Efficiently Implementing Episodic Memory in Soar

Nate Derbinsky
September 8, 2008

Abstract. Endowing an intelligent agent with an episodic memory affords it a multitude of cognitive capabilities that may be crucial to its efficacy. However, implementing a task-independent episodic memory within a cognitive architecture, while maintaining reactivity over long agent lifetimes, requires efficient algorithms and data structures to store and retrieve memories in a potentially monotonically increasing episodic store. This work is a data-driven exploration into efficient methods for implementing an episodic memory in the Soar cognitive architecture. We present results from Soar-EpMem, an efficient, task-independent integration of episodic memory with Soar 9.

1. Introduction

Episodic memory, as first described by Tulving, is a long-term, contextualized store of specific events [14]. Episodic memory, what an individual "remembers," is contrasted by semantic memory, a long-term store of isolated, de-contextualized facts that an individual "knows." As an example, a memory of viewing artwork during one's last vacation would be episodic, whereas recalling the name of the gallery would likely be semantic (unless, for example, producing this information relied upon a memory of reading a brochure).

Episodic memory research is closely related to studies in case-based reasoning (CBR). In CBR, an agent maintains a database of situations faced and the solutions to problems encountered [4]. When faced with a similar problem, the agent can make use of adaptation and creativity to develop innovative solutions and improve problem performance. However, in CBR systems, the structure of cases is pre-specified by agent designers, limiting their generality. Additionally, CBR databases are typically fixed in size or grow at a limited rate. By comparison, an episodic store can be considered a highly generalized case database to facilitate episodic learning, CBR, as well as other learning mechanisms over long agent lifetimes.

Nuxoll has demonstrated that making effective use of an episodic store affords an intelligent agent a multitude of cognitive capabilities that may be crucial to its efficacy [9]. For example, an agent that examines the results of actions taken in situations similar to its current state can learn to predict the immediate consequences of its actions (i.e. action modeling). In order to facilitate further research into effective use of these cognitive capabilities, the major problem we address is to efficiently implement task-independent episodic storage and retrieval over long agent lifetimes in the Soar cognitive architecture. In this work we introduce Soar-EpMem: an efficient, stable integration of Soar 9 with an artificial episodic memory system.

The remainder of this paper is organized as follows. Section 2 describes some related work, including additional characterization of artificial episodic memory implementations. Section 3 details the functional specifications adopted for Soar-EpMem. The bulk of the research is described in Section 4, which walks through the iterative development and evaluation of alternative implementations for Soar-EpMem. This part concludes with final performance results, demonstrating an order of magnitude improvement in memory utilization and execution time as compared to our original implementation. These results suggest that a Soar-EpMem agent can operate nearly 2 days of non-stop execution in cognitive real-time, making our episodic memory implementation feasible for real-world, long-living agents. A high-level discussion of research contributions and future work is provided in Section 5. The paper concludes with referenced works and appendices.

2. Related Work

This section briefly discusses research related to this work. Nuxoll (2007) gives a more comprehensive overview of artificial episodic memory implementations; however, few of them concentrate on comprehensive implementations as well as efficient storage and retrieval. Here we focus on two systems not covered by Nuxoll that implement many aspects of a task-independent episodic memory and that have lessons for efficient implementation. We then discuss Nuxoll's characterization of artificial episodic memory implementations.

2.1. MAC/FAC

MAC/FAC is a model of similarity-based retrieval that has been shown to reproduce interesting psychological phenomena [3]. The MAC/FAC framework dictates a two-stage process for judging similarity between memory items (episodes) and probes (cues). The second stage, FAC ("few-are-chosen") involves a rich, structural comparison between candidate episodes and the cue. While this process may be reasonably efficient on a small scale, it cannot be tractably applied to large sets of episodes. Therefore, the MAC ("many-are-called") stage applies a computationally cheap filter to constrain the size of the FAC candidate pool. Gentner and Forbus implement the MAC stage using a novel, *task-specific* "content vector" that can be efficiently generated and stored for each episode. When a memory probe is proposed, a content vector based on this query is generated and compared with all memory item vectors. The top 10% of matches continue to the FAC stage, thus achieving tractability.

Evaluating MAC/FAC in context of an efficient episodic memory implementation yields two major points of contention. First, MAC/FAC's strengths are in efficiently querying a small number of content-rich, structured memory items. The experimental setup included only 32 episodes, each of which was a feature-rich story. In contrast, a cognitive agent may develop thousands to millions of episodes, though each episode is likely to be significantly less complex than an analogical story (see section 4.1). Second, a major goal of the MAC/FAC model is to reproduce human psychological phenomena, including inaccuracies in query retrievals with respect to surface vs. structural similarity. However, this model may be incompatible with the characterization of a given episodic memory implementation, particularly with respect to retrieval selection. For example, the functional specification for Soar-EpMem (see section 3.2) calls for optimal results with respect to cue-based retrievals.

While many goals of MAC/FAC differed from our own, we were inspired and informed by its methods. The content vector approach inspired the bit-vector representation in our Hybrid indexing method (see section 4.7). The idea of applying a relatively expensive computational filter at a size-limited second stage inspires our future work on "provenance" as a structural bias (see section 5). Also, our future ventures into evaluating non-optimal, cue-based query retrievals (see section 5) will likely be informed by MAC/FAC results.

2.2. EM: A Generic Memory Module for Events

Tecuci and Porter developed EM as a generic store to support episodic memory functionality in a variety of systems, including planning, classification, and goal recognition [13]. EM itself exists as an external system with a well-defined API, where host systems must implement a thin interface layer. The term "episode" in EM defines a sequence of actions with a common goal and is represented as a dimension triple: context ("general setting" of the episode), content (ordered set of the events that make up the episode), and outcome (a domain/task-specific evaluation of the result of the episode). Queries on EM take the form of a partially defined episode and a single evaluation dimension (from amongst the three defined above). EM utilizes a MAC/FAC-type two-stage evaluation scheme, whereby candidate episodes are first pruned to a constant number of potential matches (5 in [13]), which are then compared using a relatively expensive semantic matcher.

When evaluated as an episodic memory system, several design decisions for EM come into question. First, the definition of an episode in EM is quite limiting for general-purpose intelligent agents. While a useful and efficient metric in systems like planners, designers of long-living cognitive agents may not be able to easily define action sequences and outcome evaluation functions at design-time. It should be noted that this type of abstraction could be implemented at the agent level within our system, which, as a further benefit, allows for standard architectural learning mechanisms to affect the episode evaluation process. Second, in order to maintain reactivity, EM limits second-stage evaluation of episodes to a constant number of candidates, as defined by a compile-time parameter in the host system. While Tecuci and Porter have shown positive results for learning in short (250 episode), single-task domains, it is unclear that such a constraint is reasonable in a long-living cognitive agent. There is also a question as

to whether the first stage filter can scale to the significantly larger number of episodes we wish to store. Finally, EM makes a strong case for a system-independent representation of episodes. However, the potential performance cost of an external system may be considerable in long-living agents. Our choice of SQLite as an in-process storage library (see section 4.2) was motivated by the desire for a standard external representation, while still maintaining high run-time performance.

2.3. Characterizing Artificial Episodic Memory Implementations

In his work, Nuxoll has constrained artificial episodic memory implementations by several functional characteristics [8]. First, it is an architectural capability, in the sense that its functionality does not change from task-to-task (though agents can indirectly affect the storage process through deliberate rehearsal). Second, storage of episodic memories proceeds without deliberate action by the agent. Third, retrieved episodic memories are autonoetic: the agent is able to distinguish details of a memory from the current environment. Finally, episodic memories are temporally indexed, such as to afford the agent a relative sense of when the episode occurred as compared to other memories.

Nuxoll has furthermore explored the variables that define the space of artificial episodic memory implementations [9]. Functional specifications in the next section will be enumerated in context of these variables. It should be noted that design decisions within this space, particularly dynamics of episodic storage, can lead to a monotonically increasing episodic store.

3. Soar-EpMem

This section discusses the functional specifications adopted for our Soar-EpMem implementation. We begin by discussing the Soar cognitive architecture and then proceed to the issues involved with integrating an episodic memory. Finally, we review important algorithmic details adapted from Nuxoll's implementation.

3.1. Soar

Soar is a symbolic cognitive architecture that uses a production system to encode its procedural knowledge [6] [7]. Soar has a working memory (WM), in which to store short-term knowledge describing its current state, and a production memory, in which to store long-term knowledge (encoded as condition => action rules).

Soar's WM is implemented as a directed, connected graph of working memory elements (WMEs). Each WME is a triple, consisting of an "identifier," "attribute," and "value." The identifier is a reference to an existing WME within working memory. The attribute is a string constant, providing a semantic descriptor of the augmentation provided by the WME to the associated identifier. The value can be a reference to an existing identifier or a constant (numeric or string). For any identifier, Soar implements an attribute-value pair set: thus no two WMEs can have the same identifier, attribute, and value.

