We evaluate four versions of SimStudent, 1) the original SimStudent without external world knowledge and the new conflict resolution strategy<sup>2</sup>, 2) the extended SimStudent with external world knowledge using the original conflict resolution strategy, 3) the extended SimStudent with external world knowledge using the non-decaying conflict resolution strategy, 4) the extended SimStudent with external world knowledge using the time-decaying conflict resolution strategy. In order to rule out the affect of other parameters, we set  $G$  and  $C_i$  to be the same across all production rules, so that the production rule priorities are decided by  $P_i$  and  $D_{i,t}$ . We report the average accuracy of SimStudent's first attempts at each step over 12 test problems.

Since the original SimStudent without external world knowledge considered that all words in the sentence form a flat hierarchy, it fails to learn how to identify the noun that the article is pointing at. In fact, it learns overly general production rules, and could not finish training in a reasonable amount of time. Therefore, we do not report the learning curve of the original SimStudent here, but please keep in mind that, it is much worse than the extended SimStudents.

As we can see in Figure 7.2, all three SimStudents learn reasonably well, reaching accuracies more than 0.75 given 60 problems. The extended SimStudent using the time-decaying conflict

<sup>2</sup>The conflict resolution strategy of the original SimStudent is to fire the the most recently activated non-buggy production rule.

resolution strategy learns fastest among the three SimStudents. It reaches an accuracy of 1.00 given 60 training problems. The extended SimStudent using the non-decaying conflict resolution strategy is slightly worse than the one using the time-decaying strategy, reaching an accuracy of 0.92 with 60 training problems. The extended SimStudent using the original conflict resolution strategy is the worst. This result indicates that by integrating representation learning with external world knowledge, the extended SimStudent is able to successfully learn the six grammar rules. Better conflict resolution strategy can further improve SimStudent’s learning effectiveness. Time-decaying conflict resolution strategy yields a faster learning curve than the non-decaying strategy.

## 7.5 Discussion

The objective of this work is to extend representation learning with external world knowledge, and integrate it into a simulated student. Previous work on article selection (e.g., [Wylie et al., 2010](#)) has shown that learning in this domain contains challenges that cause effective instructional strategies (e.g., self-explanation) in math and science domains to become less effective. In order to better understand the cause of this phenomenon, we take one more step in this direction by constructing a learning agent that models knowledge acquisition.

There have been recent efforts (e.g., [Neves, 1985](#), [Anzai and Simon, 1979](#), [Matsuda et al., 2009](#), [Vanlehn et al., 1994](#)) in developing intelligent agents that model student learning, but most of the existing works have been done in well-defined domains, where little real-world knowledge is needed. There has also been considerable research on learning within agent architectures [[Laird et al., 1986](#), [Anderson, 1993](#), [Taatgen and Lee, 2003](#)]. Unlike those theories, SimStudent puts more emphasis on knowledge-level learning (cf., [Dietterich, 1986](#)) than speedup learning. Moreover, to the best of our knowledge, none of them have focused on integrating representation learning with skill learning as we have done with SimStudent.

To demonstrate the generality of the approach in this chapter in a world-knowledge rich domain, we extend representation learning with external world knowledge, and integrate it into SimStudent. Results show that given a reasonable number (e.g., 60) of training examples, the extended SimStudent successfully learns six frequently used article selection rule.

## Chapter 8

# Using SimStudent to Discover Better Learner Models

As mentioned above, we are not only interested in building a learning agent: we would also like to construct a learning agent that simulates how students acquire knowledge. In this section, we are going to present an approach that automatically discovers learner models using the extended SimStudent. If the discovered model turns out to be a good learner model, we should be able to conclude that the extended SimStudent simulates the real student learning process well.

One common approach to represent a learner model is a set of *knowledge components (KC)* that encoded in intelligent tutors to model how students solve problems. The set of KCs includes the component skills, concepts, or percepts that a student must acquire to be successful on the target tasks. For example, a KC in algebra can be how students should proceed given problems of the form  $Nv=N$  (e.g.,  $-3x = 6$ ). The learner model provides important information to automated tutoring systems in making instructional decisions. Better learner models match with real student learning behavior, that is, changes in performance over time. They are capable of predicting task difficulty and transfer of learning between related problems, and can be used to yield better instruction.

Traditional ways to construct models include structured interviews, think-aloud protocols, rational analysis, and so on. However, these methods are often time-consuming, and require expert input. More importantly, they are highly subjective. Previous studies [Koedinger and Nathan, 2004, Koedinger and McLaughlin, 2010] have shown that human engineering of these models often ignores distinctions in content and learning that have important instructional implications. Other methods such as Learning Factor Analysis (LFA) [Cen et al., 2006] apply an automated search technique to discover learner models. It has been shown that these automated methods are able to find better learner models than human-generated ones. Nevertheless, LFA requires a set of human-provided factors given as input. These factors are potential KCs. LFA carries out the search process only within the space of such factors. If a better model exists but requires unknown factors, LFA will not find it.

To address this issue, we propose a method that automatically discovers learner models with less

dependent on human-provided factors. The system uses the extended SimStudent to acquire skill knowledge. Each production rule corresponds to a KC that students need to learn. The model then labels each observation of a real student based on skill application.

To demonstrate the generality of this approach, we present evaluations of the SimStudent-generated models in three domains: algebra, stoichiometry, fraction addition, and article selection. We validate the quality of the cognitive models using human student data as in [Koedinger and MacLaren \[1997\]](#). Instead of matching with performance data, we use the discovered cognitive model to predict human learning curve data. Experimental results show that for algebra and stoichiometry, SimStudent directly finds a better cognitive model than humans. In the article selection domain, SimStudent successfully recovers the human-constructed model. For fraction addition, SimStudent results assist LFA in finding a better cognitive model than a domain expert. We have also carried out an in-depth study using Focused Benefits Investigation (FBI) [[Koedinger et al., 2012](#)] to better understand this machine learning approach, and discussed possible ways of further improvements.

## 8.1 Methods

In order to evaluate the effectiveness of the proposed approach, we carry out a study using three datasets. We compare the SimStudent model with a human-generated KC model by first coding the real student steps using the two models, and then testing how well the two model coding predicts real student data. Note that DataShop [[Koedinger et al., 2010](#)] has multiple KC models for each dataset in the current study, the human-generated KC model we select here is one of the best models among the existing learner models.

