Integrating Declarative Long-Term Memory Retrievals into Reinforcement Learning

Justin Li JUSTINNHLI@OXY.EDU
Computer Science, Occidental College, Los Angeles, CA 90041 USA

Abstract

The common model of cognition defines both declarative long-term memories (LTM) and reinforcement learning (RL) components, but their interaction remains relatively unexplored. We present a framework in which agents that follow the common model of cognition can learn to retrieve from LTM based only on task rewards, and use the resulting knowledge to select actions. This task- and knowledge-agnostic approach defines an internal RL problem where the agent's working memory is the state, with actions that allow the agent to query LTM. We use linear weights on predicate-object-pair features to approximate the value function, speeding up learning when new entities are encountered. Empirical experiment evidence show that the agent can learn to retrieve relevant knowledge even when it has access to the complete contents of DBpedia, and that it can adapt even when the knowledge required changes over time.

1. Introduction

The recently proposed common model of cognition (Laird et al., 2017) represents a convergence of research in the cognitive systems and cognitive modeling communities. The common model describes the fixed-mechanisms of a human-like mind, including components for the storage of both procedural and declarative knowledge, as well as the interfaces that define how they interact. The model also defines the means of learning in these components: where declarative learning occurs via the storage of new information, procedural learning occurs via reinforcement that adjusts action selection. While declarative long-term memory and reinforcement learning are core parts of the memory and learning systems in the model, research has tended to focus on each system individually, and rarely on how these two components interact.

This represents a significant gap in the literature. The vast majority of agents created in cognitive architectures have hard-coded rules for memory access, with no narrative for how those rules might be learned. These agent- and task-specific rules for memory retrieval are necessarily brittle, as they embed assumptions both about the task and about the representation of knowledge, and therefore fail to allow the agent to adapt to changes in the environment. Finally, the fixed patterns of memory access fail to explain how an agent might learn domain-independent higher-level memory strategies, such as rehearsal and knowledge search. These strategies build on top of the architectural memory mechanisms, and have been posited to be necessary for complex reasoning in general autonomous intelligent agents (Laird & Mohan, 2018). In order to model the complex memory processes in people (Burgess & Shallice, 1996), as well as to endow artificial agents with general memory capabilities, a generic method for learning to use memory is necessary.

In this paper, we take a step towards this goal by proposing a reinforcement learning framework that is capable of learning to retrieve knowledge from a large knowledge base in declarative long-term memory. Our framework is capable of generalizing across different entities in a knowledge base, and can adapt to different task domain rewards. To our knowledge, we are the first to investigate reinforcement learning over large knowledge bases in a cognitive architecture. The main contributions of our paper are:

- We present a task- and knowledge-agnostic system that is capable of retrieval knowledge from long-term memory, which can then be used for action selection, both via reinforcement learning.
- We show that our architecture is able to learn different patterns long-term memory access, that corresponds to different graph structures in the knowledge base.
- We show that our architecture is able to learn even when the knowledge base has millions of entities and tens of millions of facts.

2. Background

We are interested in the problem of how an agent might, in the context of a reinforcement learning task, bring in relevant knowledge from long-term memory for action selection. Crucially, it is the task reward that determines the appropriate knowledge to retrieve. In this section, we first describe the relevant components of the common model, before casting them into reinforcement learning and knowledge representation terms. We do this both for notational clarity, and to show that the system is agnostic to the contents of knowledge base.

Any non-trivial artificial agent must represent its environment internally. In the common model of cognition, this representation is stored in *working memory*, which contains not only the agent's percepts, but also other knowledge that is immediately relevant to the situation. The contents of working memory represents the current state of the agent, and serves as the only source of knowledge for reasoning and decision making.

In addition to working memory, the agent also has a *long-term memory* (LTM), which contains all knowledge which may or may not be relevant at any time. LTM is commonly further divided into autobiographical knowledge stored in episodic memory and world knowledge stored in semantic memory; this paper focuses on the latter. Knowledge in semantic memory may be vast, and previous work have included linguistic knowledge from WordNet and common sense knowledge from DBpedia (Salvucci, 2015; Li & Kohanyi, 2017). For clarity, we will use "*knowledge base*" (KB) to refer to a source of knowledge and "long-term memory" to refer to the component in an agent architecture, into which a KB may be loaded.

Since only working memory is usable for reasoning, LTM knowledge must first be copied into working memory. The common model of cognition defines two ways to access LTM. First, the agent could *query* or search LTM by creating a *cue*, which is a description of the desired knowledge. For example, a query might ask for "a rock band formed in 1965", is then matched with the contents of

LTM. A result of this search (e.g., Pink Floyd), as well as associated information about that entity (e.g., its country of origin), is then copied into working memory for agent use. The agent is also able to *retrieve* information about an entity that is already in working memory. We overload the term "retrieval" to mean either method of copying LTM knowledge into working memory.

Formally, the contents of LTM is an edge-labeled directed graph, defined by the tuple $\langle P$, U, L, $C\rangle$ where U is the set of entities; P the set of predicates; L the set of literals, which correspond to data such as numbers and strings; and C the contents of the KB, a set of *triples* $\langle U, P, U \cup L\rangle$. For convenience, we refer to the three elements of a triple as the *subject*, the *predicate*, and the *object*. For example, the fact that Pink Floyd's album *The Wall* was released on November 30, 1979 is represented by the triple $\langle$`The_Wall`, `releaseDate`, `"1979-11-30"`$\rangle$. Since we will refer to the predicate-object pair of a triple frequently, we denote the objects as $V = U \cup L$ and call a pair $\langle p \in P, v \in V\rangle$ an *attribute* of the subject entity.