The following sequence illustrates the Soar execution cycle:

Input => Proposal => Selection => Application => Output

During the input phase, environmental data is populated in a special input branch of working memory. Based upon the agent's current state, rules may match. During the proposal phase, these rules will nominate potential actions (termed "operators") that can be applied in the current state. Rules also fire to assert preferences as to the relative desirability of applying the proposed operators in the current state. During the selection phase, asserted preferences are compared, a decision is made, and a single operator is selected. In the application phase, more rules may fire as a result of operator selection. Finally, a special output section of working memory is sent to the environment in the output phase.

3.2. Episodic Memory Integration

We describe here the functional specifications we have adopted for our episodic memory integration, enumerated in context of Nuxoll's design space [9]. Note that most design decisions mirror Nuxoll's baseline implementation.

Encoding

Encoding Initiation – The automatic "trigger" (per the second functional requirement of an episodic memory implementation) of a new memory to be encoded is left to the agent designer as a run-time parameter. Possible triggers for encoding new episodes include taking an action in the world and the end of a decision cycle. When a new memory is deemed necessary, the actual encoding/storage step takes place during the output phase at the end of the current Soar decision cycle.

Episode Determination – In order to implement a task-independent episodic memory (per the first functional requirement), episodes consist of most of Soar's working memory. This includes input (sensing), internal data structures from the "top state" (excludes any sub-goal reasoning), and output (actions taken in the world). For efficiency purposes, Soar's working memory graph structure is reduced to a tree and multi-valued attributes are "flattened" (see section 3.3). For performance reasons, a run-time parameter is afforded to the agent designer to "exclude" certain features from being stored (see section 4.7 for discussion of the Hybrid indexing approach).

Feature Selection – All features of the episode are eligible to participate in retrieval.

Storage

Episode Structure – Episode encoding is dependent upon an "indexing" run-time parameter (indexing implementation details are discussed in sections 3.3, 4.5, and 4.7).

Episode Dynamics – Our episodic memory integration has no episode dynamics.

Retrieval

Retrieval Initiation – The agent triggers an episodic retrieval by deliberately creating an episodic cue in Soar's working memory.

Cue Determination – An agent uses productions to construct a cue in a special "epmem" link in working memory, similar to structures used for input/output. The cue can include any number of working memory elements. For purposes of match determination, the root of the query structure corresponds to the top state of working memory. The agent can specify additional cue modifiers, including negative cue elements (indicator of elements whose presence in a candidate episode is non-desirable), a valid temporal range (specified as "before" and/or "after" a fixed temporal id), and episode prohibition (exclusion of enumerated episodes from retrieval, such as to allow the agent to deliberately choose episodes designed by the architecture as "second-best," "third-best," etc).

Selection – During cue-based episodic retrieval, candidate episodes are defined as containing at least one cue element. The best candidate is defined as the episode with the greatest "balanced" sum of match cardinality and element weighting (using recency of the episode as a tie-breaking bias). Feature weighting is provided by Soar's Working Memory Activation (WMA) system. The balance between match cardinality and element weighting is a run-time parameter. Agents are also able to perform non-cue-based episodic retrievals based upon a temporal id, as well as request an episode immediately preceding or following the last successfully retrieved episode.

Retrieval – Episodes are retrieved to a reserved structure of Soar's working memory, such as to avoid confusion with the current state (per the third functional requirement). Episodes contain all stored elements and structure.

Retrieval Meta-Data – Retrieved episodes provide match confidence and relative temporal identification (per the fourth functional requirement).

This functional specification defines three core episodic operations: storage, cue-based retrieval, and non-cue-based (NCB) retrieval. During storage, the episodic memory system associates a subset of the WMEs currently in Soar's working memory with the current temporal id. Given a retrieval cue (and potentially cue modifiers), cue-based retrieval searches the store for candidate episodes, ranks them with respect to the cue/modifiers, selects the best match, and then reconstructs the episode in WM. Given a valid temporal id, NCB retrieval is simply the last phase of a cue-based retrieval, namely reconstructing an episode in Soar's working memory.

The theoretical complexity implications of this functional specification should be noted. The lack of Episode Dynamics (such as forgetting) dictates a monotonically increasing episodic store. The storage requirement will depend upon Encoding Initiation, but, due to our choice of Encoding Determination, will at least grow linearly with the agent's changes in Soar's working memory, with respect to trigger frequency. Due to Cue Determination and Selection specification, the cue-based retrieval process must, in worst case, require inspection of all episodes and thus retrieval time will grow at least linearly in the number of recorded episodes.

3.3. Initial Episodic Indexing Methods

In his integration of episodic memory with Soar, Nuxoll developed two indexing methods for organizing episodes within the store: "instance" and "interval" (henceforth referred to as Instance and Range, respectively) [9]. Both relied upon developing a structure called a Working Memory Tree (WMT). The WMT was a compact, structural representation of all WMEs that had ever existed in WM during the lifetime of an agent. Nuxoll showed reduced processing time during episode reconstruction when using the WMT, as opposed to storing the entire working memory structure with each episode.

The WMT was constructed iteratively during storage of new episodes. During this process, the episodic memory system would simultaneously walk the WMEs of Soar's working memory and the current WMT in a breadth first manner. If this dual-traversal visited a WME in working memory that did not exist in the WMT (meaning no WME with the attribute/value pair existed in the same structural location in the WMT), it was added to the WMT. If the WME already existed in the WMT, or the WME had already been visited (as in the case with cycles), no action was taken. When the WM traversal completed, the WMT would contain structures representing all unique attribute-value pairs in current WM, as well as the WM of all previous episodes.

To maintain efficiency in cue-based retrievals, the WMT exhibited a "flattening" behavior with respect to multi-valued attributes (defined as two or more WMEs with the same identifier and attribute, but different values). When the dual WM-WMT traversal encountered an identifier node with more than one child node identifiers of the same attribute, all grandchildren of these nodes in WM would be grouped under a single identifier in the WMT (see Figure 1).

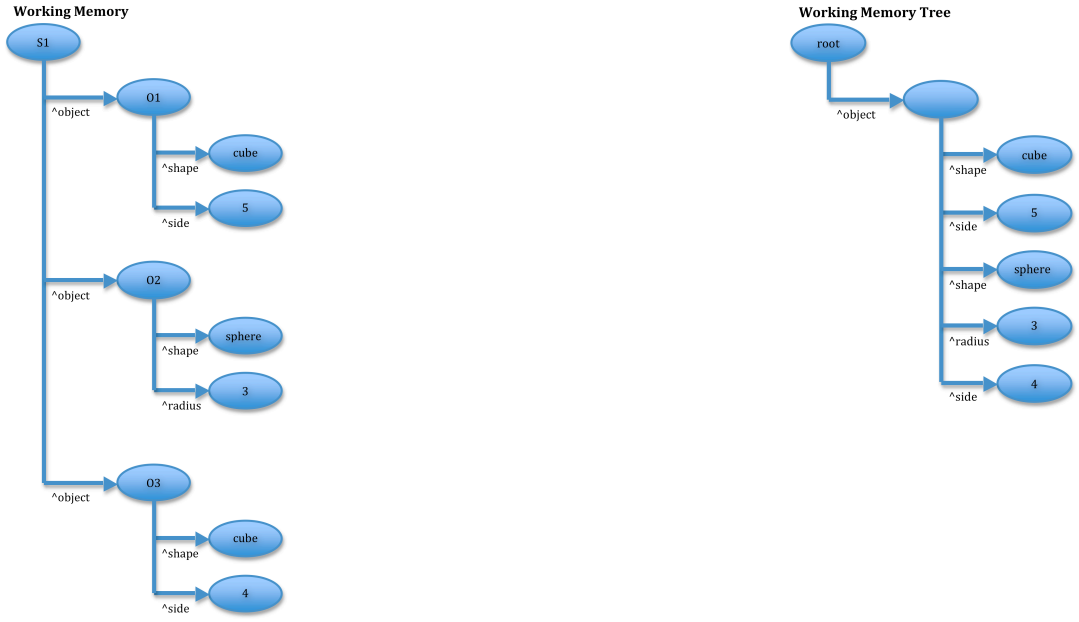

Figure 1: Working Memory to Working Memory Tree Example

Given a WMT, an episodic memory indexing method is simply an organization that explicitly or implicitly associates episodes (temporal ids) with nodes in this tree. The most straightforward approach is to define an episode as a list of pointers to nodes within the tree. This was Nuxoll's Instance approach (see Figure 2) [8].

