Advances in Cognitive Systems 2 (2012) 1-18 Submitted 9/2012; published 12/2012

Efficient Complex Skill Acquisition Through Representation Learning

Nan Li NLI1@CS.CMU.EDU
Abraham J. Schreiber ABRAHAMJSCHREIBER@GMAIL.COM
William W. Cohen WCOHEN@CS.CMU.EDU
Kenneth R. Koedinger KOEDINGER@CMU.EDU
School of Computer Science, Carnegie Mellon University, 5000 Forbes Ave., Pittsburgh PA 15213 USA

Abstract

One of the fundamental goals of artificial intelligence is to understand and develop intelligent agents that simulate human-level intelligence. A lot of effort has been made to develop intelligent agents that simulate human learning of math and science, e.g., for use in cognitive tutors. However, constructing such a learning agent currently requires manual encoding of prior domain knowledge for each domain and even for each level of problem difficulty, which hurts the generality of the learning agent and is less cognitively plausible. Li et al. (2012) recently proposed an efficient algorithm that acquires representation knowledge in the form of "deep features," and use the acquired representation to automatically generate feature predicates to assist future learning. The authors demonstrated the generality of the proposed approach across multiple domains. The results showed that by integrating this algorithm into a simulated student, SimStudent, the extended agent achieves efficient skill acquisition, while requiring less prior knowledge engineering effort, and being a more realistic model of the state of prior knowledge of novice algebra students. In this work, we further explore the generality of the proposed approach within one domain, but across multiple difficulty levels. The results indicates that the new, extended SimStudent is able to acquire skill knowledge of harder problems using only its learned problem representations, while the original SimStudent requires its domain-specific prior knowledge to be engineered explicitly to handle these harder problems. The extended SimStudent's performance is shown to match and even exceed the original as the complexity of problems increases.

1. Introduction

One of the fundamental goals of artificial intelligence is to understand and develop intelligent agents that simulate human-like intelligence. A large amount of effort (e.g., Laird, Newell, & Rosenbloom, 1987; Anderson, 1993; Langley & Choi, 2006) has been put toward this challenging task. Further, education in the 21st century will be increasingly about helping students not just to learn content, but also to 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. These goals are inherently interrelated. Our understanding of the way real students learn informs our design of intelligent learning agents, and intelligent agents help improve our teaching methodology by allowing us to examine models of student learning in a controlled setting.

To contribute to both goals, considerable efforts (e.g., Neves, 1985; Anzai & Simon, 1979; Matsuda et al., 2009; Vanlehn, Ohlsson, & Nason, 1994; Langley & Choi, 2006) have been made to develop 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. Such prior knowledge includes encoding how to extract a coefficient from an expression, how to correctly parse a given equation (e.g., $-3x = 6$), and so on. Giving imperfect or insufficient prior knowledge often leads to unsuccessful learning. Having to hard code prior knowledge increases the difficulty of constructing an intelligent agent, since manual encoding of prior knowledge is often time-consuming, error-prone, and not reusable across learning tasks and across domains. It also reduces the cognitive plausibility of the constructed agent, as human students entering a course do not necessarily have substantial domain-specific or domain-relevant prior knowledge. An intelligent agent that requires only domain-independent prior knowledge (across learning tasks, and across domains) would be a great improvement.

Li et al. (2010) have recently reported a learning algorithm that acquires world state representations automatically with only domain-independent knowledge (e.g., what is an integer) as input. Previous work in cognitive science (Chi, Feltovich, & Glaser, 1981; Chase & Simon, 1973) showed that different prior knowledge of world state representation is one of the key factors that differentiates experts and novices in a field. Experts view the world in terms of deep functional features (e.g., coefficient and constant in algebra), while novices only view it in terms of shallow perceptual features (e.g., integer in an expression). While deep features can take many different forms, they often act as hidden layers between the raw input and the solution. More specifically, deep features are derived from raw input or shallower features by a non-trivial computational process (e.g., non-linear functions, complex logic functions), and are then used to calculate the solution through another non-trivial process. They ease the learning task of the system by serving as the bridge between the raw input and the solution. Ideally, deep features are general across problems, and are associated with some interpretable meaning. In contrast, shallow features can often be derived from the raw input relatively easy, but may make the learning process from shallow features to the solution much harder than the deep features do.

Therefore, learning and effectively using such "deep features" are essential in modeling human-like intelligence. Li et al. (2012) made use of the acquired representation knowledge to automatically generate feature predicates, and integrated this deep feature learner into a learning agent, SimStudent (Matsuda et al., 2009). The authors evaluated the "breadth" of the proposed approach in three domains: fraction addition, equation solving, and stoichiometry. Results show that the new SimStudent requires a much smaller amount of prior knowledge encoding effort than the old SimStudent, while maintaining as good or better performance. Moreover, the extended SimStudent can be used to discover student models that fit with human student data better than the ones found by experts (Li et al., 2011).

In this paper, we further evaluate the "depth" of the proposed approach, by training SimStudent on sequences of problems of increasing difficulty. The main claim of the paper 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)

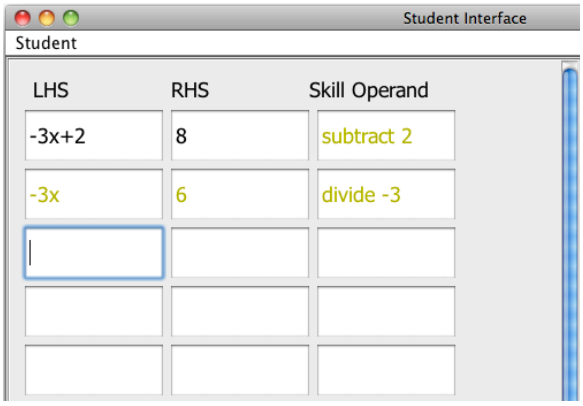

Figure 1. The interface through which SimStudent is tutored in an equation solving domain.

maintain equally good performance, compared with human-engineered intelligent agents. Problem representations/eatures 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.

