Hashing for Lightweight Episodic Recall

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

Andrew Nuxoll
University of Portland
5000 N. Willamette Blvd.
Portland, OR 97203

Abstract

We demonstrate a supplemental episodic memory system that can help arbitrary Soar agents use reinforcement learning in environments with hidden state. Our system watches for learning bottlenecks and then specializes the agent's existing rules by conditioning on recent history. Because we avoid a full episodic retrieval, performance scales well regardless of the agent's lifespan. Our approach is inspired by well established methods for dealing with hidden state.

Introduction

Complex, durative tasks that are performed by intelligent agents over a series of months, years, or even decades present significant challenges for the field of artificial intelligence. For such tasks, we take as a given that learning will be required as it is exceedingly unlikely that the agent's initial knowledge (as supplied by a programmer) will be sufficient to perform all of its tasks correctly over the course of time. These knowledge deficiencies may be a result of bugs introduced during programming, an insufficient model of the task, or they may be due to changes in the environment.

For the purposes of this paper, we focus on two key challenges that arise from this context. The first is the challenge of dealing with, and overcoming, flawed knowledge. Specifically, we will examine situations in which the the programmer's understanding, or model, of the environment is imperfect and thus fails to make some critical features of the environment available to the agent. The second challenge is to deal with these changes in such a way that scales well with the environment's complexity and with the duration of task.

In this paper, we present our recent work with the Soar agent architecture targeting the challenges listed above. Soar provides a compelling reference platform since much of its development effort has been aimed at supporting large scale and potentially durative tasks. In particular, we are interested in providing Soar with a mechanism to identify when hidden state variables make the agent's own internal state representation inappropriate for successful reinforcement learning. For this task, we leverage existing work on k-Markov models and Utile Distinction Memory, and present an initial platform based on these techniques.

The contributions of this paper are threefold: first, we demonstrate a supplemental episodic memory system that can help arbitrary Soar agents use reinforcement learning in environments with hidden state; second, we show that our approach scales well (regardless of the agent's lifespan) because we are able to avoid full episodic recall; and third, we provide an argument as to why Soar, and other agent architectures, may benefit from architecturally supported mechanisms to track state novelty and familiarity.

The remainder of the paper proceeds as follows. First, we provide an overview of the Soar architecture including two of its salient learning mechanisms. Next, we present two simple environments that illustrate the problems associated with hidden state. At the same time we review recent work to explore how existing Soar agents deal with hidden state. In the third section, we introduce our lightweight episodic store and learning mechanism. Finally, we evaluate the performance of our implementation and then conclude with ideas for future work.

Soar

Soar (Laird, Newell, and Rosenbloom 1987) is a cognitive architecture and agent development platform with a long history in both artificial intelligence research and in cognitive psychology research. For many years, Soar has aimed to facilitate developing agents for complex, real-world environments (Jones et al. 1999; Wray et al. 2004; Laird et al. 2012).

In Soar, a knowledge engineer (KE) defines an agent's knowledge as a set of production rules. The rules match against sensory information obtained from the external environment and against internal data structures that the agent may generate and maintain on its own. Collectively these external and internal data structures are known as working memory. Rules match working memory and fire in parallel so that at any point in time, all relevant knowledge is brought to bear.

Operators are actions that are proposed, selected and performed by production rules. In contrast to rules, operators can be selected only "one at a time" and typically represent actions or more complex goals that can themselves be decomposed into sub-goals and actions. Operators can generate persistent internal data structures to support the agent's reasoning, or can be used to execute primitives within an external environment.

Soar agents perform tasks by executing a five phase decision cycle as follows:

Input Phase First, the agent's sensors acquire information from the external world and make this information available for reasoning.

Proposal Phase Second, the agent creates new truth-maintained data structures based on its current internal state and its perceptions about the world using elaboration rules. At the same time, proposal rules determine a set of operators that may be relevant for the current situation and preference rules provide knowledge about the relative suitability of each proposed operator. The phase continues with proposal and elaboration rules firing and retracting until a quiescent state is reached in which no more productions match.