Within this formalism, the cue for querying LTM for knowledge is a set of attributes that describe the same entity. Assuming such an entity exists, all the attributes of that entity will be placed into working memory. Similarly, the retrieval of an entity would place all its attributes into working memory.

Finally, we assume that the agent task to be a reinforcement learning problem, which is defined by the tuple $\langle S, A, T, R\rangle$, where S is the set of states, A the set of actions, $T : S \times A \rightarrow S$ the transition function, and $R : S \times A \rightarrow \mathbb{R}$ the reward function. Importantly for this work, we do *not* consider working memory or LTM as part of the state, but only the state of the environment. This allows us to define a task (e.g., finding an album at the record store) independently from the knowledge the agent can access (e.g., information about bands, genres, etc.). To make this distinction clear, we call the task the *external environment*, defined by the external states S_{ext}, actions A_{ext}, and so on. We define the corresponding *internal environment*, which includes actions for accessing LTM, in Section 5.

3. Problem Definition

The underlying motivation for integrating LTM retrievals with RL is that knowledge in LTM may allow the agent to generalize and transfer its the policy to new percepts. The knowledge retrieved from LTM serve as features that could reduce the state space, or that may de-alias partially-observable states. Since the use of RL with working memory is well-studied (Nason & Laird, 2005; Taatgen et al., 2005), the main challenge is in representing the interface between working memory and LTM, and in propagating the reward to queries and retrievals.

This problem of learning to retrieve from memory is difficult for several reasons. First, LTM may contain large amounts of knowledge — DBpedia, for example contains information about 3M entities, and multiple attributes about each entity. It is therefore computationally intractable to consider the entire contents of LTM during action selection. Second, the agent does not know what information is relevant. While we assume that relevant knowledge exists in LTM, only a tiny subset of the KB is likely to be meaningful in any particular situation. Finally, creating cues for LTM presents a large discrete state and action space, which have traditionally been difficult for RL agents. Techniques that apply elsewhere are not applicable to the problem of retrieving from

LTM, as the actions cannot be embedded in a higher-dimensional space (Dulac-Arnold et al., 2015). At the same time, the semantic structure of KBs suggests that generalizations should be possible, where the same predicates apply to multiple entities. These challenges require an efficient value function approximation that could ignore the majority of irrelevant knowledge in the KB while taking advantage of predicate semantics for generalization.

We make two simplifying assumptions in our approach. First, we assume that the external percepts are represented as sets of predicate-object pairs, and that they match their representation in LTM. This representation of percepts is in line with other cognitive architectures, and it allows us to sidestep the problem of symbol alignment with the KB, which we consider out of scope for this paper. Second, we assume that the appropriate knowledge to retrieve from LTM is fixed — that is, the same graph structure relates the environmental percept to the desired knowledge. This is again in line with how ACT-R and Soar agents are written, and as it is hard to imagine the situation in which knowledge is necessary but is completely unrelated to the current percepts.

We are additionally concerned with two desiderata. First, the architecture should be agnostic to both the task and the knowledge base — it should not depend on particularities of the KB or the domain. In particular, the architecture should not require knowledge of the structure of the KB, or what knowledge should be retrieved for the task. Second, the system should be consistent with how reinforcement learning is implemented in existing architectures such as ACT-R and Soar, as they are the foremost examples of the common model.

4. Related Work

While there is work on using large KBs for cognitive modeling in the cognitive systems community (Douglass et al., 2009; Li & Kohanyi, 2017), the queries and retrievals are almost always hard-coded and not learned. The closest work to this paper is Gorski & Laird (2011), where the agent learned to use Soar's episodic memory, but retrieving knowledge from a large declarative KB presents additional challenges. First, KBs such as DBpedia are significantly larger and richer than the accumulated experiences of an agent in a limited domain. Where previous work looked at a domain that has on the order of twenty distinct symbols, common-sense KBs often contain millions of entities and even more relations about them. Second, the graphs of KBs present representational difficulties as the agent must be able to encode the graphical structure, while prior work focused on propositional symbols. Finally, to the best of our knowledge, prior work has failed to generalize across different instances of a predicate, thereby negating the benefits of the semantics of a KB. Again, this poses challenges as to the representation of the KB and the value function. This is in contrast to the flat propositional representation in the prior work.

While RL has been used for reasoning (Taylor et al., 2007) and relation learning in KBs (Das et al., 2018), using KBs in RL have received relatively little attention. Prior work on improving RL with external knowledge have focused on providing features (Bougie & Ichise, 2018; Bougie et al., 2018) and value advice (Daswani et al., 2014). Other work have integrated RL, symbolic and relational learning, and natural language processing in the same system, with an emphasis on online knowledge acquisition (Mitchell et al., 2015). None of these systems directly address the question of using knowledge to solve tasks through reinforcement.

It should be noted that KB feature extraction for RL is different from two other AI tasks, question answering and relation learning, even though all three tasks might apply RL to a KB. A key difference of this work from question answering is that questions such as "Who is the artist of *The Wall*?" is never explicitly posed — or indeed, the agent may not know whether answering such that question is appropriate, as opposed to answering another question or not answering any questions at all. Rather, the agent is trying to maximize its reward, without regard to whether any particular question is answered correctly. Additionally, work in question answering tends to take a supervised learning approach, by using sequence-to-sequence translations of English questions to KB queries (Soru et al., 2017). Such training data is not generally available in an RL setting, and it's unclear that a similar translation from state sequences would be possible.

