SCALABLE KNOWLEDGE ACQUISITION THROUGH MEMORY ORGANIZATION

David J. Stracuzzi

Department of Computer Science, University of Massachusetts, Amherst, MA 01003
stracudj@cs.umass.edu

ABSTRACT

Memory organization plays a critical role in knowledge acquisition. An agent must select a small subset of existing knowledge to serve as the basis for new learning; otherwise each problem becomes more complex than the previous. Selecting this subset remains a challenge, however. We propose that existing knowledge be organized in order for a learning agent to achieve its full potential. The SCALE algorithm is presented as a method for knowledge acquisition and organization, and is used to demonstrate both the computational and training benefits of memory organization.

1. INTRODUCTION

Acquiring new knowledge is an essential element in intelligent behavior. Humans are adept at learning and organizing knowledge from diverse sources over extended periods of time. Artificially intelligent agents can likewise benefit from the ability to improve upon current knowledge. Toward this end, the machine learning community tends to view learning as a search problem. The task is to find a specific relationship between input stimuli and target outcomes that accurately models the underlying concept or function. This is one of the most basic forms of learning, known as supervised learning.

Clark & Thornton (1997) refer to such direct input-to-output mappings as type-1 learning. Suppose now that the relationship between input and output is complex, so that a given supervised algorithm cannot find a direct mapping between inputs and output. Clark and Thornton refer to this scenario as type-2 learning and assert that any solution to such a problem must involve finding one or more intermediate representations of the stimuli. Although they do not recommend a specific method, Clark and Thornton state that type-2 learning requires the input stimulus to be recoded until the gap between input and output becomes sufficiently narrow for a type-1 algorithm to succeed.

The central issue associated with producing intermediate representations is again one of search. To solve type-2 problems, an agent must be able to discover or otherwise acquire new concepts, called *building blocks*, which close the input-output gap. This is a necessarily vague description. The agent does not know in advance what these building blocks should represent, or how many layers will be needed. The agent knows only that it must search for some new combination of input stimuli and existing building blocks that will improve the statistical regularity of the data. In the absence of additional information, a brute force search is intractable as the search must span the range of functions learnable by the agent.

Making matters worse, the agent also does not know which existing building blocks will contribute to learning a new block. Imagine a system in which, for each new building block, the agent must consider all existing blocks as a potential source of useful knowledge. The number of input combinations considered by the agent would grow for each new concept. Each new intermediate learning problem becomes more computationally expensive than the previous. Learning based on this method cannot scale up.

In this paper, we take the view that intermediate representations are just as important as the high-level target knowledge. Development of intermediate representations deserves at least as much learning effort and attention as an agent's high-level learning goals. Most importantly, we propose that intermediate knowledge be organized in order for the agent to achieve the full potential of its learning capabilities.

2. ACQUIRING INTERMEDIATE KNOWLEDGE

The term *many-layered learning* refers to methods which, in the spirit of type-2 learning, allow the agent to construct as many layers of intermediate knowledge as needed (Utgoff & Stracuzzi, 2002). The structures may be nested to an arbitrary depth, as required by the complexity of the learning problem and the agent's learning capabilities. A many-layered approach differs from other methods in that the original input space is not partitioned with respect to the top-level concepts. Instead, the layers form new spaces that are subsequently partitioned with respect to the *next layer*. Each intermediate concept and each layer may be viewed as adding new dimensions to the existing space, or as forming an entirely new space.

The Stream-to-Layers (STL) algorithm by Utgoff & Stracuzzi (2002) implements this idea. Here, the environment provides a stream of labeled training stimuli. Each stimulus contains data for exactly one building block (concept). As the training points arrive, each building block attempts to learn its target concept. Those that are successful become available as inputs to more complex, unlearned blocks. Thus the algorithm constructs the basis for complex concepts over time. STL makes no assumptions about the ordering of the incoming data, simply constructing intermediate representations by learning about each building block wherever possible.

The STL algorithm yielded several positive results. The algorithm learned and organized a layered network of building-block concepts given an unorganized stream of data. This is not to say that the algorithm reproduced the original structures as conceived by the authors. STL often found different yet equivalent representations of the data, and displayed both the ability to learn and to separate building blocks from multiple domains.

STL's primary weakness is its approach to acquiring new inputs. By considering all learned building blocks as potential inputs to an unlearnable block, the algorithm creates a situation in which the input dimensionality of high-level concepts grows steadily throughout the learning process. Clearly this approach cannot scale up. A more organized approach to dealing with potential inputs is required.

