Center for Cognitive Architectures
University of Michigan
2260 Hayward Ave
Ann Arbor, Michigan 48109-2121

TECHNICAL REPORT
CCA-TR-2006-02

Integrating Semantic Memory into a Cognitive Architecture

Investigators

Yongjia Wang
John Laird

June 30 2007

Abstract:

Semantic memory stores a person’s general knowledge about the world and plays an important functional role in generating intelligent behaviors. Semantic memory has been an active research field in psychology and is implemented in cognitive architectures such as ACT-R [1] to model various related phenomena in human. However, functionally-based cognitive architectures, such as Soar [2], have not included a semantic memory component and related functions due to their knowledge engineering oriented efforts and the limited learning demands of their tasks. Inspired by semantic memory research in psychology, and aimed to studying the general computational functionalities of semantic memory, its interactions with cognitive architecture and solving more challenging tasks requiring learning, we have started integrating a semantic memory component into Soar. This paper introduces the motivations, architectural design and initial implementations. Empirical results on two simple tasks are presented and future work is proposed.

1. Introduction

According to Tulving, "*semantic memory refers to a person's general knowledge about the world. It encompasses a wide range of organized information, including facts, concepts, and vocabulary. Semantic memory can be distinguished from episodic memory by virtue of its lack of association with a specific learning context.*" [3] In other words, semantic knowledge is not tied to specific context. For example, the semantic knowledge about what a typical bird looks like, such as having two legs, a pair of wings, sharp beak and covered by feathers, could be readily accessed without referring to memory about a particular bird, such as the canary I saw yesterday on my way home, which should be in episodic memory. More abstract factual knowledge such as 'Washington DC is the capital of USA' is also in semantic memory. Accessing it is not necessarily tied to the context of the situation when such a fact is learned. Semantic memory and episodic memory are both declarative memories in contrast to procedural memory (Figure 1), in that they store knowledge representations whose complete contents are the basis for their retrieval, and that they can be examined in working memory after memory retrieval. Besides semantic and episodic memory, there are other functionally distinctive memory systems in humans. Figure 1 illustrates a commonly accepted view of the taxonomy of human memory [3]. Working memory represents the current situation; procedural memory contains skill knowledge – how to do things; the perceptual representation system holds imagery information.

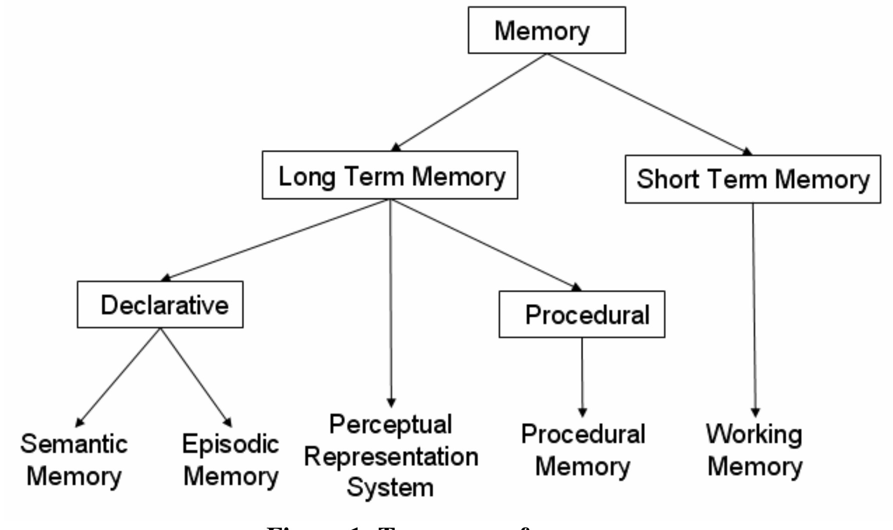


Figure 1: Taxonomy of memory

Due to the vital importance of semantic memory, a deficiency in semantic memory function will result in serious behavioral problems in human. One such disease is semantic dementia (SD), a neurodegenerative disorder related to impairment of semantic memory. Patients of SD typically show a progressive deterioration of semantic memory, while relatively preserving their day-to-day episodic memory [4]. Although precise brain mapping of semantic memory remains an open question and actively investigated topic, current hypotheses on learning and representing semantic memory involve brain regions including temporal lobes, hippocampus and neocortex.

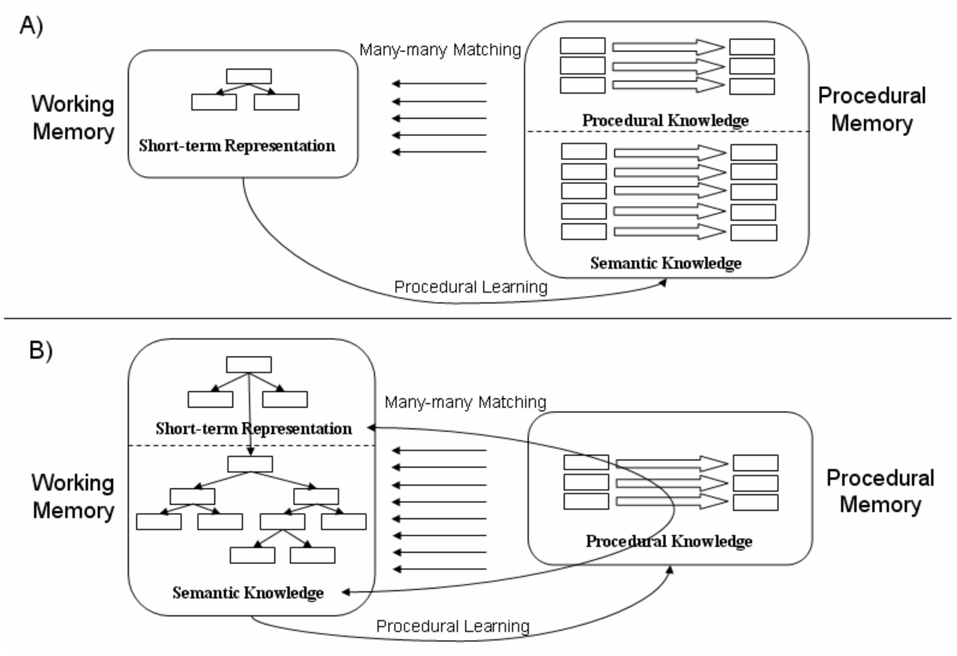
Semantic memory has long been an active area of research in psychology. In ACT-R (Adaptive Control of Thought - Rational) [1], an architecture for psychological modeling, there is a declarative memory module serving as the semantic knowledge store for tasks. There are rich psychological phenomena and human data which have been successfully modeled in ACT-R with the declarative module, such as Fan effect [5], category learning [6], theory of list memory [7], *etc.,* just to name a few. The declarative memory module is one of the most important modules in the ACT-R architecture and has been under active investigation among the community for many years.

On the other hand, semantic memory has been relatively ignored in AI architectures. Many AI architectures are production systems, which lack a long-term semantic memory component and related learning capabilities. Such architectures include Soar[2] (State Operator And Result), ICARUS[8], PRODIGY[9], EPIC[10], etc. These architectures make a distinction between a declarative short-term working memory and a procedural long-term memory (Figure 1), where the procedural memory is often represented as production rules. The purpose of working memory is to hold declarative knowledge relevant to current reasoning. It's not appropriate to use working memory as long-term knowledge store because it can interfere with ongoing reasoning when working memory holds more and more data. Abusing working memory in that manner is not only psychologically implausible, but will also hurt the performance of reasoning. Many production systems, including Soar, are based on the Rete algorithm [11] for efficient many pattern to many object matching . According to theoretical analysis, both the time and space worst case complexity of single rule firing using the algorithm is in the order of $O(W^C)$, where W is size of working memory and C is the number of patterns in the rule. Certain restrictions, such as unique attributes, can reduce matching complexity to linear but greatly restrict the expressibility of knowledge. In addition, adding large bodies of knowledge to working memory can only make such restrictions more difficult to be realized. Therefore, for just functional reasons, knowledge should be stored in a separate long-term memory.

In these AI architectures, declarative semantic knowledge can be embedded in the production rules provided to the system by the agent designer, such as through rules that retrieve relevant semantic knowledge into working memory. Unused knowledge structure will be removed from working memory to prevent it from growing overtime when performing multiple tasks; while there is always the permanent copy of knowledge in long term procedural memory so that the above performance issue will not happen. Practically, the performance degradation is also much less sensitive to the number of production rules than to the size of working memory. However, using a single procedural memory will result in problems from at least two aspects. One aspect comes from the accessibility of the knowledge representation: production rules are not appropriate for representing explicit semantic knowledge, as rules cannot be flexibly accessed by reasoning processes but only triggered under specific conditions. The other aspect concerns the capability of learning semantic knowledge, such as brought up by the frequent question: 'where do those rules come from?'. Some production systems do not have any learning mechanisms, such as EPIC. For other learning systems in cognitive architectures, symbolic procedural learning has its limitation in that it is explanation based (such as the procedural learning mechanisms in Soar and PRODIGY), which are 'trapped' by the entailment of initial knowledge. Other AI architectures like blackboard systems [12] and PRS [13] can encode declarative knowledge in a dynamic database, but the main effort is still knowledge engineering and not learning.

Contrary to production systems which encode all long-term knowledge as production rules, traditional knowledge representation systems, such as KLONE [14] and Cyc [15] organize all knowledge declaratively. The purpose of these systems is to build up general ontology and common sense knowledge base for generic reasoning systems. Again, they are knowledge engineering efforts that do not emphasize learning from ongoing experience and do not address the general issues of integrating separate declarative knowledge systems with arbitrary task-specific procedural reasoning.

The aforementioned declarative learning problem has been relatively ignored in AI architectures, mainly due to the limited demand of the tasks they have been dealing with. The tasks are mainly knowledge engineering problems, where the primary focus is to efficiently organize and utilize available procedural domain knowledge by creating high-performance systems, such as TacAir Soar [16]. Knowledge discovery is usually not the concern. Nevertheless, Soar has been used in some projects requiring learning new knowledge, such as category learning [17], instruction taking [18], where data learning problem [19] is always involved, and the chunking solution has not been satisfactory. With the demand of dealing with increasingly complex tasks or exploring in novel environments, where existing domain knowledge is incomplete or liable to contain errors, one option that requires exploration is separate representation and learning mechanisms for semantic knowledge, namely, a long term declarative memory. Compared to the long-term procedural semantic memory approach (Figure 2A), a separate declarative memory (Figure 2C) with flexible matching algorithm increases the accessibility of knowledge, as well as the reasoning power and learning capabilities. Compared to including all semantic knowledge in working memory (Figure 2B) which impacts reasoning performance all the time, a separate declarative (Figure 2C)memory only impacts the performance when the knowledge is needed (retrieving information from a large body of knowledge is inevitably more expensive). In addition, the separate component will allow designs of semantic learning and retrieval algorithms that are specific to the needs of desired functionalities.



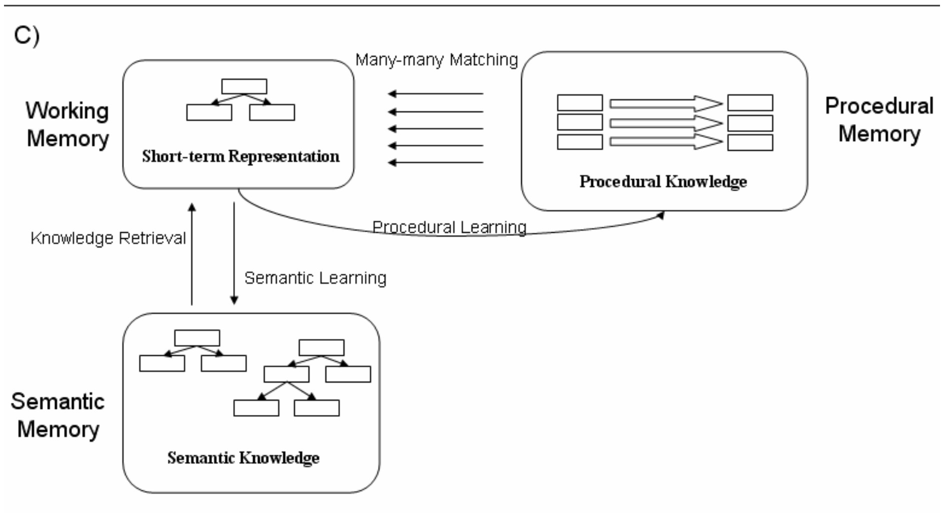


Figure 2: Alternative approaches to semantic learning in AI architectures

Figure 2 illustrates and compares the alternative approaches mentioned above. The big boxes on the left are declarative memories (with declarative representation inside), and on the right are procedural memories (with rules inside). 'Short-term representation' refers to knowledge and beliefs relevant to current reasoning. 'Semantic knowledge' is about general facts and 'procedural knowledge' is about reasoning procedures and controls. Figure 2A is the approach to represent all long term knowledge in procedural memory. The problem with this approach is the accessibility and the ability to learn the procedural representation. Figure 2B is the approach to keep declarative knowledge in working memory. This results in performance degradation as working memory size grows. Figure 2C is the currently proposed approach to add a separate semantic memory component. In addition to obviating problems associated with the above approaches, a separate semantic memory component will open the design space for retrieval and learning algorithms to investigate the functionalities related to semantic memory. For the previous two approaches knowledge retrieval is achieved with and restricted by existing production rule matching algorithm, which is optimized for fast rule matching and not for semantic knowledge retrieval.

Both because of the current functional limitation of Soar, and inspired by the ACT-R architecture, we embarked on integrating a semantic memory component into Soar. ACT-R's declarative module is by far the most detailed and mature model of semantic memory and our current design will share many features with ACT-R, such as declarative representation and memory retrieval. However, integrating semantic memory into Soar faces with many different challenges due to the different underlying assumptions between the two architectures. These assumptions include the working memory representations, the decision making process and the existing learning mechanisms. For example, ACT-R does not distinguish between semantic memory and episodic memory, but attempts to provide the necessary functionality for both with a single declarative memory. While in Soar, the distinction between the two memory systems are enforced, because Soar's more complex (multi-level) working memory representation requires different treatment between 'context' and 'knowledge' during learning and retrieving (more explicitly, episodic memory encodes and retrieves complete working memory snapshots and uses them for case based reasoning, while semantic memory extracts consistent substructures that represent general knowledge independent of context). Other functions such as helping to learn prototypes via generalization over instances, which may be involved in many Soar tasks, also forces such distinction. Currently, the relation between episodic memory and semantic memory is still under debate in neuro-psychology, and Soar will provide a unique opportunity to investigate the computational implications of this distinction. Finally, ACT-R and Soar have been used for different purposes. ACT-R is designed for modeling human behavior and matching human data, while the goal of Soar has moved towards building functional AI applications.

In this proposed approach, semantic memory may appear to be like a database of knowledge, where learning corresponds to data entry insertion and knowledge retrieval corresponds to database query. However, what distinguish semantic memory from standard databases are some specifically desired properties for communicating with Soar, as well as some specific performance requirements. Semantic memory should support representations inherent to Soar (working memory representation) and learning of such representations. Its internal dynamics should support statistical operations for desired statistical sub-symbolic learning effects beyond the symbolic interface to working memory. It should integrate more advanced machine learning techniques for enhanced learning capability, as learning is the major focus. It should also provide speed-accuracy tradeoffs for real-time performance in highly interactive environments, which implies the requirement of a more subtle communication ‘protocol’ between semantic memory and Soar. Solutions to these problems are non-trivial and will require continued research efforts. Furthermore, there will probably be more issues emerging along the path.

In this paper, we investigate the different aspects of semantic memory capabilities via empirical tasks and evaluations. The purpose is to determine whether semantic memory indeed enhances Soar’s general capability, how do new functions interact with the rest of the system, and what critical functionalities are still missing. Moreover, working with empirical tasks will help generate methods for using semantic memory to deal with more challenging tasks.