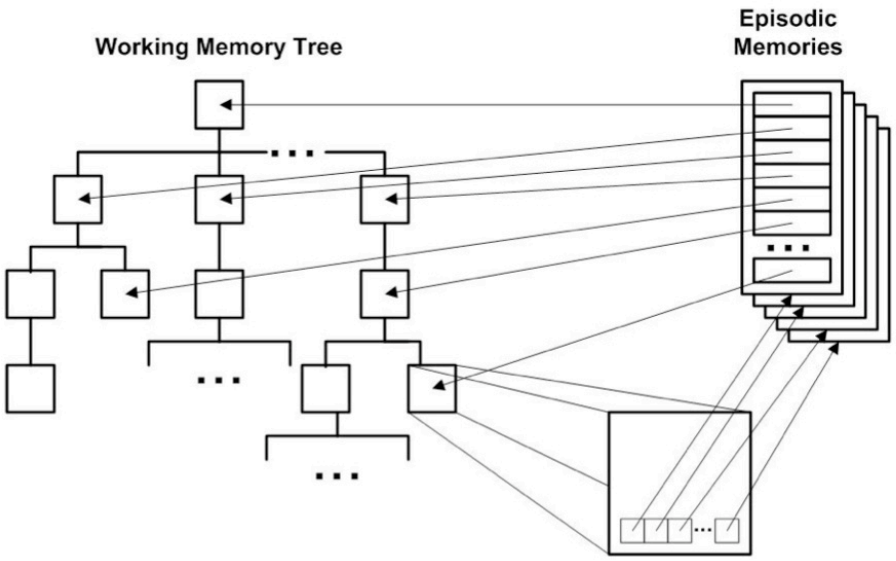

Figure 2: Nuxoll Instance Indexing Method

While episodic operations are computationally very simple, the Instance indexing method is very memory-intensive, requiring space O(nm), where *n* is the number of recorded episodes and *m* is the average number of items in working memory during a single episode.

Nuxoll claimed to improve upon the Instance method by recognizing and exploiting the fact that WMEs tend to persist in WM for more than one episode. In the Range method, we no longer represent episodes explicitly, but rather associate with each node in the WMT a list of ranges, designating temporal ids during which the WME was present (see Figure 3). Episodic operations become more complex in this representation, but memory requirements are significantly reduced, depending instead upon *changes* in working memory.

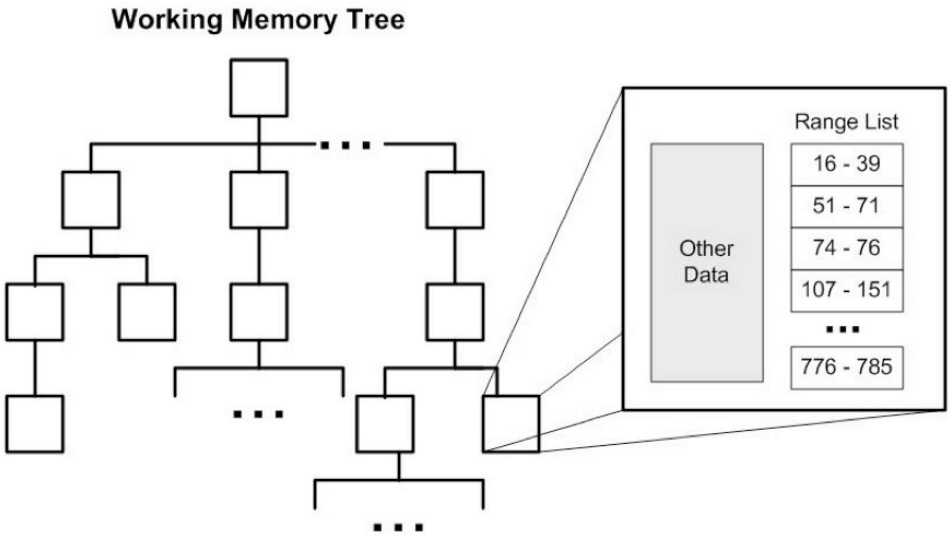

Figure 3: Nuxoll Range Indexing Method

4. Research Progression

This section discusses the research progression for our work. We begin by discussing the experimentation methodology. We then move step-by-step through the experimental algorithmic choices made, providing results in each stage, leading to the final Soar-EpMem implementation.

4.1. Experimentation Methodology

In examining our problem statement for this research, the term "efficient" is quite prominent. However, very little work has been done to establish performance benchmarks for artificial episodic memory implementations. As a result, we found ourselves guided by theoretical bounds (discussed in section 3.2) and intuition with respect to the core episodic operations. Thus, our work was a data-driven exploration, and so it is in this format that we present our progression. Subsequent subsections represent phases of study. In each phase we ran an experiment representative of one or more core operations, made observations regarding any discrepancies from our expectations, hypothesized as to the cause, implemented one or more contributions to the existing system, and observed the results with respect to the original experiments. Through this iterative process we have made major performance leaps, both in operation time and memory consumption. It should be noted that in the interest of time, the data we collected prior to the final phase were typically low in quality, used only for establishing and verifying intuition about run-time performance. Also, for convenience, the data sets in the legends of all graphs are ordered relative to their respective appearance in the graph (top-to-bottom, greatest-to-least).

Our experiments, as a continuation of Nuxoll's work, consisted of running variations on the standard *simple-bot* agent within the default map of the TankSoar domain [9]. In TankSoar, agents control tanks that move in a two-dimensional maze. Agent actions include turning, moving, firing missiles, raising shields, and controlling its radar. The TankSoar agent has access to a rich set of environmental features through its senses, including smell (shortest-path distance to the nearest enemy tank), hearing (the sound of a nearby enemy), path blockage, radar feedback, and incoming missile data. In context of a single TankSoar agent, "running to completion" refers to executing the agent within Soar for a fixed number of decision cycles (as designated in an experiment instance).

The *simple-bot* agent working memory typically contains about 100 elements on any given decision cycle (with an observed maximum of over 200). A majority (usually 70-90%) of the agent's WMEs will change within one episode, defined as the time between actions taken in the environment. When run by itself on the default map for about 10,000 decision cycles, *simple-bot* will typically produce about 50 unique non-identifier attributes and about 300 unique attribute-value pairs in working memory. Additionally, the TankSoar environment provides two utility elements, "random" and "clock," that update every decision cycle to provide a random number and indicator of the current world count, respectively. While we have not yet profiled multiple domains with Soar-EpMem (see section 5), we expect that these properties, especially the proportion of WMEs changing per decision cycle, make TankSoar an atypically stressful environment for episodic memory experimentation.

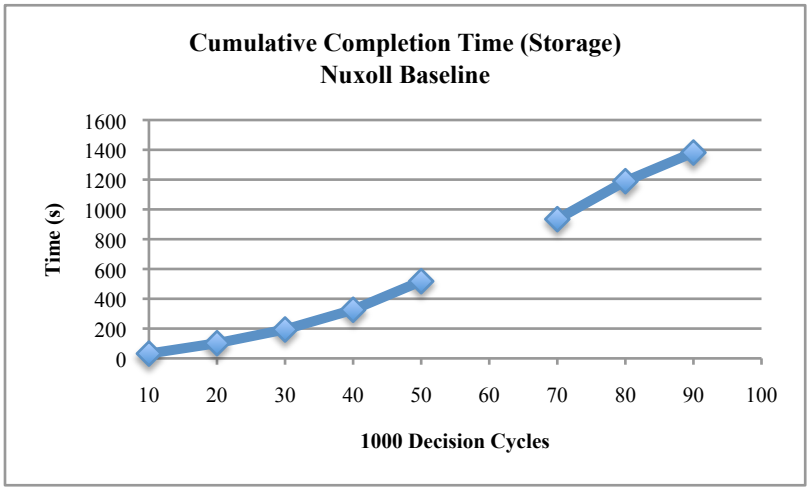

Figure 4: Phase 1

4.2. Phase 1: Improving Nuxoll's Baseline

We began our work with Nuxoll's baseline implementation [9]. We designed a *storage* agent, based on *simple-bot*, which recorded episodes after each decision cycle (the agent does not perform any type of episodic retrieval during execution). Running this agent produced the data shown in Figure 4. Note that the x-axis reports not only the number of decision cycles, but also the number of episodes stored in memory. Holes in the data (at 60k and 100k) represent crashes during the experimental run. After additional use-cases we found three core problems: (1) the system was noticeably slow and suffered from instability; (2) Nuxoll's Range indexing method had not been integrated into Soar; and (3) there was not an efficient episode memory-disk policy.

Our foundational hypothesis for all later work was that the Soar-EpMem functional specification (see section 3.2) had many elements that could be efficiently implemented within a standard relational database management system (DBMS). Furthermore, most of the observed weaknesses of Nuxoll's implementation aligned to established strengths of DBMSs: efficient and stable management of large data sets and associated queries (relating to 1 above) and well-defined policies for efficient, transactional memory-disk management (3).

We integrated Nuxoll's Instance and Range indexing methods within Soar 8.6.4 beta (and later ported the work in Soar 9). We opted to use SQLite as a standardized, in-process external storage library due to its high-level, yet efficient support for standard SQL queries and B*-Tree (hereby referred to as b-tree) indexes; seamless support for in-memory and disk storage; as well as strong community support for the standardized file format, allowing for external tools to access episodic stores.

A major contribution of this phase was an efficient mapping of Nuxoll's implementation onto a relational schema (see Appendix A). For clarity, in both Instance and Range methods, the WMT was mapped onto the *ids* relation and the implicit/explicit representation of episodes was mapped onto the *episodes* relation. Note that the conceptual details of Nuxoll's indexing methods, described section 3.3, did not change during the mapping.