3. THE SCALE ALGORITHM

The SCALE algorithm is designed to provide long-term learning capabilities by building upon STL's strengths. As such, SCALE (Stracuzzi, 2005) improves upon many of the methods originally designed for the STL algorithm. From an abstract point of view, the two algorithms are founded on the same basic idea: an agent with limited immediate learning capabilities learns about complex domains in the long-term by building up a basis of knowledge one small block at a time.

A closer look however, shows that SCALE is concerned not only with acquiring new blocks of knowledge, but also with organizing and managing the concepts to facilitate their efficient application to future learning. We begin with a brief overview of SCALE, and turn to memory organization methods in the following section.

3.1. Limited Learning Ability

Like STL, the SCALE algorithm is founded on the idea that complex type-2 learning abilities are founded on simple type-1 learning mechanisms. Specifically, neither algorithm can learn any concept that is not linearly separable. This restriction provides several attractive properties. First, it is a learning bias not specifically designed for a particular class of problems. Second, limited learning provides a comparatively small search space for each building block concept, with simpler learning abilities providing smaller search spaces. Third, the set of learnable functions grows as the agent learns. Each new building block expands the agent's horizons.

Growth in knowledge learnable by the agent is key. This space must be allowed to grow, otherwise the agent could only learn concepts in the very simple class to which it is restricted. The growth however, means that the search space will once again become intractably large over time. The building blocks learned by the agent therefore *must* be organized so as to maintain a tractable search space.

3.2. Representational Language

The representational language defines a system's capacity to learn. If a concept cannot be expressed in a language, then no amount of training will allow the system to acquire that concept. Conversely, an overly powerful language may cause learning to fail because the language defines an intractably large hypothesis space (Utgoff, 1986). Representational languages should always be merely appropriate to the task at hand, providing a balanced tradeoff between sufficient learning capacity and a tractable hypothesis space. This is a fundamental idea used in developing support vector machines (kernel selection depends on the properties of the given problem) (Burges, 1998).

SCALE relies on form of first-order logic. Building blocks are represented by predicates and may be viewed as parameterized functions. A given predicate may be called by any number of higher-level predicates, and multiple times by a single predicate. This provides many opportunities for knowledge transfer, and a significant reduction in redundant learning over a propositional representation.

First-order logic provides more power than necessary for layered learning. Several modifications and restrictions are placed on the language in order to suit the needs of SCALE. Each restriction is motivated by compliance with the agent's limited learning ability. However, none of the following restrictions reduce the agent's ability to learn complex features in the long run. The agent can always reconstruct them from simpler features.

1. Each predicate may only represent a linearly separable concept. This is the primary enforcement of the limited learning assumption.
2. Predicates have a limited number of arguments (arity). The number of unique bindings between the arguments of one predicate and another grows exponentially with predicate arity.
3. The agent has a limited *focus of attention*. This is the set of domain objects, such as cards in a poker hand, under explicit consideration by the agent. Increasing the number of objects simultaneously considered by an agent increases the number of object-variable binding combinations exponentially.
4. For brevity, we do not discuss quantification in this paper.

These restrictions act very much along the same lines as Miller's magic number seven (1956), which refers to the approximate upper limit on the number of objects a person can keep in immediate memory. Miller argues that humans use simple but powerful mechanisms to increase their capacity. These mechanisms include increasing the number of dimensions along which a stimulus may vary, recoding objects into larger chunks, and handling objects sequentially rather than simultaneously.

Figure 1 shows a sample SCALE representation along with an example state for a cards domain such as poker or solitaire. We assume that the input stimulus received by

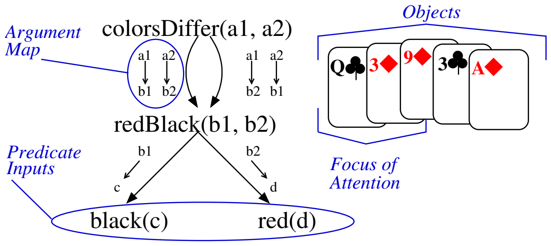

Figure 1. Illustration of SCALE's representation.

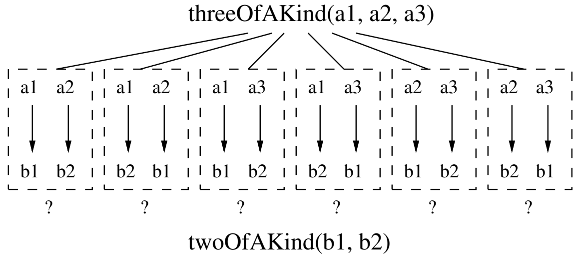

Figure 2. Illustration of predicate mapping enumeration.

the agent is rudimentary. Cards are indexed 0–51 such that 0–12 represent ace through king of spades, 13–25 represent hearts and so on for clubs and diamonds. The agent receives the index and must learn to establish the card suit, rank and color, along with any relationships among cards.