2. The Soar Architecture

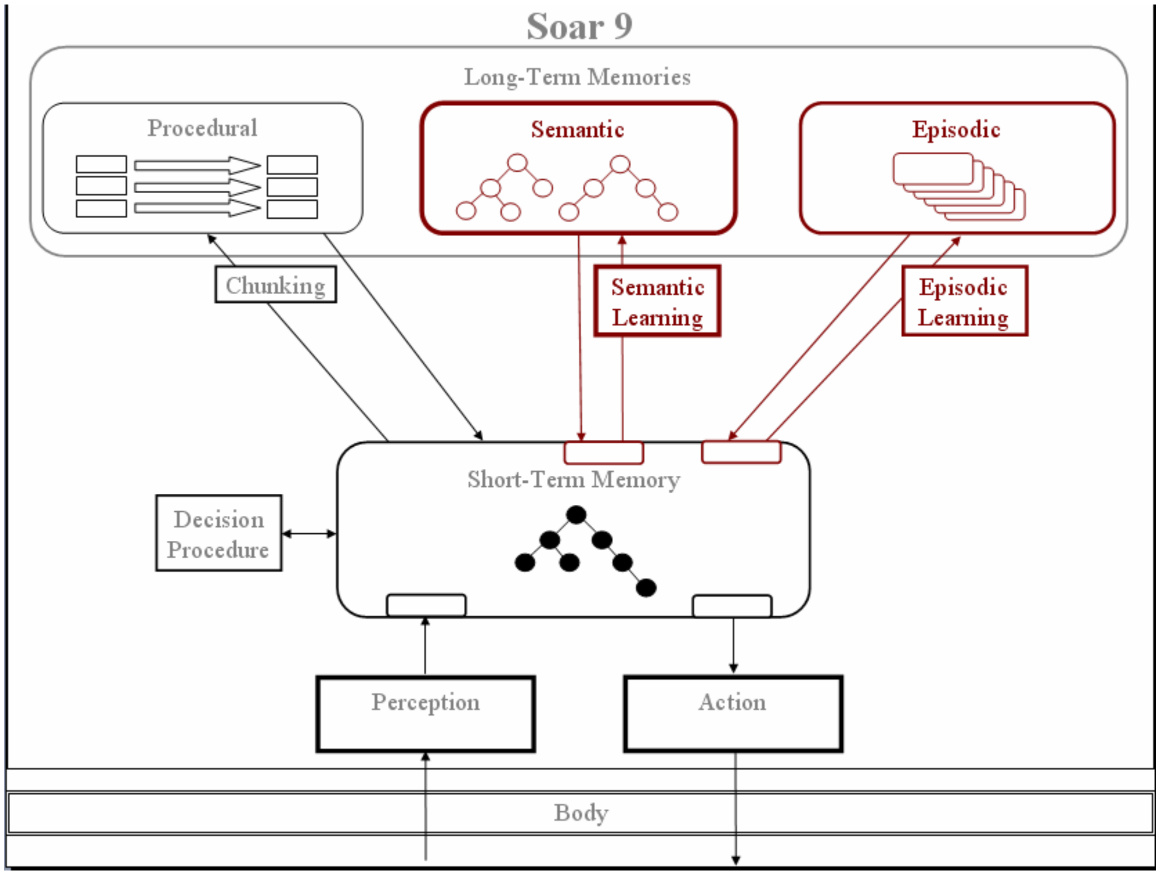


Figure 3: Soar Architecture

Figure 3 illustrates the new Soar architecture with semantic memory, episodic memory and corresponding learning mechanisms. Both of the memory systems are under development and not in official releases yet. Each of the memory systems in Soar will be briefly introduced and compared below.

Working Memory is the short term memory, and should have limited capacity as stated in the introduction section. However, there is no architectural commitment on limit of working memory capacity in Soar. It encodes relevant information for ongoing reasoning in static declarative form, which can be viewed as the internal/mental representation of current situation. The atomic unit of working memory structure is called working memory element (wme), which consists of a triple of identifier, attribute and value (value can be either another identifier or a constant). Therefore, the entire working memory of Soar is a graph structure. It has a single root, the 'state' identifier. Each working memory element obtains its 'context' as the relative path to the 'state' identifier (Figure 4). Soar systems interact with an external environment via architectural working memory structures designated as the input-link and output-link.

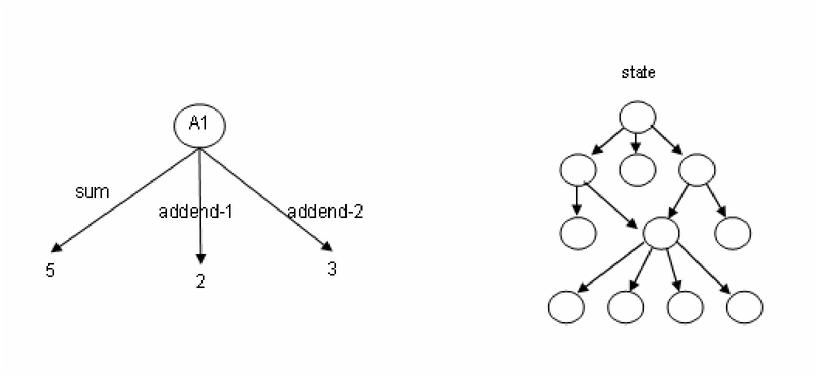


Figure 4: Working Memory Structure

Figure 4 shows an addition fact, 2+3=5, represented by three working memory elements (wmes) on the left. The entire working memory structure is rooted from 'state', which is shown on the right.

Production Memory encodes procedural knowledge as production rules with condition→action pairs. In Soar, multiple production rules can fire in parallel within a single phase (elaboration phase and operator application phase). Production rules match against short term working memory elements, and must match from root 'state' (required for each rule) The matching algorithm is implemented by a Rete [11] network, which is optimized for many-to-many exact matching.

Soar can compile steps of problem solving into a new rule, using a process called *chunking* [2], so that over time, problem solving in subgoals is replaced by rule-driven decision making. Chunking has been the only learning mechanism in official Soar releases.

Episodic Memory is about specific events. Soar's episodic memory stores the entire snapshot of working memory images which encodes the agent's experience [20].

Episodic memory is context sensitive in that the units of storage and retrieval are entire working memory structures including the 'root' (state) which contains the complete context of the original situation. In contrast to semantic learning, episodic learning remembers events and history that are embedded in experience, while semantic learning extracts facts from their experiential context.

Another distinguishing feature of episodic memory is the 'temporal awareness' – information about the chronological order of episodes is somehow encoded. Different from exact rule matching, what is retrieved from episodic memory is the best partial match. Only one episode is retrieved at a time.

Semantic Memory is about general facts independent of a specific context. One intuitive way of thinking about the difference is 'remembering' (episodic) vs. 'knowing' (semantic). The storage and retrieval unit of semantic memory are groups of attribute-values pairs describing a coherent concept or object, such as the addition fact in Figure 4. For retrieval, semantic memory works the same as episodic memory - what is retrieved from semantic memory is the best partial match. But the cue for semantic memory does not need to specify the complete path to top-state (arbitrarily reduced context). The four memory systems in Soar are compared in Table 1. We want to take advantage of the different memory schemas with different performance characteristics for different purposes.

Table 1: Memory schemas properties comparison

	Production Memory	**Working Memory**	**Episodic Memory**	**Semantic Memory**
Representation	Procedural	Declarative	Declarative	Declarative
Persistency and capacity	Long term memory	Short term memory, limited capacity	Long term memory	Long term memory
Matching	Exact match of condition. All matched rules are fired	NA	Partial match of the episode. Single best match is retrieved.	Partial match of the declarative chunk. Single best match is retrieved.
Context specificity	Complete context		Complete context	Reduced context
Temporal awareness (chronological order)	No	No	Yes	No
Learning	Chunking	NA	Episodic learning	Semantic learning

The main execution loop of Soar is called a decision cycle, which consists of fixed phases. (Figure 5). In input phase, input from external environment enters working memory. In the elaboration phase all matched rules fire to elaborate the current state and propose operators that are applicable in current state. Then multiple applicable operators are compared based on preference knowledge and only a single operator is selected. The selected operator is applied by firing associated application rules that specify the actions to be performed. At the end of the decision cycle, new outputs are generated to communicate with external environment.

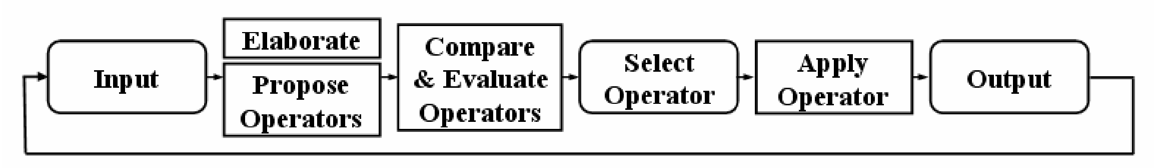


Figure 5: Soar Decision Cycles

3. Semantic Memory Design

In this section, a systematic framework for semantic memory design, and more generally, for adding any new memory component, is briefly introduced.

Any system dealing with memory must handle the following phases: encoding, storage, retrieval and use. Encoding and storage can be grouped together as the acquisition phase, retrieval and use can be grouped as application phase. Under each phase, there is a list of design decision points with a list of options, which is open to be extended.

From the integration point of view, the design needs to be constrained by the rest of the system. According to this view, the design decision points can be grouped as either related to the interface between the memory and the rest of the system or completely internal to the semantic memory component (Table 2).

Table 2: Considerations of the design

Knowledge Cycle / Integration		**Interface**	**Internal**
Acquisition	**Encoding**	Encoding initiation Target determination	Knowledge integration
	Storage		Storage structure Storage dynamics
Application	**Retrieval**	Retrieval initiation Cue determination Cue specification Retrieved result representation Retrieval meta-data	Retrieval algorithm
	Use		

The following are detailed descriptions about the phases of semantic memory according to the general framework

Encoding

- **Encoding initiation** (Interface):

This decision is about how semantic memory encoding is initiated. The options here are deliberate initiation or automatic initiation. 'Deliberate' initiation means encoding is controlled by domain knowledge, which in Soar means controlled by production rules. By using deliberate initiation, the decision can be made with task-specific knowledge. 'Automatic' initiation means encoding is initiated based on general task independent information. The options for initiating encoding include: every decision cycle when an element is being added or removed, or based on some general task-independent features such as when there are significant changes in working memory.

- **Target determination** (Interface):

This decision is about what are the structures that need to be saved into semantic memory. There are again the options of deliberate determination and automatic determination. Deliberate target determination will be controlled by rules. For automatic determination, there will be several options, such as picking every working memory structure, picking the working memory structure most frequently tested by rules, or picking the one with certain connectivity features.

- **Knowledge integration** (internal):

This decision is about how the newly added knowledge is to be integrated with existing knowledge. It's very unlikely that all declarative chunks are independent structures. One option is to let similar structures combine with each other, which may result in effects such as automatic consolidations and saving of storage space. One of the simplest forms of knowledge integration is that when identical structures are repeatedly encoded, they are merged together.

Storage

- **Storage structure** (internal)

The structure of semantic memory storage, or the representation, depends on the architecture upon which it is built. The structure must be compatible with the other design considerations of the memory system. One possible design consistent with the working memory structure of Soar is a graph structure, with additional meta-information such as when the structure was stored, the number of times the structure has been stored, and so on.

- **Storage dynamics** (internal)

How semantic knowledge changes over time. This may include merging or cross indexing with other memories (such as episodic memory, or imagery information), or removal from the semantic store. The dynamics may result in phenomena such as forgetting, implicit learning and concept formation, *etc.*

Retrieval

- **Retrieval initiation** (interface)

Acquired knowledge that is relevant to the current situation is put into working memory to assist reasoning. Similar to the considerations for encoding, retrieval can be triggered either deliberately by rule firings or automatically by task independent mechanisms.

- **Cue determination** (interface)

A cue is the stimulus that triggers a response. For semantic memory retrieval, the cue is the structure used to retrieve relevant knowledge, which should 'match' the cue according to certain criteria. Cue determination should be task-dependent. It is natural to couple cue determination with retrieval initiation.

- **Cue specification** (interface)

What is the language to specify the retrieval cue? What should be the expressiveness? Should it support variable, negation, disjunctions and other relations? For example, can you say something like retrieve a creature that has the same number of wings as legs (variable and relation), either has feather or fur (disjunction), but not a bird (negation). There will be tradeoffs related to expressiveness and computational complexity of the retrieval algorithm.

- **Retrieval algorithm** (internal):
What is the actual algorithm used to perform retrieval? The algorithm will be constrained by both the storage and the cue structures. Related issues are speed-accuracy and space-accuracy tradeoffs.
- **Retrieved result representation** (interface)
How will the retrieved information be represented in working memory? One possible way that is minimally invasive is to represent the retrieved information as working memory structures under a specific 'retrieved' link (such as that for input and out link).
- **Retrieval meta-information** (interface)
What is the extra information needed to be returned from semantic memory? For example, if no matches are found, a failure status needs to be returned to signify such situation; if succeeded, the retrieval confidence value for best partial match might be useful. They are called 'meta' because they are about information above the level of the retrieved knowledge itself.

Use

How the retrieve knowledge is used will depend on task specific knowledge encoded by rules. For example, retrieved addition fact will be used towards solving addition problem.

4. Implementations and experiments

One of the major goals of this project is to explore the use of semantic memory in the context of a general cognitive architecture. The approach we took was to look at related tasks that are awkward or computationally expensive to solve using Soar without semantic memory. We developed general solutions using the new semantic memory mechanism and evaluated the performance of these systems using task-relevant metrics. The tasks are all relatively simple, but they allow an exploration of both the structural and functional integration of semantic memory with a general cognitive architecture.

4.1. Cognitive Arithmetic (benefit of declarative representation)

The problem with the above design is that the learned knowledge is limited by the procedural representation. An important distinction between declarative and procedural representation is how the knowledge is allowed to be accessed. Productions would be declarative if they could be examined by other task knowledge in the system (such as rules that match against rules). Productions are declarative for a human programmer, but procedural for the production system. A procedural representation has the advantage of being more efficient for retrieval, but also has the disadvantage of being inflexible.

In the second implementation of semantic memory, there is a separate declarative knowledge representation (which will be described in detail later). We will demonstrate the benefit of this approach by looking at a more complex task, the cognitive arithmetic task. In addition to demonstrating the benefit of declarative learning, this task also demonstrates how semantic learning can work together with chunking and leads to better learning performance than either learning mechanism alone. Being able to analyze the interaction among multiple learning mechanisms is a major advantage of cognitive architecture approach and has become an important goal of Soar.

4.1.1. Task description

The cognitive arithmetic task is to solve simple multi-column arithmetic problems (only subtraction and addition) by using the knowledge about basic arithmetic facts (such as 2+3=5) and primitive procedures (such as using counting to derive addition/subtraction facts). We chose this problem because it is easy to understand, is universally performed by billions of humans, and demonstrates different types of learning.

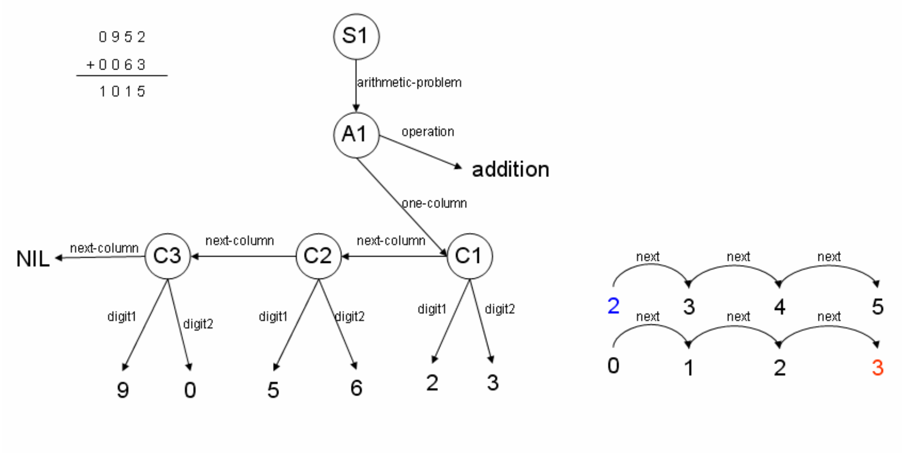
The standard addition procedure is to process the problem column by column (Figure 6 a). If the sum for a column is over ten, an extra one is carried over to next column (Figure 7). Subtraction problems follow the same procedure with minor modifications for borrowing instead of carrying. The agent initially starts without knowing the arithmetic facts (and without access to a general add or subtract operation), and must compute the facts by doing basic counting with general counting procedural knowledge (Figure 6 b). Learning happens at three different aspects in this simple task which is applicable to more general situations.

