Proceedings of the Twenty-Ninth International
Florida Artificial Intelligence Research Society Conference

Hash Functions for Episodic Recognition and Retrieval

Emilia Vanderwerf, Robert Stiles
Alexandra Warlen, Allison Seibert,
Kevin Bastien, Andrew Meyer, Andrew M. Nuxoll
University of Portland
5000 N. Willamette Blvd.
Portland, OR 97203

Scott A. Wallace
Washington State University Vancouver
14204 NE Salmon Creek Ave.
Vancouver, WA 98686

Abstract

Episodic memory systems for artificially intelligent agents must cope with an ever-growing episodic memory store. This paper presents an approach for minimizing the size of the store by using specialized hash functions to convert each memory into a relatively short binary code. A set of desiderata for such hash functions are presented including locale sensitivity and reversibility. The paper then introduces multiple approaches for such functions and compares their effectiveness.

Introduction

Episodic memory is a commonly identified type of human long-term memory (Tulving 1983). It is distinguished from other types of memory in that episodic memories record specific events in the past such as where you parked your car or the last time you rode on a ferris wheel. A human with a damaged episodic memory, an amnesiac, has impaired cognitive ability. This implies that episodic memories improve a human's ability to reason and learn and, thus, such a memory can be valuable to an artificially intelligent agent as well (Nuxoll and Laird 2012).

A healthy human brain has the capacity to store a lifetime of memories of events as they occur. Therefore, it is reasonable to presume that creating an artificial episodic memory for an intelligent agent will require the ability to record and recall a large sequence of events without prior knowledge of their content or their relevance to future tasks.

Logically, the ability to record a sequence of episodic memories (episodes) in a relatively small, finite memory store requires that the episodic memory system make choices about what elements are retained and what elements are forgotten. Forgetting might take many forms (discussed in Related Work below) including: not recording some events, decay of memory elements, combining or merging similar memories, or a gradual reduction of memory fidelity.

Wallace et al. (2013) explored the idea of minimizing the size of the episodic store by using it for recognition only rather than recall. Specifically, a hash function was used to generate a unique id for each episode. The id was stored but the episode was then discarded entirely. Thus, the agent could recognize an episode if seen again by comparing its hash value to those previously seen.

In effect, this approach compresses the content of each episode to a single integer. It can be thought of as being at one extreme on the continuum between greater fidelity of individual episodes versus greater capacity for recording episodes given a fixed-size episodic store. Whereas, most forgetting mechanisms are at or near the opposite end.

Wallace et al. (2013) demonstrated that a recognition-only episodic memory system can still be quite effective for some tasks. However, their work does not explore the *choice* of the hash function itself. Rather, their choice of hash function was essentially arbitrary. In this research, we investigate hash functions that provide particularly useful capabilities: namely the ability to recognize similar episodes and also to perform a limited degree of episode retrieval.

We begin with a set of desired properties of an episodic memory hash function and then describe a general approach for such functions called a hash formula. We then describe and compare several general approaches to creating and maintaining a hash formula over time as the agent perceives subsequent episodes. We conclude with a comparison of these approaches within the context of the original desiderata. These results show that our best approach is comparably effective to a hash formula that has prior knowledge of the episodic memory content.

Episode Hash Function Desiderata

Typically, a hash function has the following requirements:

Efficient A hash function should execute in $O(n)$ time or better where n is the size of the input.

Deterministic A hash function should generate the same hash code given the same input.

Uniform given a set of inputs, the function should produce a small number of collisions.

We propose that a hash function used to encode episodic memories has the following additional desired properties:

Reversible Given a hash code and the definition of a hash function, it is possible to partially reconstruct the episode from the code. Presumably, a full reconstruction is impossible given the large difference in information stored

in the episode as compared to the code. However, a partial reconstruction may be sufficient to enhance an agent's decision making. The goal is to retain as much information as possible within the hash code.

Locality Sensitive In short, similar episodes encode to similar hash codes. This allows the agent to gauge the relative familiarity of an episode even if it is unique. A measure of the familiarity of an episode can be used to guide learning, particularly if it is used to extract a past episode that is most similar to its current state.

Related Work

The simplest and most obvious approach to maintain a fixed size memory store is to simply discard episodes from the store to make room for new ones. Researchers have discovered several heuristics for selecting which episode to discard in this situation (Ram and Santamara 1997; Kennedy and De Jong 2003; Dodd and Gutierrez 2005; Brom and Lukavsk 2009; Nuxoll et al. 2010). In contrast, our approach is to show how episodes can be substantially compressed using hashcodes while still preserving an agent's ability to recognize situations and partially reconstruct the past.