2. A Brief Review of SimStudent

Before detailing how the learned deep feature representation is incorporated into SimStudent, let's first review SimStudent's basic architecture. SimStudent is an intelligent agent that inductively learns skills to solve problems from demonstrated solutions and problem solving experience. It is an extension of programming by demonstration (Lau & Weld, 1998) using inductive logic programming (Muggleton & de Raedt, 1994) as underlying learning techniques. Figure 1 shows a screenshot of the interface used to tutor SimStudent to solve algebra equations.

2.1 Knowledge Representation

Skill knowledge in SimStudent is represented as a set of production rules. Figure 2 shows an example of a production rule learned by SimStudent in a readable format.[1] There are three parts in a production rule: the perceptual information part (***where*** to obtain the data needed), the precondition part (***when*** to apply the skill), and the operator sequence part (***how*** to perform the skill). The rule to "divide both sides of $-3x = 6$ by -3," shown at the left side of Figure 2, would be read as "given a left-hand side (i.e., $-3x$) and a right-hand side (6) of the equation, when the left-hand side does

1. Actual production rules are implemented using Jess, a rule engine for Java.

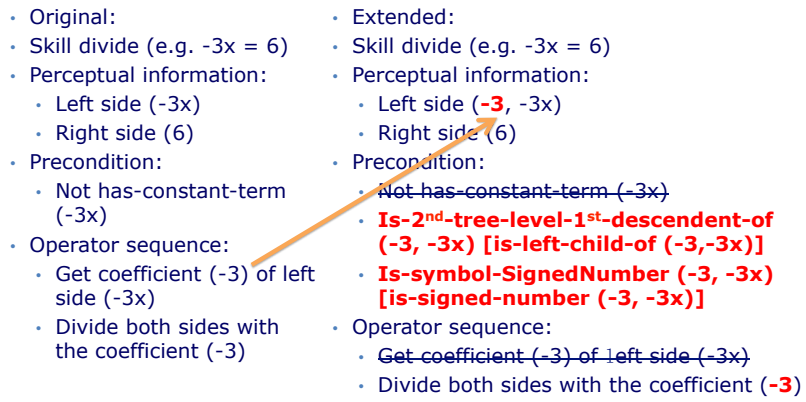

Figure 2. Original and extended production rules for the divide operation.

not have a constant term, get the coefficient of the term at the left-hand side and divide both sides by the coefficient." The ***where*** and ***when*** portions of a rule, taken together, make up the if-part of the rule, while the ***how*** portion makes up the then-part. On the right in Figure 2 is a functionally similar production rule acquired with the addition of perceptual representation learning (described in Section 4.1), and feature predicate learning (in Section 4.2).

2.2 Learning by Tutoring

SimStudent learns its production rules in much the same way a human student learns to solve problems along side a tutor who assists them. Table 1 describes SimStudent's algorithm for learning by tutoring in this fashion.

For each problem step (e.g., $-3x = 6$), SimStudent first tries the problem step itself as a real student would, before requesting assistance. Based on the skill knowledge it has acquired so far, it proposes a next step if possible. If SimStudent finds a next action and the tutor gives positive feedback that it is correct, SimStudent's existing skill knowledge is reinforced using this new example and SimStudent continues on with the problem. If the proposed next action is not correct and the tutor gives negative feedback, SimStudent integrates this negative example into it's knowledge of the skill which it applied incorrectly. If all SimStudent's attempts are wrong or it cannot find any applicable skill and it doesn't know what to do, it enlists help from a tutor.[2] The tutor demonstrates how to apply a skill by providing the correct next action and indicating where SimStudent should focus its attention for information relevant to this skill. SimStudent then updates its skill knowledge to incorporate this new training example, which may be a new example of a skill SimStudent has already begun learning or a new skill entirely.

2. Although other feedback mechanisms are also possible, in our case, the feedback is given by automatic tutors, that have been used to teach real students.

Table 1. SimStudent's algorithm for learning by tutoring.

1. Let R be the existing set of rules. Let $R' \subseteq R$ be the set of rules which are applicable to the current problem state.
2. Until a correct rule application is found or we have run out of rule applications, attempt an application of a rule $r \in R'$.
   - 2.1 Let V be the set of values in the focus of attention in the current application of rule r.
   - 2.2 If the current application of rule r produces a correct next step, add V as a positive example of a tuple used in applying rule r. For each $r^* \in R$ s.t. $r^* \neq r$, add V as a negative example of a tuple used in applying rule r^*, iff $|V|$ is the same as the cardinality of example tuples of r^*.
   - 2.3 If r produces an incorrect next step, add V as a negative example of a tuple used in rule r.
   - 2.4 Update the if-part of rules with new positive or negative examples.
3. If a rule is correct, proceed to the next problem step by applying the correct rule.
4. If a rule is incorrect, request and input an example of a correct next step.
   - 4.1 Let I be the demonstrated correct step. Let S be the name of the problem-solving skill demonstrated in I. Let V be the set of values from the focus of attention in I.
   - 4.2 Let $R_s \subseteq R$ be existing rules for skill S. Until a rule is successfully updated or the agent runs out of rules for skill S, attempt to update both the if-part and then-part of $r_{s_i} \in R_s$ so that it is applicable to the current problem state and when applied, produces the result demonstrated in I.
     - 4.2.1 If rule r_{s_i} is updated successfully, add V as a positive example of a tuple used in applying rule r_{s_i}. For each $r^* \in R$ s.t. $r^* \neq r_{s_i}$, add V as a negative example of a tuple used in applying rule r^*, iff $|V|$ is the same as the cardinality of example tuples of r^*.
   - 4.3 If no rule has been successfully updated, create a new rule $r_{s_{n+1}}$ using the information from instruction I, where n is the number of existing rules for skill S. For each $r^* \in R$, add V as a negative example of a tuple used in applying rule r^*, iff $|V|$ is the same as the cardinality of example tuples of r^*.
   - 4.4 Add $r_{s_{n+1}}$ to the set of of rules R and proceed to the next problem step as shown by instruction I and as results by applying the new rule $r_{s_{n+1}}$.