The domain objects are the cards, while the focus of attention is some small subset of current interest. Each predicate represents a card relation (such as *card-is-red* or *card-colors-differ*), while the links between predicates show dependencies. Each link also contains a binding of arguments from the link-output to the link-input.

A typical evaluation proceeds top-down. For example, an agent can test whether the cards in its focus of attention have different colors, as in Figure 1. The cards are first bound to the predicate argument variables, and then *colorsDiffer(Q♣, 3♢)* is evaluated. The predicate *colorsDiffer* represents a learned linear threshold function over its inputs, so the next step is to compute values for these inputs. The arguments to *colorsDiffer* are therefore bound, through specific mappings established via learning, to the predicate *redBlack*.

RedBlack(Q♣, 3♢) is now evaluated via a similar procedure and found to be *false* while *redBlack(3♢, Q♣)*, via the right binding from *colorsDiffer* is found to be *true*. In this case, *colorsDiffer* is represented by a logical-or over its inputs, resulting in a final evaluation of *true*.

The language determines SCALE's ability to represent knowledge, but says nothing of how knowledge is formed. We turn now to the questions of how SCALE determines predicate dependencies, threshold functions, and argument mappings.

3.3. Learning Predicates

Learning in SCALE is a highly distributed and parallel operation. Each building block (predicate) receives its own supervised data, and attempts to learn independently of other blocks. For the purposes of this paper, we assume that the environment provides this data decomposition, and that some higher-level mechanism directs data to the correct building block. Note that SCALE is an online and incremental algorithm; all training examples are processed upon receipt and immediately discarded.

The online and distributed nature of SCALE places two requirements on the building-block (predicate) learning algorithm. Each building block must detect whether the target knowledge has been successfully learned. Likewise, a building block must detect whether it is unlearnable with respect to the current knowledge basis (set of input dependencies) and recent training points. To meet these requirements, SCALE uses a modification of the perceptron algorithm (Minsky & Papert, 1972) originally developed for perceptron trees Utgoff (1989).

Briefly, the Perceptron Convergence and Cycling Theorems (Minsky & Papert, 1972) are combined with empirically established threshold values to provide precise definitions for *learned* and *unlearnable*. A building block with n input dependencies is learned if it correctly evaluates $l(n)$ consecutive training examples. Similarly, the block is unlearnable if it makes $u(n)$ consecutive weight updates without exploring a new area of weight-space. Given this ability to recognize blocks as learned and unlearnable, each building block must determine a knowledge basis that is sufficient for learning.

Using perceptrons, which are propositional, to represent predicates may initially appear to be an unusual application. Closer inspection reveals that each unique argument mapping is equivalent to a single proposition. Recall SCALE restricts the number of arguments available to a given predicate, ensuring that the number of possible mappings between two predicates remains small. Thus the predicate is propositionalized, with the perceptron treating each argument mapping as a unique binary-valued input.

Figure 2 illustrates the approach with another example from the cards domain. There are six possible mappings from the three-argument predicate *threeOfAKind* to the two-argument *twoOfAKind*. Each mapping produces a distinct evaluation of *twoOfAKind*. In this case, multiple combinations of two mappings will produce the desired relationship (say the first and third). Over the course of many training stimuli, the perceptron underlying *threeOfAKind* gives large weight to two of these, while the others receive small weight.

All learned knowledge becomes available as a potential basis for any unlearnable knowledge. However, the question of which learned predicates are appropriate to a specific unlearned predicate remains, and is the subject of the next section.

4. MEMORY ORGANIZATION IN SCALE

There are three constraints on building-block organization methods. The first stems from the goal of designing a system capable of long-term learning. The number of building blocks may become quite large, so the organizational method must scale up. Specifically, the number of comparisons between building blocks must grow slowly. Notice that even pairwise comparisons among blocks will become impractical over time. The second constraint arises from the organizational process itself. The goal is to reduce the number of potential inputs considered by each building block during training. Thus there will be no standard input vector describing all building blocks. Finally, the goal of memory organization in SCALE is to select a set of related dependencies, not a minimal set of inputs. This allows useful redundancy and more flexibility in predicate learning.

