Automated Learning from Textbooks

Paul E. Utgoff
Department of Computer Science
University of Massachusetts
Amherst, MA 01003 U.S.A.

Technical Report 05-41
June 22, 2005

Keywords: machine learning, many-layered learning, building block, representation, prerequisite, textbook, ill-posed learning task, LFR, memory item, linear algebra, coherent knowledge.

Abstract

The project is motivated by the need to develop methods of automated learning that are capable of progressively learning simple building blocks of knowledge for which one building block may be a prerequisite to one or more others. To this end, a computer program LFR will be constructed that can read a suitably written (massaged) textbook, and comprehend its content, as measured by the program's ability to apply its knowledge correctly to practice exercises and to other problems posed to it. A textbook is a marvelous exposition of coherent information, in which the simple building blocks of related knowledge are arranged in a favorable order so that all learning is simple when it is encountered. This new modality for automated learning mechanisms is implementable now, due to two critical design choices that enable mapping sequential sentences of the text to procedure calls that effect incremental change in the learner's knowledge state. The project will produce a critical advance in how machines learn and acquire useful expertise.

1 Introduction

Humans learn by a variety of means, not all of which are known or understood. For both intellectual and practical reasons, we need to challenge ourselves to explore how to model and integrate these various forms of learning. We know that humans acquire perceptual, motor, language, and cognitive skills. Commonly acknowledged modes of learning include learning from practice, from experience, from observation, from advice, from instruction, from being told, from reading, from experimentation, and from analysis. Indeed, any source of new information provides grist for learning. How can a wealth of such stimulation lead to the vast repositories of knowledge and skills that we each build incrementally over a lifespan of many decades? How can the processes that create such repositories be modeled?

Of central importance is to invent mechanisms for modeling learning that transpires over long periods, as measured in decades. Most present-day methods are designed to solve small single stand-alone learning tasks. While such methods and their solutions serve a useful purpose, how can we produce still stronger methods that require solving myriad learning tasks in a progressive manner? Much of human education and skill acquisition is organized around the fundamental notion of prerequisites and building blocks of knowledge or skill. Can we build a learning mechanism that thrives in such a world? The answer is 'yes,' and the purpose of the proposed project is to demonstrate that it is feasible to build practical systems with such capabilities.

2 The Need for Indefinitely Many Layers of Learning

Clark and Thornton (1997) provide a highly useful dichotomy for learning problems. For a type-1 problem, a direct statistical association exists between the provided input and output representations. More specifically, the individual outputs are statistically dependent on one or more of the individual inputs. For a type-2 problem however, no such direct association exists, requiring that one or more new intermediate representations be inserted between the inputs and outputs. For type-2 learning, the outputs depend not on the inputs, but instead on some function (relation) of the inputs. Direct statistical association is not transitive in general. For example consider the exclusive-or function f of two Boolean input variables a and b. Knowing the value of a or b alone or separately does not provide any information regarding the value of f, so f is independent of a and b. However, suppose we have a Boolean function $x = a\overline{b}$ and a Boolean function $y = \overline{a}b$. Then x is dependent on a and b, y is dependent on a and b, and f is dependent on x and y.

The seemingly simple distinction between type-1 and type-2 learning problems provides considerable clarity regarding the capabilities and limitations of the learning algorithms studied to date. Most of the present-day learning algorithms are designed to solve type-1 problems, and not type-2 problems. Typically, a first attempt to provide an input and output representation produces a type-2 learning problem, which fails to be solved by a standard algorithm designed for type-1 problems. Common practice is to rework the input representation by hand, until a type-1 algorithm runs successfully, indicating that a type-1 problem has been achieved from the hand-engineering efforts. A learning task is *ill-posed* for type-1 learning if it is not a type-1 problem. It is futile to apply a type-1 learning algorithm alone to the task of solving a type-2 learning problem. We need to develop algorithms that are capable of solving type-2 learning problems, even when many intermediate layers of representation may be necessary. Note that although artificial neural networks can sometimes find one or even two intermediate layers of representation, they do not extend to the problem of finding arbitrarily deep intermediate representations. Our interest is with type-2 learning in its generality.

Numerous methods have been devised for learning in the limit, e.g. White (1990), but these are of little use in practical applications in which the output mapping is typically highly irregular over the provided inputs. The arguments are similar to claiming that a real-valued function can be approximated with arbitrary accuracy by repeatedly adding terms that reduce the residual error. Memory requirements and hence learning time requirements can grow arbitrarily large. All too often, such algorithms bog down and succumb to an ill-posed learning problem.

Suppose that an input representation A and an output representation C are provided, and that the learning problem is of type-2, as in the exclusive-or example above. Suppose further that there exists an intermediate representation B, such that learning the mapping $A \mapsto B$ is a type-1 problem and learning the mapping $B \mapsto C$ is also a type-1 problem. In principle, a type-2 method could search a space of representations for such a B, but this will be infeasible in general. All is not lost however, if we happen to learn $A \mapsto B$ and then sometime later confront the type-1 problem of learning $B \mapsto C$. The knowledge of mapping $A \mapsto B$ is *prerequisite* to learning the mapping $B \mapsto C$. That which is known provides a basis for that which can be learned simply (type-1) next. Is having prerequisite knowledge in place simply a matter of luck? Usually not; humans benefit from structured environments in which they are nurtured, from educational programs with planned curricula and textbooks, and from the constraints of our real world.

It will be problematic to construct a learning algorithm that solves a lone type-2 problem that has been posed to it. Prerequisite chains can be arbitrarily long, which means that one *or more* intermediate representations may be needed. It would be infeasible to search among the possible arbitrarily deep layerings of arbitrarily chosen candidate representations. Instead, one would need to pose an education plan along with the type-2 problem so that the prerequisite knowledge would be in place, providing a basis for learning the mapping to the outputs as a type-1 problem. Type-2 learning is a by-product of repeated type-1 learning, with layering of the learned building blocks of knowledge according to their input (knowledge) dependencies. This view provides a path to algorithms that are capable of type-2 learning, which is our goal. It is critical to adjust our usual thinking about methods that solve a single type-1 learning task to methods that solve multiple type-1 learning problems with the potential for long chains of type-2 dependencies.

3 Lessons from STL: A type-2 Learning System

Utgoff and Stracuzzi (2002) devised the STL algorithm, which is capable of type-2 learning by way of repeated type-1 learning. The algorithm is based on the tenet that all learning is simple (type-1) if and only if the prerequisites are in place. Indeed, one can use the success or failure of a type-1 learning attempt to indicate whether the prerequisites are in place. Failure does not indicate any shortcoming of the type-1 learning algorithm. Rather, failure means simply that sufficient prerequisites are presently lacking. STL uses a single linear threshold unit or linear function for each predicate or arithmetic function that it needs to learn. This imposes a practical definition for what constitutes type-1 learning for STL.