1) Specific knowledge is derived from a general procedure. The result of semantic learning is that specific facts about sums of number pairs are recorded in semantic memory as declarative chunks, so that the general counting procedure for adding a particular pair can

be replaced by retrieving from semantic memory if the same pair of numbers has been computed before. The general counting procedure requires multiple operator applications and rule firings in Soar (linear with the smallest number being added or subtracted), while the memory retrieval requires near constant time.

2) Declarative knowledge can be transferred to different situations. The benefit of learning declarative structures (compared to production rules) is that the arithmetic facts learned from addition could be reused for subtraction and *vice versa*. For example, when 2+3=5 is learned, not only does it eliminate computation when encountering 2+3=5 again, but also for 5-3=2 and 5-2=3, because both the addition and subtraction problem depend on the same arithmetic fact. In general, this is the transfer learning effect that arises from flexible declarative representation.

3) Orthogonal to transfer learning, the existing chunking mechanism in Soar can chunk over semantic retrieval and compile the entire procedure into a single rule to further speed up the execution. Although Soar could go directly from the counting procedure to chunks, it would then be unable to take advantage of the transfer provided by the semantic memory between addition and subtraction problems.



a Representation of the problem 952 + 63 **b. Procedure to compute 2+3=5**

Figure 6: An addition problem

Figure 6a shows the internal representation of a problem in Soar's working memory. There is a pointer to the operation (addition) and then a pointer to the right-most column (C1), which then has pointers to the column to its left (C2) and the two digits in the column (2 and 3). We do not attempt to model the details of human behavior (such as eye movements and writing on a piece of paper), but this is an abstract representation that is consistent with representations used in other models of addition and subtraction. Figure 6b show the processing steps to use the general counting procedure that relies on ordering facts. In this example, 2+3 is converted to the problem of adding 3 more numbers starting from 2, therefore one counter start at 2 (the top one), and one counter starts at 0 to record the absolute counts. When the bottom counter reaches the number being added (3), the top counter stops at the answer (5). This is the standard procedure, believed to be performed in human, to derive addition results only using general counting knowledge.

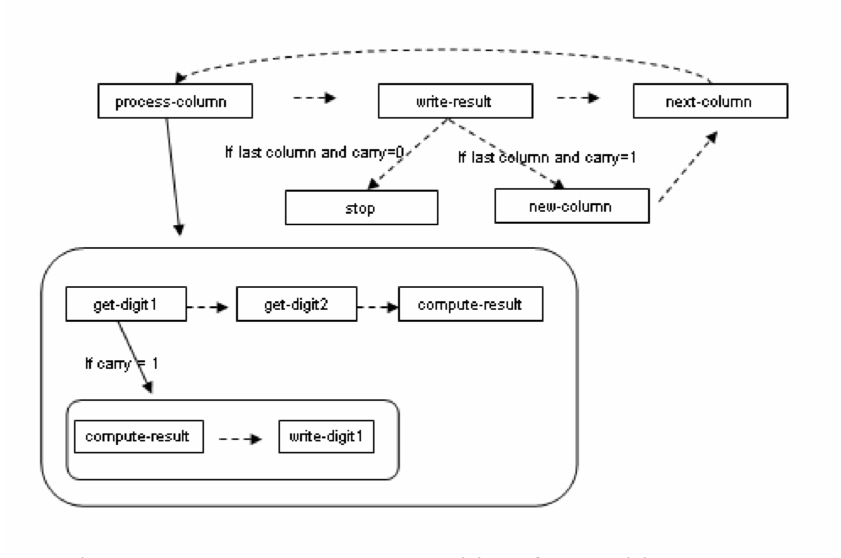


Figure 7: Problem space decomposition of an addition problem

Figure 7 shows the problem space decomposition of the knowledge used in an arithmetic task in Soar. In the top-state, the agent is in a loop of 'process-column', 'write-result' and go to 'next-column', until it finishes the last column. If there is a 'carry' after computing the last (left-most) column, a new column is created to hold the carry. Dashed arrows represent the execution flow of Soar operators and solid arrows represent the creation of a sub-state. The solid rectangle represents the created sub-state. When executing 'process-column', the agent needs to focus on the two digits being added. If the current column contains a 1 carried from previous column, the 'get-digit1' operator needs to compute the result of adding 1 to the current digit1, which is done in a sub-state. The 'get-digit2' operator simply prepares the second digit ready for the 'compute-result' operator. The 'compute-result' operator may also invoke a sub-state to solve the problem (not shown in the figure). In the 'compute-result' sub-state, the agent will do the counting procedure to find the answer, if it does not have the corresponding arithmetic fact in semantic memory. After counting, semantic learning can learn the fact by storing the result into semantic memory.

4.1.2. Implementation of declarative semantic memory

This section describes the details of the semantic memory system. The implementation is completely task independent, but will be illustrated using examples from the arithmetic domain. In this version, semantic memory has a separate storage with declarative representation. The structures it stores away are essentially copies of working memory elements. The detailed implementation is described based on our general design framework.

One major purpose of semantic memory is to save away declarative structures encoding task knowledge. Different from ACT-R, Soar working memory representation is a multi-level graph structure, which contains interleaving 'knowledge' and 'context' at the same time. Semantic memory should be able to both preserve the relevant context and efficiently answer queries about general knowledge independent of the context. More complicated issues can arise. For example, it is a common situation that the same knowledge structure can appear under different context links at different times (the same structure with different paths to root 'state', i.e. 'bird in a tree' vs. 'bird in a cage'), and it is still an open question whether such 'partially' duplicated structures should be merged into a single copy, the effect of which is multiple referencing pointers among knowledge structures are built up along semantic learning. Another related problem is whether modifying existing semantic knowledge is allowed or not, as modifying structure for one context can potentially affect the structure under different context. We currently take the ACT-R approach that identical declarative chunks will always be merged and modifying existing semantic knowledge will always result in a new copy being created in semantic memory.

- ***Encoding***

Both deliberate and automatic encoding options are available in the current implementation in order to allow experimentation. Deliberate encoding is triggered via a special working memory structure, the 'save' link. If automatic encoding is turned on, it will initiate encoding and save all the contents in working memory for each decision cycle. In the arithmetic task, automatic encoding is used.

Figure 8 shows an example, step by step, about how semantic knowledge is encoded. At decision cycle 1, fact-1 which contains two attributes is deposited into semantic memory. At decision cycle 2, fact-1 has a new attribute and the previous attributes are removed from working memory, but in long term semantic memory, all three attributes have the same identifier A1, which serves as the unique identifier for the addition fact 2+3=5. At decision cycle 3, a sub-state S2 is created, which contains another fact-1: 3+3=6, this new structure is also deposited into semantic memory. At decision cycle 4, a new fact, fact-2 which is 2+3=5 is added to working memory under S2. The new chunk A3 is added to working memory but it is identical to chunk A1, so they merged together, and the value of fact-2 under S2 becomes A1 instead A3.

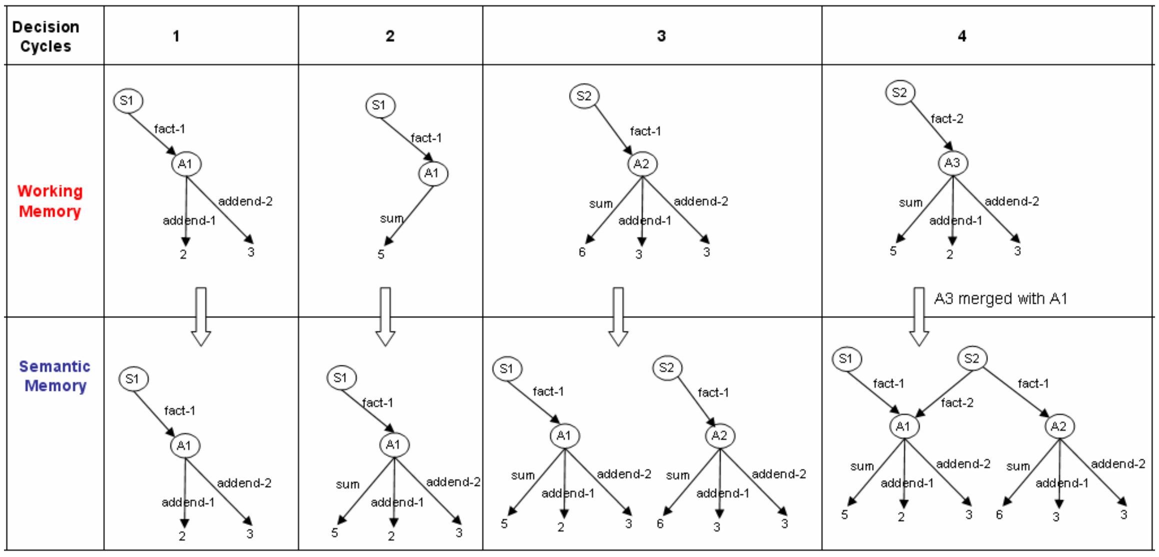


Figure 8: Encoding

- ***Storage***

At the conceptual level, the storage structures are *declarative chunks*. A declarative chunk consists of {identifier, attribute, value} triples just as working memory elements, and identifier is unique for each declarative chunk (will be simply called chunk later following the ACT-R convention, but do not confuse with chunking – the procedural learning mechanism in Soar). For efficient retrieval from semantic memory, hashing indexes on identifier, attribute and value are created respectively (Figure 9).

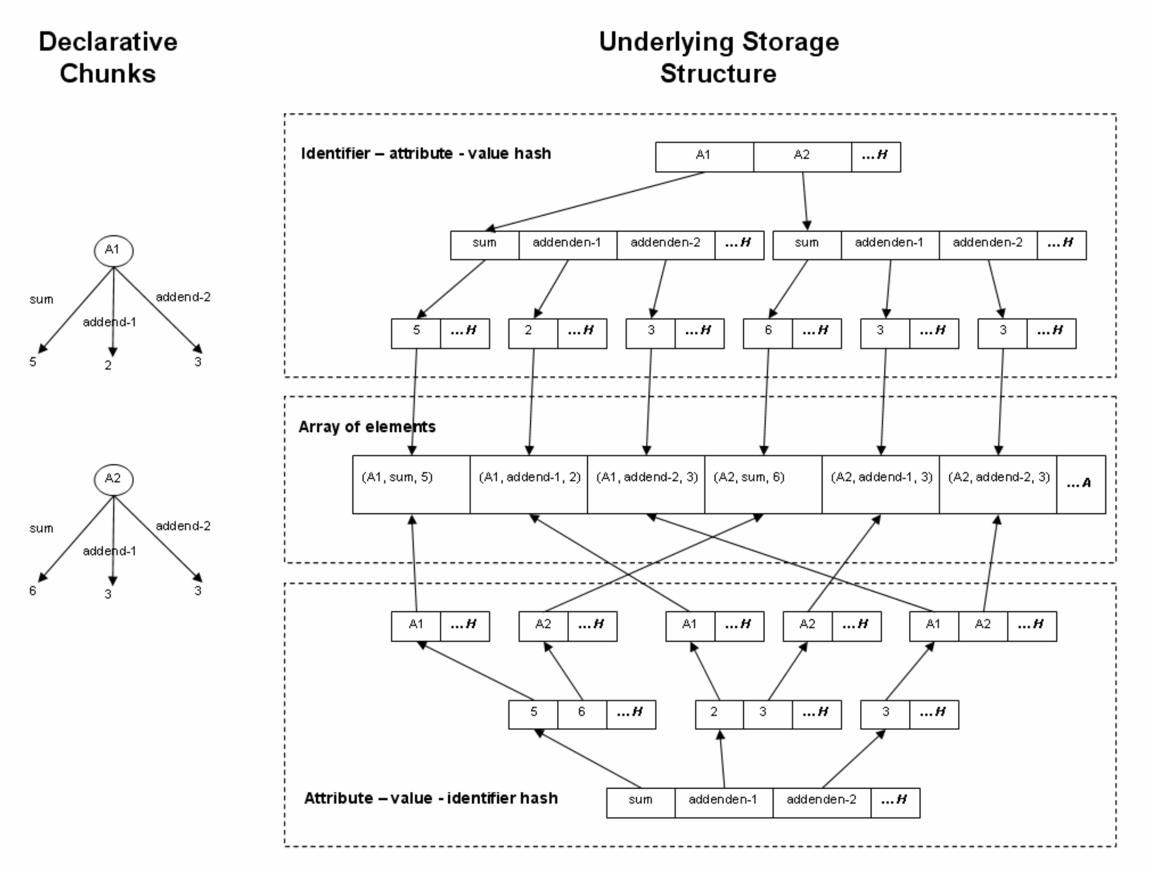


Figure 9: Storage

Figure 9 shows the semantic memory storage structure for the situation when it contains only two chunks for addition facts: 2+3=5 and 3+3=6. There are two hierarchical hashing structures to facilitate operations, such as *retrieval* and *merging* in semantic memory. '…A' denotes array data structure, and '…H' denotes hash data structure. Attribute-value-identifier hash will facilitate attribute-value based matching operations, where the top-level hash-keys are attributes, and the hash-value for each attribute is a second level hash structure containing all information related to that particular attribute. The second level hash uses values under the previous attribute as the hash-keys , which points to a third level hash, using the identifier having the same previous (attribute, value) pair as the hash-key and the pointer to an array of (thus far unique identified) triplet structures as its hash-value. The triplet element may contain extra information such as the reference history (recording the time indexes such structure is saved into semantic memory) and other data structures for the intended purposes. Similarly, the identifier-attribute-value hash will facilitate retrieving the attributes being queried based on the matched identifier.

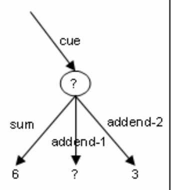
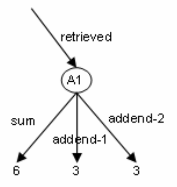
For this version, the only dynamics in semantic memory is adding new elements. There is no deliberate removal or automatic forgetting mechanisms. Nor are there automatic background learning mechanisms such as associating or consolidating similar chunks above and beyond the automatic merging that happens during encoding.

- ***Retrieval***

Retrieval initiation is always triggered deliberately via a special working memory structure – the ‘cue’ link. The cue structure is determined by task specific knowledge encoded as rules. Currently, the cue specification language supports single-level retrieval of a declarative chunk with variables (variables provide the ability for partial matches), but does not support across variable binding, negations, disjunctions and relations other than equality. None of these were necessary for this task, but may be useful in other tasks so extending the query language is an important part of the proposed future work.

The retrieval algorithm is a complete search algorithm which searches the semantic knowledge store and finds exact matches of the specified cue (variables provide the ability for partial matches). If multiple matches are found, an arbitrary selection is made. If no match is found, a failure chunk is returned. A failure chunk is the meta-information that indicates the status of retrieval. Retrieved information is put under a special working memory structure, the ‘retrieved’ link.