These constraints suggest that SCALE's knowledge organization problem does not fit into the formulations popular in the machine learning community, such as feature selection and data clustering. Both methods tend to rely heavily on pairwise comparisons and on the presence of a common input vector. These approaches will be computationally expensive with respect to SCALE

SCALE uses a combination of bottom-up (stimulus-driven) evaluation, and fast variable selection algorithms to establish dependencies. Bottom-up evaluation eliminates many potential inputs to an unlearned predicate by evaluating only predicates related to the current stimulus. Two variable selection algorithms then work in conjunction with each predicate's perceptron to prune further the list of inputs.

4.1. Bottom-Up Evaluation

Each training stimulus in SCALE specifies three pieces of information: (1) the predicate addressed by the example, (2) the set of objects bound to the predicate's arguments (also serves as the focus of attention), and (3) the desired Boolean output. In order for an unlearned predicate to determine which of the learned predicates should serve as inputs, evaluations for all learned predicates must be determined with respect to the focus of attention.

SCALE evaluates individual predicates in a top-down manner, but a purely top-down evaluation strategy must visit every predicate in the network at least once per stimulus. Even in a moderate sized domain, many predicates may be completely irrelevant to a specific learning problem.

The bottom-up evaluation task is to *determine the values of all predicates with respect to a small set of domain objects* (the focus of attention). Contrast this to a query system, such as Prolog, whose task is to *find a set (or all sets) of bindings that make a given predicate true*. The difference between the two problems is important. A query system must search among all possible bindings to satisfy a single predicate, while SCALE must search among a limited number of possible bindings to satisfy as many predicates as possible.

4.1.1. Definitions

We begin with a formal description of predicates and training examples. Let:

- $P = (p, V_p, I_p, O_p)$ be a predicate such that

  p is a unique identifier for P,
  V_p is the set of argument variables for P
  I_p is the set of input predicates (with fixed argument mappings) for P, and
  O_p is the set of predicates to which P is an input.

- $X = (p, d, B)$ be a training instance such that

  p is a unique identifier for the target predicate,
  d is the desired output, and
  B is a set of bindings from objects to V_p.

- F be the set of objects in the focus of attention.
- $\mathcal{G}$ be the set of ground predicates, or those that do not depend on input from other features.

We also define the term *improve* with respect to a predicate evaluation. The truth value of predicate P improves the evaluation of $P' \in O_p$ if the output value of P constitutes evidence that P' will produce a positive (*true*) output value. With respect to the perceptrons underlying each predicate, P improves P' if the weighted evaluation of P is greater than zero. This brings the left side of the perceptron evaluation rule ($\sum_{i=1}^{n} w_i P_i \geq \theta$) closer to or beyond the threshold value on the right side.

4.1.2. Algorithm

The main idea of bottom-up evaluation is to evaluate predicates recursively as long as the evaluation of the current (lower level) predicate improves the evaluation of its successor (higher-level) predicates. Notice that each unique binding from the focus F to arguments V_p constitutes a unique evaluation path. The number of possible binding combinations is small however, since both $|F|$ and $|V_p|$ are small and constant (five here).

Table 1 shows the algorithm for bottom-up evaluation. The main loop iterates over the set $\mathcal{G}$ of ground predicates. Each predicate P is evaluated with respect to all possible bindings between F and V_p. If the evaluation improves any of P's successors, then the algorithm recurses. The bindings of P are mapped to its successor P' (against the arrows in Figure 1). In some cases P' will have more arguments than P, such as when $P = \mathit{threeOfAKind}$ and $P' = \mathit{twoOfAKind}$ in Figure 2. The extra arguments are then bound to objects in F before bottom-up evaluation continues.

4.1.3. An Example

An example from card solitaire illustrates both the method and benefits of bottom-up evaluation. Figure 3 shows a partial SCALE representation for this task. The agent has already learned several lower-level concepts, but has not yet acquired high-level concepts such as *columnStackable*. A card c_1 is column-stackable on card c_2 if the rank of c_1 is one less than the rank of c_2 and the cards are of opposite color. For simplicity we ignore domain aspects related to card rank.

Given:
Set $\mathcal{G}$ of ground predicates
Example $X = (p, d, B)$

Algorithm:
BottomUp$(\mathcal{G}, X)$ *//main loop*
  $F \leftarrow$ objects in B
  for each $P \in \mathcal{G}$ **do**
    $\mathcal{B} \leftarrow$ set of all bindings between F and V_p
    for each $B' \in \mathcal{B}$ **do**
      BUEvaluator(P, B')

BUEvaluator(P, B) *//helper procedure*
  $\mathcal{B} \leftarrow$ set of bindings between F and V_p given B
  for each $B' \in \mathcal{B}$ **do**
    bind B' to V_p and evaluate P
    for each $P' \in O_p$ **do**
      if P' is *learned* **and** P *improves* P' **then**
        $B'' \leftarrow \mathrm{map\_bindings}(P, P')$
        BUEvaluator(P', B'')

Table 1. Online algorithm for bottom-up evaluation.

