Towards Efficiently Supporting Large Symbolic Declarative Memories

Nate Derbinsky (nlderbin@umich.edu)
John E. Laird (laird@umich.edu)
University of Michigan, 2260 Hayward Street
Ann Arbor, MI 48109-2121

Bryan Smith (bryanesmith@gmail.com)
Ann Arbor, MI

Abstract

Efficient access to large declarative memories is one challenge in the development of large-scale cognitive models. Prior work has provided an initial demonstration of declarative retrievals using ACT-R and a relational database. In this paper, we provide extended analysis of the computational challenges involved. We detail data structures and algorithms for an efficient mechanism over a large set of retrievals, as well as for a class of activation bias. We have implemented this work in Soar, and present detailed evaluation on synthetic data as well as the WordNet 3 lexicon.

Keywords: large-scale cognitive modeling; declarative memory; cognitive architecture; Soar.

Introduction

Typical cognitive models have very modest declarative memory (DM) requirements. In these cases, naïve data structures and algorithms, despite inefficiencies, suffice for declarative retrievals. However, prior work (Douglass et al., 2009) has shown that cognitive models of complex tasks require more substantial DMs, such as a large subset of the WordNet lexicon (Miller, 1995), and that existing retrieval mechanisms, such as the ACT-R implementation, do not scale to large DMs. If we are ever going to study human behavior in knowledge-rich, temporally extended tasks, additional research is required on the underlying computational data structures and algorithms that support declarative memory storage and retrieval.

In an effort to efficiently support large declarative memories in ACT-R (Anderson et al., 2004), Douglass et al. developed a DM using the PostgreSQL relational database management system. While their work produced an ACT-R module supporting persistent declarative access to large declarative knowledge stores, there are significant opportunities for extension and improvement. First, while achieving significant empirical performance improvements over the ACT-R retrieval mechanism, the authors do not address the analytical computational profile of the DM retrieval problem, thereby missing, for instance, situations in which even DBMS query optimizers will not support efficient performance. Additionally, their presented evaluation is limited to their target application and DM, and does not include any calculation of chunk activation.

In this paper, we extend that work along many dimensions. First, we contribute an extended analysis of the computational challenges of efficient declarative retrievals. To address many of these problems, we describe system-independent methods for efficient retrieval functionality. Also, while not achieving the full functionality of ACT-R activation, we move towards that goal by formulating and efficiently supporting a simpler class of activation bias.

To evaluate this work, we have implemented a semantic memory system in the Soar cognitive architecture (Laird, 2008). We evaluate the system on a scalable, synthetic data set, as well as the entire WordNet 3 lexicon. For successful retrievals on data sets scaling to millions of declarative chunks, we achieve retrieval times that are two orders of magnitude faster than previously reported results.

A forewarning: much of the presented work delves into the details of data structures, algorithms, and complexity analysis, which are critical for communicating the results of our work to developers of cognitive architectures. However, these details may be of less interest to model developers. We recommend that modelers focus on the problem formulation sections and the empirical evaluation.

Symbolic DM Retrieval Problem

To begin, we develop an abstract problem formulation of symbolic declarative retrievals. To exemplify this formulation, we then map it onto the ACT-R DM.

Problem Formulation

We define a *declarative memory* (DM) as a set of *elements*. A DM element is decomposed into a set of symbolic *augmentations*. For example, consider the following example DM, in which the letters A-D identify elements and lower-case Greek letters represent augmentations:

A: $\{\alpha, \beta, \varepsilon, \phi\}$
B: $\{\alpha, \varepsilon\}$
C: $\{\gamma\}$
D: $\{\gamma, \phi\}$

We define a DM symbolic retrieval *cue* as having a required *positive* component and an optional *negative* component, each of which is expressed as a set of symbols (corresponding to the augmentations of a DM). For instance, consider the following retrieval cue, corresponding to the example DM above, consisting of both positive (+) and negative (–) components: $+\{\alpha, \varepsilon\}$, $-\{\gamma\}$. Semantically, the positive set specifies augmentations that an element *must* contain, and the negative set those that it must *not* contain.