Suppose that one were not given a prerequisite structure ahead of time, and that a stream of labeled instances for a set of *related* linear units were available. How could all the units be learned, even though (over the given basic inputs) some of them are of type-1 and others are of type-2? The STL algorithm attempts to learn *all* of the linear units simultaneously using its type-1 method. For the problems that happen to be of type-1, the units are learned successfully, but for those that are presently of type-2, the units naturally fail to be learned. The units that were not learned are effectively pushed to a deeper layer, and all those units that have been learned so far are connected as additional inputs (new prerequisites) to each of the failed units. Then the process repeats. One or more of the previously failed units may now be learned successfully with a type-1 method because the new input basis (representation) may have rendered the problem of type-1.

As an illustration, the STL algorithm was trained on a set of learning tasks, with the ultimate task being to learn the relational predicate *column_stackable*(c_1,c_2) over ordered pairs of integer-valued playing cards. For this problem, the set of learning tasks was known to be sufficient if learned and organized correctly. STL found a successful mapping while constructing its own six-layer organization. The same relational predicate could not be learned with a three-layer feedforward artificial neural network. For STL, the step of considering as inputs all the outputs of those functions that have been learned successfully is progressively more expensive, so the algorithm will eventually become mired in input complexity for new units. This problem of deleting useless inputs can be handled efficiently (Stracuzzi & Utgoff, 2004), but an improved approach, not part of this project, is being developed (Stracuzzi, in press).

There are two important lessons to be drawn from the STL algorithm study. First, we have a demonstration that type-2 learning can be accomplished by repeated attempts at type-1 learning, when coupled with a method for searching for an organization of units by discovered prerequisite relationships. It is this combination of 1) processing multiple learning tasks simultaneously, 2) repeatedly attempting type-1 learning for each task, and 3) driving the self-organization process by type-1 successes and failures, that distinguishes STL from other approaches. The algorithm can be paraphrased: *learn successfully that which is presently simple*. As simple learning tasks are mastered, some formerly difficult tasks become simple. There is a *frontier* of learning tasks, each of which is presently simple, with a deep forest of presently difficult tasks beyond. One can succeed at the frontier, and one can cause the frontier to recede by developing further mastery. This relates well to Vygotsky's (1978) zone of proximal development. The learning of tasks succeeds in a bottom-up direction, where 'up' means appending new knowledge to existing prerequisite chains to deepen the agent's knowledge.

Second, type-1 learning is an essential component of practical type-2 learning. It is not a matter of choosing between them; rather, it is a matter of integrating type-1 learning into a larger design. Type-1 methods cannot alone solve type-2 problems. For the purpose of building learning systems that can sustain learning indefinitely, type-1 learning and self-organizing methods (according to knowledge dependencies) are essential ingredients. We know that humans cannot learn knowledge or skills for which the prerequisites are lacking. One cannot pound away, hoping that the missing knowledge will materialize. Learning simple building blocks (units or chunks) indefinitely appears to be at the root of human learning. It may seem paradoxical that the ability to do complex (type-2) learning comes about from being able to do only simple (type-1) learning at any given time. There is a great deal that a human cannot learn immediately, yet there is very little that a human cannot learn eventually.

4 Other Approaches to Layered Learning

Shapiro (1987) developed *structured induction*, which is a form of layered learning in which Boolean features are constructed by manually casting each one as a learning task. Such learned Boolean features then serve as inputs to subsequent learning tasks. Shapiro demonstrated beautifully very dramatic improvements in compression and in learning time. Stone & Veloso (2000) offered *layered learning*, which poses subsequent learning in terms of features learned in an earlier training task. A human decides when the learning at one layer is sufficiently complete and then selects the next representation, type-1 learning algorithm, and training regimen for the next layer. Shultz and Rivest (2000) built the KBCC system, in which cascade-correlation (Fahlman & Lebiere, 1990) can use as features those predicates that it has learned previously. Sammut and Banerji's (1986) MARVIN system can learn new relational predicates from those that it has learned previously through active learning. Banerji (1980) refers to this layering as a 'growing language.'

A class of learning approaches known as *cognitive architectures* learn chunks for resolving problem-solving impasses, as in SOAR (Laird, Rosenbloom & Newell, 1986), or search control heuristics, as in PRODIGY (Minton, Carbonell, Etzioni, Knoblock & Kuokka, 1987). In principle, a chunk can refer to other chunks that have been learned previously. The chunks or control rules pertain to search control for the system's problem solver. Both SOAR and PRODIGY learn from problem solving, with the choice of each problem left to the user.

There has been work on *hierarchical learning*, in which a system of predicates arranged by hand in a fixed hierarchy is trained simultaneously (Rivest & Sloan, 1994; Cesa-Bianchi, Gentile, Tironi & Zaniboni, 2005). Applications include taxonomies for web content. Valiant (2000a, 2000b) has proposed a *neuroidal architecture* in which predicates are represented in layers of linear threshold units, arranged by the human designer. Learning takes place in this fixed architecture.

More recently, Quartz (2003) has expressed arguments similar to those of Clark and Thornton (1997), but from the perspective of neural development. For example, on page 294, Quartz states "the fundamental problems of learning are not those involving statistical inference; instead they center around how to find appropriate representations to support efficient learning." Quartz discusses hierarchical (layered) neural development, pointing to various experiments that show that synaptic formation in the pre-frontal cortex is regional, depending on activity. Quartz quotes Fuster (1997) "The cortical substrate of memory, and knowledge in general, can be viewed as the upward expansion of hierarchy of neural structures." Quartz continues "This hierarchical organization of representations, combined with its hierarchical development pattern, lends support to the view of development as a cascade of increasingly complex representational structures, in which construction in some regions depends on the prior development of others."

There are other works, but we have not yet uncovered any that automatically use simplicity of learning as a guide for self-organization of building blocks. New designs for learning systems of increased autonomy will need to address how to learn multiple serially-dependent tasks over long periods of time, and how to manage memory organization.

5 Learning from Textbooks: A Critical and Feasible Challenge Problem

We have discussed the need for building type-2 learning systems, how success and failure at simple learning tasks can guide self-organization to realize type-2 learning over time, and how scientists and practitioners will benefit from pondering systems that can solve a stream of learning tasks in a process of education. Now we turn our attention to showing in concrete terms that it is possible to build such systems for non-toy problems.