Decision Phase Once quiescence has been reached, Soar evaluates the preferences for each of the proposed operators and selects the best one to pursue. If the decision phase cannot identify a suitable candidate, an impasse is generated to trigger further deliberation.

Application Phase Once a single operator has been selected, Soar uses knowledge in application rules to carry out the operation.

Output Phase Soar informs the external environment of any actions that the agent is attempting to initiate so that the environment can respond for the next input phase.

For many years, Soar has employed a learning mechanism called chunking that allows a Soar agent to operationalize its knowledge and potentially consolidate the effects of many operators into a single, newly learned, rule. Modern versions of Soar (Laird 2008) also include reinforcement learning and episodic memory. Both of these additions hold significant potential for dealing with complex environments in which the knowledge engineer may not have a full understanding of the state space or state transitions.

Reinforcement Learning

In Soar, operator preferences are traditionally specified with binary or unary symbolic relations such as "better than", "worse than", "best" and "reject". Modern versions of Soar also allow a numeric preference to be used when symbolic preferences are insufficient for Soar to select a single best course of action.

In Soar, reinforcement learning acts to modify these numeric operator preferences. Put another way, an operator preference rule can be viewed as a mapping between a set of states (as specified by the rule's conditions) and an operator. Reinforcement learning modifies the numeric preference a rule assigns to an operator so that it is a function of the expected total discounted reward of pursuing the specified action in the specified state. Since many rules may match a given state, many numeric preferences may be assigned to a given operator. Thus, the Q-value for a specific state, operator (action) pair, (s, a), is distributed across, and thus a function of, *all* the rules that match the state s and provide a numeric preference for the operator a.

Episodic Memory

Soar's episodic memory keeps detailed information about the history of states that an agent has visited during the course of performing a task. Episodes can be viewed as a stored record of the agent's state along with any actions it is attempting to perform. Episodes are then linked together in memory so that a series of events can be replayed forward or backward.

Retrieving an episodic memory begins when the agent constructs a cue in working memory. The cue is then used to perform a fast *surface* match against stored episodes. Here, episodes are scored based on how many of the cue's leaf elements (fully specified by their path from the cue's root) are matched in the candidate episode. If a perfect surface match is found, a structural graph match can be performed to break ties. The structural match performs unification to ensure, for example, that identifiers shared by multiple leaves in the cue are similarly shared by corresponding leaves in the candidate episode. The retrieved episode is either the most recent structural match or the highest scoring surface match. Episode storage and retrieval is currently efficient enough to support real-time behavior (with a typical matching time of 50ms) in an environment that produces 1M episodes over a fourteen hour time period (Derbinsky and Laird 2009).

While episodic memory, is not, in itself a learning mechanism (it is a storage vehicle), it does lend itself to a variety of learning approaches. In the following section, we will explore some recent work integrating Soar's episodic memory and reinforcement learning. From this, we will highlight limitations of the current episodic memory system and explore an initial method for rectifying them.

Soar and Hidden State

Well World

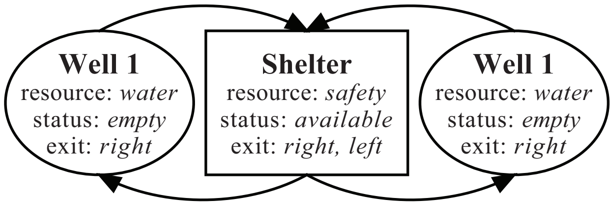

Figure 1: The Well World Environment

In prior work, Gorski and Laird examined how Soar could learn to use its episodic memory to overcome problems associated with hidden state in its task environment (Gorski and Laird 2011). The environment explored in this work was entitled "Well World" and consists of three locations: two wells and a shelter (Figure 1).

In Well World, the agent obtains rewards such that the optimal behavior is to stay at the shelter location until sufficient time has elapsed that it becomes thirsty. At this point the agent should move to the well to drink water and then immediately return to the shelter until it becomes thirsty again.