Given a DM and a cue, we define the *result* of a declarative retrieval to be a single element from the DM, including all augmentations, that satisfies the constraints represented semantically by the cue. Thus, the result of the example cue and the example DM would either be element A or B (with respective augmentation set {α, β, ε, ϕ} or {α, ε}). A retrieval is considered a *success* if there exists a result (as with our example) and a *failure* otherwise.

ACT-R DM

We now compare our symbolic declarative retrieval problem formulation to ACT-R's declarative memory module retrieval interface. We begin with a review of the ACT-R DM and then map it onto our definitions above.

In ACT-R, declarative knowledge is encoded as a set of *chunks*, which are collections of labeled *slots* that have *values*. For example, consider this chunk, representing one of the noun senses of the word "roach" from the WN-LEXICAL interface to WordNet (Emond, 2006):

```
(S-105261088-1 ISA S
        SYNSET-ID           105261088
        W-NUM               1
        WORD                "roach"
        SS-TYPE             "n"
        SENSE-NUMBER        1
        TAG-COUNT           0)
```

To retrieve declarative knowledge, a production rule issues a request to the declarative module by populating the declarative buffer with positive and negative slot-value pairs. These pairs are interpreted as hard constraints that either *must* be met (positive tests) or *must not* be met (negative tests). The DM module also supports non-symbolic tests (<=, >, etc), but we do not consider them.

For example, consider a cue that requests a sense chunk ("ISA S") where the value of the WORD slot is equal to "roach" and the SS-TYPE is not equal to "v" (*verb*):

```
+retrieval>
        ISA             S
        WORD            "roach"
      - SS-TYPE         "v"
```

Given this request, the ACT-R DM module searches the store for matching chunks. If any are found, the module, given default module parameter settings, indicates a successful retrieval and selects randomly amongst the candidates chunks and reconstructs it in the appropriate buffer. The module also supports the use of non-symbolic activation to bias selection amongst candidate chunks, functionality that is used in many cognitive models. We comment on this functionality later in this paper. If no perfect match is found, the default behavior of the DM is to report a retrieval failure. The module also supports the use of customizable partial matching. While some modelers may use this functionality, it makes the retrieval problem strictly harder computationally, and we leave research on an efficient implementation of it to future work.

We now map the ACT-R DM to our abstract formulation. First, without loss of generality, we interpret the chunk type (above, "ISA S") as a slot-value pair (slot label "ISA" and value "S"). Next, since we are considering qualitative matching (equality is defined as symbolic equivalence), each distinct slot-value pair can be equivalently represented as a single, composite symbol (by concatenating the slot label and value with a unique separating character, such as "ISA:S"). Since slot-value pair order is arbitrary, a chunk instance can be equivalently represented as a set of [composite] symbols. In ACT-R, all chunks of a given type must contain values for the same set of slots and a chunk type can only have one slot of a given label; without loss of generality, we eliminate both of these constraints. Given the analysis above, a chunk maps to a declarative memory *element*, and slot-value pairs to *augmentations*.

We apply a similar analysis to DM retrieval requests, with distinct slot-value pairs compressed to a single composite symbol. If we require that equivalent slot-value pairs in chunks and retrieval requests resolve to the same composite symbols, then the set of positive tests form the *positive* cue component and the negative tests the *negative* component.

With this analysis, we claim that the symbolic ACT-R DM retrieval interface is an instance of our problem formulation. Thus, results from our work, though implemented in Soar, extend to ACT-R models, and any other system that can be similarly mapped.

Supporting Efficient Retrievals

In this section, we discuss indexing structures and processes to efficiently support a large class of symbolic DM retrievals, accompanied by a brief computational complexity analysis. We decompose our description into the required *positive* cue component, followed by the *negative*. Prior to getting lost in the weeds of data structures and algorithms, however, let us first consider what is meant by *efficient support* with respect to our problem formulation.

Contextual Meaning of Efficient Support

As a baseline, consider a naïve retrieval mechanism that iterates through the DM, comparing each element to the cue, and returning the first valid result, if one exists. To understand the costs, we define E as the set of elements in a DM, and a as the average number of augmentations per element. Given a cue C, we define P as the positive cue component and N as the negative cue component. Sets surrounded with vertical bars, such as $|E|$, refer to the *cardinality*, or number of items contained in the set.