2.3 Learning Mechanisms

Each of the three portions of SimStudent's rules have their own learning mechanisms, which are used together to create and update rules. Though the algorithms themselves operate independently, their effectiveness is bolstered by their use together, as detailed within their respective sections.

2.3.1 Perceptual Learning

The "where" learner acquires the perceptual information portion of a rule by finding paths which identify useful information in the GUI. These pieces of useful information, *percepts*, are observed within the GUI *elements*, such as cells/textboxes. Elements of a problem, and its associated user-interface, of which the system is aware, are called *working memory elements*. These elements are organized in a hierarchical, tree structure with the problem at the root. Within the hierarchy, there reside more specific elements of the user interface used in the problem domain. For example, the

table node has columns as children, and each column has multiple cells as children. Each element is *covered* by a set of paths ranging from specific to general. For instance, consider a cell in the second row of the first column, *Cell 21*. The possible paths to *Cell 21* are: (1) the exact path to the cell, P_{Cell21}, (2) the generalized path to any cell in row 2 or column 1, $P_{Cell2?}$ or $P_{Cell?1}$, and (3) the most general path to any cell in the table, $P_{Cell??}$.

Because working memory elements are organized in a tree format, for all working memory elements there exists a single, specific path from the root to that element. Any such path can then be obtained via a straight-forward search procedure. A generalized path to multiple working memory elements can be expressed using multivariables whose names are prefixed by a '$'. For example, "?var1 <- (table (columns $?m1 ?var2 $?))" denotes that ?var1 may be any column in the table. This is accomplished using the named multivariable $?m1 and anonymous multivariable $?, each of which may match any number of columns, including none at all. So to describe that ?var2 is any column in the table, we say that ?var2 is a column that is preceded by any number of columns and followed by any number of columns.

Each training example provides a list of GUI elements that are useful in generating the next action. For example, *(Cell 21, Cell 22)* is a list of cells from one training example for the skill "divide". The learning process for locating working memory elements proceeds from specific to general. The learner uses a brute-force depth-first search algorithm to find the most specific paths that cover all training examples. If we have received three training examples of skill "divide", *(Cell 21, Cell 22)*, *(Cell 11, Cell 12)*, and *(Cell 51, Cell 52)*, the most specific paths that cover these training examples are *($P_{Cell?1}$ and $P_{Cell?2}$)*. Each exactly specifies one portion of the location, either the column or the row, while expressing that the other portion of the location could be anywhere. While the ability to generalize the location of a cell as anywhere within a column or row is powerful, there are other types of generalizations pertaining to location that are not directly handled using multivariabes. For instance, expressing that two cells are in different rows is not expressed in this way. For restrictions such as these, the "when" portion of a rule compliments the "where."

2.3.2 Precondition Learning

The "when" section of a rule describes a set of preconditions to be met indicating that this is an appropriate time to apply the rule. Conditions for applying a rule are expressed in terms of *feature predicates*. Each feature predicate is a boolean function that describes relations among objects in the domain or among elements of the GUI. For example, *(has-coefficient $-3x$)* means $-3x$ has a coefficient. Similarly, *(same-row ?var1 ?var2)* means that ?var1 and ?var2 are cells in the table which occur in the same row, without requiring them to be in any specific row. If the paths to cells ?var1 and ?var2 are $P_{Cell?2}$ and $P_{Cell?3}$, as found by the perceptual learner, then ?var1 and ?var2 must be in the second and third columns and in the same row. This is an example in which the "when" learner strengthens the "where" learner.

The precondition learner employs an inductive logic programming system, FOIL (Quinlan, 1990) to acquire a set of features that describe the desired situation in which to fire the production rule. FOIL is a concept learner that acquires Horn clauses that separate positive examples from negative examples. For instance, *(precondition-divide ?percept$_1$?percept$_2$)* is the precondition predicate to be learned for the production rule named "divide." *(precondition-divide -3x 6)* is a positive

example since when we have $-3x$ on one side of the equation and *6* on the other side, we would like to divide both sides by -3. *(precondition-divide* $2x + 4$ 6*)* is a negative example since when we have $2x + 4$ on one side of the equation and *6* on the other, we would like to subtract *4*. For all values that have appeared in the training examples (e.g., $-3x$, 6, $2x + 4$), we test the truthfulness of the feature predicates given all possible permutations of the observed values. For the feature predicate *(has-coefficient ?val0)*, *(has-coefficient* $-3x$*)* is true, and *(has-coefficient* $2x + 4$*)* is false. Given these inputs, FOIL will acquire a set of clauses formed by feature predicates describing the precondition. In the case of skill "divide," the feature test learned is *(not (has-constant-term ?val0))*. The "when" learning process proceeds from general to specific. FOIL starts from an empty feature test set, and grows the test set gradually until all of the training examples have been covered.