We propose to design and build a computer program that will be capable of absorbing the content of a suitably written textbook. This is an excellent challenge for six reasons:

1. A strong form of automated learning will result. Comparing a learner's knowledge before and after reading a textbook of new material will show that a wealth of new

terminology, concepts, procedures, skills, and the ability to apply the new knowledge will have been acquired. This goes to the heart of understanding and comprehension, which can be measured.

2. This challenge problem can be shaped in a manner that renders it feasible, as described below. It is important for a challenge problem to be realistically achievable, and our approach makes this so.
3. A textbook is a marvelous collection of coherent knowledge, presented in the form of simple building blocks, ordered in a sequential presentation that obeys the dependencies among them. Indeed, the very organization of a textbook itself provides compelling evidence that humans learn effectively from presently simple tasks whose prerequisites are in place.
4. Learning from reading is an unexplored modality for learning that offers considerable power. Even within a single text, there are multiple modes of presentation, including textual explanation, worked examples, and practice exercises. Knowledge is communicated as directly as possible, usually compiled from the findings of others who used a variety of first principles to produce it. Imagine the difficulty in obtaining an education if all the centuries of research in each topic needed to be repeated by each student.
5. A reader accumulates knowledge incrementally while reading. We need to model incremental change of a program's knowledge structures over an extended period, measured in years or decades. The ability to learn next only that which is simple suggests that each incremental change in knowledge will be small. We need to be able to handle a relentless accretion of layered knowledge.
6. Automatic construction and modification of knowledge structures based on reading offers a productive path for engineering of knowledge. When better representations can be devised, the improved reader-learner program can be run anew on its former inputs to produce the new knowledge structures. This is an important step in knowledge construction and maintenance.

6 The LFR Program For Learning From Textbooks

We shall design and build a software system named LFR that will learn from sentences presented to it, principally from textbooks of such sentences. One critical element of the design is the view that each sentence may cause some incremental change in the reader's knowledge state. If the system can map each sentence to a procedure call that effects the desired incremental change, then it will be possible to read, interpret, and take appropriate action for each sentence encountered. Following this view that the reading of a textbook effects a series of incremental changes in the reader's knowledge, a textbook corresponds to a program for the reader to execute, simply by reading it. For a human, the concepts, terminology, procedures, skills, and abilities take shape in as yet unknown ways, but these effects are real and measurable. For the LFR program, we must design the learning mechanisms that will achieve the same kinds of effects.

The top level of LFR is an interpreter loop that accepts and acts on each sentence presented to it. A user (teacher) can direct LFR to read a book stored in a file, consisting of a sequence of such sentences. It is immaterial whether an input sentence comes directly from the user, or indirectly from a book selected by the user, because in either case the sentence is simply read and executed.

The next five sections describe the input language, and memory organization for declarative memory elements, procedural memory elements, associations, and a knowledge applier. The need for other memories or components may arise as the research progresses. The purpose of providing this detail is to show that there are no major impediments to proceeding with the project.

7 Input Language for LFR

We have taken the view that each sentence of a textbook causes a procedure to be called that makes a change in LFR's knowledge. A textbook written for LFR to read is a rich program written for LFR to execute. The sentences should look like natural sentences, and they should convey the same sequence of information as the original human text. A second critical element of the design is that we shall put aside the difficult problem of sentence analysis and semantic interpretation in favor of defining a method for mapping sentences to procedure calls. This will allow us to sidestep the myriad thorny issues that arise when dealing with natural language in its generality.

Our sentences shall be very short, much like an assembly language of English. Indeed, following this analogy, we anticipate that a finite and manageable number of sentence structures (corresponding to procedure calls) will suffice to express any desired computation, much as a machine architecture implements a sufficient finite instruction set. We do not yet know the complete set of procedure calls, but we know many of them. The language will be computationally complete. Consider an example of how a long sentence can be reëxpressed as multiple shorter sentences. The sentence Create a memory item named "column" of organization "vector." can be written as the two sentences Create item "column." Set organization "vector.". Sentences for LFR will generally be either two or three words in length. Both of these shorter sentences start with an imperative verb, followed by a direct object noun, followed by an adjective specifier. There is an implied reference in the second sentence to the direct object of the first sentence. Implied references such as these can be resolved in a well-defined manner through a short term memory, and with additional sentences if necessary that can refresh elements of the short term memory.

An input sentence is converted to a procedure call as follows. Convert the sentence to a sequence of tokens. Consider each double-quoted string to be a single token. Note that open-double-quote and close-double-quote nest, meaning that a string may contain a string. Separate the token sequence into two sequences, one with the double-quoted string tokens removed, and the other consisting of just the double-quoted string tokens. The sequence of non-quoted tokens is concatenated, with '_' connecting each pair, to produce a procedure name. The sequence of double-quoted strings forms the argument list of the procedure. If the procedure name is recognized, either as a built-in or as a learned procedure, then it is called. For example, the sentence Create item "column." would map to the procedure call create_item("column").

This method of mapping sentences to procedure calls is governed by two additional modifiable grammars, which we simply mention here. One grammar holds the rules for the lexical analyzer that breaks the sentence into a token sequence. The second grammar holds rewrite rules for modifying the non-string token sequence before it is used to produce the procedure name. This is useful for mapping synonyms, reordering tokens, removing superfluous words, and otherwise defining a canonical form.

It may seem that some sentences might need to be very long, as in a hypothetical accept structure "...", where the ellipsis would be analogous to an entire C++ class definition. This would be unnatural and too long. Instead, the equivalent of a class definition should be communicated by a list of short sentences, first by creating a class, and then by prescribing its particulars, one at a time. These ideas are explained further below in the discussion of memory items. Arithmetic expressions, and other kinds of expressions too, can be communicated as a sequence of operations to be performed, again much like an assembly language program, but it will be expeditious also to admit formal expressions into the input language as an optional alternative. The sentences of a human text are to be read sequentially, and so too are those of LFR.

There is a spectrum of possible input languages. At one end is a sequential version of a standard programming language, which bears little resemblance to natural language. At the other end are sentences without special syntax, very much like natural language, that make the written form of some computations extraordinarily lengthy. Given our view that reading sequential sentences of text causes sequential incremental changes in the reader's knowledge, we will need to consider the tradeoffs in accepting the many computational shorthands that are so useful when expressing computations.