KCs associated with the basic arithmetic operations (i.e., add, subtract, multiply, and divide) are then further split into two KCs for each, namely a skill to identify an appropriate basic operator and a skill to actually execute the basic operator. The former is called a *transformation* skill whereas the latter is a *typein* skill. As a consequence, there are 12 KCs defined (called the Action-Typein KC model). Not all steps in the algebra dataset were coded with these KC models – some steps are about a transformation that we do not include in the Action KC model (e.g., simplify division). There are 9487 steps that can be coded by both KC models mentioned above. The “default” KC model, which were defined by the productions implemented for the cognitive tutor, has only 6809 steps that can be coded. To make a fair comparison between the “default” and “Action-Typein” KC models, we took the intersection of those 9487 and 6809 steps. As a result, there are 6507 steps that can be coded by both the default and the Action-Typein KC models. We then define a new KC model, called the Balanced-Action-Typein KC model that has the same set of KCs as the Action-Typein model but is only associated with these 6507 steps, and used this KC model to compare with the SimStudent model.

To generate the SimStudent model, SimStudent is tutored on how to solve problems by interacting with an automated tutor, like the one used by human students in studies. As the training set for SimStudent, we selected problems that were used to teach real students. Given all of the

acquired production rules, for each step a real student performed, we assigned the applicable production rule as the KC associated with that step. In cases where there was no applicable production rule, we coded the step using the human-generated KC model. Each time a student encounters a step using some KC, it is considered as an “opportunity” for that student to show mastery of that KC, and learn the KC by practicing it.

Having finished coding real student steps with both models (the SimStudent model and the human-generated model), we used the Additive Factor Model (AFM) [Cen et al., 2006] to validate the coded steps. AFM is an instance of logistic regression that models student success using each student, each KC, and the KC by opportunity interaction as independent variables,

$$\ln \frac{p_{ij}}{1 - p_{ij}} = \theta_i + \sum_k \beta_k Q_{kj} + \sum_k Q_{kj} (\gamma_k N_{ik}) \quad (8.1)$$

Where:

**i** represents a student i.

**j** represents a step j.

**k** represents a skill or KC k.

$p_{ij}$  is the probability that student i would be correct on step j.

$\theta_i$  is the coefficient for proficiency of student i.

$\beta_k$  is coefficient for difficulty of the skill or KC k

$Q_{kj}$  is the Q-matrix cell for step j using skill k.

$\gamma_k$  is the coefficient for the learning rate of skill k;

$N_{ik}$  is the number of practice opportunities student i has had on the skill k;

We utilized DataShop [Koedinger et al., 2010], a large repository that contains datasets from various educational domains as well as a set of associated visualization and analysis tools, to facilitate the process of evaluation, which includes generating learning curve visualization, AFM parameter estimation, and evaluation statistics including AIC (Akaike Information Criterion) and cross validation.

## 8.2 Dataset

We carry out our study in three domains: algebra, stoichiometry, and fraction addition. In algebra, we analyze data from 71 students who used an Carnegie Learning Algebra I Tutor unit on equation solving. The students are typical students at a vocational-technical school in a rural/suburban area outside of Pittsburgh, PA. A total of 19,683 transactions between the students

and the Algebra Tutor were recorded, where each transaction represents an attempt or inquiry made by the student, and the feedback given by the tutor.

The stoichiometry dataset contains data from 3 studies. 510 high school and college students participated in the studies, and generated 172,060 transactions. Instructional videos on stoichiometry are intermingled with the problems. Instructional materials were provided via the Internet. It took students from 1.5 hours to 6.5 hours to complete the study.

In fraction addition, we analyze data from 24 students who use an intelligent tutoring system as part of a larger study. Approximately half of the students are recruited from local schools, the PSLC subject pool, and word of mouth. Participants in both settings are given a brief video demonstration on how to use the tutors, and then have 30 minutes to solve 20 fraction addition problems with the tutor. Students are given immediate correctness feedback on each step, and are offered on-demand text hints. Each interaction is logged through Datashop, and the 24 students yielded 4558 transactions.

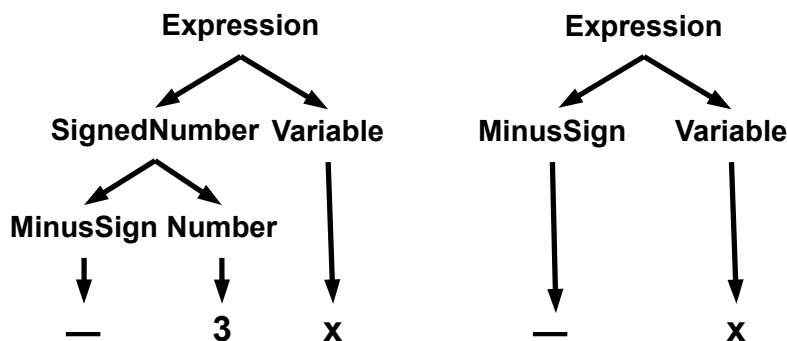
Finally, in the article selection domain, we use data collected from [Wylie et al.’s \[2010\]](#) recent study on second language learning. The study is conducted at the University of Pittsburgh’s English Language Institute. Students ( $N = 99$ ) are adult English language learners ( $meanage = 27.9$ ,  $SD = 6.6$ ) and participate as part of their regular grammar class. Data collection is completed within one 50-minute class period. Pre- and post-test items are identical in the form of the practice problems students has seen during tutoring without feedback and hints. All of the student behaviors are recorded during the process, and encoded with rules applied to the problems and whether students answers are correct.

### 8.3 Measurements

To test how well the existing and generated models predict with real student data, we use AIC and a 10-fold cross validation. AIC measures the fit to student data while penalizing over-fitting. We did not use BIC (Bayesian Information Criterion) as the fit metric, because based on past analysis across multiple DataShop datasets, it has been shown that AIC is a better predictor of cross validation than BIC is. The cross validation was performed over three folds with the constraint that each of the three training sets must have data points for each student and KC. We report the root mean-squared error (RMSE) averaged over three test sets.

### 8.4 Experimental Results

As shown in Table 8.2 and Table 8.3, in algebra and stoichiometry, the SimStudent-discovered models that have lower AICs and RMSEs than the human-generated models. This means the SimStudent models better match the data (without over-fitting). However, in fraction addition, the human-generated model performs better than the SimStudent-discovered ones.