Similarly, while prior work have used RL for learning new relations (Xiong et al., 2017; Das et al., 2018), that approach still requires existing question-answer pairs (represented by the starting and ending entities), with the agent being rewarded for navigating to the correct answer. In contrast, no such pairs are provided to the agent in our work, and the agent is not rewarded for locating the answer in the KB. Instead, the reward depends entirely on the domain, independent of how the agent uses LTM.

5. System Architecture

The general approach we take in framing LTM retrieval as a sequential decision problem is to create an *internal problem space* where the agent has actions for querying and retrieval. One approach that directly mirrors how current agents are written is to define each cue as an action. This approach is necessarily domain-specific, however, as the valid cues must be defined based on the task. The obvious alternative is to consider all possible attributes in LTM, and make them available as building-blocks for cues. The size of these KBs makes this computationally intractable, however; DBpedia, for example, has over 1,300 distinct predicates and over 4M distinct literals. Presenting this many possible actions to the agent will make learning extremely slow.

Instead of learning to create the correct cue for retrieval, we reframe the problem to that of navigating LTM as an edge-label directed graph. By the assumptions laid out in Section 3, the environment percepts directly correspond to entities in LTM. The agent can then "move" along the incoming and outgoing edges of those entities, which correspond to querying and retrieving LTM respectively. Querying also allows the agent to "jump“ to more distant entities in LTM. The goal then is to "navigate" to the entity of interest, retrieve it into working memory, then use it for the selection of external actions. Unlike using arbitrary attributes to build a cue suggested above, the navigation framing suggests that the agent only has a limited number of actions available: only the immediate attributes of an entity in working memory can be used for querying, which greatly reduces the action space.

Our approach to facilitate this is to create an internal reinforcement learning problem $\langle S_{int}, A_{int}, T_{int}, R_{int}\rangle$, which is heavily inspired by on ACT-R's buffers and the PRIMS actions (Taatgen, 2013). The internal state S_{int} subsumes the external state S_{ext}, with the internal actions A_{int} being a strict superset of the external actions A_{ext}, with additional internal actions that do not affect the external state. The internal state must represent the results of queries and retrievals, and the internal actions must allow the agent to create queries from external and internal percepts. One of our key insight is that it is sufficient for the agent to manipulate attributes in an internal state, and that they map onto querying and retrieving from LTM. The result of retrievals and queries are themselves sets of attributes, which can then be used for additional retrievals and queries; more complex queries of multiple attributes can be constructed by adding one attribute at a time.

We describe each of these components separately in the subsections below.

5.1 Internal State

The functional purpose of the internal state is to represent the knowledge necessary for the agent to make decisions, and as such, it is equivalent to the working memory of the common model. Taking inspiration from ACT-R, the internal state is further divided into three buffers: the PERCEPTUAL buffer, the QUERY buffer, and the RESULT buffer. Each of these buffers contain a set of attributes, and they serve as interfaces to the environment and to LTM. The PERCEPTUAL buffer contains the external percepts, and its contents directly correspond to the external state. Since the PERCEPTUAL buffer is equivalent to the external state, the internal state is a strict superset of the external state. The QUERY buffer represents the query that the agent is executing in LTM, if any, while the RESULT buffer represents the result of that query or retrieval. Both of these buffers could remain empty if the agent is not executing a query or if there are no results from LTM.

Formally, an internal state is a set of tuples $S_{int} = \{\langle b, p, v \rangle, ...\}$, where $b \in B = \{$PERCEPTUAL, QUERY, RESULT$\}$ is one of the three buffers. Changes to the internal state therefore consist of adding and removing these tuples, which can be thought of as adding attributes to and removing attributes from any of the buffers. The details of the action space are described in the next section.

As an example, Table 1 shows part of the internal state of an agent tasked with determining the artist of *The Wall*. In this example, the only attribute in the QUERY buffer is ⟨`label`, `"The Wall"`⟩, which is the query issued to LTM. The attributes of retrieved entity — the album *The Wall* — has been added to the RESULT buffer, and is can be used by the agent to determine its next action. The mechanics of how LTM interacts with working memory is described in the next section; the external state and task in this example is described in Section 6.1.

Buffer	Predicate	Object
PERCEPTUAL	`label`	`"The Wall"`
	`location`	`"Shelf A"`
QUERY	`label`	`"The Wall"`
RESULT	`subject`	`The_Wall`
	`label`	`"The Wall"`
	`artist`	`Pink_Floyd`
	`releaseDate`	`"1979-11-30"`
	`...`	`...`

Table 1. An example of the internal state of the agent as it queries for the artist of *The Wall*.