8 Declarative Memory for LFR

The LFR program will maintain several distinct dedicated memory structures. The declarative memory holds the knowledge that corresponds most directly to the material that LFR has seen through reading. The data type for an element of this memory is called a *memory item*. Such an item is very much like an *object* in the object-oriented sense, but there are enough differences that it is less confusing to use the different term.

A memory item is an element that is referenced by a symbolic name. There will be no storing and manipulating of internal memory addresses for memory items, beyond the original allocation of storage for the item (e.g. malloc() in C). A hash table of symbolic names will cause all equivalent symbol names to have a single identical address (e.g. an obarray in Lisp). Memory items will be accessed through an associated pointer. In short, any named item can be accessed quickly. The principal advantage will be readability of the data structures during development and in presentation. This choice will also facilitate the task of saving and restoring memory items between system sessions.

A memory item has a specified memory organization that is either built-in or learned. The built-in organizations include various atomic values, vector, structure, and procedure. The others are more typical of a standard programming language. Memory items are created and incrementally specified by a sequence of sentences, as discussed in the previous section.

There is no distinction between a type and a constructed item of that type. For example, if a field of a structure has not yet been assigned a value, its value is the distinguished value undefined. One can *create* a memory item from scratch, or one can *clone* an existing memory item to produce a child. The cloned item is given a unique symbolic name as its reference. All of the cloned item's component values are initially set to be undefined. A reference to a component either succeeds immediately, or is passed up the clone's parent link recursively until the reference can be satisfied by a non-undefined value. In this way, values are inherited (Pohl, 1994; Riel, 1996). An assignment of a value to a variable is done locally. The assignment location is not inherited, but of course the locally stored value is inherited downward in the normal manner.

There is no need for a "schema", as any memory item can serve the role. For example, a memory item for "fruit" may hold values for the generic fruit, while a descendant (clone), say "citrus" may hold specialized information. For example, the method for producing juice from a citrus fruit differs from that of other kinds of fruit. The memory item for "fruit" is a schema, but so too is the memory item for "citrus."

A memory item that was *created* has no parent, but a *cloned* item has the original item as its parent. Thus there can arise an assortment of memory item hierarchies, each with a *created* root, and possibly with *cloned* descendants. There will be local name spaces for local environments, much as in Scheme (Abelson & Sussman, 1996). For a field of a structure, its elemental type will be a named memory item, which may be built-in or constructed. The initial field value is undefined, but when an assignment is made, the elemental type will be cloned to produce an instance of the memory item. The value stored in the field is the name of the cloned memory item (each item holds a gensym counter, enabling a unique new name for the clone). The value is then stored in the cloned item. Shortcuts will be implemented for the built-ins.

Consider how a 3x5 matrix could be created. An item of organization 'vector' would be named and created, indexable by 1..3. The elemental type would be an item of organization vector, indexable by 1..5, and of elemental type 'real.' Initially, each of the three components for the vector of rows would have the value *undefined*. If a value were assigned to a matrix cell m[i][j], the row vector would first be allocated, if necessary, by cloning the elemental type. Every element of the newly allocated row vector would be set to *undefined*, and then finally the indexed column component would be assigned the indicated value.

This raises several questions, for example why there is a built-in vector organization, and why the square bracket syntax for vector indexing has been made part of the initial system. It is matter of starting somewhere. In the longer term we will develop in LFR the ability to learn about vector organization, memory allocation, index computation, and the syntax of indexing. We shall pursue this goal, though not immediately.

9 Procedural Memory for LFR

Each procedure is stored as an ordered list of production rules. It may well be that other procedural representations will be of use, but this is where we shall start, due to previous studies that have shown them to be a good medium for procedural learning, e.g. Anderson (1983). Indeed, we are interested in modeling improvement from practice (Rosenbloom & Newell, 1987) and other mechanisms of representational improvement (Karmiloff-Smith, 1994).

As discussed in the preceding section, a procedure is a built-in organization for a memory item, so the distinction between declarative and procedural memories is a convenience of presentation. A memory item that holds a procedure is much like an item that holds a structure. The list of production rules is stored in a distinguished field. Local variables are fields within the procedure's structure. Invocation of a procedure causes a clone to be produced, and the argument values to be copied into the parameters, which are otherwise equivalent to local variables. The production system (rules) are executed until no rule applies. The cloned procedure item is freed when it exits. This mechanism provides the functionality of a run-time stack for parameters, local variables, and returns. Top-level (root) procedures will need to store additional information accumulated from usage, so there will be additional dedicated fields in use that were neither specified nor implied in the definition of a procedure.

A production rule is built by a sequence of sentences that creates its antecedent (left hand side) and its consequent (right hand side). While working on the antecedent, sentences may specify each of the conjunctive literals. As mentioned above, it will be convenient to allow an expression, here a logical expression, to appear in a sentence. Each rule will have a unique name associated with it, which will aid reference to particular rules for various kinds of editing operations. The consequent of a rule is a list of comma-separated sentences to be executed, each one typically assigning a new value to one or more of the procedure's state variables.

When a procedure executes, for whatever reason, the execution provides a trial run of the procedure. Self-modification of the procedure is possible when the correct result is known, for example when a worked example or a practice exercise and its solution are available. We shall examine automated methods for learning within a procedure represented as a production system, discussed below. The implied call graph represents a second organizational structure, an implied hierarchy of procedures. Each one is self-contained and may benefit from use (Utgoff & Cohen, 1998; Shapiro & Langley, 2002). For example, the procedure for long division requires a procedure for subtraction.

Anderson (1983) discusses how students learn to produce proofs in geometry, and he compares this to how his ACT system learns. The underlying representation of knowledge for ACT is a set of production rules. Knowledge is transformed over time to increasingly usable forms, through processes such as composition and proceduralization. Introducing a new procedure or production rule affords a system new paths of activity, which means that the system may come to revise its policies. Anderson observes that a human student acquires knowledge from a geometry textbook, such as postulates, theorems, and definitions. From the worked examples included in the text, or produced when solving the exercises, the student learns how to convert knowledge to procedural rules and how to apply them efficiently to solve problems of interest.

VanLehn (1987) created the Sierra program, which learns procedures in the domain of arithmetic. Of particular interest is his discussion of how human teachers and human learners obey two conventions that facilitate the education process. First, a lesson attempts to teach exactly one procedure at a time, where a procedure is comparable to a production rule. To attempt more than this is computationally too difficult. Second, the student must be shown, at least once, all the calculations that are necessary to carry out a given step. VanLehn observes (page 3), "Each lesson builds on the procedure learned in the previous lesson." Sierra learns from examples of action sequences, the worked examples that one would find in a text book, but it does not learn from the sentences and explanations provided in the textbook.