**Figure 8.1:** Different parse trees for  $-3x$  and  $-x$ .

**Table 8.1:** Number of KCs in SimStudent models and Human-Generated Models.

	Human-Generated Model	SimStudent-Discovered Model
Algebra	12	21
Stoichiometry	44	46
Fraction Addition	8	6
Article Selection	19	22

**Table 8.2:** AIC on SimStudent-Generated models and Human-Generated Models.

	Human-Generated Model	SimStudent-Discovered Model
Algebra	6534.07	<b>6448.1</b>
Stoichiometry	17380.9	<b>17218.5</b>
Fraction Addition	<b>2112.82</b>	2202.02
Article Selection	6221.49	<b>6221.39</b>

A closer look at the models reveals that in algebra, the SimStudent-discovered model splits some of the KCs in the human-generated model into finer grain sizes. For example, SimStudent creates two KCs for division, one for problems of the form  $Nv = N$ , and one for problems of the form  $-v = N$ . This is caused by the different parse trees for  $Nv$  and  $-v$  as shown in Figure 8.1. Due to this split, the SimStudent-generated model predicts a higher error rate on problems of the form  $-v = N$  than problems of the form  $Nv = N$ . It matches with human student error rates better than the human-generated model, which does not differentiate problems of these two forms.

In stoichiometry, instead of finding splits of existing KCs, SimStudent discovers new KCs that overlap with the original KCs. There are three basic sets of skills in this domain. Within each set,

**Table 8.3:** CV RMSE on SimStudent-Generated models and Human-Generated Models.

	Human-Generated Model	SimStudent-Discovered Model
Algebra	0.4024	<b>0.3999</b>
Stoichiometry	0.3501	<b>0.3488</b>
Fraction Addition	<b>0.3232</b>	0.3343
Article Selection	0.4044	<b>0.4033</b>

the human-generated KCs are assigned based on the location of the input, while the SimStudent-discovered KCs are associated with the goals of the input. Hence, suppose in two different problems, there are two inputs at the same location in the interface. If they are associated with different goals, the human-generated model will not differentiate them, while the SimStudent-discovered model will put them into two KCs. This indicates that SimStudent not only splits existing KCs, but also discovers totally different KCs.

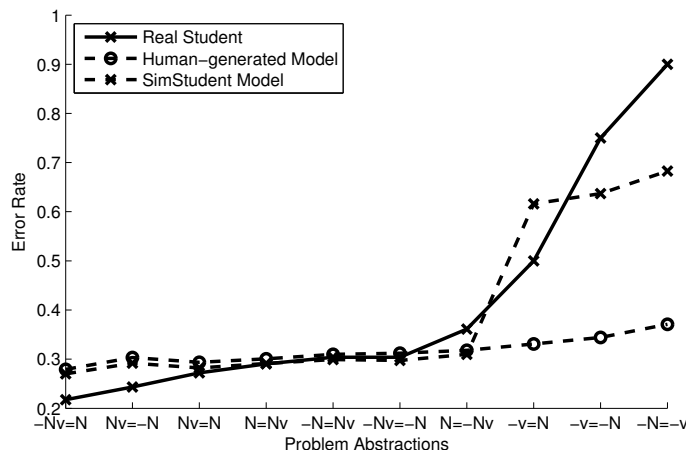
The fraction addition problem set consists of three types of problems in increasing difficulty: 1) addends have equal denominators; 2) the denominator of one addend is a multiple of the other; 3) addends have unrelated denominators. The human-generated model differentiates these three types of problems in calculating the common denominators and the scaled numerators, and ends up having six KCs. SimStudent, however, associates all of the numerator scaling steps with one KC and associates the common denominator calculations with two KCs. In other words, in this domain, SimStudent partially recovered three out of six KCs, but did not further split them into six KCs. SimStudent did discover the other three KCs, but eventually removed them when they were superseded by more generalized rules. This bias towards more general production rules over specific ones regardless of computational cost appears to be a limitation of SimStudent as a cognitive model. Perhaps if we had let SimStudent keep a utility function for each production rule and retrieve them based on the computational cost, last retrieval time, and correctness, SimStudent may have arrived at all six KCs in the human-generated model.

In article selection, SimStudent successfully recovers the KCs associated with the six grammar rules. Moreover, it splits the rule “number-letter” into two KCs, one for number and one for letter. The SimStudent-generated model is as good as the human-generated model both in terms of AIC (6221.39 vs. 6221.49) and the root mean-squared error in cross validation (0.3769 vs. 0.3777). This suggests that SimStudent can reproduce a human-generated model in terms of its quality of fit with human learning data.

## 8.5 FBI Analysis and LFA on Fraction Addition

The differences of AIC and RMSE between the models are small. This is partially because the difference between the models is small in the sense that most the KCs in the new model are the same as in the old model. Thus, the predictions for problem-steps modeled by those





**Figure 8.2:** Error rates for real students and predicted error rates from two learner models.

unchanged KCs will be highly similar (not exactly the same because the the student proficiency parameter estimates will be slightly different as a consequence of the changed KCs). FBI, a recently developed technique, is designed to analyze which of these differences improves the model, and by how much. For the original KCs that are different in the discovered model. FBI shows the greatest reduction in prediction error. We apply FBI to the SimStudent and human-generated models in each domain to determine why the SimStudent models are better in two of the three cases. In the analysis, we set the human-generated models as the base.

FBI shows that in algebra, splitting “divide” reduces the RMSE of those steps by 1.02%. Further, splitting subtraction and addition decreases the RMSE of those steps by 3.78% and 3.10%, respectively. Similar results are also observed in the article selection domain, where the split of “number-letter” rule into two KCs leads to better RMSE up to 2.25% in of the human-generated KCs. This also indicates that SimStudent is able to discover KCs of finer grain sizes that match with human data well.

The stoichiometry results are different. SimStudent discovered new KCs that were not part of any existing KCs. Given the 40 KCs in the human-generated model, SimStudent improves 26 of them. The biggest improvement is on skill molecular weight (4.60%), since there are sometimes more than one skill applicable to the same step. The human-generated model misses the additional skill, while the SimStudent model successfully captures both skills.

As described previously, SimStudent does not differentiate the numerator-scaling and common-denominator steps by problem type. This hurts the RMSE of the associated KCs in the SimStudent-generated model. Nevertheless, SimStudent considers finding the common denominator to be a different KC than copying it to the second converted addend. This split decreases by 7.43% for problems with unrelated denominators, and by 0.12% for denominator steps of problems where one addend denominator was a multiple of the other.