Cue specification	Retrieval Algorithm		Retrieved result representation
	Retrieve via attribute-value-id hash	Retrieve via identifier-attribute-value hash	

	attribute = addend-2
	value = 3 ⇨ {A1, A2}
	attribute = sum
	value = 6 ⇨ {A1}
	Final Matches ⇨ {A1}
		identifier = A1
		attribute = addend-1 ⇨ 3
			 retrieved
			A1
			sum
			addend-1
			addend-2
			6
			3
			3

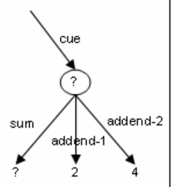
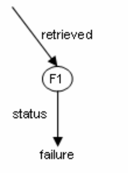
	attribute = addend-1
	value = 2 ⇨ {A1}
	attribute = addend-2
	value = 4 ⇨ { }
	Final Matches ⇨ { }
			

Figure 10: Retrieval

The retrieval procedure is demonstrated by an example in Figure 10. The content of the semantic memory is the same as in the previous example. The first retrieval is to query 3 + ? = 6. The cue structure is represented as working memory elements. The attribute-value-id hierarchical hash facilitates retrieving candidate identifiers matching each of the attribute value pairs. Intersection of the matched identifiers contains the final result – identifier A1. Then the identifier-attribute-value hash retrieves the attribute being queried, which is ‘addend-1 = 3’. The complete retrieved chunk is represented under a special link, ‘retrieved’, in working memory. The second retrieval (2 + 4 = ?) failed to find a match, and a failure chunk is put under the ‘retrieved’ link.

It should be straight-forward to see that the complexity of retrieval is bounded by the number of matches for a single attribute-value times the total number of such conditions (each attribute-value pair in the cue is a condition and the entire cue is a conjunction of such conditions). This is psychologically plausible in that if there are more structures, in semantic memory, matching a particular description in the cue (a none-specific cue), then the interference effect arising from multiple hits can slow down retrieval. ACT-R architecture explicitly models the interference effect [5] and generates retrieval timing data to match real human data. Although matching human data is not the main goal of Soar, it does provide clues about the underlying constrains, and serve for justification purposes.

In general, there could be more complex retrievals involving multi-level structures. In such situations, the value of a particular attribute is an identifier which has more structures underneath it and therefore needs to be ‘expanded’ in order to access those structures. Expanding is also implemented via the cue link, where the cue is the identifier of the target chunk instead of its partial description. Expanding mechanism is analogous to retrieving via chunk-id in ACT-R. Expanding is in constant time (faster than normal retrieval) since it already has the unique identifier. Complex retrieval is not required in arithmetic task, but may be useful for other tasks.

- ***Use***

Use of retrieved result is completely task dependent. In the arithmetic task, retrieved result directly gives the answer to adding a pair of number.

4.1.3. Results and Analysis

Simple associative learning effect

The semantic learning mechanism learns to associate a question and it’s answer by recording the structure representing a fact such as {A1 ^digit1 5, A1 ^digit2 6, A1 ^sum 1, A1^carry-borrow 1}. In a later situation, if 5 and 6 are again presented as digit1 and digit2 respectively, this structure will be retrieved to avoid doing the counting again.

Transfer learning effect

As arithmetic is a small closed domain, it’s straightforward to see that there are a total of 100 possible addition facts. For simplicity and for the purpose of identifying the advantage of semantic learning compared to chunking, let’s assume the agent always distinguishes ‘2+3’ from ‘3+2’. These reversed pairs can be merged together and result in 55 instead of 100 facts, but this reduction of representation benefits from domain specific knowledge (digit1 and digit2 are interchangeable) and is not a distinctive property of semantic learning vs. procedural learning (chunking). After acquiring the chunk in semantic memory, the same set of addition facts can be used for the purpose of subtraction as well, for example 5+6=1 with carry 1, also indicates 1-6=5 with borrow 1 if there is knowledge about the general underlying relation of addition and subtraction (Figure 11).

For chunking, however, it must learn 2 separate rules, which have different condition and action sets, to encode the two facts: 5+6=1 with carry 1, and 1-6=5 with borrow 1 (Figure 11). Therefore, chunking requires leaning 200 rules for addition and subtraction. This comparison demonstrates the benefit of learning declarative representation, for which, facts and retrieval procedures are separated, so that only 2 accessing rules (one for retrieving addition result, one for retrieving subtraction result) plus 100 facts are required. For chunking, where rules are used as the representation, knowledge contents are embedded within the accessing procedures, so that 2×100 rules are required to represent the same amount of knowledge. In general, the same facts may have (exponentially) many ways to be accessed, so that the situation is $M + N$ vs. $M \times N$, where M is the number of potential ways to access some coherent factual knowledge and N is the number of such facts. As it has been shown in the previous associative learning task, sharing of same knowledge by different procedures is a common situation.

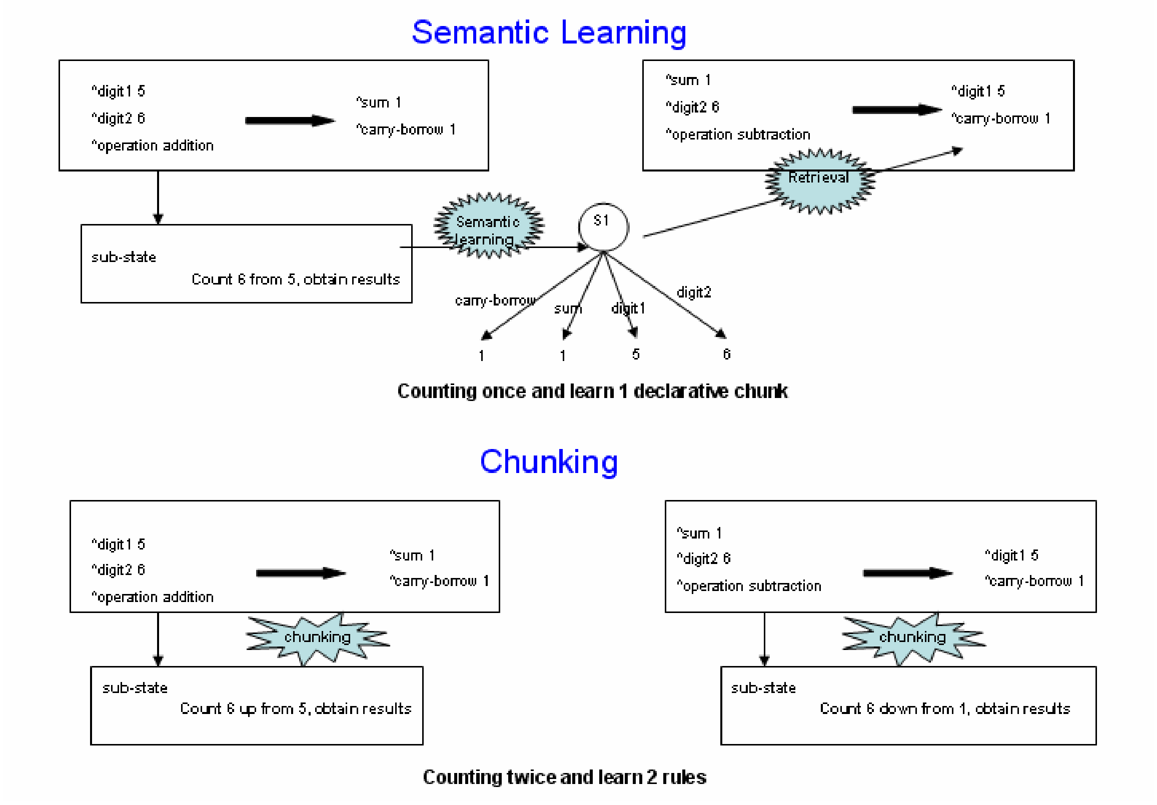


Figure 11: Comparison between chunking and semantic

Interaction among learning mechanisms

Chunking and semantic learning are complementary mechanisms rather than replacements for each other. Semantic memory retrieval is more expensive than firing a single rule because it requires separate steps to place cue and access the retrieved structure, plus the complexity for searching semantic memory varies a lot depending on specificity of the cue. Therefore, chunking over a semantic retrieval can provide further speed ups. Intuitively, chunking speeds up execution via practicing, while semantic learning acquires flexible knowledge structures that are potentially transferable to different procedures without prior practicing in the exact situation.

Figure 12 demonstrates the effects from the two different aspects of learning, and Table 3 shows the detailed breakdown of the decision cycles.

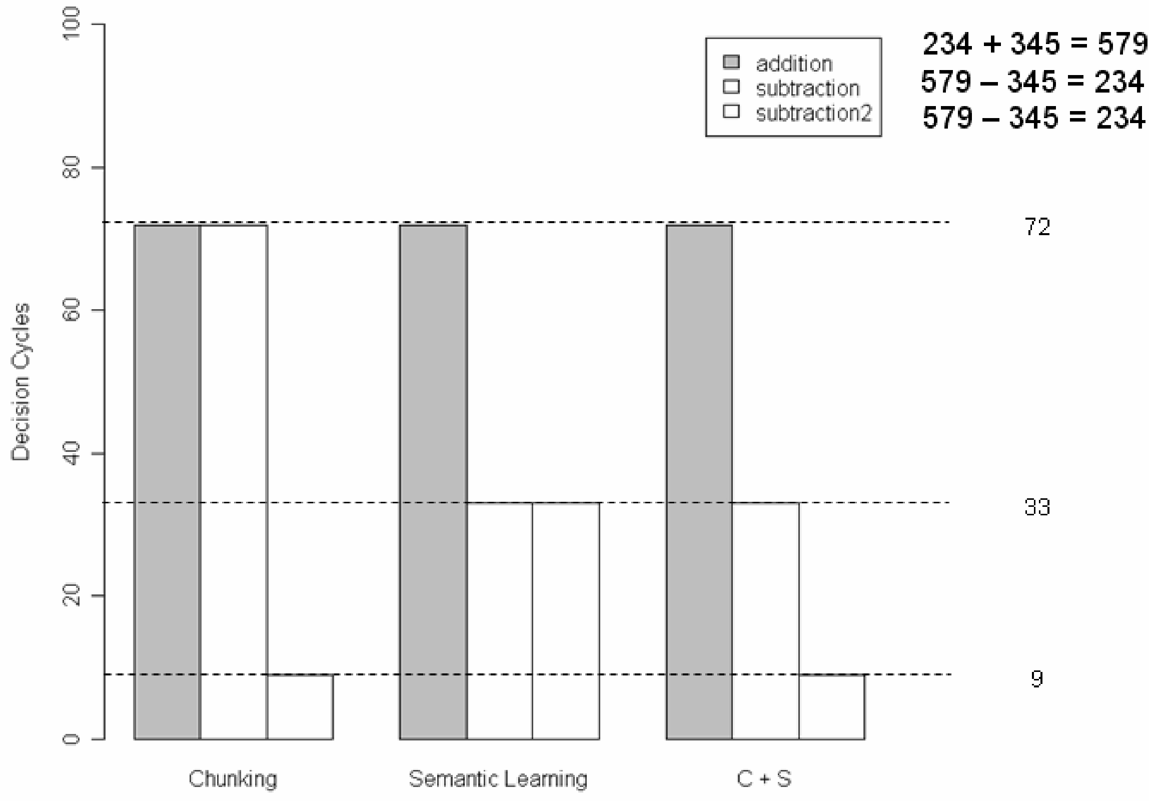


Figure 12: Semantic learning helps achieve transfer learning

Table 3: Comparison of decision cycles breakdown for different situations

Situations / Decision Cycles / Operators	Before learning	Semantic learning	Chunking
operators in top-state (initialization, process-column, next-column)	9	9	9
get-digits (from top-state)	3×3=9	9	0
write-result (to top-state)	3×1=3	3	0
retrieve	3×4=12	12	0
counting	39	0	0
Total	72	33	9

The addition problem in this test is: 234 + 345 = 579. The subtraction problem is: 579 - 345 = 234. They are reverse problems and completely depend on the same set of arithmetic facts. In this experiment, the agent sequentially did the addition problem once, and then did the subtraction problem twice. Chunking speeds up the problem solving by practicing with the same problem, but cannot transfer between different but related situations. Semantic learning is able to transfer knowledge from addition to subtraction. Having chunking and semantic learning work together (C + S), the performance will benefit from both aspects of learning.

Table 4 summarizes the effects of the above different configurations on learning all addition and subtraction problems. Chunking needs to go through counting procedures 200 times for each of the combination of digits for both addition and subtraction. Semantic learning avoids counting in subtraction again if it has already learned the reverse problem in addition, and *vice versa*. On the other hand, firing a rule, which happens just within a single decision cycle, is a more efficient process than performing a semantic memory retrieval, which needs to use rules to help with the retrieval as well as perform a more expensive matching and require extra decision cycles (refer to Table 3 for details).

Table 4: Performance comparison among different configurations of chunking and semantic learning

	# of counting procedures (of transfer learning effects)	**# extra decision cycles to use the knowledge**
Chunking only	200	None. Automatically apply the learned rule.
Semantic Learning only	100	Extra decision cycles related to retrieval
Chunking and Semantic learning	100	None

4.1.4. Conclusion

First, this simple task demonstrates the benefit of declarative learning. Although the arithmetic problems are trivial for a computer, it is representative of more complex computations in general. The phenomenon that different procedures may share the same knowledge is a common situation and it is functionally important. For example, in the previous associative learning task, the system is able to response to both the questions ‘Which city is USA capital?’ and ‘What is special about Washington DC?’, as long as the fact that ‘Washington DC is USA capital’ has been learned in one of the contexts. On the other hand, with chunking alone, the system will be stuck on both questions once. Second, the integration of multiple learning mechanisms has also been demonstrated. Since the two mechanisms cover complementary spaces of learning, it is able to gain benefit from both when they work together. The overall learning demonstrated in this task is from general primitive procedural knowledge (counting) to specific transferable declarative knowledge (remembering the facts) then to more specific untransferable procedural knowledge. And the execution is first from slow computation to faster direct knowledge retrieval and then to even faster rule firing. Experiments with this simple task have contributed to detailed understanding of complementary learning mechanisms from computational and functional views.

4.2. The Eater’s domain (interactive noisy environment)

The cognitive arithmetic domain is a purely internal symbolic task, for which there is always unique correct answer from semantic memory. This is typically the case for a task where the domain theory is complete and it applies to many traditional reasoning tasks. However, in many real world tasks, the domain theories are incomplete or not guaranteed to be correct, and the agent needs to learn gradually from experience of interaction with external environment using general heuristics. In addition, the input from the environment may contain various sources of noise. If such inputs are stored directly into semantic memory, the answer from a semantic query might not be unique, but drawn from a distribution due to intrinsic noise or hidden noise. In such situation, the agent must perform bottom-up empirical learning, instead of purely top-down explicit knowledge driven learning.

Classical symbolic production systems (e.g. Soar) are good at top-down knowledge based reasoning and explanation based learning, where domain knowledge is pre-programmed, and deductive learning helps derive the most useful entailments of the initial knowledge base via executions. In this task, we set up a scenario demanding experience-driven learning and empirically compared solutions using a general semantic learning mechanism and discuss general issues that arise in this and similar problems.

In order to demonstrate the advantage of explicit semantic learning, hierarchically organized knowledge will be used in this task. Hierarchical structure appears to be a ubiquitous property of our world and encoding input hierarchically is a functionally efficient way to organize explicit semantic knowledge, such as in the hierarchy of abstract concepts like animal -> bird -> canary. Hierarchical relation is useful for strategies like systematic generalization, where unobservable properties for novel instances can be predicted through category recognition systematically at different abstraction levels in the hierarchy. For example, a person who has never seen a canary but with sufficient related experience could easily recognize it as a bird, and predict that it must lay eggs in a nest without actual observation. He could also go up to the ‘animal’ level to make more general predictions such as it can move and must eat things and drink water to keep alive, *etc* (How useful are these predictions depends on the task being performed). Although semantic memory is not architecturally committed to hierarchical structure, it should be able to support such structure and exploit the associated functionality. In this task, simple generalization strategies, utilizing hierarchically structured input from the environment, are implemented and compared.

4.2.1. Task description

The Eater domain involves a Pacman-like agent (eater) eating food in a grid world (Figure 13). The goal for the eater is to get high score by eating the most “nutritious” food it can find. There are different types of food randomly distributed in the world. Each food has a set of visible features and an invisible “nutritional” value associated with it. The agent must learn the association between the value and features by actually eating the food. However, the value of food could be either positive (edible food) or negative (poisonous food). So while learning the food value by experience, the eater also needs to avoid poisonous food as much as possible (that’s where generalization strategies may be useful).