Assuming no specialized indexing, the memory cost of the baseline mechanism grows with the product of the number of elements and the average augmentation cardinality ($a|E|$). In the worst case, the baseline mechanism must traverse all of this memory for each cue element, and thus the time cost multiplies by the size of the cue ($a|E||C|$). In context of large declarative memories, it is likely that $|E|$ will dominate a and $|C|$, and thus memory and retrieval costs will scale linearly with the number of elements in the DM.

Memory, though not unlimited, is generally considered cheap and plentiful, while time is expensive and limited, and thus our goal is to minimize retrieval time, possibly at the cost of memory. Thus we pose *efficient support* for declarative retrievals as sub-linear in the number of elements in the DM, $|E|$, while remaining linear in memory. We further require that these computational bounds hold in the general case of our problem formulation, supporting a broad variety of DMs and retrieval cues, as opposed to an optimized mechanism for a specific knowledge-base and/or query load. We now present our mechanism, revisiting these requirements for theoretical evaluation.

Positive Cue Component

To review, the positive cue component for symbolic declarative retrievals is a non-empty set of augmentations that a declarative element *must* contain. To assist in our analysis, we define R_p as the elements that contain an augmentation p and, accumulated over all p in P, R to be the bag of candidate elements (which may contain duplicates, if an element contains more than one augmentation, p, in P).

Before presenting our mechanism, we note that this component of the retrieval problem is a constrained form of a *subset* query on *set-values*, which has been widely studied in database and information retrieval (IR) communities (Terrovitis et al., 2006). In its general form, the worst-case time cost is known to be linear in the sum of the number of candidate elements for each positive cue augmentation, $|R|$, though clever indexing methods have shown massive average-case improvements in real-world data.

Indexing Building on this prior work, the primary indexing structure for our mechanism is an inverted table of DM elements, combined with cached frequency statistics. The structure contains a sorted list of each augmentation, p, in the DM, each paired with a sorted list of elements in which they are contained as well as the size of this list, R_p. We note that this structure roughly doubles the size of the store and can be updated very efficiently as the DM changes. Consider the following index over the example DM above:

α (2): [A, B]
β (1): [A]
γ (2): [C, D]
ε (2): [A, B]
ϕ (2): [A, D]

Algorithm To retrieve based only on the positive cue component, we first generate a sorted list, Q, of all augmentations p in P, keyed ascending on R_p, which requires $|P|$ queries on the inverted index. Q represents a specialized query plan, sorted in ascending order of candidate element list size. With the example positive component above, Q is either $[\alpha,\beta]$ or $[\beta,\alpha]$ (as $R_\alpha = R_\beta$), and we use the former for the remainder of this analysis.

Next, we pop the first augmentation from Q (α) and retrieve a pointer, w, to the head of the element list in the inverted index (initially referring to the first element, A). Note that since this list is updated incrementally with changes to the DM, we do *not* have to compute this list in response to the query. Iterating over the remaining augmentations in Q ($[\beta]$), we verify, using the original DM, that w satisfies all remaining positive constraints. If so, return w and *success*. Otherwise, increment w to point to the next element in the inverted index and retry verification. If no element successfully verifies, the retrieval is a *failure*.

Analysis In the worst case, this retrieval mechanism grows linearly with $|E|$ (as demonstrated later). However, the small amount of indexing and query optimization bounds element iteration to $\min(R_p)$, the set of elements containing the most selective positive query augmentation. Furthermore, we only need to fully examine this list in the *failure* case, which, as we see in the later empirical evaluation, can be achieved in near constant-time queries in many cases.

Negative Cue Component

The negative cue component for symbolic declarative retrievals is an optional set of augmentations that a declarative retrieval must *not* contain.

We have struggled with how to efficiently support this type of constraint given our problem formulation. What makes this component difficult is that given a large DM with a sparse distribution of augmentations, it can be prohibitively expensive to maintain an index of the elements *not* containing an augmentation, analogous to issues surrounding the closed-world assumption and negated conditions in production matching (Doorenbos, 1995).