Given the above results, we carry out a third study on fraction addition to test that whether the new KCs created by SimStudent can be used to discover better cognitive models. We used

LFA to discover cognitive models given two sets of factors. The baseline LFA model was generated based on the factors (KCs) in the human-generated model. The other LFA model was discovered using both the factors (KCs) in the human-generated model and those in the SimStudent-generated model. Both LFA models were better than the original human-generated model in terms of AIC and RMSE. Moreover, the LFA model using both human-generated and SimStudent-generated factors had better AIC (2061.4) and RMSE (0.3189) than the baseline LFA model (AIC: 2111.96, RMSE 0.3226). In other words, with the help of SimStudent, LFA discovered better models of human students.

## 8.6 Detailed Implications for Instructional Decision in Algebra

We can inspect the data more closely to get a better qualitative understanding of why the SimStudent model is better and what implications there might be for improved instruction. Among the 21 KCs learned by the SimStudent model, there were 17 transformation KCs and four typein KCs. It is hard to map the SimStudent KC model directly to the expert model. Approximately speaking, the distribute, clt (i.e. combine like terms), mt, rf KCs as well as the four typein KCs are similar to the KCs defined in the expert model. The transformation skills associated with the basic arithmetic operators (i.e., add, subtract, multiply and divide) are further split into finer grain sizes based on different problem forms.

One example of such split is that SimStudent created two KCs for division. The first KC (simSt-divide) corresponds to problems of the form  $Ax=B$ , where both  $A$  and  $B$  are signed numbers, whereas the second KC (simSt-divide-1) is specifically associated with problems of the form  $-x=A$ , where  $A$  is a signed number. This is caused by the different parse trees for  $Ax$  vs  $-x$  as shown in Figure 8.1. To solve  $Ax=B$ , SimStudent simply needs to divide both sides with the signed number  $A$ . On the other hand, since  $-x$  does not have  $-1$  represented explicitly in the parse tree, SimStudent needs to see  $-x$  as  $-1x$ , and then to extract  $-1$  as the coefficient. If SimStudent is a good model of human learning, we expect the same to be true for human students. That is, real students should have greater difficulty in making the correct move on steps like  $-x = 6$  than on steps like  $-3x = 6$  because of the need to convert (perhaps just mentally)  $-x$  to  $-1x$ . To evaluate this hypothesis, we computed the average error rates for a relevant set of problem types – these are shown with the solid line in Figure 8.2 with the problem types defined in forms like  $-Nv=N$ , where the  $N$ s are any integrate number and the  $v$  is a variable (e.g.,  $-3x=6$  is an instance of  $-Nv=N$  and  $-6=-x$  is an instance of  $-N=-v$ ). The problem types are sorted by increasing error rates. In other words, the problem types to the right are harder for human students than those to the left.

We also calculated the mean of the predicted error rates for each problem type for both the human-generated model and the SimStudent model. Consistent with the hypothesis, as shown in Figure 8.2, we see that problems of the form  $Ax=B$  (average error rate 0.283) are much simpler than problems of the form  $-x=A$  (average error rate 0.719). The human-generated model predicts

all problem types with similar error rates (average predicted error rate for  $Ax=B$  0.302, average predicted error rate for  $-x=A$  0.334), and thus fails to capture the difficulty difference between the two problem types ( $Ax=B$  and  $-x=A$ ). The SimStudent model, on the other hand, fits with the real student error rates much better. It predicts higher error rates (0.633 on average) for problems of the form  $-x=A$  than problems of the form  $Ax=B$  (0.291 on average).

SimStudent’s split of the original division KC into two KCs, *simSt-divide* and *simSt-divide-1*, suggests that the tutor should teach real students to solve two types of division problems separately. In other words, when tutoring students with division problems, we should include two subsets of problems, one subset corresponding to *simSt-divide* problems ( $Ax=B$ ), and one specifically for *simSt-divide-1* problems ( $-x=A$ ). We should perhaps also include explicit instruction that highlights for students that  $-x$  is the same as  $-1x$ .

## 8.7 Discussion

The objective of this work is to use a learning agent, SimStudent, to automatically construct learner models. [Conati and VanLehn \[1999\]](#) also applied machine learning techniques to generate cognitive models that fit with human data, but they focused on assessing self-explanation instead of student learning. Additionally, there has been considerable work on comparing the quality of alternative cognitive models. LFA automatically discovers cognitive models, but is limited to the space of the human-provided factors. Other works such as [Pavlik et al. \[2009\]](#), [Villano \[1992\]](#) are less dependent on human labeling, but the models generated may be hard to interpret. In contrast, the SimStudent approach has the benefit that the acquired production rules have a precise and usually straightforward interpretation.

Other systems (e.g., [Tatsuoka, 1983](#), [Barnes, 2005](#)) use a Q-matrix to find knowledge structure from student response data. [Baffes and Mooney \[1996\]](#) apply theory refinement to the problem of modeling incorrect student behavior. In addition, some research (e.g., [Langley and Ohlsson, 1984](#), [VanLehn, 1990](#)) uses artificial intelligent techniques to construct models that explain student’s behavior in math domains. Brown and Burton’s [[Burton, 1982](#)] DEBUGGY, and [Sleeman and Smith’s \[1981\]](#) LMS also make use of artificial intelligent tools to construct models that explain student’s behavior in math domains. [VanLehn’s \[1990\]](#) Sierra models the impasse-driven acquisition of hierarchical procedures for multi-column subtraction from sample solutions. However, his work focused on explaining the origin of bugs for real students, which is not the focus here. In addition, Sierra is given a CFG for parsing the visual state of the subtraction problems, whereas our system automatically acquires a pCFG.

Besides SimStudent, there has also been considerable research on learning within agent architectures (e.g., [Laird et al., 1986](#), [Anderson, 1993](#), [Taatgen and Lee, 2003](#)). Other research on creating simulated students (e.g., [Chan and Chou, 1997](#), [Pentti Hietala, 1998](#)) is also closely related to our work. Nevertheless, none of the above approaches focused on modeling how representation learning affects skill learning. Moreover, none of them compared the system with human learning curve data. To the best of our knowledge, our work is the first combination of the

two whereby we use cognitive model evaluation techniques to assess the quality of a simulated learner, and demonstrate it across multiple domains.

In the study, we show that the integration of the representation learning component into skill learning is key to the success of SimStudent in discovering learner models. Results indicate that in three out of four domains, SimStudent-generated models are either as good as the human-generated models, or become better predictors of human students' learning performance than human-coded models. For the fourth domain, when given the SimStudent- and human-generated KCs, LFA finds a better model than the human-generated one. A closer analysis shows that SimStudent is able to split existing KCs into finer grain sizes, discover new KCs, and uncover expert blind spots.