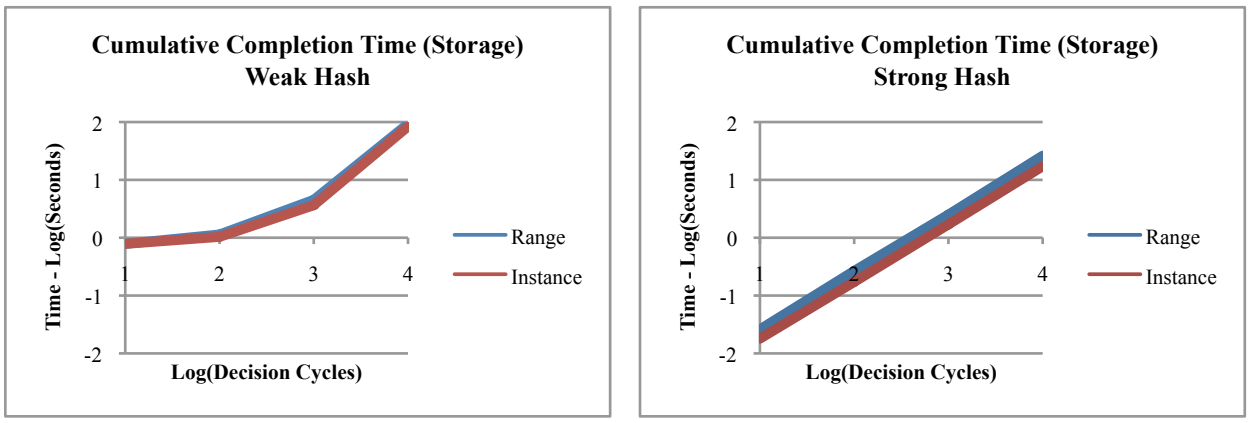

Figure 5: Phase 2

4.3. Phase 2: Improving WME Hash

As a first test of our initial Soar-EpMem implementation, we re-ran the *storage* agent from Phase 1. At this point, agent completion time was quite long, so we only collected timing data at 10, 100, 1k, and 10k decision cycles. When we graphed the logarithm of agent completion time against the logarithm of decision cycles (analogous to the number of stored episodes), we found a quadratic increase in both the Instance and Range indexing methods (see Figure 5, left panel). This growth rate equated to a linear increase in the time required to store an episode, relative to the number of previously observed episodes.

When exploring the reason for this growth characteristic, we found Nuxoll's implementation of WME attribute/value hashing (used extensively in episode storage) to be weak in the case of numerical WME values. The algorithm called for adding a proper string hash of attribute names to their respective numerical values. In the case of attributes having many closely valued pairs (such as in the case of the special "clock" WME), the summed hash values would be very close. We hypothesized that in context of the *ids* relation (see Appendix A), used in both

Instance and Range methods, a weak hash function would cause disproportionate growth of the b-trees and thus an increased need for rebalancing. As support for our intuition, when we profiled the running agent we found approximately 26% of process time was dedicated to maintaining SQLite b-trees.

To test this hypothesis, we implemented an improved hash function that afforded numerically similar values greatly different hash values. Re-running the initial experiment with this improvement yielded constant episodic storage time, and thus linear cumulative experiment completion time (see Figure 5, right panel). In a later section we will provide a comparison between our implementation and the Nuxoll baseline (see section 4.6).

4.4. Phase 3: Improving Range Update Efficiency

An implementation result of Nuxoll's work was the Range indexing method, which boasted both reduced memory requirements and improved access performance [9]. However, we observed greater storage times for the Range approach than the Instance approach (see Figure 5, right panel).

Our hypothesis for the source of this behavior had to do with our implementation of the Range indexing method. During storage, ranges representing WMEs that had not been removed from Soar's working memory were updated to reflect the current temporal id as the conclusion of the range. We reasoned that this update (including the associated b-tree adjustments) accounted for the relatively large performance loss.

To test our hypothesis we appealed to a core mantra of Soar: only update when necessary. We modified the Rete matching algorithm within Soar to flag WMEs as they were added to/removed from working memory [2]. We also modified our usage of the *episodes* relation (see Appendix A) to store NULL for open-ended ranges. Our new episode storage algorithm for the Range method would only replace NULL values when a WME had been removed from working memory. Additionally, we added some WME caching mechanisms within the Rete to benefit both Instance and Range methods. Thus, when a WME stayed in working memory, no processing had to be performed to update the ranges. Only when a new WME was created or when an existing WME was removed would there be any processing.

To evaluate our changes, we re-ran the *storage* agent on Range and Instance methods for 1000 decision cycles and measured time to agent completion (see Figure 6). The results showed a noticeable improvement from the caching mechanisms (as indicated by the difference between the Instance and Instance-Change results), as well as a shift in relative performance between Range and Instance (as indicated by Range-Change taking relatively less time than Instance-Change). This data validated Nuxoll's findings regarding storage time of the Range approach [9].

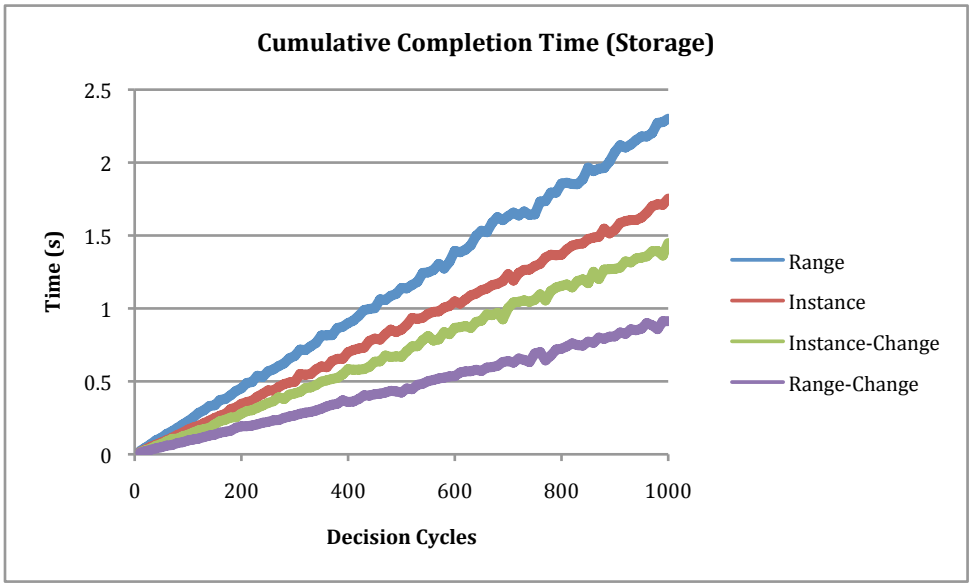

Figure 6: Phase 3

4.5. Phase 4: Improving Range Non-Cue-Based (NCB) Retrieval

Having demonstrated constant time storage using both Instance and Range methods, we proceeded to test NCB retrieval. In this episodic operation, an agent specifies an episode to be retrieved by temporal id. The episodic memory system identifies WMEs contributing to this episode and installs them in Soar's working memory. For the Instance method, this operation involves traversing an SQLite b-tree associating a temporal id (index key) with a set of WMEs (leaf nodes). For the Range method, episode reconstruction is a range intersection query: WMEs associated with ranges that intersect the temporal id contribute to the episode.

We developed a new *install* agent that extended the *storage* agent. This agent stored episodes every decision cycle and performed an NCB retrieval every other decision cycle. We used three different settings of the agent: *first* always retrieved the first stored episode, *1000* always retrieved the 1000th episode, and *recent* always retrieved the most recently stored episode. Running these settings with the Range indexing method for relatively long runs exposed a disturbing trend: NCB retrieval time was dependent (and increased linearly with) the episode chosen for retrieval (see Figure 7). This linear growth rate is most clear in the cumulative results of the "Range Recent" data set. For comparison, all Instance runs showed constant time retrievals, irrespective of the chosen episode.

As mentioned previously, identifying WMEs contributing to an episode using the Range indexing method is a range intersection query. We had implemented b-tree indexes keyed on WME, range start value, and range end value (in this order). However, examining the reconstruction query plan revealed that while the b-tree reduced finding valid start values to constant time (in a static tree), checking valid end values (even with a sorted list) became a reverse linear scan of the b-tree. Earlier retrievals began earlier in the tree, hence their reduced cost. Since ranges were only added with greater start values, the computational cost of retrieving a fixed episode changed only with the height of the tree. We hypothesized that implementing an interval tree to represent the element ranges would reduce the linear growth to logarithmic, a much more palatable complexity profile [1].