Initial Integration Currently, we integrate this functionality with the positive cue component above by special-casing negative augmentations. First, $|R'_n|$, the number of candidate elements that do *not* contain a particular augmentation n, equals $(|E| - |R_n|)$, the total number of elements less the number of elements that *do* contain the augmentation. This quantity can be computed efficiently and used to order Q with negative augmentations. Second, because we cannot efficiently enumerate R'_n, w is initialized as the head of the list of the first *positive* augmentation in Q. Finally, when verifying a candidate element, we simply invert the result of the set-inclusion query on E.

Analysis Using this approach, our mechanism loses a major performance benefit. This forfeiture arises when there exists an augmentation in the negative component that is more selective than any positive component augmentation, which is probably not uncommon. While we are theoretically able to integrate this functionality, we have neither implemented nor evaluated this work empirically in Soar, and plan to address this deficiency in the future.

Supporting Efficient Activation Bias

A major contribution of the ACT-R DM module to cognitive modeling is the sub-symbolic influence of the current context and prior retrievals as a form of activation bias for declarative retrievals (Anderson et al., 2004). This functionality, however, has been shown to come at a significant computational cost that does not scale to large declarative memories (Douglass et al., 2009).

While we have not achieved the functionality of all aspects of ACT-R's activation scheme, we have made progress by formulating and efficiently supporting a simpler class of activation bias. In this section, we first extend our problem formulation to include retrieval bias, then define the class of activation update processes we can efficiently support, and discuss how we achieve this functionality.

Problem Formulation Extension

To integrate activation bias in our problem formulation, we extend our definition of a declarative memory element to include a numerical *activation value*, as exemplified below by the numbers in square brackets:

A [1.41]: $\{\alpha, \beta, \varepsilon, \phi\}$
B [1.73]: $\{\alpha, \varepsilon\}$
C [3.14]: $\{\gamma\}$
D [2.72]: $\{\gamma, \phi\}$

We refine our previous definition of a retrieval result as an element from the DM, including all augmentations, that satisfies the constraints represented semantically by the cue *and* has the maximal activation value. Given the example cue ($+\{\alpha, \varepsilon\}, -\{\gamma\}$) and this expanded DM, the result is now unambiguously B (and its associated augmentations), as it has a greater activation value than A.

Efficient Activation Bias Updates

The expanded retrieval mechanism described in the next section efficiently incorporates activation. However, just as the DM must support efficient updates to elements and augmentations, so too must it support efficient updates to activation values. In this context, for large DMs, we propose that an activation value update process must be *locally efficient*. An activation update process is locally efficient if it satisfies two properties: (1) the update can affect the activation value of *at most* a constant number of elements and (2) updating the activation value of an element takes time strictly sub-linear in the number of DM elements.

The locally efficient activation update process we implement in Soar is a straightforward mechanism to bias retrievals towards recency. After each successful retrieval, the activation value of the retrieved element is updated to be one greater than the previously largest activation value. This update process is local, as it only changes a single element per retrieval, and it is efficient, as the largest activation value can be cached to avoid any search over *E*.

In ACT-R, chunk activation includes retrieval history (base-level), current context (spreading), partial matching, and noise. Both the base-level approximation and permanent noise computations appear to be local, so it should be possible to extend our approach to cover those components. However, transient noise, partial matching, and spreading activation appear to be global to the elements of the DM, which suggests significant further theoretical and engineering research are necessary to develop locally efficient mechanisms. For reference, the mechanism in Douglass et al. does not efficiently compute any portion of ACT-R chunk activation, and those components were not included in their empirical evaluations.

Efficient Support

The most direct method of integrating activation values in our efficient algorithm is to sort the candidate list (w) by activation values on demand. This approach, henceforth referred to as Scheme I, suffers from retrieval times that are *always* dependent upon augmentation selectivity, as the candidate list must be fully computed to be sorted.