2.3.3 Operator Sequence Learning

The "how" learner is given a set of basic transformations (e.g., add two numbers) called *operator functions* that can be applied to the problem. If the conditions of the "when" part are met by the percepts identified in the "where" part, the "how" part seeks to find a sequence of operator functions that generates the correct next step. For each training example T_i, the learner takes the percepts, *percepts*$_i$, as the input, and the step, *step*$_i$, as the output. We say an operator function sequence *explains* a percepts-step pair, $\langle$*percepts*$_i$*, step*$_i\rangle$, if the system takes *percepts*$_i$ as input and yields *step*$_i$ after applying the operator function sequence. The operator function sequence *(coefficient* $-3x$ *?coef) (divide ?coef)* is a possible explanation for $\langle(-3x,\ 6)$*, (divide* $-3)\rangle$. Given all training examples for some skill, 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.

Should the operator sequence learner fail to find a single operator sequence which explains all examples, a new rule is created to handle those examples not explained by the operator sequence used previously. Not all operator functions can be usefully applied to all inputs. Consider an operator *get-variable* which returns the variable within the equation provided as input. Applying this variable to an input which does not contain a variable, such as *(get-variable* 12*)* makes little sense. Restrictions placed on the types of inputs to which operators can be applied to limit the space of operator sequences which must be searched. These restrictions also augment both the "where" and "when" learners because although a particular set of precepts may be located by the "where" portion of a rule and meet the conditions of the "when" portions, the precepts may not be applicable to a given operator sequence.

Note that operator functions can be divided into two groups, domain-independent operator functions and domain-specific operator functions. Domain-independent operator functions are basic skills used across multiple domains (e.g., adding two numbers, *(add* 1 2*)*, copying a string, *(copy* $-3x$*)*). Human students usually have knowledge of these simple skills prior to class. Domain-specific operator functions, on the other hand, are more complicated skills, such as getting the coefficient of a term, *(coefficient* $-3x$*)* and adding two terms, *(add-term* $5x-5$ 5*)*. Human students may not have enough domain expertise to perform these operations prior to taking a class in the domain. As we will see later, by integrating deep feature learning into SimStudent, the learning agent is able to achieve comparable performance without domain-specific operator functions.

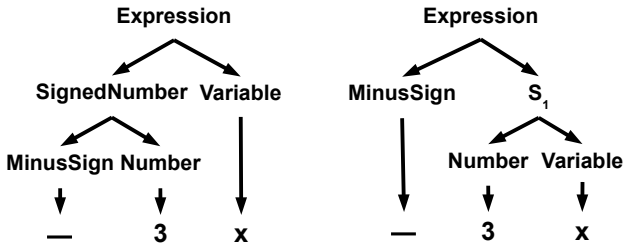

Figure 3. Correct (left) and incorrect (right) parse trees for the expression $-3x$.

3. Deep Feature Learning as Grammar Induction

Li et al. (2010) analyzed the nature of deep feature learning in algebra equation solving, and found that it could be modeled as a problem of inducing a grammar from a set of observed data (e.g. equations in algebra). As shown at the left side of Figure 3, the coefficient of $-3x$ can be identified by extracting the signed number before a variable in the parse tree. Table 2 shows a context-free grammar that generates the parse tree.[3] The deep feature "coefficient" then becomes a non-terminal symbol in the grammar rule used to produce this parse. Viewing feature learning tasks as a grammar induction problem also explains many of the causes of student errors. For example, if the learning algorithm incorrectly learned the parse tree shown the right side of Figure 3, it will consider *3* as the coefficient, which is one of the most frequently observed errors in human student data.

To support deep feature learning, Li et al. extended an existing probabilistic context-free grammar (pCFG) learner (Li, Kambhampati, & Yoon, 2009). Specifically, this learner implemented a variant of the *inside-outside algorithm* (Lari & Young, 1990). The input to the pCFG learner is a set of observation sequences. Each sequence is a string of characters obtained directly from user input (e.g., $-3x$). The output is a pCFG that can generate all 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.

For example, as shown in Figure 4, the pCFG learner is given three examples, 2, -5, and $-3x$, and it knows that 2, 3, and 5 are numbers. In the first step, the structure hypothesizer finds that the ⟨*MinusSign, Number*⟩ pair appears more often than the ⟨*Number, Variable*⟩ pair. It creates a rule that reduces an automatically-generated non-terminal symbol, *SignedNumber*, into *MinusSign* and *Number*. This procedure continues until every training example has at least one parse tree, as shown in Figure 4. The hypothesized grammar rules as presented in Figure 5 are then sent to the Viterbi training step, where the probabilities associated with grammar rules are refined, and redundant grammar rules are removed.

Learning the pCFG alone is not enough, however. Deep features must be associated with symbol-rule pairs in a learned grammar, which upon initial construction contains only arbitrary symbols. In order to accurately associate deep features, the system is given pairs of values, such as $\langle -3x, -3 \rangle$,

3. The nonterminal names were manually added here to make the discussion more readable. These would normally be arbitrary identifiers generated by the grammar induction algorithm until assigned a deep feature label later.