5.2 Internal Actions and Transitions

In order for the internal state space to interact with LTM, the internal actions must be able to retrieve the attributes of specific entities from LTM, as well as be able to query LTM with cues of one or more attributes. While retrieval is an atomic action, the building of query cues is more complicated. Taking inspiration from the PRIMS set of actions (Taatgen, 2013), we instead define actions that copy and delete attributes from buffers, with the query itself occurring automatically. With this in mind, the internal problem space has five parameterized actions:

- The *copy* action copies an attribute from one buffer to another. The action is parameterized by the source and destination buffers, as well as the attribute to copy, and is instantiated as $\mathit{copy}(\textsc{buf}_{\textsc{src}}, \texttt{pred}, \texttt{obj}, \textsc{buf}_{\textsc{dst}})$. When this action is taken, a new triple $\langle \textsc{buf}_{\textsc{dst}}, \texttt{pred}, \texttt{obj} \rangle$ is added to working memory.
- The *delete* action deletes an attribute from a particular buffer. The action is parameterized by the buffer and the attribute to delete, and is instantiated as $\mathit{delete}(\textsc{buf}, \texttt{pred}, \texttt{obj})$. When this action is taken, a triple $\langle \textsc{buf}, \texttt{pred}, \texttt{obj} \rangle$ is removed to working memory.
- The *retrieve* action retrieves all the attributes of an LTM entity into the RESULT buffer. The action is parameterized by the buffer and attribute, with the constraint that the object of the attribute must be an entity and not a literal. The action is instantiated as $\mathit{retrieve}(\textsc{buf}, \texttt{pred}, \texttt{obj})$. When this action is taken, all triples in the RESULT buffer are removed, and all attributes of `obj` are added to the RESULT buffer.
- The *prev-result* action gets the previous result of a query. When this action is taken, the contents of the RESULT buffer is replaced with the attributes of the previous query result.
- The *next-result* action gets the next result of a query. When this action is taken, the contents of the RESULT buffer is replaced with the attributes of the next query result.

Note that while a *retrieve* action exists, there is no corresponding *query* action. Instead, query happens automatically when the query buffer is changed by the *copy* and *delete* actions, which would modify the cue. Together with the *prev-result* and *next-result* actions, this set of actions allows the agent to query and retrieve from LTM, and to iterate over the results.

The various actions on the QUERY and RESULT buffers also affect each other. For example, if no entity in LTM matches the query cue, the RESULT buffer is cleared to indicate that no results were found; the same occurs if the agent deletes the last attribute in (that is, empties) the QUERY buffer. Similarly, the QUERY buffer is cleared when a *retrieve* action is taken, to avoid the situation of having cues that do not match the attributes in the RETRIEVAL buffer. These automatic changes to the buffers reduce the internal state space and thereby speed up learning.

As an example, consider how an agent would determine the name of the artist of *The Wall*. The PERCEPTUAL buffer contains $\{\langle \texttt{label}, \texttt{"The Wall"} \rangle\}$ (as illustrated in Table 1, and LTM contains a portion of DBpedia as show in Figure 1. Retrieving the name `"Pink Floyd"` would require two actions:

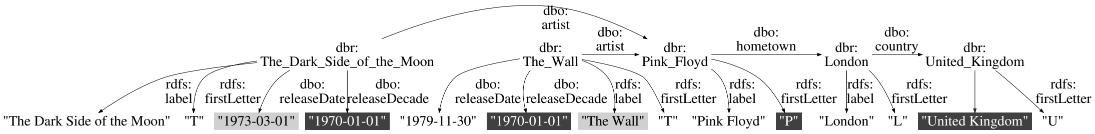

Figure 1. A portion DBpedia around Pink_Floyd.

1. *copy* the ⟨label, "The Wall"⟩ attribute from the PERCEPTUAL buffer to the QUERY buffer. Since the QUERY buffer has changed, the architecture would automatically query LTM for an entity that has the label "The Wall", and would find the entity The_Wall . The RESULT buffer would then be populated with the attributes of The_Wall, including the attribute ⟨artist, Pink_Floyd⟩.

2. *retrieve* the entity Pink_Floyd, which is the object of the artist predicate in the RESULT buffer. This will clear the QUERY buffer, and also populate the RESULT buffer with the attributes of Pink_Floyd, including the attribute ⟨label, "Pink Floyd"⟩.

With the string "Pink Floyd" now in working memory, the agent can now use it for action selection.

We make several optimizations to reduce the size of the internal action space. First, the availability of the *copy*, *delete*, *retrieve* actions depends on the contents of the internal state. For example, no *retrieve* actions are possible if none of the buffers contain an entity whose attributes could be retrieved, as would be the case if all attributes were literals. Similarly, no *prev-result* or *next-result* actions are possible if no query has been constructed, or if there is no next result. Since these actions are parameterized by entities that ultimately come from either perception or from LTM, the set of possible concrete actions will grow monotonically as the agent encounters more entities from the environment and from the KB, even while there may only be a few available actions for any particular internal state. The space of internal actions is therefore a function of the size of the KB.

We further limit the valid parameters for these actions to only those that are meaningful. Since the PERCEPTUAL and RESULT buffers represent the external environment and the KB respectively, directly modifying these buffers offer no benefits, and so they are invalid as targets for the *copy* and *delete* actions. Similarly, the contents of the QUERY buffer were themselves copied from another buffer, and therefore cannot serve as the source of a *copy* action. A final limit is on when internal actions are available at all, which we discuss in Section 5.5 below.

5.3 Internal Rewards

Since the reward ultimately comes from the external task, the reward for internal actions is constrained by the domain. We make two assumptions about the internal reward function. First, we assume that the internal reward is negative, $r_{int} < 0$, since a positive reward could lead to undesirable loops Second, we assume that the internal reward is smaller (closer to zero) than the external reward, which matches the intuition that accessing LTM is much less expensive than acting in the environment.