Thus, the research we present in this paper can be viewed as either an alternative or a complement to the forgetting mechanisms listed above.

Bloom filters (Bloom 1970) are an effective way to recognize previously seen inputs. Rather than using one hash function to generate a bit string for each input, a Bloom filter uses one very large bit string and many independent hash functions. The bit string begins at all zeroes and each hash function sets one bit in the string. Thus, if there are n hash functions, n bits are set each time you hash a new input. To recognize a previously seen input, the program can simply check all bits that would be set by hashing it.

However, to use Bloom filters for recall as well as recognition would require that the hash functions be reversible. Furthermore, a Bloom Filter requires at least a limited foreknowledge of the scope of data it is recording since the size of the bit string typically cannot be easily changed once it is created (Almeida et al. 2007).

Locality-sensitive hash (LSH) functions were initially introduced in (Indyk and Motwani 1998). LSH functions rely upon transforming the input into N different projections. Inputs are then hashed into N sub-hashcodes (often used to index separate tables) that are each $1/N$ the size of the actual desired hash code length. The resulting sub-hashcodes are concatenated to form the final code. In this way similar inputs will have at least one subcode that is identical.

LSH functions can also be used in combination with Bloom filters to create a data store that recognizes similar inputs (Hua et al. 2012).

In order to create effective projections of each input, a LSH presumes some foreknowledge of the general format and content of the inputs in order to define the projections. Particularly, the number of dimensions of the input must be known in advance. For many LSH hash functions, the domain of each dimension must also be known.

These requirements are infeasible for a strictly general purpose episodic memory for which the scope and content of the episodes is not known in advance. Also, as with Bloom filters, to provide recall the LSH functions used would need to be reversible.

The idea of a reversible hash function is essentially lossy data compression. Existing lossy data compression techniques are primarily focused on audio or image data and also rely upon some foreknowledge of the content and format of the data to be compressed (Cover and Thomas 2006). It is not clear how such techniques could be applied to arbitrary episodic memories.

Hash Formulae

To create an effective hash function with the desired properties we use a simple approach we refer to as a hash formula. A hash formula is a mapping of atomic elements of an episode to individual bits in a hash code. If a given element is present in an episode, then the corresponding bit is set.

For example, consider a case where each episode is a phrase consisting of one or more English words. The text below depicts a hash formula that maps English words to bits. To keep this example simple, the hash code has only 5 bits.

```
0: down
1: never
2: desert
3: let
4: run
```

For a given input episode, say "never gonna let you down," the hash formula can be used to produce a hash code by setting the bits corresponding to matching words in the input. In this example, the resulting hash code would be 11010. A subsequent episode, say "never gonna run around," would hash as 01001. It's notable that this example approach could be viewed as a lossy version of the bag of words model (Harris 1954).

Baseline Hash Function

To provide some insight into the effectiveness of a hash function, it's useful to have a "good" hash function to compare it to. We use a genetic algorithm (Mitchell 1996) to select an effective set of elements from all episodes in a given test database.

A hash formula generated in this way is not useful for an episodic memory system since it requires access to the entire database *a priori*. Nonetheless, it does provide a valuable basis for comparison for other hashing algorithms as intended. We use the baseline for this purpose.

Online Hash Formula Definition

A key consideration when constructing a hash formula is selecting the correct elements to use.

We hypothesize that the most useful elements for a hash formula are those that are neither particularly rare nor particularly ubiquitous. This hypothesis is based on information theory and the fact that the information content of a

boolean random variable, such as we consider here, is maximized when the variable is equally likely to take either of its possible values. There are two important corollaries of this property: first, selecting elements with high information content also means that we preserve bits in the hashcode for elements whose presence (or absence) cannot easily be predicted; and second, this method should also help ensure the uniformity requirement discussed above.

This hypothesis is also consistent with the term frequency-inverse document frequency (tf-idf) statistic used in information retrieval (Jones 1972).

However, by definition, the systems we are interested in exploring are not privy to the content of all the episodes to be hashed. Instead, they are only aware of the current episode and, possibly, limited information about previous episodes. As a result of these limitations, there is no guarantee that a hash-formula-based approach yields a useful hash function if that hash formula is defined *a priori*. In the extreme case, if none of the words in the hash formula appear in any of the input episodes, then all the hash codes will be identical.