The environment is interesting for two reasons: first, the agent cannot see the contents of any location it does not occupy, and more importantly, the wells are not always full of water. Instead, at any given time, water is only available at one of the wells. Once the agent drinks the water from the well, it becomes empty and the other well becomes full. Thus, the agent's problem is to learn that it must use it's episodic memory to recall which well it drank from previously so that it can go to the well that currently contains water.

In the Well World, Gorski and Laird demonstrate that the existing capabilities of Soar's reinforcement learning system and episodic memory are sufficient for the agent to learn this dynamic without help of background knowledge. The agent can determine which well to move toward by recalling a single episode (the last well visited) and then moving to the other well. Alternatively, the agent may try to recall how it dealt with thirst previously and base its solution on this memory. In Well World, these strategies work because the best course of action can be determined by knowing a *single* historic state (which well contained water previously) even though that state may be arbitrarily deep in the agent's history (since it may have waited a long time at the shelter before getting thirsty).

The Flip Environment

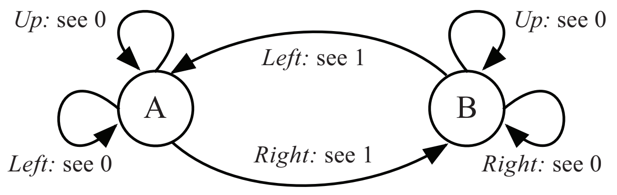

Figure 2: The Flip Environment

The flip environment (Singh et al. 2003) consists of two states *A* and *B* which are illustrated as the left and right circles in Figure 2 respectively. From any state, movements of *Up*, *Left* and *Right* are possible although two of the three actions will not result in a transition between *A* and *B* or vice-versa. The agent has a sensor *see* which can take on the values 0 or 1. *see* becomes 1 only when the agent transitions between *A* and *B* or vice-versa. The agent's goal is to predict the value that *see* will take. However, the fact that the environment contains two states is hidden from the agent.

Note that because the state labels are hidden from the agent, this environment is non-Markovian. Moreover, although we could improve the situation by re-encoding the state to contain information from the previous action, problems remain. Since the *Up* action is, in essence, a null operation, no fixed (and finite) length history will ensure that all the information required to make a correct prediction about the value of *see* is available.

Soar and Flip

The Flip environment differs in a subtle but critical way from the Well World environment. In the Well World, a suitable strategy for episodic retrieval begins by recalling an episode that meets a known criteria: either "recall the last time I was drinking water", or "recall the last time I was thirsty". And then, in the case of the later recall, following the episodic trail until the most recently used well is discovered (i.e., until state that meets the criteria "last time I was drinking water" is reached). Thus, determining the correct well to visit can be done simply by recalling the last well from which the agent drank.

In the Flip environment, however, making a correct prediction is not quite as simple. Instead of being able to find a *single* state upon which to base its behavior, an agent in the Flip environment must base its behavior off multiple historic states.

To make the relative complexities of the Well World and Flip environments concrete, below we will discuss the behavior of three possible agents that use episodic memory to inform their behavior.

Ep0 An Ep0 agent makes the simplest possible use of its episodic memory. This agent searches episodic memory for a successful course of action that begins in a situation similar to the current state. It then applies this prescribed behavior with the hope that it will lead to success in the current situation.

In an environment that is Markovian with respect to the agent's sensors, this strategy is sufficient for success. However, the practical limits of this strategy are serious The Ep0 agent will not succeed in the Well World, since wells alternate between full and empty based on where the agent drank last. Similarly, the Ep0 agent will not succeed in the Flip environment because the agent's sensors do not tell it whether it is in state *A* or *B*.

Ep1 An Ep1 agent performs the same episodic recall as its predecessor Ep0. However, instead of simply performing previously successful behavior in a new situation, the Ep1 agent *conditions* its current behavior on a single recall from episodic memory. In effect, the agent conditions its behavior on two states (the current state and a previous state), but unlike a second order Markov model, the states need not be adjacent in time.