For internal actions to be learnable in the standard discounted-reward RL setting, the reward structure should discourage the agent from randomly trying external actions until one can be exploited. This implies that, at any internal state, the internal reward plus the discounted external reward must be greater than the external reward alone, such that internal actions are preferred. Formally, the following equations with discount rate γ (with $0 < \gamma < 1$) and internal and external rewards r_{int} and r_{ext} must hold:

$$\begin{aligned} r_{int} + \gamma * E[r_{ext}] &> E[r_{ext}] \\ 0 > r_{int} &> (1-\gamma) * E[r_{ext}] \end{aligned}$$

This inequality implies that the external reward must be negative, with the internal reward being a smaller negative that is modulated by the discount rate. We note that a constant negative external reward, until the end of an episode, is commonly used to encourage reaching the goal in as few actions as possible. We therefore do not consider this constraint to be overly burdensome.

5.4 Value Function Approximation

Although the reinforcement learning problem defined by the states, actions, and rewards thus far is sufficient to propagate external rewards through long-term memory retrievals, it remains computationally intractable. The size of the internal state space is exponential in the size of the KB, since each buffer could contain each unique attribute. The parameterized internal actions further increases the complexity, making learning difficult for an agent that naively stores a Q-value with each state-action pair.

Complexity considerations aside, treating each state as distinct also negates the primary advantage of using KBs: that of the semantics encoded within the graph structure. The normalized predicates in a KB such as DBpedia means that entities with the same predicate should have the same relation, and that the actions appropriate for one entity should generalize to another. In order to capture this generalization, a value function approximation is necessary.

Our approach is to use a linear value function approximation, which was chosen for its ability to include new features incrementally, by assuming that all unseen features have a weight of 0. Formally, the value of each state-action pair is approximated separately, with no shared weights, by the equation:

$$Q(s,a) = \sum_i w_{i,a} \phi_i(s)$$

The features $\phi_i(s)$ of a given state are all the tuples of buffer and predicate; all tuples of buffer, predicate, and object; and a bias term. That is, a state $\{\langle b_1, p_1, v_1 \rangle, ..., \langle b_n, p_n, v_n \rangle\}$ would have $2n+1$ features $\{\langle b_1, p_1 \rangle, \langle b_1, p_1, v_1 \rangle, \langle b_2, p_2 \rangle, ..., 1\}$, plus a bias term. To appropriately assign credit, the weight of each feature is normalized by dividing by the number of features. The weights are then updated by the standard Bellman update, parameterized by the learning rate α (with $0 <$

$\alpha < 1$):

$$\begin{aligned} \delta &= r + \gamma \max_{a_t} Q(s_t, a_t) - Q(s_{t-1}, a_{t-1}) \\ w_{i,a} &\leftarrow w_{i,a} + \alpha \delta \phi_i(s_{t-1}) \end{aligned}$$

Note that since the features are the attributes in working memory, which could come from retrievals from LTM, the number of features ultimately scales with the size of the KB. Unlike a tabular value function, however, this scaling is merely quadratic and not exponential. Still, this presents a not insignificant amount of memory, so the next section presents additional considerations to further reduce the state space.

5.5 Other Design Considerations

A final design decision is to prevent the agent from endlessly changing its internal state while ignoring the external environment. We do so by setting a limit on the number of consecutive internal actions that the agent could take, as well as a threshold on the cumulative negative reward of an episode, below which the episode will end. Forcing the agent to take high-penalty external actions (and bringing it closer to the threshold) effectively enforces a depth limit on how much of the KB the agent could explore. To be clear, the depth limit is *not* the number of consecutive internal action, but the number of internal actions the agent can take per external action, given the negative reward threshold. In theory, the depth limit is calculated by $\left\lfloor \frac{lt}{lr_{int} + r_{ext}} \right\rfloor$, where l is the internal action limit and t is the threshold. In practice, agents rarely reach this limit, as they are likely to select an external action before reaching the maximum consecutive internal action limit. As long as the internal action limit is set such that the desired entity is reachable, the effective depth limit only affects learning speed and memory use, but not the optimal policy. A secondary consequence of this limit is that it bounds the number of unique attributes that is retrieved from LTM, thus reducing the memory needed for the value function approximation.

We note that the internal RL problem as described above is compatible with both ACT-R and Soar. The system of buffers was inspired by ACT-R, and while Soar's semantic memory allows for graph-structured cues, it could also emulate the flat structure described here. Reinforcement learning in both ACT-R and Soar occurs by assigning values to production rules, with the conditions of those rules being the features. The main difficulty of implementing this system is generating rules as new features are added, when the agent encounters or retrieves new knowledge. While this is not currently possible within ACT-R and Soar, we suspect this is not due to conflicts with core architectural assumptions, but because reinforcement learning has not been attempted at this scale. Thus while this system is not implementable within ACT-R and Soar as they currently stand, we believe it still satisfies both of the desiderata listed in Section 3.

As a final note in comparing this system to ACT-R and Soar, many cognitive architectures use metadata such as base-level activation to rank results from LTM. While these memory mechanisms may be generally useful, they make learning to use memory harder, as they introduce an non-Markov element. If activation is used, querying with the same cue with the same working memory state could lead to different results at different times, as different activation values in LTM could change which entity is retrieved. For this reason, in this paper the knowledge to return is chosen deterministically, based only on the cues. Where multiple matching symbols exist, the agent could use the *prev-result* and *next-result* actions to iterate through possible results instead.