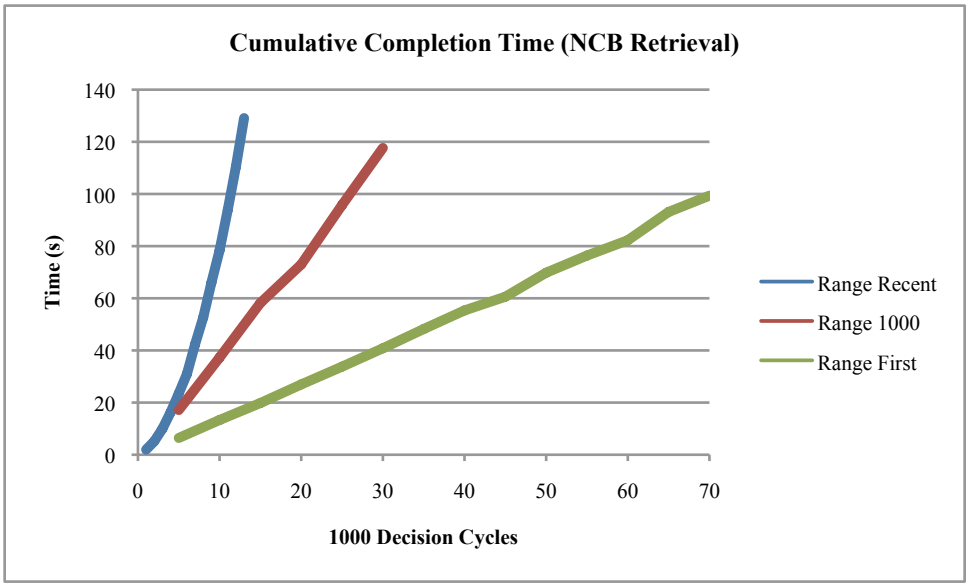

Figure 7: Phase 4 Observations

To test our hypothesis, we implemented the Relational Interval Tree algorithm, a mapping of the interval tree data structure onto relational schemas and b-tree indexes [5]. The algorithms described in section 3.3 for the Range method still applied. However, during storage, additional computation was required to compute a *node* value for each stored range. This *node* value was used to speed intersection range queries required to reconstruct an episode chosen for NCB or cue-based retrieval. Due to the modified relational schema, we termed this new indexing method RIT.

We re-ran our experiments and found RIT NCB retrieval time was independent of the episode to be retrieved (see Figure 8, left panel), growing logarithmically with the number of episodes (R^2=0.9988 in a logarithmic regression),

and memory consumption was still considerably less (about 266% in initial tests) than that of the Instance approach (see Figure 8, right panel).

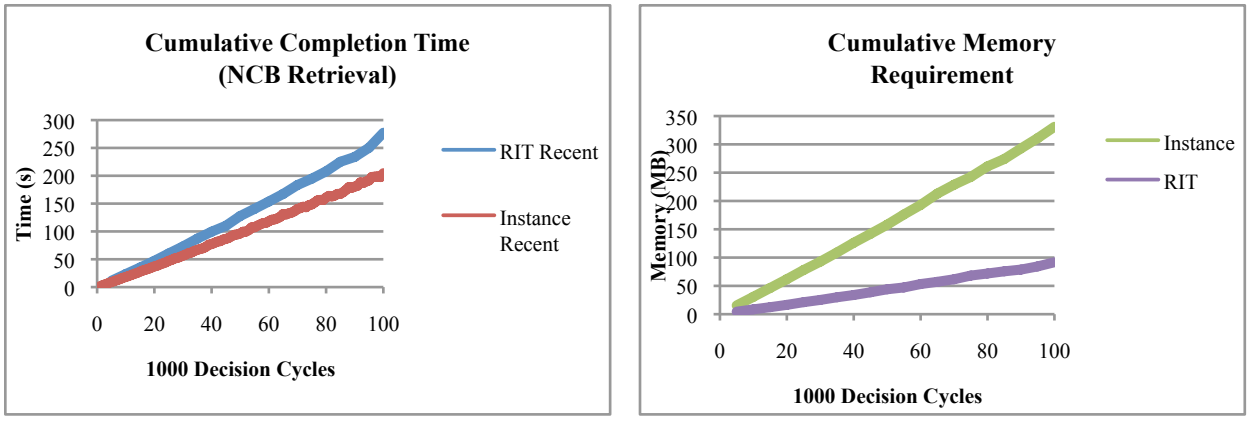

Figure 8: Phase 4 Results

4.6. Phase 5: Improving RIT Storage Time

After implementing the RIT indexing method, we re-ran the *storage* agent, comparing all three methods and found RIT storage time to be much greater than Range and Instance, though demonstrating the same growth characteristic. At this point, the storage operation for the RIT method was identical to Range, with added computation. Thus, we expected slightly reduced performance as compared to the Range method. However, we were not ready to accept that this added cost was greater than the computational savings achieved in representing episodes as temporal ranges rather than WME instances.

In the RIT paper, Kriegal resolves integrating "now" ranges (open-ended ranges for elements that have not yet been removed from working memory) in the following way: ranges are initially added to the RIT with a special "now" end value; when the respective WME is removed from working memory, the now range is removed from the RIT; finally, a complete range is added back to the RIT [5]. We hypothesized that the computational cost of calculating RIT placement, combined with maintenance of multiple b-trees during the insertion/deletion processes, was responsible for significant expense during episodic storage.

In order to test our hypothesis, we added a *now* relation (see Appendix A) to represent "now" ranges. New WMEs were initially added to this relation. When removed from working memory, they were removed from the *now* relation, and only then incurred the computational cost of insertion in the RIT *episodes* relation. We implemented this change only in the RIT method. We reasoned that NCB retrieval cost of the Range method (see section 4.5) would dominate any improvements in storage. Modifications to range storage make no impact on the Instance method.

After implementing the improved RIT algorithm, we re-ran the *storage* agent, comparing all three methods. As expected, RIT storage time improved dramatically, performing better than Instance or Range methods (see Figure 9). It should be noted that different random seeds were used for each experimental length (10k, 20k, etc) in this data set. Consequently, comparisons cannot be drawn for a single indexing method between different numbers of decision cycles. This explains the apparent reduced time requirement between 80,000 and 90,000 decision cycles.

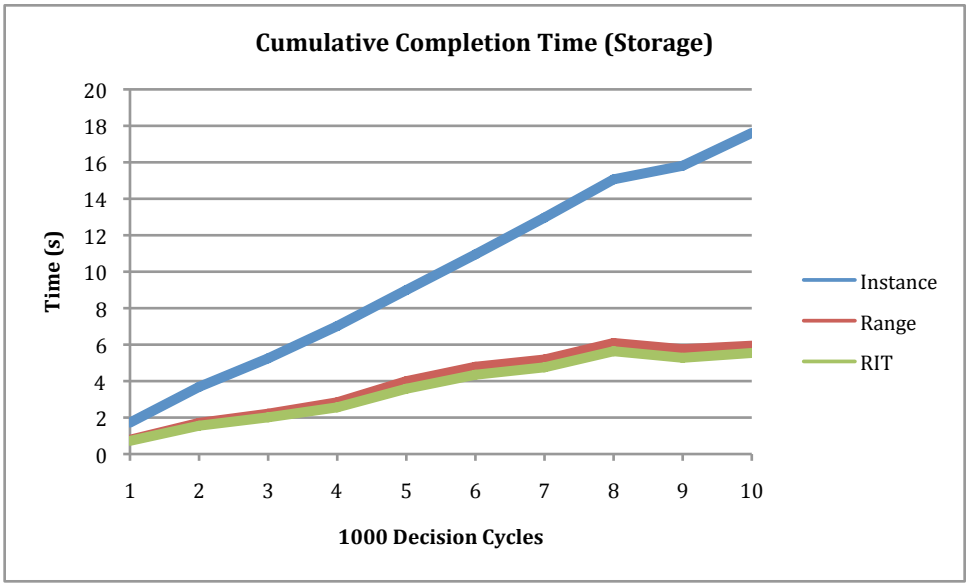

Figure 9: Phase 5

Upon receiving these results we integrated Soar Working Memory Activation (WMA) as the nearest-neighbor feature weighting value in all three indexing methods, as per Nuxoll's thesis. With a fully functioning system, we compared the storage operation of our Instance and RIT methods with Nuxoll's baseline implementation (see Figure 10). As a reminder, holes in the Nuxoll data represent crashes during the run. In addition to a very different growth profile, our RIT implementation ran reliably and about 24X faster than Nuxoll's baseline Instance approach (at 100,000 decision cycles).