We begin with the assumption that the agent has just received the following training instance for a new concept: *column-Stackable(*$v_1 \equiv 6\heartsuit$*,* $v_2 \equiv 3\diamondsuit$*) = false*. Thus the agent's focus of attention (the set F) consists of the six of hearts and the three of diamonds. The set of ground predicates $\mathcal{G}$ consists of predicates *spade* and *diamond*.

Bottom-up evaluation first selects a predicate from $\mathcal{G}$, say *spade*, and an initial binding from F, $y \equiv 6\heartsuit$. The predicate evaluates, resulting in *spade*(6♡) = *false*. Next each of *spade*'s output connections is tested. The output value of *false* does not improve *black*, so evaluation halts there. The predicate *heart* is improved (*heart* is *true* if the card index is less than 26 and not a spade), so evaluation continues along that path.

The next step is to map argument bindings. The 6♡ from y in *spade* to w in *heart* (moves against the arrow in Figure 3). Now the predicate *heart*(6♡) evaluates to *true*. *Heart*'s output connections are then tested revealing that *red* is improved by the result *heart*(6♡) = *true*. The algorithm therefore recurses and finds that *red*(6♡) = *true* (after a top-down evaluation of *diamond*), which improves *redBlack*. Another round of recursion yields that *redBlack*(3♢, 6♡) = *false*. This outcome does not improve *colorsDiffer*, so the recursion finally unwinds back to *spade*.

The second round of bottom-up evaluation begins by binding $y = 3\diamondsuit$ in *spade*. Only two evaluations are performed, on *spade* and *heart* before the recursion unwinds. The algorithm then shifts to the *diamond* predicate and proceeds in a manner very similar to that of *spade*.

There are two points to note in this example. First, given the focus of attention (6♡, 3♢), the predicate *colorsDiffer* is never evaluated. Second, evaluation along many other paths through the representation halts prior to reaching all of the predicates along the path. Many predicate/binding combinations, namely those unrelated to the input stimulus, go untested. Ultimately, these combinations are eliminated from consideration during input selection at no computational cost. Only predicate/binding combinations explicitly evaluated during bottom-up evaluation are eligible for input selection.

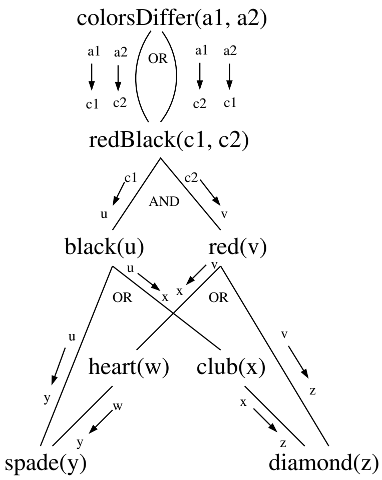

Figure 3. Partial SCALE representation for solitaire.

4.2. Forward Input Selection

SCALE's approach to input selection is founded on the limited learning assumption in two ways. First, many learned predicates, as highlighted by bottom-up evaluation, can be ignored during the selection process. The portions of a representation unrelated to a given training point are simply not good candidates. The second aspect relates to the simplicity of the perceptron. More powerful learning algorithms can detect and represent subtle relationships between inputs and output, but this makes predicting the utility of a given input difficult. The perceptron can represent only simple relationships, so predicting which inputs may be useful, without incurring the cost of training, is relatively simple.

There are only two basic types of inputs to a perceptron. Both cases can be coarsely detected without training the perceptron. *Excitatory inputs* indicate when the perceptron should evaluate positively. To detect an excitatory input, SCALE computes the conditional probability $s^+ = \Pr[P_{out} = \mathit{true} | P_{in} = \mathit{true}, \mathrm{Data}]$ that the unlearned (output) predicate is *true* given that the learned predicate is *true*. Probabilities closer to one indicate a stronger relationship (more overlap) between P_{in} and P_{out}.