# Chapter 9

## Conclusion

To sum up, building an intelligent agent that simulates human-level learning is an essential task in AI and cognitive science, but building such systems often requires manual encoding of prior domain knowledge. In this paper, we proposed a learning mechanism that automatically acquires representations of the problems in terms of deep features from observations without any annotation or with light annotations. We then integrate this stand-alone representation learner into an intelligent agent, SimStudent, as an extension of the perception module. We showed that after the integration, the extended SimStudent is able to achieve comparable or better performance without or with few domain-specific knowledge engineering efforts across multiple domains.

We have shown that because of the addition of the greedy structure hypothesizer and the Viterbi training phase, the search space of the proposed representation learner is smaller than that of the inside-outside algorithm [Lari and Young, 1990]. Experimental results demonstrate that the proposed representation learner acquires grammar more effectively than the inside-outside algorithm [Lari and Young, 1990]. We believe that this result should also hold for the integrated setting. One interesting future study is to carry out more comprehensive experiments on the comparison between the proposed representation learner and other grammar induction techniques.

We further evaluated the generality of the approach in a world-knowledge rich domain. We extended representation learning with external world knowledge, and integrated it into SimStudent. Results show that given a reasonable number (e.g., 60) of training examples, the extended SimStudent successfully learns six frequently used article selection rules, and can be used to find learner models that predict human student behavior as well as a human-generated model.

The current implementation of the simulated student consists of multiple learning mechanisms in the system, which is consistent with Ohlsson's 2008 claim on how different learning models are employed during different learning phases in intelligent systems. Nevertheless, one interesting question to ask is that whether it is possible to build a more joint learning model that captures representation learning as well as skill learning. We have carried out a preliminary study in this direction. The task of acquiring the precondition of the rule is a classification task. In SimStudent, we decoupled this learning module into two components, the unsupervised statistical mod-

ule that learns the world representation and generates a set of feature predicates, and a supervised logic-based module that uses the generated feature predicates to acquire the precondition of the production rule. In comparison with this decoupled learning strategy, we adapt a joint model, deep belief network [Hinton et al., 2006], to the precondition learning task. Experiment results show that deep belief network achieves reasonable performance ( $>80\%$ ), but is not as effective as the decoupled strategy ( $>90\%$ ). More extensive studies on the comparison between a joint model and a decouple model are needed in better understanding each type of the models.

In addition to being an effective learner, we further showed that the extended SimStudent could be used to discover better models of real students. We introduced an innovative application of the extended SimStudent for an automatic discovery of learner models. An empirical study showed that a SimStudent generated learner model was a better predictor of real students learning performance than a human-coded learner model. The basic idea is to have SimStudent learn to solve the same problems that human students did and use the productions that SimStudent generated as knowledge components to codify problem-solving steps. We then used these KC coded steps to validate the models prediction. Unlike the human-engineered learner model, the SimStudent generated learner model has a clear connection between the features of the domain contents and knowledge components. An advantage of the SimStudent approach of learner modeling over previous techniques like LFA is that it does not depend heavily on the human-engineered features. SimStudent can automatically discover a need to split a purported KC or skill into more than one skill. During SimStudents learning, a failure of generalization for a particular KC results in learning disjunctive rules. Discovering such disjunctive rules is equivalent to splitting a KC in LFA, however, whereas human needs to provide potential factors to LFA as the basis for a possible split, SimStudent can learn such factors. The use of the perceptual learning component, implemented using a probabilistic context-free grammar learner, is a key feature of SimStudent for these purposes as we hypothesized that a major part of human expertise, even in academic domains like algebra, is such perceptual learning.

Our evaluation demonstrated that representing the rules SimStudent learns in the learner model improves the accuracy of model prediction, and showed how the SimStudent model could provide important instructional implications. Much of human expertise is only tacitly known. For instance, we know the grammar of our first language but do not know what we know. Similarly, most algebra experts have no explicit awareness of subtle transformations they have acquired like the one above (seeing  $-x$  as  $-1x$ ). Even though such instructional designers may be experts in a domain they have thus have some blind spots regarding subtle perceptual differences like this one, which may make a real difference for novice learners. A machine learning agent, like SimStudent, can help get past such blind spots by revealing challenges in the learning process that experts may not be aware of.

Being a better model of human student learning, we conducted a controlled simulation study on SimStudent in three math and science domains (i.e., fraction addition, equation solving and stoichiometry) to understand how different problem orders affect learning effectiveness. The results show that the interleaved problem order yields as or more effective learning in all three domains, as the interleaved problem order provides more or better opportunities for error detection and correction to the learning agent. The study shows that learning when to apply a skill benefits more

from interleaved problem orders, and suggests that learning how to apply a skill benefits more from blocked problem orders. Given all the results, we conclude that the extended SimStudent is able to acquire skill knowledge with a small amount of knowledge engineering.





# Bibliography

- V. Aleven, B. M. McLaren, J. Sewall, and K. R. Koedinger. A new paradigm for intelligent tutoring systems: Example-tracing tutors. *International Journal of Artificial Intelligence in Education*, 19:105–154, April 2009. ISSN 1560-4292. [31](#), [54](#), [57](#)
- J. R. Anderson. *Rules of the Mind*. Lawrence Erlbaum Associates, Hillsdale, New Jersey, 1993. [1](#), [9](#), [34](#), [66](#), [75](#)
- J. R. Anderson and R. Thompson. Similarity and analogical reasoning. chapter Use of analogy in a production system architecture, pages 267–297. Cambridge University Press, New York, NY, USA, 1989. [34](#)
- Y. Anzai and H. A. Simon. The theory of learning by doing. *Psychological Review*, 86(2): 124–140, 1979. [1](#), [2](#), [66](#)
- A. Arasu and H. Garcia-Molina. Extracting structured data from web pages. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 337–348, New York, NY, USA, 2003. ACM. [50](#)
- P. Baffes and R. Mooney. Refinement-based student modeling and automated bug library construction. *Journal of Artificial Intelligence in Education*, 7(1):75–116, 1996. ISSN 1043-1020. [75](#)
- T. Barnes. The Q-matrix method: Mining student response data for knowledge. In *Proceedings AAAI Workshop Educational Data Mining*, pages 1–8, Pittsburgh, PA, 2005. [75](#)
- R. V. Belavkin and F. E. Ritter. OPTIMIST: A New Conflict Resolution Algorithm for ACTR. In *Proceedings of the sixth International Conference on Cognitive Modeling*, pages 40–45, Pittsburgh, PA, 2004. [64](#)
- Y. Bengio. Learning deep architectures for ai. *Foundations Trends in Machine Learning*, 2: 1–127, January 2009. ISSN 1935-8237. [35](#), [60](#)
- Y. Bengio, O. Delalleau, and C. Simard. Decision trees do not generalize to new variations. *Computational Intelligence*, 26(4):449–467, Nov. 2010. [36](#)
- D. Blei and J. McAuliffe. Supervised topic models. In *Proceedings of the Twenty-Fifth Annual Conference on Neural Information Processing Systems*, pages 121–128, Cambridge, MA, 2007. MIT Press. [2](#)
- R. R. Burton. Diagnosing bugs in a simple procedural skill. In *Intelligent Tutoring Systems*, pages 157–184. Academic Press, 1982. [75](#)