To address this problem, we explore the impact of hash formulae that are defined, and modified, online. As new episodes arrive, the current hash formula is adjusted to reflect the episodic elements the agent has seen so far. When the hash formula is modified there is a danger that a previous episode might generate a different hash code if it were to be rehashed after the modification. Clearly, this approach exposes a tradeoff between the desire for uniformity (under the constraint of limited *a priori* knowledge) and the desire for determinism (as defined above). Our experiments provide an initial attempt to explore this tradeoff in the context of episodic memory retrieval.

We have identified three general approaches to implementing online hash formula definition: folding, sampling and frequency-based selection. These are each described below.

Folding

In traditional hash functions, the need for online hash function definition is solved by multiplying or combining all the elements in some manner. For example, hashing a string is often accomplished by multiplying each character together (usually with other factors) before truncating to appropriate hashcode length using a modulus or shift.

When using a hash formula (as defined in the previous section) a similar result can be achieved by associating multiple atomic elements of an episode with the same bit in the hash code. We call this approach a folding hash formula. Unlike other methods, this approach preserves (partially) the reversible nature of the hash formula.

A simple example is illustrated below along with some samples of how different episodes would be hashed.
Folding Hash Formula:

```
0: never, let, desert
1: gonna, down
2: give, run
3: you, around
4: up, or
```

Example Hashes:

```
never gonna give you up  11111
never gonna let you down 11010
never gonna run around   11110
or desert you            10011
```

In the above example, new episode features (i.e., individual words in the phrases) are simply added to the hash formula in order and rolled over to index 0 when the end is reached.

A folding hash formula also allows the episodic memory system to adapt to changing inputs. In effect, as new episodes arrive with novel elements, those novel elements can be added to the formula for future use without altering the hash codes that would result from previous inputs. Thus, the hash function retains the property of being online but also remains deterministic.

Hash formula folding introduces a new issue: the size of the hash formula definition grows without bound since new elements can be constantly added to it. In our experiments, we determined that we could keep the formula at a given fixed, maximum size by discarding elements that appeared the most rarely (breaking ties with those that occurred least recently). Doing this had a negligible impact on overall performance in our tests (in some cases it was even an improvement) but at the cost of reducing our hash functions' determinism.

Sampling

Another way to define a hash formula online is to attempt to select a representative sample of all episode elements seen to date. Specifically, each element has an equal chance of being selected for the formula.

To create a sample, we used a a simple reservoir sampling algorithm (Vitter 1985). The hash function must track the total number, N, of elements seen to date in all episodes. Each time a new element is encountered, there is a $1/N$ chance that the element will replace one already selected for in the hash formula.

This approach has the advantage of requiring virtually no meta-data to be stored for use by the hash function. However, it results in a non-deterministic hash function. We discuss this issue further below.

Frequency-Based Selection

Another way to define a hash formula online is to select the elements for the formula based upon their frequency of appearance in the episodes seen to date. As discussed above, we hypothesize that a good hash formula will use elements that are neither ubiquitous nor rare.

For this research, we chose to model the elements from episodes as independent Bernoulli random variables. This convenient simplification is commonplace to Naive Bayes models and is known to yield useful results in many real-world situations, even when the independence assumption is clearly violated (Maron and Kuhns 1960).

In our baseline experiment, we observed that the genetic algorithm selected episodic elements in a fashion that was consistent with our hypothesis about useful elements and

with our independence assumption. Figure 1 shows the frequency with which various elements were used by each formula in the final population created by the genetic algorithm.

We tracked the frequency with which each episodic element appeared across all episodes. For the environment under investigation, this resulted in eleven frequency bins. In the figure, each bin is represented by the x-coordinate of a point on the graph. Within each bin, we averaged the number of times the episode element in that bin appeared in the final hash formula produced by the genetic algorithm. This value is represented as the y-value for each point in the figure. Since many elements are unique or nearly unique, most of the points in the graph lie near $x = 0$. Finally, we plotted the Entropy curve (red dashed line) for a Bernoulli random variable scaled (here by a factor of 5) to appropriately fit the scatter plot. The line appears to predict our observations reasonably well.

The depicted results are drawn from the Eaters environment (described below). The same experiment yielded similar results in our other environments. None of these environments strictly adhere to the simplifying assumptions.