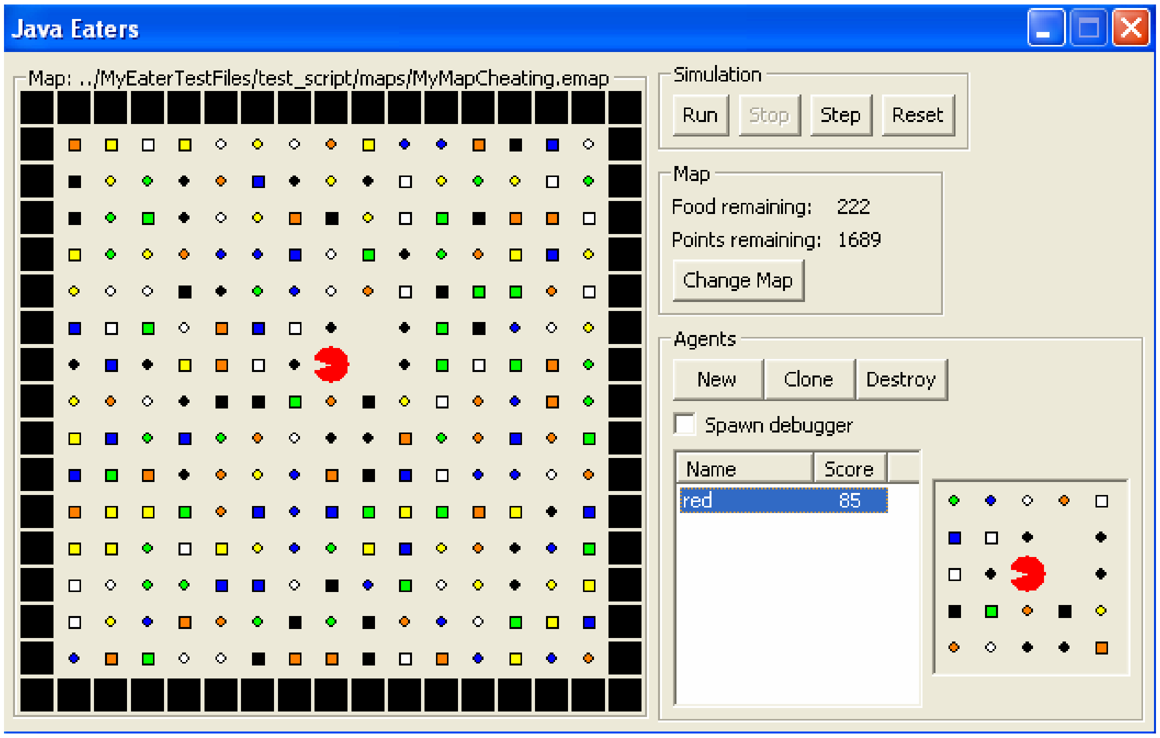


Figure 13. Snapshot from the eater's environment.
The different colors and shapes represent different types of food.

In order to mimic the primitive sensory input from the perceptual system to central cognition and address the empirical learning problem, the input features will be noisy. On the other hand, the environment is not completely arbitrary, but has significant structure. Semantic memory should be able to extract and take advantage of the underlying structure, while also able handle the accompanying noise which would cause serious problems for purely symbolic learning (The noise from the input may cause the system fail to recognize a pattern and other sources of noise during learning may cause the system to erroneously learn special cases and fail to extract consistent information). Our general design strategy to deal with noisy environments is to separate the noisy syntactic learning (learning of input) from semantic learning (learning of task knowledge) by introducing a statistical learning component, which performs unsupervised learning and transforms the noisy input into more compact discrete symbolic features. There could be noise involved in both stages, and they will be treated separately (details in later sections). The overall structure is shown in Figure 14.

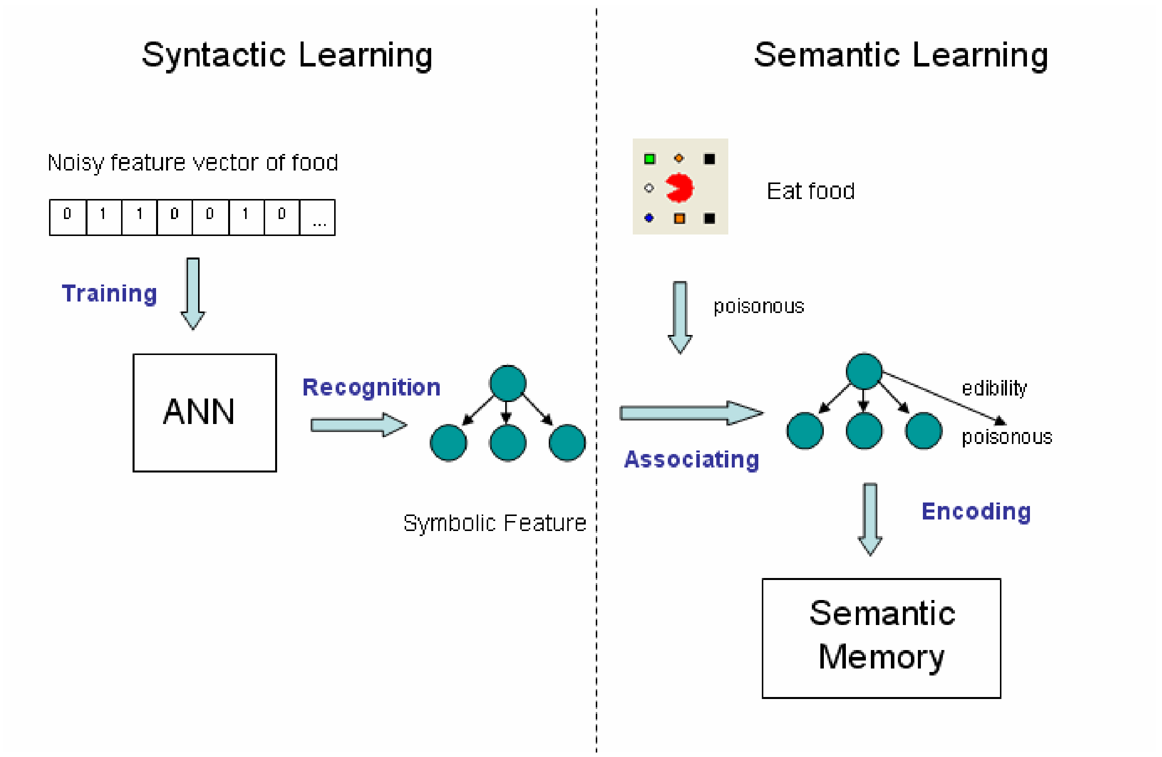


Figure 14: Dealing with noisy input

In Figure 14, ANN refers to an artificial neural network. In general the ANN box could be any component that serves the purpose, with ANN being a very representative class. In our current implementation, we use a particular neural network based learning algorithm which performs unsupervised hierarchical clustering [21, 22] on the input. The benefit of the hierarchical clustering algorithm is that the original inputs are reduced to lower dimensional symbolic representations with the hierarchical structure being preserved. As semantic learning is based on saving and retrieving instances, saving original instances without clustering not only wastes storage space, but also hurt retrieval performances: exact match based on noisy input will not find matches while partial match in high dimensional space is computational expensive.

4.2.2. Syntactic Learning

Construction of noisy input

In order to make the task both challenging and interesting enough for demonstration purposes, the considerations for constructing the input are: 1. each input feature of food is represented by a feature vector; 2. the input contains noise; 3. the input contains significant structures that can be exploited by semantic learning. Based on the above considerations, we decided first to construct features for food prototypes with hierarchically structured relations without noise, and then generate noisy instances from the underlying prototype with certain degrees of noise.

Prototypes

We arbitrarily created 12 underlying prototypes hierarchically related at three levels. The first level of the hierarchy has 2 super-groups, each of which contains 3 sub-groups and each of the 6 sub-group at level2 contains 2 prototypes. At each level, instances are grouped/discriminated by mutually orthogonal feature sets. Figure 15 illustrates the prototypes both in bit representation and by Dendrogram.

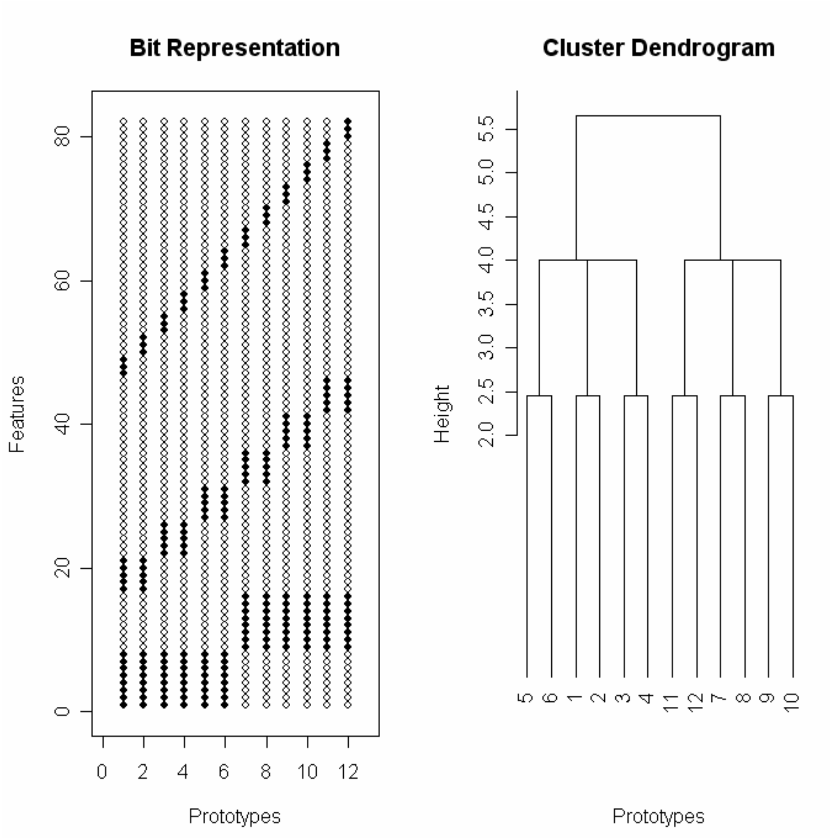
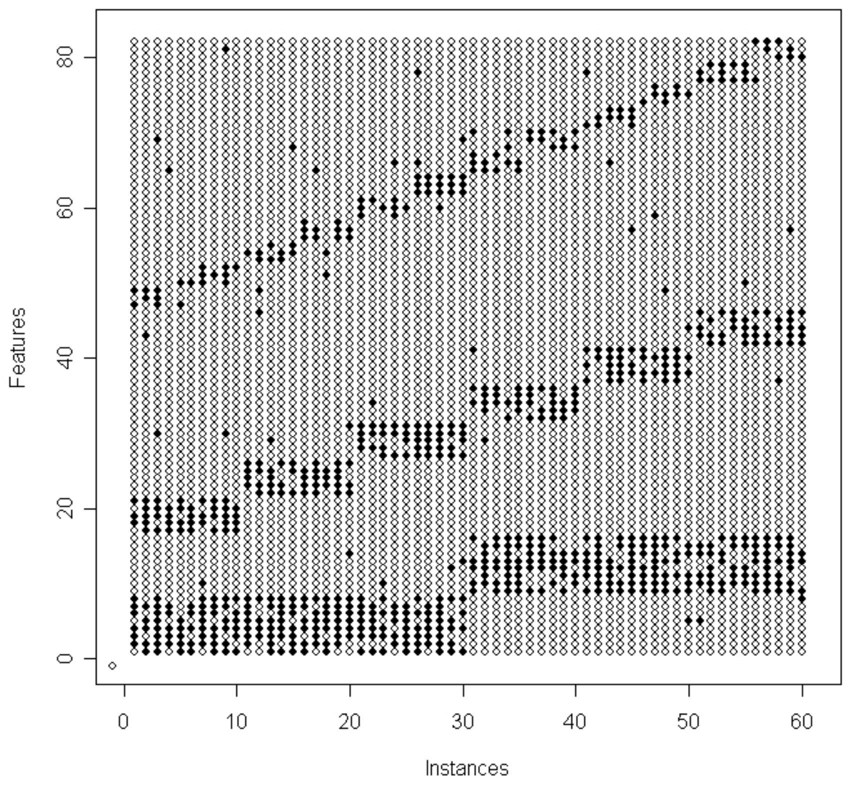


Figure 15: Prototypes

On the left of Figure 15 is the bit representation of the 12 food prototypes. Each column corresponds to one prototype. Each bit represents one dimension of the feature vector. A black dot means that the feature has a value of 1 (presence of the corresponding feature) and empty dot means a value of zero (absence of the corresponding feature). The 12 prototypes are partitioned into 2 super-groups at the first level, where memberships are determined by features 1 to 16, with 8 features representing each group. Differences among groups at lower levels of the hierarchy are more subtle, represented by fewer numbers of features (5 features for each group in level two, and 3 features inlevel three). It is purely for simplicity consideration that the feature set separating different groups are completely orthogonal. In addition, the feature vector is binary, although the algorithm works for continuous valued vectors as well. On the right of Figure 15 is the hierarchical clustering dendrogram of the 12 prototypes. The hierarchical clustering is performed in R (http://www.r-project.org/) using the default parameters: Euclidean distances and complete linkage method.

Noisy Instances

Based on the 12 underlying prototypes, instances are generated from each of them with a certain amount of noise. There are 2 parameters in the simple noise model being used here. *Alpha* is the probability that the prototype feature is 1 (true positive), and *beta* is the probability that the non-prototype feature is 1 (false positive). It is also for simplicity considerations that the noise for each dimension are independent.



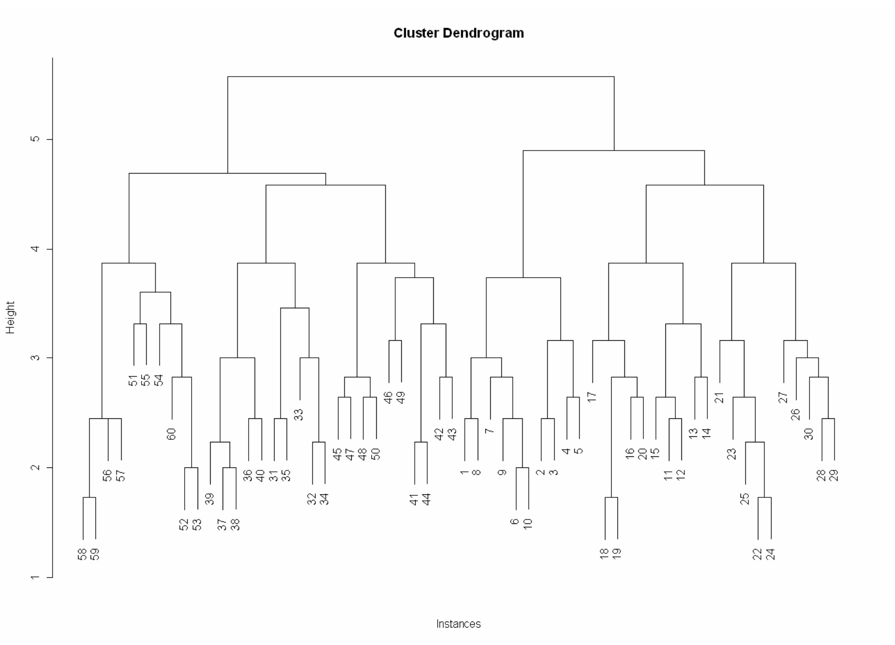


Figure 16: Noisy instances from prototypes

In Figure 16, the top figure shows 60 instances of food, 5 for each prototype, with noise alpha=0.7 beta = 0.01. The bottom figure shows hierarchical clustering of the same 60 food instances. Given the presented noise, not all instances are correctly clustered by the standard algorithm. The hierarchical clustering is performed in R (http://www.r-project.org/) using the default parameters: Euclidean distances and complete linkage method.