Production systems are strongly related to finite state machines. The machine state can be encoded as a local state variable that is tested or set in the rules as appropriate. We shall exploit such homomorphisms in order to take advantage of research that uses other representations of procedure, such as Markov Control Processes. Other choices for representing a procedure include storing code fragments in an interpreted language, such as a dialect of Lisp. However, we are very much interested in modeling the effects that practice have on refining and speeding up the execution of a procedure.

10 Associative Memory for LFR

An associative memory will be maintained automatically. Suppose a taxonomy of memory items has been created for various classes of fruit. If grapefruit is a kind of citrus, then the associative memory would automatically hold two associations, one for grapefruit indexing citrus, and the other for citrus indexing grapefruit. Associations from the memory items will be updated incrementally during operations on the memory items, such as creating, cloning, modifying, or freeing.

Such associations have myriad uses, but we shall focus on harnessing them to guide the knowledge applier (problem solver) of LFR, as discussed below. For example, suppose that an exercise requires solving a rate-time-distance problem. Any procedures that include any of these three terms would be a good candidate in the search process for solving the exercise. More distant associations (associations of associations) can provide additional candidates.

11 Knowledge Applier (Problem Solver) for LFR

A problem solver will be included in LFR's architecture. When LFR reads or otherwise encounters a practice (homework) exercise, it will suspend reading while it attempts to complete the exercise. This is a critical aspect of learning from a textbook. The problem solver will search among its procedures, guided by associations with characteristics of the exercise or problem at hand. There are many well known state space search methods and variants, and we shall not review them here. If the exercise has not been solved correctly, by comparing its solution to an answer given with it or in an answer key, then LFR will need to halt and seek help from a human teacher. Otherwise, if the solution is correct or the correct answer is not available for comparison purposes, then LFR will resume reading.

In many textbooks, solutions for some or all of the exercises are available in an answer key supplied as an appendix. The ability to consult an answer key requires a non-sequential reading of the book. We may put aside this detail by distributing solutions close to the corresponding exercises. A second choice would be for LFR to read the appendix of solutions, associating each solution with an exercise number. Then whenever it comes to an exercise, the solution would be immediately accessible by the association. Of course, the solution is helpful for purposes of verification and learning, not for bypassing the need to solve the problem.

Word problems are known to present a special challenge for humans. It will be essential to be able to accept a list of givens (start state), and to show how to arrive at a goal (stop state). One capability that LFR will require is the ability to suppose. An exercise will often present hypothetical temporary facts that are true for the purpose of the exercise, and then are to be forgotten. One possible approach will be to create a temporary environment (such as when a procedure executes) in which the givens are represented as local memory items. When the environment is reclaimed, the temporary facts (givens) will vanish too. Certain words, like 'suppose' will trigger these mechanisms automatically.

12 Subject Domains

Domains of interest are those with some formal structure, for example mathematics broadly construed, or a science such as Chemistry or Physics. We would not expect to make headway in a subject like Creative Writing. We shall commence with a textbook on Linear Algebra (Lay, 2003). This immediately raises the question of how one can start at an advanced state of knowledge when the prerequisites have not been learned previously. This can be handled by creating translations near the beginning of the text that may be wordier than would otherwise have been necessary. An alternative is to start with prerequisite texts, but then one must again choose the prerequisites of those, until one arrives at the first texts, such as those from grade school. Of course these start at a stage when a child can read, but children learn elementary mathematics, such as counting, at an even younger age. We shall undertake earlier texts, and later texts too, but we will start 'in the middle' in this case. We do not want to design a system that is required to start at the earliest stages of knowledge formation. We do want to design a system whose performance will not degrade as large amounts of knowledge are accumulated over time (Minton, 1990).

Consider the knowledge dependency graph (manually constructed) shown in Figure 1, which corresponds to the first twenty pages (of 498) of Lay's text. The book assumes prior knowledge of