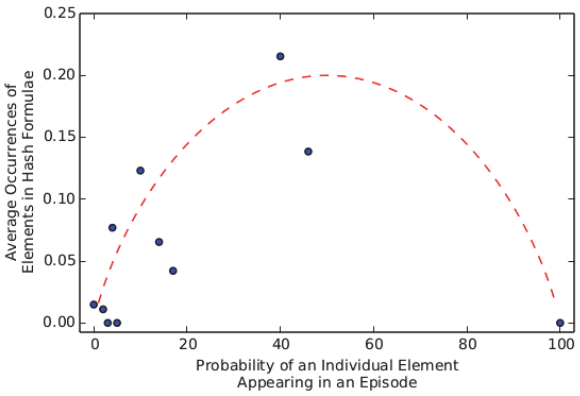

Figure 1: Frequency of Element Usage by Baseline

It is clear from the figure that elements that occur only once are disfavored. We further observed that ubiquitous elements were also disfavored but this is not clear from the figure since very few elements were this frequent and a relatively large number of elements were used to generate the graph.

As with folding, frequency-based selection requires the hash function to maintain meta-data: the list of elements and their occurrences. This list could grow unbounded. However, as with folding, we observed that we could keep this meta-data at a fixed size by discarding infrequent, not-recently-used elements.

Combining Techniques for Formula Definition

Using a sample-based or frequency-based element selection approach results in a hash function that is non-deterministic. A sampling-based approach deliberately replaces elements in the hash formula over time, albeit with decreasing frequency. For a frequency-based approach, an element that appears about 50% of the time after 10 episodes may only occur about 5% of the time after 100 episodes. Thus, a frequency-based approach might select a particular element for the hash formula at one point but reject it at some point in the future.

One way to address this concern is to ignore it. In many environments, we observed that new elements occur with decreasing frequency over time.

Another way to address the non-determinism implied by sample-based or frequency-based element selection is to combine the approach with folding. Since folding allows for an unlimited number of elements to be used in the formula definition, a sample-based or frequency-based solution can opt to never evict a particular element from its formula. Previous episodes will continue to hash identically using the updated formula since they lack the missing element while subsequent episodes can have hash codes that reflect the presence of this element.

Combining the two approaches also greatly reduces the growth rate of the hash formula compared to normal folding approach since only a fraction of the elements are selected.

Test Environments

To evaluate our hash functions, we used four different test environments in order to mitigate results that might be specific to any single environment. These environments are described briefly as follows:

Eaters Eaters is a Pac-Man-like game that is provided with the Soar cognitive architecture (Laird 2012). The agent operates in a two-dimensional grid world maze where each cell is initially occupied by a wall or food. The agent can perceive nearby cells and its only actions are movements in the cardinal directions.

An episode in the Eaters environment consists of a snapshot of the entirety of the agent's working memory which includes its sensors and effectors.

Tanksoar Tanksoar is another grid world based game that is provided with Soar. This environment is considerably more complex than Eaters with multiple sensors and actions. The agent must also balance resources and interact with competing agents.

An episode in the Tanksoar environment is also a snapshot of the agent's working memory. Due to the complexity of this environment, the size of the episodes was considerably more variable than for Eaters.

English Corpus In this environment, each episode consisted of subsequent sequences of ten words drawn in order from a public domain text. Individual words in the text were treated as atomic elements. This resulted in many more episodes than Eaters and Tanksoar but episodes were much smaller.

This environment had several interesting differences compared to the others. First, each successive episode was completely different from its predecessor whereas other environments tended to have great similarity between subsequent episodes. This environment was also most likely

to introduce new episode elements (i.e., novel words) at each step.

Video In this environment, each episode consists of one frame from a short video. Each episode was presented as a textual representation of each color component (red, green, blue) of each pixel in the frame. As a result, these episodes were much larger than for other environments.

Hash Function Evaluation

In our research, we tested several different hash functions and variations. For brevity, we present here a small sample of those that we feel are representative. Specifically, we present results for each of the hash formula definition techniques described above: folding, sampling and frequency-based. We also show the results from a hash function that uses a combination of frequency-based selection and folding as this proved to be one of the most effective in our tests. Additionally, the results are compared to our baseline agent which used genetic algorithm to preselect the best features.

As described above, we have defined six desiderata for an episodic memory hash function: efficient, deterministic, uniform, reversible, locality sensitive and online. In this section we evaluate how effective the hash functions we tested are at meeting these.

Efficient Since all of the hash functions we tested used a hash formula, generating a hash code consisted of a fixed amount of processing for each element in the episode to be hashed. Therefore, all our functions are efficient by the given definition presented earlier.

Deterministic Unlike a traditional hash function, a hash function used for encoding episodic memories has some room for being not perfectly deterministic. This is because an agent making an episodic memory is often looking for a best match rather than a perfect match.