Hierarchical Clustering algorithm

The task of the hierarchical clustering algorithm is to filter out noise and extract the underlying hierarchical structure, which will be useful for symbolic semantic learning.

A hierarchical clustering algorithm using a winner-take-all neural network [21, 22] was integrated into Soar to perform learning (preprocessing) on the input and reduce the noise. This algorithm, like most neural network algorithms, is an online learning algorithm. The capability to learn incrementally from continued experience is the advantage that will be appreciated in such real-time interactive domains, compared to traditional offline hierarchical clustering algorithms, such as the one used in R to generate the plot.

The algorithm

The hierarchical clustering algorithm is based on a neural network model of cerebral cortical circuits. The network consists of competitive learning units (neurons), which are governed by the winners-take-all competitive learning principle. For simplicity, the detailed network structure and brain circuit mapping is omitted here. The following is the pseudo code for the algorithm.

```
for input X
    for Wi in win(X, W)
        Wi <- Wi + r(X - Wi)
    end for
    X <- X - mean(win(X, W))
end for
```

[where X = input vector; Wi = weight vector for unit i; W = weight matrix for all units; r = learning rate; win(X, W) = most responsive unit for X in W, in out implementation it's based on dot product]

The general algorithm selects and updates multiple winners, and subtracts the mean of winners to generate new input. In out tests, however, only a single winner is picked for simplicity.

The simplified network with relevant structure is shown below in Figure 17.

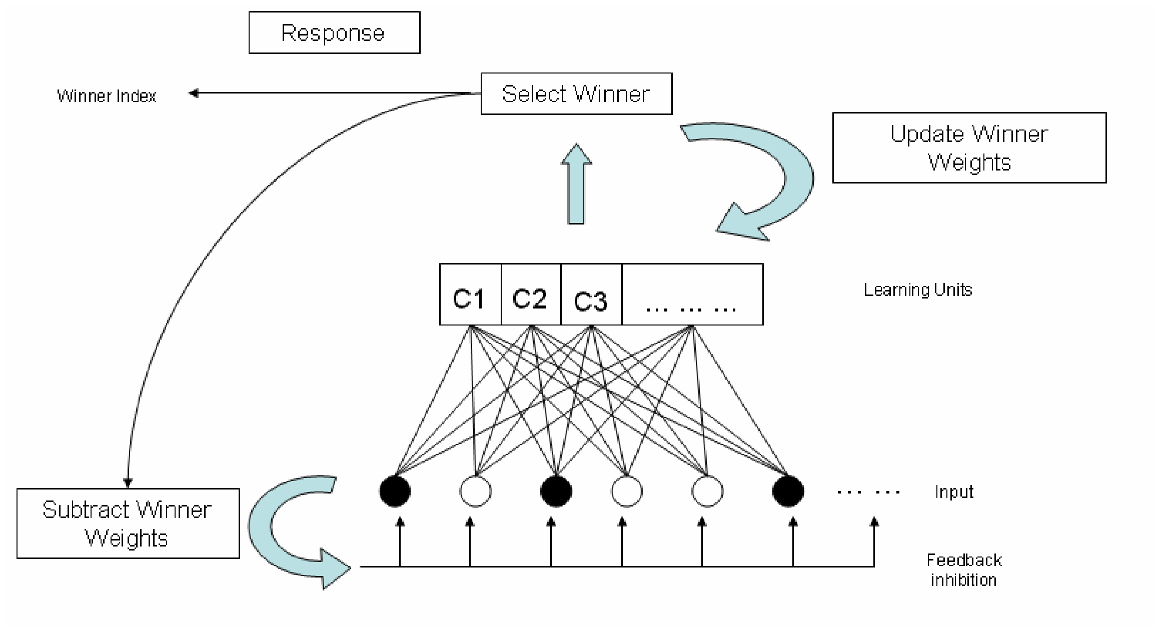


Figure 17: Simplified structure of the hierarchical clustering network

As depicted in Figure 17 the network consists of a set of learning units. Each learning unit in the network has a weight vector (representing a point in the high dimensional inputspace) represented by the lines from a unit to input layer (inputs are binary values as represented by the black and white dots). The winner is determined by taking the dot product between the weight vector and the input. The unit with the highest dot product wins and is updated according to the Hebbian learning rule shown in the pseudo code, which moves the weight vector closer to the particular training input. After updating, the original input is subtracted by the winner's weight vector (feedback inhibition), and the remaining signal is used as the new input to feed into the network. In testing mode, the network will generate a sequence of winners as the response to a particular input to designate the recognized category for that input.

This network has been shown to be able to generate cell-firing responses (sequence of winner units) that group learned inputs by similarity. For our purpose here, we need to test, empirically, how well the algorithm can learn the hierarchical structure of the 12 food prototypes with the existence of noise among the inputs.

Analysis and results

Here we evaluated the results of this hierarchical clustering algorithm on the inputs constructed by the protocol in the previous section. For each trial, the entire sample contains 50 instances from each prototype, therefore totally 600 instances. Training is on one half of the instances, and testing is on the other half of the instances. The results are averaged from 10 independent trials, and the 600 instances are randomly generated for each trial. The feature sets for each prototype from level one through level three contain 16, 10 and 6 dimensions respectively. The total dimension of the input vector is therefore 164 ($16\times2 + 10\times6 + 6\times12 = 164$).

Several factors, both from the input and from the algorithm parameter will affect the result of clustering.

1) Input

   a. Noise degree: The less the degree of individual noise and the bigger the dimension of feature sets, the less the overall noise degree will be. Also, the deeper down the level, the larger the noise propagated to that level.
   b. Orthogonality among feature sets: In the test, the prototype feature sets are perfectly orthogonal. Non-orthogonality will increase the difficulty of learning the underlying structure.

2) **Algorithm**
   a. Number of network units. If there are more units, the network will potentially be less likely to give the same response for related inputs. On the other hand, if there are fewer units, the network will more likely give the same responses even for unrelated inputs. In the test, 100 units are used, which is big enough to capture the underlying structure. The number of units also directly determines the network capacity (refer to the publications[21, 22] for detailed analysis).
   b. Learning rate, a decreasing learning rate is used according to the original publication [22]. The learning rate used here is $r = 0.2\times (count+1)^{-0.5}$, where *count* is number of times a particular unit has been the winner.
   c. Winner selection algorithm: Dot product is used. Alternatively it could be other proper distance metrics such as the Euclidian distance.
   d. Initialization of the network: A badly started network will likely result in bad performance. Here, the network is initialized based on a random hypersphere with radius of 1.

The analysis here is not intended to give a comprehensive understanding of the algorithm (only the degree of noise dimension is manipulated, and the other dimensions are left out without being tested), but mainly to evaluate whether it meets the requirements, at least, for our simple task environment.

The inputs are trained through the network with 3 cycles intending to learn each of the level of the hierarchy. In the testing phase, the network also gives 3 levels of responses sequentially for each input to designate the grouping of that particular instance. The results are evaluated based on how well the responses identify the correct hierarchical clustering structure. *Within-group-overlap* and *between-grou- overlap* are used to evaluate the quality of clustering. *Response-changes* is used to evaluate the convergence behavior.

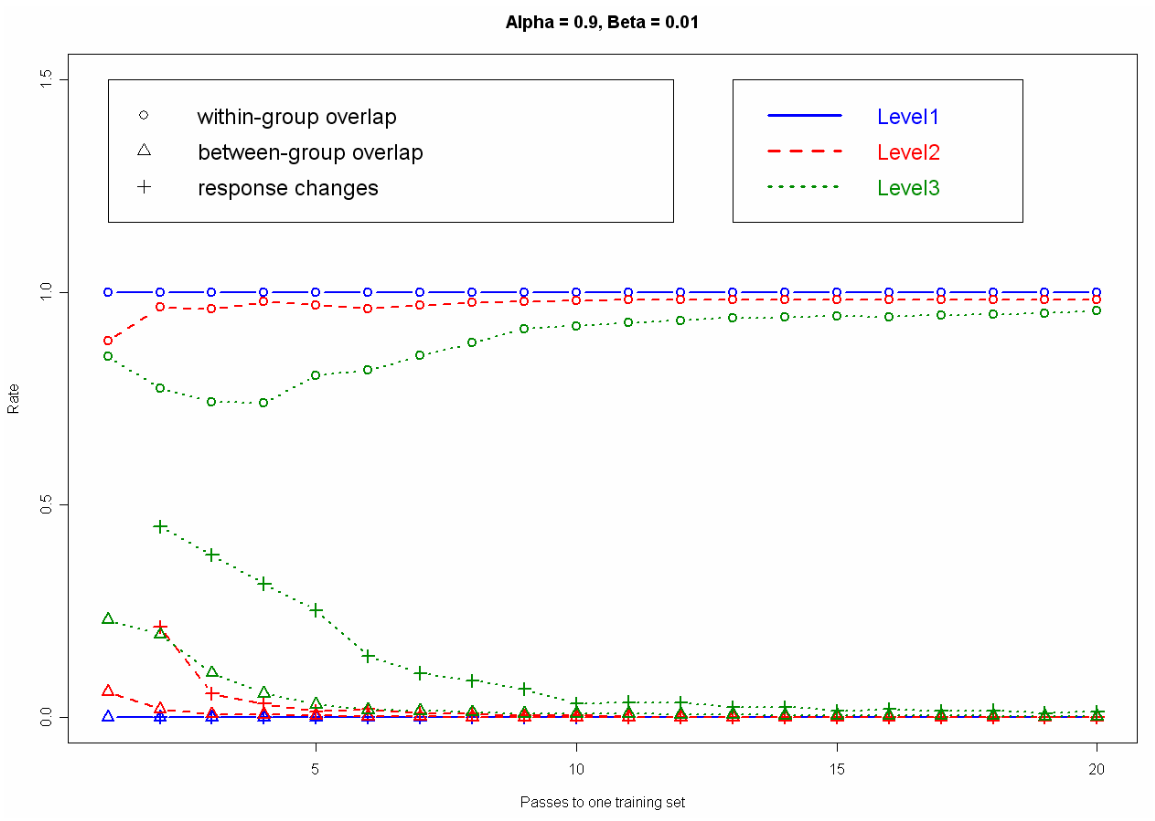
1) **Within-group-overlap** is measured by the percentage of instances having the same responses within the same group at a given level. Only the majority (or most common) response is used. For example, if there are 10 instances belong to group1, in which 6 of them have response A, 3 of them have response B, and the other one has response C, then the within-group response would be 0.6 (for the majority response A). This metric is an underestimate of the within-group overlap.

2) **Between-group-overlap** is measured by the percentage of pairs of instances from different groups having the same response.

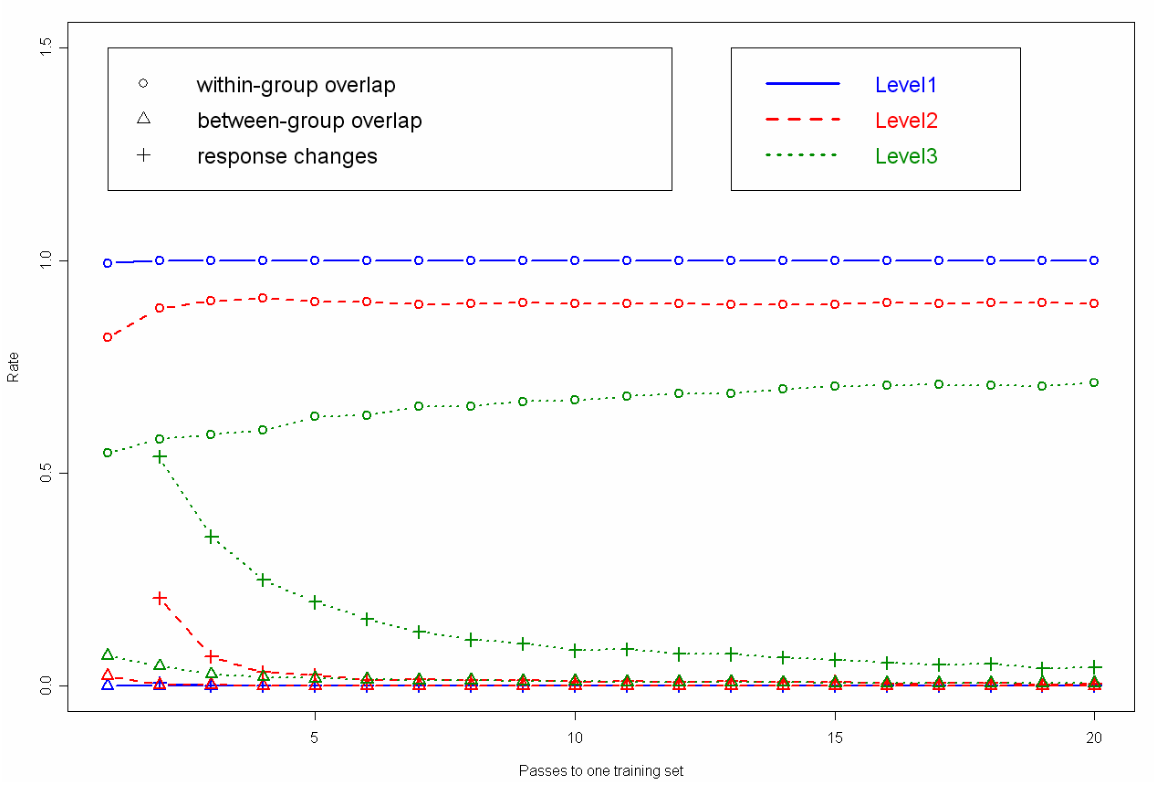
3) **Response-changes** is measured by percentage of changed of responses for the same instance between consecutive training passes (each training pass contains 1 instance from

each of the 12 prototypes). Ideally, the response should converge after all instances have been learned (at least for such a straight forward structure as used in this experiment, the weight vector of the network unit that consistently responds to a particular group of inputs is expected to converge to the mean of all instances belongs to the group). The purpose of response-changes is to evaluate the stability and convergence of learning.

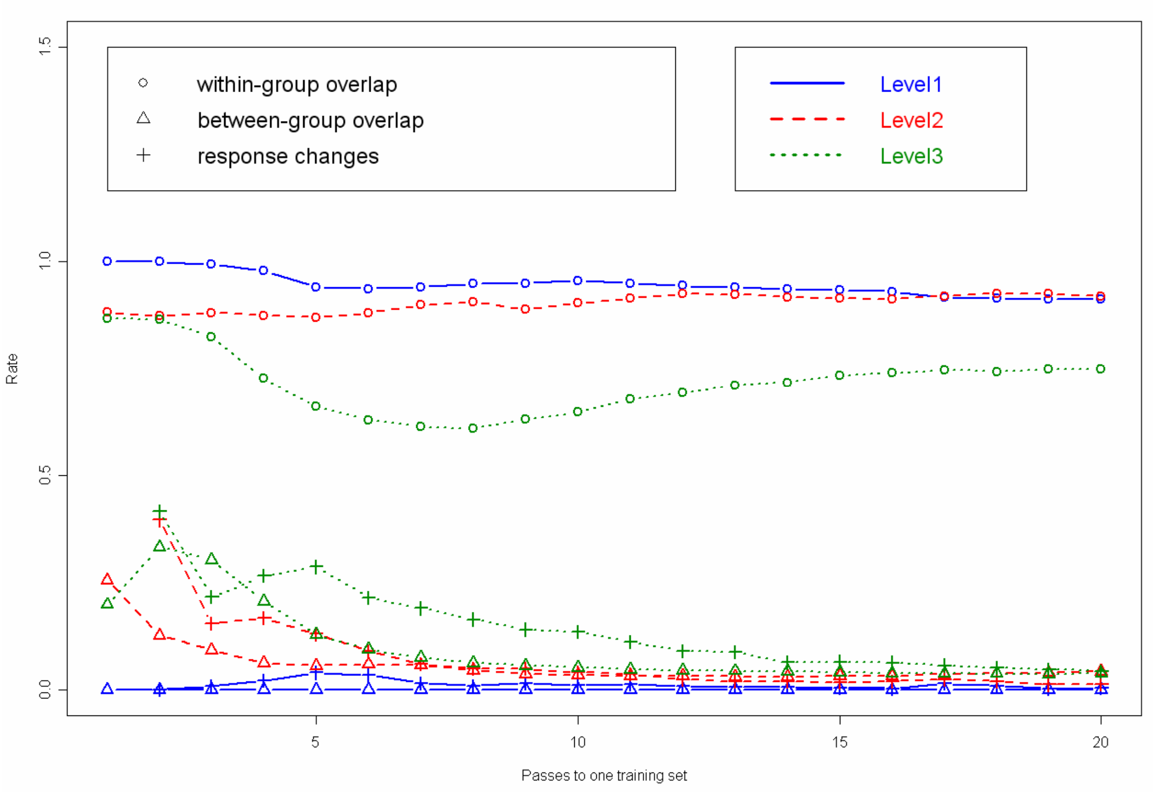
Ideally, the desired performance should have high within-group overlap, low between-group overlap and fast convergence. The following plots in Figure 18 illustrate the performance of clustering algorithm with different degrees of noise. The purpose is to evaluate the algorithm's robustness against noise.