In the Well World, this approach is sufficient for success. The agent can condition its behavior on where it drank last as we described previously. In this fashion, the agent can learn to visit alternating wells. The Ep1 approach, however, is not sufficient for the Flip environment since predicting the value of *see* following a *Left* movement *requires* information that can only be obtained by examining multiple historic states.[1]

Ep2 An Ep2 agent is capable of incorporating information about interactions between previous episodes. A domain-*dependent* implementation of an Ep2 agent may recall a particular episode from memory and then walk to other episodes that must be used to inform current behavior. This is the approach that is taken in the Well Worlds for the agent that informs its behavior by recalling the last

[1]i.e., the agent could retrieve the prior state $s_i|see = 1$, but this information would not be useful unless the agent also viewed s_{i-1} to determine which action had caused the observation.

time it was thirsty. In the Flip environment, a similar strategy could be employed. In both cases, however, a strategy would require a detailed understanding of the environment's dynamics in order to minimize the number of necessary recalled states.

A domain-*independent* Ep2 agent can be obtained by conditioning on a *sequence* of previous episodes. Assume that the agent's history is a sequence of episodes: $e_0, e_1, e_2, \ldots, e_j$ such that e_j is the episode immediately preceding the current state. Call the k-suffix of the history the subsequence $e_{j-k+1}, e_{j-k+2}, \ldots, e_j$. If, for increasing values of k, the agent iteratively searches its episodic memory for a previous occurrence of the k-suffix, then the longest match will be the agent's best correspondance between a historical situation and the agent's current situation. In essence, the domain-independent Ep2 agent is using the episodic store to create a k-Markov model representation. Although no finite k will allow perfect predictions in the flip environment, Ep2's k-length match will grow over time as the agent gains more experiences. As a result, the number of situations in which correct predictions are possible increases at the obvious cost of increasing match complexity.

Implementing a domain-independent Ep2 strategy in Soar is impractical with the current episodic memory system. Episodic recall can be performed once per decision cycle. So, Soar would need to iteratively deepen recall of its immediate past history to use as cues to find matches further back in time. Thus, Soar would require at least $2k$ episodic retrievals to match a k-suffix (k to retrieve the suffix itself, and at least k more to find the same sequence in the past).

Unfortunately, the true cost would be much worse for two reasons: first, we would expect many more matches of a 1-suffix than a 2-suffix; and second, we should expect match cost to increase as more episodes are stored. Thus, the shortcomings of the Ep2 strategy are significant. In addition to a high implementation complexity, the approach suffers from growing cost of episodic retrieval as the agent's episodic memory grows.

A Modified Episodic Store

As illustrated above, Soar's episodic memory system as currently implemented makes learning in the Flip environment extremely difficult without domain dependent knowledge. Our aim is to substantially improve Soar's learning performance within this environment (and similar environments) while making a minimal set of changes to Soar's architectural commitments.

If we examine the behavior of the Ep2 agent from an implementation-independent standpoint, the key requirement is efficient retrieval of a sequence of episodes matching the agent's recent history. In Soar's current implementation, this is difficult because the episodes must be recalled independently – there is no way to match or recall based on a sequential cue. An obvious approach to deal with this might be to modify Soar's episodic memory so that retrieval cues can span multiple episodes. However, this approach will still suffer from a growing retrieval cost as the size of episodic memory increases.

If we leverage the fact that the Ep2 agent is *conditioning* its current behavior on its k-state history, then a different approach is possible. Specifically, we can avoid full, explicit, retrieval of the historic state. In other words, it is enough to know that the historical sequence we are interested in is $e_{j-k+1}, e_{j-k+2}, \ldots, e_j$. We do not need to look inside the episodes themselves. Instead, we can simply generate a set of symbols that serves as a unique identifier for the relevant sequence. This distinction is subtle but critical as it will free us from growing overhead as our episodic memory increases in size.

The approach we have pursued is to implement exactly this symbolic identifier using a hashcode generated from the agent's recent history. The hashcode allows the agent to identify equivalent states (and histories) since these share the same hashcode and thus allows us to avoid a full explicit episodic recall. The benefit of this approach is that it's extremely lightweight. The hash code can be generated as the episode is being stored and the agent can thus keep a history of all hashcodes encountered with relatively small overhead.