To measure the degree of impairment from lack of determinism, we tested each hash function's ability to generate identical hash codes over time. Specifically, we hashed a sequence of unique episodes but regularly re-inserted episodes into the sequence that the agent had already seen. We then measured the degree of similarity (number of bits in common) between the hash code returned presently and the hash code returned previously.

Figure 2 shows the relative performance of each hash function on this task for a variety of hash code sizes. The baseline and folding hash functions are not shown as they are perfectly deterministic. The x-axis measures each of the hash code sizes we tested (10 to 200 bits in increments of 10). The y-axis measures the average degree of similarity between the new and previous hash codes for the same episode.

These results, drawn from our tests with the Eaters environment, show that the combination of folding and frequence-based selection has near-perfect performance by this metric compared to other approaches. Results from the other environments were comparable.

Uniform A hash function that is uniform tends to reduce the number of collisions. To measure this quantitatively

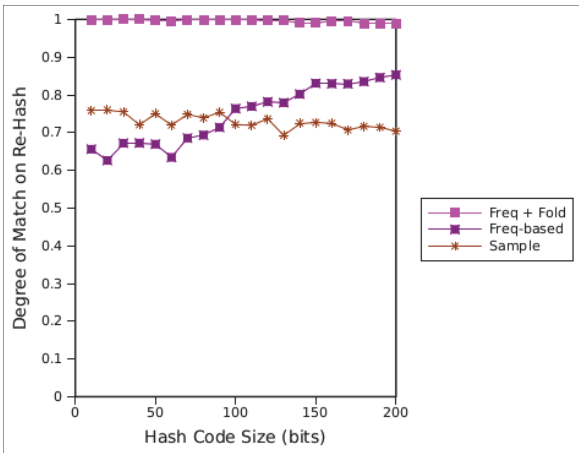

Figure 2: Determinism Comparison

we exposed the hash function to the same sequence of episodes as was used for our measurements of determinism and measured the number of collisions that occurred.

Figure 3 shows the relative performance of each hash function (as well as the baseline function for comparison) for a variety of hash code sizes. As before, the x-axis measures each of the hash code sizes we tested. The y-axis measures the fraction of times that the hash function generated a unique hash code (rather than a collision).

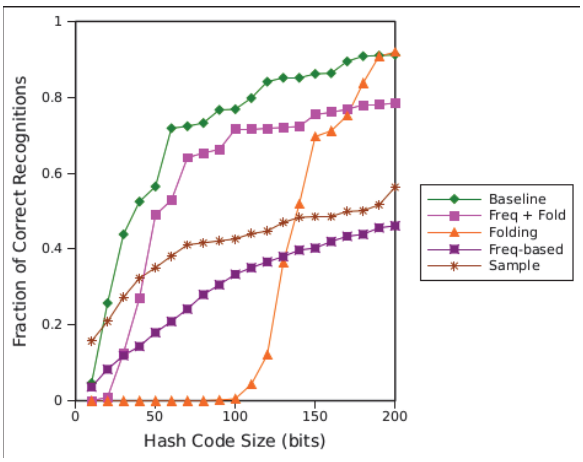

Figure 3: Uniformity Comparison

These results, drawn from the Eaters environment, show that the combination of folding and frequence-based selection performs best when hash code sizes are not too generous. Results from the other environments were comparable.

Reversible The use of a hash formula allows the resulting hash function to be reversible. In particular, the episode can be *partially* reconstructed by comparing the bits in the hash code to the hash formula used to extract the set of elements that was originally used to generate the code.

Figure 4 compares how effectively an episode could be rebuilt from its hash code for each of the hash functions. In cases where a bit was associated with multiple elements (a folding hash formula) the recovered element was selected at random. The x-axis indicates the number of episodes that had been processed and the y-axis shows the average number of elements a rebuilt episode had in common with the original.

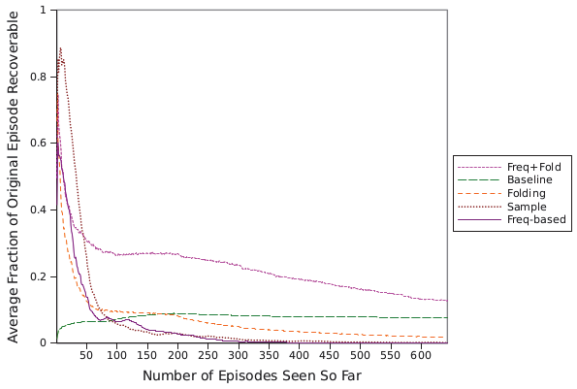

Figure 4: Reversibility Comparison

Again, these results show that folding and frequency-based solution maintains more fidelity than other methods. They were gathered using a hash code size of 100 bits and the TankSoar environment. We saw similar results in other environments and with other hash codes sizes provided that the number of bits in the hash code was significantly less than the number of elements in the episodes.