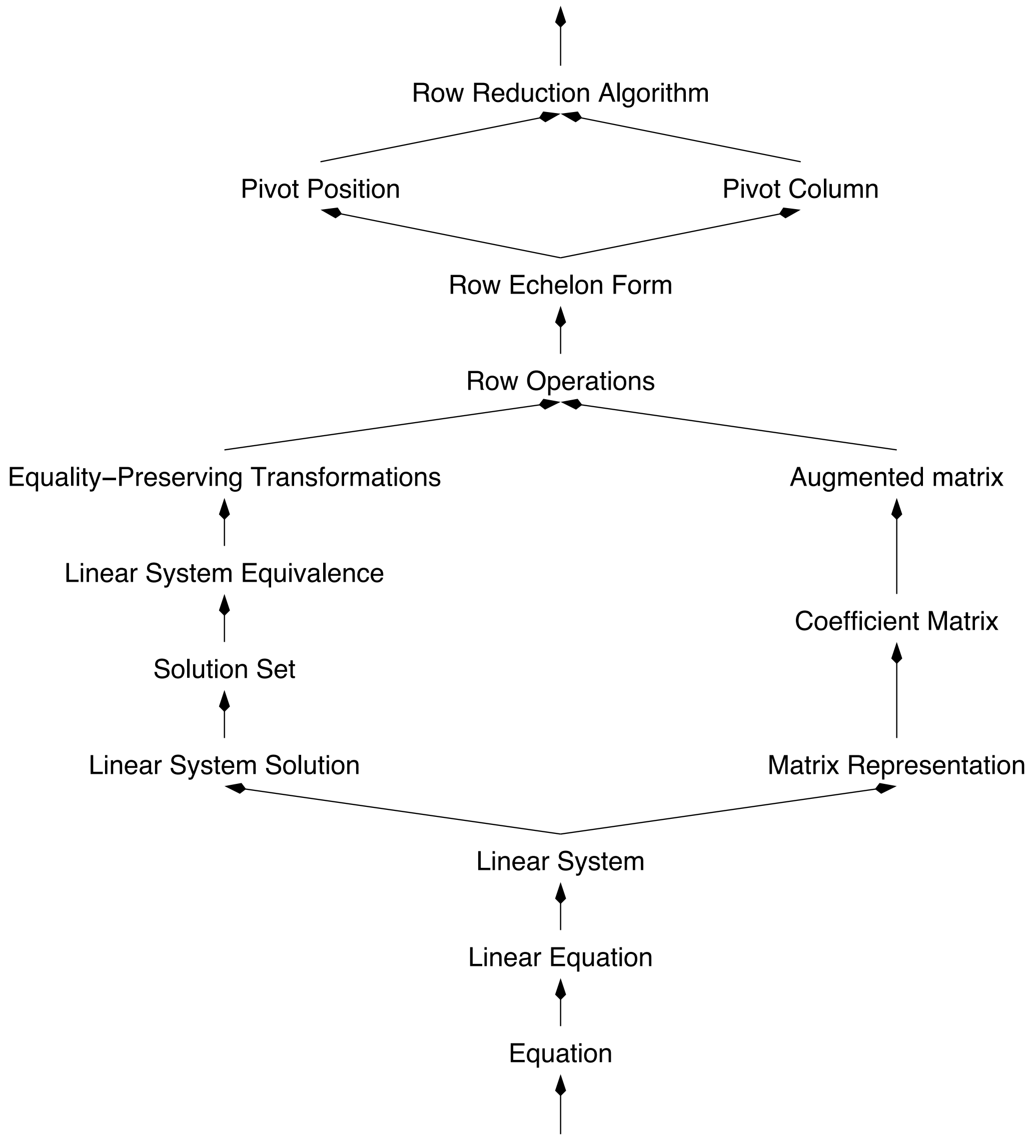

Figure 1. Layers of Knowledge Dependencies for Row Echelon Form

equation, and then defines *linear equation* in terms of it and several other constraints. Then a *linear system* of simultaneous linear equations is introduced. The left hand path of explanation continues with *linear system solution*, what it means to solve a set of simultaneous linear equations. The notion of *solution set* comes next, followed by the important concept of *linear system equivalence*. Recall that two linear systems are equivalent if they have the same solution set. Next comes *equality-preserving transformations*, showing that one can change the current linear system to another that is equivalent, but easier to solve.

The right hand path of explanation introduces the compact matrix representation of a linear system. Following this comes *coefficient matrix* and then *augmented matrix*. The reader is now

1. The Row Reduction Algorithm

2.

3. The algorithm that follows consists of four steps,and it produces a matrix in echelon
4. form. A fifth step produces a matrix in reduced echelon form.We illustrate the algorithm
5. by an example.

6.

7. Example 3: Apply elementary row operations to transform the following matrix first
8. into echelon form and then into reduced echelon form:

9.

10.
$$\begin{bmatrix} 0 & 3 & -6 & 6 & 4 & -5 \\ 3 & -7 & 8 & -5 & 8 & 9 \\ 3 & -9 & 12 & -9 & 6 & 15 \end{bmatrix}$$

11.

12. Solution

13.

14. Step 1
15. Begin with the leftmost nonzero column. This is a pivot column. The pivot posi-
16. tion is at the top.

17.

18.
$$\begin{array}{l} \begin{bmatrix} 0 & 3 & -6 & 6 & 4 & -5 \\ 3 & -7 & 8 & -5 & 8 & 9 \\ 3 & -9 & 12 & -9 & 6 & 15 \end{bmatrix} \\ \quad\uparrow \\ \text{pivot} \\ \text{column} \end{array}$$

Table 1. Excerpt at Page 17 of Lay's Text

prepared to understand how equality-preserving transformations can be modeled as operations on one or more rows of a matrix. For the purpose of obtaining a solution to the linear system, it is especially convenient for the matrix to be in *reduced row echelon form*. After introducing *pivot position* and *pivot column*, the *row reduction algorithm* is given, consisting of five explicit steps.

Even in this relatively short exposition, there are already prerequisite chains of length eleven. At the rate of eleven levels per twenty pages, we can extrapolate to estimate roughly 265 layers of knowledge in this text alone. If one were to include high school prerequisites or subsequent courses that require linear algebra, we could easily expect several thousands to many thousands of layers.

Starting at the bottom of page 17 of Lay's text, there is a presentation of the *Row Reduction Algorithm*. Table 1 shows an excerpt, with line numbers added for ease of reference. There is much information even in just these eighteen lines. Lines 3-5 explain that Steps 1-4 comprise one algorithm, and that Step 5 comprises a second algorithm that is applicable to the result of applying the first. Later, on page 20 (not shown), the first is labeled *forward phase* and the second is labeled *backward phase*. Although the phases are identified, the author clearly wishes the reader to conceive of a single algorithm, not two.

The author (Lay) has chosen to illustrate the algorithm in parallel with presenting it. This helps the reader to see the algorithm in action, making it that much easier to understand and retain each step. An alternative would have been to present all the steps of the algorithm, and then immediately afterward to work the same example. The author has assumed that the reader can track the algorithm steps in their generality, and the example in its specificity, without confusion.

Consider how lines 15-16 can be restated so that a sequence of one or more sentences would be evident. The first sentence of line 15 Begin with the leftmost nonzero column. is at quite a high level. One needs to understand 'column' and to be able to identify or see each of the columns of the matrix. It is necessary to understand that a non-zero column is a column that contains at least one non-zero component (entry). The reader must also understand 'leftmost.' The sentence itself describes a procedure, which for a human can be accomplished by scanning the columns of the matrix visually in left-to-right order.

In high level pseudo-code, the computation would be: *column_index = min_index_true(m.min_col,m.max_col,i,non_zero_column(extract_column(m,i)))*. In contrast, Table 2 shows a lower level elaboration of this sentence and the two that follow it, modeling the implied doubly nested loop as a procedure. We would rather compose short procedures or functions as suggested by the pseudo-code, but our preference for this example was to show a complete definition. While a state diagram might be clearer, our purpose is to show machine processable text that corresponds to the original.

The most striking difference between the original text and the above elaborated version is the level of detail. The translated version requires some 41 lines for the three sentences of the original. The difference would be smaller if, as shown in the pseudo-code, a predicate had been learned for 'non-zero column.' The inner loop checking the column for a non-zero entry would vanish. Similarly, the outer loop could disappear with a suitable argument selection construction. More typically, procedures will be short, built from those that already exist.

One can be concerned about the human translation process, from an actual textbook such as Lay's to a version that will be interpretable by an automated reader. This is an important element of the project, to determine what expressions of information is sufficiently informative for the intended reader. Writing for an adult human is very different from writing for a mechanical learner. Our goal is to be able to write for a reader such as our proposed system LFR. An existing textbook serves as a model for a good order in which to present a succession of simple learning tasks. It will be possible to follow such an order very closely while writing/translating for a mechanical reader such as LFR.