Alpha = 0.7, Beta = 0.01



Alpha = 0.9, Beta = 0.05



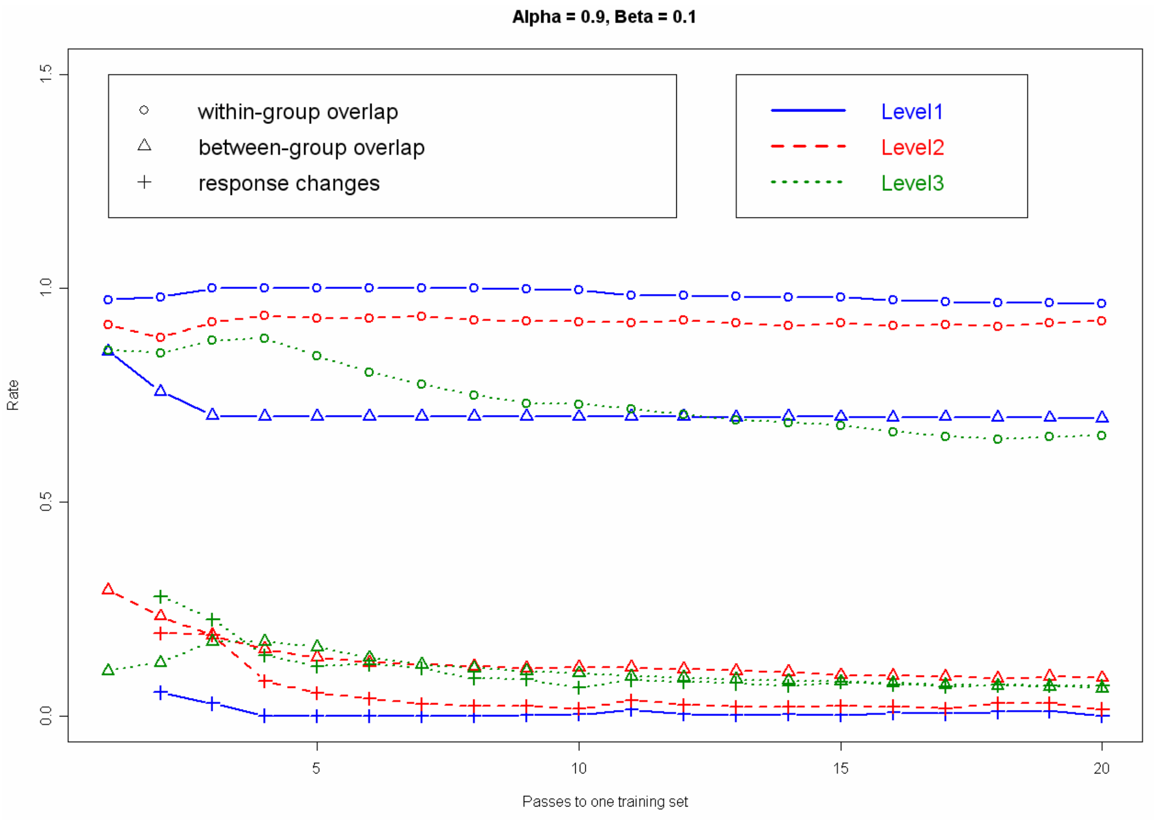


Figure 18: Performance of hierarchical clustering with different degrees of noise

Figure 18 demonstrates results for different degrees of noise: (alpha=0.9, beta=0.01), (alpha=0.7, beta=0.01), (alpha=0.9, beta=0.5) and (alpha=0.9, beta=0.1). The results suggest that ‘level1’ stabilizes very quickly, followed by ‘level2’, and ‘level3’. Eventually the within-group-overlap of the responses is high, and between- group-overlap is close to zero. Therefore, the different types could be both related and discriminated at different levels based on the symbolic outputs. For noise (alpha=0.9, beta=0.1) (the 4th one), between-group-overlap of level 1 is high at convergence, which means it fails to make the correct discrimination at level 1 (failed to recognize the exactly underlying structure). The empirical results also suggest that the algorithm is pretty robust against false negative noise (low alpha value), but more sensitive to false positive (high beta value). Such considerations need to be taken when connecting Soar with real input devices.

One important effect is that noise in input leads to significant growth in number of unique instances, even the underlying prototypes are from a small set. The consequence is that as the agent experiencing more and more instances of food, each of them will just appears novel to the system as a unique instance even though many of them are from exactly the same prototype. This effect will have impact on subsequent semantic learning, since it learns by recording each instance for future predictions. With this clustering algorithm, noise is reduced and instances are collapsed into groups close to their original categories with compact representation. The following plot (Figure 19) compares the number of resulted unique instances with clustering and without clustering at different degrees of noise.

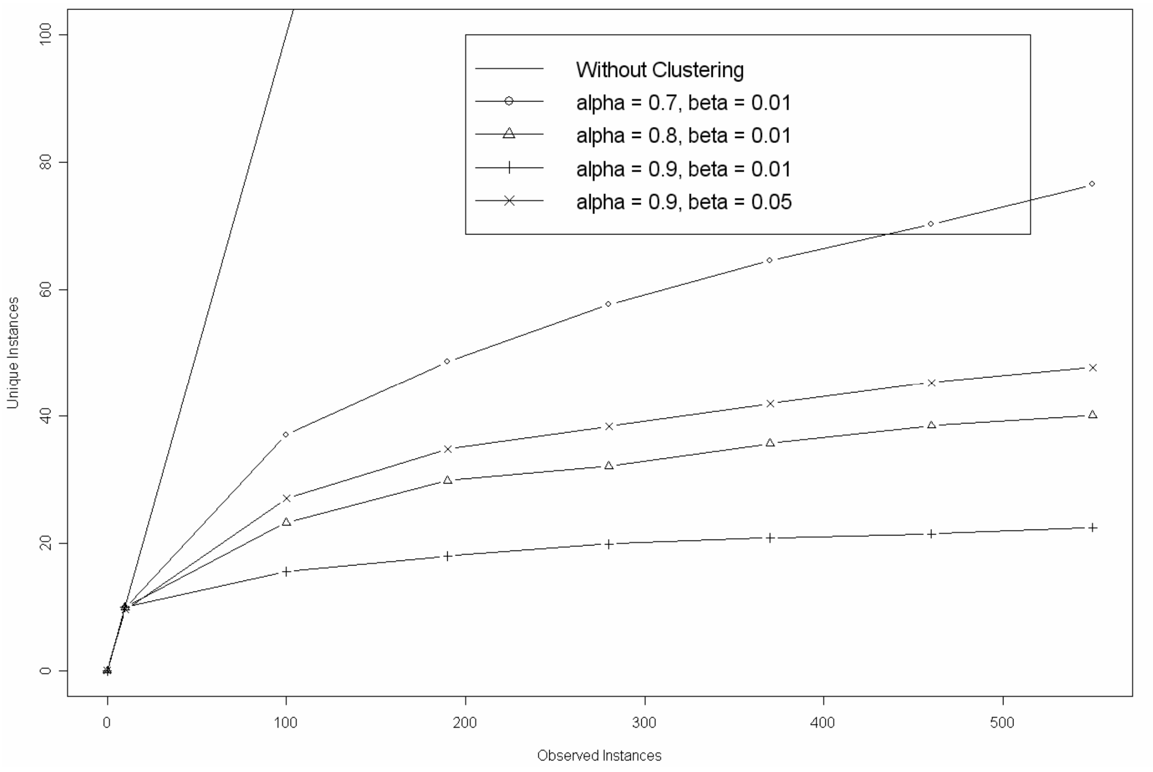


Figure 19: Resulted unique instances under different situations

In Figure 19, the straight line is for original inputs without clustering, showing the number of unique instances increases linearly. One important characteristic is that beta has much bigger effect on the growth than alpha. As learning proceeds, more and more instances will be observed. Due to the noisy nature of environment, few of them will be exactly the same even they are from the same type. If semantic learning needs to store away the features of the observations, the storage will increase linearly (bounded by $2^{\text{dimension}}$ in this case), although only 12 types exist. Consequently, the complexity for matching will also increase linearly if a closest match is performed on the original instances. On the other hand, the symbolic clustering output contains only the necessary information with compact representation, so that efficient hashing techniques can be used for exact matching at different levels for different degrees of specificity.

In summary, this clustering algorithm can achieve hierarchically fine-grained discrimination and grouping in the input feature space. It is expected to work well on inputs with significant underlying structures, such as the inputs in current experiments. In most other domains, however, the inputs are not likely to have such significant structure, and the algorithm will not be able to learn such structures consistently. But, the general hierarchical categorization capability and noise resistance property is expected to be useful in all regulated environments. Nevertheless, the study of this algorithm in Soar is still very preliminary.

4.2.3. Semantic Learning

Semantic learning is the most critical part in this task and it is where it learns feature-value association of the foods via experiences. The value of the food is viewed as the 'semantics' of this task, in the sense that the symbolic response of food feature does not have any meaning until it is associated with the value of food, which is directly used to make decisions.

This task intends to make a concrete example to show how semantic learning can handle structured information, such as hierarchical relations. Hierarchical relations provide useful information for generalization strategies described previously. In this experiment, the task is set up to demonstrate one aspect of the generalization strategy, the tradeoff between making generalization and doing exploration. The purpose of the task is mainly to give a working example on how to use semantic memory to deal with hierarchical structure.

Given the above considerations, nutritional values are assigned to each prototype of food, and the values of foods are designed to be consistent with their features at level 2 in the hierarchy (Table 5). Although any other arbitrary assignment is possible, this particular assignment was designed to demonstrate useful properties related to hierarchical structure as mentioned above and will be shown below.

The following table shows the assignment of the food values to each of the prototypes used in this experiment. The eater agent only knows the value of food after eating one, and then it can associate the value with observed features via semantic learning mechanism.

Table 5: Assignment of food values

Level1	**C11**						**C12**					
Level2	**C21**		**C22**		**C23**		**C24**		**C25**		**C26**	
Level3	**C31**	**C32**	**C33**	**C34**	**C35**	**C36**	**C37**	**C38**	**C39**	**C310**	**C311**	**C312**
Value	-5	-10	6	15	-15	-20	55	60	-50	-45	-70	-80

In Table 5, the response from clustering is represented by symbols such as C11, which means it's at level 1, and it is cluster 1; and C12, which means level1 cluster 2, *etc*. They are just symbols, as internal representation of the category, without semantic meanings. In semantic learning, eater will learn the task related designations of those symbols.

In our solution using a general semantic memory mechanism, what has been saved in semantic memory are declarative chunks, each associating observed features (after clustering) with the value of food (value is obtained after eating the food, and the numerical value of food is converted into discrete symbols including *edible* and *poisonous* when doing semantic learning). Figure 20 demonstrates the details of semantic learning. The retrieval algorithm is extended to summarize quantitative statistical information from multiple matches to provide the necessary sub-symbolic information about retrieval.

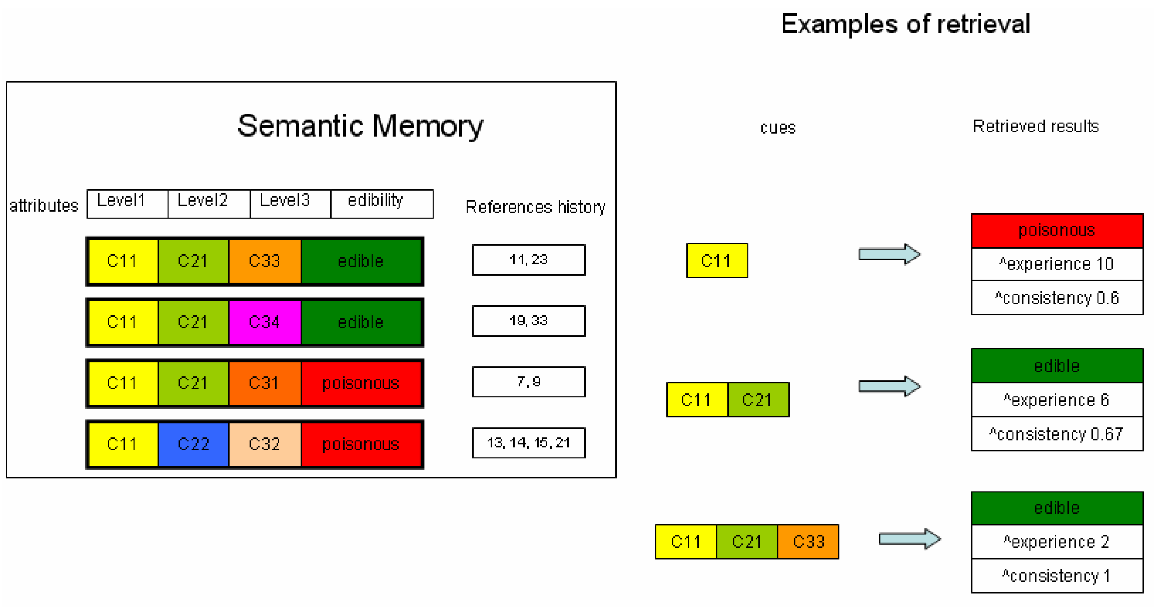


Figure 20: Semantic memory with statistical meta-information

On the left of Figure 20, it shows the contents stored in semantic memory. ‘Reference history’ records the index of decision cycles where the same chunk is saved into semantic memory. On the right, it shows examples of cues and corresponding retrieved results based on the current contents in semantic memory. It also illustrates how the *experience* and *consistency* meta-information are calculated. When there are multiple matches, experience is the count of matched instances. Consistency is the percentage of retrieved value among all possible values for the same attribute. Consistency and experience together can provide some ‘confidence’ measure of a semantic retrieval. Such information is called meta-information since they are about global properties of the entire semantic knowledge (they are knowledge about knowledge). They will play important functional roles in reasoning and decision making especially in noisy environment. For example, if the eater predicts that a food is likely to be poisonous, but not confident enough due to inconsistent and/or limited experience, it should try more times before making a permanent conclusion about the food. On the other hand, if it is confident about the prediction, then it should never try a predicted poisonous food and always eat a predicted edible food. The functional roles of meta-information will be shown in the next section.

4.2.4. Results and Analysis

Noise in input only

We programmed eaters with different strategies using the above architectural mechanisms, and compared the results. Different generalization strategies and baselines are described by

Table 6. The performances are compared in Figure 21.