Locality Sensitive The use of a hash formula met our requirement for a hash function that is locality sensitive as described above. In particular, episodes that contain a similar set of elements are more likely to result in a code that has similar bits set.

Online All of the hash functions we tested were specifically designed to be online algorithms and have been successfully implemented as such.

Discussion and Future Work

In this paper, we present techniques for creating episodic memory hash functions that provide recognition, similarity detection and partial recall for an arbitrary sequence of episodes using limited storage. Our results indicate that a hash-formula-based approach works best when using a combination of two techniques that we call folding and frequency-based selection. We hypothesize that this is the most effective because a combination of both approaches produces hash codes with the highest information density.

For future work, it may be possible to improve the fidelity of retrieved episodes using these techniques by storing a finite number of "frame" episodes that fill in ubiquitous details in the retrieved episode.

Additionally, these approaches have yet to be tested holistically. Comparing the performance of an episodic memory agent using hash-code based storage instead of a simpler forgetting mechanism may be informative.

References

Almeida, P.; Baquero, C.; Preguica, N.; Hutchison, D. 2007. Scalable Bloom Filters. In *Information Processing Letters* 6:255261.

Bloom, B. 1970. Space/Time Trade-offs in Hash Coding with Allowable Errors. In *Communications of the ACM* 7:422426.

Brom, C.; Lukavsk, J. 2009. Towards More Human-Like Episodic Memory for More Human-Like Agents In *Proceedings of the 9th International Conference on Intelligent Virtual Agents*.

Cover, T.; Thomas, J. 2012. *Elements of Information Theory*. New York, NY: John Wiley & Sons.

Dodd, W.; Gutierrez, R. 2009. The role of episodic memory and emotion in a cognitive robot In *IEEE International Workshop on Robot and Human Interactive Communication*.

Harris, Z. 1954. *Distributional structure*. In *Word* 10:23.

Hua, Y.; Xiao, B.; Veeravalli, B.; Feng, D. 2012. Locality-Sensitive Bloom Filter for Approximate Membership Query In *IEEE Transactions on Computers* 61(6):817-830.

Indyk, P.; Motwani, R. 1998. Approximate Nearest Neighbors: Towards Removing the Curse of Dimensionality. In *Proceedings of 30th Symposium on Theory of Computing*.

Jones, K. S. 1972. A Statistical Interpretation of Term Specificity and Its Application in Retrieval. In *Journal of Documentation* 28:11-21.

Kennedy, W.; De Jong, K. 2003. Characteristics of Long-Term Learning in Soar and its Application to the Utility Problem. In *Proceedings of the Twentieth International Conference on Machine Learning*.

Laird, J. E. 2012. *The Soar Cognitive Architecture*. Cambridge, MA: MIT Press.

Marron, M. E.; Kuhns, J. L. 1960. On relevance, probabilistic indexing, and information retrieval. In *Journal of the Association for Computing Machinery* 7(3):216-244.

Mitchell, M. 1996. *An Introduction to Genetic Algorithms*. Cambridge, MA: MIT Press.

Nuxoll, A.; Tecuci, D.; Ho, W. C.; Wang, N. 2010. Comparing Forgetting Algorithms for Artificial Episodic Memory Systems. In *Proceedings of the Thirty Sixth Annual Convention of the Society for the Study of Artificial Intelligence and Simulation of Behaviour* (AISB).

Nuxoll, A., and Laird, J. E. 2012. Enhancing Intelligent Agents with Episodic Memory. In *Cognitive Systems Research* 17-18:34-48.

Ram, A.; Santamara, J. 1997. Continuous Case-Based Reasoning. In *Artificial Intelligence* 90(1-2):25-77.

Tulving, E. 1983. *Elements of Episodic Memory*. Oxford, UK: Clarendon Press.

Vitter, J. 1985. Random Sampling with a Reservoir. In *ACM Transactions on Mathematical Software*.

Wallace, S.; Dickinson, E.; Nuxoll, A. 2013. Hashing for Lightweight Episodic Recall. In *Proceedings of the AAAI Spring Symposium: Lifelong Machine Learning*.