For the purposes we are interested in, however, we can suffice with a set of hash codes for recent states (as opposed to for the entire history). By itself, this is not a learning mechanism. For that, we need to explore the implications of different hash functions and approaches for identifying when to employ our state-based hashing.

Hashing

The hash function defines an equivalence relationship over states. Our current choice of hash function is motivated by Soar's cue matching. Specifically, we base our hash code on the union of all grounded literals in the episode Thus, episodes that have a perfect *surface* match, will also share the same hashcode.

The upside to this approach is that it allows the system to distinguish states at a very fine level of granulation. The obvious downside, however, is that very little generalization takes place; the agent can only detect an exact match between states.

An interesting side effect of the lightweight state hashing scheme we have implemented, is that it gives agents a built in mechanism for identifying familiar and novel situations. The familiarity of a particular state can simply be measured as the number of times a particular state has been encountered. While this may not seem to be of particular importance to readers most familiar with HMM or POMDP models where the state representation is monolithic, it is much more interesting from the standpoint of Soar where *there is no single data structure* representing the agent's state, and thus there is no obvious way to store information that indicates whether a particular situation is either new or familiar.

Employing Hashing

Hashing provides a lightweight mechanism to distinguish states and state sequences from one another. As such, it gives us a critical stepping stone to help improve Soar's learning capabilities. On its own, however, hashing is insufficient. For autonomous learning, we need a mechanism that will detect when the programmer-supplied domain knowledge is insufficient to allow reinforcement learning to succeed. For this task as well, we look for inspiration in the well-established methods of prior work.

Utile Distinction Memory Utile Distinction Memory (UDM) was introduced by McCallum (Mccallum 1993) to identify situations in which an agent could improve its ability to predict future discounted reward by refining its notion of state representation. UDM works by tracking the future discounted rewards through a given state. Confidence intervals for these rewards are associated with the actions that lead to a particular state (i.e., incoming edges in a transition diagram) and one confidence interval is maintained for each action that may be performed in the state (i.e., for each outgoing edge in the transition diagram).

UDM performs a utile distinction test by examining the confidence intervals for an outgoing action a_i^{out} in state s maintained for each incoming action a_j^{in}. If s satisfies the Markov property, then the confidence intervals for the outgoing action should be consistent across each incoming action. In other words, knowing the incoming action (i.e., incoming edge) should not impact the discounted future rewards of the agent. If, on the other hand, the confidence intervals are significantly different based on the incoming action, this suggests that the agent would benefit from refining its notion of s by incorporating knowledge about its incoming actions. Thus, the original state s is cloned to create multiple states that, in essence, differ based on their one step history.

Implementing UDM within Soar, however, is non-trivial. While a Soar agent can clearly be viewed as moving through a series of states by performing actions with operators, as noted before there is no single data structure to represent state in Soar. Instead, the notion of state arises from the set of currently matching rules and thus is distributed as opposed to being monolithic. This seemingly minor difference introduces a significant hurdle for implementing UDM in Soar.

Reward Variance Tracking UDM is designed to work with environments that have non-deterministic action transitions and non-deterministic rewards. If we make the simplifying assumption that the environment is deterministic both with respect to rewards and action transitions, then we can attribute any variance in rewards to an imperfect state representation. We'll call this approach Reward Variance Tracking (RVT).

From an implementation stand point, the critical difference between UDM and reward variance tracking is that the additional data structures to track the future discounted reward distribution are associated with a state and its incoming actions in UDM, and a state and its outgoing actions for RVT.

In Soar, a state and it's outgoing action are directly encapsulated by the matching operator preference rules. This means that reward distribution data structures can be maintained by each operator preference rule along with the expected current reward and expected discounted future reward information that Soar's RL system already maintains. Using an incremental variance calculation (Knuth 1998) we can add RVT to Soar's RL system using changes to the rule data structure that are constant in both time and space.