- M. J. Cafarella, A. Y. Halevy, D. Z. Wang, E. W. 0002, and Y. Zhang. Webtables: exploring the power of tables on the web. *Proceedings of the VLDB Endowment*, 1(1):538–549, 2008. [50](#)
- H. Cen, K. Koedinger, and B. Junker. Learning factors analysis - a general method for cognitive model evaluation and improvement. In *Proceedings of the 8th International Conference on Intelligent Tutoring Systems*, pages 164–175, 2006. [67](#), [69](#)
- T.-W. Chan and C.-Y. Chou. Exploring the design of computer supports for reciprocal tutoring. *International Journal of Artificial Intelligence in Education*, 8:1–29, 1997. [75](#)
- W. G. Chase and H. A. Simon. Perception in chess. *Cognitive Psychology*, 4(1):55–81, Jan. 1973. [2](#), [13](#), [23](#), [24](#)
- M. T. H. Chi, P. J. Feltovich, and R. Glaser. Categorization and representation of physics problems by experts and novices. *Cognitive Science*, 5(2):121–152, June 1981. [2](#), [13](#)
- P. A. Chou. Recognition of Equations Using a Two-Dimensional Stochastic Context-Free Grammar. In *Proceedings of Visual Communications and Image Processing*, volume 1199, pages 852–863, Nov. 1989. [39](#), [50](#)
- C. Conati and K. VanLehn. A student model to assess self-explanation while learning from examples. In *Proceedings of the seventh international conference on User modeling*, pages 303–305, Secaucus, NJ, 1999. Springer-Verlag New York, Inc. [75](#)
- V. Crescenzi, G. Mecca, and P. Merialdo. Roadrunner: Towards automatic data extraction from large web sites. In *Proceedings of the 27th International Conference on Very Large Data Bases*, pages 109–118, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc. [50](#)
- A. Cypher, D. C. Halbert, D. Kurlander, H. Lieberman, D. Maulsby, B. A. Myers, and A. Turransky, editors. *Watch what I do: programming by demonstration*. MIT Press, Cambridge, MA, 1993. ISBN 0-262-03213-9. [35](#)
- P. Dayan, G. E. Hinton, R. M. Neal, and R. S. Zemel. The Helmholtz Machine. *Neural Computation*, 7(5):889–904, Dec. 1995. [35](#)
- A. P. Dempster, N. M. Laird, and D. B. Rubin. Maximum Likelihood from Incomplete Data via the EM Algorithm. *Journal of the Royal Statistical Society. Series B (Methodological)*, 39(1):1–38, 1977. [16](#)
- T. G. Dietterich. Learning at the knowledge level. *Machine Learning*, 1(3):287–315, 1986. [7](#), [34](#), [66](#)
- T. Fawcett. Knowledge-based feature discovery for evaluation functions. *Computational Intelligence*, 12(1), 1996. [60](#)
- F. Gobet. Chunking models of expertise: implications for education. *Applied Cognitive Psychology*, 19(3):183–204, Jan. 2005. [24](#)
- F. Gobet and H. A. Simon. Five seconds or sixty? presentation time in expert memory. *Cognitive Science*, 24(4):651–682, 2000. [24](#)
- P. Harrison, S. Abney, E. Black, C. Gdaniec, R. Grishman, D. Hindle, R. Ingria, M. P. Marcus, B. Santorini, and T. Strzalkowski. Evaluating syntax performance of parser/grammars of English. In *Natural Language Processing Systems Evaluation Workshop*, Technical Report,

- pages 71–78, Griffis Air Force Base, NY, 1991. [46](#)
- J. Hay and J. Bresnan. Spoken syntax: The phonetics of giving a hand in new zealand english. In *The Linguistic Review: Special Issue on Exemplar-Based Models in Linguistics*, pages 321–349, 2006. [61](#)
- G. E. Hinton. To recognize shapes, first learn to generate images. *Progress in brain research*, 165:535–547, 2007. [2](#), [36](#)
- G. E. Hinton, P. Dayan, B. J. Frey, and R. M. Neal. The ”wake-sleep” algorithm for unsupervised neural networks. *Science*, 268(5214):1158–1161, May 1995. [35](#)
- G. E. Hinton, S. Osindero, and Y.-W. Teh. A fast learning algorithm for deep belief nets. *Neural Computation*, 18(7):1527–1554, 2006. [78](#)
- R. Hwa. Supervised grammar induction using training data with limited constituent information. In *Proceedings of the 37th annual meeting of the Association for Computational Linguistics on Computational Linguistics*, pages 73–79, Stroudsburg, PA, USA, 1999. Association for Computational Linguistics. [23](#)
- C. Kemp and J. B. B. Tenenbaum. The discovery of structural form. *Proceedings of the National Academy of Sciences of the United States of America*, July 2008. ISSN 1091-6490. [35](#)
- C. Kemp and F. Xu. An ideal observer model of infant object perception. In D. Koller, D. Schuurmans, Y. Bengio, and L. Bottou, editors, *NIPS*, pages 825–832. MIT Press, 2008. [35](#)
- D. Klein and C. D. Manning. Accurate unlexicalized parsing. In *Proceedings of the 41st Annual Meeting on Association for Computational Linguistics - Volume 1*, ACL ’03, pages 423–430, Stroudsburg, PA, USA, 2003. Association for Computational Linguistics. [61](#)
- K. R. Koedinger and J. R. Anderson. Abstract Planning and Perceptual Chunks: Elements of Expertise in Geometry. *Cognitive Science*, 14:511–550, 1990. [13](#)
- K. R. Koedinger and A. Corbett. Cognitive Tutors: Technology Bringing Learning Sciences to the Classroom. pages 60–77. Cambridge University Press, Cambridge, 2006. [8](#)
- K. R. Koedinger and B. A. MacLaren. Implicit strategies and errors in an improved model of early algebra problem solving. In *Proceedings of the Nineteenth Annual Conference of the Cognitive Science Society*, pages 382–387, Hillsdale, NJ, 1997. Erlbaum. [68](#)
- K. R. Koedinger and E. A. McLaughlin. Seeing language learning inside the math: Cognitive analysis yields transfer. In *Proceedings of the 32nd Annual Conference of the Cognitive Science Society*, pages 471–476, Austin, TX, 2010. [67](#)
- K. R. Koedinger and M. J. Nathan. The real story behind story problems: Effects of representations on quantitative reasoning. *The Journal of Learning Sciences*, 13(2):129–164, 2004. [67](#)
- K. R. Koedinger, R. S. Baker, K. Cunningham, A. Skogsholm, B. Leber, and J. Stamper. A data repository for the EDM community: The PSLC DataShop, 2010. [68](#), [69](#)
- K. R. Koedinger, E. A. McLaughlin, and J. C. Stamper. Automated student model improvement. In *Proceedings of the 5th International Conference on Educational Data Mining*, pages 17–24, Chania, Greece, 2012. ISBN 978-1-74210-276-4. [68](#)