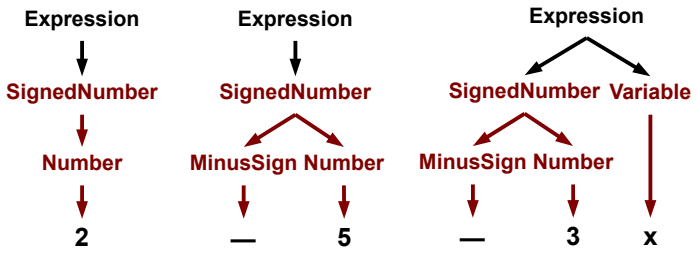

Figure 4. Candidate parse trees constructed during learning in algebra.

which are examples of a particular deep feature. Here we are considering the deep feature *coefficient*. The first element is an example sequence in which the second element should be considered a coefficient. Parse trees of the observation sequences used to produce the grammar are examined to locate the symbol most often associated with the deep feature and the rule in which the symbol occurs. For instance, if the most input records match with *SignedNumber* in *Expression* $\rightarrow 0.33$*, SignedNumber Variable*, this symbol-rule pair will be considered as the target feature pattern.

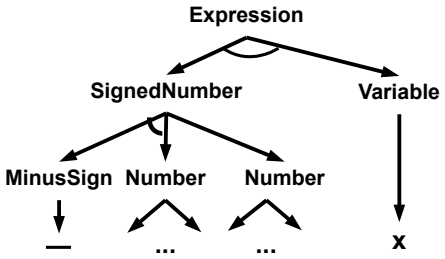

Figure 5. Example context free grammar constructed during learning in algebra.

Simple grammars incorporating deep features, constructed in this way, can be used as building blocks in the creation of more complex grammars, transferring the knowledge of the simpler grammar to the more complex one. Given a previously constructed grammar for signed numbers, we construct a grammar for simple algebra problems in the following manner. First, partial parses of the set of observation sequences for algebra are produced using the grammar for signed numbers. If $-3x$ were one of our observation sequences for algebra, we would be able to obtain a partial parse in *(SignedNumber x)* as shown in red in Figure 4. The existing grammar for signed numbers successfully parses -3. Each observation sequence for simple algebra is processed this way. The original algorithm used in producing the pCFG is then run on the new sequences after having been reduced by the partial parse. The result of this subsequent pCFG construction is a grammar for simple algebra problems that includes our previously acquired grammar for signed numbers as a subgrammar as shown in Table 2. Analogously, a grammar for complex algebra problems is produced from one for simple problems. In this way, a grammar for complex algebra problems is incrementally built up from one for numbers and simpler problems.

Table 2. 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: $Expression, SignedNumber,$ $Variable, MinusSign, Number$;
$Expression \rightarrow 0.33,\ [SignedNumber]\ Variable$
$Expression \rightarrow 0.67,\ SignedNumber$
$Variable \rightarrow 1.0,\ x$
$SignedNumber \rightarrow 0.5,\ MinusSign\ Number$
$SignedNumber \rightarrow 0.5,\ Number$
$Number \rightarrow 0.091,\ Number\ Number$
$Number \rightarrow 0.091,\ 0$
$Number \rightarrow 0.091,\ 1$
$Number \rightarrow 0.091,\ 2$
$Number \rightarrow 0.091,\ 3$
$Number \rightarrow 0.091,\ 4$
$Number \rightarrow 0.091,\ 5$
$Number \rightarrow 0.091,\ 6$
$Number \rightarrow 0.091,\ 7$
$Number \rightarrow 0.091,\ 8$
$Number \rightarrow 0.091,\ 9$
$MinusSign \rightarrow 1.0,\ -$

4. SimStudent with Integrated Deep Features

Having built the deep feature learner and used it to acquire a grammar for algebra problems, we now have a means of extending greatly the problem representations available to SimStudent. Algebra problems, and portions thereof, may be converted from string representations to parse tree representations using a parser and our learned grammar. These parse trees provide a "problem topology" which can be incorporated into SimStudent's perceptual learner, as we will explain later. The problem topology can be further used to automatically generate feature predicates. This automatic generation could eliminate the need for authors/developers to hand code complex domain-specific predicates like *(has-constant-term ?var0)* (see precondition in Figure 2).When parse trees produced by the grammar are incorporated into SimStudent, previous work has shown that the number and complexity of operators necessary for skill acquisition is greatly reduced (Li, Cohen, & Koedinger, 2012). Integrating the feature learner into SimStudent reduces the amount of prior knowledge engineering needed for SimStudent and results in a more psychologically plausible model of human students.

4.1 Extended Perceptual Learning

As mentioned in the perceptual learning section of the description of the original SimStudent, GUI elements are organized into a tree structure with individual cells or textboxes at the leaf nodes. The entire contents of a cell at a leaf node is the smallest grained working memory element of the orig-

inal SimStudent. Interaction with the contents of an individual cell occurs only within hand-coded, predefined operators or feature predicates. To incorporate the problem representations acquired by the deep feature learner into SimStudent, we extended the memory element hierarchy. The parse trees for the contents of each cell is attached to the working memory element of the leaf nodes (cells).

This augmentation of SimStudent's working memory brings with it some design challenges. In the original SimStudent, the number and location of working memory elements are fixed throughout an individual problem and even across multiple problems in the same domain. That is, the textboxes are in the same location regardless of where you are within a problem or of the problem on which you are working. In extended SimStudent, working memory elements are added dynamically during the problem solving process and their locations may vary. The path to an individual node in a parse tree is of variable length, for example, while the path to a textbox is fixed. SimStudent's perceptual learner, which generalizes paths to working memory elements, is extended to generalize paths to individual nodes within a parse tree.