Implementation

As our baseline implementation, we began by adding Reward Variance Tracking to Soar 9.3.2. Reward mean and variance values are associated with every rule in Soar that can be updated by reinforcement learning. Our episodic hashing system, dubbed ZigguSoar, is implemented in Java and listens to the interaction between a Soar agent and its environment. Each time Soar issues a command to the external environment, ZigguSoar captures the agent's current state (by examining its input and output link) and generates a hash-code.

Periodically, as the agent is pursuing its task, ZigguSoar requests reward variance data from the agent. A rule that reports significant variance can be considered a candidate for specialization by augmenting its conditions with information from the episodic hash.

Consider, for example, the following Soar rule which provides a numeric preference to the operator that would predict the *see* variable will be 1 after a *Left* action.

```
sp { prefer*predict-yes*left
    (state <s> ^io.input-link.dir left
               ^operator <o> +)
    (<o> ^name predict-yes)
 -->
    (<s> ^operator <o> = 0.0)}
```

Because this rule is overly general, the rewards obtained will be inconsistent. ZigguSoar will observe this variance and add a corresponding *template* rule that is conditioned on a one-step history. The template rule will create child rules based on all observed groundings of the one-step history. The process of rule specialization will continue in a recursive fashion with these child rules if needed. Currently, we do not remove overly general rules from the agent's rule-base. The numeric preferences obtained via reinforcement learning will correctly organize themselves once the appropriate level of specialization is obtained regardless of whether or not there are also overly general rules in the mix.

Experiments

Figure 3 illustrates the performance of the ZigguSoar agent in the Flip environment during the first 10,000 steps averaged over 20 runs and smoothed with a 100-step moving average. The *Baseline Soar* agent shows the performance of Soar using the same rules that are initially supplied to *ZigguSoar*, but without the rule specialization provided by our lightweight state hashing and episodic learning module. Note that *both* agents are performing reinforcement learning with the same learning parameters. However, for the baseline agent, learning is ineffective because of the environment's hidden state.

Figure 4 illustrates the performance of the same agents (using the same data as before) but focuses on *yes* predictions for *Up* and *Left*. Here we see that, indeed, *Baseline Soar* is capable of learning, but learning is limited to the operation with a deterministic result (*Up*) while *ZigguSoar* is capable of learning predictions for both operations.

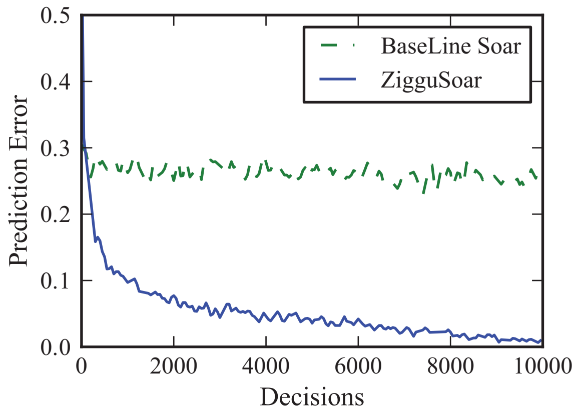

Figure 3: Overall Performance in Flip

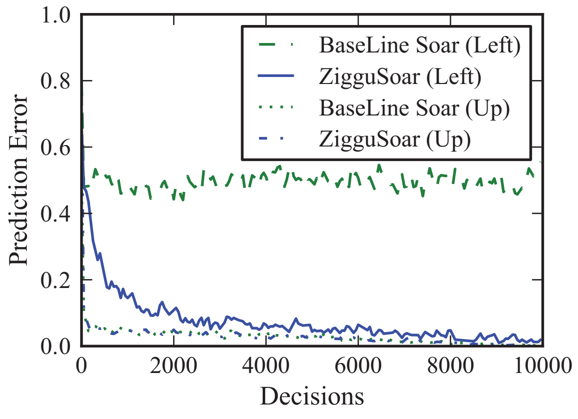

Figure 4: Performance of Up and Left predictions

Finally, Figure 5 illustrates how the ZigguSoar agent's rule set grows over time. The upper line (*Total Rules*) shows the total number of rules averaged over the course of 20 runs. *Templates* indicates the number of variablized Soar template rules that are added by ZigguSoar to assist with learning. Since no finite history can capture all the information required to make all correct predictions in this environment, the size of the rule set will continue to grow unbounded. In Flip, new templates will be acquired progressively slowly as the length of necessary history increases.