Inhibitory inputs indicate when the perceptron should evaluate negatively. Detecting inhibitory inputs is similar to detecting excitatory inputs. SCALE computes the conditional probability $s^- = \Pr[P_{out} = \mathit{true} | P_{in} = \mathit{false}, \mathrm{Data}]$ that the unlearned predicate is *true* given that the learned predicate is *false*. Note that while both measures produce values in the range zero to one, they are not complementary (they do not sum to one).

Given these detection measures, the algorithm for se-

Given:
Set of predicates $\mathcal{E}$ evaluated via BottomUp($\mathcal{G}, X$)
Predicate $P = (p, V_p, I_p, O_p)$
Example $X = (p, d, B)$

Algorithm:
Select($\mathcal{E}, P, X$)
  for each $P' \in \mathcal{E}$ **do**
    update s^+, s^-
  if P has seen enough examples **then**
    $S \leftarrow$ highest score predicates in $\mathcal{E}$
    $I_p \leftarrow I_p + S$
    initialize P for learning

Table 2. Online algorithm for selecting predicate inputs.

lecting inputs is straightforward. For a given training example, the learned predicates are first evaluated bottom-up. In the remaining steps, only predicates explicitly evaluated by the bottom-up procedure are considered. The excitatory and inhibitory scores are computed over several training instances. Candidate predicate(s) with the highest score are then added as input to P_{out}. The predicate P_{out} then attempts to learn its target concept. Table 2 lists the input selection algorithm.

4.3. Solidifying the Structure

So far, memory organization in SCALE has been primarily concerned with reducing the number of potential inputs to unlearned predicates. The combination of bottom-up learning and forward input selection typically restricts the number of new inputs acquired by an unlearned predicate to just one or two at a time.

After the unlearned predicate selects new inputs, it returns to training its perceptron. If the perceptron fails to learn successfully, the predicate again tries to acquire new inputs, repeating this process until the predicate is successfully learned. Note that this select-then-train procedure may loop many times before the necessary basis is acquired. This is particularly true if the predicate represents very high-level knowledge.

When learning is successful, the final stage of pruning unnecessary inputs begins. Pruning is important for two reasons. First, bottom-up evaluation is most efficient when the number of inputs to a given predicate is small. Unnecessary inputs can quickly lead to unnecessary predicate evaluations. Second, the presence of extra inputs can affect generalization adversely.

SCALE employs an online version of the Randomized Variable Elimination (RVE) algorithm (Stracuzzi & Utgoff, 2004). Briefly, RVE is motivated by the idea that, in the presence of many irrelevant variables, the probability of successfully selecting several irrelevant variables simultaneously at random is quite high. The algorithm computes the cost of attempting to remove k input variables of n remaining variables given that r are relevant. A sequence of values for k (given n and r) is then found by minimizing the aggregate cost of removing all $N - r$ irrelevant inputs. Note that n represents the number of remaining variables, while N denotes the total number of variables in the original problem.

In summary, SCALE uses a combination of implicit and explicit knowledge organization. Much of SCALE's learning effort goes toward feature selection, which explicitly sets knowledge dependencies. However, the implicit organization produced through bottom-up evaluation plays a key role in maintaining efficiency. SCALE's dependency selection algorithm may be viewed as a two-part stepwise feature selection algorithm. Bottom-up evaluation serves as a filter for the set of considered dependencies, while the conditional probability measures alternate with the perceptron learners in a wrapper selection approach. Alternatively, dependency selection in SCALE may e viewed as a predicate clustering algorithm operating over a subset of the predicates (selected by bottom-up evaluation) and using the conditional probability measures as the similarity metric.

5. AN APPLICATION

We demonstrate SCALE's organizational skills on an expanded version of the cards domain discussed throughout this article. The goal here is to demonstrate the influence of memory organization on the larger knowledge acquisition process. Of particular interest are changes in the number of training stimuli required by the agent, changes in the amount of computation required for learning, and the agents general ability to detect relationships among knowledge building blocks (predicates).

5.1. Cards Domain Overview

The cards domain is composed of predicates related to the rules of solitaire. As previously noted, the agent must also learn to recognize the suit, rank and color of cards based solely on the card index. There are a total of 33 predicates, two of which are ground predicates. The domain representation requires nine layers as conceived by the author, although SCALE may find a different organization.

Training data were generated at random and with replacement during execution. Thus, the recorded training times include the cost of data generation. Table 3 shows the final results produced by SCALE. The left side of the table shows the order in which concepts became *learned*, along with the CPU time, number of inputs and number of examples required for learning. The right side shows the same statistics, along with the concept layer (or longest path from concept to a ground predicate), ultimately achieved after all unnecessary inputs were pruned.