- J. E. Laird, P. S. Rosenbloom, and A. Newell. Chunking in soar: The anatomy of a general learning mechanism. *Machine Learning*, 1:11–46, 1986. [34](#), [66](#), [75](#)
- J. E. Laird, A. Newell, and P. S. Rosenbloom. Soar: an architecture for general intelligence. *Artificial Intelligence*, 33(1):1–64, 1987. ISSN 0004-3702. doi: [http://dx.doi.org/10.1016/0004-3702\(87\)90050-6](http://dx.doi.org/10.1016/0004-3702(87)90050-6). [1](#), [9](#)
- P. Langley and D. Choi. A unified cognitive architecture for physical agents. In *Proceedings of the Twenty-First National Conference on Artificial Intelligence*, Boston, 2006. [1](#), [34](#)
- P. Langley and S. Ohlsson. Automated cognitive modeling. In *Proceedings of the Fourth National Conference on Artificial Intelligence*, pages 193–197, Austin, TX, 1984. Morgan Kaufmann. [75](#)
- P. Langley and S. Stromsten. Learning context-free grammars with a simplicity bias. In *Proceedings of the 11th European Conference on Machine Learning*, pages 220–228, London, UK, 2000. Springer-Verlag. [23](#)
- K. Lari and S. J. Young. The estimation of stochastic context-free grammars using the inside-outside algorithm. *Computer Speech and Language*, 4:35–56, 1990. [14](#), [41](#), [77](#)
- T. Lau and D. S. Weld. Programming by demonstration: An inductive learning formulation. In *Proceedings of the 1999 international conference on intelligence user interfaces*, pages 145–152, 1998. [5](#), [35](#)
- Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel. Backpropagation applied to handwritten zip code recognition. *Neural Comput.*, 1:541–551, December 1989. [35](#)
- Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. In *Proceedings of the IEEE*, pages 2278–2324, 1998. [35](#)
- N. Li, S. Kambhampati, and S. Yoon. Learning probabilistic hierarchical task networks to capture user preferences. In *Proceedings of the 21st International Joint Conference on Artificial Intelligence*, Pasadena, CA, 2009. [13](#), [14](#), [16](#), [17](#), [20](#), [23](#)
- N. Li, W. W. Cohen, and K. R. Koedinger. A computational model of accelerated future learning through feature recognition. In *ITS'10: Proceedings of 10th International Conference on Intelligent Tutoring Systems*, pages 368–370, 2010. [3](#), [38](#), [41](#)
- N. Li, W. W. Cohen, N. Matsuda, and K. R. Koedinger. A machine learning approach for automatic student model discovery. In *Proceedings of the 4th International Conference on Educational Data Mining*, pages 31–40, 2011a. [13](#)
- N. Li, W. Cushing, S. Kambhampati, and S. Yoon. Learning probabilistic hierarchical task networks as probabilistic context-free grammars to capture user preferences. Technical Report arxiv:1006.0274 (Revised), Arizona State University, 2011b. [42](#)
- N. Li, W. W. Cohen, and K. R. Koedinger. Efficient cross-domain learning of complex skills. In *Proceedings of the Eleventh International Conference on Intelligent Tutoring Systems*, pages 493–498, Berlin, 2012a. Springer-Verlag. [57](#)
- N. Li, W. W. Cohen, and K. R. Koedinger. Integrating representation learning and skill learning in a human-like intelligent agent. Technical Report CMU-MLD-12-1001, Carnegie Mellon