Another method of integrating activation values, Scheme II, is to maintain, for each augmentation, an element list sorted by activation value. Thus, w is sorted in order of activation, independent of augmentation selectivity. However, the time required for updating activation values is dependent upon the number of different augmentations an element can have (its augmentation cardinality), and for large cardinalities, this cost can be prohibitive.

Our approach to integrating activation values combines these schemes by exploiting an assumption that most elements will have "small" augmentation cardinality. Given this information, we explain how we can extend our implementation to yield efficient retrievals and then we validate our assumption empirically by studying three large, commonly used knowledge bases.

Our Approach. If an element has small augmentation cardinality, Scheme II is efficient, independent of DM size. If few elements must be sorted per retrieval, Scheme I is efficient, independent of element augmentation cardinality. To resolve this tension between augmentation cardinality and element selectivity, we apply these schemes on a per-element basis: we apply Scheme II when an element has small augmentation cardinality, and otherwise apply Scheme I. What we describe here are the data structure modifications and additional processing necessary to efficiently implement this split strategy.

First, we introduce a threshold parameter, t, which represents a *small* value of augmentation cardinality. By default, we integrate activation bias as described in Scheme II above. However, if the augmentation cardinality of a particular element is greater than t, we associate a one-time special "infinity" (∞) activation value with all its augmentations and maintain a separate list associating the element with its activation value, per Scheme I. For instance, if t=3, we would have a list wherein [A=1] and our inverted index would contain the following information:

α (2): [A=∞, B=2]
β (1): [A=∞]
γ (2): [D=4, C=3]
ε (2): [A=∞, B=2]
ϕ (2): [A=∞, D=4]

By default, an update to an element's activation value will involve updating a small number of references ($\leq t$) throughout the inverted index. For elements with augmentation cardinality greater than t, such as A, we need only update this value once, thereby bounding the update to constant time and addressing the weakness of Scheme II.

During retrieval, as we are populating the list of augmentations, *Q*, which is sorted by activation level, we may now encounter one or more infinite activations at the head of the list. If so, we perform a lookup for its true activation level and execute insertion sort into a second, special list, *Q'*. We then merge *Q* and *Q*' to form our query plan. Notice that if the size of *Q'* is small (i.e. few elements have augmentation cardinality greater than *t*), this process is cheap and independent of augmentation selectivity, the weakness of Scheme I. Thus, if we can select an appropriate value of *t*, we will achieve efficient activation bias support.

Validation. To validate that our split strategy works well on real data sets, we studied three large, commonly used knowledge bases (KBs): SUMO (Niles et al., 2001), OpenCyc (Lenat, 1995), and WordNet (Miller, 1995). For each KB, we extracted the number of features of each named entity. Each distribution was unimodal and exhibited strong right skew, suggesting that while most elements had a similar feature size, there were rare cases with exceptionally large cardinalities. Then, we sampled from these distributions to form synthetic data sets that were reasonably large (5040 elements) and empirically valid in augmentation cardinality. We then collected empirical retrieval data, summarized in Table 1, showing that for each KB there was a range over the value of *t* that optimally balanced the performance effects of cue selectivity and augmentation cardinality. For two of the KBs, we could efficiently employ Scheme II above for more than 99% of elements, versus only about 93% for the SUMO data set.

Important components of this analysis for future examination are (1) automatically selecting a value of *t* for a given DM and (2) tuning this value online for changing DM contents. As to the former, we see in Table 1 that the optimal threshold typically covers greater than 90% of the elements using augmentation cardinality, but that value is not constant across data sets. Further analysis of the KBs may uncover why this is the case and suggest better factors for prediction. As for the latter, we expect that *caching t* in indexing structures will allow the algorithm to adapt in real time, while maintaining efficient retrievals.

Table 1: Optimal Thresholds.

Data Set	Optimal *t* Range	Element Coverage
SUMO	50 – 70	92.78 – 93.86%
OpenCyc	40 – 60	99.17 – 99.74%
WordNet	20 – 40	99.50 – 99.90%

Evaluation

To evaluate our work, we implemented our data structures and algorithms as the Semantic Memory long-term, symbolic memory system in the Soar cognitive architecture (Laird, 2008). We used version 3 of the SQLite in-process relational database engine to manage the semantic store and all experimental results were run on a 2.8GHz Core 2 Extreme processor with 4GB of RAM.