Figure 2 shows a comparison between production rules acquired by the original SimStudent and the extended SimStudent. As we can see, the coefficient of the left-hand side (i.e., -3) is included in the perceptual information portion of the extended production rule. This reflects the way someone familiar with algebra is able to identify a coefficient by evaluating *where* it is in the problem.

4.2 Extended Precondition Learning

The structure and labels provided by a deep-feature problem representation provide a wealth of information about the state of the problem. Further, this information is abstract in that the type of information (the tree structure and nonterminal symbols corresponding to features) is applicable to any domain which we can accurately model using a context free grammar. It is natural to think that these deep features can be used in describing desired situations to fire a production rule, the *feature predicates*. In a classroom setting, it is exactly because these deep features are useful in the problem solving process that they are taught to the students initially. Deep features are thus used to learn the "when" part of a rule as well. More specifically, using domain-specific information encapsulated in the deep features of the grammar, we automatically generate a set of predicates that can be used by the inductive logic programming component. These automatically generated feature predicates replace manually constructed ones.

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 and assists the inductive logic subcomponent by explicitly providing these combinations. Each of these types of predicates are applicable to a general context-free grammar and the parse trees it generates.

4.2.1 Topological Feature Predicates

Topological feature predicates examine the location of nodes in a parse tree. To an extent, these locations are learned by the perceptual learner, but as in the case of the *(same-row ?var1 ?var2)* predicate in original SimStudent, these predicates compliment the perceptual leaner by covering topological

conditions not directly handled. 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. A separate, automatically-generated predicate is created for each *m* between *0* and *m-1* where *m* is the maximum number of non-terminal symbols on the right side of the rules in the grammar and for each *n*, where *n* is the maximum height of the parse trees encountered.

4.2.2 Nonterminal Symbol Feature Predicates

The second set of automatically generated 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 2. 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 symbol in the grammar.

4.2.3 Parse-Tree Relation Feature Predicates

The third class of feature predicates, which focus on parse-tree relations, examine *both* the positions of nodes in the tree *and* their associated symbols. These allow SimStudent to examine the nodes surrounding the focus of attention in the parse tree and determine if they have a particular symbol from the grammar associated with them. A generic templates of these predicates is *(i-j-relation-is-symbol-x ?val0 ?val1)*. This indicates that the nodes are reached by moving up *i* times in the tree and then down *j* times from the focus of attention in order to determine whether a node at that location has a particular symbol.

4.2.4 An Example

The extended SimStudent can make use of such automatically learned feature predicates to acquire preconditions of the production rules. As shown in Figure 2, the precondition learned by the original SimStudent given domain-specific feature predicates is *(not (has-constant-term ?var0))*. With the automatically generated feature predicates, the extended SimStudent instead, uses one topological feature predicate (i.e., *(is-2nd-tree-level-1st-descendent-of ?var0 var1)*), and one nonterminal symbol feature predicates (i.e., *(is-symbol-SignedNumber ?var0 ?var 1)*), and replaces the original precondition with the combination of these two automatically learned feature predicates. This combination of automatically generated predicates actually represents a more general precondition for the skill "divide" than *not (has-constant-term ?var0)*. Consider the slightly more complicated equation $-3(x+4)=6$. There are two promising next steps to take given this equation: "distribute the -3" to obtain $-3x-12=6$ or "divide by -3" to obtain $x+4=-2$. $(x+4)$ is located in the same position in the parse tree for $-3(x+4)$ as x is in the parse tree for $-3x$, and -3 is identified as a signed number in each. Because the automatically generated predicates can identify these simi-

larities, skill knowledge gained from solving simple problems can be generalized and transferred to more complex problems.

4.3 Learning Extended Operator Sequences

With the parse tree representations of the problem inserted into SimStudent's working memory, we have more individual elements which may be used in a possible operator sequence. In the original SimStudent, the values available to the operator sequence search are limited to the set of values in the cells specified as the focus of attention. In the extended SimStudent, the operator sequence search has available to it not only the values in the cells, but also all values represented in the parse trees produced from the values in the cells. This means that there are far more possible combinations of values to consider in applying operators.

The addition of finer grained values lets the operator sequence learner find shorter and less domain-specific operator sequences which still produce the desired result. Operator sequences in extended SimStudent often operate directly on values below the root of the parse tree for the cell in which they reside. In the original SimStudent, a domain-specific operator is required to obtain this value. A example would be the *(coefficient $-3x$)* operator in Figure 2. It is unnecessary to have an operator obtain -3 from $-3x$ when -3 is directly represented in working memory. This value also provides an additional, more specific piece of information about the focus of attention. When the operator sequence learner identifies a value in the parse tree which is used in the operator sequence for a rule, the path to this value is also encoded in the "where" portion of the rule, along with the paths to the cells themselves. The operator sequence learner is thus used to strengthen the perceptual learner.

5. Experimental Study

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 carried out an experiment in algebra equation solving. Original and extended SimStudents were trained on sequences of increasingly difficult algebra problems and their performance was compared.

5.1 Experimental Design

Two extended versions of SimStudent were 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 deep feature learner was 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 was then incorporated into SimStudent, as described in the section on SimStudent with integrated deep features. The two extensions were compared with an original SimStudent which was 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.