6. Experimental validation

To demonstrate and evaluate this framework, we show results from three experiments on a Record Store domain. The goal of these experiments are to show, first, that the system is able to converge on the optimal policy for different queries; second, that the learned policy can be transfered to new percepts; and finally, that the agent is able to adapt to changes in the environment that require different knowledge to be retrieved. Note that these experiments do not correspond to any cognitive task, by only demonstrate the ability of the RL system to learn to query and retrieve from LTM. Due to the above mentioned difficulty of dynamically creating features in ACT-R and Soar, the experiments described in this section were done in Python. The source code is available at `https://github.com/justinnhli/2020acs`.

6.1 The Record Store Domain

As in the running example throughout this paper, this domain presents the agent with some information about an album, and tasks the agent with finding it on the shelves of a record store. Importantly, the information given to the agent does not correspond to the organization of the store, so while the agent may be asked with picking up *The Wall*, the shelves may instead be organized by the decade of album release. Instead, the agent must access LTM for additional information about the album, and use the results to determine which shelf to go to. Since the focus of this paper is on the interface between the agent and LTM, and not on advancing fundamental RL algorithms, we deliberately chose a fully observable and deterministic domain. Observability here refers to environment, which corresponds to our definition of S_{ext}, and not the full contents of the KB,

In every episode, the agent is presented with information about one album. The external state in this domain contains only the album information (e.g., the title) and the agent's location in the store. The agent has access to the complete contents of DBpedia in this domain, with $\sim$24.1M triples. We used a version of DBpedia where all predicates have been mapped to the DBpedia ontology, meaning that the ~1,300 predicates are reused throughout the KB. Additionally, to reflect how record stores often arrange albums chronologically by decade and alphabetically by artist, we augment DBpedia with two pieces of information. All entities with the predicate `releaseDate` also get a `releaseDecade` predicate, with an appropriately-modified literal. Similarly, all entities with the predicate `label` are also augmented with a `firstInitial` predicate.

Each external action in this domain (moving to a shelf) results in a reward of -10; the episode ends when the agent reaches the shelf that contains the desired album. The number of external actions available depends on the metadata on which the shelves are organized (e.g., the first letter of the name of the artist), with one action corresponding to each category. Each internal action results in a reward of -0.1, which satisfies the constraints described in Section 5.3.

6.2 Convergence Experiment

This experiment serves as a basic validation of our framework, to show that it is possible for agents to learn to retrieve knowledge from a large KB. Three variants of the Record Store domain are used in this experiment, which differ by the "organization" of the shelves:

1. Given an album title, find the shelf corresponding to the decade of the album's release. This only requires one query using the album title, and is the same as the task in the first experiment.
2. Given an album title, find the shelf corresponding to the first letter of the artist. This requires one query using the album title, followed by a retrieval on the artist.
3. Given an album title, find the shelf corresponding to the country of origin of the artist. This requires one query using the album title, followed by three successive retrievals on the artist, their hometown, and finally the country in which that town is located.

Figure 1 illustrates how these sequences of queries and retrievals correspond to traversing further along the semantic network. For this experiment, the agent is tasked with finding 100 albums, with a different album every episode, and allowed to learn until its policies converge. All agents use a learning rate of 0.1 and a discount rate of 0.9.

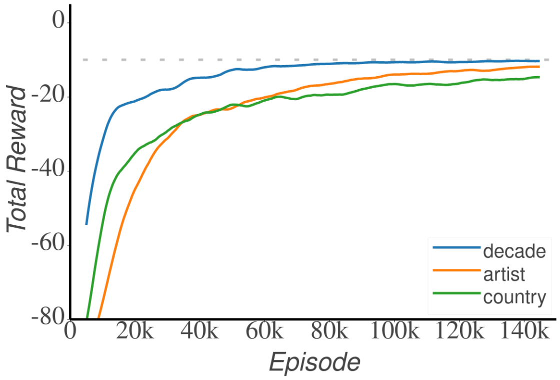

Figure 2. Performance in the Record Store domain with different retrieval sequences.

The overall results are show in Figure 2, with the x-axis showing the number of episodes into training, and the y-axis showing the total reward for that episode. In all three cases, the agent learns the optimal policy of first retrieving the relevant knowledge from LTM, before moving to the appropriate shelf. All three agents are able to learn the optimal policy, which suggests the agents have learned to query LTM and use the results for further retrievals (or to decide on the external action). This optimal policy is confirmed by examining the learned weights, where the weights for the external actions are dominated by the appropriate feature of the albums. As expected, the larger the graph distance between the external percept and the desired entities, the longer it takes for agent learning to converge, as a longer sequence of actions is required; this can be seen in how the "decade" agent learns faster than the others. Although retrieving the country of the artist requires more actions, however, learning those actions eventually converges more quickly than learning to retrieve the artist. Our hypothesis for this behavior is that there are fewer countries than artists, and so less exploration is required to identify the correct external action. Regardless, these results suggest that the architecture can be applied to tasks and KBs that require different and more complex features.

6.3 Transfer Experiment

This experiment serves to demonstrate transfer of the policy to new percepts, and therefore that the linear value approximation is valid. We ask three questions in this experiment. First, how does the learning speed compare to a naive agent that simply uses trial and error to learn the correct action? That is, how does our framework compare to a simple tabular agent for the external RL problem? Second, can the internal action limit effectively serve to control the exploration of the agent, so large portions of LTM can be ignored? And finally, is the agent capable of transferring action selection knowledge to previously-unseen entities?