The author Lay's interleaving of presenting the algorithm and illustrating it on an example is quite effective. The reader can check and otherwise ensure his/her understanding of Step 1 before proceeding. The procedure defined for just Step 1, as above, is already executable. It would be possible to apply it to the given example. As discussed above, it is not clear how one should proceed when encountering a failure. When all is going well, we expect that LFR will absorb the content perfectly at breakneck speed. Upon failure however, proceeding with reading will not be productive, so LFR will need to suspend reading, seek input from the user (teacher), and explain the stumbling point, e.g. having encountered an unknown term, being stuck on a problem, or producing a solution different from that provided in the text.

Does one debug the presentation of a textbook? Yes, authors certainly test their books, and produce new editions from time to time. We will need to develop a discipline for when to debug LFR and when to debug the text that LFR reads. For example, was an error reasonable, given what LFR knew and what the text said? If so, debug the text, and otherwise debug LFR. In any case, a human reader does not normally read a textbook just once from beginning to end. It will be important to be able to unlearn the wrong, and to revisit problematic material. This is somewhat orthogonal to the primary goal of being able to build data structures that scale up to sustained learning, but it will be necessary.

13 Research Dimensions

There are several large components to the LFR system that we propose to build and study. There is also a need for sophisticated input, primarily in the form of textbooks written is a well-defined English-like language. The research will proceed along several fronts:

Input Texts - Finish translating Lay (2003), translate at least one early mathematics text, one text on probability, e.g. Devore (1995), and one text from a science such as Physics or Chemistry. This will be undertaken largely by the undergraduate assistants.

Input Modes - In addition to plain text, study how to represent figures, tables, diagrams, worked examples, exercises (both imperative and interrogative), and word problems (given(s) and goal(s)).

Input Language - Add sentences for modifying the lexical grammar rules of the tokenizer, enhance the sentence-to-procedure mapper and its grammar rules, add sentences for modifying the expression parser and its grammar rules.

Declarative Memory - Develop further the mechanics for memory items, including creating, cloning, modifying, lazy construction, inheritance, save and restore, structure, vector, procedure, reference counts, and leak-free memory management.

Procedural Memory - Develop further the mechanics for procedures, call protocol for binding parameters to arguments, local memory items, executor, learning and compiling through use (practice), relationship to policy iteration and reinforcement learning, non-deterministic procedures.

Associative Memory - Design and build automatic associations from memory item construction and modification, success and failures of problem solving efforts, other sources of associations.

Knowledge Applier - Design and build search mechanism for applying knowledge to problems, guided by degree of association and basic state-space search.

Known Information - design a method for passing over sentences when the knowledge being conveyed is already known. Consider the problem of rereading.

Experimentation - Consider how LFR could read experimental outcomes and produce generalizations in the more traditional machine learning modes.

Evaluation - Design evaluation metrics based on becoming stuck on sentences, success rate on exercises and other tests, successful transfer, and resource consumption such as time and space.

Presentations - Publish each finding, visit scientists at other institutions. Design visualizations of LFR's knowledge structures and operations.

Literature - Read related literature in Machine Learning, Knowledge Representation, Developmental Psychology, Education (theory) regarding textbook writing, and Linguistics.

Documentation - Maintain program documentation using doxygen and other tools.

14 Discussion

The proposed research is on a critical path to advancing automation in the areas of Machine Learning, Knowledge Acquisition, and Knowledge Representation. Simple (type-1) learning can succeed only when a simple (direct statistical) relationship from inputs to outputs can be found. Most learning problems are difficult (type-2). Current practice is to convert a difficult problem to a simple problem by manually providing new inputs that render the problem simple. We need methods that succeed at difficult (requiring prerequisites currently lacking) learning without manual conversion of the problem. The author (with co-author) has designed and reported the STL algorithm, which is capable of difficult learning, realized by using the successes and failures of multiple simple learning attempts to guide a self-organization process that dynamically arranges the successfully learned building blocks into a deeply layered compositional structure that defines the mapping of representations. This is a research path that others will likely need to follow as they seek to improve learning automation and to understand the nature of learning a great amount of knowledge over a large expanse of time.

The proposed work on learning from textbooks will develop a new modality for automated learning programs. Existing learning techniques are largely data driven. For example, classifier inducers fit a model to labeled data, cluster inducers fit a model to unlabeled data, and policy inducers fit a performance model to trial data gathered over time. These data-centric approaches have useful applications. However, one does not need to limit learning to the fitting of observations. The general principles, procedures, and concepts that have been learned or discovered over the centuries can be passed on by describing and explaining them in words and pictures for another to read. We have only to look to our libraries, bookstores, publishing houses, private collections, and educational institutions to see plentiful overwhelming testimony to the utility of transferring knowledge in this manner. Other researchers will likely need to follow this path as they seek to educate their learning programs.

It may seem ambitious to move into largely unexplored territory. However, the time has arrived for fruitful research in this area. The risk is low for two reasons. First, we have already been able to demonstrate that the approach will work with our initial version of LFR. The main risk has already been taken. Second, we have framed the research in a manner that avoids the significant potential pitfalls. For example, avoiding the natural language understanding problem, by mapping sentences to procedures calls, sidesteps numerous thorny issues. This allows the research to remain focused on the incremental knowledge transfer problem, issues of representation, and organizational issues that will enable LFR to scale up to absorbing large quantities of coherent knowledge over long periods of time. As a second example, LFR's input language is computationally rich, most easily characterized as a dynamic interpreted sequential language with capabilities that subsume those of C++. Third, the simple computational model for procedures (local production system) avoids pitfalls regarding code modification. The focused approach makes reaching the main research goals highly likely.

15 Summary

We observed that there is a fundamental and technical need for strong learning methods that take on the problem of educating learning programs on systems of serially dependent tasks. Such approaches will make use of simpler methods for learning simple tasks, not displace them. We noted that knowledge must accrete through a sustained process of learning tasks that are presently simple. To this end, we outlined how to build a system capable of learning from suitably written textbooks, in which the simple building blocks of related knowledge are arranged in a favorable order so that all learning is simple when it is encountered. This new modality for automated learning is now approachable due to our view of learning and organizing simple building blocks, and due to our framing the problem in a doable and realistic manner, free of major impediments. The project will produce a critical advance in how machines learn and acquire useful expertise. This advance will open a door to new capabilities and research issues that formerly were out of reach, and now are not.

Acknowledgments

This work was supported by National Science Foundation Grant IRI-0097218. Helpful comments were provided by David Stracuzzi, Manuel Nickschas, Dennis Gove, Steven Murtagh, Gary Holness, Phil Kirlin, and Andy Barto.