Our final evaluation spans two data sets: (1) the WordNet 3 lexicon and (2) a scalable synthetic benchmark of our design. WordNet offers a large, ecologically valid knowledge base with which we can compare to previous results in this space (Douglass et al., 2009). Our synthetic dataset offers us the ability to exhaustively benchmark our retrieval mechanism on arbitrarily large DMs.

WordNet

As with Douglass et al., we used the WN-LEXICAL WordNet 3 data conversion (Emond, 2006). The data set has over 820K chunks, which includes over 212K word/sense combinations. Once imported, Soar's semantic store, including all indexing structures, is about 400MB.

Our first experiment was to verify (a) that retrieval time was independent of augmentation selectivity and (b) that the activation bias was processed efficiently in under-specified cues. We performed DM retrievals on 100 randomly chosen, single-augmentation cues, averaged over 10 trials. Retrieval time was 0.1887 msec. each (0.0216 std. deviation).

Our next experiment focused on larger cues. We randomly chose 10 nouns and formed a cue from their full sense description (such as the "roach" example above). Retrieval time was an average of 0.2973 msec. over 10 trials each (0.0108 std. deviation).

Douglass et al. used a derived subset of the WN-LEXICAL dataset, so direct replication of their work is difficult. They reported retrievals of about 40 msec. with cues of 1-4 augmentations on a DM with about 232.5k chunks. Our results show 100x faster retrievals on a comparable set of cues scaling to a 3x larger DM.

Synthetic Data

In addition to running on a known data set, we tested our implementation more exhaustively to measure how it scales with much larger DMs. We developed a scalable, synthetic DM generator and, in Table 2, we list statistics of the data sets we used as they scale with *k*, the size control parameter:

Table 2: Synthetic Statistics.

k	Elements	Store Size (MB)
7	5,040	3.00
8	40,320	27.81
9	362,880	291.95
10	3,628,800	2048.00

While we have a DM generator, we do not have a model of what are typical cues used to access a DM and how those cues could interact with the performance profile of the DM retrieval mechanism. For instance, we do not know how *selective* the cues are likely to be, meaning how many elements, termed *candidates*, could possibly satisfy any part of the cue. Furthermore, we do not know the proportion of cues that will have no perfect matches. To allow us to test these different interactions, we constructed the DMs so that we can generate cues with independently controlled selectivity. In each KB, there are *k*! elements and each element has augmentation cardinality of (k+1). For i = 2 … k, the ith augmentation of an element has selectivity (k!/i). The 0th augmentation of each element is shared by all elements and the 1st augmentation is unique.

Selectivity Sweep. Our first question is whether the DM mechanism provides bounded retrievals for under-specified cues, independent of the number of candidate elements. For each distinct augmentation in the DM, we constructed a cue and measured retrieval time. We found nearly constant-time retrievals within each data set, independent of augmentation selectivity, measuring just under 0.4 msec. for k=10.

Cue Sweep. Our next question is whether combinations of augmentations result in complex cues that adversely affect retrieval time. We constructed all possible lengths of cues using all combinations of augmentation selectivity and measured retrieval time. As shown in Figure 1, the only factor affecting retrieval time within a data set was the number of augmentations in the cue ($R^2 \approx 1$), achieving a maximum of about 0.5 msec. for k=10.

Failure Sweep. For our mechanism, retrieval failure is the algorithmic worst-case, as it must examine and fail to verify all candidate elements. We constructed our last experiment to measure retrieval time for cues that fail only after examining significant proportions of the elements in the KB. While our mechanism minimizes the chance of this situation, these results are useful to set an expectation for the unlikely worst-case retrieval time in any given DM. As shown in Figure 2, the number of inspected candidate elements was the only factor affecting retrieval time, independent of the data set. Because the time is linear in the number of candidates, and not the total number of KB elements, our mechanism, for even worst worst-case cues, scales to arbitrarily large data sets when cue augmentations are sufficiently selective.