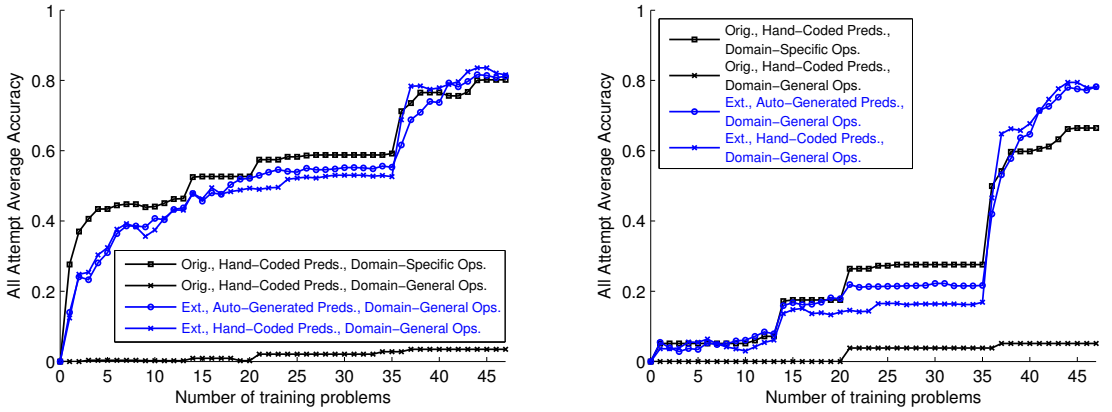

Figure 6. Learning curves of SimStudents in equation solving measured by a) all test problems, b) hard problems (category 4) only, using all attempt accuracy.

All versions were tutored using an automatic tutor, CTAT (Aleven et al., 2009), which was used by 71 human students in a classroom study. Four training sets, each consisting of 47 problems were constructed for use in teaching the SimStudents. Each training set consisted of problems in 4 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, was constructed for use in evaluation of performance. Problems for both the training and test sets were likewise obtained from actual classroom studies.

Measurements: We assessed the accuracy of the SimStudents' skill acquisition by measuring each SimStudent's *first attempt accuracy* and *all attempt accuracy* 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. All attempt accuracy is the 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 counted the number of lines of Java code used in the implementation.[4] There are two locations where this information is used, the operators and the feature predicates. Each were measured separately and reported as such.

4. 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.

5.2 Learning Curves

Average learning curves for all attempt accuracy are shown in Figure 6(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 achieved 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 was 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.

5.3 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(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 causes 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 $-3(x + 4) = 6$ in Section 4.2.4, 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.

5.4 Knowledge Engineering Effort

We also compared 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. (2012) 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 found that this approach completely remove the prior 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.

6. Related Work

The primary contribution of this paper is that the representation learning and integration procedure reduces the amount of knowledge engineering required in constructing such human-like intelligent agents, and can be extended to harder problems without extra knowledge engineering. Previous work has shown that "chunking" is an important component of human knowledge acquisition. Theories of the chunking mechanisms (e.g., Richman, Staszewski, & Simon, 1995; Gobet & Simon, 2000) have been constructed. Our work is similar to these works as we are also modeling the learning of perceptual chunks, a kind of deep feature learning, but differs from these theories since none of the above theories reuse chunks in later learning as we did in this work.

There has been considerable research on learning within agent architectures, such as Soar (Laird, Rosenbloom, & Newell, 1986), ACT-R (Taatgen & Lee, 2003), and ICARUS (Langley & Choi, 2006). SimStudent is similar to these architectures in that it also models aspects of the learning process in intelligent agents. Unlike those theories, SimStudent puts more emphasis on knowledge-level learning (cf., Dietterich, 1986) achieved through induction from positive and negative examples. It is inspired by theories of perceptual chunking, and uses grammar induction techniques to improve 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 some algorithms in explanation-based learning (Segre, 1987; Nejati, Langley, & Konik, 2006; Li et al., 2009), learning apprentices (Mitchell, Mahadevan, & Steinberg, 1985) and programming by demonstration (Cypher et al., 1993; Lau & Weld, 1998). Most of these approaches used analytic methods to acquire candidate procedures. However, to the best of our knowledge, none of the above approaches uses the transfer learner to acquire a better representation that reveals essential perceptual features and to integrate it into an intelligent agent. Ohlsson (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 systems. In summary, our system incorporates ideas from a number of earlier lines of research, and combines them in a novel way that produces more effective learning in procedural domains.

7. Concluding Remarks

To sum up, building an intelligent agent that simulates human-level learning is an essential task in AI and education, but building such systems often requires manual encoding of prior domain knowledge. Previous effort has shown that by integrating a deep feature learning algorithm into an intelligent agent, SimStudent, as an extension of the perception module, the extended SimStudent is able to achieve comparable performance without requiring any domain-specific operator function as input. In this paper, we further evaluated the "depth" of the proposed approach by training SimStudent with problems of increasing complexity. Results show that given a reasonable number (e.g., 50) of training examples, the extended SimStudent learns as well as the original SimStudent while not requiring any human-engineered domain-specific prior knowledge.

Until now, we have mainly evaluated our approach in algebra, but the proposed approach is not limited to this domain. To further explore how representation learning affects skill learning, we

plan to carry out similar experiments in other domains such as fraction addition, stoichiometry, and second language learning, and test to see whether the same results occur. Moreover, the precondition learner of SimStudent now uses two separate strategies, an unsupervised module (i.e., deep feature learning) and a supervised module (i.e., FOIL). We intend to explore whether a single approach such as deep belief networks (Hinton, 2007) instead would learn faster than the disjoint strategy. Finally, since the prior knowledge engineering required is reduced, we potentially have a more psychologically plausible learning agent. To contribute to learning sciences, we would like to use the extended SimStudent to help us understand human learning better.

References

Aleven, V., Mclaren, B. M., Sewall, J., & Koedinger, K. R. (2009). A new paradigm for intelligent tutoring systems: Example-tracing tutors. *International Journal of Artificial Intelligence in Education*, *19*, 105–154.