References

Abelson, H., & Sussman, G. J. (1996). *Structure and interpretation of computer programs (2nd edition)*. McGraw Hill.

Anderson, J. R. (1983). Acquisition of proof skills in geometry (pp. 191-219). In Michalski, Carbonell & Mitchell (Eds.), *Machine learning: An artificial intelligence approach*. San Mateo, CA: Morgan Kaufmann.

Banerji, R. B. (1980). *Artificial intelligence: A theoretical approach*. Elsevier.

Cesa-Bianchi, N., Gentile, C., Tironi, A., & Zaniboni, L. (2005). Incremental algorithms for hierarchical classification (pp. 233-240). In Saul, Weiss & Bottou (Eds.), *Advances in Neural Information Processing Systems*. MIT Press.

Clark, A., & Thornton, C. (1997). Trading spaces: Computation, representation, and the limits of uninformed learning. *Behavioral and Brain Sciences, 20*, 57-90.

Devore, J. L. (1995). *Probability and statistics for engineering and the sciences, fourth edition.* Pacific Grove, CA: Brooks/Cole Publishing Company.

Fahlman, S. E., & Lebiere, C. (1990). The cascade correlation architecture. *Advances in Neural Information Processing Systems, 2*, 524-532.

Fuster, J.M. (1997). Network memory. *Trends in Neuroscience, 20*, 451-459.

Karmiloff-Smith, A. (1994). Précis of Beyond modularity: A developmental perspective on cognitive science. *Behavioral and Brain Sciences, 17*, 693-745.

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

Lay, D. C. (2003). *Linear algebra and its applications, 3rd edition*. Addison-Wesley.

Minton, S., Carbonell, J. G., Etzioni, O., Knoblock, C. A., & Kuokka, D. R. (1987). Acquiring effective search control rules: Explanation-based learning in the PRODIGY system. *Proceedings of the Fourth International Workshop on Machine Learning* (pp. 122-133). Irvine, CA: Morgan Kaufmann.

Minton, S. (1990). Quantitative results concerning the utility of explanation-based learning. *Artificial Intelligence, 42*, 363-391.

Pohl, I. (1994). *C++ for C programmers, second edition*. Benjamin/Cummings.

Quartz, S. R. (2003). Modeling the neural basis of cognitive development (pp. 291-313). In van Ooyen (Ed.), *Modeling Neural Development*. MIT Press.

Riel, A. J. (1996). *Object-oriented design heuristics*. Addison-Wesley.

Rivest, R. L., & Sloan, R. (1994). A formal model of hierarchical concept learning. *Information and Computation, 114*, 88-114.

Rosenbloom, P. S., & Newell, A. (1987). Learning by chunking: A production system model of practice (pp. 221-286). In Klahr, Langley & Neches (Eds.), *Production system models of learning and development*.

Sammut, C., & Banerji, R. B. (1986). Learning concepts by asking questions (pp. 167-191). In Michalski, Carbonell & Mitchell (Eds.), *Machine learning: An artificial intelligence approach*. San Mateo, CA: Morgan Kaufmann.

Shapiro, A. D. (1987). *Structured induction in expert systems*. Addison-Wesley.

Shapiro, D., & Langley, P. (2002). Separating skills from preference: Using learning to program by reward. *Proceedings of the Nineteenth International Conference on Machine Learning* (pp. 570-577). Morgan Kaufmann.

Shultz, T. R., & Rivest, F. (2000). Using knowledge to speed learning: A comparison of knowledge-based cascade-correlation and multi-task learning. *Proceedings of the Seventeenth International Conference on Machine Learning* (pp. 871-878). Palo Alto, CA: Morgan Kaufmann.

Stone, P., & Veloso, M. (2000). Layered learning. *Proceedings of the Eleventh European Conference on Machine Learning* (pp. 369-381). Springer-Verlag.

Stracuzzi, D. J., & Utgoff, P. E. (2004). Randomized variable elimination. *Journal of Machine Learning Research, 5*, 1331-1362.

Stracuzzi, D. J. (in press). Scalable knowledge acquisition through memory organization. *International and interdisciplinary conference on adaptive knowledge and reasoning (AKRR 2005)*. Espoo, Finland: Helsinki University of Technology.

Utgoff, P. E., & Cohen, P. R. (1998). Applicability of reinforcement learning. *Proceedings of the 1998 ICML Workshop on the Methodology of Applying Machine Learning* (pp. 37-43). AAAI Press Report WS-98-16.

Utgoff, P. E., & Stracuzzi, D. J. (2002). Many-layered learning. *Neural Computation, 14*, 2497-2529.

Valiant, L. G. (2000a). A neuroidal architecture for cognitive computation. *Journal of the Association for Computing Machinery, 47*, 854-882.

Valiant, L. G. (2000b). Robust logics. *Artificial Intelligence, 117*, 231-253.

VanLehn, K. (1987). Learning one subprocedure per lesson. *Artificial Intelligence, 31*, 1-40.

Vygotsky, L. S. (1978). *Mind in society: The development of higher psychological processes*. Cambridge, MA: Harvard University Press.

White, H. (1990). Connectionist nonparametric regression: Multilayer feedforward networks can learn arbitrary mappings. *Neural Networks, 3*, 535-549.

```
create item “set_pivot_column_pivot_position”
set organization “procedure”

accept argument “m”
set organization “matrix”
accept argument “pivot_column”
set organization “integer reference”
accept argument “pivot_position”
set organization “integer reference”
accept argument “result”
set organization “integer reference”

create field “state”
set organization “symbol”
assign state “START”

append rule “R1”
append lhs “state == START”
append rhs “assign pivot_column “m.min_column”, assign pivot_row “m.max_row””
append rhs “assign state “A””
append rule “R2”
append lhs “state == A”
append rhs “assign pivot_row “m.max_row”, assign state “B””
append rule “R3”
append lhs “state == B, m.val[pivot_col][pivot_row] == 0”
append rhs “assign state “C””
append rule “R4”
append lhs “state == B”
append rhs “assign result “SUCCESS”, assign state “STOP””
append rule “R5”
append lhs “state == C, pivot_row ≥ m.min_row”
append rhs “assign state “B””
append rule “R6”
append lhs “state == C”
append rhs “assign state “D””
append rule “R7”
append lhs “state == D, pivot_col < m.max_col”
append rhs “assign pivot_col “pivot_col + 1”, assign state “A””
append rule “R8”
append lhs “state == D”
append rhs “assign result “FAILURE”, assign state “STOP””
```

Table 2. Procedure for Setting Pivot Column and Pivot Position