Conclusions

In this work, we formulate and address the computational challenges involved with supporting efficient symbolic retrievals for the core functionality required in representing and accessing large DMs. We extend the research of

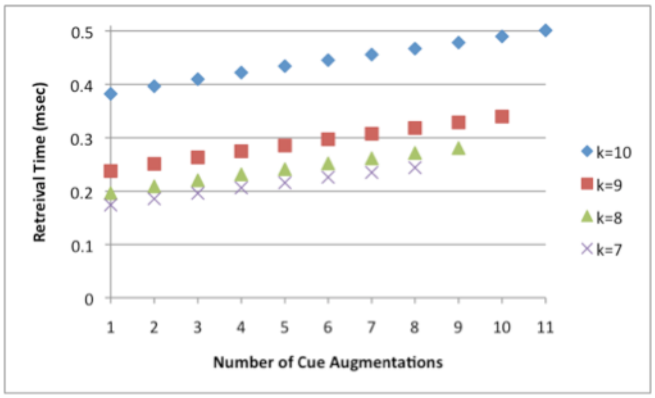

Figure 1: Synthetic cue sweep results.

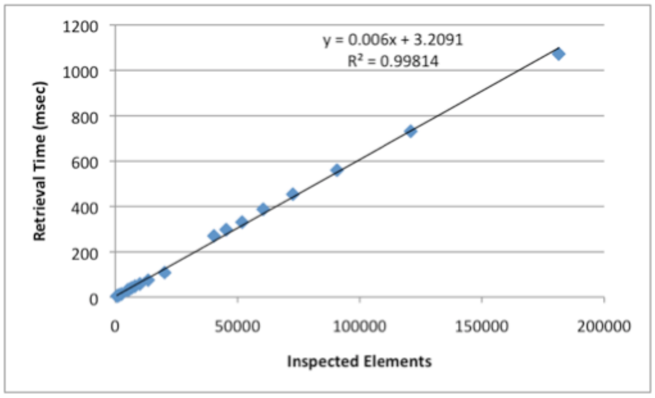

Figure 2: Synthetic failure sweep results.

Douglass et al., demonstrating two orders of magnitude improvement in retrieval times for comparable functionality on significantly larger data sets. There are still challenges ahead to efficiently support partial match, spreading activation, and other non-local biases for retrieval for large data sets, for which it may be necessary to explore algorithm approximations or massively parallel computation.

Acknowledgments

The authors acknowledge the funding support of the Office of Naval Research under grant number N00014-08-1-0099.

References

Anderson, J.R., Bothell, D., Byrne, M.D., Douglass, S., Lebiere, C., & Qin, Y. (2004). An Integrated Theory of the Mind. *Psychological Review* 111, (4). 1036-1060.

Doorenbos, R.B. (1995) *Production Matching for Large Learning Systems*. PhD Thesis, Carnegie Mellon.

Douglass, S., Ball, J., & Rodgers, S. (2009). Large Declarative Memories in ACT-R. *Proc. of the 9th International Conference on Cognitive Modeling.*

Emond, B. (2006). WN-LEXICAL: An ACT-R Module Built from the WordNet Lexical Database. *Proc. of the 7th International Conference on Cognitive Modeling*.

Laird, J.E. (2008). Extending the Soar Cognitive Architecture. *Proc. of the First Conference on Artificial General Intelligence (AGI)*.

Lenat, D. (1995). CYC: A Large-Scale Investment in Knowledge Infrastructure. *Communications of the ACM* 38, (11). 33-38.

Miller, G.A. (1995). WordNet: A Lexical Database for English. *Communications of the ACM* 38, (11). 39-41.

Niles, I., Pease, A. (2001). Towards a Standard Upper Ontology. *Proc. of the Second Conference on Formal Ontology in Information Systems (FOIS)*.

Terrovitis, M., Passas, S., Vassiliadis, P., Sellis, T. (2006). A Combination of Trie-trees and Inverted Files for the Indexing of Set-valued Attributes. *Proc. of the 15th Conference on Information and Knowledge Management (CIKM)*.