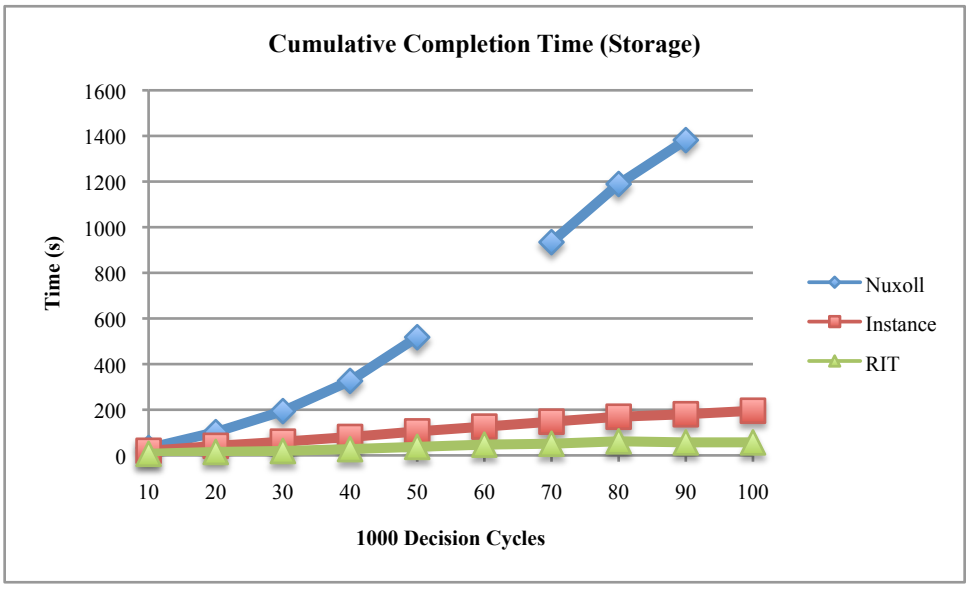

Figure 10: Comparison with Nuxoll Baseline

4.7. Phase 6: Development of a Hybrid Indexing Method

Having demonstrated performance advantages of the RIT indexing method, we recalled that the computational and memory overhead incurred by the RIT data structure was only paying off in the episodic NCB retrieval operation (not storage or cue-based retrieval). We were curious if we could adapt a hybrid approach, making use of the storage/query efficiency of the Range method with the NCB retrieval efficiency of the Instance method. Our

hypothesized Hybrid approach was inspired in part by the content vector introduced in MAC/FAC [3]. Our theory: we could cheaply implement a bit-vector representation of episodes that could be used for efficient NCB retrievals, while still maintaining the Range representation for use in cue-based retrievals.

Our bit-vector representation of an episode was a string of bytes, where each bit was valued 1 if the episodic element corresponding to that bit location was present in the episode. Generating this string would take time linear in the number of unique episodic elements. The NCB retrieval operation was implemented by identifying the positions containing 1's. We supplemented Wegner's efficient Hamming Weight algorithm to supply this information, which resulted in a method whose time was linear in the number of WMEs comprising an episode [15].

We briefly investigated using cheap bit operations for cue-based retrievals, but while the operations on a single episode were quite inexpensive, the asymptotic growth would always require operations linear in the size of the episodic store. Thus we supplemented the bit-vector representation with a "lean" Range representation, removing superfluous b-trees needed for NCB retrieval.

Of important note is that the storage requirements of the Hybrid approach depends greatly on the number of unique attribute-value pairs encountered over time. In our testing domain, this value would be reasonable if the "random" and "clock" special WMEs were excluded from episodic storage, as their values changed every decision cycle. Note that these values were never tested in any of the cognitive capabilities demonstrated by Nuxoll and are thus functionally superfluous. While the ability to exclude values is available to the agent designer at run-time, we recognize that this Hybrid implementation would not scale well to long-living agents, experiencing rich environments and/or multiple tasks, where unique attribute-value pairs are likely to increase greatly over time (although it might be possible for the system to dynamically identify such pairs itself – a topic for future research).

We ran *storage* and *install* agents, comparing Hybrid and RIT indexing methods (see Figure 11). We found Hybrid to be at least as fast as RIT and, when "random" and "clock" elements were removed (denoted as "-RC" data sets), memory requirements were reduced about 40% as compared to RIT. Note that as in Figure 9, different random seeds were used for each experimental length (10k, 20k, etc) in this data set. Consequently, comparisons cannot be drawn for a single indexing method between different numbers of decision cycles.

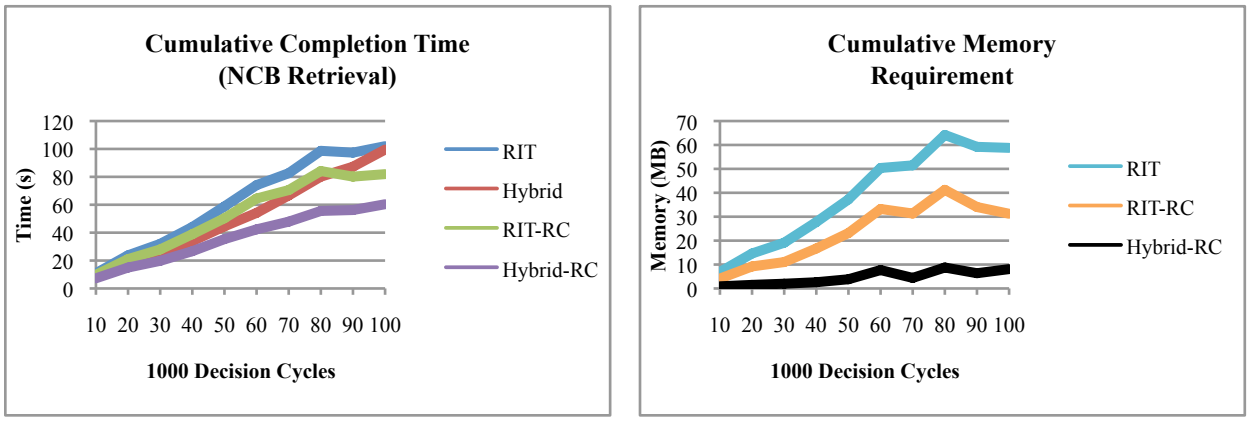

Figure 11: Phase 6

4.8. Phase 7: Improving Memory Usage and Cue-Based Retrieval

As described in Section 4.1, a large proportion of the WMEs in our testing domain are present for only a single episode. As a result, we hypothesized that implementing a separate relation for "point" ranges (start and end values being equal) would reduce memory requirements. Also, while our functional specification biases towards episode recency, our range algorithm for cue-based retrieval (used in Range, RIT, and Hybrid methods) was searching in the direction of increasing time. We hypothesized that reversing the search direction of the algorithm would increase cue-based retrieval speed for retrievals of recently stored episodes while maintaining an identical worst-case growth characteristic.

We implemented the proposed changes and proceeded to evaluation. For point ranges, we re-ran our *storage* agent and found that while time requirements were about equal for all major operations, RIT memory requirements were reduced about 25% (see Figure 12, left panel; note again the use of different random seeds). For the reverse range query algorithm, we implemented a new *query* agent that performed cue-based retrievals for an element that would always be found in the most recent episode. We found that cue-based retrieval performance increased about 20-25% for both the Hybrid and RIT indexing methods (see Figure 12, right panel). In this graph, note that RIT and Hybrid methods, as well as their respective reversed counterparts, display nearly identical results at these experiment lengths. This significant overlap reflects the degree to which performance of the range query algorithm dominates operation time.

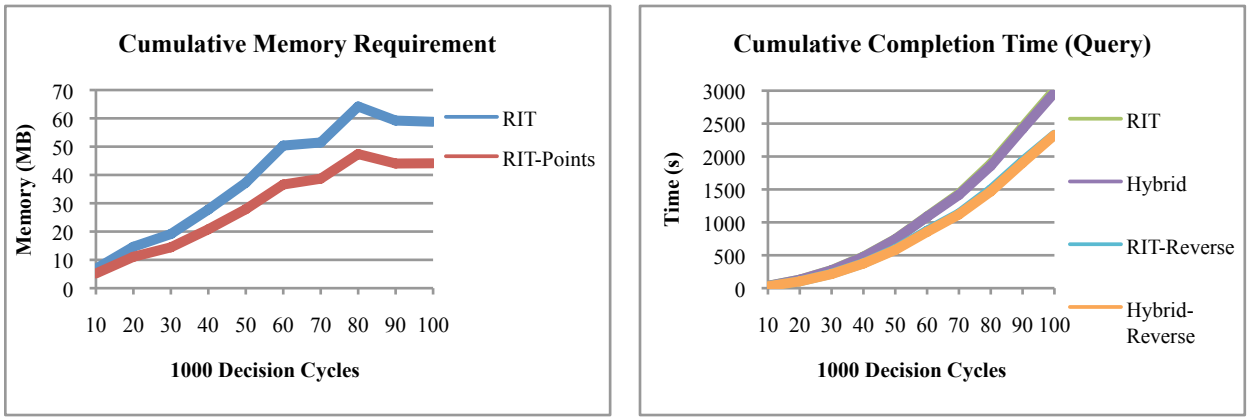

Figure 12: Phase 7

4.9. Our Final Implementation Results

In this section we provide representative time and memory data sets from our final Soar-EpMem implementation. Experiments were averaged over 5 trials and were run on a 2.8GHz Core 2 Duo iMac (4GB RAM) running Mac OS X 10.5.4 (with similar results obtained on comparable Linux and Windows hardware).