Table 6: Strategies and baselines being compared

Strategies	Descriptions
Level1-learning	If there are matches from semantic memory under the same level1 category, eater will use that value as a prediction. If the prediction is negative, it does not eat the food. If there is no match (it has not eaten any food under this category), treat it as unknown (value = 0), and it will eat it if there are no other known edible foods around. The eater will prefer more specific matches, for example, if there are matches at level3, it will use that result, otherwise it will try level2, then level1. Intuitively, this strategy is to predict the value of food by making generalization up to level1.
Level2-learning	Similarly, eater predicts the value of food by making generalization up to level2. Everything else is the same.
Level3-learning	Eater makes no generalization in the hierarchy. Only exact match at level3 will be used towards prediction. Everything else is the same.
Baseline1: Optimal-no-learning	This is the optimal baseline. Eater knows the value of food, so it can always make the best choice.
Baseline2: Random-no-learning	This is the bottom baseline. Eater makes moves and chooses between eat or don't-eat randomly. There are 8 possible actions: action = {move-east, move-south, move-west, move-north }× {eat, don't-eat}

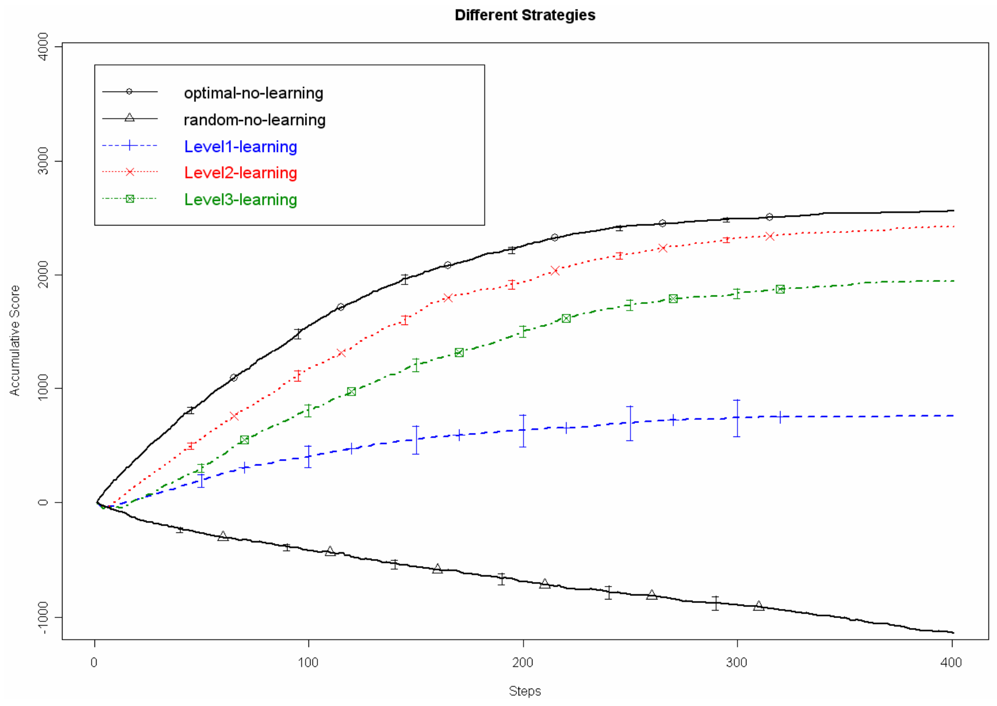


Figure 21: Performance with different strategies

The result is averaged from 40 random trials for each strategy. Error bars show one standard deviation above and below the mean. For each trial, the eater agent is first trained on input only with 50 random noisy instances (alpha=0.9, beta=0.01) to hierarchically categorize food without eating. For different learning strategies, the initial performance was worse than optimal since the agent does not have any knowledge about the food when it starts. After experiencing with a few samples, the agent is able to find edible food to eat and avoid poisonous food, as can be seen from the much better performance than random-baseline. Level2-learning is optimal since it's at the right level for generalization. Although level3 should also learn to eat all edible food it will cause the agent to eat more poisonous food than level2 because it does not make any generalization to avoid poisonous food (it will eat all poisonous food once, while level2 strategy just eat one of the poisonous food under the same level2 category). Level1 is too conservative with the effect being that once the eater encounters a poisonous food, all unexperienced edible foods without a closer prior experience will be avoid because it will assume it is poisonous by making wrong generalization (over-generalization at level1). The accumulated score eventually stops increasing because the predicted edible foods are all consumed. The random-baseline agent will randomly eat food and get a negative total score because there are more poisonous foods in the environment.

Noise in the value of food

When there is noise in the value of a food, for example, the edibility of the same food type may have both positive and negative values, the agent needs to learn the long-term expected value instead of any single experience. In this case, sub-symbolic aspects about semantic retrieval need to be exposed to the knowledge level for decision making. To test the performance of the system with this type of noise, the probability of getting opposite edibility (encountering a negative valued instance for a type of food with expected positive edibility and *vice versa*) is made relatively high using the following model: for a type of food with expected value of x, the actual distribution of values will consist of 1/3 –x and 2/3 2x. For example, for an edible food of expected value 10, 1/3 of them will be -10 and 2/3 will be 20. This is not a natural distribution, but arbitrarily created for better demonstration purposes. Random instances are generated with this extra noise based on the same underlying prototypes (Table 5), and different strategies are compared.

In addition to the previous strategies, two new strategies (Table 7) were added which make use of the meta-information returned with a semantic retrieval. In current implementation, the meta-information consists of first-order statistics about the retrieved value for a target attribute, including *consistency* and *experience,* as described before. A target attribute is the intended attribute whose value is being queried, which can be specified in the retrieval cue. Here the target attribute is 'edibility'. These simple statistics can help achieve more robust statistical learning than naïve one-shot semantic learning as in previous experiments. The purpose of the two new strategies is to give concrete examples to show this point.

Table 7: New strategies to handle noise in food values

New Strategies	Descriptions
level2-learning-meta1	Experience >1, consistency>0.6 Which means, in order for the agent to trust the retrieved value, matches of the cue must have at least 2 prior experiences, and over 60% of them should have the same value. For example, if the first 2 experiences include one edible and one poisonous food, then the agent will treat a food as unknown until a third one of the same category has been eaten. At any point after the second experience, if the food is predicted to be poisonous (over 60% are poisonous), then the agent will never try the food, as it is permanently assigned as poisonous and the eater just stops further exploration with it (this is a simple strategy). The eater agent makes generalization up to the optimal level 2 like in the level2-learning strategy. It will still first try to find out the result at level3, if level3 does not give sufficient information (not enough experience or the majority is not above 60%), then it'll generalize at level2, but will not generalize to level1
level2-learning-meta2	Experience >3 and consistency >0.6. An agent with this strategy is more exploratory, as it requires more experience before stopping eating a poisonous food with higher probability to discover all edible foods, but also likely to get more penalty from eating poisonous foods. Everything else is the same as above.

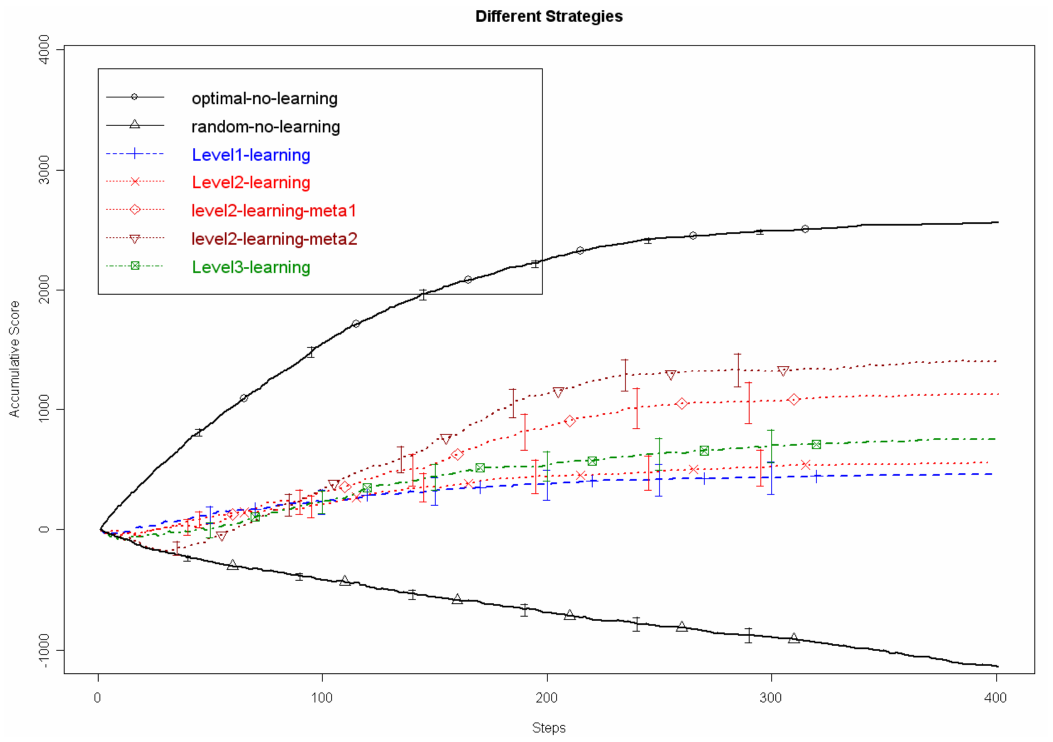


Figure 22: Performance with two new strategies and with noise in the values of food

All strategies including the two new strategies are compared in Figure 22. None of the original strategies are as good in the long run, since they don not expect noise in value of food. ‘Level2-learning-meta2’ requires more experience, so it is least likely to block edible food by random bad experiences, and has the best long-term benefit. However, the initial score of this strategy is low (close to random), as the inevitable trade-off for exploration. ‘Level2-learing-meta1’ has reasonable performance both initially and in the long-term. This experiment demonstrates the importance of exposing necessary meta-information for experience-driven learning. Since an edible food type based on observable feature can have poisonous instances due to hidden noise, and same for poisonous food types, one-shot learning is not as robust as learning the expected value by taking more samples and using statistical information provided via meta-information.

5. Summary

We have gone through increasingly more complicated tasks and incrementally refined design decisions for semantic memory.

The arithmetic task demonstrates the transfer learning effects by using declarative knowledge representation, and illustrates how different learning mechanisms (semantic learning and chunking) work in a complementary way. The implementation requires a separate declarative long-term memory and integration with existing system. The new component communicates with Soar via ‘cue’ link, ‘retrieved’ link and ‘save’ link to perform the required knowledge encoding, retrieval and application functions. In addition, internal storage structure and retrieval algorithm need to support efficient matching against entire semantic memory. Dynamics of storage structure is an open problem, for which we started from the approach taken by ACT-R.

The eater’s task demonstrates, with existence of noise, how bottom-up experience driven learning can be integrated with semantic memory in Soar. One challenge of this task is the noise from environment. Noise in input is reduced by an unsupervised hierarchical clustering component for syntactic learning. Noise in the association (hidden noise beyond the underlying input structure) is handled by semantic learning and memory retrieval meta-information which provides statistical information about a query and is exposed to knowledge level for decision making.

6. Future Work

The empirical work has led to more questions need to be addressed. Some important ones are listed below.

1) Applicability of current implementation to more tasks:
   So far we have designed tasks to demonstrate the functionality of semantic memory with particular implementations. From the other direction, we also need to learn from tasks and characterize both features that will make current approach work well and features that will make it work poorly, and motivate new design considerations. Therefore, the path is to iteratively evaluate implementations and learn from tasks, make design revisions and find new tasks.
2) Many functionalities of hierarchical clustering remain to be explored:
   First, the clustering process does not have to be limited to sensory input, and it's natural to hypothesize that the same clustering mechanism works as well for semantic memory and episodic memory at any abstraction level, which is part of the hypothesis in the original publication [21]. This may be helpful to automatically organize large body of knowledge by creating hierarchically nested structures – clustering declarative chunks that contain clustered symbols. The result is that more general semantic knowledge can be derived from more specific semantic knowledge or episodic instances.
   Second, hierarchical clustering can result in learning of stable category prototypes. In the eater's domain example, edibility of food is consistent at level2, and the eater can form the categories of edible food and poisonous food at that abstraction level. A related functionality is the speed-accuracy tradeoffs: if the edibility is consistent at level2, then the eater agent doesn't need to go to level3 in order to make the prediction.
3) Query language:
   Currently, the query language to semantic memory is limited to one level matching with variable. Other features such as negative conditions and matching at multiple levels might be required in general situations.

References

1. Anderson, J.R., et al., *An integrated theory of the mind.* Psychological Review, 2004. **111**(4): p. 1036-1060.
2. Laird, J.E., A. Newell, and P.S. Rosenbloom, *Soar - an Architecture for General Intelligence.* Artificial Intelligence, 1987. **33**(1): p. 1-64.
3. Daniel L. Schacter, A.D.E., Randy L. Buckner, *Memory Systems of 1999.* The Oxford Handbook of Memory, 2000: p. 627-643.
4. Murre, J.M.J., K.S. Graham, and J.R. Hodges, *Semantic dementia: relevance to connectionist models of long-term memory.* Brain, 2001. **124**: p. 647-675.
5. Anderson, J.R. and L.M. Reder, *The fan effect: New results and new theories.* Journal of Experimental Psychology-General, 1999. **128**(2): p. 186-197.
6. Anderson, J.R., *A Rational Analysis of Categorization.* Bulletin of the Psychonomic Society, 1988. **26**(6): p. 507-507.
7. Anderson, J.R., et al., *An integrated theory of list memory.* Journal of Memory and Language, 1998. **38**(4): p. 341-380.
8. Langley, P., Choi, D, *A unified cognitive architecture for physical agents.* Proceedings of the Twenty-First National Conference on Artificial Intelligence, 2006.
9. Minton, S., et al., *Explanation-Based Learning - a Problem-Solving Perspective.* Artificial Intelligence, 1989. **40**(1-3): p. 63-118.
10. Kieras, D.E. and D.E. Meyer, *An overview of the EPIC architecture for cognition and performance with application to human-computer interaction.* Human-Computer Interaction, 1997. **12**(4): p. 391-438.
11. Forgy, C.L., *Rete a Fast Algorithm for the Many Pattern Many Object Pattern Match Problem.* Artificial Intelligence, 1982. **19**(1): p. 17-37.
12. Craig, I.D., *Blackboard Systems.* Artificial Intelligence Review, 1988. **2**(2): p. 103-118.
13. Ingrand, F.F., M.P. Georgeff, and A.S. Rao, *An Architecture for Real-Time Reasoning and System Control.* Ieee Expert-Intelligent Systems & Their Applications, 1992. **7**(6): p. 34-44.
14. Brachman, R.J. and J.G. Schmolze, *An Overview of the Kl-One Knowledge Representation System.* Cognitive Science, 1985. **9**(2): p. 171-216.
15. Guha, R.V. and D.B. Lenat, *Cyc - a Midterm Report.* Ai Magazine, 1990. **11**(3): p. 32-59.
16. Jones, R.M., et al., *Automated intelligent pilots for combat might simulation.* Ai Magazine, 1999. **20**(1): p. 27-41.
17. Miller, C.S. and J.E. Laird, *Accounting for graded performance within a discrete search framework.* Cognitive Science, 1996. **20**(4): p. 499-537.
18. Huffman, S., Miller, C., Laird, J, *Learning for instruction: a knowledge-level capability within a unified theory of cognition.* Proceedings of the 15th Annual Meeting of the Cognitive Science Society. Boulder, Colorado, 1993.
19. Young, R.M., *The data learning problem in cognitive architectures.* Cognitive Systems Research, 2005. **6**(1): p. 89-97.
20. Nuxoll, A., Laird, J., *A Cognitive Model of Episodic Memory Integrated With a General Cognitive Architecture.* International Conference on Cognitive Modeling, 2004.
21. Rodriguez, A., J. Whitson, and R. Granger, *Derivation and analysis of basic computational operations of thalamocortical circuits.* Journal of Cognitive Neuroscience, 2004. **16**(5): p. 856-877.

22. Ambrosingerson, J., R. Granger, and G. Lynch, *Simulation of Paleocortex Performs Hierarchical-Clustering*. Science, 1990. **247**(4948): p. 1344-1348.

23. Marinier, R., Laird, J., *Toward a Comprehensive Computational Model of Emotions and Feelings.* International Conference on Cognitive Modeling, 2004.