In this experiment, the agent is given the title of an album, and must navigate to the shelf for the decade in which the album was released — information it can obtain via a single query. To explore the question of transfer, we use two disjoint sets of 100 albums, out of about 15,000 albums that exist in DBpedia. For the first 150,000 episodes, only the first set of albums will be presented, allowing time for the agent to learn the optimal policy. After 150,000 episodes, only the second set of albums will be used instead. The switching point of 150,000 episodes was determined from a pilot run to be sufficient for agent learning to converge. Note that although the album entities and titles will be different, the sequence of actions required to retrieve information from LTM remains the same.

We look at three different agents: a naive tabular Q-learner, and two agents that use our framework with access to the complete contents of DBpedia through LTM, which we label the KB agents. These two agents differ only by their limit on the number of consecutive internal actions; we label these agents KB-4 (with an internal action limit of 4) and KB-1 (with a limit of 1). All agents use a learning rate of 0.1 and a discount rate of 0.9.

The overall results are shown in Figure 3. The solid line shows the performance of the tabular Q-learner, and the dashed and dotted lines show the performance of the KB-4 and KB-1 agents respectively. The tabular agent converges more quickly to the optimal policy, despite the generalizations afforded by the value function approximation. This is not surprising, given that the internal environment is a superset of the external environment, and thus has a larger state and action space. Even with the amount of knowledge available in DBpedia, however, the KB agents do not require significantly more episodes to learn the optimal policy. This suggests that the features for the value function approximation allow for efficient learning.

Examining the results more closely, the KB-4 agent is slower to converge than the KB-1 agent. In fact, although the KB-4 agent learns over time, its learning has not reached convergence before episode 150,000 when new albums are introduced. This confirms our hypothesis that the internal action limit is an effective control for agent learning. With a lower limit, the KB-1 agent has fewer opportunities to retrieve from or query LTM, which in turn reduces the number of distinct entities

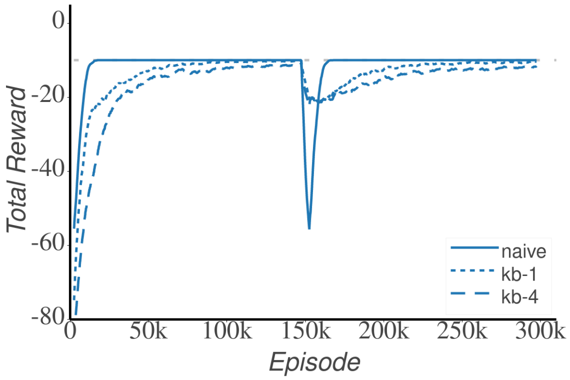

Figure 3. Performance in the Record Store domain by different agents, where the store organizes albums by decade. A new set of albums are presented at episode 150,000.

that it encounters. In effect, this reduces the number of weights the agent has to learn, since each new predicate and attribute requires a new linear weight. This also has the effect of reducing the amount of memory required by the agent.

Finally, the performance of the agents are clearly affected at episode 150,000, when a new set of albums are presented. At this point, the performance of the tabular agent drops to initial levels, which is expected, since the new entities do not match any entries in its value function table, and the agent must learn from scratch. In contrast, we only see a moderate drop in performance for the two KB agents, with the slight decrease occurring as the agent learns the weights for the new entities. We attribute this behavior to the value function approximation. By learning that the predicate, but not the predicate-object pair, is informative of which action to take, the agent is able to transfer its knowledge of how to use LTM to new entities it has not encountered before.

In summary, this experiment shows that our representation is sufficient for agents to learn an optimal policy for using LTM, that it does not slow down learning dramatically, that we can control learning with an internal action limit parameter, and that it allows agents to transfer its LTM strategy to new environmental percepts.

6.4 Adaptation Experiment

This final experiment demonstrates the advantage of an agent capable of learning to retrieve, as opposed to having hard-coded retrieval rules. In this experiment, the Record Store reorganizes its shelves in the middle of the agent's run, from being organized by artist, to be organized by the release decade of the album. The agent initially learns to retrieve the artist and use that knowledge to decide its action; at episode 150,000, however, the agent must learn to retrieve the release date instead. As before, the agent uses a learning rate of 0.1 and a discount rate of 0.9.

The results of this experiment is show in Figure 4. As expected, agent performance drops drastically when the environment changes since, unlike the previous experiment, the agent cannot simply apply its learned policy to new percepts, but must learn a new policy. Reinforcement learning allows this to occur, however, and the agent is eventually able to converge on the optimal policy for

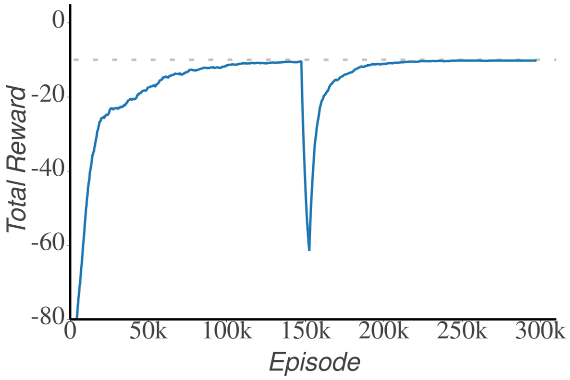

Figure 4. Performance in the Record Store domain when the shelves are initially organized by artist, to be organized by decade at episode 150,000.

the new environment. This demonstrates the necessity of applying procedural learning to memory retrieval, removing the need for programmer intervention in new environments.