This data set was collected using a variation on the *query* agent. The agent stored episodes every decision cycle and issued a cue-based retrieval request at a fixed frequency (once every 1000 decision cycles). The cue, intended to represent the current location, consisted of three WMEs: current x coordinate, current y coordinate, and cardinal direction in which the tank was facing. The coordinate values ranged from 1-13, while direction was either "north," "south," "east," or "west." In a single agent experiment, one of the two direction coordinates changed if the agent executed a "move" command. If the agent issued a "rotate" command (typically at corners), the direction changed.

The optimal retrieval for the cue used in these data sets was always the most recent episode. We recognized that this property of the experimental setup was likely to affect the ratio of time requirement between the cue-based retrieval operation and storage/NCB retrieval. However, this constraint greatly simplified verification of the results and reduced data collection time. Additionally, initial tests (see Figure 12) suggested cue-based retrieval dominated overall agent completion time for even a single WME cue. Thus, we felt our improved ability to present a wide variety of verified, qualitatively correct data outweighed the quantitative cost of potentially more expressive cues.

Figure 13 compares time per Soar decision cycle (as opposed to cumulative agent completion time) between the indexing methods as number of stored episodes increases. Our data demonstrates a linear increase in cycle time for all indexing methods, with approximately a 190% greater growth constant in the Instance method as compared to all others. Because the cue-based retrieval operation dominates cycle time, and both RIT and Hybrid make use of the same range query algorithm, the data for RIT, RIT-RC, and Hybrid-RC (depicted as the lower line in Figure 13) were nearly identical.

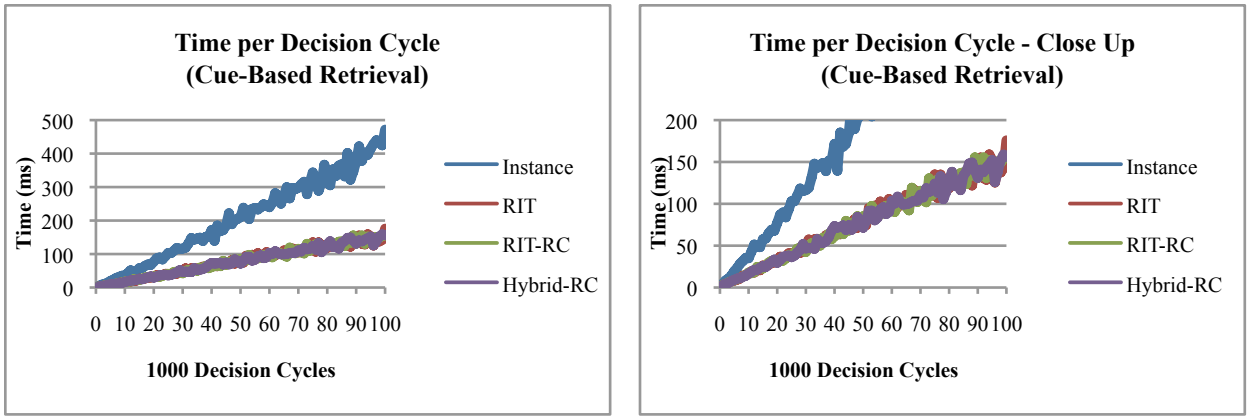

Figure 13: Final Indexing Method Time Comparison – Cue-Based Retrieval

Note that we purposefully excluded the Range method from this data set. The range query algorithm (shared by Range, RIT, and Hybrid) was well-represented in this data set. More importantly, episode reconstruction time in the Range method (as discussed in section 4.5) increases linearly with the number of observed episodes. For comparison, episode reconstruction alone at 100,000 decision cycles in the Range method consumes over 400ms (much greater than the entire cue-based retrieval operation of RIT and Hybrid). This computational cost makes the Range method unusable for any reasonable episodic memory agent and is thus not shown.

Figure 14 compares cumulative memory consumption of the episodic store between the indexing methods as number of stored episodes increases. Our data shows a linear increase in the memory requirement of all indexing methods, with a 740% greater growth constant in the Instance method as compared to RIT. We notice also that when the "random" and "clock" WMEs are excluded from episodic storage (as indicated by the "-RC" data sets), the linear growth constant of RIT is within 40% of Hybrid (adding only about 9MB at 100k decision cycles). This data set, combined with the cue-based retrieval comparison above, validates the RIT indexing method as an efficient, general-purpose approach (while Hybrid requires WME exclusions). Thus, we have opted to focus our remaining data sets on the RIT indexing method.

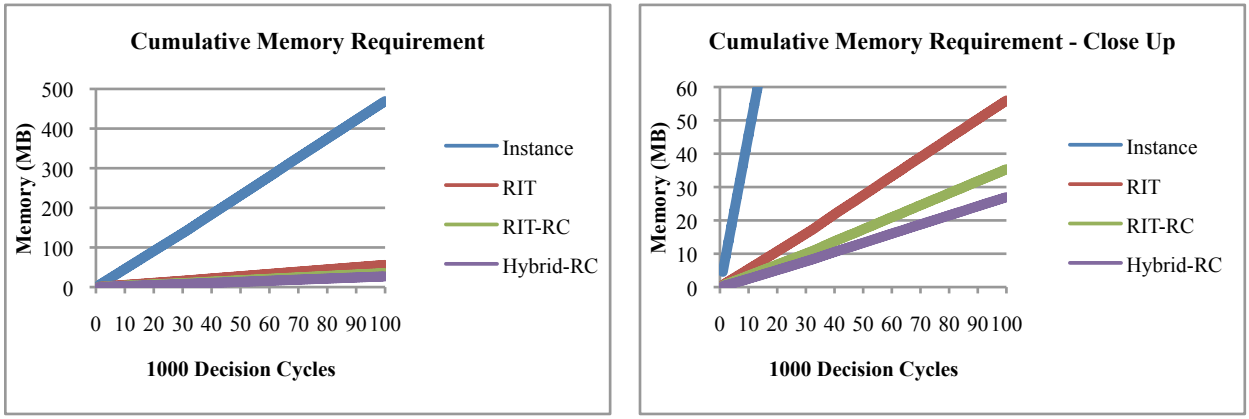

Figure 14: Final Indexing Method Memory Comparison

Figure 15 compares time per Soar decision cycle for the core episodic operations using the RIT indexing method. The "Baseline" data represents running the *simple-bot* agent with the episodic memory components of Soar-EpMem disabled. The right panel provides a time scale such as to compare the storage (averaging 0.54 ms/cycle) and NCB retrieval (averaging 1.12 ms/cycle) operations with the baseline agent (averaging 0.15 ms/cycle). Under more constrained time scales, these operations display logarithmic growth with respect to the number of stored episodes, as expected. The cue-based retrieval (about 175 ms at 100k cycles) grows linearly with the number of stored episodes and clearly dominates all other operations, as demonstrated in the left panel. This data coincides with

Figure 13, wherein we observed nearly identical agent completion time for RIT and Hybrid methods, irrespective of WME exclusions.

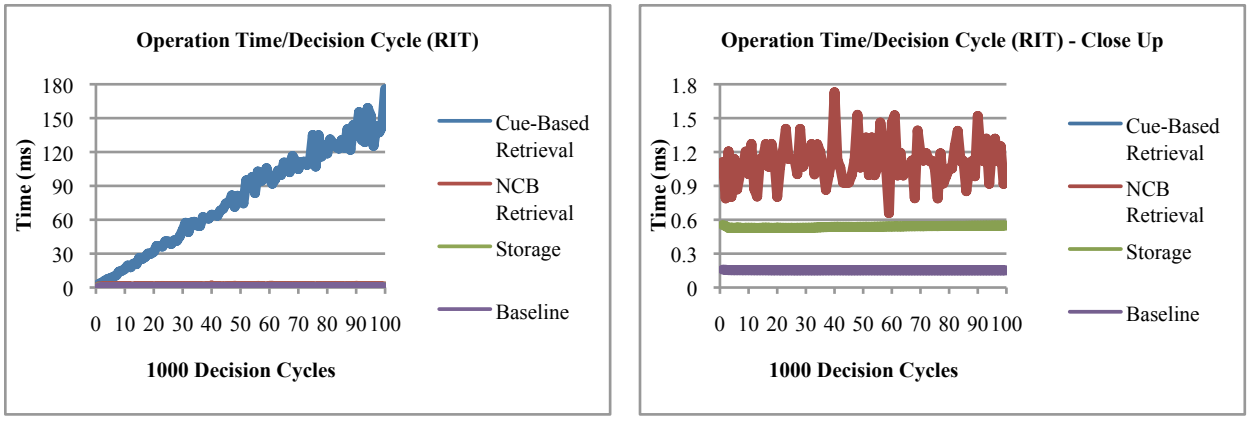

Figure 15: Final Operation Time Comparison - RIT

To provide context for these results, we considered some long-term, real-world agent scenarios. Running an agent in cognitive real-time (50 ms/decision), we predict storing a new episode approximately once each second (20 times less frequent than in our experiments). Note that Soar runs thousands of times faster than this, but for real-world applications we have found the need for only about 10-20 decisions per second. Based upon our experience with agents using episodic memory, we estimate that cue-based retrieval operations must complete in about 250ms to be considered sufficiently reactive. Under these conditions, using the RIT method and retrieval cues similar to the one described above, we could run an agent for about 164,000 seconds (3.28 million decision cycles) at a cost of about 95MB of memory. This corresponds to 1.9 days of non-stop execution, making our episodic memory implementation feasible for real-world, long-living agents. There is a completely separate research question as to whether this is the best use of this computation (as opposed to using that time for some other types of processing); however, because in theory episodic memory can run independently of other processing on a single dedicated core of a multi-core processor, that question is less pressing than if episodic memory retrievals precluded other processing.