Note that the "learned inputs" column also shows the total number of inputs available when the concept was learned. Each arity-1 concept may take one input from all other previously learned arity-1 concepts, while each arity-2 concept may take *two* inputs from *all* previously learned concepts. All times are aggregated across all concepts, and are relative to a 1.13 GHz Pentium III processor.

	Learned				*Completed*				
Concept	*Order*	*Time (h:m:s)*	*Inputs*	*Examples*	*Order*	*Time (h:m:s)*	*Inputs*	*Examples*	*Layer*
diamond	1	1	0 / 0	3117	1	1	0	3170	0
spade	2	2	0 / 1	5498	2	2	0	5542	0
red	3	4	2 / 2	9528	3	4	2	10287	1
black	4	4	2 / 3	9697	4	5	2	10711	1
redBlack	5	9	5 / 8	17155	6	24	2	32374	2
club	6	9	1 / 4	17363	5	10	1	18581	1
colorsDiffer	7	11	10 / 12	20270	7	29	3	38380	3
heart	8	14	2 / 5	5903	10	1:29	1	26945	1
bothClub	9	27	3 / 16	36006	8	42	2	52106	2
bothSpade	10	59	2 / 18	72548	9	1:06	2	80051	1
king	11	1:05	6 / 6	79582	11	2:48	5	185328	2
ace	12	1:22	6 / 7	98208	14	4:11	5	278708	2
queen	13	1:51	7 / 8	129383	15	4:16	5	285043	3
rankOneOrLess	14	2:36	6 / 26	173808	16	4:46	6	323207	2
bothHeart	15	2:53	5 / 28	190016	12	3:16	4	216026	2
rankGreater	16	3:07	8 / 30	206407	20	10:53	6	876874	2
rankLess	17	3:31	13 / 32	234160	13	4:03	1	270214	4
jack	18	4:11	9 / 9	139404	21	13:57	7	590953	4
sameSuit	19	4:14	13 / 36	281962	17	5:46	7	401074	4
two	20	4:19	9 / 10	288215	22	16:04	6	1367719	4
ten	21	5:07	10 / 11	349702	19	10:03	8	792007	5
bothDiamond	22	6:59	2 / 42	506307	18	7:03	2	511705	1
three	23	7:42	10 / 12	286810	26	41:32	6	1725779	5
nine	24	12:00	12 / 13	322625	25	39:36	10	1091119	6
eight	25	13:40	14 / 14	1155257	24	38:58	9	3233442	7
four	26	14:06	13 / 15	397885	28	1:05:04	10	2710101	6
five	27	18:27	15 / 16	514755	33	1:14:58	9	4026922	7
seven	28	19:15	15 / 17	1042457	30	1:13:01	9	5132714	8
rankSame	29	20:56	14 / 56	589233	23	22:17	4	626534	4
six	30	46:57	18 / 18	1607050	27	50:54	12	1740028	9
rankOneGreater	31	1:10:03	38 / 60	3694841	29	1:11:42	4	3827154	5
columnStackable	32	1:12:15	28 / 62	5618831	31	1:13:57	2	5923287	6
bankStackable	33	1:12:49	32 / 64	2290050	32	1:14:02	2	2428876	6

Table 3. Results of running SCALE on the cards domain.

5.2. Results

There are two main points to note about SCALE's performance. First is that most concepts are learned quickly. The first 29 concepts (predicates) learn in just over 20 CPU minutes total, and all learn within approximately 75 CPU minutes total. Removing irrelevant inputs requires more CPU time, but this is expected. Improvements to the dependency structure must not destroy existing knowledge, and therefore must be made carefully. This is known as stability (Quartz & Sejnowski, 1997).

The second key point is the general success of the organization process. All concepts completed initial learning using substantially fewer inputs than the number available. For example, *rankSame* selected only 14 of the 56 available inputs. This reduction serves to reduce evaluation costs, reduce training costs (notice he large number of examples required by the concepts that were less successful in input selection), and improve concept generalization.

To further illustrate the effect of knowledge organization, consider the learned structure as a whole. Given the order in which concepts were learned, a total of 661 dependencies were available. Less than half of these available dependencies were selected during concept learning (ignoring the subsequent removal process). The number of dependencies in the final structure (after removal) was just 154. The structures used by SCALE during learning were therefore very sparse compared to the structures that would have been used by an agent with no knowledge organization.