7. General Discussion

In this paper, we presented a framework that can learn to retrieve from LTM from external task rewards. We provided evidence that our approach of using internal actions with a linear value function approximation allows the agent to converge to the optimal policy for different patterns of accessing LTM. The agent is able to transfer its the memory access strategy to new percepts, and also to adapt when the environment requires different knowledge from LTM. Other agent parameters effectively limit the depth of exploration of the LTM, allowing our approach to work even with KBs with millions of triples. This work is therefore a step towards more tightly integrating the declarative memory and reinforcement learning components defined in the common model of cognition.

This work opens the door for several areas of future work. First, the capabilities of the current approach is not isomorphic to the capabilities of ACT-R or Soar, as those architectures have additional features for LTM access, such as prohibiting a particular attribute matching. We also have not tested our approach with an additional buffer for the intermediate storage of retrieved entities, which would be necessary for any query patterns that require graph matching; the imaginal buffer in ACT-R serves exactly this purpose. ACT-R and Soar can also perform additional reasoning with the results from LTM, which was not the focus of this paper. While we do not have empirical evidence of performance, we do not seen any major obstacles towards combining our approach with reasoning and additional LTM mechanisms.

Second, one of the motivations for this work was the possibility for agents to learn higher-order memory strategies such as rehearsal. Many of these strategies interact with subsymbolic metadata, such as base-level activation, which we've set aside in this work. More generally, human memory retrieval is highly strategic, with multiple processes for problem solving, elaboration, and verification (Burgess & Shallice, 1996). Similarly, metamemory phenomena such as feeling of knowing may be explanable as additional features for deciding whether to query LTM. By allowing task rewards to drive the pattern of LTM access, future work can explore tasks under which these memory strategies and mechanisms are beneficial.

References

Bougie, N., Cheng, L. K., & Ichise, R. (2018). Combining deep reinforcement learning with prior knowledge and reasoning. *Applied Computing Review*, *18*, 33–45.

Bougie, N., & Ichise, R. (2018). Deep reinforcement learning boosted by external knowledge. *Proceedings of the 33rd Annual ACM Symposium on Applied Computing* (pp. 331–338). ACM.

Burgess, P. W., & Shallice, T. (1996). Confabulation and the control of recollection. *Memory*, *4*, 359–412.

Das, R., Dhuliawala, S., Zaheer, M., Vilnis, L., Durugkar, I., Krishnamurthy, A., Smola, A., & McCallum, A. (2018). Go for a walk and arrive at the answer: Reasoning over paths in knowledge bases using reinforcement learning. *Proceedings of the 2018 International Conference on Learning Representations*.

Daswani, M., Sunehag, P., & Hutter, M. (2014). Reinforcement learning with value advice. *Proceedings of the 6th Asian Conference on Machine Learning* (pp. 299–314).

Douglass, S. A., Ball, J., & Rodgers, S. (2009). Large declarative memories in ACT-R. *Proceedings of the 9th International Conference on Cognitive Modeling (ICCM)*.

Dulac-Arnold, G., et al. (2015). Reinforcement learning in large discrete action spaces. From `https://arxiv.org/abs/1512.07679`.

Gorski, N. A., & Laird, J. E. (2011). Learning to use episodic memory. *Cognitive Systems Research*, *12*, 144–153.

Laird, J., & Mohan, S. (2018). Learning fast and slow: Levels of learning in general autonomous intelligent agents. *Proceedings of the 32nd AAAI Conference on Artificial Intelligence (AAAI)*.

Laird, J. E., Lebiere, C., & Rosenbloom, P. S. (2017). A standard model of the mind: Toward a common computational framework across artificial intelligence, cognitive science, neuroscience, and robotics. *AI Magazine*, *38*, 13–26.

Li, J., & Kohanyi, E. (2017). Towards modeling false memory with computational knowledge bases. *Topics in Cognitive Science (TopiCS)*, *9*, 102–116.

Mitchell, T., et al. (2015). Never-ending learning. *Proceedings of the 29th AAAI Conference on Artificial Intelligence (AAAI)*.

Nason, S., & Laird, J. E. (2005). Soar-RL: Integrating reinforcement learning with Soar. *Cognitive Systems Research*, *6*, 51–59.

Salvucci, D. D. (2015). Endowing a cognitive architecture with world knowledge. *Proceedings of the 37th Annual Conference of the Cognitive Science Society (CogSci)*.

Soru, T., Marx, E., Moussallem, D., Publio, G., Valdestilhas, A., Esteves, D., & Neto, C. B. (2017). SPARQL as a foreign language. *Proceedings of the Posters and Demos Track of the 13th Inter-*

national Conference on Semantic Systems (SEMANTiCS).

Taatgen, N., Lebiere, C., & Anderson, J. (2005). Modeling paradigms in act-r. In R. Sun (Ed.), *Cognition and multi-agent interaction: From cognitive modeling to social simulation*, 29–52. Cambridge University Press.

Taatgen, N. A. (2013). The nature and transfer of cognitive skills. *Psychological Review*.

Taylor, M. E., Matuszek, C., Smith, P. R., & Witbrock, M. (2007). Guiding inference with policy search reinforcement learning. *Proceedings of the 20th International Florida Artificial Intelligence Research Society Conference (FLAIRS)*.

Xiong, W., Hoang, T., & Wang, W. Y. (2017). DeepPath: A reinforcement learning method for knowledge graph reasoning. *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing (EMNLP)* (pp. 564–573). Association for Computational Linguistics.