Anderson, J. R. (1993). *Rules of the Mind*. Hillsdale, NJ: Lawrence Erlbaum.

Anzai, Y., & Simon, H. A. (1979). The theory of learning by doing. *Psychological Review*, *86*, 124–140.

Chase, W. G., & Simon, H. A. (1973). Perception in chess. *Cognitive Psychology*, *4*, 55–81.

Chi, M. T. H., Feltovich, P. J., & Glaser, R. (1981). Categorization and representation of physics problems by experts and novices. *Cognitive Science*, *5*, 121–152.

Cypher, A., Halbert, D. C., Kurlander, D., Lieberman, H., Maulsby, D., Myers, B. A., & Turransky, A. (Eds.). (1993). *Watch what I do: Programming by demonstration*. Cambridge, MA: MIT Press.

Dietterich, T. G. (1986). Learning at the knowledge level. *Machine Learning*, *1*, 287–315.

Gobet, F., & Simon, H. A. (2000). Five seconds or sixty? Presentation time in expert memory. *Cognitive Science*, *24*, 651–682.

Hinton, G. E. (2007). To recognize shapes, first learn to generate images. *Progress in brain research*, *165*, 535–547.

Laird, J. E., Newell, A., & Rosenbloom, P. S. (1987). SOAR: An architecture for general intelligence. *Artificial Intelligence*, *33*, 1–64.

Laird, J. E., Rosenbloom, P. S., & Newell, A. (1986). Chunking in Soar: The anatomy of a general learning mechanism. *Machine Learning*, *1*, 11–46.

Langley, P., & Choi, D. (2006). A unified cognitive architecture for physical agents. *Proceedings of the Twenty-First National Conference on Artificial Intelligence* (pp. 1469–1474). Boston, MA: AAAI Press.

Lari, K., & Young, S. J. (1990). The estimation of stochastic context-free grammars using the inside-outside algorithm. *Computer Speech and Language*, *4*, 35–56.

Lau, T., & Weld, D. S. (1998). Programming by demonstration: An inductive learning formulation. *Proceedings of the 1999 International Conference on Intelligence User Interfaces* (pp. 145–152). New York: ACM.

Li, N., Cohen, W. W., & Koedinger, K. R. (2010). A computational model of accelerated future learning through feature recognition. *Proceedings of Tenth International Conference on Intelligent Tutoring Systems* (pp. 368–370). Pittsburgh: Springer-Verlag.

Li, N., Cohen, W. W., & Koedinger, K. R. (2012). Efficient cross-domain learning of complex skills. *Proceedings of the Eleventh International Conference on Intelligent Tutoring Systems* (pp. 493–498). Berlin: Springer-Verlag.

Li, N., Cohen, W. W., Matsuda, N., & Koedinger, K. R. (2011). A machine learning approach for automatic student model discovery. *Proceedings of the Fourth International Conference on Educational Data Mining* (pp. 31–40). Eindhoven, Netherlands: www.educationaldatamining.org.

Li, N., Kambhampati, S., & Yoon, S. (2009). Learning probabilistic hierarchical task networks to capture user preferences. *Proceedings of the Twenty-First International Joint Conference on Artificial Intelligence* (pp. 1754–1759). Pasadena, CA: Morgan Kaufmann Publishers Inc.

Li, N., Stracuzzi, D. J., Langley, P., & Nejati, N. (2009). Learning hierarchical skills from problem solutions using means-ends analysis. *Proceedings of the Annual Meeting of the Cognitive Science Society*. Amsterdam, Netherlands.

Matsuda, N., Lee, A., Cohen, W. W., & Koedinger, K. R. (2009). A computational model of how learner errors arise from weak prior knowledge. *Proceedings of the Annual Meeting of the Cognitive Science Society* (pp. 1288–1293). Austin, TX.

Mitchell, T. M., Mahadevan, S., & Steinberg, L. I. (1985). LEAP: A learning apprentice for VLSI design. *Proceedings of the Ninth International Joint Conference on Artificial Intelligence* (pp. 573–580). San Francisco, CA.

Muggleton, S., & de Raedt, L. (1994). Inductive logic programming: Theory and methods. *Journal of Logic Programming*, *19*, 629–679.

Nejati, N., Langley, P. W., & Konik, T. (2006). Learning hierarchical task networks by observation. *Proceedings of the 23rd International Conference on Machine Learning*. Pittsburgh, PA.

Neves, D. M. (1985). Learning procedures from examples and by doing. *Proceedings of the Ninth International Joint Conference on Artificial Intelligence* (pp. 624–630). San Francisco: Morgan Kaufmann.

Ohlsson, S. (2008). *Computational models of skill acquisition*. Cambrige, UK: Cambridge University Press.

Quinlan, J. R. (1990). Learning logical definitions from relations. *Machine Learning*, *5*, 239–266.

Richman, H., Staszewski, J., & Simon, H. (1995). Simulation of expert memory using EPAM IV. *Psychological Review*, *102*, 305–330.

Segre, A. (1987). A learning apprentice system for mechanical assembly. *Proceedings of the Third IEEE Conference on AI for Applications* (pp. 112–117). Orlando, FL.

Taatgen, N. A., & Lee, F. J. (2003). Production compilation: A simple mechanism to model complex skill acquisition. *Human Factors*, *45*, 61–75.

Vanlehn, K., Ohlsson, S., & Nason, R. (1994). Applications of simulated students: An exploration. *Journal of Artificial Intelligence in Education*, *5*, 135–175.