5. Discussion

This research project emphasized performance of storage and NCB retrievals and only recently did it involve explorations of cue-based retrievals. Although our final results indicate that cue-based retrieval dominates other costs, our current enhancements have led to a stable and efficient artificial episodic memory system integrated with Soar 9. We have contributed a robust system and baseline performance evaluations, which will facilitate future research into the effective development and use of episodic memory systems.

In examining the context of our final implementation results (see the paragraph concluding section 4.9), we acknowledge the problem of utility as it relates to efficiency of episodic retrievals. With respect to a particular problem, while we expect solution quality to increase with the number of episodes to which an agent has access, we recognize that assessment of learning efficiency takes into account time to solution. Analysis of the utility problem with respect to CBR has demonstrated a "saturation point" in utility, after which increased solution quality no longer compensates for increased agent deliberation time [11]. We expect a similar result to materialize when assessing episodic learning efficiency within a single domain.

To strive towards optimal episodic learning efficiency, we first need to extend Nuxoll's work into the meaning of episodic retrieval quality, ideally in multiple domains [9]. Fundamentally, we need to better understand and develop metrics for what constitutes an acceptable/desirable episodic retrieval in a particular situation, and how this relates to properties of the episodic cue.

Armed with this knowledge, we will then pursue an in-depth study of methods for efficient cue-based retrievals. For instance, we could adopt a two-stage match process (ala MAC/FAC) that, while potentially producing non-optimal

cue-based retrievals, may yield significant query performance gains [3]. One such second level filter we have labeled "provenance": an in-depth, structural exploration of episodes with respect to multi-valued attributes [12]. Another filter may involve applying DBMS query optimization techniques to efficiently find exact-matches using episode statistics, with lesser matches considered based upon relative working memory activation (WMA) values. We have also considered a two-tier memory system: fast access to an episodic cache paired with parallel, slower access to a complete store. Other development extensions include embedding WMA values in Range-based indexing methods and updating stored WMA values based upon retrievals.

We also plan to investigate how properties of the target domain impact efficiency of episodic retrievals. For instance, we have found that the efficiency of our current indexing methods depends upon the number of unique attribute-value pairs the agent encounters, as well as the rate at which WMEs are added to/removed from working memory. If we are able to identify other environmental properties, as well as predictably quantify their effects upon retrieval time, we could potentially enable Soar-EpMem to intelligently tune properties of episodic storage in real-time, such as to improve retrieval efficiency.

Finally, we intend to explore Soar-EpMem implementations of the cognitive capabilities Nuxoll has identified [9]. An ideal result would be general, reusable libraries/rule templates for achieving benefits from the episodic memory system in a variety of situations. We are also excited by the prospect of comparison/integration studies with other learning mechanisms in the Soar cognitive architecture, such as semantic memory and reinforcement learning.

References

[1] Cormen, T., Leiserson, C., Rivest, R., Stein, C. (2003). Introduction to Algorithms, 2nd ed. London: MIT Press.

[2] Forgy, C. (1991). Rete: a fast algorithm for the many pattern/many object pattern match problem. Expert systems: a software methodology for modern applications. IEEE Computer Society Press, Los Alamitos, CA.

[3] Gentner, D. and Forbus, K. (1991). MAC/FAC: A model of similarity-based retrieval. Proc. of Cognitive Science Society.

[4] Kolodner, J. (1992). An Introduction to Case-Based Reasoning. Artificial Intelligence Review.

[5] Kriegel, H., Potke, M., Seidl, T. (2000). Managing Intervals Efficiently in Object-Relational Databases. Proc. 26th Int. Conf. on Very Large Databases (VLDB).

[6] Laird, J. (2008). Extending the Soar Cognitive Architecture. Artificial General Intelligence Conference.

[7] Laird, J. and Rosenbloom, P. (1996). The Evolution of the Soar Cognitive Architecture. Mind Matters, T. Mitchell (Ed.), 1-50.

[8] Nuxoll, A. (2007). Enhancing Intelligent Agents with Episodic Memory. PhD Dissertation, University of Michigan.

[9] Nuxoll, A. and Laird, J. (2007). Extending Cognitive Architecture with Episodic Memory. Proc. 22nd National Conference on Artificial Intelligence (AAAI).

[10] Nuxoll, A., Laird, J., James, M. (2004). Comprehensive Working Memory Activation in Soar. International Conference on Cognitive Modeling (ICCM), Poster.

[11] Smyth, B. and Cunningham, P. (1996). The Utility Problem Analysed: A Cased-Based Reasoning Perspective. Third European Workshop on Case-Based Reasoning.

[12] Sun, R. (2008). The Cambridge Handbook of Computational Psychology. Cambridge University Press.

[13] Tecuci, D. and Porter, B. (2007). A Generic Memory Module for Events. Proc. FLAIRS20 Conference.

[14] Tulving, E. (1983). Elements of Episodic Memory. Oxford: Clarendon Press.

[15] Wegner, P. (1960). A technique for counting ones in a binary computer. CACM 3.

Appendix A - Relational Schemas

Final relational schemas below are grouped by indexing method and then relation name.

A1) Instance

ids (WME name/value registration)

- child_id (integer, primary key) - element id
- parent_id (integer) - element id of parent
- name (text) - WME name
- value - WME value, NULL for identifiers
- hash (integer) - hash of the name/value pair

episodes (Episode element instances)

- id (integer) - element id, relates to ids (child_id)
- time (integer) - temporal id
- weight (real) -WMA, NULL for identifiers

A2) Range

ids (WME name/value registration)

- child_id (integer, primary key) - element id
- parent_id (integer) - element id of parent
- name (text) - WME name
- value - WME value, NULL for identifiers
- hash (integer) - hash of the name/value pair

times (Efficient lookup of valid temporal ids)

- id (integer, primary key) - valid temporal id

episodes (Registry of valid episode ranges)

- id (integer) - element id, relates to ids (child_id)
- start (integer) - temporal id when element instance started
- end (integer) - temporal id when element instance ended, NULL for current ranges

weights (Temporary look-up table of element weights during query transactions)

- id (integer, primary key) - element id, relates to ids (child_id)
- weight (real) - WMA value

ranges (Temporary list of query ranges during query transactions)

- start (integer) - range start
- end (integer) - range end
- weight (real) - range sum WMA
- ct (integer) - range sum cardinality

A3) RIT

ids (WME name/value registration)

- child_id (integer, primary key) - element id
- parent_id (integer) - element id of parent
- name (text) - WME name
- value - WME value, NULL for identifiers
- hash (integer) - hash of the name/value pair

times (Efficient lookup of valid temporal ids)

- id (integer, primary key) - valid temporal id

now (Registry of "now" element ranges)

- id (integer, primary key) - element id, relates to ids (child_id)
- start (integer) - temporal id when element instance started

points (Registry of completed valid element ranges lasting one decision cycle)

- id (integer) - element id, relates to ids (child_id)
- start (integer) - temporal id when element instance started/ended

episodes (Registry of completed valid element ranges)

- id (integer) - element id, relates to ids (child_id)
- start (integer) - temporal id when element instance started
- end (integer) - temporal id when element instance ended
- node (integer) - RIT node value, expedites intersection searches

left_nodes (RIT left table, temporary)

- min (integer)
- max (integer)

right_nodes (RIT right table, temporary)

- node (integer)

weights (Temporary look-up table of element weights during query transactions)

- id (integer, primary key) - element id, relates to ids (child_id)
- weight (real) - WMA value

A4) Hybrid

ids (WME name/value registration)

- child_id (integer, primary key) - element id
- parent_id (integer) - element id of parent
- name (text) - WME name
- value - WME value, NULL for identifiers
- hash (integer) - hash of the name/value pair

episodes (Episode bit-vector storage)

- time (integer, primary key) - temporal id
- ids (blob) - bit-vector where each bit position (mod 8) corresponds to ids (child_id)

now (Registry of "now" element ranges)

- id (integer, primary key) - element id, relates to ids (child_id)
- start (integer) - temporal id when element instance started

points (Registry of completed valid element ranges lasting one decision cycle)

- id (integer) - element id, relates to ids (child_id)
- start (integer) - temporal id when element instance started/ended

nodes (Registry of completed valid element ranges)

- id (integer) - element id, relates to ids (child_id)
- start (integer) - temporal id when element instance started
- end (integer) - temporal id when element instance ended

weights (Temporary look-up table of element weights during query transactions)

- id (integer, primary key) - element id, relates to ids (child_id)
- weight (real) - WMA value