- University, January 2012b. [21](#)
- M. Martín and H. Geffner. Learning generalized policies from planning examples using concept languages. *Applied Intelligence*, 20:9–19, January 2004. [60](#)
- N. Matsuda, A. Lee, W. W. Cohen, and K. R. Koedinger. A computational model of how learner errors arise from weak prior knowledge. In *Proceedings of Conference of the Cognitive Science Society*, 2009. [1](#), [2](#), [3](#), [5](#), [10](#), [51](#), [66](#)
- B. M. McLaren, S.-j. Lim, and K. R. Koedinger. When and how often should worked examples be given to students ? new results and a summary of the current state of research why isn't the science done ? *Cognitive Science*, pages 2176–2181, 2008. [31](#)
- T. Mitchell. Generalization as search. *Artificial Intelligence*, 18(2):203–226, 1982. [5](#), [11](#)
- T. M. Mitchell, S. Mahadevan, and L. I. Steinberg. Leap: a learning apprentice for vlsi design. In *Proceedings of the 9th international joint conference on Artificial intelligence*, pages 573–580, San Francisco, CA, 1985. ISBN 0-934613-02-8, 978-0-934-61302-6. [35](#)
- R. J. Mooney. *A General Explanation-Based Learning Mechanism and its Application to Narrative Understanding*. Morgan Kaufmann, San Mateo, CA, 1990. [34](#)
- S. Muggleton and W. Buntine. Machine invention of first-order predicates by inverting resolution. In *Proceedings of the Fifth International Conference on Machine Learning*, pages 339–352. Morgan Kaufmann, 1988. [60](#)
- S. Muggleton and L. de Raedt. Inductive logic programming: Theory and methods. *Journal of Logic Programming*, 19:629–679, 1994. [5](#)
- D. M. Neves. Learning procedures from examples and by doing. In *Proceedings of the 9th international joint conference on Artificial intelligence - Volume 1*, pages 624–630, San Francisco, CA, USA, 1985. Morgan Kaufmann Publishers Inc. [1](#), [2](#), [35](#), [66](#)
- A. Niculescu-Mizil and R. Caruana. Inductive transfer for bayesian network structure learning. In *Proceedings of the 11th International Conference on AI and Statistics*, 2007. [35](#)
- S. Ohlsson. *Computational Models of Skill Acquisition*, chapter 13, pages 359–395. Cambridge University Press, 2008. [36](#), [77](#)
- P. I. Pavlik, H. Cen, and K. R. Koedinger. Learning Factors Transfer Analysis: Using Learning Curve Analysis to Automatically Generate Domain Models. In *Proceedings of 2nd International Conference on Educational Data Mining*, pages 121–130, 2009. [75](#)
- T. N. Pentti Hietala. The competence of learning companion agents. *International Journal of Artificial Intelligence in Education*, 9:178–192, 1998. [75](#)
- J. R. Quinlan. Learning logical definitions from relations. *Machine Learning*, 5(3):239–266, 1990. [11](#), [60](#)
- L. D. Raedt and L. Dehaspe. Clausal discovery. *Machine Learning*, 26(2):99–146, 1997. [60](#)
- R. Raina, A. Y. Ng, and D. Koller. Constructing informative priors using transfer learning. In *Proceedings of the 23rd international conference on Machine learning*, pages 713–720, New York, NY, 2006. ISBN 1-59593-383-2. doi: <http://doi.acm.org/10.1145/1143844.1143934>. [35](#)
- M. Ranzato, F. J. Huang, Y. L. Boureau, and Y. LeCun. Unsupervised Learning of Invariant



- Feature Hierarchies with Applications to Object Recognition. *Computer Vision and Pattern Recognition, IEEE Computer Society Conference on*, 0:1–8, 2007. 35
- M. Richardson and P. Domingos. Markov logic networks. *Mach. Learn.*, 62(1-2):107–136, 2006. ISSN 0885-6125. doi: <http://dx.doi.org/10.1007/s10994-006-5833-1>. 35, 60
- H. B. Richman, J. J. Staszewski, and H. A. Simon. Simulation of expert memory using EPAM IV. *Psychological Review*, pages 305–330, 1995. 24, 34
- B. Roark and M. Bacchiani. Supervised and unsupervised pcfg adaptation to novel domains. In *Proceedings of the 2003 Conference of the North American Chapter of the Association for Computational Linguistics on Human Language Technology - Volume 1*, NAACL '03, pages 126–133, Stroudsburg, PA, USA, 2003. Association for Computational Linguistics. 23
- L. K. Saul, T. Jaakkola, and M. I. Jordan. Mean field theory for sigmoid belief networks. *Journal of Artificial Intelligence Research*, 4:61–76, 1996. 35
- A. Segre. A learning apprentice system for mechanical assembly. In *Proceedings of the Third IEEE Conference on AI for Applications*, pages 112–117, 1987. 34
- P. Y. Simard, D. Steinkraus, and J. C. Platt. Best Practices for Convolutional Neural Networks Applied to Visual Document Analysis. In *ICDAR '03: Proceedings of the Seventh International Conference on Document Analysis and Recognition*, Washington, DC, USA, 2003. IEEE Computer Society. 35
- J. M. Siskind, J. J. Sherman, I. Pollak, M. P. Harper, and C. A. Bouman. Spatial random tree grammars for modeling hierarchical structure in images with regions of arbitrary shape. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 29(9):1504–1519, Sept. 2007. ISSN 0162-8828. 50
- D. H. Sleeman and M. J. Smith. Modeling students' problem solving. *Artificial Intelligence*, 16: 171–187, 1981. 75
- A. Srinivasan. *The Aleph Manual*, 2004. URL <http://www.comlab.ox.ac.uk/activities/machinelearning/Aleph/>. 60
- A. Stolcke. *Bayesian learning of probabilistic language models*. PhD thesis, Berkeley, CA, USA, 1994. 23
- N. A. Taatgen and F. J. Lee. Production compilation: A simple mechanism to model complex skill acquisition. *Human Factors*, 45(1):61–75, 2003. 34, 66, 75
- K. K. Tatsuoka. Rule space: An approach for dealing with misconceptions based on item response theory. *Journal of Educational Measurement*, pages 345–354, 1983. 75
- I. Titov and J. Henderson. Constituent Parsing with Incremental Sigmoid Belief Networks. In *Proceedings of the 45th Annual Meeting of the Association of Computational Linguistics*, pages 632–639, Prague, Czech Republic, June 2007. Association for Computational Linguistics. 35
- L. Torrey, J. Shavlik, T. Walker, and R. Maclin. Relational macros for transfer in reinforcement learning. In *Proceedings of the 17th Conference on Inductive Logic Programming*, Corvallis, Oregon, 2007. 35

- P. E. Utgoff. *Shift of Bias for Inductive Concept Learning*. PhD thesis, Department of Computer Science, Rutgers University, New Brunswick, NJ, 1984. 60
- K. VanLehn. *Mind Bugs: The Origins of Procedural Misconceptions*. MIT Press, Cambridge, MA, USA, 1990. ISBN 0262220369. 75
- K. Vanlehn and W. Ball. A version space approach to learning context-free grammars. *Machine Learning*, 2(1):39–74, Mar. 1987. 23, 50
- K. Vanlehn, S. Ohlsson, and R. Nason. Applications of simulated students: an exploration. *Journal of Artificial Intelligence in Education*, 5:135–175, February 1994. 1, 2, 66
- M. Villano. Probabilistic student models: Bayesian belief networks and knowledge space theory. In *Proceedings of the 2nd International Conference on Intelligent Tutoring Systems*, pages 491–498, Heidelberg, 1992. 75
- J. G. Wolff. Language acquisition, data compression and generalization. *Language and Communication*, 2:57–89, 1982. 23
- R. Wylie, K. Koedinger, and T. Mitamura. Analogies, explanations, and practice: examining how task types affect second language grammar learning. In *Proceedings of the 10th international conference on Intelligent Tutoring Systems - Volume Part I*, ITS’10, pages 214–223, Berlin, Heidelberg, 2010. 62, 64, 66, 70