6. DISCUSSION

One important aspect of memory organization is flexibility. The structuring of an agent's knowledge must permit a broad range of relationships among building blocks. The alternative is redundant learning and recreation of existing knowledge. In SCALE, building blocks are not explicitly clustered or physically configured, although the weighted connections learned by the perceptrons do indicate knowledge dependencies. This approach has the advantage of being both sufficiently structured to provide learning efficiency, yet unrestrictive with respect to opportunities for knowledge transfer.

Flexibility in memory organization may also be considered as a form of unsupervised learning. The agent is free to select whichever organization allows the building-blocks to be learned. There is no explicit training signal to indicate dependencies.

With respect to representation, Valiant's (2000) theoretical neuroidal architecture is similar to SCALE. The architecture uses a first order representation with limits on predicate complexity, arity and quantification. However, Valiant assumes that local inputs and outputs are available for each predicate during training to simplify theoretical analysis. SCALE produces the inputs to each predicate from the low-level stimulus, making no assumptions about the accuracy of the result. The neuroidal architecture also makes no attempt at knowledge organization.

SCALE also bears a broad resemblance to Bayesian networks, which represent relationships among random variables probabilistically. Although SCALE's perceptrons cannot be considered equivalent to the conditional probability tables associated with nodes in a Bayes net, they do represent the strength of dependence between two entities. Predicates in SCALE are also strictly more general than random variables in Bayes nets. With respect to learning however, SCALE may be viewed as an incremental algorithm for learning deterministic (noise-free) Bayes nets with missing data. Missing data here refers to the assumption that data appears in a stream of input-output pairs for individual concepts instead of an entire vector describing all concepts simultaneously. Thus the SCALE learning problem is both easier (deterministic) and harder (incremental and missing data) than standard Bayesian network learning.

Although SCALE is not intended to act as a model for human cognition, the method does resemble some of the high-level processes. For example, bottom-up evaluation is related to stimulus-driven neural activation. Similarly, if we view the accumulation of values in the s^+, s^- measures as a form of "growth", then the forward input selection process is reminiscent of axonal growth. Likewise, pruning of unnecessary connections from a predicate is similar to the regression of unused dendritic pathways. Unnecessary inputs exert little influence over predicates, and are allowed to "wither away".

7. CONCLUSION

Memory organization plays a critical role in knowledge acquisition. A learning agent must be able to apply existing knowledge to new learning problems. However, distinguishing related and unrelated knowledge is a challenging task. Without some organization method, every piece of existing knowledge must be considered as possibly related to the learning problem. This creates a situation in which each new learning problem becomes more complex than the previous. The SCALE algorithm provides a simple, yet effective approach to memory organization. The algorithm uses a combination of implicit and explicit organizational techniques to improve the learning agent's ability to acquire new knowledge.

8. ACKNOWLEDGEMENTS

This material is based upon work supported by the National Science Foundation under Grant No. 0097218. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation. Paul Utgoff, Gary Holness, Steve Murtagh, Philip Kirlin and the anonymous reviewers provided helpful comments on this article.

9. REFERENCES

Burges, C. J. C. (1998). A tutorial on support vector machines for pattern recognition. *Data Mining and Knowledge Discovery*, *2*(2), 121–167.

Clark, A., & Thornton, C. (1997). Trading spaces: Computation, representation, and the limits of uninformed learning. *Behavioral and Brain Sciences*, *20*, 57–97.

Miller, G. A. (1956). The magical number seven, plus or minus two: Some limits on our capacity for processing information. *The Psychological Review*, *63*(2), 81–97.

Minsky, M., & Papert, S. (1972). *Perceptrons: An introduction to computational geometry (expanded edition).* Cambridge, MA: MIT Press.

Quartz, S. R., & Sejnowski, T. J. (1997). The neural basis of development: A constructivist manifesto. *Behavioral and Brain Sciences*, *20*, 537–596.

Stracuzzi, D. J. (2005). *Scalable learning in many layers* (Tech. Rep. No. 05-02). Amherst, MA: Department of Computer Science, University of Massachusetts.

Stracuzzi, D. J., & Utgoff, P. E. (2004). Randomized variable elimination. *Journal of Machine Learning Research*, *5*, 1331–1364.

Utgoff, P. E. (1986). *Machine learning of inductive bias*. Hingham, MA: Kluwer.

Utgoff, P. E. (1989). Perceptron trees: A case study in hybrid concept representations. *Connection Science*, *1*(4), 377–391.

Utgoff, P. E., & Stracuzzi, D. J. (2002). Many-layered learning. *Neural Computation*, *14*(10), 2497–2529.

Valiant, L. G. (2000). A neuroidal architecture for cognitive computation. *Journal of the ACM*, *47*(5), 854–882.