Conclusions and Future Work

We have demonstrated lightweight episodic recall using hashing implemented as a standalone Java framework that can be integrated with existing Soar environments and agents. Our framework leverages previous methods for handling hidden state and provides Soar agents with the capability of responding to demands posed by such environments.

While we believe that this implementation does represent a valuable improvement to Soar's episodic memory system for some agents, we also recognize that the methods we have employed are very early predecessors of the current state of the art. Our next steps will be to explore how to incorporate current methods for dealing with hidden state such as Probabilistic State Representations (Singh et al. 2003) or Looping Suffix Trees (Holmes and Isbell 2006) into the ZigguSoar framework.

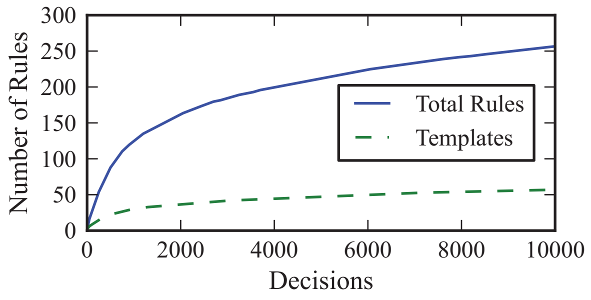

Figure 5: Rule Count Over Time

References

Derbinsky, N., and Laird, J. E. 2009. Efficiently implementing episodic memory. In *Proc. of the 8th Int. Conf. on Case-Based Reasoning*, 403–317.

Gorski, N. A., and Laird, J. E. 2011. Learning to use episodic memory. *Cognitive Science Research* 12:144–153.

Holmes, M. P., and Isbell, Jr, C. L. 2006. Looping suffix tree-based inference of partially observable hidden state. In *Proc. of the 23rd Int. Conf. on Machine Learning*, 409–416. ACM.

Jones, R. M.; Laird, J. E.; Nielsen, P. E.; Coulter, K. J.; Kenny, P.; and Koss, F. V. 1999. Automated intelligent pilots for combat flight simulation. *AI Magazine* 20(1):27–42.

Knuth, D. E. 1998. *The Art of Computer Programming, Volume 2: Seminumerical Programming*. Boston: Addison-Wesley, third edition.

Laird, J. E.; Kinkade, K. R.; Mohan, S.; and Xu, J. Z. 2012. Cognitive robotics using the soar cognitive architecture. In *Proc. of the 8th Int. Conf. on Cognitive Robotics*. Toronto, Canada: AAAI Press.

Laird, J. E.; Newell, A.; and Rosenbloom, P. S. 1987. Soar: An architecture for general intelligence. *AI* 33(1):1–64.

Laird, J. E. 2008. Extending the soar cognitive architecture. In *Proc. of the 1st Conf. on Artificial General Intelligence*, 224–235. Amsterdam, The Netherlands, The Netherlands: IOS Press.

Mccallum, R. A. 1993. Overcoming incomplete perception with utile distinction memory. In *Proc. of the 10th Int. Conf. on Machine Learning*, 190–196. Morgan Kaufmann.

Singh, S.; Littman, M. L.; Jong, N. K.; Pardoe, D.; and Stone, P. 2003. Learning predictive state representations. In *Proc. of the 20th Int. Conf. on Machine Learning*.

Wray, R. E.; Laird, J. E.; Nuxoll, A.; Stokes, D.; and Kerfoot, A. 2004. Synthetic adversaries for urban combat training. In *Proc. of the 16th Conf. on Innovative applications of artifical intelligence*, IAAI'04, 923–930. AAAI Press.